University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Modelling Contracts and Workflows for Verification and Enactment

Andrew D H Farrell
BSc (Hons 1), MSc (Dis), DIC

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, March 2008

Abstract

The work presented in this thesis concerns some aspects related to the *Modelling of Contracts and Workflows for Verification and Enactment.* Workflows help coordinate the enactment of business processes. Lacking in most contemporary approaches to workflow is a formal grounding to the semantics of workflow. A principal aim of this work is to address this shortcoming.

We provide formal characterisations of workflow using a number of formal tools, viz. Milner's CCS, Cleaveland et al's Prioritised CCS (which we abbreviate to PCCS) and the Situation Calculus (thanks mainly to Reiter), which is based on First-Order Logic. We define the `Liesbet` meta-model for workflow to provide a reference ontology for the task of formalisation. We have also implemented a framework for the verification and enactment of `Liesbet` workflow models. Regarding verification, we are particularly interested in the key property of soundness, which is concerned with an absence of locking and redundant tasks in a workflow model. Our framework is capable of verifying this property of workflow models, as well as arbitrary temporal constraints which are constraints whose satisfaction is determined over successive states of enactment of a model.

It has been widely noted that traditional approaches to workflow are too rigid and brittle to cope adequately with the typical operation of business processes. Thus, there is an evident need to support workflows that are able to be flexibly enacted and are better able to cope with exceptional behaviour. We also seek to address this need.

We make novel use of *Hierarchical Task Network* (`HTN`)-based planning techniques in order to provide a modelling, verification and enactment framework for flexible workflow. The framework uses a planner, called `Theodore`, that we have defined and implemented in this work. We define a similar notion of soundness which the `Theodore`-based framework is able to verify, along with arbitrary temporal constraints. We also support the modelling of collaborative workflow where participating agents decide collectively how a prescribed task or process should be realised, as well as the notion of "what may I do next" querying where an agent is able to reason over which actions they may do next.

Finally, we have been interested in investigating how concepts inherent in workflow might apply in the modelling of contracts. To achieve this, we have explicated a new perspective for workflow, namely an *institutional perspective*, and define the notion of *Institutional Workflow Modelling* (IWM). The essence of IWM lies (in part) in the identification that the structure of a workflow model necessarily entails the existence of *counts as* relations. These relations prescribe how the occurrence of certain actions, in the context of a particular workflow model, count as the occurrence of other actions. We argue that IWM should be considered as a foundational basis for contract modelling. We have also defined and implemented a framework for IWM-based contract modelling,

verification and enactment, which reuses tools from our framework for flexible workflow at its core. We make available similar mechanisms for verifying the notion of soundness and arbitrary temporal constraints for contract fragments, and for performing "what may I do next" querying.

Acknowledgements

To my wife, Natalie, for finding limitless reserves of patience in allowing me to do and complete this thesis. To my (as yet) two children, whose names emblazon my work: Liesbet (Beth) and Theodore (Theo) – you bring me so much joy and happiness. I love you all very much.

To my loving parents, for your unwavering belief in me. I thank you from my heart. To my siblings, for making me determined and resolute. To Martin and Mariette, for your continued support and encouragment.

To my supervisor Marek. Amongst your peers, you stand out as a person of great integrity and ability. You bring humour and humility to an academic world that so often lacks both of these. I am immensely honoured to have had the opportunity to work with you.

To my colleagues within BDIM at HP: Abdel, David, Athena, Maher, Mathias and Claudio, I will be forever grateful for the opportunities that you have afforded me, and I look forward to future collaborations also.

To my new colleagues within ESSL at HP, especially Peter, Patrick and John, I thank you for your patience in allowing me to finish working on this thesis, and look forward to a fruitful time within my new team.

March 2008

For Natalie, Beth, Theo and ...

Forever yours ...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis concerns some aspects related to the *Modelling of Contracts and Workflows for Verification and Enactment.* The term workflow (as will be elaborated) pertains to the automated embodiment of business processes. Lacking in most contemporary approaches to workflow is a formal grounding to the semantics of workflow. A principal aim of this work is to address this shortcoming.

It has been widely noted (as we describe below) that traditional approaches to workflow are too rigid and brittle to cope adequately with the typical operation of business processes. Thus, there is an evident need to support workflows that are able to be flexibly enacted and are better able to cope with exceptional behaviour. We also seek to address this need.

Finally, we have considered it of interest to see how our research on workflow might be reused in other contexts. We have long held an interest in the modelling of contracts, such as agreements between service providers and customers, which are in some contexts known as *Service Level Agreements* [68]. We are interested in investigating how concepts inherent in workflow, and thus how workflow modelling techniques, might apply in the modelling of contracts. Studying contract modelling has also been of interest in itself, irrespective of how we might reuse our other work.

In summary, the aims of the work described in this thesis are to address the issues of:

- Providing a formal grounding of workflow.
- A more flexible approach to workflow.
- How workflow concepts might apply in the modelling of contracts, and looking at the modelling of contracts generally.

In order to realise these aims, we have carried out a diverse array of investigative work. We will elaborate on what we have done in the course of this introduction, which has the following structure. Firstly, we provide an overview of the areas of interest in our work. Then, we outline the contributions of this thesis, followed by a description of the structure of the thesis. It is in these two sections that the reader should get a flavour of how we have proceeded to realise our aims. We conclude the chapter with a declaration of originality and a list of publications contributing to the work described herein.

1.1 Overview of Areas of Interest

In this section, we start by looking at what is workflow, and how we might provide a formal grounding to it. We then look at the need for flexible workflow modelling, followed by a brief synopsis of contract modelling.

1.1.1 Workflow: An Approach to the Automated Modelling of Business Processes

Workflows are primarily concerned with the co-ordination of tasks comprising *business processes.* The operation of companies and organisations is characterised by a number of business processes that need to be carried out in a way that is strategically aligned with the objectives of the business. The Workflow Management Coalition (WfMC) defines a *business process* to be "a set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships" [136].

Business Process Management (BPM) is a term that has been used to refer to "aligning business processes with an organisation's strategic goals, designing and implementing process architectures, establishing process measurement systems that align with organisational goals, and educating and organising business managers so that they will manage processes effectively" [2]. In [74], BPM is described as "process technology enhanced with process management capabilities, implemented in a way that is appealing to business users". Although BPM tends to be a term that is differently applied, the consensus behind its use seems to be the notion of a *managed automation of business processes*, where the management generally is meant to align the enactment of a process to the objectives of the (business) enterprise.

Workflow technologies [62, 50] have become a key enabling technology for the implementation of BPM. They handle the co-ordination of activities in a business process by initiating their execution through assigning agents at appropriate times to carry out the work. The term *workflow* is defined by the WfMC to be: "[t]he automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules" [136]. Note that the term *workflow model* refers specifically to the machine representation of a business process.

The language used to express a workflow model is commonly referred to as a workflow language. In the context of formalising such languages, the term *workflow meta-model*, or *workflow ontology*, is commonly used to refer to the collection of constructs used to represent a workflow model. Finally, the term *workflow management system (WfMS)* (a.k.a. process engine) is used to refer to the engine responsible for executing workflow models.

Referring to Figure 1.1, a distinction can be made between different sorts of workflows based on their repetition, that is, how frequently a particular workflow is enacted by an enterprise [70]. Highly-repetitive workflows are called *production workflows*, after the metaphor of a production line. Less repetitive workflows are often called *collaborative workflows*, capturing the notion that they come about through *ad hoc* collaborations. Both sorts of workflows may be of high value to a business. Notably, workflow technologies have focussed on providing automation for production workflow. In the literature, this sort of workflow is also known as *traditional workflow.* It is the processes to which such workflows correspond that implement the core business of the company;

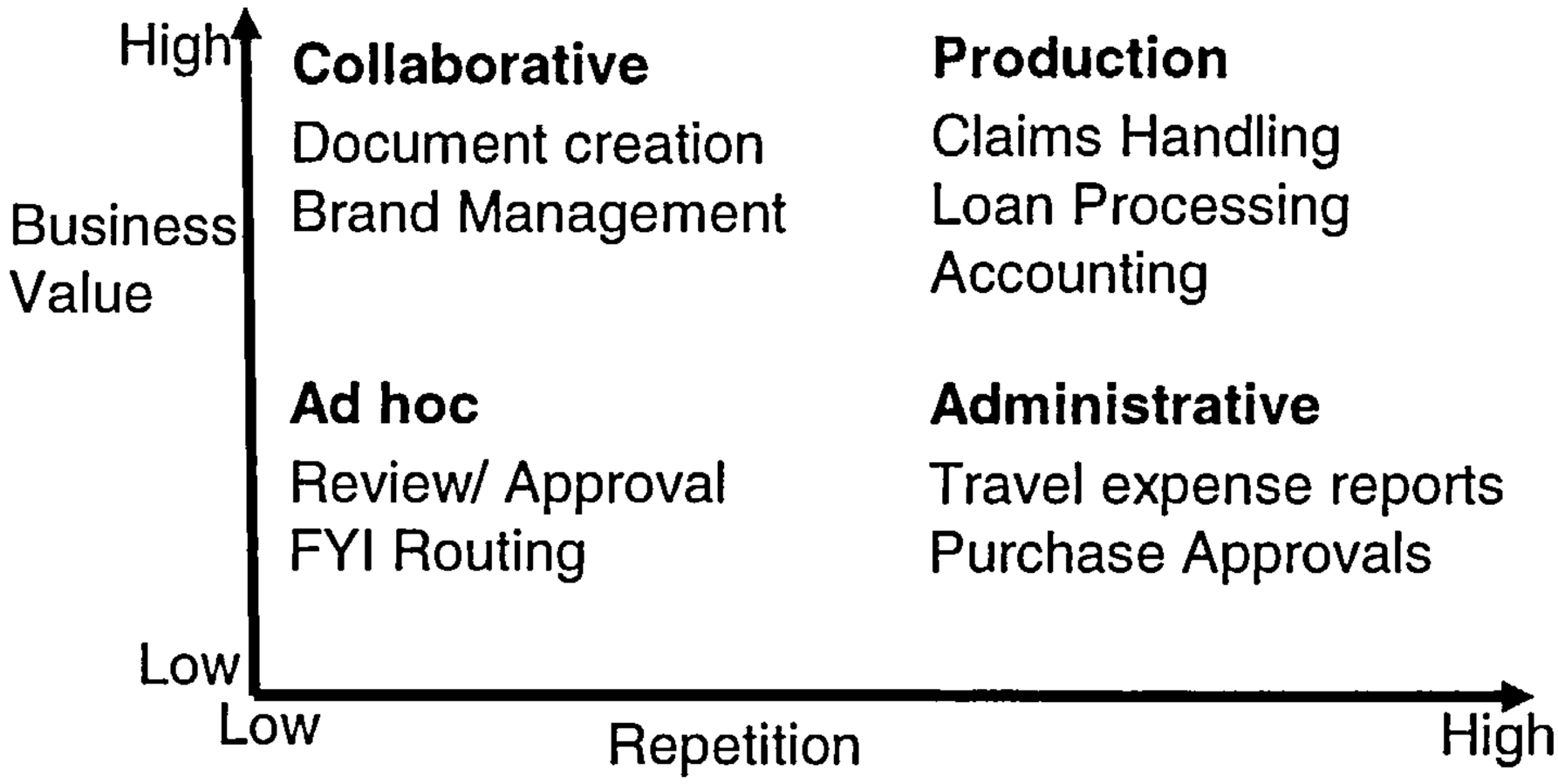


Figure 1.1: Taxonomy for Workflow [70].

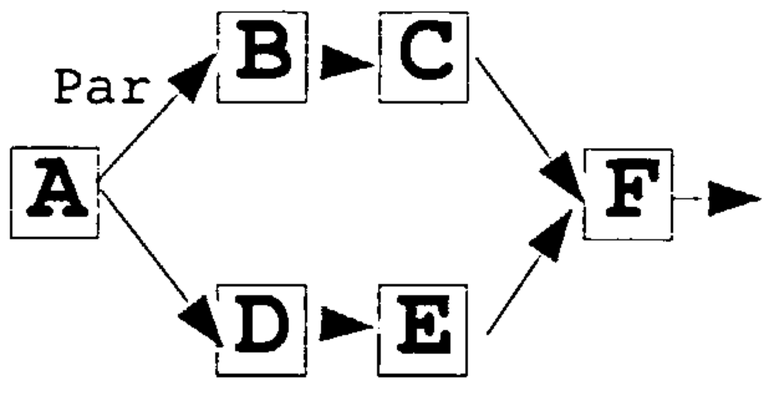


Figure 1.2: An Example Workflow Model.

and it is their efficient execution that provides a company with its competitive edge [70].

It is often convenient to divide the description of production workflows into several different perspectives. There have been several suggested taxonomies for workflow perspectives, e.g., [62, 120]. We follow the one presented in [120]. Here, Van der Aalst describes a number of different perspectives, but we shall concentrate on just two – the *control* and *data* perspectives. Automation, through workflow technologies, has focussed primarily on these two perspectives. The control perspective is arguably the most important in the definition of a workflow model. It is concerned with the definition of a (partial) ordering by which activities should be executed (by a WfMS).

Figure 1.2 is an example of a workflow model defined at the control perspective. In the example, activity A is executed first. Once the execution of A has completed, the execution of two sequences is initiated, in parallel. The first sequence consists of activities B and C, the second of D and E. When initiation of the sequences occurs, the execution of B and D, respectively, is initiated (at the same time). It may occur that B and D do not complete together, but, as soon as either does complete, the next activity in its containing sequence is executed, i.e. C (following B) or E (following D). Once either C or E completes, its containing sequence also immediately completes, and, once both sequences have completed, not necessarily at the same time, execution of their containing parallel artefact immediately completes. Then, once this occurs, the execution of activity F is initiated, and, once that completes, execution of the workflow model completes.

The data perspective is concerned with the management of data during the enactment of the

Name	Organisation	Type
Business Process Modelling Notation (BPMN) [24]	BMI	Notation
UML Activity Diagram [46]	OMG	Notation
WS Business Process Execution Language (WS-BPEL) [87]	OASIS	Orchestration
XML Process Definition (XPDL) Language [135]	WfMC	Orchestration
XLANG [103]	Microsoft	Orchestration
Web Services Flow Language (WSFL) [69]	IBM	Orchestration
Web Services Choreography Interface Language (WSCI) [129]	W3C	Choreography
Web Services Choreography Description Language (WS-CDL) [137]	W3C	Choreography

Table 1.1: Some Principal Business Process Management Languages. Adapted from [56].

workflow model. We can define two types of data: *control* and *application* (or *production*) data. Control data is used to evaluate branching conditions, or, more generally, is used by the WfMS to determine how execution should proceed [13]. It is usually declared, or allocated, within a workflow model, and its scope of existence is the workflow model. It is simply meant to control the enactment of the model. Application data, on the other hand, is data that primarily exists outside of the model, but is imported and used by the model. For example, in the case of workflow models, such data may be documents, forms and tables [120]; or, in the case of service composition (see below), such data would be that sent and received in messages that are exchanged between services [13].

Production workflows may be encapsulated as *Web Services* in order to make their functionality readily available to other business logic within the same company, or within another company. Web services are a key enabler of the *Service-Oriented Architecture* (SOA) [86]. They can be invoked by applications or other web services using standardised XML-based Internet protocols, such as HTTP, SOAP, WSDL and UDDI [32]. The SOA is a proposed means of improving the agility and competitiveness of enterprises – business logic may be packaged as components, with standard interfaces, and dynamically composed to provide new services which add value to a business' portfolio. *Web services* are proposed as "the cornerstone for architecting and implementing business processes and collaborations within and across organisational boundaries" [86]. W3C defines a web service as "a software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artefacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols" [139].

Web Services Composition (WSC) is a principal aspect of the Web Services framework, where *composite (web) services* may be created by inter-connecting deployed web services from potentially many different service providers. WS-BPEL [87] is a popular standardised language for (web) service composition. A composition is the equivalent of a workflow model in the context of SOA. Just as for a workflow model, a composition is concerned with the co-ordination of activities and the data that passes between them, except that these activities are now packaged as web services. As compositions and workflows share many similarities, they are typically discussed together when talking about business process modelling.

An important distinction should be made between *Web Services Orchestration* (WS-ORCH) and *Web Services Choreography* (WS-CHOR). WS-ORCH is concerned with defining composite web services from web services that may belong to the same enterprise, or many enterprises. WS-CHOR is concerned with defining collaborations between web services [86, 133]. WS-ORCHs are typically viewed as under-writing WS-CHORs, or facilitating the driving of WS-CHOR-style interactions across enterprise boundaries. That is, the WS-ORCH is the private, end-point, or local, perspective of the operation of a business process, which will need to support the public, global view (WS-CHOR) of the collaboration between the business process and others. In this work, we are solely concerned with Web Services Orchestration. An example of a WS-ORCH language is WS-BPEL [87]. An example of a WS-CHOR language is WS-CDL (Web Services Choreography Description Language) [137].

In Table 1.1, we present a summary of some of the principal languages for BPM, pertaining to both *workflow* and *service composition specification*.

1.1.2 Providing a Formal Grounding to Workflow

In our work, we are interested in providing a formal grounding to workflow. In order to do so, it is incumbent to attempt to understand the true nature of workflow, and its representational requirements, so that we may have a point of reference for any formal characterisation. In doing this, it is worthwhile being minded of an important distinction between various abstractions or *views* of workflow that may be used by different people or pieces of computer logic.

- Presentation view: Business managers, executives, customers.
- Authoring view: Business analysts and process authors – i.e. those responsible for capturing/authoring workflows. This view would have an associated ontology whose constructs would be considered to be intuitive to a process author. The ontology would most likely be graphical in nature. For instance, Figure 1.2 might constitute a workflow model defined using such an ontology.
- Information view: Serialisation (or file) format and reference point for the computational view (see below), in that it fixes the sufficient and (as much as possible) necessary representational requirements of the modelling approach. Note that in some modelling approaches, it may be appropriate to divide this view into two, along these two themes. However, we have not needed to make such a distinction in our modelling approach.

  Note that the information view will typically be closely aligned to the authoring view (for ease of mapping between the two views) and will, as a consequence, make similar ontological commitments to that of the authoring view, albeit they will likely be represented by distinct ontologies.
- Computational view: Process engine, or the process engine implementer. `(a.(b.c.s|d.e.s|s̄.s̄.f))\{s}` might be a computational view of a particular workflow model, such as the one illustrated in Figure 1.2, where the ontology used would be CCS/π-calculus-like[1].

[1] It is not important, at this stage, for the reader to necessarily understand the presented CCS process term. Just to know that it is a possible representation of the model presented graphically is sufficient.

Primarily, the computational view will define an ontology to provide a semantic characterisation of the ontology defined at the information view. That is, the computational view fixes the precise meaning of workflow models, by providing a semantic characterisation of information view models. The definition of the computational view will be facilitated by the use of some formal tool, such as Petri nets or CCS/π-calculus.

A computational view workflow model may be directly executable by a workflow engine; that is, the engine may directly understand and execute Petri nets or CCS/π-calculus. In this case, a translator will map models serialised using the information view format to the computational view. Or, as the computational view fixes the meaning of models, an engineer may implement a process engine capable of understanding models written at the information view, and ensure their enactment according to computational view semantics. In either case, it is imperative that the computational view provides an intuitive and tidy characterisation of the information view ontology.

The authoring, information and computational views of a workflow model may be represented using the same ontology or using distinct ontologies. An example of the former is the use of Petri nets for workflow modelling where the same formal tool is used for all views. In the case where there are distinct ontologies for different workflow views, it is typical for the information and authoring views not to be defined formally, i.e. using some mathematical formalism. Rather, they will usually be abstracting syntaxes, or ontologies, for the computational view.

The existence of the computational view is important for precision and robustness in the definition of workflow models, and for verifying properties of workflow models, such as workflow soundness (see below). It is a notable characteristic of most workflow languages that they lack a robust semantics [121], which would be provided by the computational view, and a notable characteristic of most commercial workflow products that they have no support for verification of workflow models.

Indeed, recently, the lack of formal models for such languages has become a contentious issue in workflow. Correspondingly, there has been a lot of confusion regarding the role of formal methods per se. In fact, as Van der Aalst explains: "[i]t seems that formal methods are used to advertise languages rather than to improve their quality and applicability" [122].

The lack of any solid notion of a formal semantics – even for the control perspective – has not been helped by the main contributor to the standardisation process for workflow: the Workflow Management Coalition (WfMC). As [124] notes: "[t]he lack of a formal semantics [for the workflow constructs defined by the WfMC] has resulted in different interpretations by vendors of even basic control flow constructs, [and] definitions in natural language such as provided by WfMC are not precise enough."

The importance of a well-defined formal model for a workflow language is clear. It is only with such a model that we can go on to prove desirable properties about workflow models that have been specified using the associated workflow language.

Workflow soundness [120] is an essential property of the control perspective, corresponding to an absence of certain deficiencies which would compromise the behavioural integrity of a workflow model. These deficiencies are locking and redundant tasks, which can quickly creep into workflow models as they are being defined. As [120] says: "errors [in the definition of workflow models] may lead to angry customers, back-log, damage claims, and loss of goodwill". It is important, therefore,

that soundness of workflow models is verified prior to model deployment.

In this thesis, we are concerned with capturing the computational view of workflow as an end in itself, as well as for facilitating the verification of workflow properties. For these purposes it is also appropriate to define an information view ontology, to serve as an abstract syntax which can, on the one hand, act as a serialisation syntax, and on the other hand, act as a reference point for the computational view ontology to target. Its primary purpose, however, is to fix concisely what we are concerned with representing. As a result, it may closely resemble an authoring view ontology – which we do not define in this thesis.

We have been interested in investigating the use of existing formal tools and languages for the characterisation of the computational view of workflow, as we describe in the Section 1.2. There are a number of advantages to such an approach, including:

- The availability of accompanying tools provides a means of quickly validating characterisations.
- They provide a quick means of specifying characterisations, in that one is freed from thinking about defining an appropriate representational device, and may concentrate instead on thinking about the appropriate semantics for (in this case) workflow.
- The characterisation of workflow can be used as a vehicle for understanding the representational weaknesses of these formal tools and languages, in the sense of how efficacious they are in characterising workflow in a succinct and clear way.

We define *mapping functions*, which translate workflow models specified at the information view into models specified at the computational view. *It is the definition of the mapping function, together with the semantics of the pertaining computational view ontology, which are constrained by the mapping function, that define a semantic characterisation of the information view meta-model.*

Finally, the ontological commitments that any approach to business process modelling makes should be sourced from an understanding of the behavioural nature of business processes. Members of the BPM community have previously set about characterising the behavioural nature of business processes, in the form of the YAWL (*Yet Another Workflow Language*) workflow patterns [125, 126, 123, 64]. We have used these patterns as a basis for modelling in our work.

1.1.3 The Need for Flexible Workflow

Although workflow technologies are generally considered to be an important tool for most business enterprises, it is notable that their deployment has been limited to the support of simple and well-defined business processes. It has been questioned whether workflow, with its roots in characterising manufacturing processes and consequential rigid pre-defined control structure, is suitable to be applied to the representation of business processes generally [96].

Workflows and WfMSs have problems dealing with exceptional circumstances constituting deviations from the set workflow. Often in contemporary WfMSs, the only way to handle change is to circumvent the system by going "behind its back". However, "if users are forced to bypass the WfMS quite frequently, the system is more a liability that an asset" [127].

It is possible in some (mainly academic WfMSs) to specify exception handlers for so-called expected exceptions [28]. Expected exceptions, according to [25], are: "those anomalous situations

that are known in advance to the workflow designer". Provision can be made in the specification of a workflow for this kind of exception. Often this provision will be in the guise of *active rules* (a.k.a. Event Condition Action (ECA) rules), as these exceptional situations cannot be efficiently modelled and handled within the flow structure [25].

Notwithstanding the very few commercial WfMSs providing some support for handling expected exceptions, it is notable that the vast majority of commercial systems can only handle very simple exceptions, such as task deadline expiration. Instead, the workflow designer is forced to model exception handling using the flow constructs provided by the WfMS, which is not efficient, and will most often lead to spaghetti-coding [27].

Another type of workflow exception is unexpected exceptions. These, rather self-explanatorily, are exceptions that the workflow designer has not anticipated. They are typically handled by halting process execution and modifying the workflow definition at the schema or instance level in order to make it consistent with the actual process it represents [27].

In handling expected exceptions through active rules, or in handling unexpected exceptions through stopping the workflow execution, some academic WfMSs support the notion of flexible workflows through allowing manual changes to the workflow definition at run-time. This form of workflow flexibility constitutes a significant research effort within the workflow community, see, for example, [39, 26, 95, 134, 36, 119]. A variant on this notion is elaborated in [35, 73, 105], wherein flexibility stems from the capability, at run-time, to determine how a workflow (according to constraints) may be glued together using a selection of *workflow fragments*.

In our work, however, we are interested in a kind of workflow flexibility that has been proposed in just a handful of academic research efforts, including: CrossFlow [55], Collaboration Management Infrastructure (CMI) [107, 51], Case Handling [127, 16, 96], and work by Wainer and colleagues [131, 132]. The notion is to build flexibility into the specification of a workflow, rather than specifying a rigid control structure that is to be followed at all costs.

Such flexibility means that, for any workflow instance, at any stage of its enactment, there will be potentially many possible path continuations that may be pursued, any of which represents a correct execution of the workflow. It is important not to confuse this sort of flexibility with the possible path continuations that may exist within a traditional workflow specification. In the case of traditional workflow, (it is usually the case that) only one path continuation is possible at any one time. Which continuation is chosen depends on conditions that are specified on the branches that come out of the current activity. In the flexible workflows that we are describing here, many possible path continuations may be possible. Which is chosen depends on some criteria that are applied during workflow enactment. For example, we may have a set of *operational policies* that serve to constrain, or direct, the enactment of a workflow instance. Which path continuation is chosen, at any one time, may be the one that best satisfies these policies.

This idea is neatly encapsulated by the slogan: *Flexible Workflow = Abstract Model + Policies for Refinement*. That is, we define workflow in terms of a relatively abstract and flexible artefact, which is grounded by the use of applicable policies. We identify a correspondence between the refinement of abstract workflow to concrete workflow, through the use of a set of rules, and the operation of a *Hierarchical Task Network* HTN-based planer, which refines abstract task networks to concrete ones, through the use of decomposition rules. In identifying this correspondence, we are able to make use of an HTN-based planner to effect flexible workflow verification and planning.

Our flexible workflow modelling approach also provides support for *collaborative workflows*, where agents are endowed with the capability to decide collectively how a prescribed task or process should be realised.

1.1.4 Contract Modelling

In the field of contract modelling, there have been a number of research contributions, such as [82, 115, 34, 33, 81, 88, 22, 89, 75, 93], which have attempted to address the modelling of contracts for a number of purposes including automating reasoning over them. Common to all of them is the identification of *normative concepts* in contracts.

From [5], a norm may be defined as: "a principle of right action binding upon the members of a group and serving to guide, control, or regulate proper and acceptable behaviour". A normative concept is a conceptualisation of a norm. *Obligation, Permission, Power, Entitlement* are common examples. (See also [63, 92].)

In order to explicate a means by which our work on workflow modelling may be reused in the context of contract modelling, we identify a new perspective for workflow, namely an *institutional perspective*. We call our institutional account of workflow *Institutional Workflow Modelling* (IWM). The institutional perspective draws out the concepts of *counts as* and *permission* which we argue are inherent in workflow. The concept of *count as*, as [63] identifies, is closely related to the legal/contractual concept of *power*, in prescribing how powers may be exercised, arguably the most important and useful aspect of the concept.

Through identifying the institutional concepts of *counts as* and *permission* in workflow, which quite naturally map respectively onto the notions of *power* and *permission*, or *privilege*, in contracts, we are able to propose a way in which IWM may be reused in the modelling of contracts. Moreover, we argue that IWM should be considered as a foundational basis for both normative and contract modelling.

1.2 Contributions and Approach

We consider the contributions made by this thesis to be ten-fold, viz.

1) We address the issue of providing a formal grounding to traditional workflow. In doing so, we contrast the suitability of a number of formal tools for this purpose, namely:

   - Milner's Calculus of Communicating Systems (CCS) [78, 80]
   - Cleaveland et al's Prioritised CCS (PCCS) [30, 29], which we shall call PCCS for convenience
   - Situation Calculus (`SitCalc`) [76, 77, 98], based on First-Order Logic (FOL)

   We chose these formalisms as they provide an interesting contrast in approach when used for modelling workflow, as will be elaborated in later chapters.

   A key motivation in addressing the issue of formal modelling of workflow was to facilitate verification of workflow models, as well as providing a point of reference for implementing workflow engines. As described in the previous section, the verification of workflow models is crucial in order to avoid costly errors in deploying workflow.

We have sourced the representational requirements for our formalisations from the need to be able to represent the YAWL patterns [125, 126, 123, 64], which is a key benchmark in the field of research in Business Process Management (BPM), as well as being able to represent the control flow perspective of WS-BPEL, which is the primary WS-ORCH language today.

2) We have provided an authoring, verification and enactment framework for traditional workflow based on our formalisation. As van der Aalst and colleagues argue [123] "any proposed language should be supported by at least a running prototype in addition to a formal definition". We are of the same opinion, and thus considered it essential to provide such a framework.
3) We have identified a reduced set of workflow patterns, using which (we have shown) all others may be represented, which we believe is a first to be published. Being able to propose such a set enables us to articulate the true nature of workflow and its fundamental representational requirements, which is an important result.
4) We have demonstrated a number of important results using our formal characterisations of traditional workflow.
5) We have proposed a characterisation of flexible workflow to be: *Flexible Workflow = Abstract Model + Policies for Refinement*, in order that we are able to support a more flexible approach to workflow, including support for *collaborative workflows*. In doing so, we have identified a correspondence to *Hierarchical Task Network* (HTN)-based planning. This enables us to recommend the use of an HTN planner for the verification and planned enactment of flexible workflows.
6) We have implemented our own HTN-based planner, which in itself is a useful contribution as it provides many novel features.
7) We have provided an authoring, verification and planned enactment framework for flexible workflow, which uses our HTN-based planner at its core. The term *planned enactment* means that a domain expert may plan the enactment of an abstract workflow through policies that are made available for its refinement.
8) We have proposed a new perspective of workflow, namely an *institutional perspective*. We call our institutional account of workflow *Institutional Workflow Modelling* (IWM). The institutional perspective draws out the concepts of *counts as* and *permission* that we consider to be inherent in workflow. By drawing out these concepts, we are able to identify how workflow may be reused in other contexts. We consider our institutional interpretation of workflow to be novel.
9) We have been keen to see how our work on workflow modelling might be usefully applied elsewhere. By drawing out institutional concepts inherent in workflow, we have been able to propose how workflow may be used in the modelling of contracts. We assert that IWM should be considered as a foundational basis for both normative and contract modelling. Our IWM-based view of contract modelling is novel, and we consider it to be an extremely useful contribution to the field of contract modelling.
10) We have provided an IWM-based framework for contract authoring, verification and (planned) enactment.

1.3 Structure of Thesis

The structure of this thesis is as follows. In **Chapter Two**, we present a comprehensive overview of work that has been carried out regarding the modelling of traditional (i.e. production) workflows. This provides important contextual information for the presentation of the remainder of the thesis.

In **Chapter Three**, we define a meta-model for workflow called `Liesbet`, which constitutes an *information view* abstraction of, or ontology for, workflow. In defining `Liesbet`, we have sought to understand the *true nature* of workflow, and thus the fundamental concepts that need to be represented. We are then able to use this information view of workflow as a point of reference for computational view formalisations of workflow. The representational requirements for `Liesbet` have been sourced from the need to be able to represent the `YAWL` workflow patterns, as well as the control flow aspects of business process languages, such as `WS-BPEL` [87].

In this chapter, we also take our first step towards greater flexibility in workflow models through the proposal of *Synchronisation Rules*. In contrast to the view of flexible workflow that is principally espoused in this thesis (i.e. abstract model + policies for refinement), the appropriate slogan in this instance is more *Flexible Workflow = Concrete Model + Policies for Constraint*. That is to say, the initial model is fully-specified and the policies (i.e. synchronisation rules) *constrain* enactment. The model may contain many possible enactment paths (in contrast to traditional workflow, where typically only one will turn out to be possible). Which of the multiple paths is chosen is constrained by the policies.

Later in Chapter Three, we present a reduced set of patterns with which (we show) all patterns may be represented. This is a useful result as it enables us to propose the true nature of workflow to be this reduced set. We propose the reduction at the level of the `Liesbet` meta-model. That is, we define equivalences for the remaining constructs as definitions which make use only of constructs from the reduced set. These equivalences are argued (and shown) to be sound in Chapter Six.

We finish the chapter by showing how `Liesbet` captures all of the `YAWL` patterns, as well as describing its support for modelling the control flow perspective of `WS-BPEL`.

Chapter Four shows how `Liesbet` can be used to represent some examples of workflow proposed by members of the BPM community. These chosen examples have been suggested as benchmarks by which ontologies for workflow should be evaluated. They also provide a good coverage of `Liesbet`'s constructs, thus providing some examples for the interested reader to understand the operation of the patterns.

In **Chapter Five**, we present our proposed CCS-based characterisations of the operational semantics of `Liesbet`. We selected CCS/PCCS for two reasons:

1) There has been quite a lot of talk within the BPM community as to whether Petri nets or CCS/π-calculus is better suited for the characterisation of workflow, and specifically the `YAWL` patterns [122]. While we do not seek to compare these two formalisms at length, by characterising `YAWL` with CCS we are able to provide a contribution to this debate from one perspective. Note that we do present some points regarding their respective suitability at the end of Chapter Five.
2) The operational semantics of CCS/PCCS (in terms of facilitating compositional specifications of behaviour) should lend themselves quite well to the representation of workflow, and this is a point we seek to investigate.

Ultimately, we conclude that while CCS/PCCS is able to capture certain aspects of semantics of `Liesbet` well, it is deficient in being able to capture straightforwardly the additional constraints

specified for the intended semantics of `Liesbet` which are described above, and in Chapter Three.

In **Chapter Six**, we present our Situation Calculus-based characterisation of `Liesbet`. A motivation for investigating the use of the Situation Calculus was that, as a logic-based formalism, it is quite different to a process algebra based approach for characterising the behaviour of dynamic systems. Moreover, we felt that certain aspects in which CCS/PCCS may be deficient may be better addressed using the Situation Calculus, such as capturing the additional constraints to the intended semantics. This intuition proved to be sound. It is interesting to note that the shortcomings of using CCS/PCCS tend to be advantages when using the Situation Calculus and vice-versa, and thus presenting both in this thesis provides an insightful contrast. At the end of Chapter Six, we provide a discussion regarding the relative merits of each approach.

In **Chapter Seven**, we provide details regarding the verification approach for `Liesbet` models that we have implemented in our work. We are able to verify both soundness (i.e. absence of locking and dead tasks), and arbitrary temporal constraints written in a language such as `CTL*`. We present a number of ways in which the complexity of verification may be ameliorated, and give an interesting characterisation of the complexity of our verification approach.

In **Chapter Eight**, we address the particularly significant issues of traditional workflow identified earlier, that they are brittle in nature in the face of exceptional behaviour and ill-suited to the definition of collaborative workflow. In collaborative workflows, agents should have the facility to decide collectively how a prescribed task or process is to be realised. In this chapter, we provide a contribution to the modelling of flexible workflows, in order to address these issues.

Our approach to flexible workflow modelling is based on the view that *Flexible Workflow = Abstract Model + Policies for Refinement.* We identify a correspondence between refining (as prescribed by our view on workflow) an abstract workflow (specified for flexible enactment) into a concrete one, and the operation of an *Hierarchical Task Network* (`HTN`)-based planner, which refines abstract task networks into concrete ones. In light of this correspondence, we make use of an `HTN`-based planner in our work, implementing our own planner called `Theodore`.

A key theme in our work in flexible workflow modelling is the notion that we combine structure with flexibility. That is, we start with an abstract workflow model which provides some initial structure. Furthermore, we note there is structure inherent within the policies for refinement in that they prescribe networks of actions which are acceptable refinements of tasks being decomposed. Moreover, structure may be prescribed from the bottom-up, in specifying complete refinements of tasks. All of these dispensations, with respect to structuring, help reduce the complexity of verification.

Our work on both traditional and flexible workflow modelling leads us to question how we might apply this work in other contexts. We address this question in **Chapter Nine**. For our work, a natural application is that of contract modelling, where contracts are often cast as protocols (i.e., workflows) of behaviour between two or more parties. We have been motivated to look at the issue of contract modelling for its own sake as well.

In **Chapter Ten**, we present details of the implementation of the verification and enactment frameworks for `Liesbet` (for traditional workflow modelling) and `Theodore` (for flexible workflow modelling). In **Chapter Eleven**, we give examples of using our implementation to verify `Liesbet`-specified and `Theodore`-based workflow models.

We conclude the thesis in **Chapter Twelve** with some salient points. We are able to propose

a minimal view of workflow, which we propose as its fundamental and true nature. We conclude that:

- Workflow is little more than parallel composition with arbitrary synchronisation constraints on the progression of individual activities.

- Expressivity of workflow rests with the choice/suitability of the language for the synchronisation constraints.

We argue that process algebras (such as CCS/PCCS) and logic-based formalisms (such as Situation Calculus) provide rather complementary features for the formalisation of workflow, as we see it. As such, the most appropriate means of formalisation would need to lie somewhere between the two. We specify a number of criteria that any bespoke formalism for workflow would need to take into account. We conclude the chapter with a statement regarding future work activities.

Two appendices provide additional information regarding the CCS- and `SitCalc`-based characterisations of `Liesbet`.

1.4 Declaration

This thesis describes work carried out in the Department of Computing at Imperial College London between 2003 and 2007. I declare that the work presented in this thesis is my own, except where acknowledged.

Andrew Derrick Hotchkiss Farrell

1.5 Publications Contributing to Thesis

The following publications are solely the work of the author of this thesis together with his supervisor Professor MJ Sergot. Other co-authors contributed by reviewing the works prior to publication.

Journal Articles

- Andrew D H Farrell, Marek J Sergot, Mathias Sallé, and Claudio Bartolini. *Using the Event Calculus for Tracking the Normative State of Contracts.* International Journal of Cooperative Information Systems (IJCIS), 14(2–3), 2005. World Scientific.

- Andrew D H Farrell, Marek J Sergot and Claudio Bartolini. *Formalising Workflow: A CCS-inspired Characterisation of the YAWL Workflow Patterns.* Group, Decision and Negotiation (GDN) Journal. 16(3):213-254, 2007. Springer.

Workshops

- Andrew D H Farrell, Marek J Sergot. *Characterising Contracts and Workflows for Static Analysis and Enactment According to Strategic Objectives.* Extended Abstract for FMEC (Formal Modelling in Electronic Commerce) 2005 Workshop, Bologna, 2005.

- Andrew D H Farrell, Marek J Sergot, Mathias Sallé, and Claudio Bartolini. *Using the Event Calculus for the Performance Monitoring of Service-Level Agreements for Utility Computing.* Workshop on Contract Architectures and Languages (CoALa04), 21 September 2004, Monterey, CA, USA. 2004.

- Andrew D H Farrell, Marek J Sergot, Mathias Sallé, Claudio Bartolini, David Trastour, and Athena Christodoulou. *Performance Monitoring of Service-Level Agreements for Utility Computing Using the Event Calculus.* In Proceedings of First IEEE International Workshop on Electronic Contracting, 6 July 2004.

Chapter 2

Background on Traditional Workflow Modelling

We now present an overview of work related to the traditional modelling of workflow, i.e. production workflow. It is necessary to do so in order to provide contextualising information that is pertinent to the rest of the thesis. The layout of the chapter is as follows.

We start with a brief overview of the YAWL Workflow Patterns, which constitute the principal representational requirements for our work on workflow modelling. We follow that with an introduction to the Web Service Composition language WS-BPEL which also contributes to the set of representational requirements.

We then present an overview of other formal modelling approaches to workflow, in order that we may later contrast our approach with these. Finally, as we are interested in verification, it is interesting to note some approaches to the verification of WS-BPEL compositions.

2.1 YAWL Workflow Patterns

The YAWL[1] Workflow Patterns[2] are a collection of artefacts, for the control perspective of workflow. We proceed with an overview of these patterns.

We categorise the presentation (slightly differently from [64]) into the following six categories:

(1) Parallel and Sequence Patterns – Sequence, Interleaved Parallel Routing, Parallel Split.

(2) Choice Patterns – Exclusive Choice, Multiple Choice, Deferred Choice.

(3) Synchronisation Patterns – Synchronisation, Simple Merge, Synchronising Merge, Multiple Merge, Discriminator, Milestone.

(4) Multiple Instance Patterns.

(5) Cancellation Patterns – Cancel Activity, Cancel Case.

(6) Structural Patterns – Arbitrary Cycles, Implicit Termination.

We now describe each of these categories, in turn. The numbering associated with the pattern definitions is that used in [64]. The descriptions here are quite concise. More information concerning the patterns is given in Chapter Three.

[1] YAWL is an acronym for *Yet Another Workflow Language*.

[2] The collation of these patterns preceded the definition of YAWL (by the same research group); but we choose to label them as the YAWL patterns, as they constitute the fundamental representational criteria for the YAWL language.

2.1.1 Parallel and Sequence Patterns

The patterns in this category concern prescribing an ordering over the execution of activities:

- *Sequence* (Pattern #1 in [64]) prescribes a total ordering over (the running of) a collection of activities.

- *Interleaved Parallel Routing* (#17) which, in this thesis, we intuitively call *UnorderedSeq*, prescribes no ordering over a defined collection of activities, but stipulates that they may not run concurrently.

- *Parallel Split* (a.k.a. AND-split) (#2) prescribes that a defined collection of activities may run concurrently, without ordering constraints.

2.1.2 Choice Patterns

At a point, within a workflow model, execution may diverge along many branches. These patterns pertain to this notion:

- *Exclusive Choice* (a.k.a. XOR-split) (#4) – execution continues along one branch of the choice artefact.

- *Multiple Choice* (a.k.a. OR-split) (#6) – execution may continue along $\leq n$ branches of the choice artefact, where n is the number of branches of the artefact.

- *Deferred Choice* (#16) – For the first two choice patterns, the choice of branches to follow is made based on control data maintained within the model (at the data perspective). For *Deferred Choice*, the choice of branch/es, along which execution may continue, is made by an external agent.

2.1.3 Synchronisation Patterns

When execution, of a workflow model, has gone down one or more branches of a choice or parallel artefact, it would (typically) be desirable to (eventually) merge the pertaining threads of execution at a synchronisation point, before continuing execution. These patterns relate to such synchronisation:

- *Synchronisation* (a.k.a. AND-join) (#3) – For merging the threads of a *Parallel Split* artefact. That is, synchronisation is satisfied once all threads of the split have completed.

- *Simple Merge* (a.k.a. XOR-join) (#5) – For synchronising on the completion of the single branch chosen for execution in an *Exclusive Choice* artefact.

- *Synchronising Merge* (a.k.a. OR-join) (#7) – For synchronising on the completion of the, potentially, many branches chosen for execution in an *Multiple Choice* artefact.

- *Multiple Merge* (#8) – Whenever any branch chosen (for execution) in a *Multiple Choice* artefact completes, execution of a named continuation activity is initiated. There will be a separate instance of the continuation activity created for every branch that completes.

- *Discriminator* (#9) – Once a number of branches, chosen (for execution) in a *Multiple Choice* artefact, have completed then the execution of a named continuation activity is initiated. Only a single instance of the continuation activity will be created.

- *Milestone* (#17) – A synchronisation point, in a workflow model, that is satisfied iff one named activity has completed *and* another named activity is yet to start.

2.1.4 Multiple Instance Patterns

The following patterns allow a multiple number of instances of a named activity to be executed:

- *Multiple Instances without Synchronisation* (#12) – A number of instances of a named activity may be executed, without the need to synchronise continuation of the pertaining thread of execution on their completion.

- Multiple Instances with Synchronisation (#13-#15) – A number of instances of a named activity need to be executed, *with* the need to synchronise continuation of the pertaining thread of execution on their completion:

  - *Multiple Instances with a priori design-time knowledge* (#13) – The number of instances is known when the workflow model is being authored.

  - *Multiple Instances with a priori run-time knowledge* (#14) – The number of instances is known only at run-time, but before execution of the multiple-instance artefact has been initiated.

  - *Multiple Instances without a priori run-time knowledge* (#15) – The number of instances is not known until execution of the multiple-instance artefact has been completed.

2.1.5 Cancellation Patterns

These patterns effect the cancellation of activities in a workflow model:

- *Cancel Activity* (#19) – Effects cancellation of a named activity.

- *Cancel Case* (#20) – Effects cancellation of the entire instance of the workflow model being executed.

2.1.6 Structural Patterns

These patterns pertain to miscellaneous artefacts which are considered to be essential in making the specification of workflow models as straightforward as possible:

- *Arbitrary Cycles* (#10) – Entails the need to allow a thread of execution to jump to arbitrary points within a model.

- *Implicit Termination* (#11) – A thread of execution should be implicitly terminated when there is nothing else for the thread to do.

<receive>	Causes execution thread to wait for a (matching) message to arrive
<reply>	Facilitates execution thread sending a message in reply to a message that was received by an inbound message activity (IMA), i.e. <receive>, <onMessage>, or <onEvent>
<invoke>	Allows execution thread to invoke a one-way or request-response operation
<assign>	Used to update the values of variables with new data
<exit>	Causes process instance to end immediately
<wait>	Causes execution thread to wait for a given time period or until some absolute time
<empty>	"no-op" activity, which trivially completes
<sequence>	Defines a collection of activities to be performed sequentially, in lexical order
<if>	Used to select exactly one activity for execution from a set of choices
<while>	Used to define that its single child activity is to be repeated as long as the specified <condition> is true
<repeatUntil>	Used to define that its single child activity is to be repeated until the specified <condition> becomes true, with at least one iteration
<forEach>	Iterates its child scope activity a number of times, either sequentially or concurrently
<pick>	Used to wait for one of several possible messages to arrive or for a time-out to occur. When one of these triggers occurs, the associated child activity is performed
<flow>	Used to specify one or more activities to be performed concurrently. <links> can be used within a <flow> to define explicit control dependencies between nested child activities

Table 2.1: Some WS-BPEL Activity Types (Section 5.2 of [87]).

2.2 Web Services Business Process Execution Language WS-BPEL

In this section, we introduce the *Web Services Business Process Execution Language* WS-BPEL [87]. WS-BPEL is an XML [130]-based language for specifying compositions of web services; and is quickly emerging as the language of choice for this purpose.

As presented in Section 1.1, WS-BPEL describes service compositions from an end-point, or local, perspective. It is, thus, an orchestration language. It can describe business processes in two ways, namely, as *abstract processes*, or as *executable processes*. Abstract processes model the multi-party conversation protocol, by which an end-point may effect its apposite behaviour, pertaining to a composition. Executable processes specify the implementation logic underwriting this end-point behaviour.

A WS-BPEL process is a nested specification of activities, consisting of a single root activity. Table 2.1 summarises a subset of the activity types of WS-BPEL. The set of activity types in WS-BPEL is not minimal. There are cases where the semantics of one activity can be represented using another activity. For example, sequential processing may be modelled using either the <sequence> activity, or by a <flow> with properly defined links (as described in Section 11 of [87]).

The reason that the set is not minimal is due, in part, to the history of WS-BPEL. It is a fusion of two approaches to workflow description, namely, graph- and block-structured specification. A graph-structured specification is essentially a partial ordering over a collection of activities,

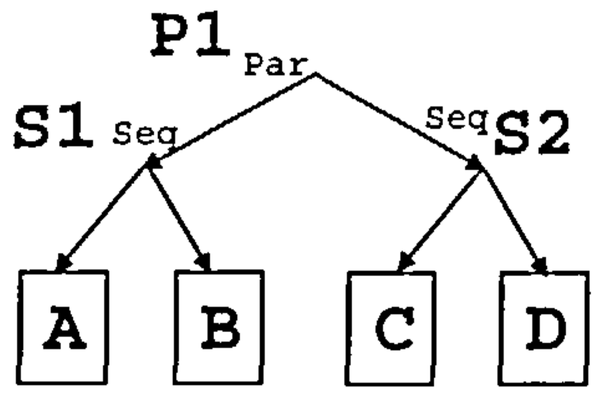


Figure 2.1: Simple Workflow Model

giving rise to constructs such as <flow> and <link> in WS-BPEL; whereas a block-structured specification defines a workflow specification as a nested collection of workflow artefacts, using constructs such as <flow> and <sequence>.

This can be seen in Figure 2.2, which shows WS-BPEL representations of a workflow model, depicted in Figure 2.1, that will be used for illustration purposes throughout this thesis. The model is a parallel composition (P1) of two sequences (S1 and S2), each consisting of two atomic activities (A and B, and C and D, respectively). The first representation, shown in Figure 2.2, is the block-structured version; and the second representation is the graph-structured version.

In the block-structured representation, the two sequences are explicitly represented. In the graph-structured representation, the partial orderings pertaining to the sequences are prescribed by links instead; thus, the sequencing is somewhat obscured.

2.2.1 Start Activities

A WS-BPEL process instance is created whenever a *start activity* completes. A start activity is a <receive> or <pick> activity whose createInstance attribute is set to yes. When a *message* is received by such an activity, an instance of the business process is created if it does not already exist. (Section 5.5 of [87].)

2.2.2 Standard Attributes and Standard Elements

All activities have two optional standard attributes: the name of the activity, and an attribute, suppressJoinFailure, indicating whether the failure of a join condition (described in Section 2.2.3) should be suppressed. (Section 10.1 of [87].)

All activities, also, optionally have two standard elements: <sources> and <targets>, which themselves contain standard elements: <source> and <target>, respectively. These elements are used to establish synchronisation relationships through links (see Section 2.2.3). (Section 10.2 of [87].)

2.2.3 Information Concerning <flow> Activity Type

The <flow> activity provides for concurrent execution of activities, while facilitating the expression of synchronisation dependencies between activities that are nested within it to any depth. The <link> construct, as well as the standard attributes and standard elements (2.2.2) for activities, are used to express these dependencies. (Section 11.6 of [87]). We have already seen an example of the use of links, in Figure 2.2.

```
<process name="Simple Workflow, Block Structured">
   <flow name="P1">
      <sequence name="S1">
         <empty name="A"/>
         <empty name="B"/>
      </sequence>
      <sequence name="S2">
         <empty name="C"/>
         <empty name="D"/>
      </sequence>
   </flow>
</process>

<process name="Simple Workflow, Graph Structured">
   <flow name="P1">
      <links>
         <link name="toB"/>
         <link name="toD"/>
      </links>

      <empty name="A">
         <sources>
            <source linkName="toB"/>
         </sources>
      </empty>

      <empty name="B">
         <targets>
            <target linkName="toB"/>
         </targets>
      </empty>

      <empty name="C">
         <sources>
            <source linkName="toD"/>
         </sources>
      </empty>

      <empty name="D">
         <targets>
            <target linkName="toD"/>
         </targets>
      </empty>
   </flow>
</process>
```

Figure 2.2: WS-BPEL Representations of the Simple Workflow Model

An activity may declare itself to be the source of one or more links by including one or more <source> elements. Similarly, an activity may declare itself to be the target of one or more links by including one or more <target> elements.

The source and target of a link may be nested arbitrarily deeply within the <flow> activity in which the link is declared, except for some boundary-crossing restrictions, described below in Section 2.2.4.

Each <source> element may specify a <transitionCondition>, as described in Section 2.2.5. Moreover, the <targets> container, within an activity definition, may specify a <joinCondition>, as also described in Section 2.2.5.

2.2.4 Link Boundary Crossing Restrictions

A link is said to cross the boundary of an activity iff its source, and/or target, is nested inside the activity, at any level, but the link is not declared inside that construct at any level. A link may not cross the boundary of a repeatable construct (<while>, <repeatUntil>, <forEach>, <eventHandlers>) element. That is, a link used within a repeatable construct must be declared within a <flow> activity that is itself nested inside the repeatable construct. Also, links may not create control dependency cycles. (Section 11.6.1 of [87]).

2.2.5 Link Semantics

If an activity that is otherwise ready to start (e.g., it is the current activity to be executed in a sequence) has incoming links then it may not start until the status of all its incoming links has been determined and the, implicit or explicit, join condition has been evaluated. Evaluation of the join condition may only be performed after the status of all incoming links has been determined. The expression for a join condition is constructed using boolean operators and the status values of the pertaining activity's incoming links. If no join condition is specified, its value is taken to be the disjunction of the status values of all incoming links. (Section 11.6.2 of [87].)

A link may be in one of three states: `true`, `false`, or `unset`. When an activity `A` completes, we must determine the effects that the links, of which `A` is the source, has on the join conditions of activities which are the targets of the given links.

We determine, in sequence, the status of the outgoing links of `A`. To determine the status for each link its transition condition is evaluated. If a transition condition has been omitted for a link, its value is taken to be `true`. For each activity `B` that has a synchronisation dependency on `A`, we check that `B` is otherwise ready to start and that the status of `B`'s other links have been determined. If both of these conditions are satisfied then we evaluate `B`'s join condition. If it evaluates to `true`, then execution of `B` is initiated. If false, then the (possibly inherited) value of the `suppressJoinFailure` attribute would determine what entails. For the purpose of our work, we assume that this attribute is always set to `true`. In this case, we perform *Dead-Path Elimination*, as described in Section 2.2.6.

2.2.6 Dead-Path-Elimination

The status of the outgoing links of an activity `B` must be set to `false` whenever either of the following conditions is satisfied (Sections 11.6.2 and 11.6.3 of [87]):

- If B is not performed due to the value of its (implicit or explicit) join condition being evaluated to false.

- If, during the performance of an activity A, the semantics of A dictate that activity B nested within A will not be performed as part of the execution of A.

  However, to preserve semantic integrity, this rule is only applicable once the status values of all of B's incoming links have been determined. An example where this additional criterion applies is presented in Section 3.6.1.

2.2.7 <scope> Types

<scope> is an additional activity type, used in WS-BPEL. A <scope> container has a single activity, which defines its primary behaviour. A scope may also specify a number of localised handlers of various kinds, which come into effect when the scope begins to be executed. Some of the handlers that may be specified pertain to fault, compensation and termination handling, which are issues that we do not currently address in our work. One kind of handler we do accommodate (from the control flow perspective) pertains to *Event Handling.* (Section 12 of [87].)

The type of events that an event handler may process are inbound message and timer events. Whenever a matching event is received by a handler, the single scope which the handler defines is executed. Once the primary activity of a scope completes, its contained event handlers are immediately disabled. Any outstanding instances of activities created by the scope's event handlers are allowed to complete, and the completion of the scope as a whole is delayed until they complete. (Sections 12.7 and 12.7.5 of [87].)

2.3 Formal Modelling Approaches for Workflow

In this section, we review a number of formal modelling approaches which have, and could, be used to provide a formal semantics to workflow. We consider a handful in detail. However, it should be noted that there exist many other approaches that have been used for workflow modelling, some of which will come to light when we review literature concerning workflow verification (see Section 2.4).

2.3.1 Background Concerning Formal Approaches

We proceed by giving an overview of the following formal approaches: Petri nets [97], CCS [78, 80, 79] and PCCS [30, 29]. We defer presentation of the Situation Calculus [98] to Chapter Six.

Petri nets

A classical *Petri net* [97, 18] is a directed bipartite graph, having two node types: *places* and *transitions.* Places and transitions are connected to each other by directed *arcs.* Nodes of the same type may not be directly connected. Graphically, places are represented by circles and transitions are represented by rectangles. (See, for example, Figure 2.3).

From [120], a Petri net is a triple (P, T, F), where:

- P is a finite set of places.

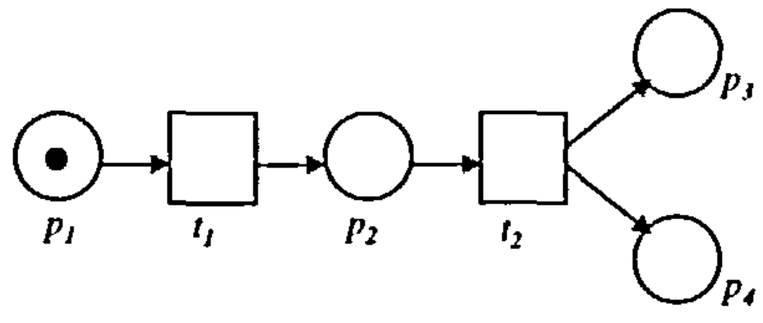


Figure 2.3: An Example Petri net.

- T is a finite set of transitions ($P \cap T = \emptyset$).
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation).

Input Place, Output Place, Preset, Postset

- A place p is an *input place* of transition t iff there exists a directed arc from p to t.
- A place p is an *output place* of transition t iff there exists a directed arc from t to p.
- The set of input places of a transition t is called its *preset*, and is denoted $\bullet t$.
- The set of output places of a transition t is called its *postset*, and is denoted $t\bullet$.

Marking, Transition Firing and Reachable

- A place can contain an arbitrary number of tokens (graphically represented as black dots). The distribution of tokens over places is called the net's *marking*.
- The marking of a Petri net evolves according to *transition firing*. A transition t may fire iff each place in $\bullet t$ has a token. When a transition fires, it consumes a single token from each $p \in \bullet t$ and deposits a single token in each $p \in t\bullet$. There are many enhancements of this simple model, such as having arcs specify an arbitrary number of tokens (≥ 1) to be removed from a place when a transition fires – see, for example, [97, 18].

  In Figure 2.3, transition t_1 is ready to fire, on account of the single place in its preset, p_1, containing a token. On firing, the token is consumed from p_1, and a token deposited in p_2. After this occurrence, t_2 is ready to fire. On firing, the token in p_2 is consumed, and a single token is deposited in each of p_3 and p_4.
- A marking M_n is *reachable* from a marking M_1 iff there is a sequence of transition firings that takes the net from M_1 to M_n. There are two markings reachable from the initial marking of the net shown in Figure 2.3.

CCS

We present a brief overview of CCS. For readers unfamiliar with CCS, [49, 78, 80, 79] are excellent starting points. In our work, we use the Concurrency Workbench for the New Century (`CWB-NC`) [11], for verification purposes. The use of `CWB-NC` prescribes certain syntactic conventions, which will be highlighted as appropriate.

We assume the availability of an infinite set of action names[3] $\mathcal{N}$, ranged over by $a,b,\ldots$, and a corresponding set of co-names (or, co-actions) $\overline{\mathcal{N}} = \{\overline{a} | a \in \mathcal{N}\}$, where $\mathcal{N}$ and $\overline{\mathcal{N}}$ are disjoint and in bijection via $\bar{}$, and where $\overline{\overline{a}} = a$. The set $\mathcal{L}{=}\mathcal{N} \cup \overline{\mathcal{N}}$ is the set of *labels*, ranged over by l and $\bar{l}$, and τ is a distinguished *silent action*, such that $\tau \notin \mathcal{L}$. The set $Act{=}\mathcal{L} \cup \tau$ is the set of actions that may be performed by a CCS *agent*. We assign $\alpha, \beta, \ldots$ to range over Act.

The set $\mathcal{E}$ of CCS *agents* is defined inductively. It is the smallest set which includes the following expressions, where E and E_i are already in $\mathcal{E}$ (from [78]):

- agent constants
- $\alpha.E$ – *prefix* ($\alpha \in Act$). Note that in `CWB-NC`, `'a` denotes $\overline{a}$, an output on a
- $\sum_{i \in I} E_i$ – *summation*, where the indexing set I may be empty, in which case we write `0` (`nil` in `CWB-NC`) to indicate the deadlocked agent
- $E_1 | E_2$ – *composition*
- $E \backslash L$ – *restriction* ($L \subseteq \mathcal{L}$)
- $E[f]$ – *relabelling*. The relabelling function $f : Act \rightarrow Act$ relabels action names, where $f(\bar{l}) = \overline{f(l)}$ and $f(\tau) = \tau$.

An *agent constant* is an agent whose meaning is given by a defining equation. In the definition $A \stackrel{def}{=} E$, A is an agent constant, and E an agent. The definitional mechanism is the means by which recursive behaviour may be defined.

In CCS, a system is characterised by a number of agents which may perform transitions. Note that we often use the term *synchronisation* for τ-transitions, in order to emphasise the notion that two agents are performing individual transitions in synchrony. The transitions that a system may make define a *labelled transition system* [78] ($\mathcal{E}$, Act, $\{\stackrel{\alpha}{\rightarrow}\}$), where $\stackrel{\alpha}{\rightarrow} \subseteq E \times E$ is a *transition relation* for each $\alpha \in Act$. The *operational semantics* for the set of agents, $\mathcal{E}$, is given by the definition of each transition relation $\stackrel{\alpha}{\rightarrow}$ over $\mathcal{E}$. The following set of transition rules enable us to build the transition relations over each agent in $\mathcal{E}$, using `Act` to begin with.

`Act`
$$\frac{}{\alpha.E \stackrel{\alpha}{\rightarrow} E}$$

`Sum`$_j$
$$\frac{E_j \stackrel{\alpha_j}{\rightarrow} E'_j}{\sum_{i \in I} E_i \stackrel{\alpha_j}{\rightarrow} E'_j} \; j \in I$$

`Com`$_1$
$$\frac{E \stackrel{\alpha}{\rightarrow} E'}{E|F \stackrel{\alpha}{\rightarrow} E'|F}$$

`Com`$_2$
$$\frac{F \stackrel{\alpha}{\rightarrow} F'}{E|F \stackrel{\alpha}{\rightarrow} E|F'}$$

`Com`$_3$
$$\frac{E \stackrel{l}{\rightarrow} E' \quad F \stackrel{\bar{l}}{\rightarrow} F'}{E|F \stackrel{\tau}{\rightarrow} E'|F'}$$

`Res`
$$\frac{E \stackrel{\alpha}{\rightarrow} E'}{E\backslash L \stackrel{\alpha}{\rightarrow} E'\backslash L} \; (\alpha, \overline{\alpha} \notin L)$$

`Rel`
$$\frac{E \stackrel{\alpha}{\rightarrow} E'}{E[f] \stackrel{f(\alpha)}{\rightarrow} E'[f]}$$

`Con`
$$\frac{E \stackrel{\alpha}{\rightarrow} E'}{A \stackrel{\alpha}{\rightarrow} E'} \; (A \stackrel{def}{=} E)$$

[3] Actions are also known as *channels*, conveying the notion of computation through communication.

- Act allows us to infer transitions for prefixed agents. That is, the agent $\alpha.E$ may make a transition labelled with action α to the agent E.

- Given an agent E_j which makes an α_j-labelled transition to agent E', we may, by $\mathtt{Sum}_j$, infer an α_j-labelled transition for a summation agent $\sum_{i \in I} E_i$, where $j \in I$, such that it too transitions to E'.

- Given an agent E which makes an α-labelled transition to agent E', we may, by $\mathtt{Com}_1$, infer an α-labelled transition for a composed agent $E \mid F$ such that it transitions to $E' \mid F$. Similarly, $\mathtt{Com}_2$ allows us to infer an α-labelled transition for the right-hand agent in a composition.

- Given two agents E and F that make complementary l-labelled transitions to E' and F', respectively, we may, by $\mathtt{Com}_3$, infer a τ-transition for the composed agent $E \mid F$ to $E' \mid F'$.

- Given an agent E which makes an α-labelled transition to agent E', we may, by Res, infer an α-labelled transition for the restricted agent $E \backslash L$ so long as α or its co-action, is not in L. The restriction $\backslash L$ has the effect of restricting the *scope* of an action in E, when named in L, to be E.

- Given an agent E which makes an α-labelled transition to agent E', we may, by Rel, infer an $f(\alpha)$ (relabelled) transition from $E[f]$ (which is the result of relabelling names comprising agent E by f) to $E'[f]$ (which is the result of a similar f-relabelling of E').

- Given an agent E which makes an α-labelled transition to agent E', we may, by Con, infer an α-labelled transition for A to E' just in case A is an agent constant whose definition is E.

PCCS

PCCS, as proposed in [29], is based on the prioritised calculus presented in [30]. It is essentially CCS with priorities which are specified as natural numbers attached to actions. The smaller the number, the higher the priority becomes, with zero having the highest priority. Note that just *τ-transitions* of a certain priority take precedence over transitions of a lower priority. Non-τ transitions are *not* capable of effecting any priority over other transitions.

The set of labels, $\mathcal{L}$, of the PCCS language is the union of a family of pairwise-disjoint, countably infinite sets of labels, $\mathcal{L}_k$ (for $k \in \mathbb{N}$). As documented in [29], $\mathcal{L}_k$ (ranged over by $l{:}k$ and $\bar{l}{:}k$) contains the action names of priority k that agents may synchronise over. The set of actions of priority k, Act_k, is defined as $\mathcal{L}_k \cup \{\tau_k\}$, where $\tau_k \notin \mathcal{L}_k$. τ_k may also be denoted as $\tau : k$, for consistency. $\tau : k$ actions represent internal computation steps of priority k within a model. The set of actions, Act, is defined as $\cup Act_k$.

The set $\mathcal{E}$ of PCCS *agents* is defined inductively. It is the smallest set which includes the following expressions, where E and E_i are already in $\mathcal{E}$:

- agent constants
- $\alpha : k.E$ $(\alpha : k \in Act_k)$
- $\sum_{i \in I} E_i$
- $E_1 \mid E_2$

- $E\backslash L$
- $E[f]$
- $E_1[> E_2$ – *disable.* E_2 disables E_1 as soon as it is able to make a transition.

The semantics of PCCS are given by a labelled transition system, in a similar way to that presented for CCS, previously.

The transitions that a system may make define a labelled transition system $(\mathcal{E}, Act, \{\stackrel{\alpha:k}{\rightarrow}\})$, where $\stackrel{\alpha:k}{\rightarrow} \subseteq E \times E$ is a *transition relation* for each $\alpha : k \in Act$. If $E \stackrel{\alpha:k}{\rightarrow} E'$, then we say that E may engage in action α of priority k and thereafter behave like agent E' [29].

The following transition rules allow us to infer transitions for agents, E, within the set of agents $\mathcal{E}$.

$$\frac{}{\alpha : k.E \stackrel{\alpha:k}{\rightarrow} E} \quad \text{Act}$$

$$\frac{E \stackrel{\alpha:k}{\rightarrow} E'}{E+F \stackrel{\alpha:k}{\rightarrow} E'} \; \tau : k \notin I_{<k}(F) \quad \text{Sum}_1 \qquad \frac{F \stackrel{\alpha:k}{\rightarrow} F'}{E+F \stackrel{\alpha:k}{\rightarrow} F'} \; \tau : k \notin I_{<k}(E) \quad \text{Sum}_2$$

$$\frac{E \stackrel{\alpha:k}{\rightarrow} E'}{E|F \stackrel{\alpha:k}{\rightarrow} E'|F} \; \tau : k \notin I_{<k}(E|F) \quad \text{Com}_1 \qquad \frac{F \stackrel{\alpha:k}{\rightarrow} F'}{E|F \stackrel{\alpha:k}{\rightarrow} E|F'} \; \tau : k \notin I_{<k}(E|F) \quad \text{Com}_2$$

$$\frac{E \stackrel{l:k}{\rightarrow} E' \; F \stackrel{\bar{l}:k}{\rightarrow} F'}{E|F \stackrel{\tau:k}{\rightarrow} E'|F'} \; \tau : k \notin I_{<k}(E|F) \quad \text{Com}_3$$

$$\frac{E \stackrel{\alpha:k}{\rightarrow} E'}{E\backslash L \stackrel{\alpha:k}{\rightarrow} E'\backslash L} \; (\alpha : k, \overline{\alpha} : k \notin L) \quad \text{Res} \qquad \frac{E \stackrel{\alpha:k}{\rightarrow} E'}{E[f] \stackrel{f(\alpha:k)}{\rightarrow} E'[f]} \quad \text{Rel} \qquad \frac{E \stackrel{\alpha:k}{\rightarrow} E'}{A \stackrel{\alpha:k}{\rightarrow} E'} \; (A \stackrel{def}{=} E) \quad \text{Con}$$

$$\frac{E \stackrel{\alpha:k}{\rightarrow} E'}{E[>F \stackrel{\alpha:k}{\rightarrow} E'[>F} \; \tau : k \notin I_{<k}(F) \quad \text{Disable}_1 \qquad \frac{F \stackrel{\alpha}{\rightarrow} F'}{E[>F \stackrel{\alpha:k}{\rightarrow} F'} \; \tau : k \notin I_{<k}(E) \quad \text{Disable}_2$$

The presentation of the given transition rules uses the notion of *initial action sets*, which are inductively defined as follows.

$$I_k(\alpha : j.E) \stackrel{def}{=} \{ \alpha : j \mid j = k \}$$
$$I_k(E + F) \stackrel{def}{=} I_k(E) \cup I_k(F)$$
$$I_k(E[f]) \stackrel{def}{=} \{ f(\alpha : k) \mid \alpha : k \in I_k(E) \}$$
$$I_k(E\backslash L) \stackrel{def}{=} I_k(E)\backslash L \cup \overline{L}$$

$I_k(E|F) \stackrel{def}{=} I_k(E) \cup I_k(F) \cup \{ \tau : k \mid I_k(E) \cap \overline{I}_k(F) \neq \emptyset \}$
$I_k(A) \stackrel{def}{=} I_k(E)$ where $A \stackrel{def}{=} E$
$I_k(E[> F) \stackrel{def}{=} I_k(E) \cup I_k(F)$
$I_{<k}(E) \stackrel{def}{=} \cup \{ I_j(E) \mid j < k \}$

Note that $I_k(E)$ denotes the set of all initial actions of E with priority k and $I_{<k}(E)$ denotes the set of all initial actions of E with a higher priority than k.

We now explain some of these rules. The rest are intuitive. Refer to [29] for more information.

- `Act` – the agent $\alpha : k.E$ may perform an α-transition of priority k.
- $\texttt{Sum}_1$ and $\texttt{Sum}_2$ – the agent $E + F$ may perform an α-transition of priority k and behave thereafter as E' (resp. F') if E (resp. F) may perform an α-transition of priority k to yield E' (resp. F') and F (resp. E) does not pre-empt it by performing a higher priority τ-transition.
- $\texttt{Com}_1$ and $\texttt{Com}_2$ – the agent $E|F$ may perform an α-transition of priority k and behave thereafter as $E'|F$ (resp. $E|F'$) if E (resp. F) may perform an α-transition of priority k to yield E' (resp. F') and $E|F$ does not pre-empt it by performing a higher priority τ-transition.
- $\texttt{Com}_3$ – the agent $E|F$ may perform a τ-transition of priority k and behave thereafter as $E'|F'$ if E may perform an l-transition of priority k to yield E' and F may perform an $\overline{l}$-transition of priority k to yield F' and $E|F$ does not pre-empt it by performing a higher priority τ-transition.
- $\texttt{Disable}_1$ and $\texttt{Disable}_2$ – the agent $E[> F$ may perform an α-transition of priority k and behave thereafter as $E'[> F$ (resp. F') if E (resp. F) may perform an α-transition of priority k to yield E' (resp. F') and F (resp. E) does not pre-empt it by performing a higher priority τ-transition

2.3.2 Application of Formal Approaches to Workflow Modelling

We now proceed to give an overview of some formal-based approaches to workflow modelling. Firstly, we look at `WF-nets` [120], and follow that with reviews of a number of contributions whose common purpose is to formalise the `YAWL` patterns.

The advantage of using well-established formal tools, such as Petri nets and CCS, for workflow modelling are the abundance of analysis techniques and automated tools that exist for them. This point will be elaborated in the review section on *verification*, see Section 2.4.

WF-nets

In [120], a Petri net which models the control perspective of workflow is called a *Workflow net* (`WF-net`). A Petri net (P,T,F) is a `WF-net` iff:

- There is one source (resp. sink) place $i \in P$ (resp. $o \in P$) such that $\bullet i = \emptyset$ (resp. $o\bullet = \emptyset$).
- Every node $x \in P \cup T$ is on a path from i to o.

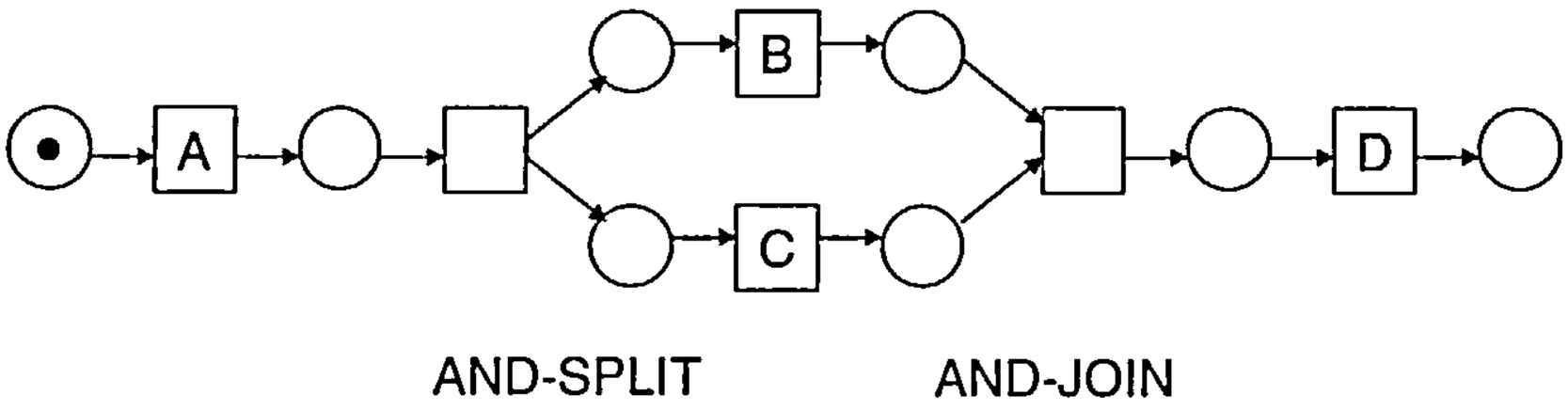


Figure 2.4: AND-split/join and XOR-split/join.

Transitions correspond to tasks, although only a subset of these will correspond to tasks required of agents – the others will pertain to internal housekeeping. Places are conditions on the execution of transitions, i.e. tasks.

The first condition ensures that it is clear how a case (i.e. execution instance of a workflow model) enters, and exits, the model. The second condition ensures that there are no dangling transitions (i.e. tasks), or conditions, in the definition of the model.

In Figure 2.4, we present example nets encoding AND-split/join and XOR-split/join behaviours, pertaining to YAWL patterns #2/#3 and #4/#5, respectively. In the first net, activity A is followed by both B and C in parallel. When both B and C are finished, D is executed. In the second net, only one of B or C is executed.

YAWL – A Petri net based Approach to Workflow Modelling

YAWL [125, 126, 123] is a Petri net-like graphical language, whose primary purpose is to facilitate the specification of workflow models, from the control perspective. It uses the YAWL workflow patterns as a basis for its definition; that is, the patterns specify what the language should be able to represent *succinctly*. The creators of YAWL highlight three patterns that they assert as being problematic to model using Petri nets (similarly, WF-nets):

- Multiple-instance activity types – the burden of keeping track, splitting and joining of instances is carried by the designer.
- Advanced synchronisation patterns – for synchronising multiple paths, where it cannot be pre-determined how many of the paths will require synchronisation.
- Cancellation patterns – it is hard to model cancellation patterns because it is not possible to predict how many tokens should be removed from each pertinent place.

Interestingly, YAWL does not resolve the issue of advanced synchronisation particularly well, as is evidenced by later efforts, such as [140], which try to resolve the matter. The lack of proper support for this issue goes against the desire, espoused in [125], to not burden a workflow author with subtle, low-level concerns.

YAWL defines EWF-nets, which extend WF-nets with direct support for *multiple instances*, *composite tasks*, *OR-joins* and *removal of tokens*. It defines a Petri net-like graphical notation; but, importantly, its semantics are not Petri net-based – rather, YAWL is based directly on a transition system-based semantics, where a *binding* relation determines possible transitions. Through these semantics, it also aggregates support for AND/OR/XOR-splits and AND/OR/XOR-joins, in that every task is a kind of join and a kind of split. This is in contrast to needing to explicitly wire this functionality using regular Petri nets; that is, specifying workflow with Petri nets may be seen as a more low-level representation.

A task in an EWF-net can have multiple instances; where it is possible to specify a lower bound and an upper bound for the number of instances that may be created after initiating the task. It is also possible to indicate that the task terminates the moment a certain threshold of instances has completed[4]. If no threshold is specified, the task completes once all instances have completed.

The YAWL patterns (Section 2.1) are trivially supported, with the following caveats:

- Discriminator (#9) – Facilitated as a multiple-instance activity, "under the assumption of multiple instances of the same task" [125]. The threshold for instances completing, which will cause the execution of the continuation activity, is set as desired. Any remaining instances, still executing, are terminated. Its representation, as a multiple-instance activity, is not *necessarily* the intended sense of the Discriminator pattern, as elaborated in Section 3.1.14.
- Implicit Termination (#11) – The authors of [125] make the point that this pattern should not be supported so that workflow authors think properly about termination. Our impression is that this is a Petri net-centric perspective, and whether an author should need to think about termination really comes down to the authoring tool they are using. If they are using a Petri net-based authoring tool, then this is a fair point – Petri nets naturally lend themselves to this way of thinking. If they are using something more abstract, then it is often the case that this pattern is justified. In our meta-model (see Section 3), it is arguably more natural to allow implicit termination.

CCS-based Approaches to Modelling of YAWL Patterns

There have been a few approaches to the modelling of the YAWL workflow patterns using CCS-based approaches, namely: Stefansen [117, 116], Dong and Shensheng [37] and Puhlman and Weske [94].

Arguably, the most mature of these, currently, would appear to be the work of Stefansen. Some (slightly tweaked) examples of the formalisation of the YAWL patterns that he proposes are as follows [117] (omitting the agent terminator nil for convenience):

- *Sequence* (#1) – $(P[go/done] \mid go.Q)\backslash\{go\}$. In Stefansen's formalisation, an agent pertaining to a piece of workflow logic will signal on *done* when it is otherwise finished. When

[4]Although termination does not appear to be enforced by the semantics. That is to say, there is a transition specified by *binding* that does force such termination; but it is not specified at a higher priority and, as such, instances of the task may continue to execute, and complete, before the transition effecting termination occurs.

using specific instances of agents, it is appropriate to rename *done* to something unique, so that its visibility is appropriately restricted. In this example, *done* is renamed to *go*, and a synchronisation on *go* will indicate that P has otherwise finished, and that Q may be progress.

- *Parallel* (#2) – $P_1 \mid \ldots \mid P_n$.

- *Synchronisation* (#3) – $(P_1[ok/done] \mid \ldots \mid P_n[ok/done] \mid ok.\ldots.ok.Q)\backslash\{ok\}$. There needs to be n synchronisations on *ok* (indicating that all n parallel threads of execution have completed) before Q may execute.

- See [117] for patterns #4–#17.

- *Milestone* (#18) – $P_{set} \mid {}_{clear}R \mid Q' \mid Milestone$, where proc $Q' = isOn.(Q \mid Q')$. Here, P_{set} is the agent P modified to output on *set* as its last action, irrespective of how it evolves, and ${}_{clear}R$ is the agent R modified to output on *clear* after its first action. *set* (resp. *clear*) sets (resp. clears) the milestone, i.e. starts (resp. stops) the period when execution of instances of Q may be initiated. *Milestone* keeps track of whether the milestone is currently set or cleared, and provides *isOn* to query whether it is set. The agent Q' is a wrapper for Q which also yields another copy of Q' meaning that unlimited copies of Q may be created while the milestone is set.

- *Cancel Activity* (#19) – If it were intended that the execution of an activity b, say, may be cancelled, then it would be represented as $b.('b + cancel)$, as opposed to b, otherwise. Synchronisation on *cancel* would have the effect of pre-empting the completion of b, which would, otherwise, be signified by the output on b. Note that in the example, given in [117], where b is part of a three activity sequence, $a.b.c$, the remainder of the sequence is also cancelled (which is not necessarily an appropriate interpretation of the intended semantics for YAWL).

- *Cancel Case* (#20) – Each activity is split in the way stated for pattern (#19), with the possibility of receiving a termination signal pertaining to case cancellation.

2.4 Workflow Verification

As we allude to in the introduction to this thesis (see Chapter One), it is important to analyse a workflow model before it is used. Indeed, [120] states "[t]he correctness, effectiveness and efficiency of the business processes supported by [a] WFMS are vital to [an] organisation". van der Aalst [120] enumerates three types of analysis:

- *Validation* – testing whether the workflow behaves as expected; achieved through interactive simulation.

- *Verification* – establishing the correctness of a workflow.

- *Performance Analysis* – evaluating the ability to meet requirements with respect to throughput times, service levels and resource utilisation.

These types of analysis are equally important when considering compositions of web services. Additionally, it may be useful to be able to check the internal behaviour (i.e., in a WS-BPEL context, executable process definitions) against the external business protocol that the participant is committed to provide (i.e. abstract process definitions) [91]. Another property of interest is to verify that two abstract processes are equivalent in their behaviour [106].

Verification and Performance Analysis require quite advanced analysis techniques. Regarding the work described in this thesis, we are only interested in verification, and will, thus, concentrate our review on this property. A good introduction to Performance Analysis techniques is [118]. For verification, in this work, we are concerned with checking that a workflow satisfies general properties such as freedom from locking; and, also, that we provide a facility for checking model-specific properties, or constraints. Regarding the first of these, we note that an important property of a WF-net is *soundness.* This property ensures that a WF-net will not be susceptible to locking, will complete properly, and does not have dead tasks.

From [120], a WF-net N, with initial marking i (there is a single token at place i), is sound iff:

- For every marking M, reachable from marking i, there exists a firing sequence leading from M to o (there is a single token at place o). This says that there is always the possibility of completing the workflow instance, i.e. an **option to complete.**
- o is the only marking reachable from i with at least one token in place o. That is, once a token appears in o, then all others places must be empty. This says that completion is **proper completion.**
- There are no dead transitions in N, given i as an initial marking. That is, for all transitions t in N, there exists a marking M, reachable from i, where t is able to fire. This says that there are **no dead tasks** (i.e. unused tasks) in the workflow instance.

The model-specific properties that we seek to verify can be split into two classes (or may be composites of properties that fall into these two classes): *safety* and *liveness* (where liveness in the sense now described is different from that used in the context of Petri nets). From [59], safety properties specify occurrences which should never happen, and liveness properties specify occurrences that should eventually happen. In *Linear Temporal Logic*, $\Box p$ is a safety property, and $\Diamond p$ is a liveness property. These classes may be further subdivided. For example, $\Diamond p$ and $p \rightarrow \Diamond q$ are both liveness properties, but the first may also be classified as *guarantee* and the second as *response.* Typically a temporal logic, such as LTL, CTL, or CTL* [38, 61] will be used to specify safety and liveness properties.

We now review a number of contributions that look at the verification of WS-BPEL compositions, as we have an interest in supporting this. These may be differentiated according to their respective foci of interest:

- Koshkina and van Breugel are concerned with verifying the integrity of individual compositions.
- Nakajima / Fu and colleagues are concerned with verifying the interactions between compositions.
- Foster and colleagues are concerned with verifying that an abstract WS-BPEL specification complies with a conversation specification represented using Message Sequence Charts (MSCs).

Koshkina and van Breugel introduce, in [65, 66], a CCS-like process algebra called BPE-calculus, which is capable of modelling a subset of activities of WS-BPEL. Given definitions of the syntax and semantics of the BPE-calculus, the *Process Algebra Compiler* [113] is used to generate an extension module to the Concurrency Workbench for the New Century (CWB-NC) [11] to allow for verification of BPE-processes. Verification options that are supported by CWB-NC include model checking and equivalence checking (as exemplified in Chapter Five).

Nakajima [83] presents a translation of WSFL [69] (a graph-structured predecessor of WS-BPEL) into Promela, which is the input language of the SPIN [59] model checker. SPIN may then be used to verify properties of WSFL compositions. [84] builds on the previous work in [83], in order to support the verification of WS-BPEL compositions, which are similarly represented in Promela, and verified using SPIN. Notably, in [84], the translation of WS-BPEL to Promela is divided into two phases. First WS-BPEL compositions are translated to representations based on finite automata. Then, specifications in the latter representation are translated to Promela. This decoupling allows for the support of alternative composition languages and for the use of different model-checking tools. The work of Fu and colleagues [48, 47] is similar to that of [84] in using an intermediary representation and target output language Promela, for use with SPIN. The intermediary representation formalism, used in Fu's work, is essentially the same as that used by Nakajima. Both works are also similarly concerned with the verification of interacting compositions.

Foster and colleagues [45, 44] propose a translation of a subset of WS-BPEL to *FSP*, which is a process calculus that can be used to concisely describe, and reason about, concurrent programs. The *Labelled Transition System Analyzer* (LTSA) can be used to analyse, and verify properties of, FSP specifications. In their work, they are able to check whether a WS-BPEL composition satisfies a behavioural specification captured by Message Sequence Charts (MSCs). The WS-BPEL and MSC specifications are translated to FSP. LTSA is then used to check compatibility between them.

In our work, we are concerned with verification of individual compositions only. In this sense, our work aligns more with Koshkina and van Breugel than the others. If we concern ourselves with the operation of a single end-point, that is, a single WS-BPEL composition, rather than the interaction between multiple compositions, verification of *the control perspective* of WS-BPEL is decidable. In this case, we abstract away from message queues. It is decidable because verification can be divided into a number of runs, according to link boundary-crossing restrictions (e.g. the contained activity within a <while> activity can be verified separately, and the <while> simply replaced by an <empty> activity), where the verification state space of each of these runs is necessarily finite – as there is no scope for infinite behaviour.

Having presented this review of related work, we now proceed with the definition of our information view meta-model for traditional workflow, called **Liesbet**. This will be used to capture the real essence of workflow, serving as a point of reference for the computational view formalisations to be presented in later chapters.

Chapter 3

Liesbet Metamodel

In this chapter, we define a meta-model for workflow called Liesbet, which constitutes an *information view* abstraction of, or ontology for, workflow. In defining Liesbet, we have sought to understand the *true nature* of workflow, and thus the fundamental concepts that need to be represented. We are then able to use this information view of workflow as a point of reference for computational view formalisations of workflow. The representational requirements for Liesbet have been sourced from the need to be able to represent the YAWL workflow patterns, as well as the control flow aspects of business process languages, such as WS-BPEL [87].

We also present the definition of additional *intended semantics* for Liesbet, which prescribe further constraints regarding the evolution of Liesbet models. We consider it to be appropriate that internal behaviour is prioritised over external behaviour in the enactment of a workflow model, and that the effects of external behaviour on the internal evolution of the model is realised atomically rather than allowing it to be interleaved with other (unrelated) internal behaviour. We provide examples that clarify this matter.

We take our first step towards greater flexibility in workflow models through the proposal of *Synchronisation Rules.* In contrast to the view of flexible workflow that is principally espoused in this thesis (i.e. abstract model + policies for refinement), the appropriate slogan in this instance is more *Flexible Workflow = Concrete Model + Policies for Constraint.* That is to say, the initial model is fully-specified and the policies (i.e. synchronisation rules) *constrain* enactment. The model may contain many possible enactment paths (in contrast to traditional workflow, where typically only one will turn out to be possible). Which of the multiple paths is chosen is constrained by the policies.

We present a reduced set of patterns with which (we show) all patterns may be represented. This is a useful result as it enables us to propose the true nature of workflow to be this reduced set. We propose the reduction at the level of the Liesbet meta-model. That is, we define equivalences for the remaining constructs as definitions which make use only of constructs from the reduced set. These equivalences are argued (and shown) to be sound in Chapter Six.

We also show how Liesbet captures all of the YAWL patterns, as well as describing its support for modelling the control flow perspective of WS-BPEL, in order to usefully facilitate verification of WS-BPEL compositions.

The layout of this chapter follows the presented description.

3.1 Liesbet: An Information View Meta-model for Workflow

3.1.1 Liesbet Fundamentals

We have defined the Liesbet meta-model as an information view ontology for workflow. Its constructs are as follows.

- Activity – Act.
- Synchronisation types – Go and Stop.
- Sequence and Unordered Sequence types – Seq, SeqCancel and UnorderedSeq.
- Parallel, and Priority Parallel – Par and PriPar.
- Exclusive Choice with and without default – DefaultChoice and Choice.
- Deferred Choice – DeferredChoice.
- Trivial Completion – Empty.
- Free Choice – FreeChoice.
- Multiple Choice – MultiChoice.
- Multiple Merge – Multimerge.
- Discriminator m from n – Discriminator.
- Multiple-Instance Activities – Multi*.
- Cancel Activity – CancelActivity.
- Cancel Case – Exit.

We define a syntax for Liesbet for the purposes of presenting it here. This is called Liesbet *Easy Syntax*. Its use is fairly intuitive. It is given a formal characterisation in later chapters using CCS/PCCS and the Situation Calculus. For the implementation of a verification and enactment engine for Liesbet, we use a persistence framework (see Section 10.1) with which an XML-based *serialisation syntax* is defined. The definition of Liesbet Easy Syntax in Extended BNF (EBNF) [90] is presented in Figure 3.1.

At this point, we introduce some terminology. A *customised activity type* is a customisation of a Liesbet meta-model construct when used in the specification of a Liesbet workflow model. In contrast, the term *generic activity type* is used synonymously with meta-model construct. For example, in the Liesbet model Seq(A,B), the Seq is a "sequence" *generic* activity type which is *customised* to mean a sequence that contains two activity types, A and B.

A *basic activity type*, defined using the Liesbet meta-model construct Act, corresponds to a self-contained piece of work, where *conceptually* we would defer to the environment to inform us when the work of the activity type has finished. Instances of basic types may be completed, or cancelled.

```
<Liesbet_Model> ::= <Activity_Type> {<Activity_Type_Def>} {<ISA_Decl>}
<Activity_Type_Def> ::= <Activity_Type_Name> = <Activity_Type>
<Activity_Type> ::= <Activity>(<ActConds>)
<ISA_Decl> ::= ctype(<Activity_Type_Name>) ISA ctype(<Activity_Type_Name>)
<Activity_Type_Name> ::= α | β | γ | ...
<Activity> ::= <Activity_Type_Name> | Act | <StructAct>
<ActConds> ::= [join(<GuardAct>)] [,] [trans(<GuardAct>)] [,] [ctype(<Activity_Type_Name)]
<StructAct> ::= <SyncActs> | <ParSeq> | <Choices> | <Merges> |
                <CancelActs> | <MultiActs> | Empty | Exit | FreeChoice
<SyncActs> ::= Go(<GoQuery>) | Go(<StopQuery>, <GoQuery>) | Stop(<StopQuery>) |
               Stop(<StopQuery>, <GoQuery>)
<ParSeq> ::= PriPar(<ExecActs>) | Par(<ExecActs>) |
             Seq(<ExecActs>) | SeqCancel(<ExecActs>) | UnorderedSeq(<ExecActs>)
<Choices> ::= DefaultChoice(<GuardContActs>, <ContAct>) | Choice(<GuardContActs>) |
              MultiChoice(<GuardContActs>) | DeferredChoice(<ContActs>)
<Merges> ::= Discriminator(<m>)(<GuardActs>, <ContAct>) | Multimerge(<GuardActs>, <ContActs>)
<CancelActs> ::= CancelActivity(<Activity_Type_Name>) |
                 CancelActivity(<Activity_Type_Name> in <Activity_Type_Name>)
<MultiActs> ::= MultiLimit(<n>)(<ExecAct>) | MultiLimitSeq(<n>)(<ExecAct>) |
                Multi(<ExecAct>) | MultiSeq(<ExecAct>)

<GuardContActs> ::= <GuardAct>, <ContAct> {; <GuardAct>, <ContAct>}
<GuardActs> ::= <GuardAct> {, <GuardAct>}
<ContActs> ::= <ContAct> {, <ContAct>}
<ExecActs> ::= <ExecAct> {, <ExecAct>}

<GuardAct> ::= <Activity_Type_Name> | <Activity_Type>
<ContAct> ::= <Activity_Type_Name> | <Activity_Type>
<ExecAct> ::= <Activity_Type_Name> | <Activity_Type>

<n> ::= 1 | 2 | ...    <m> ::= <n>

<GoQuery> ::= <Query>
<StopQuery> ::= <Query>
<Query> ::= <Query>|...|<Query> | <Query>+...+<Query> | ¬<Query> | <QueryOnAct> | True | False
<QueryOnAct> ::= <StateQualification>_<QueryOnActStripped>
<StateQualification> ::= Completed | Cancelled | Finished | Running | Initial
<QueryOnStrippedAct> ::= _Act(<Activity_Type_Name>) | _All(<Activity_Type_Name>) |
                         _Act(<Activity_Type_Name> in <Activity_Type_Name>) |
                         _Act(<Activity_Type_Name> dist in <Activity_Type_Name>) |
                         _All(<Activity_Type_Name> in <Activity_Type_Name>)
```

Figure 3.1: EBNF Definition of Liesbet Easy Syntax

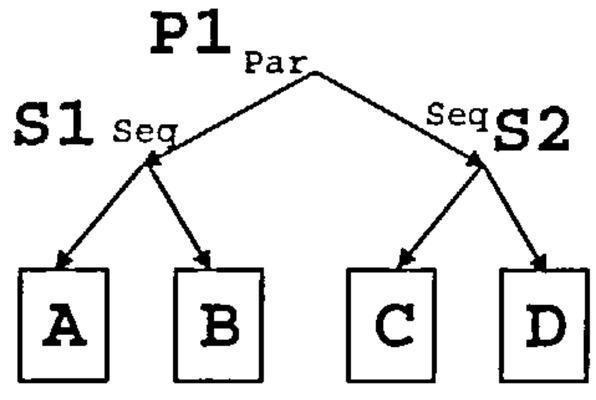


Figure 3.2: Simple Workflow Model

In contrast, *structured activity types*, defined using any other Liesbet construct, exist mainly for the purpose of marshalling instances of basic activity types (i.e. Act types), where the enactment of instances of these other constructs (e.g., Par and Seq) is handled wholly within the realms of the workflow engine.

A workflow model will consist of a number of *instances* of customised activity types. It is through activity instances that work is realised in the enactment of a workflow model. If an activity type is instantiated twice in a model, the work associated with that type will be carried out twice.

Basic activity types defined in "Easy syntax" may either be simply defined *in situ*, or in a separate definition which is then referred to when instantiating the activity type elsewhere. For basic activities, defining them *in situ* is done simply by referring to them, e.g. A, or A(join(...), ...). Defining them separately would be done thus: A = Act, or A = Act(join(...), ...). Here, A is the customised type name and Act is the (only) generic type for basic activity types. join(...) is one of three optional attributes that may be attached, in parentheses, to the right-hand side of a (customised) activity type. The others are: trans(...) and ctype(...). The latter is not applicable to basic activity types. These attributes will be elaborated as we go along.

Structured activity types which are defined *in situ* derive their customised type name from the use of such a ctype(...) qualifier. An example might be Seq(C,D)(ctype(S2), ...), where S2 is the customised type name. Structured activity types can also be defined separately and assigned a name, e.g. S1 = Seq(A,B). Here, S1 is the customised type name.

A hierarchy of type names may be specified using ISA relations. For example, we may have two sequences, S1 and S2, for which it is sometimes convenient to differentiate between them, and other times count them as the same type of sequence, with (customised) type name, S, say. In this situation, we may assert that: ctype(S1) ISA ctype(S) and ctype(S2) ISA ctype(S). Type hierarchies must be acyclic.

Activity types that are defined separately and not *in situ* are called *defined types.* Consider the simple Liesbet model, depicted in Figure 3.2, which will be used for illustrative purposes throughout this thesis. The model is a parallel composition (P1) of two sequences (S1 and S2), each consisting of two atomic activities (A and B, and C and D, respectively).

This may be specified, using Liesbet Easy Syntax, as follows.

```
Par(S1, Seq(C,D)(ctype(S2))(ctype(P1))
S1 = Seq(A,B)
```

Here, A, B, C and D are *in situ* definitions of basic activity types. We can tell this as they are not defined types. The second argument of the Par is a structured activity type defined *in situ*,

with specified customised type name: S2. In contrast, the first argument, S1, is a defined type.

The definition of a workflow model will include just one defined type that is unnamed. This is taken to be the top-level activity of the workflow model. A workflow model is a hierarchical structure with this activity at its root. In the example, Par(S1,Seq(C,D)) is the top-level workflow activity type. The ctype annotation specifies that its type name is P1.

3.1.2 Finite State Machine for Activity Instances

The following Finite State Machine (FSM) is defined for the operation of an activity instance. An activity instance may be in one of four states – Initial, Running, Cancelled or Completed. We also consider an activity instance to be *finished*, if it is in a Cancelled or Completed state.

```
Initial -execute-> Running
Initial -cancel-> Cancelled

Running -complete-> Completed
Running -cancel-> Cancelled
```

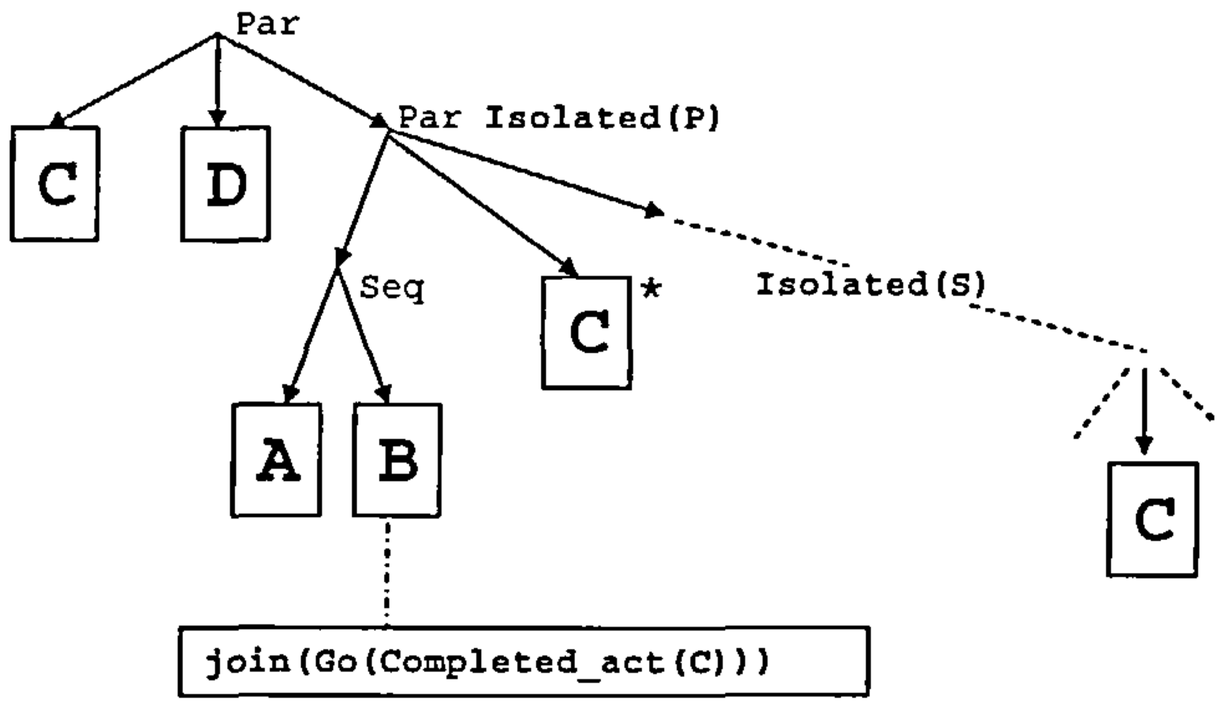
- An activity instance begins life in the Initial state. At some point, the parent of the activity instance will initiate execution of the instance. The instance will be moved into the Running state, by virtue of the execute action.

- When the work of the instance is done, it is moved to the Completed state, by means of the complete action.

- From the Initial and Running states, the instance may be moved into the Cancelled state, by means of the cancel action. This will have the effect of not only immediately cancelling the activity instance itself, but also all of its descendants, in a single, atomic action.

  Cancellation of an activity may happen because of the execution of a CancelActivity instance (Section 3.1.16), because of a failed join condition (Section 3.1.7), or because of dead-path elimination. An activity type may specify a join condition, which serves as a pre-requisite for the execution of the pertaining activity. If the join condition fails, the activity is cancelled. Dead-path elimination [70] is performed in workflow model enactment when it is identified that an activity instance will never be executed. This happens, for instance, when executing a Choice activity instance. Those continuation activity instances within the Choice instance that correspond to unselected branches are moved to the Cancelled state.

Note that, for most workflow modelling scenarios, the lifecycle of a basic instance is adequately captured by the sequence of states: Initial→Running→Completed. In some scenarios, such as those outlined in [21], it may be appropriate to model an additional, intermediate state between Initial and Running. This would capture the notion that an activity has been *enabled but is not yet running.* This may simply be modelled, in Liesbet, by inserting a dummy activity Init, say, into a SeqCancel type (see later), thus: SeqCancel(Init, X), where X is the activity type constituting the work to be done. The dummy activity Init completing would *conceptually* signify the activity X moving to this intermediate state.



The join condition on activity type B will have a visibility horizon that is restricted to the isolated scope P and its descendants, but not including the isolated scope S and its descendants. The only candidate instance of activity type C for the query in the join condition of B is thus the instance marked *.

Figure 3.3: Isolated Scopes in Operation

3.1.3 Activity Visibility Horizons

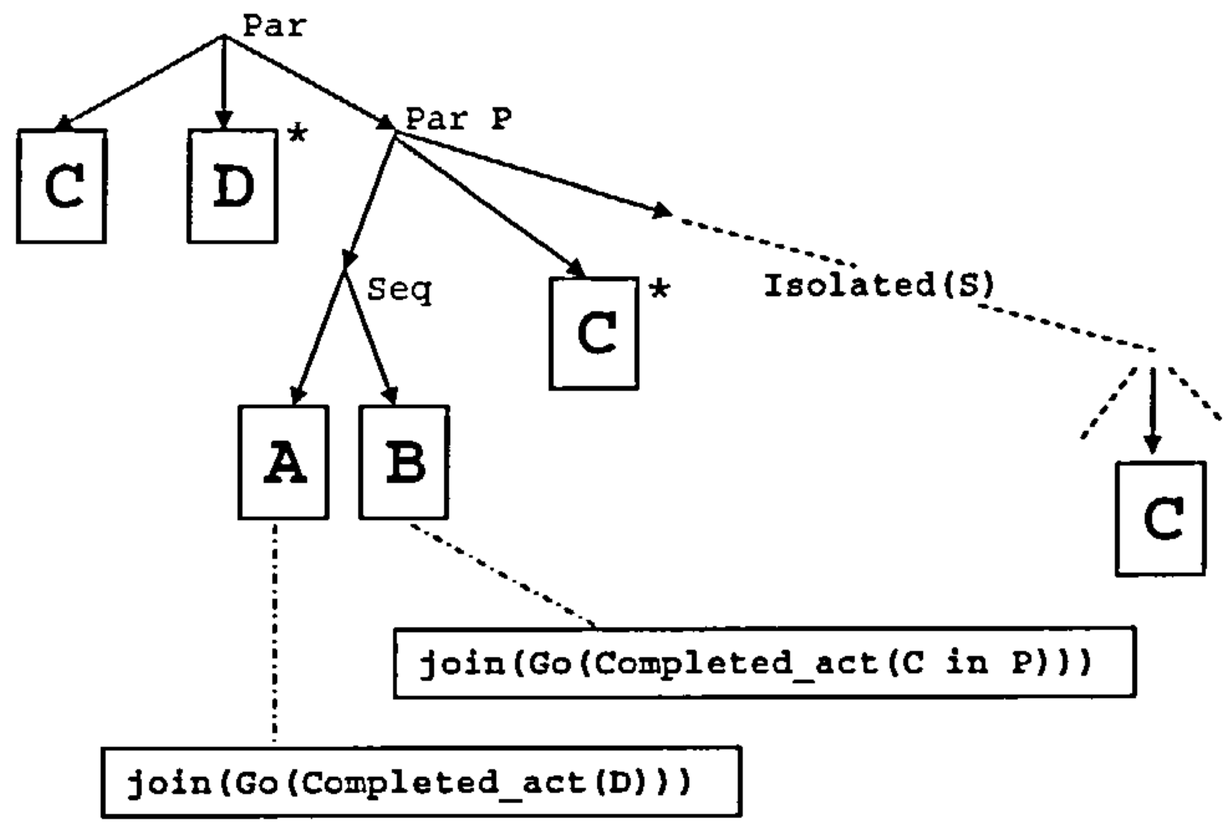
As explained in Sections 3.1.4 (below), instances (of certain activity types) may query the state of other activity instances, in order to synchronise their execution against these instances. However, since the enactment of a workflow model may create multiple instances of the same activity type, there is potential ambiguity about which specific instance is referred to in the query. In the example shown in Figure 3.3, the join condition on activity B queries the state of activity C of which there are three separate instances. Liesbet provides several methods for disambiguating such references, of which the *isolated scope* declaration is the most fundamental.

Any activity may be marked as an *isolated scope*. In Easy Syntax this is achieved by encapsulating the definition of an activity type in the container Isolated. In the example below, both activity types A and B are isolated scopes but C is not. The scope of an activity type is not isolated, by default.

```
Par(Isolated(A), B)
A = ...
B = Isolated(...)
C = Seq(...)
```

This has the effect of creating a *visibility horizon* on the workflow state for activity instances that exist within an instance of the isolated scopes A and B.

When an instance i exists within the scope of another activity instance which is isolated, the instance i can only query the state of activity instances that are descendants of the isolated scope instance that is the most immediate ancestor of i, and this isolated scope instance itself. Moreover, if any of these descendant instances more immediately fall within the scope of a different isolated scope instance, then these particular instances will not be visible to the querying instance i. The visibility horizon for a querying instance is thus the sub-tree extending from its (immediate) ancestor isolated scope instance, from which are pruned any sub-trees extending from further isolated scope instances (as is demonstrated in the figure).



Since P is not an isolated scope in this example, the visibility horizon for the join condition on activity type A extends beyond P; and thus includes the instance of type D marked *. For the join condition on activity type B, its visibility horizon is determined according to type P, which is specified as a reference type. The horizon, with respect to type C, is thus the same as previously, in Figure 3.3.

Figure 3.4: Reference Types in Operation

There is another way of specifying a visibility horizon, for a querying instance; a *reference type* can be used to set the visibility horizon for a query on workflow state. There are two sorts of reference types, "plain" and "distinct". For either sort, the idea is that the *target instances* of a query (i.e. its visibility horizon) are limited to those which are descendant instances of an instance of the specified reference type – the reference instance. Furthermore, the querying instance must be a descendant of the same reference instance. Thus, the reference instance is a common ancestor to both querying and target instances; and acts as a scoping instance, enforcing a visibility horizon for the query.

The use of a *reference type* is similar to that of an isolated scope, in that it is used to place a limit on the instances that comprise a visibility horizon. A crucial difference, however, is that, in contrast to the use of isolated scopes, we may specify within individual queries (of which a querying instance may use several) what the visibility horizon for the *query* should be. That is, multiple queries may be made by a single querying instance, all with different visibility horizons. As a result, we can set a much finer granularity for the visibility of certain queries, rather than setting a universal visibility horizon for a whole tree of querying instances.

For example, we may wish one query to be referenced according to a particular activity instance, but if we made that reference instance an isolated scope, it may undesirably hinder the visibility horizon of other querying instances that would exist within the isolated scope instance. However, through using reference queries, we avoid such a problem.

Isolated scopes are used when we want to create a visibility horizon that is appropriate for all descendant instances of that scope instance (which do not have a more immediate parent instance that is also isolated). It is also necessary that marking an instance as isolated does not inappropriately prevent instances, existing outside the sub-tree rooted at the instance, from querying the state of instances within the sub-tree. Such pruning does not occur for scopes which are not marked as isolated.

Queries with reference types are useful when we want to achieve a finer level of granularity to the visibility horizons of individual queries. A workflow model may arbitrarily use a mixture of isolated scopes, and reference queries, notwithstanding, the possibility of redundancy in certain combinations of such scopes and queries. Figure 3.4 shows an example of using queries with reference types.

We now elaborate the distinction between *plain* and *distinct* reference types. Queries that make use of distinct reference types are the same as those using plain reference types, in the sense that the visibility horizon of a query is limited to the ancestor instance of the reference type; *but* there is the added criterion that the particular target instance, that satisfies a query, can not have been used before to satisfy distinct queries made with respect to the same reference instance. We ensure this by marking the reference instance with the instances that have been used, thus far, in satisfying queries within its scope. This allows us yet further granularity in satisfying queries, and ensures that multiple querying instances that are satisfied according to a common query, such as join condition instances in a `Multi` type – see Section 4.2 for an example, are satisfied by *distinct* target instances.

3.1.4 `Go` and `Stop` Synchronisation Activity Types

The synchronisation activity types of `Liesbet` represent *synchronisation points* in the workflow model, i.e. their completion or cancellation is blocked until some query on current workflow state is satisfied. An example is `Go(StopQuery, GoQuery)` in which `StopQuery` and `GoQuery` are queries on workflow state. Here, there is a race between which of these queries is satisfied first, which ultimately determines whether the synchronisation activity itself completes successfully or not.

Easy Syntax

```
Stop(StopQuery, GoQuery)
Stop(StopQuery)
Go(StopQuery, GoQuery)
Go(GoQuery)
```

A `StopQuery` or `GoQuery` query is a *blocking* query on current workflow state that must be satisfied. That is, a query blocks until it is satisfied. A query is any boolean compound (using "|" for conjunction, "+" for disjunction and "¬" for negation) of the following (where `ctype` is a customised type name):

- Simple Queries
  - `Completed_act(qtype)` – This query is satisfied if and only if an instance of the activity type `qtype`, within the visibility horizon of the querying instance, has completed.
  - `Completed_all(qtype)` – This query is satisfied if and only if all extant instances of the activity type `qtype`, within the visibility horizon of the querying instance, have completed.
- Reference Queries

- `Completed_act(qtype in rtype)` – This query is the same as `Completed_act(qtype)` except that it specifies a *plain reference type*, `rtype`, in order to (further) constrain the visibility horizon for the query.
- `Completed_act(qtype dist in dtype)` – This query is the same as `Completed_act(qtype)` except that it specifies a *distinct reference type*, `dtype`, in order to (further) constrain the visibility horizon for the query.
- `Completed_all(qtype in rtype)` – This query is a combination of `Completed_all(qtype)` and `Completed_act(qtype in rtype)`.

We may also write `True` for the query that is trivially satisfied, and `False` for the query that can never be satisfied (i.e. it forever blocks).

Queries can also be made to ascertain the existence of activity instances in the `Initial`, `Running`, or `Cancelled` states, as well as *finished* instances (those in `Completed` or `Cancelled` states). To use such queries, the keyword `Completed` is replaced with the keywords: `Initial`, `Running`, `Cancelled` or `Finished`, as appropriate.

In the following example, the query is satisfied if either an instance of activity type `A` or `B` has completed, and an instance of activity type `C` has completed.

```
( Completed_act(A) + Completed_act(B) ) | Completed_act(C) )
```

Informal Operational Semantics

When an instance of the activity type `Go(StopQuery, GoQuery)` is running, and `StopQuery` is satisfied before `GoQuery`, then the synchronisation activity instance goes to `Cancelled`. If `GoQuery` is satisfied, and `StopQuery` is not satisfied beforehand, then the synchronisation activity instance goes to `Completed`. While neither query is satisfied, the instance remains in the `Running` state. There is a priority at work here in that, whenever we try to progress a `Go` instance, `GoQuery` is evaluated ahead of `StopQuery`. To effect the opposite, the author may use a `Stop` activity type, where the `StopQuery` is evaluated first. Thus, whether a `Go` or a `Stop` type is used in a particular circumstance depends on the appropriate priority regarding which of the queries is evaluated first.

An instance of the activity type `Go(GoQuery)` will remain in the `Running` state until the `GoQuery` query is satisfied, whereupon it will move to `Completed`. `Go(GoQuery)` is thus equivalent in behaviour to `Go(False, GoQuery)`. Similarly, an instance of `Stop(StopQuery)` will remain in the `Running` state until the `StopQuery` query is satisfied, whereupon it will move to `Cancelled`. `Stop(StopQuery)` is thus equivalent in behaviour to `Stop(StopQuery, False)`.

3.1.5 Seq and SeqCancel – Sequence and UnorderedSeq – Unordered Sequence

The `Seq/SeqCancel` and `UnorderedSeq` `Liesbet` constructs are a direct facilitation of two of the `YAWL` workflow patterns, viz. *Sequence* (#1) and *Interleaved Parallel Routing* (IPR) (#17), respectively. [125, 64] characterises *Sequence* as a pattern where "[a]n activity in a workflow process is enabled after the completion of another activity in the same process". For IPR [125, 64], "[a] set of activities is executed in an arbitrary order: Each activity in the set is executed, the order is decided at run-time, and no two activities are executed at the same moment".

Easy Syntax

```
Seq(Act1, ..., Actn)
SeqCancel(Act1, ..., Actn)
UnorderedSeq(Act1, ..., Actn)
```

Informal Operational Semantics

When a sequence (Seq/SeqCancel) instance is running, it executes each constituent activity in the order specified, waiting for each to get to a finished (Completed or Cancelled) state. For Seq, if a constituent activity is cancelled, then the sequence continues as normal. For SeqCancel, the sequence is immediately cancelled. When the last constituent activity finishes, Seq goes to Completed, and SeqCancel goes to Completed if the last constituent activity completed successfully and to Cancelled otherwise.

For UnorderedSeq, the child instances contained therein may be executed in any order, but not concurrently. When an instance of such a type is set Running, one of its child instances is non-deterministically put into a Running state. The choice could be made by the workflow engine, or could be made by the environment (in a deferred choice sense, see later). When the chosen instance eventually moves to a finished state (Cancelled, or Completed), another child instance is selected and put into the Running state. When all child instances have finished, the UnorderedSeq instance is completed.

3.1.6 Par – Parallel

The Par Liesbet construct is a direct facilitation of the *Parallel Split* YAWL workflow pattern, which is [125, 64] "[a] point in the workflow process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order".

We also support a *Priority Parallel* (PriPar) construct. PriPar allows arbitrarily complex structured activities to be specified as running in parallel, but the progression of child instances of a PriPar occurs according to a total priority ordering.

Easy Syntax

```
Par(Act1, ..., Actn)
PriPar(Act1, ..., Actn)
```

Informal Operational Semantics

When the parallel instance (Par) is running, it starts the execution of each child instance in parallel. Once all have reached a finished state (Completed or Cancelled), the parallel instance goes to Completed. Note that cancelling child instances does not cancel the Par activity.

For PriPar, progression of $\texttt{Act}_1$ occurs at the highest level of priority, then $\texttt{Act}_2$, and so on, until Actn, which has the lowest priority. In terms of what is pushed to agents by the WfMS[1],

[1] A standard deployment configuration, which uses a WfMS, is where agents are offered items of work in work lists. An agent may accept an item of their choice from the work list, while it remains in their work list. Once accepted, the agent marks the work item as finished at some later time.

a basic instance which is a descendant of, or is, $\texttt{Act}_i$, say, can only be offered, or continue to be offered, if a basic instance which is a descendant of, or is, $\texttt{Act}_j$, where $j < i$, is not offered. If it becomes possible to offer a basic instance "from" $\texttt{Act}_j$, at a particular point in time, then all basic instances pertaining to children of the PriPar instance with a lower priority than j must be blocked (i.e. temporarily withdrawn from agent work lists). The PriPar construct captures the notion that certain work items will need to be completed more expediently than others. Its utility is contingent on appropriate role assignment.

3.1.7 Activity Join and Transition Conditions

An activity definition in Liesbet may optionally specify a *join condition* and/or a *transition condition* for the activity type.

A join condition may be any activity type, although it would rarely be anything but a synchronisation activity type (Go or Stop). They are used to specify conditions under which execution of an activity may occur. When execution of an activity instance is initiated, the join condition, if specified, is evaluated. Once a join condition returns a result – a condition may block for some time, the pertaining activity is executed (moves to Running) if the join condition is satisfied, and cancelled otherwise.

A transition condition for an activity A is used to specify a number of activities that must be executed after the main work item constituting A. Notably, these activities are considered to be part of A, i.e. A is not considered to have completed until these activities have themselves finished. For example, a transition condition may specify one or more synchronisation conditions that must be satisfied before the pertaining activity may complete.

Activity types that are used as join conditions may not themselves specify join or transition conditions. The same applies to transition conditions. Moreover, it is not permitted to specify join or transition conditions for the root activity of a Liesbet workflow model.

Easy Syntax

Join and transition conditions, when specified, sit to the right of an activity type definition. They are given in a separate set of parentheses, and enclosed in the containers join and trans. There are thus three possible forms (besides an activity definition without join and transition conditions).

```
A(join(AJoin))
AJoin = ...

A(trans(ATrans))
ATrans = ...

A(join(AJoin),trans(ATrans))
AJoin = ...
ATrans = ...
```

Informal Operational Semantics

An activity type with a join condition should be considered as being equivalent to a SeqCancel activity type containing (in order) the join condition activity type and the actual activity type. This

realises the desired behaviour, namely: that if the join condition does not complete successfully, the activity instance that it is attached to is not executed. If a transition condition is specified, then the join condition (if any) and the actual activity type are run first, followed by the transition condition. Even if the join condition or the instance of the actual activity type get cancelled, the transition condition will still be evaluated.

In summary, the following mappings should be applied, at the level of the meta-model (that is, at the information view). Note that as there exist mappings for join and transition conditions at the level of the meta-model, they do not necessitate specific treatment at the computational view[2].

- `A(join(AJoin),trans(ATrans))` maps to `Seq(SeqCancel(AJoin, A), ATrans)`.
- `A(join(AJoin))` maps to `SeqCancel(AJoin, A)`.
- `A(trans(ATrans))` maps to `Seq(A, ATrans)`.

3.1.8 DefaultChoice, Choice – Exclusive Choice With and Without Default

The `DefaultChoice/Choice` `Liesbet` constructs are a direct facilitation of the `YAWL` workflow pattern *Exclusive Choice*, which is [125, 64] "[a] point in the workflow model where, based on a decision or workflow control data, one of several branches is chosen".

Easy Syntax

```
DefaultChoice(Guard1, ContAct1; ... ; Guardn, ContActn; ContActd)
Choice(Guard1, ContAct1; ... ; Guardn, ContActn)
```

Informal Operational Semantics

Each $\texttt{Guard}_i$ is a *guard* activity type, and each $\texttt{ContAct}_i$ a *continuation* activity type. A guard will usually be a synchronisation activity type (Section 3.1.4), although it could actually be any activity type. For example, `Empty`, which is the basic activity type that trivially completes (see Section 3.1.11), can be used to effect a non-deterministic choice.

The first guard instance that goes to `Completed` initiates its corresponding continuation instance. All other continuation instances go to `Cancelled`. In the case of `DefaultChoice`, if all of the $\texttt{Guard}_i$ activities go to `Cancelled`, then an instance of the default continuation activity type, $\texttt{ContAct}_d$, is executed. In the case of `Choice`, which has no default activity type, the `Choice` will itself go to `Cancelled`. The `DefaultChoice/Choice` instance completes once the executed continuation instance has finished.

3.1.9 MultiChoice – Multiple Choice

The `MultiChoice` `Liesbet` construct is a direct facilitation of the `YAWL` workflow pattern *Multiple Choice*, which is [125, 64] "[a] point in the workflow model where, based on a decision or workflow

[2]For the most part, although many `Liesbet` constructs (as discussed in Section 3.4) may be given a characterisation at the information view (in terms of a minimal set of `Liesbet` constructs), we elect to characterise them at the computational view (as discussed in later chapters). This enables us to discuss equivalences between characterisations. For simplicity's sake, join and transition conditions and `DeferredChoice` are exceptions to this convention.

control data, a number of branches are chosen".

Easy Syntax

```
MultiChoice(Guard1, ContAct1; ... ; Guardn, ContActn)
```

Informal Operational Semantics

`MultiChoice` is similar to `Choice`, except that there is no race between guard instances to complete first. For `MultiChoice`, those guard instances that complete successfully have their corresponding continuation instances executed. Those that go to cancelled have their corresponding instances cancelled.

3.1.10 `DeferredChoice` – Deferred Choice

The `DeferredChoice` `Liesbet` construct is a direct facilitation of the *Deferred Choice* `YAWL` workflow pattern, which is [125, 64] "[a] point in the workflow process where one of several branches is chosen. In contrast to the XOR-split [i.e. *Exclusive Choice*], the choice is not made explicitly (e.g. based on data or a decision) but several alternatives are offered to the environment ... This means that once the environment activates one of the branches the other alternative branches are withdrawn. It is important to note that the choice is delayed until the processing in one of the alternative branches is actually started, i.e. the moment of choice is as late as possible."

Easy Syntax

```
DeferredChoice(ContAct1, ..., ContActn)
```

Informal Operational Semantics

The conceptual meaning of the `DeferredChoice` construct is that of an exclusive choice made *by the environment* between executing instances of continuation activity types $\texttt{ContAct}_1$, ..., $\texttt{ContAct}_n$. A `DeferredChoice` instance goes to `Completed` when the chosen continuation instance has finished. For simplicity, we model the `DeferredChoice` constructs at the information view, as presented in Section 3.4.

3.1.11 `Empty`

Do nothing but trivially complete! Useful, for example, for an empty default branch in a `DefaultChoice` activity.

Easy Syntax

```
Empty
```

3.1.12 `FreeChoice`

Non-deterministically complete or cancel.

Easy Syntax

```
FreeChoice
```

3.1.13 Multimerge – Multiple Merge

The Multimerge Liesbet construct is a direct facilitation of the *Multiple Merge* YAWL workflow pattern, which is [125, 64] "[a] point in a workflow process where two or more branches reconverge without synchronisation. If more than one branch gets activated, possibly concurrently, the activity following the merge is started for every activation of every incoming branch".

[125, 64] determines that the same continuation activity instance be executed for each path that merges. We can facilitate this, but more flexibly we allow the specification of many different continuation activities.

Easy Syntax

```
Multimerge(Guard1, ..., Guardn; ContAct1, ..., ContActm)
```

Informal Operational Semantics

When running, any of the $\texttt{Guard}_i$ going to Completed will cause an instantiation and execution of one of the continuation activities – the first to be completed initiates $\texttt{ContAct}_1$, the second to complete initiates $\texttt{ContAct}_2$, and so on. Note, however, the number of continuation activities, m, may be less than (or equal to) the number of $\texttt{Guard}_i$ activities, n. Once the m continuation instances have been set running, the remaining guard instances are cancelled. Any $\texttt{Guard}_i$ instances going to cancelled do not result in the execution of a continuation activity.

3.1.14 Discriminator – Discriminator m from n

The Discriminator Liesbet construct is a direct facilitation of the *Discriminator m from n* YAWL workflow pattern, which is [125, 64] "[a] point in a workflow process that waits for m of the incoming branches to complete before activating the subsequent activity".

Easy Syntax

```
Discriminator(m)(Guard1, ..., Guardn; ContAct)
```

Informal Operational Semantics

When the Discriminator is running, it waits until m of the named $\texttt{Guard}_i$ instances have gone to Completed, and then executes an instance of ContAct. Any outstanding guard instances are then cancelled. The discriminator instance goes to Completed when the continuation activity and guard instances have finished. If a sufficient number of guard instances fail for the threshold never to be reached, the discriminator instance is cancelled.

3.1.15 Multi* – Multiple-Instance Activities

Multiple-instance activities enable the creation of multiple instances of the same ExecAct activity. There are four types of multiple-instance activity, which can be classified as four quadrants specified

along two axes. One axis is whether the number of child instances that may be created is limited, or not, and the other axis is whether child instances must execute sequentially, or not. This classification leads to the following types.

Easy Syntax

`MultiLimit(n)(ExecAct(join(ExecActJoin)))`
`Multi(ExecAct(join(ExecActJoin)))`

`MultiLimitSeq(n)(ExecAct(join(ExecActJoin)))`
`MultiSeq(ExecAct(join(ExecActJoin)))`

Informal Operational Semantics

When a multiple-instance activity is set running, an initial instance of `ExecAct` is created. It will necessarily specify a join condition which is set running. The join condition would be *in most cases* an instance of a synchronisation type whose `GoQuery` would only be satisfiable *in a distinct way* from previous instances of the synchronisation type, for the same multiple-instance activity. To this end, it would make use (not necessarily exclusively) of distinct sub-queries in the `GoQuery` part – see Section 3.1.4. This would ensure that the same satisfaction of `ExecActJoin` can not be used to create multiple instances of `ExecAct`.

If the join condition completes successfully, its pertaining `ExecAct` instance is set running. Once this has occurred, for `Multi` and `MultiLimit` types, another instance of `ExecAct` is created, and its join condition is set running. For `MultiSeq` and `MultiLimitSeq` types, we must wait until the instance of `ExecAct` has finished before another is created.

If the `ExecActJoin` instance fails (i.e. goes to `Cancelled`), at any time, the multiple-instance activity will not allow the creation of any more `ExecAct` instances, and goes to `Completed` once all its children have finished.

We now describe the informal semantics for each of the `Liesbet` types introduced above.

`MultiLimit(n)(ExecAct(join(ExecActJoin)))`

The threshold, `n`, specified for `MultiLimit` determines a maximum number of instances of `ExecAct` that may be created. The activity goes to completed when all created instances of `ExecAct` have finished executing.

`Multi(ExecAct(join(ExecActJoin)))`

`Multi` instantiates `ExecAct` instances according to `ExecActJoin`, but there is no limit on the number of instances that may be created.

`MultiLimitSeq(n)(ExecAct(join(ExecActJoin)))`

As `MultiLimit`, but instances of `ExecAct` are executed sequentially.

`MultiSeq(ExecAct(join(ExecActJoin)))`

As `Multi`, but instances of `ExecAct` are executed sequentially.

3.1.16 CancelActivity – Cancel Activity

The CancelActivity Liesbet construct is a direct facilitation of the *Cancel Activity* YAWL workflow pattern [125, 64], where an activity is cancelled.

Easy Syntax

```
CancelActivity(qtype)
CancelActivity(qtype in rtype)
```

Informal Operational Semantics

A CancelActivity instance will cancel all running (i.e. Running) and all possible future running (i.e. Initial) instances of the named activity type, qtype, within its visibility horizon. Optionally, CancelActivity may specify a plain reference type, rtype, see Section 3.1.3, to constrain the visibility horizon.

3.1.17 Exit

The Exit Liesbet construct is a direct facilitation of the *Cancel Case* YAWL workflow pattern [125, 64], where a process instance is removed completely.

Easy Syntax

```
Exit
```

Informal Operational Semantics

An Exit activity instance cancels the root instance of the given workflow instance, which has the effect of cancelling the whole instance.

3.2 Additional Constraints on the Intended Semantics for Liesbet

At this point, it is necessary to augment the definition of the Liesbet meta-model, presented in the previous section, with some further prescriptions regarding the intended semantics of Liesbet.

We consider it to be appropriate that internal behaviour is prioritised over external behaviour, and that the effects of external behaviour on the internal evolution of the model is realised atomically rather than allowing it to be interleaved with other (unrelated) internal behaviour.

- *Internal behaviour is prioritised over external behaviour, that is, the progression of structured instances takes priority over the progression of basic activity instances.*

  Elaborating, a workflow model should be seen, in enactment, as commencing with a number of basic activities, or work items, that are offered by the Workflow Management System (WfMS), for completion by agents[3]. When one of these activities is completed, the WfMS offers a new set of activities to agents. Typically, this list will be an extension of the previous

[3] A WfMS may also support cancellation of basic instances by an agent.

list (with the completed instance withdrawn), where the extension may include some or no new instances. The process then repeats – i.e. completion, then a new offer of instances – until there is no work left to be done. Every time a basic instance is completed, it is appropriate that structured instances within the model (i.e. the marshalling activities) are advanced as far as possible, before any new offer of (basic) activities to (complete) is made. This includes prior to the first offer of basic instances.

- *The effects of external behaviour on the internal evolution of the model is realised atomically rather than allowing it to be interleaved with other (unrelated) internal behaviour.*

  That is, whenever a childless instance (i.e., an instance of childless structured type, such as `FreeChoice`*, or a basic instance) completes, or is cancelled, the effects of this should be* ***atomically*** *propagated as much as possible through the activity instance hierarchy.*

  By this, we mean:

  – If the childless instance completes, then completion should be propagated up the instance tree as far as possible. For every parent instance (starting with the parent of the instance being completed), iff all its other children have finished we may, in turn, mark it as completed.

  – If the childless instance is cancelled, then completion is propagated in the same way, unless the instance is a child of a `SeqCancel` instance, in which case cancellation is propagated upwards until a parent instance is reached which is not a `SeqCancel`, then completion continues to be propagated, according to the description in the previous bullet.

  – Once we reach a parent instance whose other children have not all finished, we advance this instance as much as possible. For example, if it is a `Seq`, we propagate execution down through the instance sub-tree whose root is the next child of the `Seq` to be executed.

  According to these semantics, child-bearing instances are only ever progressed as a side-effect of the initial execution of the workflow instance, and of the subsequent completion/cancellation of childless instances. We consider such side-effects to be a true reflection of the nature of the operation workflow.

  These notions will be crystallised in subsequent chapters.

3.3 Synchronisation Rules

We now introduce the notion of *synchronisation rules*. These are meant as a first step in providing a capability for the specification of *flexible workflow models*. At the start of this chapter, we asserted that the kind of flexibility that we capture with synchronisation rules may neatly be expressed by the slogan: *Flexible Workflow = Concrete Model + Policies for Constraint*. Synchronisation rules capture the policies, constraining the enactment of the workflow model which will have a number of possible enactment paths.

The format for a synchronisation rule is:

```
SyncRule(RType, CondQuery, GoQuery)
```

where:

- RType (*rule type*) specifies the type of instances to which this synchronisation rule pertains.
- CondQuery is (in effect) a filter on states – for states which do not match the CondQuery, instances of the given RType may be advanced *without constraint* (by a single step to the next state).
- GoQuery specifies a query that must be satisfiable in the current state, if the current state satisfies the CondQuery, for any instance of RType (including descendants thereof) to be advanced in the current state.

Both CondQuery and GoQuery are constrained to be boolean compositions of simple queries (i.e. those without reference types), see Section 3.1.4.

There is also a four-argument variant, viz. SyncRule(Ref, RType, CondQuery, GoQuery). In this case, Ref is a reference type which acts as a scope for RType, CondQuery and GoQuery, in much the same way as reference types are used in queries within synchronisation types (see Section 3.1.4). That is, for any instance of RType in a model, we ascertain its ancestral instance of type Ref, which is then used to scope the queries CondQuery and GoQuery. The SitCalc-based characterisation that we afford synchronisation rules is presented in Appendix Section B.2.

To illustrate the utility of synchronisation rules, we present the following example (which is also used in Chapter Eleven to demonstrate verification of a model constrained with a synchronisation rule). Given the workflow model (which we use throughout this thesis for illustrative purposes): Par(Seq(A,B), Seq(C,D)), it may be the case that we have a business rule that says that once activity A has been completed, further progression of activities C and D must be blocked until execution of B completes. Flexibility would come from enabling/disabling this business rule, according to current priorities. A specific example might be where activity A corresponds to a customer returning a complaints form, and B corresponds to the processing of the form. In this case, we may seek to prioritise completion of B over C and D, which may relate to general Customer Relationship Management activities, such as making offers to the customer.

The described business rule would be effected as a synchronisation rule, viz. SyncRule(S2, Completed_act(A), Completed_act(S1)), where S1=Seq(A,B) and S2=Seq(C,D). The rule stipulates that as soon as the (only) instance of A is in the Completed state, descendants of S2 (namely, the instances of C and D), and S2 itself, may not advance until the sequence S1 (containing A and B) has completed.

Synchronisation rules may be used to effect Liesbet's PriPar construct. PriPar(A_0,...,A_n) will effect the running of basic activities within A_j at a higher priority than those in A_i, for $j < i$. For example, in the case of

PriPar(Seq(A,Go(...),B),Seq(C,D))

A and C are both set running; but, to begin with, only A may be completed. However, once A has been completed, and while the Go synchronisation instance is still running (and thus B has not been started), we may complete C, and then, even, complete D.

This would be expressed as a Liesbet model, using a regular Par type, thus:

P=Par(S1,S2), where S1=Seq(A,Go(...),B) and S2=Seq(C,D),

with the model being constrained according to the following synchronisation rule:

SyncRule(S2, ¬Finished_act(S1 in P), ¬Running_act(A in P) ∧¬Running_act(B in P))

The rule says that if the current state is one where S1 has not finished executing, then for S2 to be advanced then A and B cannot be Running. (For all states that do not match the CondQuery filter, i.e. those where S1 has finished, S2 may be advanced).

A workflow author needs to be careful with the use of synchronisation rules, as it may be unclear how they will behave in the context of a particular workflow model. However, these rules have a natural characterisation in our Situation Calculus-based semantics for Liesbet, and consequently are easily incorporated into our verification engine, meaning that workflow models that incorporate their use can straightforwardly be checked for properties such as workflow soundness (see Section 7.1).

3.4 Liesbet Constructs as Abbreviations

Many of the constructs of Liesbet, i.e. those in a set labelled $\texttt{Liesbet}_{abbrev}$, may be cast as abbreviations. This means that they may alternatively be expressed in terms of other Liesbet constructs from a fundamental set of constructs, labelled $\texttt{Liesbet}_{prim}$. This intuition (captured at the information view) is confirmed in Section 6.5, where we argue the following definitions of these abbreviations to be sound. This is an important contribution as it allows us to propose a *fundamental set of primitives as embodying the real essence of workflow.*

$\texttt{Liesbet}_{abbrev}$ consists of: Join/transition conditions, Seq, UnorderedSeq, Choice, DefaultChoice, MultiChoice, DeferredChoice, FreeChoice, Empty, Multimerge, Discriminator, MultiLimit, MultiLimitSeq, MultiSeq and Exit.

$\texttt{Liesbet}_{prim}$, which may be considered to be primitive, or fundamental to the expression of Liesbet models, consists of: Act, SeqCancel, Par, Go, Stop, Multi and CancelActivity.

Mapping $\texttt{Liesbet}_{abbrev}$ to $\texttt{Liesbet}_{prim}$

We now present $\texttt{Liesbet}_{prim}$-based characterisations for each of the constructs in $\texttt{Liesbet}_{abbrev}$. Note that we have used constructs in $\texttt{Liesbet}_{abbrev}$ in some of the following characterisations, such as Choice for the definition of DeferredChoice. However, their use can simply be replaced for the presented $\texttt{Liesbet}_{prim}$-based characterisations, such as that for Choice. We have presented the characterisations, in this way, for simplicity.

In the following, we omit a presentation of the $\texttt{Liesbet}_{prim}$-based characterisations for join and transition conditions, as we have already presented these, in Section 3.1.7.

- S = Seq(A,B,C)

  S = Par(A,B',C')

  B' = SeqCancel(Go(Finished_act(A in S)),B)

  C' = SeqCancel(Go(Finished_act(B in S)),C)

  The execution of activity instance B (i.e. moving B to the Running state) (resp. C) is blocked until A (resp. B) has finished.

- U = UnorderedSeq(A,B,C)

```
U = Par(A',B',C')
A' = SeqCancel(AJoin, A)
B' = SeqCancel(BJoin, B)
C' = SeqCancel(CJoin, C)
AJoin = Go(¬Running_act(B in U) | ¬Running_act(C in U))
BJoin = Go(¬Running_act(A in U) | ¬Running_act(C in U))
CJoin = Go(¬Running_act(A in U) | ¬Running_act(B in U))
```

The execution of any of the activity instances, A, B, or C, is blocked whenever one of its siblings is in the Running state.

- C = DefaultChoice(G1, C1; ...; Gn, Cn; De)

```
C = Par(GC1Cancel, GC1Main,..., GCnCancel, GCnMain, De')
GC1Cancel = SeqCancel(Stop(¬Initial_act(C1 in C) + Cancelled_act(GC1Main in C),
                           ¬Initial_act(C2 in C) + ...  +
                           ¬Initial_act(Cn in C) + ¬Initial_act(De' in C)),
                      CancelActivity(GC1Main in C))

GC1Main = SeqCancel(G1, C1Join, C1)
C1Join = Go(Initial_act(C2 in C) | ...  | Initial_act(Cn in C))
...
De' = SeqCancel(DeJoin, De)
DeJoin = Stop(¬Initial_act(C1 in C) + ...  + ¬Initial_act(Cn in C),
              Cancelled_act(G1 in C) | ...  | Cancelled_act(Gn in C))
```

The execution of any of the continuation instances (not including the default instance) may only occur if its guard has been completed, and none of the other continuation instances have moved from an Initial state. The default instance may be executed iff all of the guard instances have been cancelled.

- C = Choice(G1, C1; ...; Gn, Cn)

The characterisation is the same as that for DefaultChoice, except that we define the De type ourselves, viz.

```
De = CancelActivity(C in C)
```

This has the effect of cancelling the Choice if all of the guard instances get cancelled.

- M = MultiChoice(G1, C1; ...; Gn, Cn)

```
M = Par(SeqCancel(G1, C1), ..., SeqCancel(Gn, Cn))
```

- C = DeferredChoice(C1, ..., Cn)

```
C = Par(Ch1, C1', ..., Chn, Cn')

C1' = SeqCancel(Stop(Completed_act(Ch2) + ...  + Completed_act(Chn),
                     Completed_act(Ch1)), C1)
...
```

The environment signals a choice by completing one of {Ch1, ..., Chn}. Once a choice is made, the corresponding branch is executed and the rest are cancelled, by virtue of their join conditions.

- M = Multimerge(G1, ..., Gn; C1, ..., Cm)

```
M = Par(G1', ..., Gn', C1', ..., Cm')
G1' = Par(G1Canc, G1)
G1Canc = SeqCancel(Stop(Finished_act(G1 in G1'),
                        ¬Initial_act(C1 in M) | ...  | ¬Initial_act(Cn in M)),
                        CancelActivity(G1' in M))
...

C1' = SeqCancel(C1Join, C1)
C2' = SeqCancel(C2Join, C2)
...
C1Join = Go(Finished_act(G1 in M) | ...  | Finished_act(Gn in M),
            Completed_act(G1 dist in M) + ...  + Completed_act(Gn dist in M))
C2Join = Go(Finished_act(G1 in M) | ...  | Finished_act(Gn in M),
            ¬Initial_act(C1 in M) |
            (Completed_act(G1 dist in M) + ...  + Completed_act(Gn dist in M)))
...
```

The first continuation instance may be set running if any of the guard instances has completed. We use distinct queries so a record of which guard instance was used to satisfy the join condition for the continuation instance is kept. The second continuation instance may be set running only if the first continuation instance has already been set running and another guard instance has been satisfied. Once all guard instances have finished, we cancel the remaining continuation instances that are yet to be set running. Once all the continuation instances are no longer in the **Initial** state, we cancel outstanding guards.

- D = Discriminator(m)(G1, ..., Gn; C)

```
D = Par(G1, ..., Gn, DJoin, C', Ti)
DJoin = Go(Cancelled_act(G1 dist in D) + ...  + Cancelled_act(Gn dist in D) |
           ...  (n-m+1) times ...  |
           Cancelled_act(G1 dist in D) + ...  + Cancelled_act(Gn dist in D),

           Completed_act(G1 dist in D) + ...  + Completed_act(Gn dist in D) |
           ...  m times ...  |
           Completed_act(G1 dist in D) + ...  + Completed_act(Gn dist in D))
C' = SeqCancel(CJoin, C))
CJoin = Go(Completed_act(DJoin in D))
Ti = Choice(Cancelled_act(DJoin in D), TiCanc, Completed_act(DJoin in D), TiComp)
TiCanc = CancelActivity(D in D)
TiComp = CancelActivity(G1 in D) | ...  |  CancelActivity(Gn in D)
```

Here, we use the synchronisation activity DJoin to assess whether the threshold for guard instances completing successfully has been reached, or whether it will necessarily not be

reached. Once m guards have completed successfully, the instance of `DJoin` completes. But, if $n - m + 1$ guard instances get cancelled, then m instances will never get completed. Thus, the instance of `DJoin` gets cancelled. The join for the continuation instance depends on the status of `DJoin` – once a result for `DJoin` is determined, a result for `CJoin` will be established. If `DJoin` completes, completion of `CJoin` will follow, causing the execution of `C`. In this case, `Ti` ensures that the remaining running guard instances are cancelled. If `DJoin` gets cancelled, `Ti` ensures that the whole discriminator activity is cancelled.

- `ML = MultiLimit(n)(E(join(J)))`

```
ML = Par(E1,J(ctype(J1)),E2,J2,...,En,Jn)

E1 = SeqCancel(Go(Cancelled_act(J1 in ML), Completed_act(J1 in ML)), E)
J2 = SeqCancel(Go(Cancelled_act(J1 in ML), Completed_act(J1 in ML)), J)
E2 = SeqCancel(Go(Cancelled_act(J2 in ML), Completed_act(J2 in ML)), E)
J3 = SeqCancel(Go(Cancelled_act(J2 in ML), Completed_act(J2 in ML)), J)
...
```

n instances of `E` (the `ExecAct`) and `J` (the join condition) are created. The first join condition, which is declared to be of type name `J1` in the characterisation of `ML`, is immediately set running. If it gets cancelled, then `E1`'s join fails, as well as `J2`'s. Cancellation is propagated through the remaining join and execution activity instances. If the first join condition completes successfully, the first execution activity instance, `E1`, is set running, as well as the next join condition, `J2`. If `J2` fails, then cancellation is propagated to the remaining join and execution activity instances, as before.

- `MSL = MultiLimitSeq(n)(E(join(J)))`

```
MSL = Par(E1,J(ctype(J1)),E2,J2,...,En,Jn)

E1 = SeqCancel(Go(Cancelled_act(J1 in MSL), Completed_act(J1 in MSL)), E)
J2 = SeqCancel(Go(Cancelled_act(J1 in MSL),  Finished_act(E1 in MSL)), J)
E2 = SeqCancel(Go(Cancelled_act(J2 in MSL), Completed_act(J2 in MSL)), E)
J3 = SeqCancel(Go(Cancelled_act(J2 in MSL),  Finished_act(E2 in MSL)), J)
...
```

As `MultiLimit`, except that when the first join condition completes successfully, the instance of `E1` is set running, but the instance of `J2` has to wait until `E1` has finished before it is set running, and so on.

- `MS = MultiSeq(E(join(J)))`

```
MS = Multi(E(join(J')))
J' = SeqCancel(Go(Finished_act(E dist in MS) +  Running_act(MS dist in MS)), J)
```

A `MultiSeq` type may be considered an abbreviation for a `Multi` type whose join is a composite condition, which prescribes firstly that either:

- This is the first join condition instance, as determined by a distinct query on MS being in the Running state, or
- The previous execution activity instance must have finished, as determined by a distinct query on an instance of E finishing

And secondly, that the original join condition on the execution activity instance, J, holds.

- Exit

```
CancelActivity(Root)
```

Root is a distinguished customised type name, defined by Liesbet, for the root instance of a workflow model.

- Empty

```
Go(True)
```

- FC = FreeChoice

```
FC = Choice(Empty, Empty; Empty, CancelActivity(FC in FC))
```

There is a race between which of the two Empty guard instances, in the Choice, will complete first. If the left-hand Empty guard instance wins out, then the continuation instance to be executed is an instance of Empty, which trivially completes. Thereafter, the Choice trivially completes. If the right-hand Empty guard instance wins out, then the continuation instance to be executed is an instance of CancelActivity, which cancels the Choice instance.

3.5 Support for YAWL Workflow Patterns

In Table 3.1, we present an overview of how the Liesbet meta-model supports the YAWL workflow patterns [125].

We provide the following comments to accompany this table:

(i) Synchronisation, i.e. XOR/AND/OR-join behaviour, is supported thus:

- Implicit Synchronisation when activity completes.
- Arbitrary Synchroniser can run in parallel.

(ii) Our version of the Discriminator pattern is more general than that presented in [125]. In [125], Discriminator is supported by means of a threshold on the number of completed instances of a multiple instance activity. While we also support this sort of Discriminator, our main support for the Discriminator pattern is through the Discriminator activity type, which can be used to synchronise on arbitrary activity instance executions, not just those of a continuation activity type belonging to a single multiple-instance activity instance.

(iii) Liesbet *comprehensively* supports the definition of arbitrary cycles, by virtue of Multi types. Liesbet disallows the specification of cycles which may *otherwise* occur in a Liesbet model, as they may always be replaced by the use of Multi types. This restriction provides greater clarity and simplicity to the semantics of Liesbet. For example, the model A=Seq(A,B)

Workflow Pattern	Satisfied How?
1 Sequence	`Seq`
2 Parallel Split	`Par`
3 Synchronisation, a.k.a. AND-JOIN	Yes[(i)]
4 Exclusive Choice	`DefaultChoice, Choice`
5 Simple Merge, a.k.a. XOR-JOIN	Yes[(i)]
6 Multiple Choice	`MultiChoice`
7 Synchronising Merge, a.k.a. OR-JOIN	Yes[(i)]
8 Multiple Merge	`Multimerge`
9 Discriminator	`Discriminator`[(ii)]
10 Arbitrary Cycles	Yes[(iii)]
11 Implicit Termination	Yes[(iv)]
12 Multiple Instances (MIs) Without Synchronisation	`Par, MultiLimit, Multi`[(v)]
13 MIs With A Priori Design Time Knowledge	`Seq` with `Par`[(vi)]
14 MIs Instances With A Priori Run Time Knowledge	`MultiLimit` or `Multi`[(vii)]
15 MIs Without A Priori Run Time Knowledge	`MultiLimit` or `Multi`[(viii)]
16 Deferred Choice	`DeferredChoice`
17 Interleaved Parallel Routing (Unordered Sequence)	`UnorderedSeq`
18 Milestone	`Stop` or `Go`[(ix)]
19 Cancel Activity	`CancelActivity`[(x)]
20 Cancel Case	`Exit`[(xi)]

Table 3.1: Satisfaction of YAWL Workflow Patterns [125, 64]

---

contains a cycle, which may instead be written as `Seq(Multi(A),B)`. A cycle check is made by the verification tool that we have implemented for `Liesbet` – see Section 10.3 – to ensure that cycles that may be introduced into a `Liesbet` model, other than through the use of `Multi` types, are not present. Note that the use of arbitrary cycles (by means of `Multi` types) should be carefully marshalled by an authoring tool, as their collective effect is often unclear.

(iv) `Liesbet` operates on the basis of implicit termination. Unlike in Petri-net formalisations, for instance, where there is a need to aggregate tokens, and a useful way of doing this is to have an explicit end place for a construct, implicit termination is a good choice for `Liesbet` as it promotes an intuitive and succinct way of way of authoring `Liesbet` models.

(v) We consider the following interpretations of this workflow pattern (#12):

- The number of instances is known *a priori* at design-time, in which case we can effect the pattern according to the following `Liesbet` coding.

```
Par(ExecAct, ..., ExecAct, Cont)
```

The execution of the `ExecAct` instances and the continuation of the workflow model (`Cont`) is initiated at the same time. That is, we do not synchronise `Cont` on the instances having finished.

- The number of instances is unknown at design time, in which case we can effect the pattern with either a `MultiLimit`, or `Multi`, as follows. Again, we do not synchronise `Cont` on the `ExecAct` instances having finished.

```
Par(MultiLimit(T)(ExecAct(join(ExecActJoin))), Cont)
Par(Multi(ExecAct(join(ExecActJoin))), Cont)
```

(vi) Pattern #13 has the following simple coding in `Liesbet` and does need any additional `Liesbet` construct. The execution of the continuation instance is synchronised on the completion of `ExecAct` instances.

```
Seq(Par(ExecAct, ..., ExecAct), Cont)
```

(vii) For pattern #14, we do not know how many instances will be created *a priori* at design-time. However, this does not mean that we cannot bound the number of instances that may be created. By imposing a bound, we arrive at a simpler semantics for the pattern. The appropriate `Liesbet` construct, in this case, is `MultiLimit` (or `MultiLimitSeq`). If we do not wish to impose a bound, the appropriate `Liesbet` construct is `Multi` (or `MultiSeq`).
In order to represent a lower limit on the number of instances created, as described in [125], we could define a new `Liesbet` type as a macro, which would elaborate to:

```
Par(ExecAct, ...  l instances ..., ExecAct, MultiLimit(m-l)(ExecAct(join(ExecActJoin))))
```

Here, there are l instances of `ExecAct` which are created alongside a `MultiLimit` which specifies a limit for further instances of $m - l$, where l is the least number, and m is the maximum number, of instances that may be created overall.
In order to place a threshold on completion, after which all outstanding instances get cancelled, as described in [125], we could define another new `Liesbet` type as a macro, which would (in the case of `Multi`) elaborate to:

```
P=PriPar(CancelActivity(M in P)(join(CAJoin)), M)
M=Multi(ExecAct(join(ExecActJoin)))
```

The join condition `CAJoin`, for the `CancelActivity` type, would be a `Go` activity type which would complete successfully once the given number of instances of the `ExecAct` successfully completes. To see how this may be expressed, refer to the elaborated definition of the `Discriminator` pattern, described in Section 3.4.
The purpose of the `CancelActivity` activity is to cancel the `Multi` once enough instances of its contained `ExecAct` type have completed. This behaviour is prioritised, as specified by using `PriPar`. It is notable that the formalisation in [125] does not appear to enforce such a prioritisation. That is, even though it specifies cancellation of remaining instances as a possible next transition (according to its transition system-based semantics), it does not enforce such a transition as necessarily occurring next.

(viii) Ibid, for pattern #15.

(ix) Using the `Stop` and `Go` synchronisation types, in `Liesbet`, a workflow author has much greater expressivity in capturing intended synchronisation behaviour than just the *Milestone* behaviour in `YAWL`. See also the conclusions to this thesis (Chapter Twelve) for more discussion, regarding this matter.
The *Milestone* `YAWL` workflow pattern is where [125, 64] "[t]he enabling of an activity depends on the case being in a specified state, i.e. the activity is only enabled if a certain milestone

has been reached which did not expire yet. Consider three activities named A, B, and C. Activity A is only enabled if activity B has been executed and C has not been executed yet, i.e. A is not enabled before the execution of B and A is not enabled after the execution of C."

The behaviour described in the example of the three activity types, `A`, `B` and `C`, can be effected with a `Stop`, thus: `Stop(Completed_act(C), Completed_act(B))`.

(x) In `Liesbet`, cancellation is achieved by moving the implicit state machines for target activity instance/s, and the machines of their descendant instances, to the `Cancelled` state.

(xi) In `Liesbet`, cancellation of the process instance is achieved by cancelling the root instance, which has the effect of propagating cancellation to all extant activity instances.

3.6 Mapping WS-BPEL to Liesbet

Having shown how we support the YAWL workflow patterns, it is instructive to consider how we might support (the verification of) the Web Services Composition language `WS-BPEL`. We present a mapping for the control flow aspects of `WS-BPEL` to `Liesbet`, omitting fault, compensation and termination handling, as well as correlation, in order to provide a means of verifying properties of `WS-BPEL` compositions, such as workflow soundness (see Sections 1.1 and 7.1).

Note that the mapping, described here, will strip away all information that is not relevant to verifying `WS-BPEL` at the control perspective. As such, the resultant `Liesbet` model, while sufficient for some verification functions, does not represent a complete formalisation of the semantics of `WS-BPEL`. For instance, information pertaining to the data perspective of `WS-BPEL` models is not accounted for. In this case, we will assume that execution may proceed down any branch of `<pick>` construct, for example. This would not be a limiting assumption, but may be overly cautious for individual instances of the construct.

3.6.1 Mapping of Join and Transition Conditions

We assume that all links and activities within a composition are uniquely named, and that implicit join and transition conditions (see Section 2.2.3) are made explicit. If not, this would be trivially enforced by an assumed pre-processor.

When an activity specifies incoming links, in `WS-BPEL`, then it is specifying a join condition on its execution. Such an activity is mapped, to `Liesbet`, as we would map the specification of a join condition in `Liesbet`, i.e. to a two-child `SeqCancel`, where the first child is the join activity type, as presented in Section 3.1.7. When an activity specifies outgoing links then it specifies a number of accompanying transition conditions. These are collected together, when mapped to `Liesbet`, in the `Seq` containing the join/main activity and transition condition types, described in Section 3.1.7.

Elaborating, for an activity with incoming and outgoing links, the `Liesbet` mapping would be: `Seq(SeqCancel(JC, A), LTC`$_1$`, ..., LTC`$_n$`)`, where `JC` is the mapped join condition type, `A` is the mapped main activity, and `LTC`$_1$`, ..., LTC`$_n$ are the mapped transition conditions of the activity, in the order that they appear in the original `<sources>` `WS-BPEL` container. For an activity with just incoming links, the `Liesbet` mapping would be: `SeqCancel(JC, A)`. For an activity with just

outgoing links, the Liesbet mapping would be: Seq(A, LTC_1, ..., LTC_n). For an activity with no links, the Liesbet mapping would be: A.

For an activity with outgoing links, any of the mapped transition condition types, LTC_1, ..., LTC_n is derived as follows. If the link transition condition is simply the expression true, then the mapped type will be a named Empty type. This will trivially complete. Otherwise, as the transition condition has the possibility of evaluating to either true or false, the mapped type will be a named FreeChoice type, which will non-deterministically complete or get cancelled. The name given to a mapped transition condition type is its (unique) pertaining link name.

We now consider the mapping of join conditions. As an example, which will make the reading of the following description easier to follow, consider the WS-BPEL process fragment in Figure 3.5. Ignoring the "tricky" labelling afforded to this fragment for the time being, we note the top-level <flow> activity contains a pair of <link> definitions and three activities, which use these links. The <empty> activity, contained within the <if> activity, is a target of the toSkipped link, whose source is contained within the <receive> activity. Its implicit join condition is that the toSkipped link evaluates to true. The mapping of the source of this link would be a link transition condition, as described above, attached to the mapping of the <receive> activity.

In general terms, the join condition of an activity is mapped to a Go type. Its GoQuery is a direct mapping of the join condition specified in the original WS-BPEL source. Each query on link status, participating in the original join condition, will be mapped to a Completed_act(LINKNAME in FLOWACT) query, where LINKNAME is the unique name given to the link, and FLOWACT is the name of the <flow> activity which defines the link. In WS-BPEL terms, this has the effect of querying whether the status of the link is true. In the source presented in Figure 3.5, the implicit join condition of the <empty> activity would be mapped to a Go type with GoQuery (in part): Completed_act(toSkipped in foobar).

Arbitrary join conditions are mapped to a Liesbet expression tLiesbetJC by the mapping function, $\mathcal{M}_{BPEL}[\![-]\!]$, defined thus. Without loss of generality, or expressivity, we cast a join condition as an arbitrarily nested expression that makes use of just one- and two-argument boolean operators.

$$\mathcal{M}_{BPEL}[\![\neg C]\!] = \neg\mathcal{M}_{BPEL}[\![C]\!]$$
$$\mathcal{M}_{BPEL}[\![C_1 \wedge C_2]\!] = \mathcal{M}_{BPEL}[\![C_1]\!] \mid \mathcal{M}_{BPEL}[\![C_2]\!]$$
$$\mathcal{M}_{BPEL}[\![C_1 \vee C_2]\!] = \mathcal{M}_{BPEL}[\![C_1]\!] + \mathcal{M}_{BPEL}[\![C_2]\!]$$
$$\mathcal{M}_{BPEL}[\![\text{Query on link:LINKNAME, flow:FLOWACT}]\!] = \text{Completed\_act(LINKNAME in FLOWACT)}$$

The mapped Go type, effecting the join condition, is: Go(allLinksIn, tLiesbetJC | allLinksIn), where allLinksIn is a query which is only satisfied if all of the mapped incoming links have a value, that is all of the pertaining source activities (either FreeChoice, or Empty, as mapped) have finished. The definition of allLinksIn is thus: Finished_act($LINKNAME_1$ in $FLOWACT_1$) | ... | Finished_act($LINKNAME_n$ in $FLOWACT_n$), where there is an occurrence of Finished_act($LINKNAME_i$ in $FLOWACT_i$) for every link, i, whose status is queried within the join condition.

The specification of allLinksIn in the GoQuery means that all links have to have a value before the query, i.e. join condition, can return a result. The specification of allLinksIn *as* the StopQuery means that, once all links have a value, if the GoQuery cannot be satisfied then the join condition must be false. Thus, we fail (i.e. cancel) the join condition (i.e. Go type).

It is worth making a note about the criterion, specified at the end of Section 2.2.6; namely, that an activity's incoming links must all have had their values determined before outgoing links can be set to `false` (as part of dead-path elimination). An example of the pertaining issue is as follows. It is taken, nearly verbatim, from Section 11.6.2 of [87]. In the example, the `toSkipped` link creates a control dependency from the `<receive>` activity to the `<empty>` activity in the `<if>`. The `fromSkipped` link creates a dependency from the `<empty>` activity to the `<reply>` activity. These two links create a transitive dependency from the `<receive>` activity to the `<reply>` activity. Even though the `<if>` condition evaluates to `false`, thus skipping the `<empty>` activity, the transitive dependency is retained; and the status of `fromSkipped` is not set to `false` until after the status of `toSkipped` is known.

The semantic characterisations for `Liesbet`, in the forms presented in Sections 5 and 6, are not capable of accounting for this criterion. To do so requires a non-trivial extension to these characterisations, which is not interesting from the point of view of this thesis. However, we have accommodated it in our implementation of the verification engine for `Liesbet`, and details of the required extensions to the semantic characterisations of `Liesbet` is presented in the documentation for the engine, which is available from the author on request.

Some further notes are appropriate. `WS-BPEL` scopes may be marked as isolated, which does not carry quite the same meaning as our use of the word in a `Liesbet` context. For the discussion here, it suffices to say that for a `WS-BPEL` marked as isolated, the status of links leaving the scope will not be visible to targets outside the scope until the scope has completed (see Section 2.2.7). In mapping this criterion, the `allLinksIn` expression, for join conditions having incoming links which cross isolated scope boundaries, will include further `Finished_act(SCOPE in FLOWACT)` queries on the given scopes, where `FLOWACT` for any such query is the name of the `<flow>` activity defining the link.

The mapping from `WS-BPEL` includes the generation of a number of synchronisation rules, which prevents the execution of activities, which are not start activities, until a start activity has completed (see Section 2.2.1). We identify all of the activities, in a composition, which could count as start activities. We then divide this group into those activities which are marked as start activities (by virtue of their `createInstance` attribute being `true`), and those that are not. For every activity in the latter group, we write a synchronisation rule indicating that it may not advance until an activity from the former group has been executed. Instances of a synchronisation rule, applied for this purpose, would look as follows.

$$\texttt{SyncRule}(\texttt{Act}_i, \texttt{True}, \texttt{Finished\_act}(\texttt{Act}_{s1}) + \ldots + \texttt{Finished\_act}(\texttt{Act}_{sn}))$$

Here, $\texttt{Act}_i$ is the name of an activity from the set of non-starters, and each $\texttt{Act}_{si}$ is one of the start activities. The rule says that a non-starter may only advance once a starter has finished.

A scope with event handlers is mapped to: $\texttt{S=Par(A, EvHa}_1\texttt{, ..., EvHa}_n\texttt{)}$, where `A` is the main activity of the scope, and each $\texttt{EvHa}_i$ is an event handler. An Event Handler is a `Multi` activity type, whose join condition is a `Stop` activity type whose `StopQuery` is `Finished_act(A in S)`, which will have the effect of disabling the event handler from creating further instances once `A` has finished (see Section 2.2.7).

```
<flow name="foobar">
   <links>
      <link name="toSkipped" />
      <link name="fromSkipped" />
   </links>

   <receive ...>
      <sources>
         <source linkName="toSkipped" />
      </sources>
      ...
   </receive>

   <if>
      <condition>
         ...  <!-- evaluates to false -->
      </condition>

      <empty name="skipped">
         <targets>
            <target linkName="toSkipped">
         </targets>
         <sources>
            <source linkName="fromSkipped">
         </sources>
      </empty>
   </if>

   <reply ...>
      <targets>
         <target linkName="fromSkipped" />
      </targets>
   </reply>
</flow>
```

Figure 3.5: Process Fragment Capturing Some Tricky WS-BPEL Link Semantics

3.6.2 Mapping of Other Activity Types

The mapping for the other activity types, presented in Table 2.1, is presented in Table 3.2.

We also note that WS-BPEL provides no direct support for YAWL patterns: *Multiple Merge*, *Discriminator*, *Arbitrary Cycles*, *Interleaved Parallel Routing* and *Milestone*. In many cases, where use of one of these patterns would be desirable to a workflow model author, the specific functionality required would still be able to be captured in WS-BPEL. However, to do so may be non-trivial. The common cause behind the said modelling difficulty for all five listed patterns is WS-BPEL's link semantics. For WS-BPEL to more easily support these patterns, modification of its link semantics

<receive>, <reply> <invoke>, <assign>, <wait>	Abstracted as Empty activity type
<exit>	Exit
<empty>	Empty
<sequence>	Seq
<if>	Choice, DefaultChoice
<while>	MultiSeq
<repeatUntil>	MultiSeq with "free iteration"
<forEach>	MultiLimit, MultiLimitSeq, Multi, MultiSeq
<pick>	Choice
<flow>	Par

Notes:

- For <repeatUntil>, its mapping is a MultiSeq with a "free iteration", meaning that the join condition on the contained execution activity (ExecAct) is a distinct query on the MultiSeq having been started, in disjunction with the mapping of <repeatUntil>'s condition.
- For <pick>, the onMessage and onEvent conditions are mapped to Empty, which trivially completes.

Table 3.2: Mapping of Some WS-BPEL Activity Types to Liesbet

would be necessary.

3.7 Concluding Remarks

In defining Liesbet, we have sought to understand the *true nature* of workflow, and thus the fundamental concepts that need to be represented with Liesbet. We have presented the constructs of Liesbet, along with a specification of their informal operational semantics. In Chapters Five and Six, we will provide formal characterisations of the semantics of these constructs. We have also presented the definition of additional *intended semantics* for Liesbet, which prescribe further constraints regarding the evolution of Liesbet models.

We have taken our first step towards greater flexibility in workflow models through the proposal of *Synchronisation Rules*, which may be used to provide a notion of flexibility that may be captured as *Flexible Workflow = Concrete Model + Policies for Constraint*. We have described how such rules may be useful. For instance, we are able to capture the behaviour of Liesbet's PriPar construct using such a rule.

We have also presented a reduced set of patterns with which (we show) all patterns may be represented. This is a useful result as it enables us to propose the true nature of workflow to be captured by (the semantics of) this reduced set. We provide further elaboration of this point in the conclusions to this thesis, presented in Chapter Twelve.

We have also documented how Liesbet captures all of the YAWL patterns, as well as describing

its support for modelling the control flow perspective of WS-BPEL, in order to usefully facilitate verification of WS-BPEL compositions.

In the next chapter, we present some example workflows represented using Liesbet in order that the reader may gain some insight into how it is used.

Chapter 4

Liesbet Meta-model Examples

In this chapter, we show the use of Liesbet to represent some examples of workflow, proposed by members of the Business Process Management (BPM) community. It is insightful to consider the modelling of these examples using Liesbet as most of them (as indicated by [8] citations) have been suggested as benchmarks by which ontologies for workflow should be evaluated. They also provide a good coverage of Liesbet's constructs, which is useful for the interested reader in understanding the operation of the patterns.

- The Synchronisation Example (4.1) provides a simple example of synchronisation between the execution of one activity instance and the completion of another, where these instances execute in parallel threads. This is a motivating example from [122].
- The Distinct Query Example (4.2) provides an example of the need for distinct reference queries. This sort of query is principally required within Multi* activity types (but may also be required in other types, such as Multimerge) to differentiate between instances of activity types in satisfying a query.
- Examples (4.3), (4.4) and (4.5) are some benchmark examples from [8].

4.1 Synchronisation Example

This is an example of where we need to synchronise the execution of one activity instance to occur after another has finished. As can be seen from Figure 4.1, we execute A, then execute the sequences B,C,D and E,F,G in parallel, and after both sequences have completed, we execute H. There is a join condition on the execution of F that (the instance of) C must have completed first.

The Liesbet definition is straightforward, viz.

```
Seq(A,Par(Seq(B,C,D),Seq(E,F,G)),H)
F= Act(join(Go(Completed_act(C))))
```

We start by representing the workflow model without the additional synchronisation. This is accounted for by the first line, which is a sequence of a basic instance A, a parallel instance, followed by a basic instance H. The parallel instance consists of two sequences of three basic instances: B, C and D, and E, F and G.

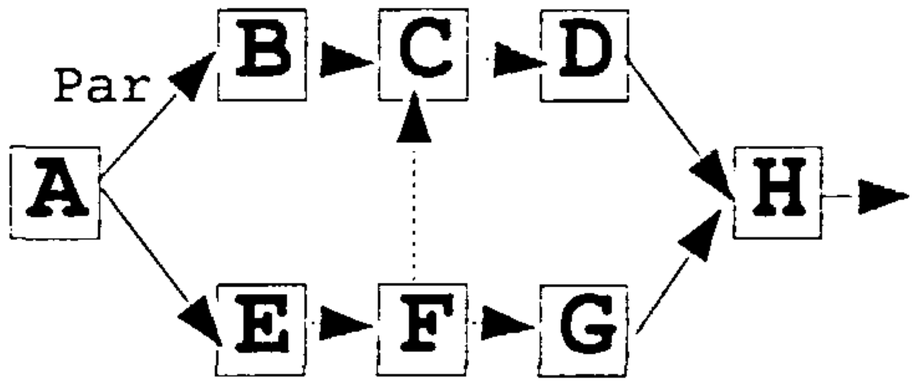


Figure 4.1: Synchronisation Example [122]

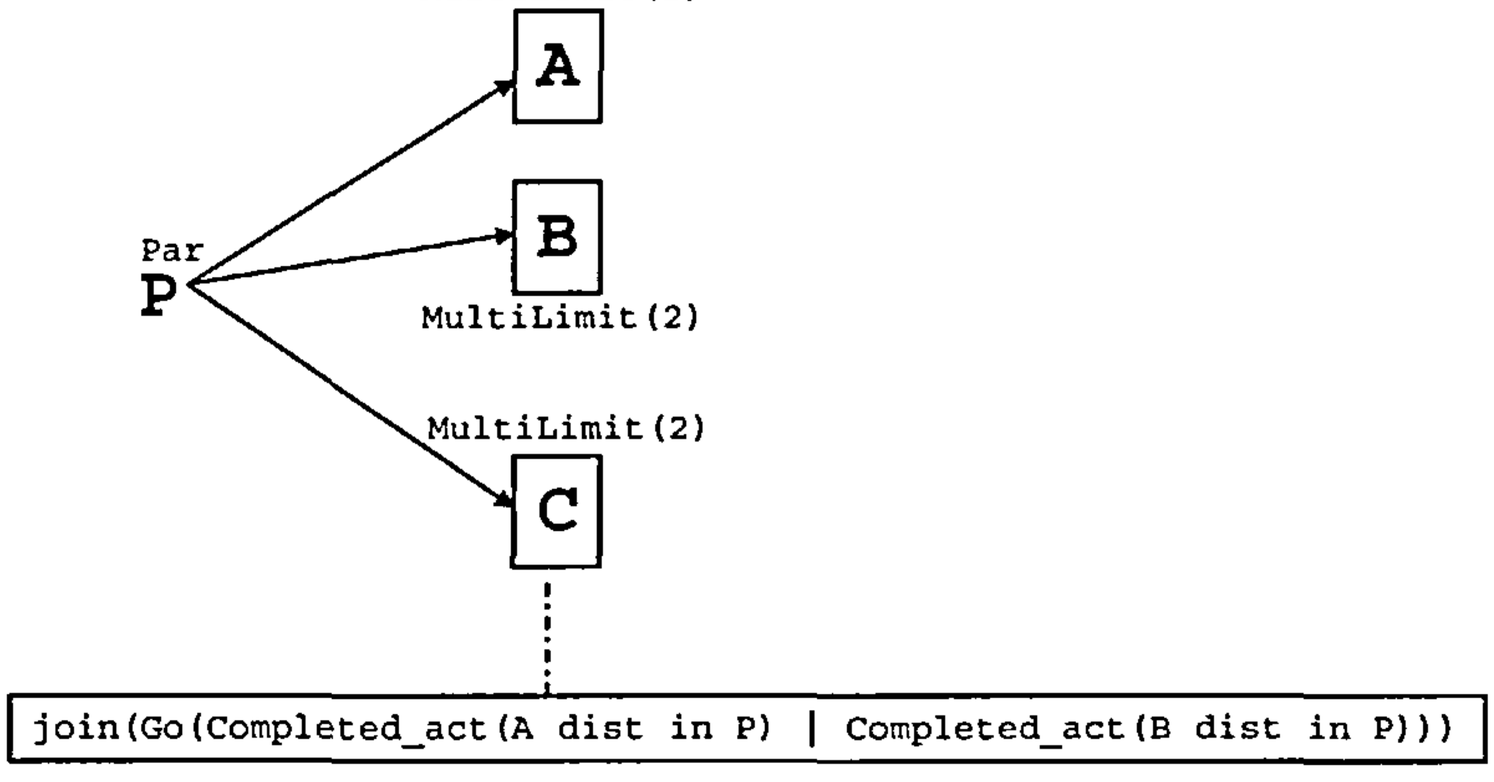


Figure 4.2: Distinct Query Example

The synchronisation is added in the second line, where it says that F is a basic activity instance that has a join condition, which is a Go activity type with GoQuery Completed_act(C), which means that the join condition will block until the instance of C has been completed.

4.2 Distinct Query Example

In Figure 4.2, we present an example illustrating the need for distinct querying. Here, we use distinct querying in order to appropriately satisfy the join condition on basic instance C, which is the execution activity type (ExecAct) in a MultiLimit instance.

The Liesbet definition of the model is as follows.

```
P= Par(MLA, MLB, MLC)
MLA= MultiLimit(2)(A)
MLB= MultiLimit(2)(B)
MLC= MultiLimit(2)(C(join(Go(Completed_act(A dist in P) | Completed_act(B dist in P)))))
```

The join condition on C, in MultiLimit instance MLC, is only satisfied when distinct instances of A and B (as created by MLA and MLB instances) have completed. If distinct querying were not used then any completed instance of A and any completed instance of B could be used to satisfy the join conditions of the two instances of C, created by MLC.

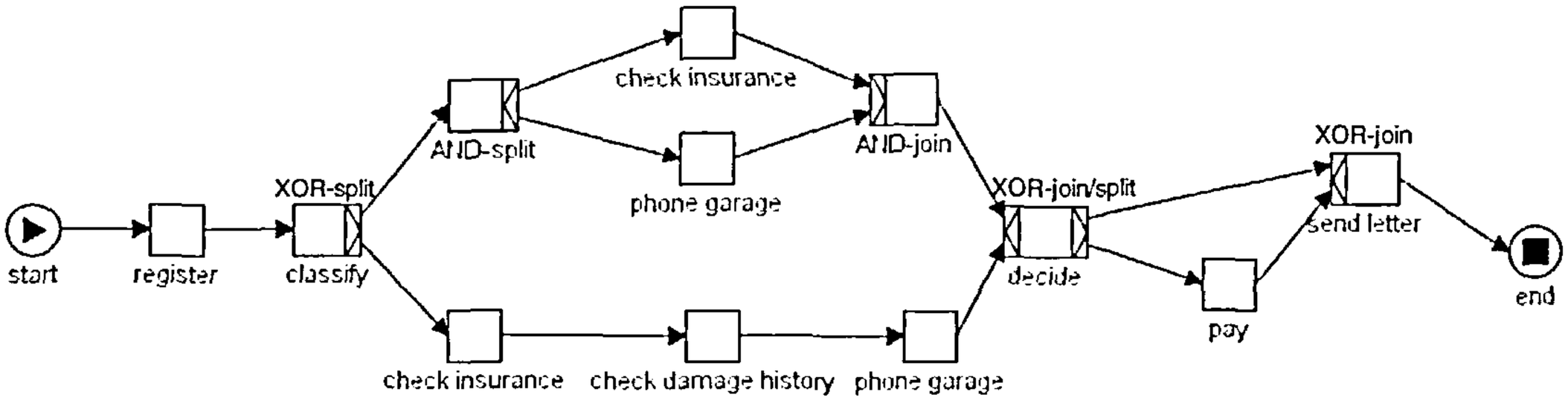


Figure 4.3: Insurance Company Workflow as a YAWL EWF-net, from [8]

4.3 Insurance Company

The following example is of an *Insurance Claim Handling Procedure*, adapted from [8].

> An insurance company processes claims from its drivers for traffic accidents, using the following procedure. Every claim made is *Registered*, and then *Classified*. There are two categories of classification: simple and complex claims. For simple claims two activities need to be executed: *Check Insurance* and *Phone Garage*. These activities are independent of each other. The complex claims require three activities to be executed: *Check Insurance*, *Check Damage History* and *Phone Garage*. These activities need to be executed sequentially in the order specified. Then, having completed the two (for simple), or three (for complex), activities, a *Decision* regarding paying the claim is made. If the decision is to pay, then *Payment* will be made. Then, in any event, a letter will be *Sent* to the claimant outlining the decision.

A representation of this example, as a workflow model, using a YAWL EWF-net is shown in Figure 4.3. A representation in Liesbet is as follows. Note that, for simplicity, we do not include a specification for the decision activities: SimpleClaimDecision and PayDecision. These would likely be elaborated to Go or Stop types.

```
Seq(Register,Classify,SimComChoice,PayChoice,SendLetter)
SimComChoice= DefaultChoice(SimpleClaimDecision, Par(CheckInsurance,PhoneGarage);
                                Seq(CheckInsurance,CheckDamageHistory,PhoneGarage))
PayChoice= DefaultChoice(PayDecision,Pay;Empty)
```

At the top-level, we simply have a sequence, largely consisting of the basic activities given in italics in the example text. The two activity types in the Seq that are not basic activities are SimComChoice and PayChoice. The first, SimComChoice, makes a DefaultChoice based on whether the claim is classified as simple or not (i.e. complex claim). This is respectively determined by whether the guard activity, SimpleClaimDecision, completes successfully, or is cancelled. If it is a simple claim then the given Par of basic activities CheckInsurance and PhoneGarage is executed. If it is a complex claim then the given Seq of basic activities CheckInsurance,CheckDamageHistory and PhoneGarage is executed. PayChoice is simply another DefaultChoice based on whether the Insurance Company decides to pay the claim or not. This is respectively determined by whether the guard activity, PayDecision, completes successfully, or is cancelled. If the claim is to be paid, Pay is executed, otherwise we do nothing (as reflected by Empty).

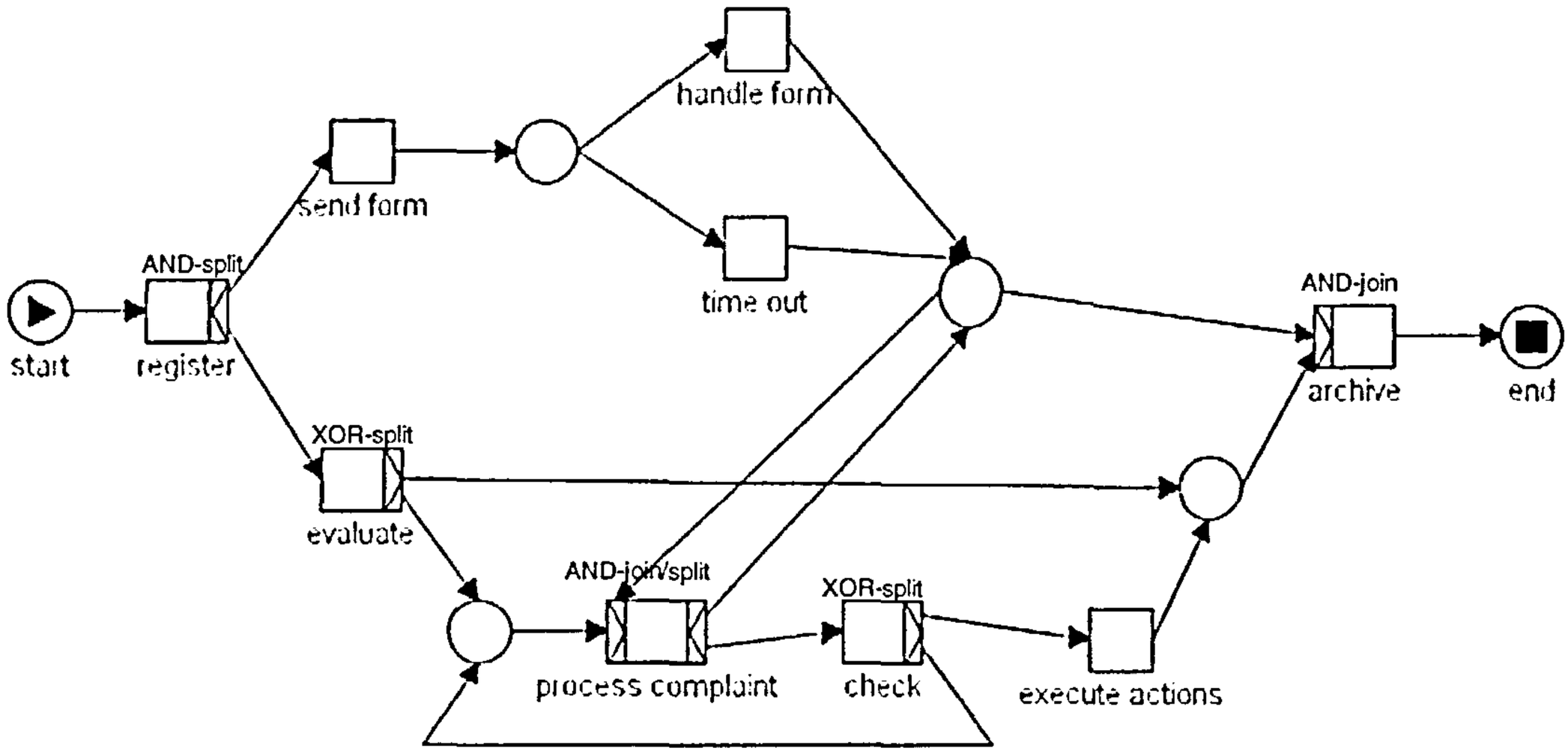


Figure 4.4: Complaints Handling Workflow as a YAWL EWF-net, from [8]

4.4 Complaints Handling

The following example is of a *Complaints Handling Procedure*, adapted from [8].

> A travel agency has a complaints department. Each complaint is first *Registered.* After registration a form is *Sent* to the customer with questions about the nature of the complaint. There are two possibilities: the customer returns the form within a stipulated time or not. If the form is returned in time, it is *Processed* automatically resulting in a report which can be used for the actual processing of the complaint. If the form is not returned on time, a time-out occurs resulting in an empty report. Note that this does not necessarily mean that the complaint is discarded.
>
> After registration and in parallel with the form handling, the preparation for the actual processing is started. First, the complaint is *Evaluated* to ascertain whether further processing is needed or not. Note that this decision does not depend on the form handling. If no further processing is required and the form is handled, the complaint is *Archived.* If further processing is required, the activity *Process Complaint* is executed, in which certain further actions may be proposed. For the processing of the complaint, the report resulting from the form handling is used. The result of the *Process Complaint* activity is *Checked* for quality. If the result of the check is not satisfactory, the *Process Complaint* and *Check* activities are repeated until the result is satisfactory. If the result is satisfactory, an employee *Executes* the proposed actions. After this the processed complaint is *Archived.*

A representation of this example, as a workflow model, using a YAWL EWF-net is shown in Figure 4.4. A representation of this example in Liesbet is as follows. Note that, for simplicity, we do not include a specification for the decision activity: EvaluateDecision. This would likely be elaborated to some Go or Stop type. The TimeOut activity is a timer, which completes once its implicit expiry time has elapsed. FormReturned completes once the complainant has returned their complaint form. We do not model these activity types further either.

```
Seq(Register,HandleProcessPar,Archive)
```

```
HandleProcessPar = Par(Seq(SendForm,HandleFormPar),
                       Seq(Evaluate,DefaultChoice(EvaluateDecision,PCMultiSeq; Empty)))
HandleFormPar= DefaultChoice(Go(TimeOut, FormReturned),HandleForm; EmptyReport)
PCMultiSeq= MultiSeq(PCSeq(join(Go(Completed_act(Check in PCMultiSeq),
                                   Cancelled_act(Check dist in PCMultiSeq) +
                                   Completed_act(HandleFormPar dist in HandleProcessPar)))))
PCSeq= Seq(ProcessComplaint, Check)
```

The workflow model, at the top-level, consists of a sequence of `Register` and `Archive` activities with some other processing, given by `HandleProcessPar` in between. `HandleProcessPar` is a parallel activity consisting of handling, and processing, the complaint.

For handling the complaint, we first execute `SendForm`, and then `HandleFormPar` to handle the form, if returned. `HandleFormPar` executes a `DefaultChoice` whose single guard activity completes successfully if the form is returned. In this case, `HandleForm` will be executed. If the guard instance gets cancelled, which would occur if a time-out for return of the form expires first, then `EmptyReport`, instead, will be executed. The result of executing `HandleForm` will be a report used in the processing of the complaint. The result of executing `EmptyReport` will be the generation of an empty report.

For processing the complaint, we evaluate the complaint in `Evaluate`. We then make a decision, on the basis of the evaluation, to proceed with processing the complaint or do nothing further with it, regarding processing. If we decide to proceed with processing, the `MultiSeq` activity, `PCMultiSeq`, is executed. This activity type handles the possible multiple iterations of processing the complaint and the result of the processing being checked as to whether it is satisfactory. Its execution activity type is `PCSeq`, which consists of basic activities `ProcessComplaint` and `Check`, in sequence. The first instance of `PCSeq`, i.e. the first iteration of the process complaint loop, will be executed once the `HandleFormPar` activity has completed. That is, once a report, either empty or completed by the complainant, has been filed. Future instantiations of `PCSeq` depend on the outcome of `Check`. If `Check` completes successfully then `PCMultiSeq` will not instantiate further iterations of `PCSeq` and will instead complete. If `Check` gets cancelled, then another instance of `PCSeq` is executed.

4.5 Travel Agency

Adapted from [8]:

> Consider a fragment of a Travel Agent's process for booking trips, which involves five steps: *Register*, *(Booking of) Flight*, *(Booking of) Hotel*, *(Booking of) Car* and *Pay*. The process starts with activity *Register* and ends with *Pay*. The booking activities *Flight*, *Hotel* and *Car*, which may succeed or fail, occur in between, in parallel. Cancellation of the instance of the booking process will occur in the event of a failed booking activity.

Presented in the following sub-sections are a number of variants of the Travel Agency scenario.

Travel Agency 1

Adapted from [8]:

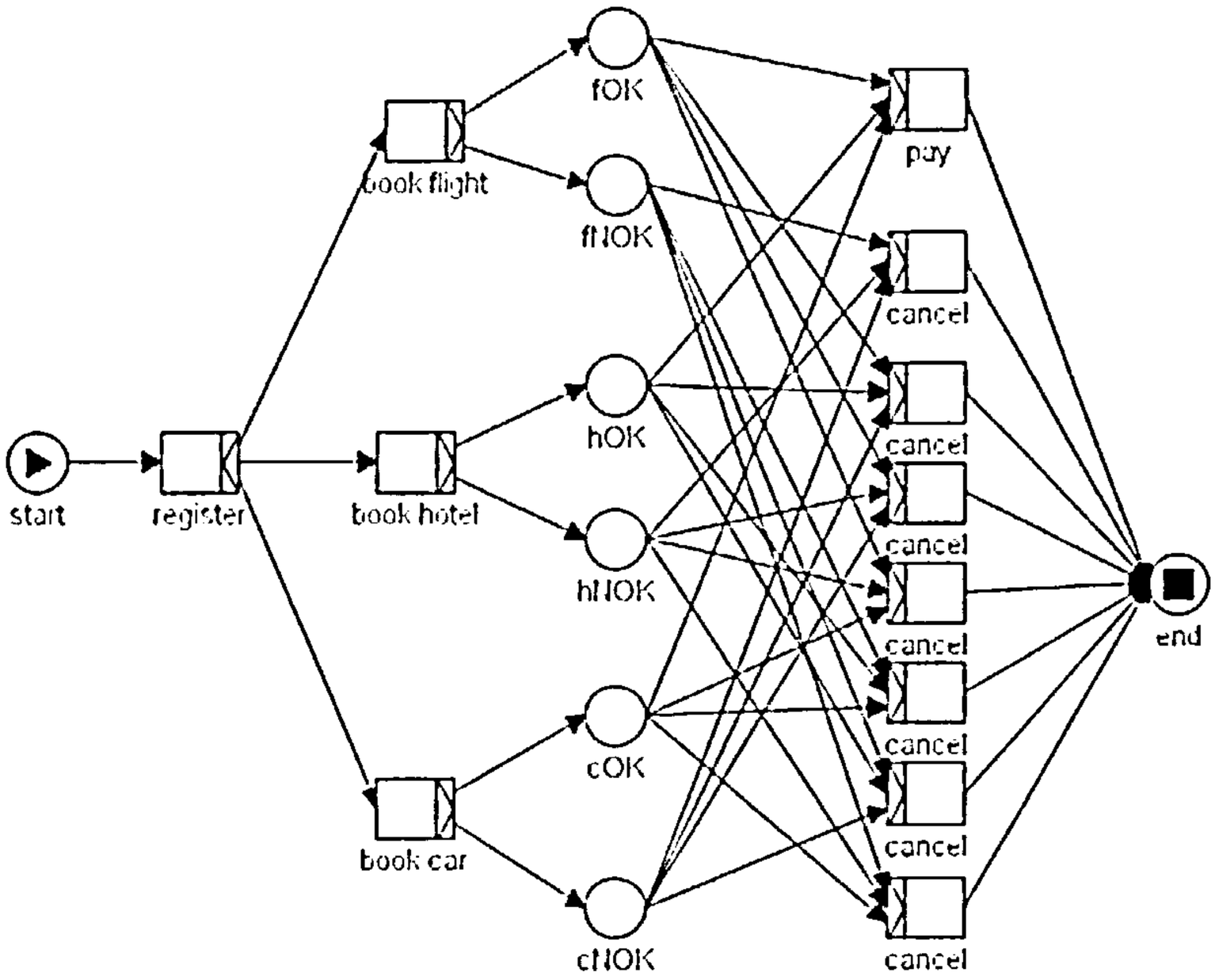


Figure 4.5: Travel Agency I Workflow as a YAWL EWF-net, from [8]

Every trip involves all three booking activities. If all three succeed, payment follows. Otherwise the process instance is cancelled. Cancellation is delayed until all three booking activities have finished.

A representation of this example, as a workflow model, using a YAWL EWF-net is shown in Figure 4.5. A representation of this example in Liesbet is as follows.

```
Seq(Register,Book,PayCancelChoice)
Book = Par(Flight,Hotel,Car)
PayCancelChoice = DefaultChoice(PayDecision,Pay; Exit)
PayDecision = Stop(Cancelled_act(Flight) + Cancelled_act(Hotel) + Cancelled_act(Car),
                   Completed_act(Flight) | Completed_act(Hotel) | Completed_act(Car))
```

Here, we execute basic activity Register and structured activities Book and PayCancelChoice in sequence. Book consists of the basic activities of booking a Flight, a Hotel and a Car, which are carried out in parallel. Once Book has finished, PayCancelChoice is executed. It is a DefaultChoice activity whose single guard is completed iff all three booking activities complete successfully but fails (i.e. gets cancelled) iff any of the booking activities fails. If the former happens, Pay is executed. If the latter happens the Liesbet activity Exit is executed, which has the effect of cancelling the process instance.

Travel Agency II

This example is the same as Travel Agency I, except that (adapted from [8]):

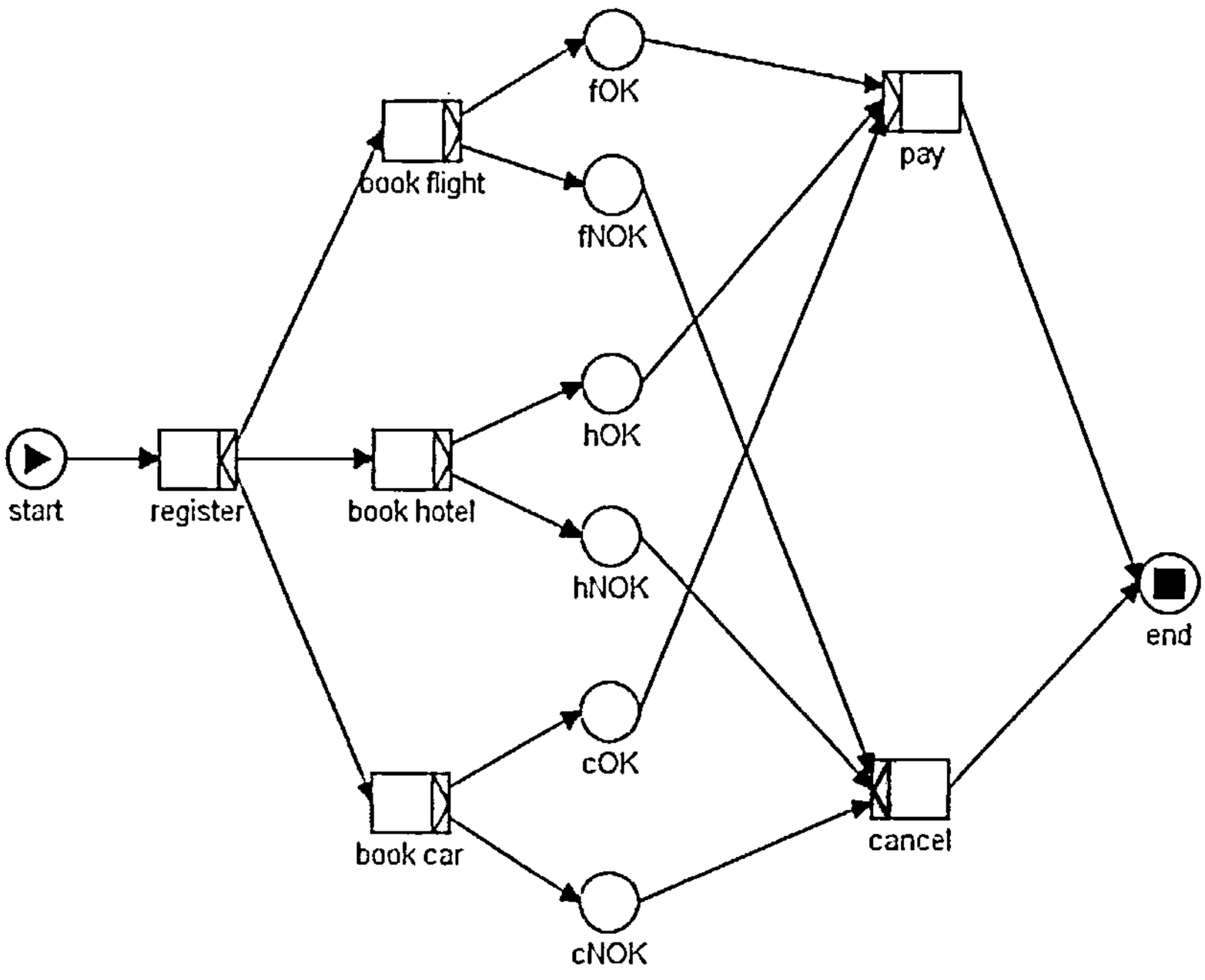


Figure 4.6: Travel Agency II Workflow as a YAWL EWF-net, from [8]

Cancellation of the process instance should occur the moment the first activity fails and, at the same time, all outstanding booking activities should be withdrawn.

A representation of this example, as a workflow model, using a YAWL EWF-net is shown in Figure 4.6. A representation of this example in Liesbet is as follows.

```
Par(Seq(Register,Book),PayCancelChoice)


Book = ...  as Travel Agency I
PayCancelChoice = ...  as Travel Agency I
PayDecision = ...  as Travel Agency I
```

This representation differs from that for Travel Agency I, in that the choice of whether to pay or cancel the process instance is made in parallel with the Book activity, meaning that the process instance may be cancelled once any of the booking attempts fail. That is, if at any time PayDecision gets cancelled (due to one of the booking activities failing), Exit, which has the effect of cancelling the process instance, will be executed.

Travel Agency III

This example is the same as Travel Agency II, except that (adapted from [8]):

> A trip may omit any of the booking activities, but clearly must involve at least one. If all of the *attempted* bookings activities succeed, the payment follows. Otherwise, the process instance is cancelled.

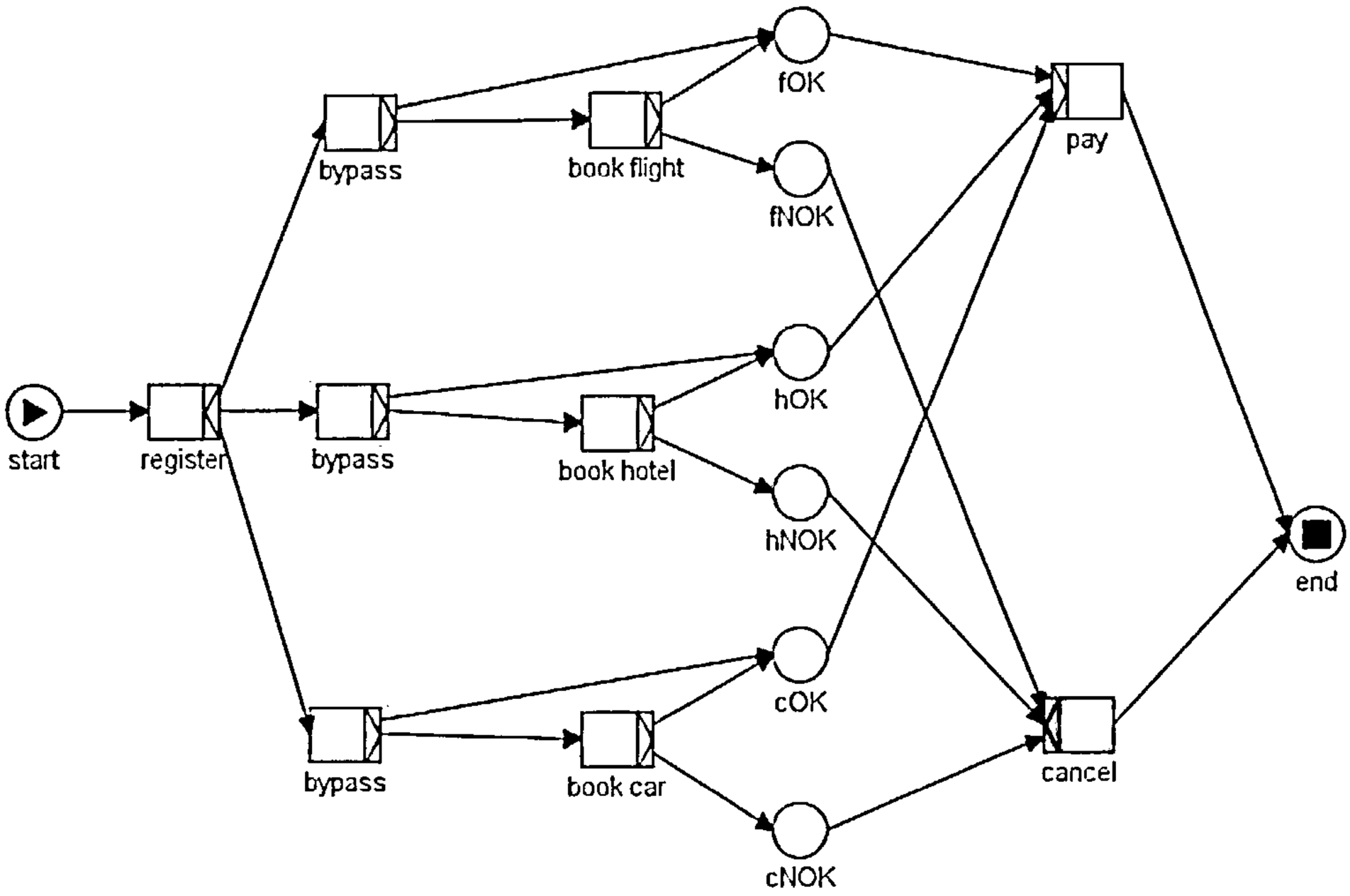


Figure 4.7: Travel Agency III Workflow as a YAWL EWF-net, from [8]

A representation of this example, as a workflow model, using a YAWL EWF-net is shown in Figure 4.7. A representation in **Liesbet** is as follows. Note that, for simplicity, we do not include a specification for the decision activities: BookFlightDecision, BookHotelDecision and BookCarDecision. These would likely be elaborated to Go or Stop types.

```
...  Root act as Travel Agency II
Book = MultiChoice(BookFlightDecision,Flight;
                   BookHotelDecision,Hotel;
                   BookCarDecision,Car)
PayCancelChoice = ...  as Travel Agency II
PayDecision = Stop(
                   (Cancelled_act(Flight) | Completed_act(BookFlightDecision)) +
                   (Cancelled_act(Hotel) | Completed_act(BookHotelDecision)) +
                   (Cancelled_act(Car) | Completed_act(BookCarDecision)) +
                   (Cancelled_act(Flight) | Cancelled_act(Hotel) | Cancelled_act(Car)),
                   (Completed_act(Flight) + Cancelled_act(BookFlightDecision)) |
                   (Completed_act(Hotel) + Cancelled_act(BookHotelDecision))   |
                   (Completed_act(Car) + Cancelled_act(BookCarDecision))
                  )
```

This representation differs from that for Travel Agency II, in that the Book activity is now a MultiChoice, meaning that not all booking activities have to be executed. As such, PayDecision is adjusted accordingly, to account for booking activities not being executed. PayDecision will

succeed iff all of the attempted booking activities succeed. It will fail iff an attempted booking activity fails, or no booking activity is attempted at all.

4.6 Concluding Remarks

It is interesting to note that the YAWL representation of this example is not quite the same as the Liesbet one. This is of note because the representation in YAWL would actually be significantly more complex to match the Liesbet representation. The difference lies in what happens if all three activities get cancelled. It would be bizarre if the *Pay* activity were still executed in this instance. This is what happens in the YAWL model, however.

We argue that the increase in complexity is significant because there would be a need to specify a number of additional places and transitions in order to capture the desired semantics. In contrast, in our Liesbet representation, it is an extra line in the StopQuery of the Stop type, i.e., (Cancelled_act(Flight) | Cancelled_act(Hotel) | Cancelled_act(Car)). We assert that the advantage of using a single artefact (the Stop instance) to model whether we Pay or not, with the capacity for arbitrarily complex querying on workflow state, is quite evident when compared with the YAWL model. An example of this even in the presented figure is the additional 'bypass' transitions that are needed to capture the choice of whether a booking activity is carried out or not. Such transitions are replicated for each activity. By admitting the possibility that activities may be cancelled as a fundamental aspect of the semantics, there would be no need to explicitly model cancellation (or bypassing) as they have done. We consider YAWL, and Petri net-based approaches generally, to be too low-level for modelling workflow, as we argue in the conclusions to this thesis, in Chapter Twelve.

Further examples of Liesbet workflow models are presented in Chapter Eleven.

In the next chapter, we consider the CCS/PCCS-based characterisation of the Liesbet meta-model. This is the first of the two main approaches, that we have taken in our work, for the formal characterisation of Liesbet.

Chapter 5

CCS-based Characterisations of Liesbet

We now present two formal characterisations of Liesbet using Milner's CCS [78, 80, 79] and Cleaveland's et al's *Prioritised* CCS (PCCS, for short, hereafter) [30, 29].

We selected CCS/PCCS as appropriate formalisms to investigate for two reasons:

1) There has been quite a lot of talk within the BPM community as to whether Petri nets or CCS/π-calculus is better suited for the characterisation of workflow, and specifically the YAWL patterns [122]. While we do not seek to compare these two formalisms at length, by characterising YAWL with CCS we are able to provide a contribution to this debate from one perspective. Note that we do present some points regarding their respective suitability at the end of Chapter Five.
2) The operational semantics of CCS/PCCS (in terms of facilitating compositional specifications of behaviour) should lend themselves quite well to the representation of workflow, and this is a point we seek to investigate.

We start with CCS, presenting a description of how we have used it to provide a characterisation of Liesbet, and how we have used the *Concurrency Workbench for the New Century* (CWB-NC) [11] for the purpose of verifying properties of Liesbet models. We also present a result regarding the completion of CCS-characterised Liesbet workflow models. Then, we briefly discuss the utility of CCS for capturing the semantics of Liesbet, and for facilitating verification. The discussion also motivates the use of PCCS for capturing the semantics of Liesbet. We then proceed with a presentation of some aspects of the PCCS characterisation that we have given to Liesbet, deferring the presentation of remaining aspects to Appendix A, in order to save space. We conclude the chapter with a further discussion.

5.1 Using CCS to Provide an Operational Meaning to Liesbet

We present our CCS-based characterisation for just a subset of Liesbet, which we label Liesbet1. It is possible to give a CCS-based characterisation to the whole of Liesbet, as we discuss in Section 5.7. We define Liesbet1 to consist of the following constructs:

- Basic Activities: A, B, C, ...

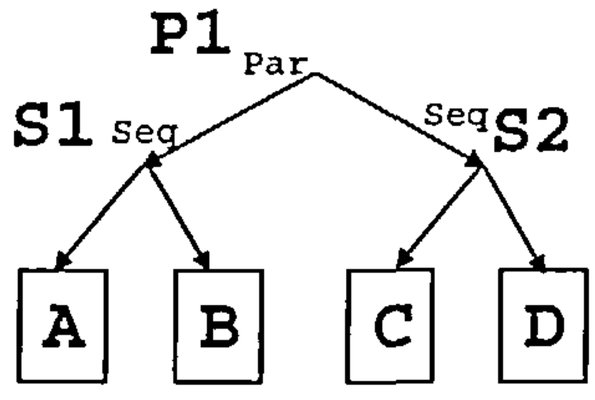


Figure 5.1: Par(Seq(A,B),Seq(C,D)) – Simple Workflow Model

- Sequence: Seq.
- Parallel Split: Par.
- Default and Exclusive Choice: DefaultChoice and Choice.
- Multiple Choice: MultiChoice.
- Empty: Empty.
- FreeChoice: FreeChoice.

5.1.1 Par(Seq(A,B),Seq(C,D)) – A Simple Example

We start by introducing the characterisation of a simple example Liesbet model in CCS, shown in Figure 5.1. The model is Par(Seq(A,B),Seq(C,D)), and is used for illustrative purposes throughout this thesis.

Informally, enactment of this model proceeds as follows. The root instance (P1) is executed, and eventually this execution is propagated to S1, S2, as well as to basic instances A and C. When either one of the basic instances completes (or cancels), the next basic instance in the parent Seq is executed. Whenever both instances in one of the Seqs have finished (i.e. have been completed, or cancelled), the respective Seq instance itself is completed. Once both sequences have completed, the root Par is completed.

A possible representation of the given workflow in CCS could be, simply: a.b.nil | c.d.nil. This adequately captures the intended semantics of the given workflow. Here, we are representing the completion of workflow tasks as CCS transitions. However, there are a number of dispensations that we have to make in representing Liesbet models, which mean that, even for the simplest workflow models, their representation in CCS will not be as simple as this. Chief amongst these are:

- The need to support arbitrary querying of workflow state. This means that we need to maintain agents for activity instances, which can be queried as to the state of their pertaining instances.
- A means of prioritising the evolution of structured instances over basic instances. This is a requirement of any characterisation of Liesbet, as described in Section 3.1. In the absence of an explicit notion of priority in CCS, we need to effect some kind of scheduler for the progression of structured instances over basic instances.

The general form of our CCS characterisation of Liesbet, in light of these requirements, is as follows.

tracker agents | *generic type agents* | *scheduling agents*

Tracker agents maintain the state of activity instances, viz. Initial, Running, Completed, or Cancelled. A tracker agent is an instance of one of the following agent schemas: $\texttt{InitialState}^{n}$, $\texttt{RunningState}^{n,r}$, CompletedState and CancelledState. In these schemas, n is the number of child instances of the pertaining instance and r is a (run-time) count of the number of child instances yet to finish. There will be an instance of one of these agents for every activity instance in a Liesbet model.

Generic type agents maintain the logic for the various Liesbet types, Seq, Par etc. There will be an instance of one of these agents for every structured instance in a Liesbet model. Finally, the scheduling agents ensure the priority, in enactment, of structured instances over basic instances.

Tracker and generic type agents employ a number of channels, which are now briefly enumerated.

- State Channels – used to update and query a tracker agent regarding the state of its pertaining activity instance:
  - exec (resp. comp, canc, pcanc) – used to instruct the instance to move to the Running (resp. Completed, Cancelled, Cancelled) state.
  - runn (resp. cotd, cald, find, ninit) – used to query whether the instance is in the Running (resp. the Completed, the Cancelled, a finished (Completed or Cancelled), or not Initial) state. An answer will follow on the yes or no channel (as appropriate). Note that for our CCS-based characterisation of Liesbet, presented here, we restrict the support for state querying *within synchronisation activities* (i.e. Go and Stop types) to just *monotonic* querying, with no support for distinct queries. Monotonic queries are queries that, once satisfied, are never able to not be immediately satisfiable. This means that the queries within synchronisation types are able to be satisfied gradually, as the workflow model evolves, safe in the knowledge that once we have marked component sub-queries of a query as being satisfied, they will remain so. This also necessitates that the use of negation not be allowed in queries; and we remove the capability of specifying queries on Running and Initial. To compensate, we add the possibility of specifying NotInitial queries, such as NotInitial_act, in Liesbet models. These queries would refer to an instance *not* being in an Initial state. Such querying is supported, by tracker agents, on ninit. Note that we *do* allow Running queries, on runn, at some points within the characterisation of Liesbet, on the basis that their use is necessarily sound.
  - yes, no – for responses to queries.
- Completion channels – used to signal to a parent instance that a child instance has finished:
  - pprec – used by a child instance to signal to its parent that it has finished.
  - prec – used by a parent instance to listen for a child instance to signal that it has finished – corresponds to the pprec channel in child instances

Propagation of execution occurs by virtue of both tracker and generic type agents; whereas propagation of completion occurs solely by virtue of tracker agents.

A representation of the model, presented in Figure 5.1, in CCS (according to our CCS-based characterisation of Liesbet) is as follows.

$\boxed{\texttt{InitialState}^2}$	$\mid \texttt{InitialState}^2$	$\mid \texttt{InitialState}^2$	$\mid$
$\texttt{Par}^2$	$\mid \texttt{Seq}^2$	$\mid \texttt{Seq}^2$	$\mid$
$\texttt{InitialState}^0$	$\mid \texttt{InitialState}^0$	$\mid \texttt{InitialState}^0$	$\mid \texttt{InitialState}^0 \mid \boxed{\ldots}$

Note that the tracker agents for P1, S1 and S2 occupy the first line of the presented process term (from left to right, respectively). The generic type agents occupy the second line. The tracker agents for the basic activity instances, A, B, C and D, occupy the third line (from left to right, respectively). There are also some auxiliary agents, which are represented by ellipsis (...).

In what follows, we present a narrative of one possible way in which this model may be enacted. There are, in fact, several ways. The boxes that surround some of the CCS agents indicate that the agent is involved in the next sequence of transitions to be presented for the narrative.

Briefly, the tracker agent for P1 is put into a Running state, by a synchronisation (with a scheduling agent) on its exec channel, execp1, the result of which is as follows.

$\xrightarrow{\tau(execp1)}$

$\boxed{\texttt{RunningState}^{2,2}}$	$\mid \texttt{InitialState}^2$	$\mid \texttt{InitialState}^2$	$\mid$
$\boxed{\texttt{Par}^2}$	$\mid \texttt{Seq}^2$	$\mid \texttt{Seq}^2$	$\mid$
$\texttt{InitialState}^0$	$\mid \texttt{InitialState}^0$	$\mid \texttt{InitialState}^0$	$\mid \texttt{InitialState}^0 \mid \ldots$

In the generic type agent for P1, i.e. $\texttt{Par}^2$, we check that P1 is in a Running state, by synchronising on runnp1, which is offered by its tracker agent.

$\xrightarrow{\tau(runnp1)}$

$\texttt{RunningState}^{2,2}$	$\mid \boxed{\texttt{InitialState}^2}$	$\mid \boxed{\texttt{InitialState}^2}$	$\mid$
$\boxed{\texttt{Par}^2}$	$\mid \texttt{Seq}^2$	$\mid \texttt{Seq}^2$	$\mid$
$\texttt{InitialState}^0$	$\mid \texttt{InitialState}^0$	$\mid \texttt{InitialState}^0$	$\mid \texttt{InitialState}^0 \mid \ldots$

Given that it is running, the $\texttt{Par}^2$ executes both of its child Seq instances, by synchronising on the exec channels, execs1 and execs2, of their tracker agents.

$\xrightarrow{\tau(execs1)}\ \xrightarrow{\tau(execs2)}$

$\texttt{RunningState}^{2,2}$	$\mid \boxed{\texttt{RunningState}^{2,2}}$	$\mid \boxed{\texttt{RunningState}^{2,2}}$	$\mid$
nil	$\mid \boxed{\texttt{Seq}^2}$	$\mid \boxed{\texttt{Seq}^2}$	$\mid$
$\boxed{\texttt{InitialState}^0}$	$\mid \texttt{InitialState}^0$	$\mid \boxed{\texttt{InitialState}^0}$	$\mid \texttt{InitialState}^0 \mid \ldots$

Then, the generic type agents for the Seq instances identify that they have been set running (by synchronising on runns1 and runns2), and respectively synchronise on the exec channels, execa and execc, of the tracker agents of their first child instances, which are basic instances. Both of the $\texttt{Seq}^2$ agents evolve to a new agent called $\texttt{Seq}^2\texttt{f}$, which is responsible for executing the next child instance as appropriate (but not yet).

$\xrightarrow{\tau(runns1)}$ $\xrightarrow{\tau(execa)}$ $\xrightarrow{\tau(runns2)}$ $\xrightarrow{\tau(execc)}$

RunningState$^{2.2}$	**RunningState$^{2.2}$**	RunningState$^{2.2}$	
nil	Seq^{2}f	Seq^{2}f	
RunningState$^{0.0}$	InitialState^0	RunningState$^{0.0}$	InitialState^0 | ...

Basic instances are completed (resp. cancelled) by synchronising on the comp (resp. canc) channels of their pertaining tracker agents. In what follows, we complete A by synchronising on compa, which is provided by its tracker agent (the first agent on the last line).

$\xrightarrow{\tau(compa)}$ $\xrightarrow{\tau(precs1)}$

RunningState$^{2.2}$	RunningState$^{2.1}$	**RunningState$^{2.2}$**	
nil	**Seq^{2}f**	**Seq^{2}f**	
CompletedState	**InitialState^0**	**RunningState$^{0.0}$**	**InitialState^0** | ...

Whenever an instance (basic or structured) finishes, completion is propagated upwards, through the tracker agents, as far as possible. That is, when an instance finishes, its tracker agent synchronises with its parent tracker agent, on pprec, to indicate as much to the parent. In the presented process term, the tracker agent for A uses the channel precs1. This causes the count of yet-to-finish instances to be decremented, as can be seen for the tracker agent for S1 (the second agent on the first line), which goes from RunningState$^{2.2}$ to RunningState$^{2.1}$.

In the example, once a basic instance in one of the sequences has finished, the pertaining Seq^{2}f agent will execute the next child instance, by synchronising on the exec channel offered by the child instance's tracker agent.

$\xrightarrow{\tau(execb)}$ $\xrightarrow{\tau(compc)}$ $\xrightarrow{\tau(precs2)}$ $\xrightarrow{\tau(execd)}$

RunningState$^{2.2}$	**RunningState$^{2.1}$**	RunningState$^{2.1}$	
nil	nil	nil	
CompletedState	**RunningState$^{0.0}$**	CompletedState	RunningState$^{0.0}$ | ...

When a parent tracker agent has been notified that all of its children have finished, it completes itself and notifies its parent of it finishing. This occurs for S1 as now shown.

$\xrightarrow{\tau(compb)}$ $\xrightarrow{\tau(precs1)}$ $\xrightarrow{\tau(precp1)}$

RunningState$^{2.1}$	CompletedState	**RunningState$^{2.1}$**	
nil	nil	nil	
CompletedState	CompletedState	CompletedState	**RunningState$^{0.0}$** | ...

The synchronisation occurring on precs1 causes a synchronisation on precp1, followed by the tracker agent for S1 moving to CompletedState. Finally, the tracker agent for the Par instance will transition to CompletedState once both of its child Seq instances have finished, and have notified it as much (on precp1), viz.

$\xrightarrow{\tau(compd)}$ $\xrightarrow{\tau(precs2)}$ $\xrightarrow{\tau(precp1)}$

```
CompletedState  | CompletedState  | CompletedState  |
nil             | nil             | nil             |
CompletedState  | CompletedState  | CompletedState  | CompletedState | ...
```

We now proceed to describe how we translate a workflow model written in Liesbet into one in CCS, which will serve to present our CCS characterisation of Liesbet.

5.1.2 Translation of Liesbet1

The translation process comprises two steps:

1. Firstly, we work on the Liesbet1 workflow model definition, which constitutes a tree of activity types, composing the CCS workflow as we work down from the root of the model tree to its leaves. The result of translating a node within the model tree, is a collection of agents (for the node) which run in parallel with the translation of the rest of the workflow model.

2. Finally, we add a few housekeeping agents, which mainly concern the scheduling of structured and basic activity instances.

Step 1

We define a translation function, $\mathcal{M}_{ccs}[\![-]\!]$, which we apply to the root activity type of the workflow model. It recurses its way down the workflow model tree. On translating a node within the model tree, we allocate a collection of *state channels* for each of the node's children (if it has any). In applying $\mathcal{M}_{ccs}[\![-]\!]$, we pass the state channels of the node that we are translating, along with the precompletion channel, prec, of its parent instance.

For convenience, in the following definition, we abbreviate the channel list:

exec, comp, canc, pcanc, ninit, runn, cotd, cald, find, yes, no

by $st\_chs \rightarrow$, corresponding to a collection of *state channels* and

$\mathtt{exec}_i$, $\mathtt{comp}_i$, $\mathtt{canc}_i$, $\mathtt{pcanc}_i$, $\mathtt{ninit}_i$, $\mathtt{runn}_i$, $\mathtt{cotd}_i$, $\mathtt{cald}_i$, $\mathtt{find}_i$, $\mathtt{yes}_i$, $\mathtt{no}_i$

by $st\_chs_i \rightarrow$.

We also abbreviate the relabelling of state channels in an agent:

$$[{}^{execi}/_{exec}, {}^{compi}/_{comp}, {}^{canci}/_{canc}, {}^{pcanci}/_{pcanc}, {}^{niniti}/_{ninit}, {}^{runni}/_{runn}, {}^{cotdi}/_{cotd}, {}^{caldi}/_{cald}, {}^{findi}/_{find}, {}^{yesi}/_{yes}, {}^{noi}/_{no}]$$

by $[SC_i]$ and where both sets are indexed:

$$[{}^{execi}/_{execj}, {}^{compi}/_{compj}, {}^{canci}/_{cancj}, {}^{pcancj}/_{pcancj}, {}^{niniti}/_{ninitj}, {}^{runni}/_{runnj}, {}^{cotdi}/_{cotdj}, {}^{caldi}/_{caldj}, {}^{findi}/_{findj}, {}^{yesi}/_{yesj}, {}^{noi}/_{noj}]$$

by $[SC_{i,j}]$.

Finally, let a,b,c in allocates channel names a,b,c, not used before in the translation process, for use in the subsequent agent definition.

In the following presentation of $\mathcal{M}_{ccs}[\![-]\!]$, we assume that a Liesbet model has been preprocessed in order to replace the use of defined types by *in situ* definitions, see Section 3.1.

- $\mathcal{M}_{ccs}[\![\texttt{Act(\_)}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i) =$
  $\texttt{InitialState}^0[\mathcal{SC}_i, {}^{\texttt{pprec}_i}/_{\texttt{pprec}}]$
- $\mathcal{M}_{ccs}[\![\texttt{Seq(Ch1,...,Chn)}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i)=$
  $\texttt{let } st\_chs_{i1} \rightarrow \texttt{ in } \ldots st\_chs_{in} \rightarrow \texttt{ in let prec}_i \texttt{ in}$
  $\texttt{Seq}^n[\mathcal{SC}_i, \mathcal{SC}_{in,1}, \ldots, \mathcal{SC}_{i1,n}]$
  |
  $\texttt{InitialState}^n[\mathcal{SC}_i, {}^{\texttt{pprec}_i}/_{\texttt{pprec}}, {}^{\texttt{prec}_i}/_{\texttt{prec}}]$
  |
  $\mathcal{M}_{ccs}[\![\texttt{Ch1}]\!](st\_chs_{i1} \rightarrow,\texttt{prec}_i) \mid \ldots \mid \mathcal{M}_{ccs}[\![\texttt{Chn}]\!](st\_chs_{in} \rightarrow,\texttt{prec}_i)$
- $\mathcal{M}_{ccs}[\![\texttt{Par(Ch1,...,Chn)}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i)=$
  $\texttt{let } st\_chs_{i1} \rightarrow \texttt{ in } \ldots st\_chs_{in} \rightarrow \texttt{ in let prec}_i \texttt{ in}$
  $\texttt{Par}^n[\mathcal{SC}_i, \mathcal{SC}_{i1,1}, \ldots, \mathcal{SC}_{in,n}]$
  |
  $\texttt{InitialState}^n[\mathcal{SC}_i, {}^{\texttt{pprec}_i}/_{\texttt{pprec}}, {}^{\texttt{prec}_i}/_{\texttt{prec}}]$
  |
  $\mathcal{M}_{ccs}[\![\texttt{Ch1}]\!](st\_chs_{i1} \rightarrow,\texttt{prec}_i) \mid \ldots \mid \mathcal{M}_{ccs}[\![\texttt{Chn}]\!](st\_chs_{in} \rightarrow,\texttt{prec}_i)$
- $\mathcal{M}_{ccs}[\![\texttt{DefaultChoice(Chg1,...,Chgn,Chc1,...,Chcn,Chd)}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i)=$
  $\texttt{let } st\_chs_{ig1} \rightarrow \texttt{ in } \ldots st\_chs_{ign} \rightarrow \texttt{ in let } st\_chs_{ic1} \rightarrow \texttt{ in } \ldots st\_chs_{icn} \rightarrow \texttt{ in let } st\_chs_{id} \rightarrow$
  $\texttt{in let prec}_i \texttt{ in}$
  $\texttt{DefaultChoice}^n[\mathcal{SC}_i, \mathcal{SC}_{ig1,g1}, \ldots, \mathcal{SC}_{ign,gn}, \mathcal{SC}_{ic1,c1}, \ldots, \mathcal{SC}_{icn,cn}, \mathcal{SC}_{id,d}]$
  |
  $\texttt{InitialState}^{2n+1}[\mathcal{SC}_i, {}^{\texttt{pprec}_i}/_{\texttt{pprec}}, {}^{\texttt{prec}_i}/_{\texttt{prec}}]$
  |
  $\mathcal{M}_{ccs}[\![\texttt{Chg1}]\!](st\_chs_{ig1} \rightarrow,\texttt{prec}_i) \mid \ldots \mid \mathcal{M}_{ccs}[\![\texttt{Chgn}]\!](st\_chs_{ign} \rightarrow,\texttt{prec}_i)$
  |
  $\mathcal{M}_{ccs}[\![\texttt{Chc1}]\!](st\_chs_{ic1} \rightarrow,\texttt{prec}_i) \mid \ldots \mid \mathcal{M}_{ccs}[\![\texttt{Chcn}]\!](st\_chs_{icn} \rightarrow,\texttt{prec}_i)$
  |
  $\mathcal{M}_{ccs}[\![\texttt{Chd}]\!](st\_chs_{id} \rightarrow,\texttt{prec}_i)$
- $\mathcal{M}_{ccs}[\![\texttt{Choice(Chg1,...,Chgn,Chc1,...,Chcn)}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i)=$
  $\texttt{let } st\_chs_{ig1} \rightarrow \texttt{ in } \ldots st\_chs_{ign} \rightarrow \texttt{ in let } st\_chs_{ic1} \rightarrow \texttt{ in } \ldots st\_chs_{icn} \rightarrow \texttt{ in let prec}_i \texttt{ in}$
  $\texttt{Choice}^n[\mathcal{SC}_i, \mathcal{SC}_{ig1,g1}, \ldots, \mathcal{SC}_{ign,gn}, \mathcal{SC}_{ic1,c1}, \ldots, \mathcal{SC}_{icn,cn}]$
  |
  $\texttt{InitialState}^{2n}[\mathcal{SC}_i, {}^{\texttt{pprec}_i}/_{\texttt{pprec}}, {}^{\texttt{prec}_i}/_{\texttt{prec}}]$
  |
  $\mathcal{M}_{ccs}[\![\texttt{Chg1}]\!](st\_chs_{ig1} \rightarrow,\texttt{prec}_i) \mid \ldots \mid \mathcal{M}_{ccs}[\![\texttt{Chgn}]\!](st\_chs_{ign} \rightarrow,\texttt{prec}_i)$
  |
  $\mathcal{M}_{ccs}[\![\texttt{Chc1}]\!](st\_chs_{ic1} \rightarrow,\texttt{prec}_i) \mid \ldots \mid \mathcal{M}_{ccs}[\![\texttt{Chcn}]\!](st\_chs_{icn} \rightarrow,\texttt{prec}_i)$
- $\mathcal{M}_{ccs}[\![\texttt{MultiChoice(Chg1,...,Chgn,Chc1,...,Chcn)}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i)=$
  $\texttt{let } st\_chs_{ig1} \rightarrow \texttt{ in } \ldots st\_chs_{ign} \rightarrow \texttt{ in let } st\_chs_{ic1} \rightarrow \texttt{ in } \ldots st\_chs_{icn} \rightarrow \texttt{ in let prec}_i \texttt{ in}$
  $\texttt{MultiChoice}^n[\mathcal{SC}_i, \mathcal{SC}_{ig1,g1}, \ldots, \mathcal{SC}_{ign,gn}, \mathcal{SC}_{ic1,c1}, \ldots, \mathcal{SC}_{icn,cn}]$
  |
  $\texttt{InitialState}^{2n}[\mathcal{SC}_i, {}^{\texttt{pprec}_i}/_{\texttt{pprec}}, {}^{\texttt{prec}_i}/_{\texttt{prec}}]$

|

$\mathcal{M}_{ccs}[\![\texttt{Chg1}]\!](st\_chs_{\texttt{ig1}} \rightarrow, \texttt{prec}_i)$ | ... | $\mathcal{M}_{ccs}[\![\texttt{Chgn}]\!](st\_chs_{\texttt{ign}} \rightarrow, \texttt{prec}_i)$

|

$\mathcal{M}_{ccs}[\![\texttt{Chc1}]\!](st\_chs_{\texttt{ic1}} \rightarrow, \texttt{prec}_i)$ | ... | $\mathcal{M}_{ccs}[\![\texttt{Chcn}]\!](st\_chs_{\texttt{icn}} \rightarrow, \texttt{prec}_i)$

- $\mathcal{M}_{ccs}[\![\texttt{Empty}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i)=$

  $\texttt{Empty}[SC_i]$

  |

  $\texttt{InitialState}^0[SC_i, \ ^{\texttt{pprec}_i}/_{\texttt{pprec}}]$

- $\mathcal{M}_{ccs}[\![\texttt{FreeChoice}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i)=$

  $\texttt{FreeChoice}[SC_i]$

  |

  $\texttt{InitialState}^0[SC_i, \ ^{\texttt{pprec}_i}/_{\texttt{pprec}}]$

For the definition of $\mathcal{M}_{ccs}[\![-]\!]$, for the types presented here, we note:

- $\mathcal{M}_{ccs}[\![-]\!]$, for basic instances, will output a single *state tracker agent* (which starts life as $\texttt{InitialState}^0$).

- $\mathcal{M}_{ccs}[\![-]\!]$, for structured types, such as Seq, will output an $\texttt{InitialState}^n$ tracker agent (where n is the number of children of the type), as well as an agent which will effect the logic of the type (a generic type agent), such as $\texttt{Seq}^n$, and the output from translating the children (if extant).

The definitions of the various tracker and generic type agents are now presented and explained. Note that as we have used CWB-NC [11] to simulate workflow models, the definitions are presented in the input syntax of the workbench.

The general form of a tracker agent will be to:

- To accept a cancellation demand on canc and evolve to the CancelledState agent having cancelled its children (if any).

- To accept a cancellation demand from its parent tracker agent on pcanc and evolve to the CancelledState agent having cancelled its children (if any). This applies to just Initial and Running state tracker agents.

- Indicate appropriate yes or no answers to queries regarding its state, and then to evolve back to the same agent.

For specific tracker agents, there may be some additional behaviour that it admits, as will be made clear.

For any particular CCS workflow model, there will exist an agent definition (output by the translator) of (the tracker agent) $\texttt{InitialState}^n$ for every distinct number, n, of child types that an activity type has within the model. In the case of the model presented in Figure 5.1, there are types with two and zero children – hence the definitions of $\texttt{InitialState}^2$ and $\texttt{InitialState}^0$.

```
proc InitialState0 =
   canc.'pprec.pyes.'yes.CancelledState +
   pcanc.'yes.CancelledState +
```

```
    exec.RunningState0_0 +
    runn.'no.InitialState0 +
    ninit.'no.InitialState0 +
    cotd.'no.InitialState0 +
    cald.'no.InitialState0 +
    find.'no.InitialState0

proc InitialState2 =
    canc.'pprec.pyes.'pcanc1.yes1.'pcanc2.yes2.'yes.CancelledState +
    pcanc.'pcanc1.yes1.'pcanc2.yes2.'yes.CancelledState +
    exec.RunningState2_2 +
    runn.'no.InitialState2 +
    ninit.'no.InitialState2 +
    cotd.'no.InitialState2 +
    cald.'no.InitialState2 +
    find.'no.InitialState2
```

$\texttt{InitialState}^n$ accepts a synchronisation on:

- canc and becomes CancelledState – which corresponds to the state Cancelled, once we have signalled on pprec to indicate to the parent instance that the instance in question has moved to a finished state, and received an acknowledgement back on pyes, then, signalled to the instance's children (if extant) that they should cancel (on $\texttt{pcanc}_i$) and have responded to the initial synchronisation on yes. We elaborate further with respect to pprec in the description of the $\texttt{RunningState}^{n\_r}$ agents.

  Note that the acknowledgement on yes is appropriate, as we need to *force* the described intermediate steps before we can allow the agent initiating the cancellation to continue.

- pcanc, signifying that the parent instance's tracker agent has been cancelled, and that the instance should itself move to a cancelled state after cancelling its own children.

  pcanc is used by a tracker agent to signal to the tracker agents of the pertaining instance's children that they too should cancel. canc, on the other hand, is used within generic type agents. The difference being that, in response to canc, we signal to the parent state tracking instance that we have finished. In the case of pcanc, as the cancellation is initiated by the parent, there is no need to do this.

- exec and becomes $\texttt{RunningState}^{n\_n}$ – which corresponds to the state Running.

- ninit, runn, cotd, cald, find and becomes $\texttt{InitialState}^n$ again – querying whether its corresponding instance is in a not Initial, Running, Completed, Cancelled, or finished (i.e. Completed or Cancelled) state. The answer is no to all – as signalled.

There will exist agent definitions (output by the translator) of $\texttt{RunningState}^{n\_r}$ for every distinct number, n, of child types that an activity type has within the model, and for all r such that $0 < r \leq n$, if $n > 0$, and for $r = 0$, if $n = 0$. For the same workflow model, then, we have copies of RunningState2_2, RunningState2_1 and RunningState0_0.

```
proc RunningState0_0 =
```

```
    canc.'pprec.pyes.'yes.CancelledState +
    pcanc.'yes.CancelledState +
    comp.'pprec.pyes.'yes.CompletedState +
    ninit.'yes.RunningState0_0 +
    runn.'yes.RunningState0_0 +
    cotd.'no.RunningState0_0 +
    cald.'no.RunningState0_0 +
    find.'no.RunningState0_0

proc RunningState2_1 =
    canc.'pprec.pyes.'pcanc1.yes1.'pcanc2.yes2.'yes.CancelledState +
    pcanc.'pcanc1.yes1.'pcanc2.yes2.'yes.CancelledState +
    prec.'pprec.pyes.'yes.CompletedState +
    ninit.'yes.RunningState2_1 +
    runn.'yes.RunningState2_1 +
    cotd.'no.RunningState2_1 +
    cald.'no.RunningState2_1 +
    find.'no.RunningState2_1

proc RunningState2_2 =
    canc.'pprec.pyes.'pcanc1.yes1.'pcanc2.yes2.'yes.CancelledState +
    pcanc.'pcanc1.yes1.'pcanc2.yes2.'yes.CancelledState +
    prec.'yes.RunningState2_1 +
    ninit.'yes.RunningState2_2 +
    runn.'yes.RunningState2_2 +
    cotd.'no.RunningState2_2 +
    cald.'no.RunningState2_2 +
    find.'no.RunningState2_2
```

When running, instances of childless types will have copies of $\texttt{RunningState}^{0,0}$ as their tracker agents. This agent constant offers the possibility of it transitioning to `CompletedState`, by accepting a synchronisation on `comp`, reflecting the completion of its pertaining activity instance. When a synchronisation on `comp` occurs, the parent is instructed to perform a `precompletion` step, as described below.

Running instances of child-bearing types will have copies of $\texttt{RunningState}^{n,r}$ as their tracker agents, where $n \geq 1$ and $r \geq 1$. The occurrence of a child finishing will cause a synchronisation on `prec` to occur. This causes a `precompletion` step to take place. When r, the count of child instances yet-to-finish, is greater than one, performing a `precompletion` step means evolving to the agent $\texttt{RunningState}^{n,r-1}$. When $r = 1$, this means reporting completion to its respective parent tracker agent on `pprec`, and evolving to the agent `CompletedState`.

There will also exist definitions of `CompletedState` and `CancelledState`, which report that their associated instances are in `Completed` and `Cancelled` states, respectively. Note that, in `CancelledState`, we support a synchronisation on `exec`, as attempting to execute a cancelled instance is allowed, albeit it has no effect. This would occur if, for instance, a `Seq` instance had had some of its children cancelled by a `CancelActivity` instance (see Section 3.1.16).

```
proc CompletedState =
```

```
        canc.CompletedState +
        ninit.'yes.CompletedState +
        runn.'no.CompletedState +
        cotd.'yes.CompletedState +
        cald.'no.CompletedState +
        find.'yes.CompletedState

    proc CancelledState =
        canc.CancelledState +
        ninit.'yes.CancelledState +
        runn.'no.CancelledState +
        cotd.'no.CancelledState +
        cald.'yes.CancelledState +
        exec.'yes.CancelledState +
        find.'yes.CancelledState
```

We now present the definitions of the various agents for generic activity types, starting with Seq. For Seq, there will be a $\texttt{Seq}^n$ agent, and $\texttt{Seq}^r\texttt{f}$ agents, for every distinct number, n, of children of Seq types in a model and for all r such that $2 \geq r \geq n$, whose definitions are output by the translator.

A $\texttt{Seq}^n$ agent first ascertains that it is not cancelled and that it is running, then it executes its first child instance, labelled n, and then transitions to a "finishing" agent, $\texttt{Seq}^n\texttt{f}$, which effects the remaining logic. Note that child instances of a sequence are numbered in decreasing order, so the first to be executed is n, the second $n-1$, and so on. This convention simplifies the definition of the agents.

$\texttt{Seq}^n\texttt{f}$ waits for the first instance to finish, and then executes the next child instance. After that, the agent $\texttt{Seq}^{n-1}\texttt{f}$ is exposed. And so on, until we reach $\texttt{Seq}^2\texttt{f}$, whereon, we wait for the penultimate instance to finish, and then execute the last. Following that, we expose the agent Idle, to effect idling, which is necessary for scheduling purposes. We explain this further in *Step 2* of the translation process, where we also explain the use of the channels lock, idle, prog and reset.

```
    proc Seq2 =
        lock.'cald.(yes.'idle.reset.Idle + no.'runn.(yes.'exec2.'prog.Seq2f +
                                                     no.'idle.reset.Seq2))

    proc Seq2f =
        lock.'find2.(yes2.'exec1.'prog.Idle + no2.'idle.reset.Seq2f)

    proc Idle =
        lock.'idle.reset.Idle
```

For Par, there will be a $\texttt{Par}^n$ agent for every distinct number, n, of children of Par types in a model, whose definitions are output by the translator. We present the agent definition for the case where n is 2. A $\texttt{Par}^n$ agent will simply execute all of its children together (within the same execution window, see *Step2*).

```
proc Par2 =
   lock.'cald.(yes.'idle.reset.Idle + no.'runn.(yes.'exec1.'exec2.'prog.Idle +
                                                 no.'idle.reset.Par2))
```

For DefaultChoice, there will be DefaultChoice^n, DefaultChoice^{n}f and DefaultChoice^{n}fComp agents, for every distinct number, n, of *continuation* child types (not including the default) of DefaultChoice types in a model, whose definitions are output by the translator. We present the agent definitions for the case where n is 2.

In DefaultChoice^{2}f, which is exposed once we ascertain that the choice activity type has been put into a running state (by its parent) and the *guard instances* of the choice type have been set running, we check to see whether any of the guard instances have completed. If so, we expose DefaultChoice^{2}fComp, which serves to execute a continuation instance pertaining to one of the completed guard instances. It also cancels the remaining continuation instances (including the default instance). If, on the other hand, none of the guard instances have completed; but, commensurately, none of them are running either, then all of them must have been cancelled. In this case, we execute the default continuation instance. If none of these possibilities obtain, we expose another copy of DefaultChoice^{2}f.

```
proc DefaultChoice2 =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'runn.(yes.'execg1.'execg2.'prog.DefaultChoice2f +
                         no.'idle.reset.DefaultChoice2))

proc DefaultChoice2f =
   lock.'cotdg1.(yesg1.DefaultChoice2fComp +
                 nog1.'cotdg2.(yesg2.DefaultChoice2fComp +
                               nog2.'runng1.(yesg1.'idle.reset.DefaultChoice2f +
                                             nog1.'runng2.(yesg2.'idle.reset.DefaultChoice2f +
                                                           nog2.'cancc1.yesc1.'cancc2.yesc2.
                                                           'execd.'prog.Idle))))

proc DefaultChoice2fComp =
   ( 'cotdg1.(yesg1.('win.'execc1.'tidy.nil + 'lose.'cancc1.yesc1.'tidy.nil) +
              nog1.'lose.'cancc1.yesc1.'cancg1.yesg1.'tidy.nil)
   | 'cotdg2.(yesg2.('win.'execc2.'tidy.nil + 'lose.'cancc2.yesc2.'tidy.nil) +
              nog2.'lose.'cancc2.yesc2.'cancg2.yesg2.'tidy.nil)
   | win.tidy.lose.tidy.'cancd.yesd.'prog.Idle)\{win,lose,tidy}
```

For Choice^n, which has no default continuation instance, we do much the same. However, in the case that all guard instances get cancelled, we cancel the choice instance, as shown.

```
proc Choice2 =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'runn.(yes.'execg1.'execg2.'prog.Choice2f +
                         no.'idle.reset.Choice2))

proc Choice2f =
   lock.'cotdg1.(yesg1.Choice2fComp +
```

```
                    nog1.'cotdg2.(yesg2.Choice2fComp +
                                  nog2.'runng1.(yesg1.'idle.reset.Choice2f +
                                                nog1.'runng2.(yesg2.'idle.reset.Choice2f +
                                                nog2.'canc.yes.'prog.Idle))))

proc Choice2fComp =
   ( 'cotdg1.(yesg1.('win.'execc1.'tidy.nil + 'lose.'cancc1.yesc1.'tidy.nil) +
             nog1.'lose.'cancc1.yesc1.'cancg1.yesg1.'tidy.nil)
   | 'cotdg2.(yesg2.('win.'execc2.'tidy.nil + 'lose.'cancc2.yesc2.'tidy.nil) +
             nog2.'lose.'cancc2.yesc2.'cancg2.yesg2.'tidy.nil)
   | win.tidy.lose.tidy.'prog.Idle)\{win,lose,tidy}
```

For MultiChoice, there will be MultiChoice^n and MultiChoice^{n}f agents, for every distinct number, n, of *continuation* child types of MultiChoice types in a model, whose definitions are output by the translator. We present the agent definitions for the case where n is 2.

For MultiChoice^n, once it is running, and we have executed its guard instances, we proceed to MultiChoice^{n}f, whereon, we check for continuation instances that are still in the Initial state. For those that are, we check their guard instances and act appropriately – for those which have now completed successfully, we execute their corresponding continuation instances, for those which have been cancelled, we cancel their corresponding continuation instances, and for those which are still running, we do nothing.

```
proc MultiChoice2 =
    lock.'cald.(yes.'idle.reset.Idle +
                no.'runn.(yes.'execg1.'execg2.'prog.MultiChoice2f +
                          no.'idle.reset.MultiChoice2))

proc MultiChoice2f =
    lock.('ninitc1.(yesc1.'done.nil +
                    noc1.'cotdg1.(yesg1.'execc1.'work.'done.nil +
                                  nog1.'caldg1.(yesg1.'cancc1.yesc1.'work.'done.nil +
                                                nog1.'done.nil))
          |
           'ninitc2.(yesc2.'done.nil +
                    noc2.'cotdg2.(yesg2.'execc2.'work.'done.nil +
                                  nog2.'caldg2.(yesg2.'cancc2.yesc2.'work.'done.nil +
                                                nog2.'done.nil))
          |
           done.done.'findg1.(yesg1.'findg2.(yesg2.'report.Idle +
                                             nog2.'report.MultiChoice2f) +
                              nog1.'report.MultiChoice2f)
    |
      work.(report.'prog.nil | work.nil) + report.'idle.reset.nil
    )\{work,report,done}
```

Finally, the definitions of FreeChoice and Empty are presented. In the first case, we may either complete or cancel the instance – a non-deterministic choice. In the second case, we trivially

complete the instance.

```
proc FreeChoice =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'runn.(yes.('comp.yes.'prog.Idle + 'canc.yes.'prog.Idle) +
                         no.'idle.reset.FreeChoice))

proc Empty =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'runn.(yes.'comp.yes.'prog.Idle +
                         no.'idle.reset.Empty))
```

Step 2

In step 2, we add three agents to run at the top-level:

- A single instance of $\texttt{Basics}^b$, where b is the number of *basic* activity instances in the workflow model. This agent is called the *basics arbiter* because it arbitrates the completion of a single basic instance. In the case of the model presented in Figure 5.1, the value of b would be four, and the agent constant that we would add would look much as follows.

  ```
  Basics4[comp_2/comp1, yes_2/yes1, comp_3/comp2, yes_3/yes2,
          comp_5/comp3, yes_5/yes3, comp_6/comp4, yes_6/yes4]
  ```

  Here, we relabel the comp and yes channels to be those of the basic activity instances. The definition of Basics4 is as follows.

  ```
  proc Basics4 =
       bas.('comp1.yes1.'bas.Basics4 + 'comp2.yes2.'bas.Basics4 +
            'comp3.yes3.'bas.Basics4 + 'comp4.yes4.'bas.Basics4)
  ```

  We complete exactly one running[1] basic activity instance, deferring completion of any others that are running.

- A single instance of $\texttt{Scheduler}^s$, where s is the number of *structured* activity types in the workflow model. For a model with three structured activity instances, the agent added would be $\texttt{Scheduler}^3$, prefixed by an exec action, to be carried out on the root instance of the workflow model, viz.

  ```
  'exec_0.Scheduler3
  ```

  The $\texttt{Scheduler}^s$ agent effects a lock on execution rights for (the generic type agents pertaining to) structured activity instances. Only one instance may hold the lock at any time; and instances may only progress when they hold the lock. $\texttt{Scheduler}^s$ allows any structured activity instance to claim such a right, which the instance will later yield. In the following example definitions, we consider the case where m is 3, i.e. there are three structured activity instances in the model.

[1] Communication on comp is not offered by tracker agents if the instance is not running.

```
proc Scheduler3 =
   'find_0.(yes_0.'rfind.nil +
           no_0.'lock.(idle.'lock.(idle.'lock.(idle.
                                   'bas.bas.'reset.'reset.'reset.Scheduler3 +
                                            prog.'reset.'reset.Scheduler3) +
                                   prog.'reset.Scheduler3) +
                           prog.Scheduler3))
```

In Scheduler^3, we first check whether the root instance of the workflow model has finished. If it has finished, we expose the action 'rfind (explained further in Section 5.1.4), and, thereafter, evolve to nil. Otherwise, we signal on 'lock to indicate to the structured activity instances that one of them may claim the lock (on execution rights). Then, the instance that claims the lock, signals on 'prog to indicate that it has made progress, or signals on 'idle to say that it has not. If all instances signal on 'idle, then none of them can currently progress. It is then appropriate to try to complete a single basic activity instance. We do so by synchronising on 'bas with the Basics^b agent, which causes the latter agent to expose logic to effect the completion of a basic instance. When that has occurred, a further synchronisation takes place on bas to hand control back to the scheduler. We then reset all of the structured instances so that they may attempt to reclaim execution rights, and re-expose the Scheduler^3 agent. If progress was made by a structured instance, previously, then we signal on reset just as many times as there were structured instances that reported idle. Then, we re-expose the Scheduler^3 agent, in order that we may try the full set of structured instances again (before trying the basics, if that becomes appropriate).

- Finally, we add the agent, pprec.'pyes.nil, to run in parallel at the top-level. This effects a synchronisation on pprec with the root instance's tracker agent, which will be a progressed copy of RunningState^{n,0} (and returns acknowledgment on pyes). Essentially, this allows the root instance's tracker agent to signal that it has finished and, thereafter, evolve to CompletedState, or CancelledState. All other activity instances signal to their respective parent instances. As the root instance has no parent, we need to make this dispensation.

5.1.3 A Complete Example

We now present the output of $\mathcal{M}_{ccs}[\![-]\!]$, in full, for the workflow model shown in Figure 5.1, which has the definition in Liesbet: Par(Seq(A,B), Seq(C,D)). Note that the file has been generated automatically using the support that we provide in our verification framework for Liesbet, which is documented in Section 10.3.

```
*****************************************
* CCS Verification Run ***************
* # 0
* Generated from: file:samples/LiesbetTest.liesbet
* On: Fri Jul 14 11:52:13 BST 2006

proc InitialState0 =
```

```
   canc.'pprec.pyes.'yes.CancelledState +
   pcanc.'yes.CancelledState +
   exec.RunningState0_0 +
   runn.'no.InitialState0 +
   ninit.'no.InitialState0 +
   cotd.'no.InitialState0 +
   cald.'no.InitialState0 +
   find.'no.InitialState0

proc InitialState2 =
   canc.'pprec.pyes.'pcanc1.yes1.'pcanc2.yes2.'yes.CancelledState +
   pcanc.'pcanc1.yes1.'pcanc2.yes2.'yes.CancelledState +
   exec.RunningState2_2 +
   runn.'no.InitialState2 +
   ninit.'no.InitialState2 +
   cotd.'no.InitialState2 +
   cald.'no.InitialState2 +
   find.'no.InitialState2

proc RunningState0_0 =
   canc.'pprec.pyes.'yes.CancelledState +
   pcanc.'yes.CancelledState +
   comp.'pprec.pyes.'yes.CompletedState +
   ninit.'yes.RunningState0_0 +
   runn.'yes.RunningState0_0 +
   cotd.'no.RunningState0_0 +
   cald.'no.RunningState0_0 +
   find.'no.RunningState0_0

proc RunningState2_1 =
   canc.'pprec.pyes.'pcanc1.yes1.'pcanc2.yes2.'yes.CancelledState +
   pcanc.'pcanc1.yes1.'pcanc2.yes2.'yes.CancelledState +
   prec.'pprec.pyes.'yes.CompletedState +
   ninit.'yes.RunningState2_1 +
   runn.'yes.RunningState2_1 +
   cotd.'no.RunningState2_1 +
   cald.'no.RunningState2_1 +
   find.'no.RunningState2_1

proc RunningState2_2 =
canc.'pprec.pyes.'pcanc1.yes1.'pcanc2.yes2.'yes.CancelledState +
pcanc.'pcanc1.yes1.'pcanc2.yes2.'yes.CancelledState +
prec.'yes.RunningState2_1 +
ninit.'yes.RunningState2_2 +
runn.'yes.RunningState2_2 +
cotd.'no.RunningState2_2 +
cald.'no.RunningState2_2 +
```

```
   find.'no.RunningState2_2

proc CompletedState =
   canc.CompletedState +
   ninit.'yes.CompletedState +
   runn.'no.CompletedState +
   cotd.'yes.CompletedState +
   cald.'no.CompletedState +
   find.'yes.CompletedState

proc CancelledState =
   canc.CancelledState +
   ninit.'yes.CancelledState +
   runn.'no.CancelledState +
   cotd.'no.CancelledState +
   cald.'yes.CancelledState +
   exec.'yes.CancelledState +
   find.'yes.CancelledState

proc Basics4 =
   bas.(
   'comp1.yes1.'bas.Basics4 +
   'comp2.yes2.'bas.Basics4 +
   'comp3.yes3.'bas.Basics4 +
   'comp4.yes4.'bas.Basics4)

proc Idle =
   lock.'idle.reset.Idle

proc Scheduler3 =
   'find_0.(yes_0.'rfind.nil + no_0.'lock.(idle.'lock.(idle.'lock.(idle.'bas.bas.
                                                              'reset.'reset.'reset.Scheduler3 +
                                                          prog.'reset.'reset.Scheduler3) +
                                                  prog.'reset.Scheduler3) +
                                          prog.Scheduler3))

proc Seq2 =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'runn.(yes.'exec2.'prog.Seq2f +
                         no.'idle.reset.Seq2))

proc Seq2f =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'find2.(yes2.'exec1.'prog.Idle +
                          no2.'idle.reset.Seq2f))
```

```
proc Par2 =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'runn.(yes.'exec1.'exec2.'prog.Idle +
                         no.'idle.reset.Par2))

proc Workflow0 =
   (
***Instance:0:P1
   InitialState2[yes_0/yes, no_0/no, runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, ninit_0/ninit, comp_0/comp, pcanc_0/pcanc, canc_0/canc, exec_0/exec,
      prec_0/prec,
   pcanc_1/pcanc1, yes_1/yes1, pcanc_4/pcanc2, yes_4/yes2] |

   Par2[yes_0/yes, no_0/no, runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, ninit_0/ninit, comp_0/comp, pcanc_0/pcanc, canc_0/canc, exec_0/exec,
      exec_1/exec1, exec_4/exec2] |

***Instance:1:S1
   InitialState2[yes_1/yes, no_1/no, runn_1/runn, cald_1/cald, cotd_1/cotd,
      find_1/find, ninit_1/ninit, comp_1/comp, pcanc_1/pcanc, canc_1/canc, exec_1/exec,
      prec_1/prec, prec_0/pprec, yes_0/pyes,
   pcanc_2/pcanc1, yes_2/yes1, pcanc_3/pcanc2, yes_3/yes2] |

   Seq2[yes_1/yes, no_1/no, runn_1/runn, cald_1/cald, cotd_1/cotd,
      find_1/find, ninit_1/ninit, comp_1/comp, pcanc_1/pcanc, canc_1/canc, exec_1/exec,
      exec_2/exec2, find_2/find2, yes_2/yes2, no_2/no2, exec_3/exec1] |

***Instance:2:A
   InitialState0[yes_2/yes, no_2/no, runn_2/runn, cald_2/cald, cotd_2/cotd,
      find_2/find, ninit_2/ninit, comp_2/comp, pcanc_2/pcanc, canc_2/canc, exec_2/exec,
      prec_1/pprec, yes_1/pyes] |

***Instance:3:B
   InitialState0[yes_3/yes, no_3/no, runn_3/runn, cald_3/cald, cotd_3/cotd,
      find_3/find, ninit_3/ninit, comp_3/comp, pcanc_3/pcanc, canc_3/canc, exec_3/exec,
      prec_1/pprec, yes_1/pyes] |

***Instance:4:S2
   InitialState2[yes_4/yes, no_4/no, runn_4/runn, cald_4/cald, cotd_4/cotd,
      find_4/find, ninit_4/ninit, comp_4/comp, pcanc_4/pcanc, canc_4/canc, exec_4/exec,
      prec_4/prec, prec_0/pprec, yes_0/pyes,
   pcanc_5/pcanc1, yes_5/yes1, pcanc_6/pcanc2, yes_6/yes2] |

   Seq2[yes_4/yes, no_4/no, runn_4/runn, cald_4/cald, cotd_4/cotd,
      find_4/find, ninit_4/ninit, comp_4/comp, pcanc_4/pcanc, canc_4/canc, exec_4/exec,
      exec_5/exec2, find_5/find2, yes_5/yes2, no_5/no2, exec_6/exec1] |
```

```
***Instance:5:C
   InitialState0[yes_5/yes, no_5/no, runn_5/runn, cald_5/cald, cotd_5/cotd,
      find_5/find, ninit_5/ninit, comp_5/comp, pcanc_5/pcanc, canc_5/canc, exec_5/exec,
      prec_4/pprec, yes_4/pyes] |

***Instance:6:D
   InitialState0[yes_6/yes, no_6/no, runn_6/runn, cald_6/cald, cotd_6/cotd,
      find_6/find, ninit_6/ninit, comp_6/comp, pcanc_6/pcanc, canc_6/canc, exec_6/exec,
      prec_4/pprec, yes_4/pyes] |

   Basics4[comp_2/comp1, yes_2/yes1, comp_3/comp2, yes_3/yes2,
         comp_5/comp3, yes_5/yes3, comp_6/comp4, yes_6/yes4] |
'exec_0.Scheduler3 | pprec.'pyes.nil

)\{
   runn_0, cald_0, cotd_0, find_0, ninit_0, comp_0, pcanc_0, canc_0, exec_0, prec_0, yes_0, no_0,
   runn_1, cald_1, cotd_1, find_1, ninit_1, comp_1, pcanc_1, canc_1, exec_1, prec_1, yes_1, no_1,
   runn_2, cald_2, cotd_2, find_2, ninit_2, comp_2, pcanc_2, canc_2, exec_2, prec_2, yes_2, no_2,
   runn_3, cald_3, cotd_3, find_3, ninit_3, comp_3, pcanc_3, canc_3, exec_3, prec_3, yes_3, no_3,
   runn_4, cald_4, cotd_4, find_4, ninit_4, comp_4, pcanc_4, canc_4, exec_4, prec_4, yes_4, no_4,
   runn_5, cald_5, cotd_5, find_5, ninit_5, comp_5, pcanc_5, canc_5, exec_5, prec_5, yes_5, no_5,
   runn_6, cald_6, cotd_6, find_6, ninit_6, comp_6, pcanc_6, canc_6, exec_6, prec_6, yes_6, no_6,
   bas, pprec, pyes, lock, idle, prog, reset}
```

5.1.4 Model Checking CCS Characterised Liesbet1 with Concurrency Workbench

For the CCS characterisation of Liesbet, we describe a single, simple test. We wish to check that along all enactment paths, the root instance will reach a finished state (either Completed, or Cancelled); and thus the workflow model as a whole will complete successfully along all paths. This is a key property to check in verifying the soundness of workflow models, as described in Section 7.1.

In the definition of the scheduling agent, Scheduler^s, once we have identified that the root instance has reached a finished state, we signal on the channel rfind. For the time being, this is the only unrestricted channel of a CCS Liesbet model. The test that we write is the simple modal-mu formula: μX.<->tt$\wedge$[-'rfind]X, which is written, for use in CWB-NC, as follows. Note that we name the proposition that we are testing cotd, as the proposition holding signifies that the model *completes* successfully along all paths.

```
prop cotd =
   min X = <->tt ∧ [-'rfind]X
```

This says that along all enactment paths the action rfind must eventually occur. The output from running this test under CWB-NC for the example model is presented.

```
>cwb-nc.bat ccs
```

```
cwb-nc.bat ccs
Currently supported languages are : ccs, pccs, sccs, tccs, csp, lotos

The Concurrency Workbench of the New Century
(Version 1.2 --- June, 2000)

cwb-nc> load test.ccs
Execution time (user,system,gc,real):(0.008,0.000,0.004,0.012)
cwb-nc> load test.mu
Execution time (user,system,gc,real):(0.000,0.000,0.000,0.002)
cwb-nc> chk Workflow0 find
Invoking alternation-free model checker.
Building automaton...
........
States: 833
Transitions: 977
Done building automaton.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(1.872,0.048,0.372,1.919)
cwb-nc>
```

5.1.5 Model Equivalence for CCS-characterised Liesbet1

The question of *when two workflows are equivalent* is an important issue in the study of workflow. As reported in [57], it may be non-trivial to arrive at a formalisation of equivalence for some approaches to workflow representation. A key issue lies with how to treat *internal actions* – those actions which progress the model but are not concerned with the fulfilment of (basic) activity instances.

Prioritising the execution of internal actions (i.e., in the case of Liesbet, progressing structured activities over basic activities) helps to resolve this issue (in part), and some examples, presented in [57], would not occur under this assumption.

Orthogonally, we may consider the observable behaviour of a workflow model to be sufficient for defining equivalence. This is the view taken in many approaches, such as [64]. There, two workflows are considered to be equivalent iff they are *observationally equivalent* (as defined by Milner for CCS [78]). Activity completions are considered to be the only observable actions; and there is an additional requirement that all enactment paths within the workflow models must lead to completion of the workflow instance.

The definition of *Observational Equivalence* requires some additional notation. The transition $E \stackrel{\alpha}{\Rightarrow} E'$, for $\alpha \in Act$, means that E may transition to E' through an α-transition prefixed and postfixed by ≥ 0 $\tau-$transitions. That is, $E \stackrel{\alpha}{\Rightarrow} E'$ if $E(\stackrel{\tau}{\rightarrow})^p(\stackrel{\alpha}{\rightarrow})(\stackrel{\tau}{\rightarrow})^q$, where $p, q \geq 0$. Also, for $t \in Act*$, $\hat{t} \in \mathcal{L}*$ is the sequence gained by deleting all occurrences of τ from t.

Observational Equivalence (or Weak Bisimilarity), from [78], $\approx$, is the largest symmetric relation such that $E \approx F$ iff whenever $E \stackrel{\alpha}{\rightarrow} E'$ then $F \stackrel{\hat{\alpha}}{\Rightarrow} \approx E'$.

Elaborating, two CCS agents are observationally equivalent iff, whenever either agent can make an α-transition, the other agent can perform a sequence of transitions $f(\alpha)$; and the agents which

result (from carrying out these transitions) are themselves observationally equivalent. If α is a non-τ transition, $f(\alpha)$ is the same non-τ-transition, prefixed and postfixed by ≥ 0 τ-transitions. If α is a τ-transition, $f(\alpha)$ is ≥ 0 τ-transitions.

In order to be able to define an appropriate notion of equivalence between Liesbet models, we need to make visible transitions pertaining to the completion of basic activities. To this end, we augment `Basics`^{b} as follows (here, for the case that n is 4). We add an additional output, for each completion option `compi`, on a visible channel `eyesi`. This will make the completion option (which is only offered by the corresponding tracker agent if the pertaining instance is running) visible in assessing observational equivalence.

```
proc Basics4 =
   bas.(
     comp1.yes1.'eyes1.bas.Basics4 +
     comp2.yes2.'eyes2.bas.Basics4 +
     comp3.yes3.'eyes3.bas.Basics4 +
     comp4.yes4.'eyes4.bas.Basics4)
```

Thus, the only transitions made visible to the environment are relabelled `eyesi` transitions, for basic activity instances, and a `rfind` transition to indicate that the workflow model is finished. In this context, we may define two CCS-characterised `Liesbet` models to be *model equivalent* iff they are observationally equivalent (according to these offered transitions).

The concept of model equivalence is demonstrated in the following examples.

`Liesbet` Model Equivalence, Example 1: v Strong Equivalence

Observational equivalence is a *weaker* notion than *strong equivalence*. For strong equivalence, we do not abstract away from τ-actions. An example that highlights this distinction is the following simple one.

Let `Liesbet` Model `Workflow0` be defined as: `A`, and `Liesbet` Model `Workflow1` be defined as `Par(A)`. These two models are *model equivalent*, as they both effect just `A`. However, they would not be equivalent if we were to define model equivalence on the basis of strong equivalence. This is because, for model `Workflow1`, there is more internal activity in encapsulating `A` within a `Par` activity type.

We present results of checking observational and strong equivalences between `Workflow0` and `Workflow1`. The CCS source for these workflow models follows. In presenting the source, we mostly omit the definition of tracker and generic type agents for brevity. Their definitions are identical to those presented in Section 5.1.2.

```
************************************
* CCS Verification Run **************
* # 0
* Generated from: file:samples/LiesbetEquivTestA0.liesbet
* On: Tue Jul 11 14:24:13 BST 2006

...appropriate tracker and generic type agents...
```

```
proc Basics1 =
   bas.'comp1.yes1.'eyes_a.'bas.Basics1

proc Scheduler0 =
   'find_0.(yes_0.'rfind.nil + no_0.'bas.bas.Scheduler0)

proc Scheduler1 =
   'find_0.(yes_0.'rfind.nil + no_0.'lock.(idle.'bas.bas.'reset.Scheduler1 +
                                          prog.Scheduler1))

proc Workflow0 =
   (
***Instance:0:A
   InitialState0[yes_0/yes, no_0/no, runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, ninit_0/ninit, comp_0/comp, pcanc_0/pcanc, canc_0/canc, exec_0/exec] |

   Basics1[comp_0/comp1, yes_0/yes1, eyes_a/eyes1] |

   'exec_0.Scheduler0 | pprec.'pyes.nil

   )\{
      runn_0, cald_0, cotd_0, find_0, ninit_0, comp_0, pcanc_0, canc_0, exec_0, prec_0, yes_0, no_0,
      bas, pprec, pyes, lock, idle, prog, reset}

proc Workflow1 =
   (
***Instance:0:P1
   InitialState1[yes_0/yes, no_0/no, runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, ninit_0/ninit, comp_0/comp, pcanc_0/pcanc, canc_0/canc, exec_0/exec,
      prec_0/prec, pcanc_1/pcanc1] |

   Par1[yes_0/yes, no_0/no, runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, ninit_0/ninit, comp_0/comp, canc_0/canc, exec_0/exec,
      exec_1/exec1] |

***Instance:1:A
   InitialState0[yes_1/yes, no_1/no, runn_1/runn, cald_1/cald, cotd_1/cotd,
      find_1/find, ninit_1/ninit, comp_1/comp, pcanc_1/pcanc, canc_1/canc, exec_1/exec,
      prec_0/pprec, yes_0/pyes] |

Basics1[comp_1/comp1, yes_1/yes1, eyes_a/eyes1] |

'exec_0.Scheduler1 | pprec.'pyes.nil

)\{
   runn_0, cald_0, cotd_0, find_0, ninit_0, comp_0, pcanc_0, canc_0, exec_0, prec_0, yes_0, no_0,
```

```
runn_1, cald_1, cotd_1, find_1, ninit_1, comp_1, pcanc_1, canc_1, exec_1, prec_1, yes_1, no_1,
bas, pprec, pyes, lock, idle, prog, reset}
```

If we check the observational equivalence of these workflow models under CWB-NC, we can see that they are found to be equivalent.

```
cwb-nc> eq -S obseq Workflow0 Workflow1
Building automaton...
States: 42
Transitions: 40
Done building automaton.
Transforming automaton...
Done transforming automaton.
TRUE
Execution time (user,system,gc,real):(0.012,0.000,0.000,0.011)
cwb-nc>
```

But, if we check for strong equivalence, we can see that they are not found to be equivalent. Strong equivalence is too strong a notion for workflow model equivalence. That is, the internal behaviour of a model is not important, as long as the observable behaviour in terms of basic instances offered for completion, and in terms of completion of the model as a whole, is the same.

```
cwb-nc> eq -S bisim Workflow0 Workflow1
Building automaton...
States: 42
Transitions: 40
Done building automaton.
FALSE...
Workflow0 satisfies:
<t><t><t><t><t><t><t><t>[t]ff
Workflow1 does not.
Execution time (user,system,gc,real):(0.008,0.004,0.000,0.012)
cwb-nc>
```

Liesbet Model Equivalence, Example 2: v Trace Equivalence

Observational equivalence is a *stronger* notion than *trace equivalence.* For trace equivalence, we are concerned solely with comparing the possible *sequences* of basic activity completion of workflow models. For observational equivalence, however, we seek to compare the choices of basic activities to complete at corresponding stages of evolution of workflow models.

An example that highlights this distinction is the following simple one. Let `Liesbet` model `Workflow0` be defined as `Seq(A, Choice(Empty, B, Empty, C))`, and let `Liesbet` model `Workflow1` be defined as `Choice(Empty, Seq(A, B), Empty, Seq(A, C))`. For `Workflow0`, we do not make a commitment on the choice between `B` and `C` until after we have performed `A`. For `Workflow1`, in contrast, we make the choice before we execute `A`. These models do not maintain the same choices of activities to complete at corresponding points in their evolution. That is, after `A` has

been completed, both B and C are available in Workflow0, whereas only one of B or C is available in Workflow1. However, the two models are *trace equivalent*, as they both manifest the sequences of activity completion: A,B and A,C.

We present results of checking observational and trace equivalences between Workflow0 and Workflow1. The CCS source for these workflow models follows. In presenting the source, we mostly omit the definition of tracker and generic type agents for brevity. Their definitions are identical to those presented in Section 5.1.2.

```
*****************************************
* CCS Verification Run **************
* # 0
* Generated from: file:samples/LiesbetEquivTestB0.liesbet
* On: Wed Jul 12 16:41:00 BST 2006

...appropriate tracker and generic type agents...

proc Basics3 =
   bas.(
   'comp1.'eyes1.yes1.'bas.Basics3 +
   'comp2.'eyes2.yes2.'bas.Basics3 +
   'comp3.'eyes3.yes3.'bas.Basics3)

proc Basics4 =
   bas.(
   'comp1.'eyes1.yes1.'bas.Basics4 +
   'comp2.'eyes2.yes2.'bas.Basics4 +
   'comp3.'eyes3.yes3.'bas.Basics4 +
   'comp4.'eyes4.yes4.'bas.Basics4)

proc Scheduler =
   'find_0.(yes_0.'rfind.nil + no_0.'lock.(idle.'lock.
                        (idle.'lock.(idle.'lock.(idle.'bas.bas.'reset.'reset.'reset.'reset.Scheduler +
                                                  prog.'reset.'reset.'reset.Scheduler) +
                                       prog.'reset.'reset.Scheduler) +
                         prog.'reset.Scheduler) +
                                              prog.Scheduler))

proc Scheduler1 =
'find_0.(yes_0.'rfind.nil + no_0.'lock.(idle.'lock.
 (idle.'lock.(idle.'lock.(idle.'lock.(idle.'bas.bas.'reset.'reset.'reset.'reset.'reset.Scheduler1 +
                                        prog.'reset.'reset.'reset.'reset.Scheduler1) +
                             prog.'reset.'reset.'reset.Scheduler1) +
                 prog.'reset.'reset.Scheduler1) +
  prog.'reset.Scheduler1) +
                                          prog.Scheduler1))
```

```
proc Workflow0 =
   (
***Instance:0:S1
   InitialState2[yes_0/yes, no_0/no, runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, ninit_0/ninit, comp_0/comp, pcanc_0/pcanc, canc_0/canc, exec_0/exec,
      prec_0/prec,
   pcanc_1/pcanc1, yes_1/yes1, pcanc_2/pcanc2, yes_2/yes2] |

   Seq2[yes_0/yes, no_0/no, runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, ninit_0/ninit, comp_0/comp, pcanc_0/pcanc, canc_0/canc, exec_0/exec,
      exec_1/exec2, find_1/find2, yes_1/yes2, no_1/no2, exec_2/exec1] |

***Instance:1:A
   InitialState0[yes_1/yes, no_1/no, runn_1/runn, cald_1/cald, cotd_1/cotd,
      find_1/find, ninit_1/ninit, comp_1/comp, pcanc_1/pcanc, canc_1/canc, exec_1/exec,
      prec_0/pprec, yes_0/pyes] |

***Instance:2:CH
   InitialState4[yes_2/yes, no_2/no, runn_2/runn, cald_2/cald, cotd_2/cotd,
      find_2/find, ninit_2/ninit, comp_2/comp, pcanc_2/pcanc, canc_2/canc, exec_2/exec,
      prec_2/prec, prec_0/pprec, yes_0/pyes,
   pcanc_3/pcanc1, yes_3/yes1, pcanc_4/pcanc2, yes_4/yes2,
   pcanc_5/pcanc3, yes_5/yes3, pcanc_6/pcanc4, yes_6/yes4] |

   Choice2[yes_2/yes, no_2/no, runn_2/runn, cald_2/cald, cotd_2/cotd,
      find_2/find, ninit_2/ninit, comp_2/comp, pcanc_2/pcanc, canc_2/canc, exec_2/exec,
      exec_3/execg1, exec_4/execc1, canc_3/cancg1, canc_4/cancc1, yes_4/yesc1, cotd_3/cotdg1,
      runn_3/runng1, yes_3/yesg1, no_3/nog1, cald_3/caldg1,
      exec_5/execg2, exec_6/execc2, canc_5/cancg2, canc_6/cancc2, yes_6/yesc2, cotd_5/cotdg2,
      runn_5/runng2, yes_5/yesg2, no_5/nog2, cald_5/caldg2] |

***Instance:3:Em1
   InitialState0[yes_3/yes, no_3/no, runn_3/runn, cald_3/cald, cotd_3/cotd,
      find_3/find, ninit_3/ninit, comp_3/comp, pcanc_3/pcanc, canc_3/canc, exec_3/exec,
      prec_2/pprec, yes_2/pyes] |

   Empty[yes_3/yes, no_3/no, runn_3/runn, cald_3/cald, cotd_3/cotd,
      find_3/find, ninit_3/ninit, comp_3/comp, pcanc_3/pcanc, canc_3/canc, exec_3/exec] |

***Instance:4:B
   InitialState0[yes_4/yes, no_4/no, runn_4/runn, cald_4/cald, cotd_4/cotd,
      find_4/find, ninit_4/ninit, comp_4/comp, pcanc_4/pcanc, canc_4/canc, exec_4/exec,
      prec_2/pprec, yes_2/pyes] |

***Instance:5:Em2
   InitialState0[yes_5/yes, no_5/no, runn_5/runn, cald_5/cald, cotd_5/cotd,
      find_5/find, ninit_5/ninit, comp_5/comp, pcanc_5/pcanc, canc_5/canc, exec_5/exec,
```

```
      prec_2/pprec, yes_2/pyes] |

   Empty[yes_5/yes, no_5/no, runn_5/runn, cald_5/cald, cotd_5/cotd,
      find_5/find, ninit_5/ninit, comp_5/comp, pcanc_5/pcanc, canc_5/canc, exec_5/exec] |

***Instance:6:C
   InitialState0[yes_6/yes, no_6/no, runn_6/runn, cald_6/cald, cotd_6/cotd,
      find_6/find, ninit_6/ninit, comp_6/comp, pcanc_6/pcanc, canc_6/canc, exec_6/exec,
      prec_2/pprec, yes_2/pyes] |

   Basics3[comp_1/comp1, yes_1/yes1, eyes_a/eyes1,
           comp_4/comp2, yes_4/yes2, eyes_b/eyes2,
           comp_6/comp3, yes_6/yes3, eyes_c/eyes3] |

'exec_0.Scheduler | pprec.'pyes.nil

)\{
   runn_0, cald_0, cotd_0, find_0, ninit_0, comp_0, pcanc_0, canc_0, exec_0, prec_0, yes_0, no_0,
   runn_1, cald_1, cotd_1, find_1, ninit_1, comp_1, pcanc_1, canc_1, exec_1, prec_1, yes_1, no_1,
   runn_2, cald_2, cotd_2, find_2, ninit_2, comp_2, pcanc_2, canc_2, exec_2, prec_2, yes_2, no_2,
   runn_3, cald_3, cotd_3, find_3, ninit_3, comp_3, pcanc_3, canc_3, exec_3, prec_3, yes_3, no_3,
   runn_4, cald_4, cotd_4, find_4, ninit_4, comp_4, pcanc_4, canc_4, exec_4, prec_4, yes_4, no_4,
   runn_5, cald_5, cotd_5, find_5, ninit_5, comp_5, pcanc_5, canc_5, exec_5, prec_5, yes_5, no_5,
   runn_6, cald_6, cotd_6, find_6, ninit_6, comp_6, pcanc_6, canc_6, exec_6, prec_6, yes_6, no_6,
   bas, pprec, pyes, lock, idle, prog, reset}

proc Workflow1 =
   (
***Instance:0:CH
   InitialState4[yes_0/yes, no_0/no, runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, ninit_0/ninit, comp_0/comp, pcanc_0/pcanc, canc_0/canc, exec_0/exec, prec_0/prec,
      pcanc_1/pcanc1, yes_1/yes1, pcanc_2/pcanc2, yes_2/yes2,
      pcanc_5/pcanc3, yes_5/yes3, pcanc_6/pcanc4, yes_6/yes4] |

   Choice2[yes_0/yes, no_0/no, runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, ninit_0/ninit, comp_0/comp, pcanc_0/pcanc, canc_0/canc, exec_0/exec,
      exec_1/execg1, exec_2/execc1, canc_1/cancg1, canc_2/cancc1, yes_2/yesc1, cotd_1/cotdg1,
      runn_1/runng1, yes_1/yesg1, no_1/nog1, cald_1/caldg1,
      exec_5/execg2, exec_6/execc2, canc_5/cancg2, canc_6/cancc2, yes_6/yesc2, cotd_5/cotdg2,
      runn_5/runng2, yes_5/yesg2, no_5/nog2, cald_5/caldg2] |

***Instance:1:Em1
   InitialState0[yes_1/yes, no_1/no, runn_1/runn, cald_1/cald, cotd_1/cotd,
      find_1/find, ninit_1/ninit, comp_1/comp, pcanc_1/pcanc, canc_1/canc, exec_1/exec,
      prec_0/pprec, yes_0/pyes] |
```

```
   Empty[yes_1/yes, no_1/no, runn_1/runn, cald_1/cald, cotd_1/cotd,
      find_1/find, ninit_1/ninit, comp_1/comp, pcanc_1/pcanc, canc_1/canc, exec_1/exec] |

***Instance:2:S1
   InitialState2[yes_2/yes, no_2/no, runn_2/runn, cald_2/cald, cotd_2/cotd,
      find_2/find, ninit_2/ninit, comp_2/comp, pcanc_2/pcanc, canc_2/canc, exec_2/exec,
      prec_2/prec, prec_0/pprec, yes_0/pyes,
   pcanc_3/pcanc1, yes_3/yes1, pcanc_4/pcanc2, yes_4/yes2] |

   Seq2[yes_2/yes, no_2/no, runn_2/runn, cald_2/cald, cotd_2/cotd,
      find_2/find, ninit_2/ninit, comp_2/comp, pcanc_2/pcanc, canc_2/canc, exec_2/exec,
      exec_3/exec2, find_3/find2, yes_3/yes2, no_3/no2, exec_4/exec1] |

***Instance:3:A
   InitialState0[yes_3/yes, no_3/no, runn_3/runn, cald_3/cald, cotd_3/cotd,
      find_3/find, ninit_3/ninit, comp_3/comp, pcanc_3/pcanc, canc_3/canc, exec_3/exec,
      prec_2/pprec, yes_2/pyes] |

***Instance:4:B
   InitialState0[yes_4/yes, no_4/no, runn_4/runn, cald_4/cald, cotd_4/cotd,
      find_4/find, ninit_4/ninit, comp_4/comp, pcanc_4/pcanc, canc_4/canc, exec_4/exec,
      prec_2/pprec, yes_2/pyes] |

***Instance:5:Em2
   InitialState0[yes_5/yes, no_5/no, runn_5/runn, cald_5/cald, cotd_5/cotd,
      find_5/find, ninit_5/ninit, comp_5/comp, pcanc_5/pcanc, canc_5/canc, exec_5/exec,
      prec_0/pprec, yes_0/pyes] |

   Empty[yes_5/yes, no_5/no, runn_5/runn, cald_5/cald, cotd_5/cotd,
      find_5/find, ninit_5/ninit, comp_5/comp, pcanc_5/pcanc, canc_5/canc, exec_5/exec] |

***Instance:6:S2
   InitialState2[yes_6/yes, no_6/no, runn_6/runn, cald_6/cald, cotd_6/cotd,
      find_6/find, ninit_6/ninit, comp_6/comp, pcanc_6/pcanc, canc_6/canc, exec_6/exec,
      prec_6/prec, prec_0/pprec, yes_0/pyes,
   pcanc_7/pcanc1, yes_7/yes1, pcanc_8/pcanc2, yes_8/yes2] |

   Seq2[yes_6/yes, no_6/no, runn_6/runn, cald_6/cald, cotd_6/cotd,
      find_6/find, ninit_6/ninit, comp_6/comp, pcanc_6/pcanc, canc_6/canc, exec_6/exec,
      exec_7/exec2, find_7/find2, yes_7/yes2, no_7/no2, exec_8/exec1] |

***Instance:7:A
   InitialState0[yes_7/yes, no_7/no, runn_7/runn, cald_7/cald, cotd_7/cotd,
      find_7/find, ninit_7/ninit, comp_7/comp, pcanc_7/pcanc, canc_7/canc, exec_7/exec,
      prec_6/pprec, yes_6/pyes] |

***Instance:8:C
```

```
InitialState0[yes_8/yes, no_8/no, runn_8/runn, cald_8/cald, cotd_8/cotd,
   find_8/find, ninit_8/ninit, comp_8/comp, pcanc_8/pcanc, canc_8/canc, exec_8/exec,
   prec_6/pprec, yes_6/pyes] |

Basics4[comp_3/comp1, yes_3/yes1, eyes_a/eyes1,
        comp_4/comp2, yes_4/yes2, eyes_b/eyes2,
        comp_7/comp3, yes_7/yes3, eyes_a/eyes3,
        comp_8/comp4, yes_8/yes4, eyes_c/eyes4] |

'exec_0.Scheduler1 | pprec.'pyes.nil

)\{
   runn_0, cald_0, cotd_0, find_0, ninit_0, comp_0, pcanc_0, canc_0, exec_0, prec_0, yes_0, no_0,
   runn_1, cald_1, cotd_1, find_1, ninit_1, comp_1, pcanc_1, canc_1, exec_1, prec_1, yes_1, no_1,
   runn_2, cald_2, cotd_2, find_2, ninit_2, comp_2, pcanc_2, canc_2, exec_2, prec_2, yes_2, no_2,
   runn_3, cald_3, cotd_3, find_3, ninit_3, comp_3, pcanc_3, canc_3, exec_3, prec_3, yes_3, no_3,
   runn_4, cald_4, cotd_4, find_4, ninit_4, comp_4, pcanc_4, canc_4, exec_4, prec_4, yes_4, no_4,
   runn_5, cald_5, cotd_5, find_5, ninit_5, comp_5, pcanc_5, canc_5, exec_5, prec_5, yes_5, no_5,
   runn_6, cald_6, cotd_6, find_6, ninit_6, comp_6, pcanc_6, canc_6, exec_6, prec_6, yes_6, no_6,
   runn_7, cald_7, cotd_7, find_7, ninit_7, comp_7, pcanc_7, canc_7, exec_7, prec_7, yes_7, no_7,
   runn_8, cald_8, cotd_8, find_8, ninit_8, comp_8, pcanc_8, canc_8, exec_8, prec_8, yes_8, no_8,
   bas, pprec, pyes, lock, idle, prog, reset}
```

If we check the observational equivalence of these workflow models under CWB-NC, we can see that they are not found to be equivalent. As reported, Workflow0 is capable of completing either B or C after completing A, but Workflow1 is not capable of this.

```
cwb-nc> eq -S obseq Workflow0 Workflow1
Building automaton...
.........1000.........2000.........3000.........4000.........5000
.........6000.........7000.........
States: 7955
Transitions: 10570
Done building automaton.
Transforming automaton...
Done transforming automaton.
FALSE...
Workflow0 satisfies:
<<'eyes_a>>(<<'eyes_b>>tt ∧ <<'eyes_c>>tt)
Workflow1 does not.
Execution time (user,system,gc,real):(646.728,65.880,383.524,712.701)
cwb-nc>
```

But, if we check for trace equivalence, we can see that they are found to be equivalent. Trace equivalence is too weak a notion for workflow model equivalence. That is, two (or more) models may demonstrate trace equivalence when their observable behaviour is not the same.

```
cwb-nc>eq -S trace Workflow0 Workflow1
Building automaton...
.........1000.........2000.........3000.........4000.........5000
.........6000.........7000.........
States: 7955
Transitions: 10570
Done building automaton.
Transforming automaton...
Done transforming automaton.
TRUE
Execution time (user,system,gc,real):(22.201,0.640,5.216,22.842)
cwb-nc>
```

5.2 Completion Result for Liesbet1 Models

Result:

> A Liesbet1 model (constructed according to the syntactical constraints defined by the meta-model) is guaranteed to complete (that is, all instances report completion, or cancellation) in a finite number of steps.

Proof.

- *Base cases* We work inductively from the base case of a Liesbet1 model consisting of one activity instance. Such an instance must be of a childless generic type, namely, a basic activity, FreeChoice, or Empty.

  In the case of a **basic activity instance**, the CCS-based model that the translator outputs would be as follows.

  `InitialState0[`$SC_{\_0}$`] | Basics1[`$SC_{\_0,1}$`] | 'exec_0.Scheduler0 | pprec.'pyes.nil`

  Here, Basics1 has the definition:

  ```
  proc Basics1 =
     bas.'comp1.yes1.'bas
  ```

  And, Scheduler0:

  ```
  'find_0.(yes_0.'rfind.nil + no_0.'bas.bas.Scheduler0)
  ```

  With reference to the definition of InitialState0, presented in Section 5.1.2, and to the foregoing, it is clear that the only transitions that the model is capable of making can be characterised by a single chain of synchronisations, followed by a visible (output) transition on rfind, indicating that the root instance (and thus the model enactment) has completed.

  The single chain is made up of the following synchronisations on channels (in order): exec_0, find_0, no_0, bas, comp_0, pprec, pyes, yes_0, bas, find_0, yes_0, followed by an output transition on rfind.

In the case of structured childless activity types, such as FreeChoice, the single-act CCS-based model output from the translator would look as follows.

InitialState0[SC_0] | FreeChoice[SC_0] | 'exec_0.Scheduler1_NoBasics | pprec.'pyes.nil

FreeChoice has the definition presented in Section 5.1.2, namely:

```
proc FreeChoice =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'runn.(yes.('comp.yes.'prog.Idle + 'canc.yes.'prog.Idle) +
                         no.'idle.reset.FreeChoice))
```

Note that, there is no Basics^b agent, as there are no basic instances in the model. The scheduler agent needs to reflect this also. Its definition would be as follows.

```
'find_0.(yes_0.'rfind.nil + no_0.'lock.prog.Scheduler0)
```

In the case of a single FreeChoice instance, there are two possible enactment chains. They diverge depending on whether the FreeChoice instance is completed, or cancelled. The chains have a common prefix of synchronisations on channels (in order): exec_0, find_0, no_0, lock, cald_0, no_0, runn_0, yes_0.

Having established the FreeChoice is running, we may either cancel or complete it. Thereafter, both enactment chains have the same suffix of synchronisations: pprec, pyes, yes_0, prog, find_0, yes_0, followed by an output transition on rfind.

For Empty, there is a single enactment chain, as we may only complete the instance.

Clearly, for all of the base cases, the instances need to be have been set running for them to complete, or cancel. We have shown that once such an instance has been set running, it will eventually finish (complete, or cancel). There is no possibility of locking for the characterisations of the base cases.

Moreover, we propagate the finishing (completion, or cancellation) of the instance up to the parent by means of pprec. (In the case of a single-act model, there is no parent so the synchronisation occurs with the dummy agent pprec.'pyes.nil.)

For completed childless instances, it is imperative that their generic types agents forever idle so that they never starve other instances of the opportunity to progress. As FreeChoice and Empty both evolve to the agent Idle, this behaviour is assured.

- *Induction step*:

  We proceed by showing that the introduction of any instance of a child-bearing generic activity type does not affect the completion result that we are seeking to prove. By introduction, we mean that the instance coalesces a number of distinct Liesbet models, for which completion is guaranteed (by the induction hypothesis), as children. An example might be to coalesce two models, M1 and M2, as children of a Par type, viz. Par(M1, M2).

  When introducing a child-bearing instance, we need to ensure the following behaviour. The instance must *eventually* propagate execution (or cancellation) to each of its children. It must *otherwise idle* until all of its children have finished, at which point the instance should

itself complete, and propagate completion upwards. Its generic type agent should thereafter forever idle. The idling ensures that instances elsewhere are not starved of the opportunity to progress.

From inspection, it is clear that all of the child-bearing types effect the required behaviour, in their agent definitions. For example, Par propagates execution to all of its child instances, and thereafter reports idle until all of them have finished, whereon it completes itself and propagates completion up to its parent.

□

As completion is guaranteed for Liesbet1 models, there can be no source of locking in such models. Furthermore, enactment is guaranteed to be finite.

5.3 Discussion: CCS for Liesbet1

As stated in Section 3.2, any characterisation of Liesbet must prioritise the progression of structured activity instances over the progression of basic instances. In the absence of an explicit notion of priority in CCS, the only way to effect such a priority is to use a scheduler for activity instances. The operation of such a scheduler would be to interrogate (potentially) all structured instances to ascertain whether they can progress, and to only progress a basic instance if the structured instances cannot.

The use of an explicit scheduler is rather costly in terms of the state space that is entailed. Often, it will be the case that an activity instance will be incapable of progressing in the current state, but there will still be the cost (in terms of transitions and states) of identifying as much. However, its cost can be somewhat offset by the use of a locking mechanism on execution rights.

In enabling structured activity instances to make numerous transitions without interleaved transitions pertaining to other activity instances – the instance enjoys an *execution window*, we effect a *partial-order reduction* (POR) on the state space [38, 59]. A POR removes certain permutations of transitions from a state space. Through a POR, we are seeking to institute a greater degree of total-ordering between the transitions that may be made between any two states, which serves to reduce the state space between the two states. This will reduce the complexity of verification, which is proportional to the size of the state space entailed by a CCS model.

However, the complexity of verification remains punitive, for even the simplest of models (as can be seen from the example of Section 5.1.3, when checked against a proposition in Section 5.1.4). A significant improvement can be made in the verification complexity by using a variant of CCS which has built-in support for the expression of priority, viz. PCCS. In the next section, we present a characterisation of Liesbet, using PCCS. The capability of expressing priorities of certain actions over others removes the need for an explicit scheduler, and as a consequence considerably reduces the size of state spaces that are entailed from PCCS-characterised Liesbet models compared with their respective CCS-characterisations.

As we argue in the concluding remarks to this chapter (in Section 5.7), it is possible to provide a mapping for the whole of Liesbet into CCS. We omit the presentation of such a mapping due to space constraints. We do present such a mapping in our presentation of PCCS-based semantics for Liesbet, which is now given.

5.4 Using PCCS to Provide an Operational Meaning to Liesbet

We are motivated to use PCCS to provide a formal characterisation of Liesbet because of the punitive verification complexity of our CCS-based characterisation. The capacity for expressing priority in PCCS means that the use of an explicit scheduler is not required. Instead, we can use transitions of differing priorities to ensure that structured instances get progressed ahead of basic instances. The absence of an explicit scheduler should greatly improve the efficiency of verification for all but the simplest of models.

In this section, we present our PCCS-based characterisation of Liesbet1. We defer the presentation of our PCCS characterisation of the rest of Liesbet to Appendix A, in order to save space here.

5.4.1 PCCS: Liesbet1

In our PCCS characterisation of Liesbet, we maintain the notion of an *execution window*, where activity instances claim a lock on execution rights in order to progress meaningfully before yielding so that another instance may grab the lock. This is effected without the use of an explicit scheduler (as in our CCS-based characterisation) by specifying that activity instances specify transitions at a certain priority level when seeking to claim the lock, and subsequently specify transitions at a higher priority level in order to maintain ownership of the lock. Eventually, an instance will yield such rights implicitly by not effecting further synchronisations at this elevated level of priority.

In summary, we make use of the following priority levels (where the higher the number, the lower the priority). Note that it is the relative ordering of priorities that is important, not the specific numbers.

- 20 – For the basic activity arbiter to *claim* execution rights. It is appropriate for the arbiter to claim execution rights at the lowest level of priority, as it should not operate unless work cannot be done otherwise in the model. However, once it has claimed execution rights, all of its subsequent behaviour operates at the highest levels of priority (3–6) until the arbiter wishes to yield execution rights.

- 10 – The initial action of a structured instance will execute at this level to claim execution rights, i.e. at a higher level of priority than the arbiter for basic instances, but at a lower level of priority than 3–6, which are the levels at which agents who have already claimed execution rights operate.

- 3–6 – Actions which are part of an agent (e.g., pertaining to a structured instance, or basic instance arbiter) which has claimed execution rights. The varying levels are used to cut down the number of possible paths between pairs of states, where these various paths are unimportant to the global evolution of the model. Thus, the variety of levels help in effecting a *POR*. Note that levels 3 and 5 are the levels that are principally used.

  – 6 – Used within synchronisation types – which largely run at 5 – to specify a lower priority for the success of StopQuerys (resp. GoQuerys) compared with GoQuerys (resp. StopQuerys) which take precedence in Gos (resp. Stops).

- 5 – The principal priority level used within synchronisation activity types. As described in the characterisation of `Liesbet2`, their queries get satisfied over the course of the enactment of the workflow model; but, for better *POR*, it is appropriate for this to occur between activity instances of other types holding execution rights. As a result, they do not hold the same priority level as actions of those instances that hold execution rights (i.e. 3); but they hold a higher priority than actions of those instances that seek to claim execution rights (i.e. 10, or 20).
- 4 – This is a level used within synchronisation types for *handshaking* synchronisations. These communicate to an agent effecting subsequent behaviour that a precursive agent has completed its activity. The priority level is used wholly for effecting *POR* between actions pertaining to handshaking and those not.
- 3 – This is the main priority level used for activity instances that have previously claimed execution rights. While synchronisations are possible at this level, logic pertaining to other activity instances may not be advanced. This is because all such other logic will be guarded by actions which have priorities set to 10 or 20, as the pertaining activity instances are yet to successfully claim execution rights, or, set to 5, in the case of synchronisation types.

The definition of the translation function, $\mathcal{M}_{pccs}[\![-]\!]$, for PCCS `Liesbet1` is largely the same as that presented in Section 5.1.2, for CCS. The only difference is that we pass the name of an instance's cancellation channel, `cald`, to $\mathcal{M}_{pccs}[\![-]\!]$ when translating one of its child instances, so that a child instance may detect when its parent has been cancelled, and, as a consequence, cancel itself. This *pull* approach is different to that taken in the CCS characterisation, where instances explicitly *push* cancellation to their children using `pcanc`. Now, we remove the use of `pcanc` actions which makes the definitions of agents simpler. In the CCS characterisation, without being able to specify priorities for transitions, there is no simple way of implementing a pull approach.

We omit a presentation of $\mathcal{M}_{pccs}[\![-]\!]$ for `Liesbet1` types as it is the same as the CCS characterisation, save for this one aspect. However, as an example of its definition for PCCS, we show $\mathcal{M}_{pccs}[\![-]\!]$ applied to the `Seq` activity type, where $st\_chs \rightarrow$ is defined as it was for the CCS characterisation, but without a `pcanc` channel, and without answer channels `yes` and `no`. That is, for the PCCS characterisation, we abbreviate the channel list: `exec, comp, canc, ninit, runn, cotd, cald, find` by $st\_chs \rightarrow$.

$$
\begin{array}{l}
\mathcal{M}_{pccs}[\![\texttt{Seq(Ch1,...,Chn)}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i, \texttt{pcald}_i) = \\
\texttt{let } st\_chs_{\texttt{i1}} \rightarrow \texttt{ in ... } st\_chs_{\texttt{in}} \rightarrow \texttt{ in let prec}_i \texttt{ in} \\
\quad \texttt{Seq}^n\,[SC_i,\ SC_{\texttt{in},1},\ \ldots,\ SC_{\texttt{i1},\texttt{n}}] \\
\mid \\
\quad \texttt{InitialState}^n\,[SC_i, {}^{\texttt{pprec}_i}/_{\texttt{pprec}}, {}^{\texttt{prec}_i}/_{\texttt{prec}}, {}^{\texttt{pcald}_i}/_{\texttt{pcald}}] \\
\mid \\
\quad \mathcal{M}_{pccs}[\![\texttt{Ch1}]\!](st\_chs_{\texttt{i1}} \rightarrow, \texttt{prec}_i, \texttt{cald}_i) \mid \ldots \mid \mathcal{M}_{pccs}[\![\texttt{Chn}]\!](st\_chs_{\texttt{in}} \rightarrow, \texttt{prec}_i, \texttt{cald}_i)
\end{array}
$$

The definition of the state tracking agent, $\texttt{InitialState}^n$, where n is 2, is as follows.

```
proc InitialState2 =
   'pcald:5.CancelledState +
```

```
canc:3.'pprec:5.CancelledState +
canc:10.'pprec:5.CancelledState +
exec:3.RunningState2
```

Notably, in using a blocking paradigm, we do not need to facilitate synchronisation on all of an activity instance's state channels. We do need to support a number of priority levels, however, which differ for the various channels.

This agent accepts:

- A synchronisation on `'pcald`, allowing us to detect when the parent instance has been cancelled. Its priority is set at level 5 to make sure that it takes place in between an agent yielding execution rights and another claiming them.

- A synchronisation on `canc`, at levels 3, and 10, to effect a cancellation. Level 3 is the main priority level used when an agent pertaining to an activity instance holds execution rights. Level 10 is used for agents seeking to claim execution rights.

  Having synchronised on `canc`, the $\texttt{InitialState}^n$ agent signals to its parent that it has finished on `pprec` (at level 5), and then moves to a `Cancelled` state. A priority of 5 is used for `pprec`, as it should have a lower priority than other possible synchronisations of the activity instance holding execution rights; but it should occur before execution rights are yielded. This distinction effects a POR.

- A synchronisation on `exec` may occur once an agent has execution rights (level 3), which exposes a `RunningState2` agent.

The definition of $\texttt{RunningState}^r$, where r is 0-2 is as follows. Note that `RunningState` agents, in the PCCS characterisation, are not annotated by the pair n_r. This is because we do not need to keep track of n any longer, as we do not push cancellation on to child instances. We simply decrement the r count whenever a child instance signals that it has finished (on `prec`).

```
proc RunningState0 =
   comp:20.'pprec:5.CompletedState + comp:10.'pprec:5.CompletedState +
                                     comp:3.'pprec:5.CompletedState +
   runn:10.RunningState0 + runn:5.RunningState0 + runn:3.RunningState0 +
   ninit:5.RunningState0 +
   'pcald:5.CancelledState +
   canc:3.'pprec:5.CancelledState + canc:10.'pprec:5.CancelledState

proc RunningState1 =
   prec:5.'pprec:5.CompletedState +
   runn:10.RunningState1 + runn:5.RunningState1 + runn:3.RunningState1 +
   ninit:5.RunningState1 +
   'pcald:5.CancelledState +
   canc:3.'pprec:5.CancelledState + canc:10.'pprec:5.CancelledState

proc RunningState2 =
   prec:5.RunningState1 +
```

```
runn:10.RunningState2 + runn:5.RunningState2 + runn:3.RunningState2 +
ninit:5.RunningState2 +
'pcald:5.CancelledState +
canc:3.'pprec:5.CancelledState + canc:10.'pprec:5.CancelledState
```

- In RunningState0, we support the completion of *childless* activity instances, at levels 20 – for basic instances, 10 – for structured instances where we are seeking to claim execution rights, and 3 – where we already have execution rights. We further support confirmation that we are running at 10 – to gain execution rights, 5 – for synchronisation types, and 3 – when we already have execution rights, and confirmation at 5 that the instance is not in the Initial state. Also, as well as supporting querying whether the parent is already cancelled, at 5, we support cancellation of the instance at 3 and 10.

- For RunningState1, compared with RunningState0, we disallow explicit completion, as we did in our CCS-based characterisation (see Section 5.1.2). Instead, when a (final) child instance signals that it has finished on prec (at priority 5), we propagate completion upwards (on pprec), and then move to CompletedState.

- The single change for RunningState^r, where $r > 1$, from RunningState1 again concerns prec. When a child signals that it has finished on prec, we decrement the count of outstanding child instances that are running. Thus, the agent to be exposed is RunningState^{r-1}.

For CompletedState and CancelledState, we support a number of querying actions at levels 10 – an agent does not have execution rights, and 5 – for queries within Liesbet synchronisation types. We do not need to support querying at level 3, as there is never a need for it (in our characterisations of the Liesbet generic activity types). However, sometimes, an instance may attempt to cancel one of its children when the child instance has already been cancelled, or completed. As such, we support a cancel action at level 3. Similarly, it may be the case that an instance attempts the execution of one its children when the child instance has already been cancelled, as already described in Section 5.1.2. As such, we support an execute action at level 3, for CancelledState.

```
proc CompletedState =

    cotd:10.CompletedState +
    cotd:5.CompletedState +

    find:10.CompletedState +
    find:5.CompletedState +
    ninit:5.CompletedState +

    canc:3.CompletedState

proc CancelledState =

    cald:10.CancelledState +
    cald:5.CancelledState +
```

```
find:10.CancelledState +
find:5.CancelledState +

canc:3.CancelledState +

ninit:5.CompletedState +

exec:3.CancelledState
```

The definitions of the agents effecting the various Liesbet generic activity types are now presented. For $\texttt{Par}^n$ and $\texttt{Seq}^n$, when n is 2:

```
proc Par2 =
   'runn:10.'exec1:3.'exec2:3.nil + 'cald:5.nil

proc Seq2 =
   'runn:10.'exec2:3.Seq2f + 'cald:5.nil

proc Seq2f =
   'find2:10.'exec1:3.nil + 'cald:5.nil
```

As can be seen, an agent such as Par2 attempts to claim execution rights at priority level 10, by establishing that its pertaining instance is in a Running state. If the attempt is successful, it thereafter operates at level 3 as it has these rights. Alternatively, to claiming execution rights, the agent may *with higher priority* synchronise on cald (at 5), which would occur if its tracker agent had moved into a Cancelled state. In this case, the logic effecting the generic activity type should be garbage-collected, as it is no longer pertinent.

Notably, these are much simpler definitions, thanks to the use of the blocking paradigm, with priority, than those presented for the standard CCS characterisation.

For $\texttt{Par}^n$, we simply initiate the execution of all child instances in turn. As execution rights are solely held by this agent, these actions will happen contiguously.

For $\texttt{Seq}^n$, child instances are indexed in decreasing order, which makes for simpler agent definitions. We start by initiating the execution of the first child instance, n. Then, we expose a $\texttt{Seq}^n\texttt{f}$ agent, and yield execution rights. $\texttt{Seq}^n\texttt{f}$ is responsible for waiting for the running child instance to finish, claiming execution rights once this occurs, and initiating the execution of the next child instance. After that, if $n > 2$, the agent constant $\texttt{Seq}^{n-1}$ is exposed, or nil, if $n = 2$.

Note that completion of child-bearing activity instances is effected within the pertaining state tracking agent, as done in the CCS-based characterisation.

The $\texttt{DefaultChoice}^n$ and $\texttt{Choice}^n$ types look as follows when n is 2.

```
proc DefaultChoice2 =
   'runn:10.'execg1:3.'execg2:3.DefaultChoice2f + 'cald:5.nil

proc DefaultChoice2f =
   ((
      'cotdg1:10.'execc1:3.'cancg2:3.'cancc2:3.nil +
```

```
      'caldg1:10.'cancc1:3.'lose:3.nil
   |
      'cotdg2:10.'execc2:3.'cancg1:3.'cancc1:3.nil +
      'caldg2:10.'cancc2:3.'lose:3.nil
   |
      lose:3.lose:3.'execd:3.nil
   )\{lose} [> 'find:5.nil)


proc Choice2 =
   'runn:10.'execg1:3.'execg2:3.Choice2f + 'cald:5.nil

proc Choice2f =
   ((
      'cotdg1:10.'execc1:3.'cancg2:3.'cancc2:3.nil +
      'caldg1:10.'cancc1:3.'lose:3.nil
   |
      'cotdg2:10.'execc2:3.'cancg1:3.'cancc1:3.nil +
      'caldg2:10.'cancc2:3.'lose:3.nil
   |
      lose:3.lose:3.'canc:3.nil
   )\{lose} [> 'find:5.nil)
```

For both DefaultChoice^n and Choice^n, we initiate the execution of the guard instances. Then, in DefaultChoice^{n}f and Choice^{n}f, if a guard instance completes successfully, we initiate the execution of its continuation instance and cancel all other guard and continuation instances. If the guard instance gets cancelled, we signal on `lose`. As detection of completion, or cancellation, is set at priority 10, and subsequent actions are set at 3, these transitions will occur contiguously. For DefaultChoice2f, if all the guard instances get cancelled then sufficient synchronisations on `lose` will take place for the `execd` action to be exposed. This action is subsequently effected, which causes the execution of the default continuation instance to be initiated. In Choice2f, in the same eventuality, the Choice instance as a whole is cancelled.

```
proc MultiChoice2 =
    'runn:10.'execg1:3.'execg2:3.MultiChoice2f + 'cald:5.nil


proc MultiChoice2f =
       'cotdg1:10.'execc1:3.nil + 'caldg1:10.'cancc1:3.nil + 'cald:5.nil
|
       'cotdg2:10.'execc2:3.nil + 'caldg2:10.'cancc2:3.nil + 'cald:5.nil
```

Similarly, for MultiChoice^n, we start by executing the guard instances. Then we seek to execute or cancel each continuation instance based on whether its pertaining guard instance has completed successfully, or has been cancelled.

```
proc FreeChoice =
   'runn:10.('comp:3.nil + 'canc:3.nil) + 'cald:5.nil
```

```
proc Empty =
   'runn:10.'comp:3.nil  + 'cald:5.nil
```

For a FreeChoice instance, we make a non-deterministic choice between completing the instance, or cancelling it. Empty instances are necessarily completed.

The definition of $\texttt{Basics}^n$ is similar to the standard CCS characterisation of Liesbet. We do not need to synchronise on a channel (bas) to claim execution rights, as we did for the CCS version, as we claim execution at a lower level of priority (20) than for structured instances. We simply complete one of the outstanding basic instances.

```
proc Basics4 =
      'comp1:20.Basics4 +
      'comp2:20.Basics4 +
      'comp3:20.Basics4 +
      'comp4:20.Basics4
```

5.5 Multi and MultiSeq

Although they are not part of Liesbet1, it is sufficiently interesting to consider the characterisation of Multi/MultiSeq types in the main text. The primary novelty in their characterisation is that they may have an unlimited number of child instances. This means that we need to maintain an auxiliary counter agent in order to keep track of the number of such instances (yet to finish). This is in contrast to the approach used for all other types where the count of such instances is maintained by virtue of which tracker agent currently obtains for the instance.

The translation of Multi and MultiSeq is defined by the following extensions to $\mathcal{M}_{pccs}[\![-]\!]$. The definition of the translation function makes use of auxiliary functions, MIT and $MITS$, which will be explained in due course.

$$\mathcal{M}_{pccs}[\![\texttt{Multi(ExecAct(join(ExecActJoin)))}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i, \texttt{pcald}_i) =$$
$$\texttt{let } st\_chs_{\texttt{ij}} \rightarrow \texttt{ in let } st\_chs_{\texttt{ie}} \rightarrow \texttt{ in}$$
$$MIT(\mathcal{M}_{pccs}[\![\texttt{ExecActJoin}]\!](st\_chs_{\texttt{ij}} \rightarrow, \texttt{prec}_i, \texttt{cald}_i),$$
$$\mathcal{M}_{pccs}[\![\texttt{ExecAct}]\!](st\_chs_{\texttt{ie}} \rightarrow, \texttt{prec}_i, \texttt{cald}_i))[\mathcal{SC}_i, \mathcal{SC}_{\texttt{ij,j}}, \mathcal{SC}_{\texttt{ie,e}}] \mid$$
$$\texttt{InitialState}[\mathcal{SC}_i, {}^{\texttt{pprec}_i}/_{\texttt{pprec}}, {}^{\texttt{pcald}_i}/_{\texttt{pcald}}]$$

$$\mathcal{M}_{pccs}[\![\texttt{MultiSeq(ExecAct(join(ExecActJoin)))}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i, \texttt{pcald}_i) =$$
$$\texttt{let } st\_chs_{\texttt{ij}} \rightarrow \texttt{ in let } st\_chs_{\texttt{ie}} \rightarrow \texttt{ in}$$
$$MITS(\mathcal{M}_{pccs}[\![\texttt{ExecActJoin}]\!](st\_chs_{\texttt{ij}} \rightarrow, \texttt{prec}_i, \texttt{cald}_i),$$
$$\mathcal{M}_{pccs}[\![\texttt{ExecAct}]\!](st\_chs_{\texttt{ie}} \rightarrow, \texttt{prec}_i, \texttt{cald}_i))[\mathcal{SC}_i, \mathcal{SC}_{\texttt{ij,j}}, \mathcal{SC}_{\texttt{ie,e}}] \mid$$
$$\texttt{InitialState}[\mathcal{SC}_i, {}^{\texttt{pprec}_i}/_{\texttt{pprec}}, {}^{\texttt{pcald}_i}/_{\texttt{pcald}}]$$

Recall from Section 3.1.15, a multiple-instance activity type defines an execution activity type, ExecAct, of which several instances may be executed. Whether an instance of ExecAct is executed depends on its associated join condition, ExecActJoin. When a Multi* instance is set running, it creates an instance of ExecAct and its join condition. If the join condition is evaluated to hold (at some point subsequently), the ExecAct instance is executed. Also, another ExecAct instance

and join condition instance are created. For `Multi`, the join condition is set running immediately. For `MultiSeq`, we wait until the previously executed `ExecAct` instance has finished. Once a join condition instance fails (i.e. goes to `Cancelled`) its pertaining `ExecAct` instance is cancelled, and no more child instances are created. When all `ExecAct` instances have finished, the `Multi` instance is completed.

For the characterisation of `Multi*` types, we maintain a counting agent, which starts life as `ZeroCount`, representing the number of outstanding `ExecAct` instances yet to finish. For the tracker agents, we make use of a reduced set of just four: `InitialState`, `RunningState`, `CompletedState` and `CancelledState`. This is sufficient because we make use of the counting agent to keep track of the number of outstanding child instances. `CompletedState` and `CancelledState` have the same definitions as presented in Section 5.4.1. The definition of `InitialState` is identical to that of `InitialState0`, presented there, except that whenever the agent evolves to `InitialState0` (resp. `RunningState0`), it now evolves to `InitialState` (resp. `RunningState`).

The `RunningState` agent is a parallel composition of `RunningStateAgent` and `ZeroCount` agents. The definition of `RunningStateAgent` is identical to that of `RunningState`^{r}, from Section 5.4.1, except that:

- The agent evolves to `RunningStateAgent` whenever it evolved back to `RunningState`^{r} before.
- The handling of `prec`, indicating the completion of a child, is now handled elsewhere (within agents output by MT and MTS).
- There is now a capability to complete the agent also, once we identify that (i) a join condition for an `ExecAct` has failed (i.e. has gone to cancelled), and (ii) there are no outstanding `ExecAct` instances to finish (as identified by a synchronisation on `zero`).

The agent `ZeroCount`, together with `OneCount`, is used to keep a count of the number of outstanding – i.e. yet to finish – instances of `ExecAct`. When the count is increased from zero to one, the agent `ZeroCount` evolves to the parallel composition: `OneCount | ZeroCount`. The presence, at any time, of n instances of `OneCount` indicates that the value of the counter is n. The definitions of `ZeroCount` and `OneCount` are from [80].

```
proc RunningState =
   RunningStateAgent | ZeroCount

proc RunningStateAgent =
   runn:10.RunningState2 + runn:5.RunningState2 + runn:3.RunningState2 +
   ninit:5.RunningState2 +
   'pcald:5.CancelledState +
   canc:3.'pprec:5.CancelledState + canc:10.'pprec:5.CancelledState +
   comp:5.'zero:5.'pprec:5.CompletedState

proc ZeroCount =
   inc:5.(OneCount[i1/i,z1/z,d1/d] | ZeroCount[i1/inc,z1/zero,d1/dec])\{i1,z1,d1} +
   zero:5.ZeroCount

proc OneCount =
   inc:5.'i:5.OneCount + dec:5.('d:5.OneCount + 'z:5.ZeroCount)
```

The auxiliary function MT (resp. MTS) is used to construct a Multi (resp. MultiSeq) agent, which is customised for the particular ExecActJoin and ExecAct types used within the multiple-instance type being translated. MT (resp. MTS) takes two parameters, which are agents that result from applying $\mathcal{M}_{pccs}[\![-]\!]$ to the given ExecActJoin and ExecAct types. In the following, we denote these agents by the names pExecActJoin and pExecAct.

MT inserts the following customised agent into the output of translating a Multi type using $\mathcal{M}_{pccs}[\![-]\!]$.

```
'runn:10.Multi_cust + 'cald:5
```

MT also inserts a number of customised agent definitions into the main PCCS source, including a definition for Multi_cust, viz.

```
proc Multi_cust =
  MultiExec_cust | pExecActJoin |'execj:3

proc MultiExec_cust =
  'cotdj:10.(pExecAct | 'exece:3.'inc:5.prece:5.'dec:5.nil | Multi_cust) +
  'caldj:5.('cald:5.nil + 'comp:5.nil)
```

In the foregoing, an instance of the join condition, ExecActJoin, is initially set running. This occurs as a result of the 'execj:3 action synchronising with its complementary action in the tracker agent for the join condition instance, $\texttt{InitialState}^n[SC_j]$, which is part of pExecActJoin. If the join condition instance completes successfully, we execute an instance of the execution activity of the Multi type, ExecAct, and increase the counter for outstanding child instances. At the same time, we execute another instance of ExecActJoin. Eventually, an execution activity instance will finish (as indicated on prece), and, when this occurs, we decrement the counter. If an ExecActJoin instance gets cancelled, at any stage, we signal completion of the Multi instance on comp, as long as the Multi instance has not already been cancelled. On synchronising on comp, RunningStateAgent will wait until all existing child execution activity instances have finished (as determined by synchronising on zero), and then evolve to CompletedState.

The characterisation of MultiSeq in PCCS is the same, except that in the definition for MultiExec_cust, *output by MTS*, we do not expose a fresh Multi_cust agent until we identify that the previous ExecAct instance has finished executing. The definition of MultiExec_cust is thus as follows.

```
proc MultiExec_cust =
  'cotdj:10.(pExecAct | 'exece:3.'inc:5.prece:5.'dec:5.Multi_cust) +
  'caldj:5.('cald:5.nil + 'comp:5.nil)
```

5.6 A Complete Example of Using PCCS for Liesbet1

We present the PCCS-based characterisation of the same example used in Section 5.1.3 for the standard CCS characterisation of Liesbet1, namely Par(Seq(A,B), Seq(C,D)).

```
*****************************************
* PCCS Verification Run ***************
* # 0
* Generated from: file:samples/LiesbetTest.liesbet
* On: Mon Jul 31 15:29:07 BST 2006

proc InitialState0 =
   'pcald:5.CancelledState +
   canc:3.'pprec:5.CancelledState +
   canc:10.'pprec:5.CancelledState +
   exec:3.RunningState0

proc InitialState2 =
   'pcald:5.CancelledState +
   canc:3.'pprec:5.CancelledState +
   canc:10.'pprec:5.CancelledState +
   exec:3.RunningState2

proc RunningState0 =
   comp:20.'pprec:5.CompletedState + comp:10.'pprec:5.CompletedState +
                                     comp:3.'pprec:5.CompletedState +
   runn:10.RunningState0 + runn:5.RunningState0 + runn:3.RunningState0 +
   ninit:5.RunningState0 +
   'pcald:5.CancelledState +
   canc:3.'pprec:5.CancelledState + canc:10.'pprec:5.CancelledState

proc RunningState1 =
   prec:5.'pprec:5.CompletedState +
   runn:10.RunningState1 + runn:5.RunningState1 + runn:3.RunningState1 +
   ninit:5.RunningState1 +
   'pcald:5.CancelledState +
   canc:3.'pprec:5.CancelledState + canc:10.'pprec:5.CancelledState

proc RunningState2 =
   prec:5.RunningState1 +
   runn:10.RunningState2 + runn:5.RunningState2 + runn:3.RunningState2 +
   ninit:5.RunningState2 +
   'pcald:5.CancelledState +
   canc:3.'pprec:5.CancelledState + canc:10.'pprec:5.CancelledState

proc CompletedState =
   cotd:10.CompletedState + cotd:5.CompletedState +
   find:10.CompletedState + find:5.CompletedState +
   ninit:5.CompletedState +
   canc:3.CompletedState
```

```
proc CancelledState =
   cald:10.CancelledState + cald:5.CancelledState +
   find:10.CancelledState + find:5.CancelledState +
   ninit:5.CancelledState +
   exec:3.CancelledState +
   canc:3.CancelledState

proc Basics4 =
      'comp1:20.Basics4 +
      'comp2:20.Basics4 +
      'comp3:20.Basics4 +
      'comp4:20.Basics4


proc Seq2 =
   'runn:10.'exec2:3.Seq2f + 'cald:5.nil

proc Seq2f =
   'find2:10.'exec1:3.nil + 'cald:5.nil

proc Par2 =
   'runn:10.'exec1:3.'exec2:3.nil + 'cald:5.nil

proc Workflow0 =
   (
***Instance:0:P1
   InitialState2[runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, ninit_0/ninit, comp_0/comp, canc_0/canc, exec_0/exec,
      prec_0/prec, cald_0/pcald] |

   Par2[runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, ninit_0/ninit, comp_0/comp, canc_0/canc, exec_0/exec,
      exec_1/exec1, exec_4/exec2] |

***Instance:1:S1
   InitialState2[runn_1/runn, cald_1/cald, cotd_1/cotd,
      find_1/find, ninit_1/ninit, comp_1/comp, canc_1/canc, exec_1/exec,
      prec_1/prec, prec_0/pprec, cald_0/pcald] |

   Seq2[runn_1/runn, cald_1/cald, cotd_1/cotd,
      find_1/find, ninit_1/ninit, comp_1/comp, canc_1/canc, exec_1/exec,
      exec_2/exec2, find_2/find2, exec_3/exec1] |

***Instance:2:A
   InitialState0[runn_2/runn, cald_2/cald, cotd_2/cotd,
      find_2/find, ninit_2/ninit, comp_2/comp, canc_2/canc, exec_2/exec,
      prec_1/pprec, cald_1/pcald] |
```

```
***Instance:3:B
   InitialState0[runn_3/runn, cald_3/cald, cotd_3/cotd,
      find_3/find, ninit_3/ninit, comp_3/comp, canc_3/canc, exec_3/exec,
      prec_1/pprec, cald_1/pcald] |

***Instance:4:S2
   InitialState2[runn_4/runn, cald_4/cald, cotd_4/cotd,
      find_4/find, ninit_4/ninit, comp_4/comp, canc_4/canc, exec_4/exec,
      prec_4/prec, prec_0/pprec, cald_0/pcald] |

   Seq2[runn_4/runn, cald_4/cald, cotd_4/cotd,
      find_4/find, ninit_4/ninit, comp_4/comp, canc_4/canc, exec_4/exec,
      exec_5/exec2, find_5/find2, exec_6/exec1] |

***Instance:5:B
   InitialState0[runn_5/runn, cald_5/cald, cotd_5/cotd,
      find_5/find, ninit_5/ninit, comp_5/comp, canc_5/canc, exec_5/exec,
      prec_4/pprec, cald_4/pcald] |

***Instance:6:C
   InitialState0[runn_6/runn, cald_6/cald, cotd_6/cotd,
      find_6/find, ninit_6/ninit, comp_6/comp, canc_6/canc, exec_6/exec,
      prec_4/pprec, cald_4/pcald] |

Basics4[comp_2/comp1, comp_3/comp2, comp_5/comp3, comp_6/comp4] |

'exec_0:3.pprec:5.nil | 'find_0:10.'rfind:10.nil

)\{
   runn_0, cald_0, cotd_0, find_0, ninit_0, comp_0, canc_0, exec_0, prec_0,
   runn_1, cald_1, cotd_1, find_1, ninit_1, comp_1, canc_1, exec_1, prec_1,
   runn_2, cald_2, cotd_2, find_2, ninit_2, comp_2, canc_2, exec_2, prec_2,
   runn_3, cald_3, cotd_3, find_3, ninit_3, comp_3, canc_3, exec_3, prec_3,
   runn_4, cald_4, cotd_4, find_4, ninit_4, comp_4, canc_4, exec_4, prec_4,
   runn_5, cald_5, cotd_5, find_5, ninit_5, comp_5, canc_5, exec_5, prec_5,
   runn_6, cald_6, cotd_6, find_6, ninit_6, comp_6, canc_6, exec_6, prec_6,
   pprec, pcald}
```

5.6.1 Model Checking PCCS Characterised Liesbet1 with Concurrency Workbench

For a simple model-checking test, with CWB-NC, we test the same proposition as that used in Section 5.1.4, save for the priority level, viz.

```
prop cotd =
```

```
min X = <->tt ∧ [-'rfind:10]X
```

This proposition asserts that, along all enactment paths, the model will reach a completed state. The result of the model checking exercise follows. A significant reduction in states and transitions is evident over the corresponding standard CCS model, viz. 53 v 833 in states, and 57 v 977 in transitions.

```
cwb-nc> chk Workflow0 find
Invoking alternation-free model checker.
Building automaton...
States: 53
Transitions: 57
Done building automaton.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(4.078,0.000,0.093,4.078)
cwb-nc>
```

5.7 Concluding Remarks

In this chapter (together with Appendix A), we have presented a comprehensive formalisation of the Liesbet meta-model using PCCS. The formalisation represents a contribution to the Business Process Management community. We argue that it trivially follows from this that a full CCS characterisation of the Liesbet meta-model is possible. That is, the only fundamental difference between the CCS- and PCCS-based characterisations is the use of the explicit scheduler in the CCS characterisation. As the issue of the scheduler is orthogonal to all other aspects of the respective characterisations, it stands to reason that a full CCS characterisation would trivially follow from the PCCS characterisation. We decided against explicitly defining a full CCS-based characterisation of Liesbet, as we believe it to not be sufficiently important to do so, in light of our PCCS-based characterisation.

As may be seen from the model checking example for Liesbet1, presented in Section 5.6.1, we are able to characterise Liesbet models using PCCS that have a significantly lower verification complexity (in terms of the size of the entailed state space) than models characterised using CCS. The principal reason for such a reduction is the absence of an explicit scheduler in the PCCS characterisation to enforce the intended semantics of Liesbet, presented in Section 3.2. Improved verification complexity was a key motivation for investigating the use of PCCS for the characterisation of Liesbet over CCS.

There is an interesting dichotomy at play in our PCCS-based characterisation of Liesbet. We could make the verification complexity of PCCS-characterised Liesbet models even better by using further priority levels to achieve an even better partial-order reduction (POR) on the state space. However, these are not strictly necessary to capture the intended semantics of Liesbet, which are sufficiently captured without their use, and they would greatly obscure the clarity of the PCCS-based characterisation of Liesbet. For instance, in the characterisation of synchronisation types, presented in Appendix Section A.3, we use many handshaking actions. These could be mutually-differently prioritised to effect better POR, but the order in which they occur is not important for the characterisation to be sound. In fact, we could remove some of the use of priorities in the

current characterisation, and still have a sound characterisation. These handshaking actions occur at a distinct level of priority from all other actions. We could soundly remove this dispensation, which arguably would make for better clarity in specification but at the cost of increased verification complexity.

Notably, even when we opt for maximising POR in order to reduce verification complexity as much as we can, the performance of verification under CWB-NC is still painfully slow for all but the simplest PCCS-characterised examples. An example is that of the Travel Agent model, presented in Section 4.5, which took several hours to return a result for checking whether the model completes along all enactment paths[2]. The principal reason for this is the inability of the CCS-based characterisations to practicably capture the intended semantics for Liesbet, presented in Section 3.2.

Our PCCS-based characterisation of Liesbet exposes the real weaknesses of using process algebra, such as CCS/PCCS, for the representation of workflow. Formalisms such as these suffer on at least two principal counts:

- It is not possible to arrive at the intended semantics for Liesbet without a lot of abstraction. It is only through abstraction that we may count more that one transition occurring at a time to be atomic, which is a key requirement of the intended semantics (in propagating effects of completing/cancelling childless instances up the tree, for instance). Although, CCS/PCCS has a notion of abstraction in distinguishing internal (τ) transitions from external ones, it is not possible to instruct CWB-NC to take account of this difference in constructing the state space of models. The lack of such a capability is hardly surprising, i.e., a CCS/PCCS model is fundamentally characterised by all of its transition types, the distinction between external and internal transition types is purely cosmetic. As such, to perform model checking on a CCS/PCCS, as CWB-NC does, it would always be necessary to construct the state space for a model accounting for all transition types, at least initially. It is the construction of the entire space that kills CWB-NC when used for verification of PCCS-characterised Liesbet models.

- The efficiency (and clarity) of performing queries as part of progressing synchronisation types is not good. *In order to carry out a single atomic query, there is no limit to the number of* instances that may be need to be queried as to their state. All of these individual queries themselves require several transitions. The state space for querying alone quickly explodes. Again, this is behaviour that needs to be captured as atomic, together with the consequences of completing/cancelling synchronisation instances being atomically propagated.

Interestingly, it is quite evident that Petri nets would not fair any better in characterising Liesbet than our CCS-based characterisations do. The principal reason lies in us making the recording of the state of activities explicit. Because of this, Petri nets would handle the characterisation of Liesbet in largely the same way in having tracker, generic type and scheduler agents. Moreover, the same shortcomings in the expression and evaluation of synchronisation conditions would exist.

It is also notable that none of the problems asserted (in Section 2.3.2) for Petri net-based characterisations of the YAWL patterns would exist in a Petri net-based characterisation of Liesbet. These

[2]On a 3.2GHz Linux box with 1GB RAM.

problems were concerned with: tracking multiple-instances, advanced synchronisation and cancellation. This is because we resolve these issues at the information view (i.e. in defining `Liesbet`) prior to any characterisation using Petri nets/CCS/PCCS. This is a point that is discussed further in the conclusions to this thesis, in Chapter Twelve.

It is worth noting, purely subjectively, that the specification of semantics for the generic type agents is quite clear and succinct. It is evidently appealing to be able to express the semantics using the programming-like, compositional constructs of CCS/PCCS. The down-side of using such a language is that we would want the operational semantics that are associated with it to admit the notion that multiple transitions may occur atomically, as we have stated. We would imagine that this would be quite difficult to achieve in a process algebra such as CCS/PCCS. Thus, we have some clarity (especially when compared with the Situation Calculus characterisation, presented in the next chapter) at the cost of atomicity, which is another apparent dichotomy.

We carry our experiences of using CCS/PCCS to characterise `Liesbet` over to the next chapter where we consider the characterisation of `Liesbet` using a logic-based formalism, namely the Situation Calculus. It proves interesting to see how the two evidently contrasting formalisms differ in the characterisation of `Liesbet`.

Chapter 6

Situation-Calculus Based Semantics

In this chapter, we present our Situation Calculus-based characterisation of `Liesbet`. A motivation for investigating the use of the Situation Calculus was that, in being a logic-based formalism, it is quite different to a process algebra based approach for characterising the behaviour of dynamic systems. Moreover, we felt that certain aspects in which CCS/PCCS may be deficient may be better addressed using the Situation Calculus, and vice-versa, making the investigation of using the Situation Calculus to characterise `Liesbet` complementary to the investigation of using CCS/PCCS.

We proceed with an introduction to the Situation Calculus (`SitCalc`), followed by a presentation of our `SitCalc`-based characterisation of `Liesbet`, deferring some aspects of the presentation to Appendix B to save space. Then, we present the definition of a translation function, $\mathcal{M}_{SitCalc}[\![-]\!]$, which translates `Liesbet` models to `SitCalc`, as well as a couple of results: one regarding the completion (in enactment) of `SitCalc`-based `Liesbet` models; and the other demonstrating that the characterisations presented in Section 3.4 of `Liesbet` constructs as abbreviations, $\texttt{Liesbet}_{abbrev}$, are sound. At the end of the chapter, we present a discussion as to the relative merits of a logic-based approach, i.e. using the Situation Calculus, versus one based on process algebra such as CCS/PCCS.

6.1 Introduction to the Situation Calculus

The Situation Calculus (`SitCalc`), originally thanks to McCarthy [76] and McCarthy and Hayes [77], is a framework for the *description of dynamic domains.* A significant contribution to the definition of `SitCalc` has been made by Reiter, and a number of his colleagues, over many years. A good summary of these efforts is presented in [98].

The language, $\mathcal{L}_{SitCalc}$, is second-order with equality; although from a domain engineer's perspective, it is essentially a first-order framework – it prescribes just a single second-order axiom. It is many-sorted, having three distinguished sorts: *action*, *situation* and *object*; where *object* counts as an aggregating sort for an unbounded number of other domain-dependent sorts.

Actions are the only means by which changes are made to the world. They have prescribed

effects on *fluents*, which are mutable domain properties. Situations are defined inductively from the initial situation S_0, using the distinguished function *do*. The application $do(\alpha, s)$ denotes the situation that follows from performing the action α in situation s. Thus, situations represent *histories* of actions on the initial situation. The notion of situation is not synonymous with that of domain state; two situations may be different and yet may assign the same truth values to all fluents comprising the state of a domain. This is a distinguishing feature of Reiter's formulation of `SitCalc` [98].

Assuming a standard alphabet of connectives and quantifiers, $\mathcal{L}_{sitcalc}$ has the following alphabet [98]:

- Countably infinitely many individual variable symbols of each sort. For actions (resp. situations), we use a (resp. s), and subscripted and superscripted variations thereof. For variables of sort object, we use lower-case roman letters other than a and s, with possible sub/superscripting. In addition, $\mathcal{L}_{sitcalc}$ includes countably infinitely many predicate variables of all arities.

- Two function symbols of sort situation: the constant S_0 and $do : action \times situation \rightarrow situation$.

- Two binary predicate symbols: $\sqsubset$: $situation \times situation$, defining an ordering relation on situations (that is, $s \sqsubset s'$ means that s is a sub-history of s'); and $Poss : action \times situation$ meaning that it is possible to perform the action a in situation s.

- For each $n \geq 0$, countably infinitely many predicate symbols of sort $(action \cup object)^n$, for situation-independent relations like $human(Joe)$, $oddNumber(m)$.

- For each $n \geq 0$, countably infinitely many function symbols of sort $(action \cup object)^n \rightarrow object$, for situation-independent functions like $sqrt(x)$, $height(MtEverest)$.

- For each $n \geq 0$, a finite, or countably infinite, number of function symbols of sort $(action \cup object)^n \rightarrow action$. These are *action functions*, and denote actions such as $pickup(x)$ and $move(A, B)$. They are distinguished by the requirement that they be axiomatised by *action precondition axioms*, as described below. In most applications, they are finite in number.

- For each $n \geq 0$, a finite, or countably infinite, number of function symbols of sort $(action \cup object)^n \times situation \rightarrow action \cup object$. These are *functional fluents* and denote functions whose value is situation-dependent, such as $age(Mary, s)$, or $primeMinister(Italy, s)$. Functional fluents always take just one argument of sort *situation*, and it is always the last argument. In most applications, they are finite in number.

- For each $n \geq 0$, a finite, or countably infinite, number of predicate symbols with arity $n + 1$ and sort $(action \cup object)^n \times situation$. These are *relational fluents* and denote relations whose value is situation-dependent, such as $ontable(x, s)$, or $husband(Mary, John, s)$. Relational fluents always take just one argument of sort *situation*, and it is always the last argument. In most applications, they are finite in number.

Note that, in this chapter, the scope of quantifiers should be taken to be remainder of the formula, from where they are used, up to the quantifier name being used again.

There are *four domain-independent, foundational axioms*, Σ, described for `SitCalc` [98], where $s \sqsubseteq s'$ is an abbreviation for $s \sqsubset s' \vee s = s'$, viz.

$$do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2 \tag{6.1}$$

$$(\forall P).P(S_0) \wedge (\forall a, s)[P(s) \supset P(do(a, s))] \supset (\forall s)P(s) \tag{6.2}$$

$$\neg s \sqsubset S_0 \tag{6.3}$$

$$s \sqsubset do(a, s') \equiv s \sqsubseteq s' \tag{6.4}$$

The first of these axioms (1) is a unique names axiom for situations, defining a necessary condition for two situations to be equal; namely that they be derived from an application of the same action to the same situation.

The second of these (2) is the single (prescribed) second-order axiom, and defines the domain of situations to be the smallest set which includes the initial situation, S_0, that is closed under the application of the function *do* to an action and a situation. This captures the notion that *situations* are *finite sequences of actions.*

From [98], we note that any model of these two axioms will have as its domain of situations the smallest set $\mathcal{S}$ satisfying:

- $\sigma_0 \in \mathcal{S}$, where σ_0 is the interpretation of S_0 in the model.
- If $\sigma \in \mathcal{S}$, and $A \in \mathcal{A}$, then $do(A, \sigma) \in \mathcal{S}$, where $\mathcal{A}$ is the domain of actions in the model.

The first two axioms imply that two situations will be the same iff they result from the same sequence of actions applied to the initial situation, S_0. The other two axioms, (3) and (4), capture the notion of *a sequence of actions preceding another* – that is, the notion of a *sub-history.* The operator $\sqsubset$ provides an ordering on situations, where $s \sqsubset s'$ means that the action sequence, or situation, s' can be obtained from s by applying one or more actions to s.

From Σ, it can be shown that the situations in any model $\mathbb{M}$ of Σ can be represented as a tree – a *situation tree*, where every node branches on all elements of *Act* (which, along with *Obj* and *Sit* partition the domain of $\mathbb{M}$, according to the sorts *action*, *object* and *situation*, respectively).

A *basic `SitCalc` action theory* (BAT), $\mathcal{D}$, consists of the foundational axioms, Σ, as well as a number of other *domain-dependent* axioms, viz.

- *Action Precondition Axioms*, constituting $\mathcal{D}_{ap}$, with the form:

  $Poss(A(x_1, \ldots, x_n), s) \equiv \Pi_A(x_1, \ldots, x_n, s)$,

  where A is an n-ary action function symbol and $\Pi_A(x_1, \ldots, x_n, s)$ is a formula that is uniform in s (that is, determined according to the current situation s, alone) and whose free variables are among $x_1, \ldots, x_n, s$.

- *Successor-state Axioms*, constituting $\mathcal{D}_{ssa}$:

  - For relational fluents, with the form:

    $F(x_1, \ldots, x_n, do(a, s)) \equiv \Phi_F(x_1, \ldots, x_n, a, s)$,

    where F is a $n+1$-ary predicate symbol and $\Phi_F(x_1, \ldots, x_n, a, s)$ is a formula which is uniform in s, and whose free variables are among $x_1, \ldots, x_n, a, s$.

– For functional fluents, with the form:
$f(x_1, \ldots, x_n, do(a, s)) = y \equiv \phi_f(x_1, \ldots, x_n, y, a, s)$,
where f is a $n+1$-ary function symbol and $\phi_f(x_1, \ldots, x_n, y, a, s)$ is a formula which is uniform in s, and whose free variables are among $x_1, \ldots, x_n, y, a, s$.

- Unique-name axioms for actions, constituting $\mathcal{D}_{una}$, which state that for any two actions to be identical, they must have identical function symbols and identical arguments:
  – For distinct action names, A and B,
  $A(\vec{x}) \neq B(\vec{y})$
  – Identical actions have identical arguments
  $A(x_1, \ldots, x_n) = A(y_1, \ldots, y_n) \supset x_1 = y_1, \ldots, x_n = y_n$.
- $\mathcal{D}_{S_0}$, the initial state of the domain – a set of first-order sentences, uniform in the initial situation S_0.

We also require that models of basic action theories satisfy the following *fluent consistency property*, which ensures that a functional fluent has just one value y for a given set of parameters and situation. Suppose that f is a functional fluent whose successor state axiom in $\mathcal{D}_{ssa}$ is:

$$f(x_1, \ldots, x_n, do(a, s)) = y \equiv \phi_f(x_1, \ldots, x_n, y, a, s).$$

Then,

$$\mathcal{D} \models (\forall \vec{x})\,(\exists y)\phi_f(x_1, \ldots, x_n, y, a, s) \wedge$$
$$((\forall y, y')\phi_f(x_1, \ldots, x_n, y, a, s) \wedge \phi_f(x_1, \ldots, x_n, y', a, s) \supset y = y')$$

Situation trees represent the evolution of situations, according to the application of all actions within the domain of actions. Notably, the application of an action to a particular situation may not always be possible, according to $\mathcal{D}_{ap}$. This means that certain sequences of actions, i.e. "ghost" situations, within the situation tree may not be possible. Typically, it is desirable to ignore these sequences within models of a `SitCalc` theory. To mark those sequences that should not be ignored, we define the notion of an *executable situation*. All of the actions named within such a situation must be executable, according to $\mathcal{D}_{ap}$, in their respective situations of application. We include the following abbreviation in the axiomatisation of `SitCalc`, which defines an executable situation to be one where it is possible to execute all of the actions occurring in the action sequence:

$$executable(s) \stackrel{def}{=} (\forall a, s').do(a, s') \sqsubseteq s \supset Poss(a, s').$$

6.2 SitCalc-based Semantics for Liesbet

In this section, we concentrate on presenting the `SitCalc`-based characterisation of just a handful of `Liesbet` constructs so as not to disrupt the presentation with too many small points of detail. A presentation of the `SitCalc` characterisation for the remaining constructs is given in Appendix B.

6.2.1 Par(Seq(A,B),Seq(C,D)) – A Simple Example

We start this section with an example, in order to ground the presentation of the `SitCalc` semantics for `Liesbet`. In Figure 6.1, we see a graphical representation of a simple workflow model:

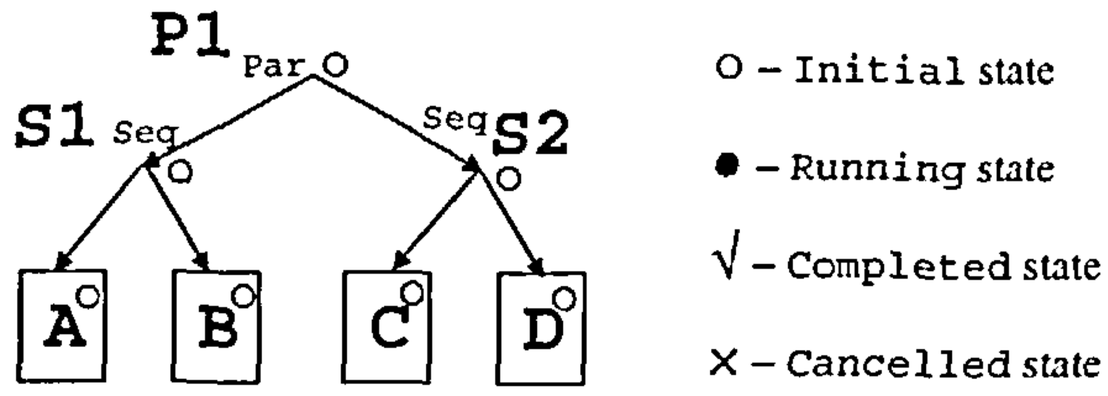


Figure 6.1: Enactment State 0 of `Par(Seq(A,B),Seq(C,D))`
Possible Successor States: 1

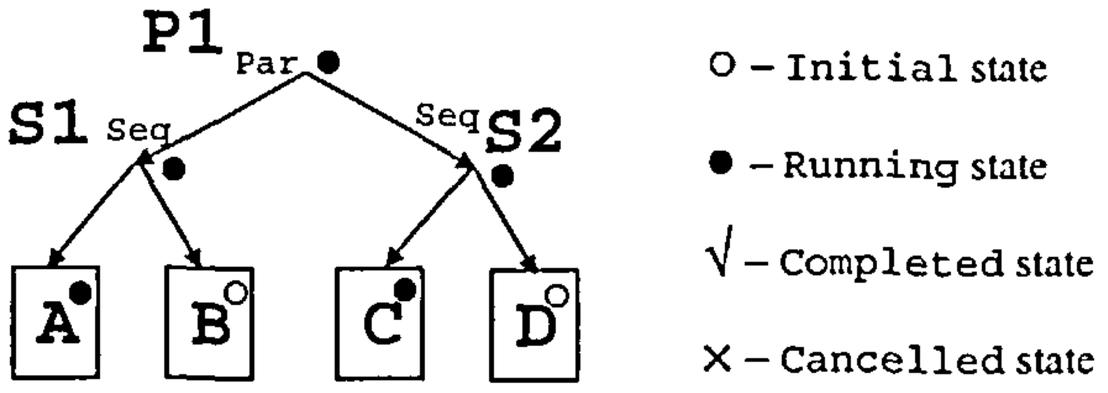


Figure 6.2: Enactment State 1 of `Par(Seq(A,B),Seq(C,D))`
Possible Successor States: 2, 8

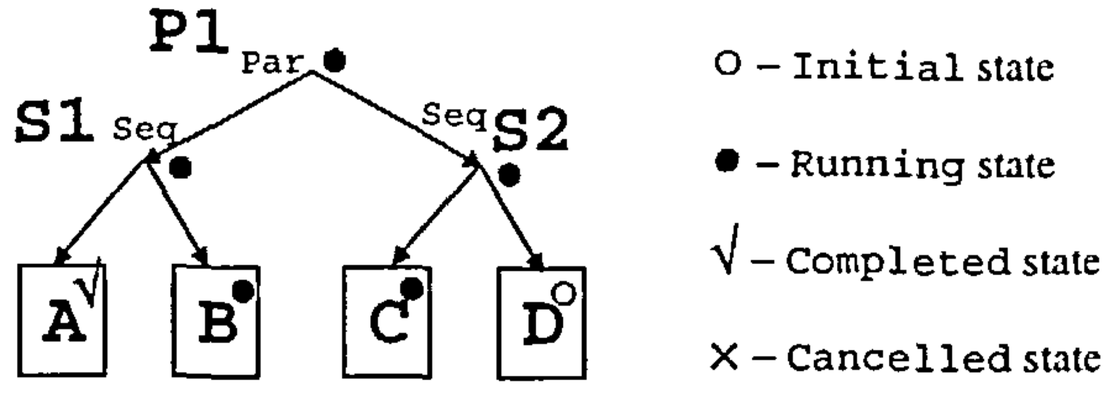


Figure 6.3: Enactment State 2 of `Par(Seq(A,B),Seq(C,D))`
Possible Successor States: 3, 6

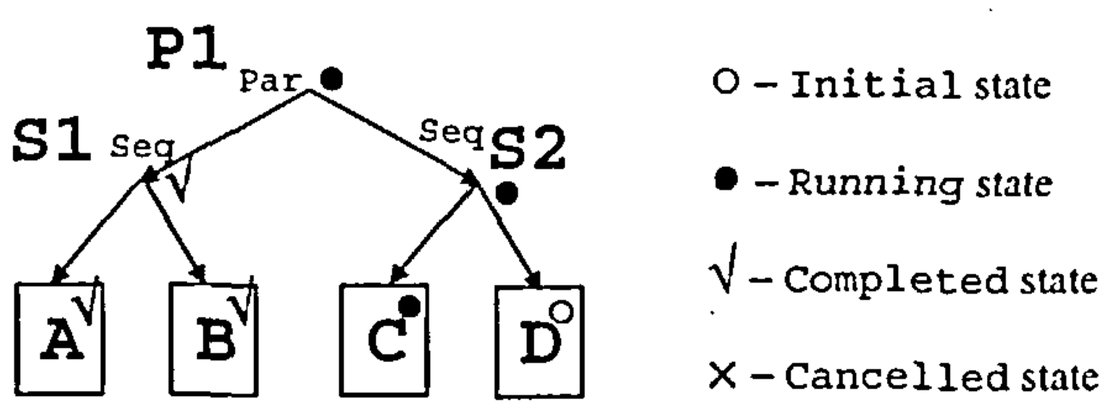


Figure 6.4: Enactment State 3 of `Par(Seq(A,B),Seq(C,D))`
Possible Successor States: 4

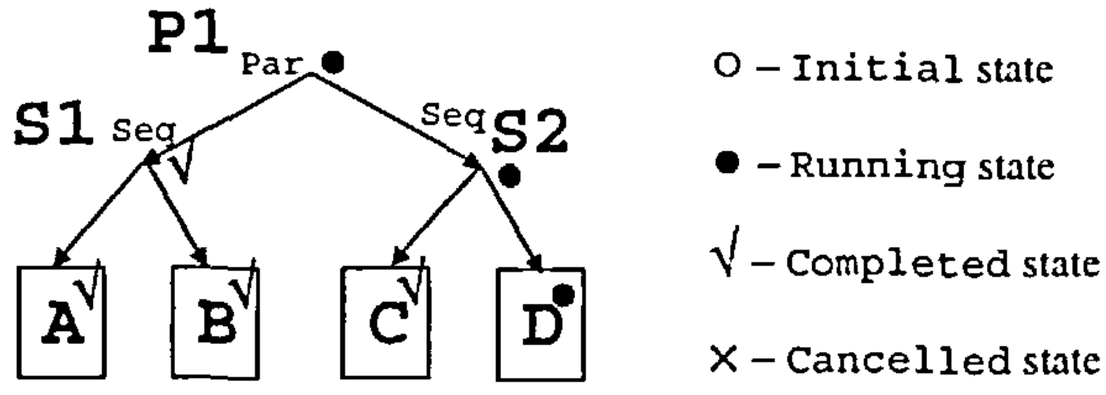


Figure 6.5: Enactment State 4 of `Par(Seq(A,B),Seq(C,D))`
Possible Successor States: 5

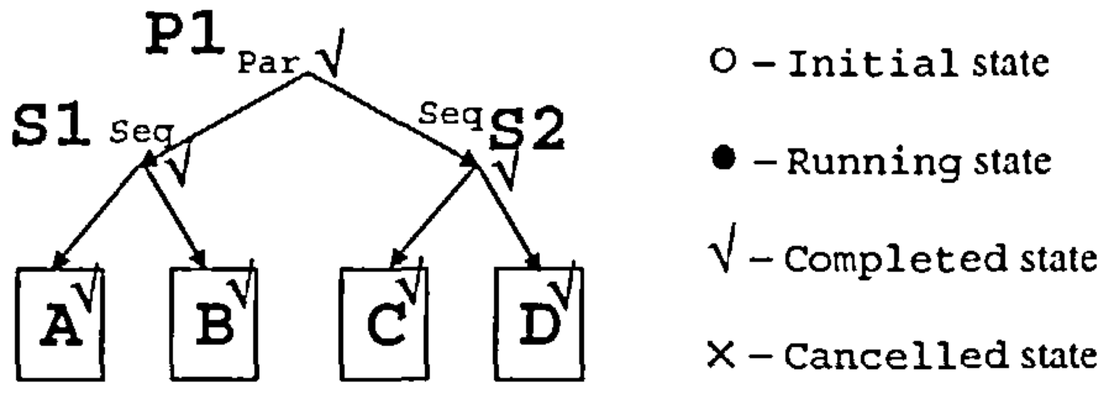


Figure 6.6: Enactment State 5 of `Par(Seq(A,B),Seq(C,D))`

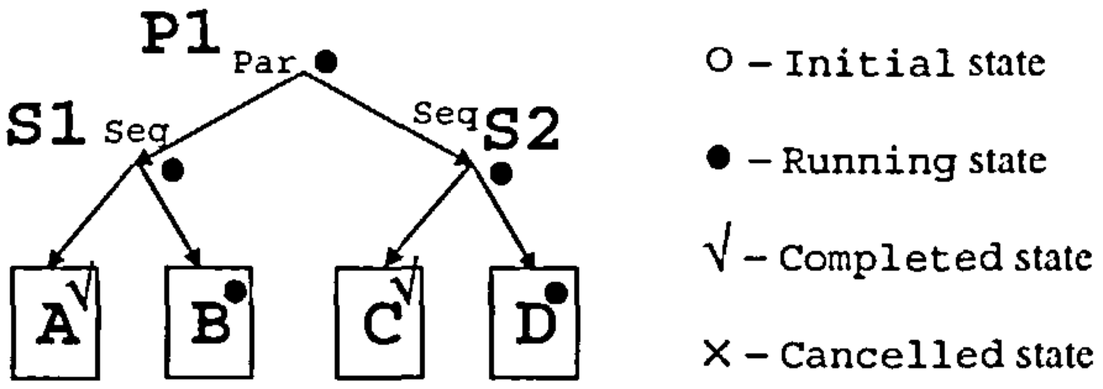


Figure 6.7: Enactment State 6 of `Par(Seq(A,B),Seq(C,D))`

Possible Successor States: 4, 7

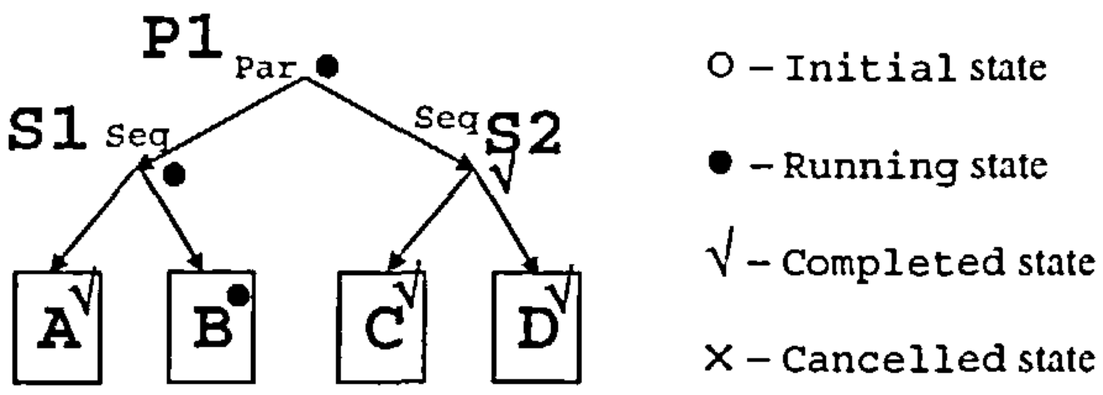


Figure 6.8: Enactment State 7 of `Par(Seq(A,B),Seq(C,D))`

Possible Successor States: 5

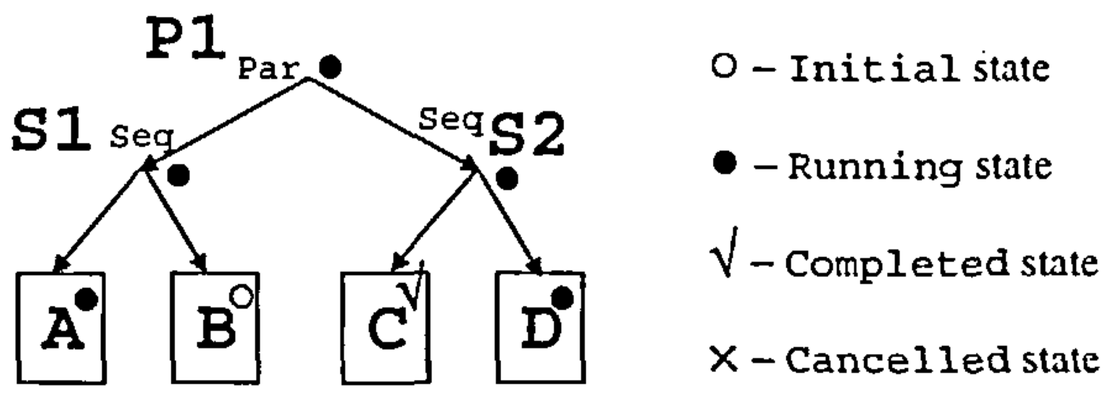


Figure 6.9: Enactment State 8 of `Par(Seq(A,B),Seq(C,D))`

Possible Successor States: 6, 9

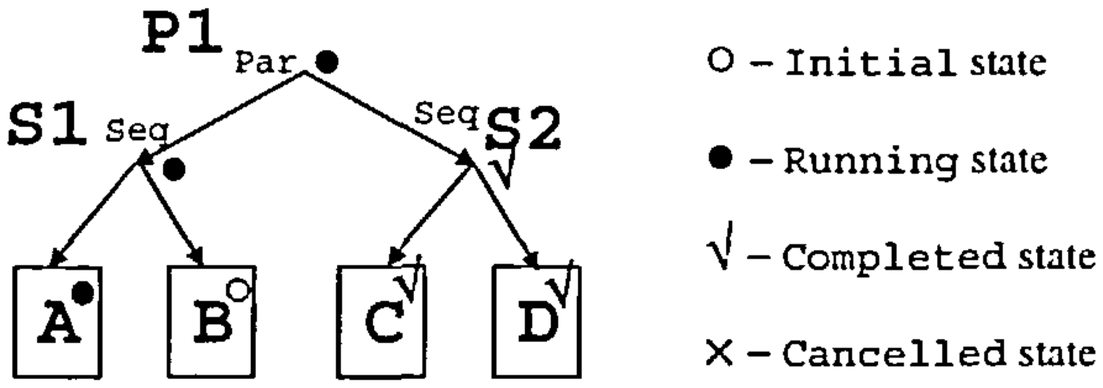


Figure 6.10: Enactment State 9 of `Par(Seq(A,B),Seq(C,D))`

Possible Successor States: 7

Par(Seq(A,B), Seq(C,D)), that we have used for illustrative purposes throughout this thesis. In the initial state of the workflow model (state 0), all activity instances are in an Initial state, as can be seen.

When the root activity instance (P1) is set running (Figure 6.2), execution is also propagated to appropriate descendants, resulting in state 1. Here, the two sequences, S1 and S2, which are the children of P1, are set running, as well as their respective first children, A and C. A key theme of these semantics is that actions that are performed on one instance, namely, completion, cancellation or execution, may have side-effects on other instances. In this case, the side-effect is to propagate execution downwards. Propagation of side-effects largely happens to ancestors or descendants of the activity on which an action is being performed. Thus, *descendant* and *child* are key relations in the semantics presented here.

In state 1 of the workflow model, we have the option to complete either basic instances A, or C[1]. Completing A takes us to state 2, represented in Figure 6.3. As can be seen, an effect of completing A is to set the second basic instance of the sequence S1, namely B, running. In state 2, we have the option of completing B, or C. Let's say that we complete B, taking us to state 3 (as shown in Figure 6.4). Completing B has the effect of also completing the sequence S1. That is, completion is propagated upwards.

In state 3, we may only complete basic instance C, and doing so has the effect of setting instance D running, as can be seen from Figure 6.5. Then, in state 4, we may only complete D, which has the effect of completing S2, P1 and thus the model as a whole, which can be seen in Figure 6.6, which is state 5.

There are some alternative enactments that are possible, which we shall now elaborate.

- In state 2, we may instead complete C, which results in the model state (#6) shown in Figure 6.7. If we then complete B, we arrive at a *matched state*. In SitCalc terms, a matched state is a situation which has the same fluent state as a previously visited situation, and, in verification, we would backtrack. In this case, the matched state is 4.

- In state 6, we may instead complete D, which results in the model state (#7) shown in Figure 6.8. If we then complete B, we arrive at another matched state, namely state 5.

- In state 1, we may instead complete C, which results in the model state (#8) shown in Figure 6.9. Completing A then takes us to matched state 6.

- Alternatively, in state 8, if we complete D, we arrive at model state (#9), shown in Figure 6.10. In this state, the sequence S2 and its descendants C and D have completed, but, A, B (and S1) are still running. Completing A then takes us to matched state 7.

In summary, there are ten distinct states for this example workflow model, shown in Figures 6.1–6.10. We now proceed to give an overview of how appropriate semantics for Liesbet, as exemplified here, are realised using SitCalc.

6.2.2 Introducing SitCalc-based Semantics for Liesbet

We start our presentation of the SitCalc-based semantics for Liesbet by describing how the example model, presented in the previous section, may be characterised using SitCalc. Firstly,

[1]For this example, we consider that basic instances may only be completed. (We do not consider cancellation.)

```
Activity(0, 0, 0,  GId_PAR, NONE, NONE, NONE, S0) //P
Activity(0, 1, 1,  GId_SEQ, NONE, EXEC, NONE, S0) //S1
Activity(0, 2, 4,  GId_SEQ, NONE, EXEC, NONE, S0) //S2
Activity(1, 3, 2,  GId_BAS, NONE, EXEC, NONE, S0) //A
Activity(1, 4, 3,  GId_BAS, NONE, NONE, NONE, S0) //B
Activity(2, 5, 5,  GId_BAS, NONE, EXEC, NONE, S0) //C
Activity(2, 6, 6,  GId_BAS, NONE, NONE, NONE, S0) //D
```

Figure 6.11: Representation of `Par(Seq(A,B),Seq(C,D))` in `SitCalc`

for any `SitCalc`-characterised `Liesbet` model, there are a number of actions which may be carried out on the model. These actions are to *complete* or *cancel* an activity. Additionally, all activity instances extant in a model need to be *added* prior to any enactment of the model[2]. We use the term *Current Workflow State* (CWS) to mean the aggregation of instances which have already been added, plus their respective states. This is different from the notion of a `SitCalc` domain/action theory, or BAT.

There is a partial ordering over these different action types, reflecting certain priorities, which is enforced by the action precondition axioms, presented a little later:

- Adding (`add_activity/7`) activity instances to the CWS has highest priority.
- Completing or cancelling (`complete,cancel/1,3`) a (childless) structured instance has next priority.
- Completing or cancelling (`comp_bas,canc_bas/1`) a basic instance has lowest priority.

Not all instances will be completed or cancelled by *explicit* occurrences of these action types. Often an instance may be *executed*, completed or cancelled as an *implicit side-effect* of an action being effected on another instance. We have already seen this phenomenon in the example presented in the previous section (6.2.1).

A representation in `SitCalc` of the `Liesbet` model discussed in Section 6.2.1 is given in Figure 6.11. There are also a number of, what we consider to be, foundational axioms for workflow, presented in Figure 6.12. These are domain-independent axioms. The `SitCalc` characterisation of a `Liesbet` model is the sum of these foundational axioms and a set of `Activity/8` initial state atoms which complete the specification of a particular workflow model (as presented in Figure 6.11, for the example in question). This sum of these axioms constitutes a Basic Action Theory (or BAT) in `SitCalc`.

Note that for simplicity, in the following account, we only allow completion (and not cancellation) of basic instances. We use a number of domain-independent and domain-dependent identifiers, such as `GId_BAS`, or `CId_A`, respectively, which resolve to natural numbers.

The representation admits the possibility of just two actions, as indicated by the action precondition axioms which comprise the definition of `Poss/2`.

---

[2]Apart from dynamically-added instances, i.e. execution activity (`ExecAct`) instances (or descendants thereof), which are added as `Multi/MultiSeq` types (described in Section 3.1.15) are enacted.

Σ

```
Poss(comp_bas(i),s) ≡ State(i,s)=Running ∧ GType(i,s)=GId_BAS ∧
                      ¬(∃p,i',c,g,sc,f,j).Poss(add_activity(p,i',c,g,sc,f,j),s)

Poss(add_activity(p,i,c,g,sc,f,j),s) ≡ Activity(p,i,c,g,sc,f,j,s) ∧
            ¬(∃p',i',c',g',sc',f',j').Activity(p',i',c',g',sc',f',j',s) ∧ i'<i

State(i,do(a,s))=st ≡ StateChange(i,a,st,s) ∨ State(i,s)=st ∧ ¬(∃st').StateChange(i,a,st',s)

StateChange(i,a,st,s) ≡ (∃p,c,g,sc,f,j).a=add_activity(p,i,c,g,sc,f,j) ∧ SetRunning(p,i,f,st,s) ∨
          Completing(i,a,st) ∨ (∃i').CompletingAction(i',a) ∧ PropagateCompleteUp(i',i,st,s)

SetRunning(p,i,f,st,s) ≡ p=i ∧ st=Running ∨
                         State(p,s)=Running ∧ (f=EXEC ∧ st=Running ∨ ¬f=EXEC ∧ st=Initial) ∨
                         ¬State(p,s)=Running ∧ st=Initial

Completing(i,a,st) ≡ a=comp_bas(i) ∧ st=Completed

CompletingAction(i,a) ≡ a=comp_bas(i)

PropagateCompleteUp(i',i,st,s) ≡ (∃i'').AllDescSiblingsFinished(i',i'',s) ∧
        (st=Completed ∧ i=i'' ∨ ExecuteNextChild(i'',i,st,s)) ∨ ExecuteNextChild(i',i,st,s)

AllDescSiblingsFinished(i',i,s) ≡ Descendant(i,i',s) ∧
                                  (∀d).(¬d=i' ∧ Descendant(i,d,s) ∧ ¬Descendant(d,i',s) ∧
                                  ¬Descendant(i',d,s)  ⊃ State(d,s)=Completed)

ExecuteNextChild(i',i,st,s) ≡ (∃p,i'').Child(p,i',s) ∧  PropagateRunningDownInc(i'',i,st,s) ∧
                                                  GType(p,s)=GId_SEQ ∧ NextInitialChild(p,i'',s)

NextInitialChild(p,i,s) ≡ Child(p,i,s) ∧ ¬(∃i').(Child(p,i',s) ∧ i'<i ∧ State(i',s)=Initial)

PropagateRunningDownInc(i',i,st,s) ≡ (∃i''). PropagateRunningDownInc(i'',i,st,s) ∧
                                                                  Executes(i',i'',s) ∨
                                                          i'=i ∧ st=Running

Child(p,i,do(a,s)) ≡ (∃c,g,sc,f,j).a=add_activity(p,i,c,g,sc,f,j) ∧ ¬p=i ∨ Child(p,i,s)

Descendant(anc,i,do(a,s)) ≡ (∃p,c,g,sc,f,j).a=add_activity(p,i,c,g,sc,f,j) ∧ ¬p=i ∧
                            (p=anc ∨ Descendant(anc,p,s)) ∨ Descendant(anc,i,s)

Executes(p,i,do(a,s)) ≡ (∃p,c,g,sc,f,j).a=add_activity(p,i,c,g,sc,f,j) ∧ ¬p=i ∧ f=EXEC ∨
                                                                  Executes(p,i,s)

Activity(p,i,c,g,sc,f,j,do(a,s)) ≡ ¬a=add_activity(p,i,c,g,sc,f,j) ∧ Activity(p,i,c,g,sc,f,j,s)
```

Figure 6.12: SitCalc Foundational Axioms for Workflow

```
Poss(comp_bas(i),s) ≡ State(i,s)=Running ∧ GType(i,s)=GId_BAS ∧
¬(∃p,i',c,g,sc,f,j).Poss(add_activity(p,i',c,g,sc,f,j),s)
Poss(add_activity(p,i,c,g,sc,f,j),s) ≡ Activity(p,i,c,g,sc,f,j,s) ∧
¬(∃p',i',c',g',sc',f',j').Activity(p',i',c',g',sc',f',j',s) ∧ i'<i
```

The first of these says that it is possible to complete a basic instance (which is of generic type `GId_BAS`), in the current situation, iff it is running and as long as it is not possible to add an activity instance to the CWS (using `add_activity/7`). Meanwhile, the action precondition axiom for `add_activity/7` says that it is possible to add an activity instance, `i`, to the CWS, in the current situation, iff it is yet-to-be-added (as indicated by an instance of the `Activity/8` fluent, pertaining to `i`, holding in the current situation) and there is no activity yet-to-be-added with a lower instance identification number.

The remaining foundational axioms, presented in Figure 6.12, are mainly successor-state axioms for fluents that are used in the characterisation of `Liesbet` models. The principal fluent in the `SitCalc`-based characterisation of `Liesbet` is `State/2`, viz.

```
State(i,do(a,s))=st ≡ StateChange(i,a,st,s) ∨ State(i,s)=st ∧ ¬(∃st').StateChange(i,a,st',s)
StateChange(i,a,st,s) ≡ (∃p,c,g,sc,f,j).a=add_activity(p,i,c,g,sc,f,j) ∧ SetRunning(p,i,f,st,s) ∨
   Completing(i,a,st) ∨ (∃i').CompletingAction(i',a) ∧ PropagateCompleteUp(i',i,st,s)
```

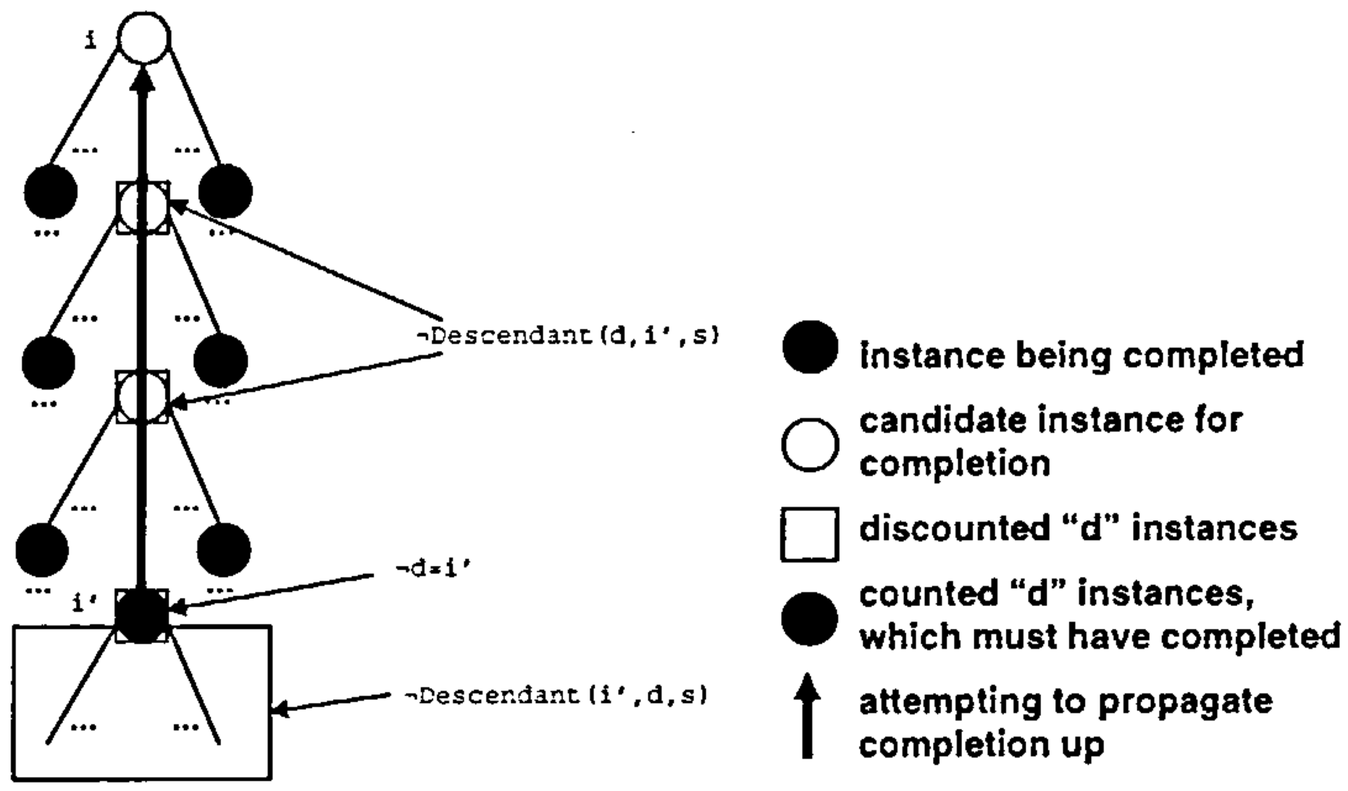
This functional fluent is inertial, i.e. instances of it only change in value according to prescribed action occurrences. The predicate `StateChange(i,a,st,s)` prescribes the circumstances according to which an instance `i` may change its state to `st`, in situation `s`, as a consequence of the occurrence of action `a`.

The first case to consider is when we are adding an instance to the CWS. As already described, adding instances is effected using `add_activity(p,i,c,g,sc,f,j)`, where `i` is a (unique) identifier – a natural number – for the instance being added, `p` is the instance number of `i`'s parent instance, `c` (resp. `g`) is an identifier specifying the customised (resp. generic) type of the instance, and `f` is a multi-purpose flag. The remaining parameters are used as we build on this initial characterisation. Specifically, `sc` is a flag indicating whether the activity instance is an isolated scope (see Section 3.1.3), and `j` is the join condition instance of a (join condition, execution activity) pair (of an instance of a `Multi/MultiSeq` type) when the instance being added is the corresponding execution activity instance.

The predicate `SetRunning/5`

```
SetRunning(p,i,f,st,s) ≡ p=i ∧ st=Running ∨
                         State(p,s)=Running ∧ (f=EXEC ∧ st=Running ∨ ¬f=EXEC ∧ st=Initial) ∨
                         ¬State(p,s)=Running ∧ st=Initial
```

is used to determine the effects of adding an activity on `State/2`, i.e. whether the instance should be put directly into the `Running` state or into the `Initial` state. If the parent instance identifier, `p`, is the same as the identifier for the instance being added, `i`, this indicates that `i` is the root instance of the workflow model. This particular instance is always set running, when it is added to the CWS. For all other instances, the following rules apply. If the parent instance *is not* running,




```
AllDescSiblingsFinished(i',i,s) ≡
   Descendant(i,i',s) ∧ (∀d).(¬d=i' ∧ Descendant(i,d,s) ∧ ¬Descendant(d,i',s) ∧
                              ¬Descendant(i',d,s)  ⊃ State(d,s)=Completed)
```

Figure 6.13: Depiction of `AllDescSiblingsFinished/3`

In the presented instance tree, (the black) instance `i'` is the one being completed. As a consequence of the action on `i'`, we may complete (the clear) instance `i` iff all descendants of `i`, bar those in boxes, have completed. The pertaining descendant instances, i.e. the counted `d` instances, are those shown in grey. Note that we show `i'` having children as a general case. For the case where `i'` is a basic instance being completed, it will have none.

---

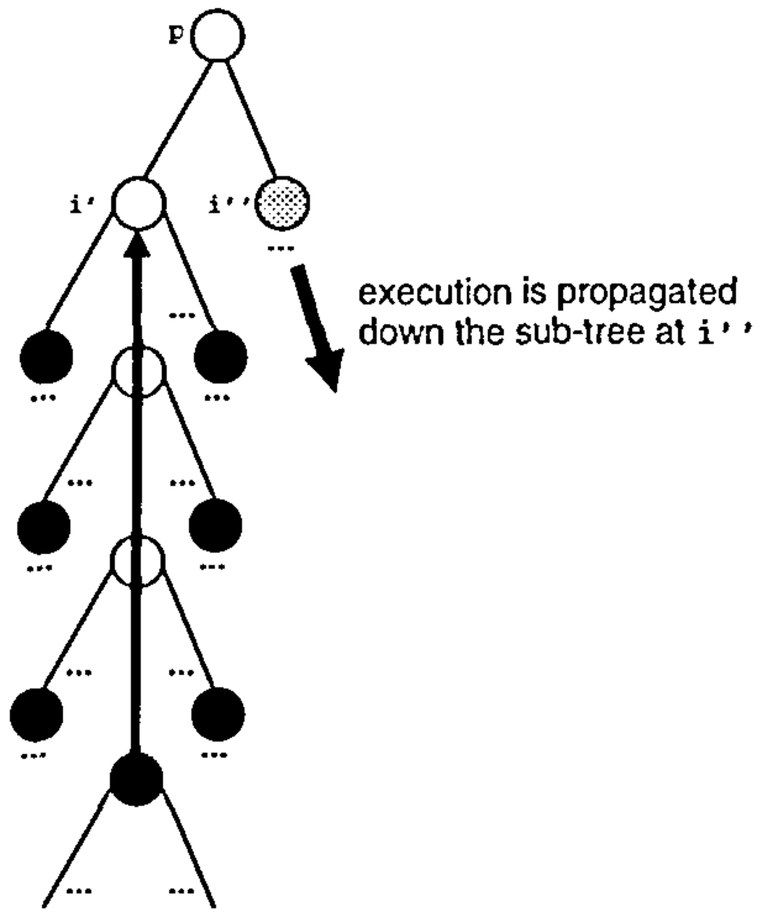
then the instance is set to the `Initial` state. If the parent instance *is* running, then the value of the multi-purpose flag, `f`, is used to determine whether the instance should be set running. The value of this flag is prescribed by the assumed translator for `Liesbet` models to `SitCalc`, presented in Section 6.3. For children of `Par` types, and for the first children of `Seq` types, the flag is set to `EXEC`; otherwise, for now, it is set to `NONE`. A value of `EXEC` indicates that the instance should be set running on the basis of its parent running.

We also capture the effects that the completion of a basic instance has on the CWS. We use the predicate `Completing/3` to prescribe that a completing basic instance should be assigned the `Completed` state. We use the predicate `CompletingAction/2` to signify (for now) a completing action on a basic instance.

```
Completing(i,a,st) ≡ a=comp_bas(i) ∧ st=Completed
CompletingAction(i,a) ≡ a=comp_bas(i)
```

On completing a basic instance, we propagate completion upwards as far as possible. We use `PropagateCompleteUp/4` for this purpose.

```
PropagateCompleteUp(i',i,st,s) ≡ (∃i'').AllDescSiblingsFinished(i',i'',s) ∧
        (st=Completed ∧ i=i'' ∨ ExecuteNextChild(i'',i,st,s)) ∨ ExecuteNextChild(i',i,st,s)
```




```
ExecuteNextChild(i',i,st,s) ≡
    (∃p,i'').Child(p,i',s) ∧  PropagateRunningDownInc(i'',i,st,s) ∧
                        GType(p,s)=GId_SEQ ∧ NextInitialChild(p,i'',s)
```

Figure 6.14: Depiction of `ExecuteNextChild/4`

If completion propagates up to an instance `i'`, then we check to see whether its parent `p` has any children left in the `Initial` state. If so, execution is propagated down the sub-tree rooted at the next child `i''` which is in the `Initial` state.

This predicate will complete an instance `i` iff the basic instance, `i'`, being completed is a descendant of `i`, and all siblings of activity instances on the path (in the activity tree) from `i'` to `i` (exclusively) have completed. It uses the predicate `AllDescSiblingsFinished/3` to determine this.

```
AllDescSiblingsFinished(i',i,s) ≡ Descendant(i,i',s) ∧
                                  (∀d).(¬d=i' ∧ Descendant(i,d,s) ∧ ¬Descendant(d,i',s) ∧
                                        ¬Descendant(i',d,s)  ⊃ State(d,s)=Completed)
```

The operation of this predicate is depicted in Figure 6.13. Instances of the `Descendant/3` fluent (resp. `Child/3`), are asserted whenever an activity instance is added to the CWS, and persist in accordance with the successor-state axiom presented next.

```
Child(p,i,do(a,s)) ≡ (∃c,g,sc,f,j).a=add_activity(p,i,c,g,sc,f,j) ∧ ¬p=i ∨ Child(p,i,s)
Descendant(anc,i,do(a,s)) ≡ (∃p,c,g,sc,f,j).a=add_activity(p,i,c,g,sc,f,j) ∧ ¬p=i ∧
                                (p=anc ∨ Descendant(anc,p,s)) ∨ Descendant(anc,i,s)
```

An example of completion propagating upwards can be found between enactment states 4 and 5 in the enactment narrative described in Section 6.1, where, on instance `D` completing, completion is propagated to `S1` and `P`.

PropagateCompleteUp/4 will also initiate the execution of children of Seq instances, using ExecuteNextChild/4.

$$\texttt{ExecuteNextChild(i',i,st,s)} \equiv (\exists \texttt{p,i''}).\texttt{Child(p,i',s)} \wedge \texttt{PropagateRunningDownInc(i'',i,st,s)} \wedge \texttt{GType(p,s)=GId\_SEQ} \wedge \texttt{NextInitialChild(p,i'',s)}$$

The operation of this predicate is depicted in Figure 6.14. Assuming that we are interested in setting the next child of a Seq instance running, this predicate stipulates that an instance i is to be set running (on account of the completion of an instance i', which is the previous child of the Seq) iff i is the instance, or a descendant of the instance, next to be set Running in the Seq instance.

The predicate NextInitialChild/3 determines the next Initial child in the Seq, so that it may be executed.

$$\texttt{NextInitialChild(p,i,s)} \equiv \texttt{Child(p,i,s)} \wedge \neg(\exists \texttt{i'}).(\texttt{Child(p,i',s)} \wedge \texttt{i'<i} \wedge \texttt{State(i',s)=Initial})$$

The parameter p is the Seq instance, i'' in NextInitialChild(p,i'',s) is the next child of the Seq instance that should be set running. The predicate PropagateRunningDownInc/4 determines whether descendants of i'' should be moved to a Running state, or kept in an Initial state.

$$\texttt{PropagateRunningDownInc(i',i,st,s)} \equiv (\exists \texttt{i''}).\ \texttt{PropagateRunningDownInc(i'',i,st,s)} \wedge \texttt{Executes(i',i'',s)} \vee \texttt{i'=i} \wedge \texttt{st=Running}$$

PropagateRunningDownInc/4 uses the predicate Executes(p,i,s) for this purpose, where p being set running also causes i to be set running in situation s iff this predicate holds.

$$\texttt{Executes(p,i,do(a,s))} \equiv (\exists \texttt{p,c,g,sc,f,j}).\texttt{a=add\_activity(p,i,c,g,sc,f,j)} \wedge \neg\texttt{p=i} \wedge \texttt{f=EXEC} \vee \texttt{Executes(p,i,s)}$$

An example of execution being propagated to the next instance of a Seq can be found between enactment states 1 and 2 in the enactment narrative described in Section 6.1, where on instance A completing, execution is propagated down to B.

As prescribed in Figure 6.11, the atomic state that holds in the initial situation, S_0, comprises seven instances of the Activity/8 fluent. According to the axiom for *executable* situations, presented in Section 6.1, the only next executable situation is S_1=*do*(add_activity(0, 0, CId_P, GId_PAR, NONE, NONE, NONE), S_0), i.e. one where instance 0 has been added to the CWS. The effects of adding instance 0 to the CWS, in terms of fluent state in S_1 are: (i) according to State/4 and associated predicates, State(0, S_1)=Running now holds, (ii) the instance of Activity/8 for instance 0 ceases to hold (according to the ssa for Activity/8, presented next), and (iii) all other instances of Activity/8 persisting from S_0 according to the following successor-state axiom.

$$\texttt{Activity(p,i,c,g,sc,f,j,do(a,s))} \equiv \neg\texttt{a=add\_activity(p,i,c,g,sc,f,j)} \wedge \texttt{Activity(p,i,c,g,sc,f,j,s)}$$

The only next executable situation to S_1 is where instance 1, which is a Seq instance, has been added. The fluent state in this new situation, S_2, is characterised by (i) State(0, S_2)=Running persisting from S_1, (ii) State(1, S_2)=Running, Child(0, 1, S_2) and Descendant(0, 1, S_2) now holding, and (iii) instances of Activity/8 for activity instances 2–6 persisting from S_1.

Whenever there are activities to be added to the CWS, the only executable next situation will involve adding the next activity instance in the order determined by their instance numbers. In this example, this means that there is a chain of eight executable situations from S_0 (inclusively). Let's label the situation which results from adding all of the activity instances to the CWS, S_7. All models of the BAT must include the following atoms. This corresponds to state 1 in Section 6.2.1 (Figure 6.2).

```
State(0, S7) = Running
State(1, S7) = Running
State(2, S7) = Running
State(3, S7) = Running
State(4, S7) = Initial
State(5, S7) = Running
State(6, S7) = Initial
```

In executable situations that extend S_7, the only possible actions involve the completion of basic instances. In S_7, there are two possible next executable situations, pertaining to the completion of A (#3), and C (#5), i.e. *do*(comp_bas(3), S_7) and *do*(comp_bas(5), S_7). Completing on A causes B (#6) to be set running, according to the ssa for State/2, specifically the part which is captured by the ExecuteNextChild/4 predicate.

Having completed A (#3), we arrive at state 2 in Section 6.2.1 (Figure 6.3). All models of the BAT must include the following atoms, where S_8=*do*(comp_bas(3), S_7).

```
State(0, S8) = Running
State(1, S8) = Running
State(2, S8) = Running
State(3, S8) = Completed
State(4, S8) = Running
State(5, S8) = Running
State(6, S8) = Initial
```

Continuing to follow the evolution of the example, presented in Section 6.2.1, when we complete the basic instance B (#4), completion is propagated to the parent Seq instance S1 (#1), according to the PropagateCompleteUp/4 predicate. This means that all models of the BAT must include the following atoms, where S_9=*do*(comp_bas(4), S_8).

```
State(0, S9) = Running
State(1, S9) = Completed
State(2, S9) = Running
State(3, S9) = Completed
State(4, S9) = Completed
```

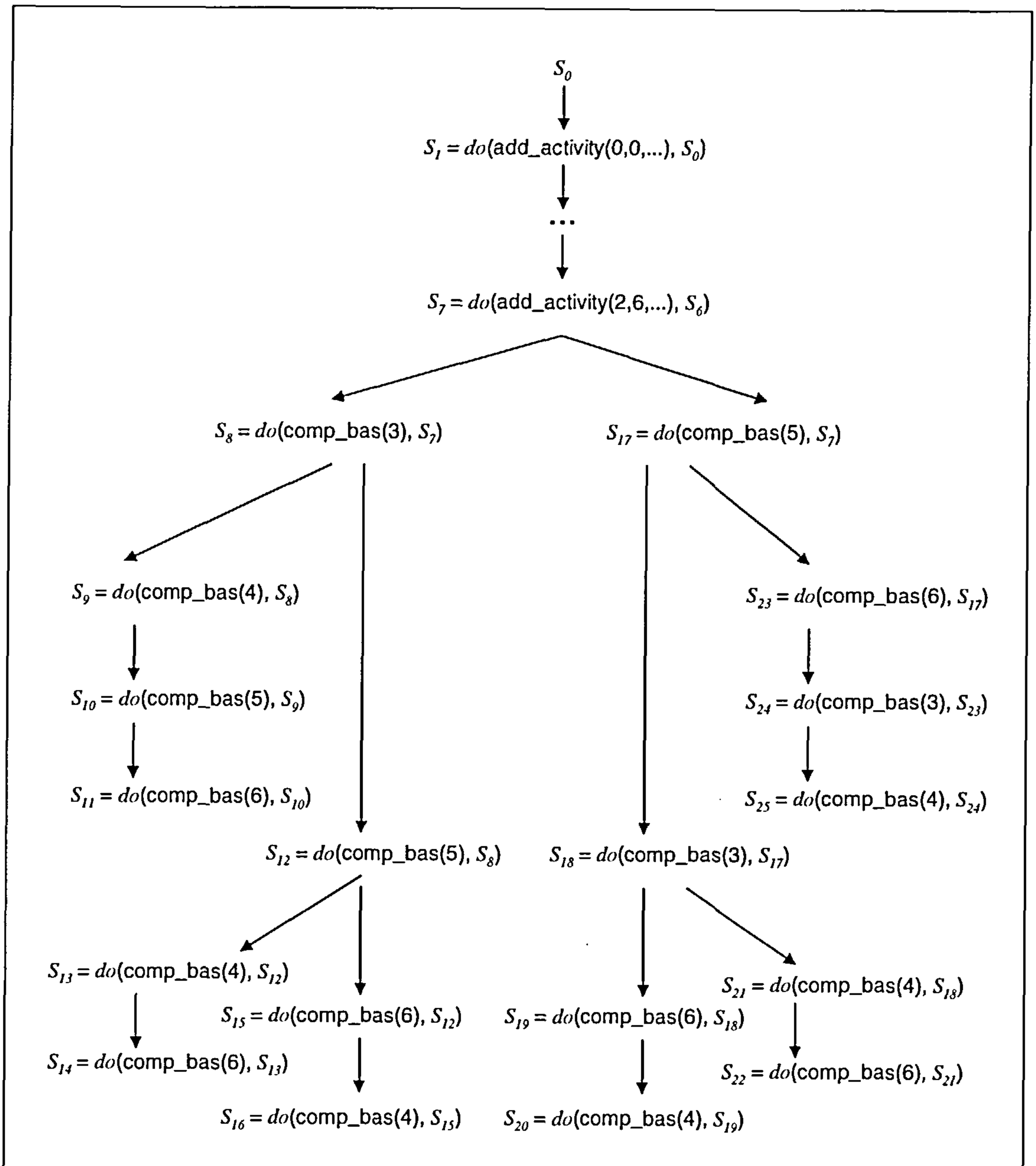


Figure 6.15: Executable Situation Tree for `Par(Seq(A,B),Seq(C,D))`

```
State(5, S9) = Running
State(6, S9) = Initial
```

When the other two basic instances complete, completion is propagated to S2 (#2) and the root instance P1 (#0). This means that all models of the BAT must include the following atoms, where S_{11}=*do*(comp_bas(6), *do*(comp_bas(5), S_9)). This is state 5 in Section 6.2.1 (i.e. Figure 6.6).

```
State(0, S11) = Completed
State(1, S11) = Completed
State(2, S11) = Completed
State(3, S11) = Completed
State(4, S11) = Completed
State(5, S11) = Completed
State(6, S11) = Completed
```

Other enactments are possible, as reflected in the narrative for the example presented in Section 6.2.1. Any model for the BAT presented in Figure 6.11 may be drawn as a tree of (executable) situations, as depicted in Figure 6.15.

Note that, in order to support the cancellation of basic instances, we make available another action, canc_bas/1. The ramifications on the BAT of supporting this action are straightforward: wherever we consider the completion of a basic instance, we must also now consider its cancellation. We need to include in the BAT an action precondition axiom for canc_bas/1, which has the same form as that for comp_bas/1. We also need to make some changes to StateChange/4, but we shall defer presentation of these until Section 6.2.3, for convenience.

Note that whenever we describe augmentations to the BAT involving changes to or additions of action pre-condition or successor-state axioms, these are to be classified as augmentations to the foundational axioms for workflow, whose initial set is presented in Figure 6.12.

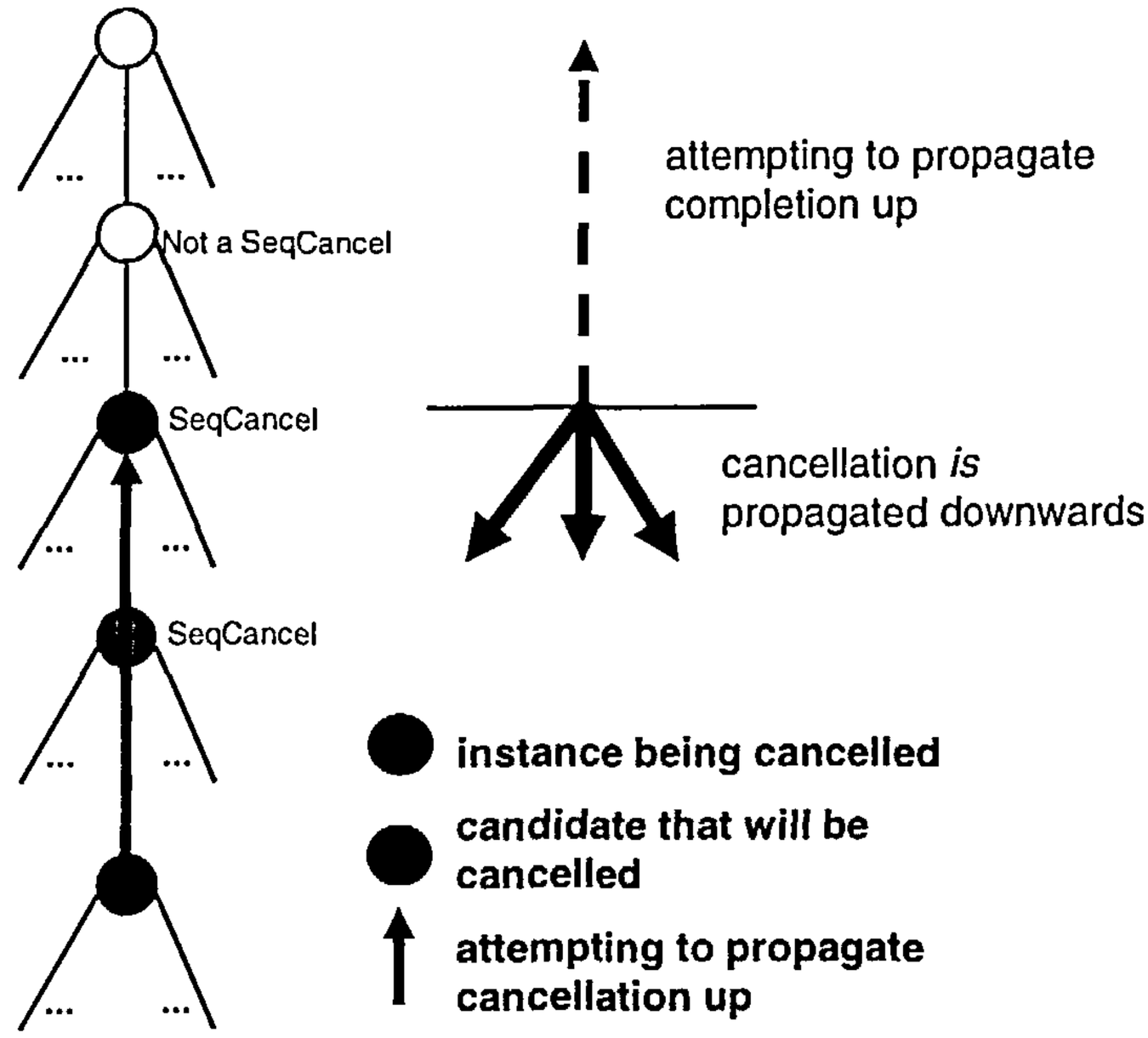
Finally, in the definition of AllDescSiblingsFinished/3, we need to modify the consequent of the implication to include a case for the state of an instance being cancelled, viz.

```
AllDescSiblingsFinished(i',i,s) ≡ Descendant(i,i',s) ∧
    (∀d).(¬d=i' ∧ Descendant(i,d,s) ∧ ¬Descendant(d,i',s) ∧ ¬Descendant(i',d,s) ⊃
     State(d,s)=Completed ∨ State(d,s)=Cancelled)
```

In the following subsections, we present the SitCalc characterisation of SeqCancel and choice types, as well as some information concerning the characterisation of Multi* types. We defer the presentation of synchronisation types (i.e. Go and Stop), UnorderedSeq, CancelActivity, Exit, merge types and multiple-instance types to Appendix B, to save space.

6.2.3 SeqCancel

There are two aspects to consider for a SeqCancel instance, namely, how execution of its child instances (after the initial child instance) is facilitated and how cancellation of the instance as a whole is facilitated, in the event that one of its child instances gets cancelled. For the first of




```
PropagateCancelUp(i',i,st,s) ≡
   ¬(∃p).Child(p,i',s) ∧ PropagateCancelDownInc(i',i,st,s) ∨
    (∃p).Child(p,i',s) ∧
      (GType(p,s)=GId_SEC ∧ PropagateCancelUp(p,i,st,s) ∨
      ¬GType(p,s)=GId_SEC ∧(PropagateCancelDownInc(i',i,st,s) ∨
                            PropagateCompleteUp(i',i,st,s))
```

Figure 6.16: Depiction of PropagateCancelUp/4

We propagate cancellation up through parent SeqCancel types, until a parent is reached which is not an instance of SeqCancel. Then, we propagate cancellation down through the tree rooted at the most senior of the SeqCancel types, and thereafter propagate completion up. Propagating cancellation downwards ensures that instances which do not lie on the path along which cancellation is propagated upwards also get cancelled. That is to say, the other children of a SeqCancel need to be cancelled, when cancellation is propagated through one of them.

these, we augment ExecuteNextChild/4 for the case where GType(p,s)=GId_SEC, which sits in disjunction with GType(p,s)=GId_SEQ.

For the other aspect, if a child instance of a SeqCancel is cancelled, then the parent SeqCancel instance must also be cancelled. In fact, cancellation should be propagated upwards, as long as each respective parent is a SeqCancel type, and, completion propagated thereafter. We achieve this behaviour through a modification to the definition of StateChange/4, viz.

```
StateChange(i,a,st,s) ≡
  (∃p,c,g,sc,f,j).a=add_activity(p,i,c,g,sc,f,j) ∧ SetRunning(p,i,f,st,s) ∨
  Completing(i,a,st) ∨
  (∃i').CompletingAction(i',a) ∧ PropagateCompleteUp(i',i,st,s) ∨
  (∃i').CancellingAction(i',a) ∧ PropagateCancelUp(i',i,st,s)
```

In the foregoing, the predicate CancellingAction(i,a) holds when the action a is a cancellation action on i, viz[3].

```
CancellingAction(i,a) ≡ a=canc_bas(i)
```

When a completing action occurs on an instance i', we try to propagate completion up, as before in Figure 6.11. When a cancellation action occurs, we first see whether we need to propagate cancellation up. The definition of PropagateCancelUp/4 is as follows. The operation of PropagateCancelUp/4 is depicted in Figure 6.16.

```
PropagateCancelUp(i',i,st,s) ≡
¬(∃p).Child(p,i',s) ∧ PropagateCancelDownInc(i',i,st,s) ∨
(∃p).Child(p,i',s) ∧
 (GType(p,s)=GId_SEC ∧ PropagateCancelUp(p,i,st,s) ∨
 ¬GType(p,s)=GId_SEC ∧ (PropagateCancelDownInc(i',i,st,s) ∨ PropagateCompleteUp(i',i,st,s))
```

Here, if there is no parent recorded (by Child/3) for i', then i' must be the root instance. In this case, we simply propagate cancellation down through the whole instance tree. On the other hand, if there is a parent p, recorded for i', then the following applies. If p is an instance of a SeqCancel type, we propagate cancellation up through p. If not, we propagate cancellation down through the sub-tree rooted at i', and propagate completion up from i'.

Cancellation is propagated downwards using PropagateCancelDownInc/4, which has the following definition:

```
PropagateCancelDownInc(i',i,st,s) ≡ st=Cancelled ∧ State(i,s)=Running ∧
(i'=i ∨ Descendant(i',i,s)).
```

6.2.4 Choice Types

The choice types, Choice, DefaultChoice and MultiChoice are accommodated as follows. The characterisation of these types is made simpler if we wrap (guard instance, continuation instance)

[3]In Appendix B, we augment the definitions of both CompletingAction and CancellingAction.

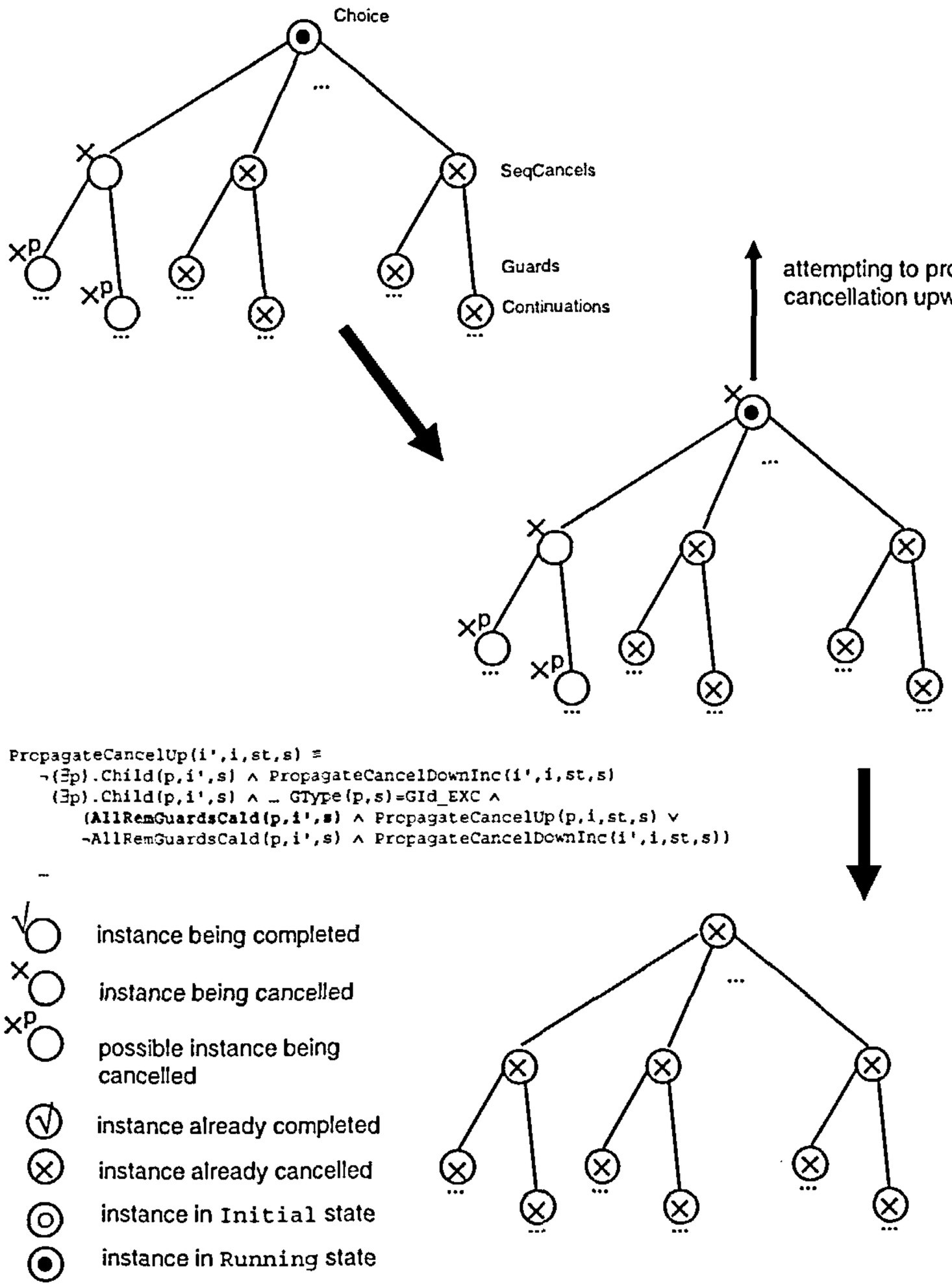
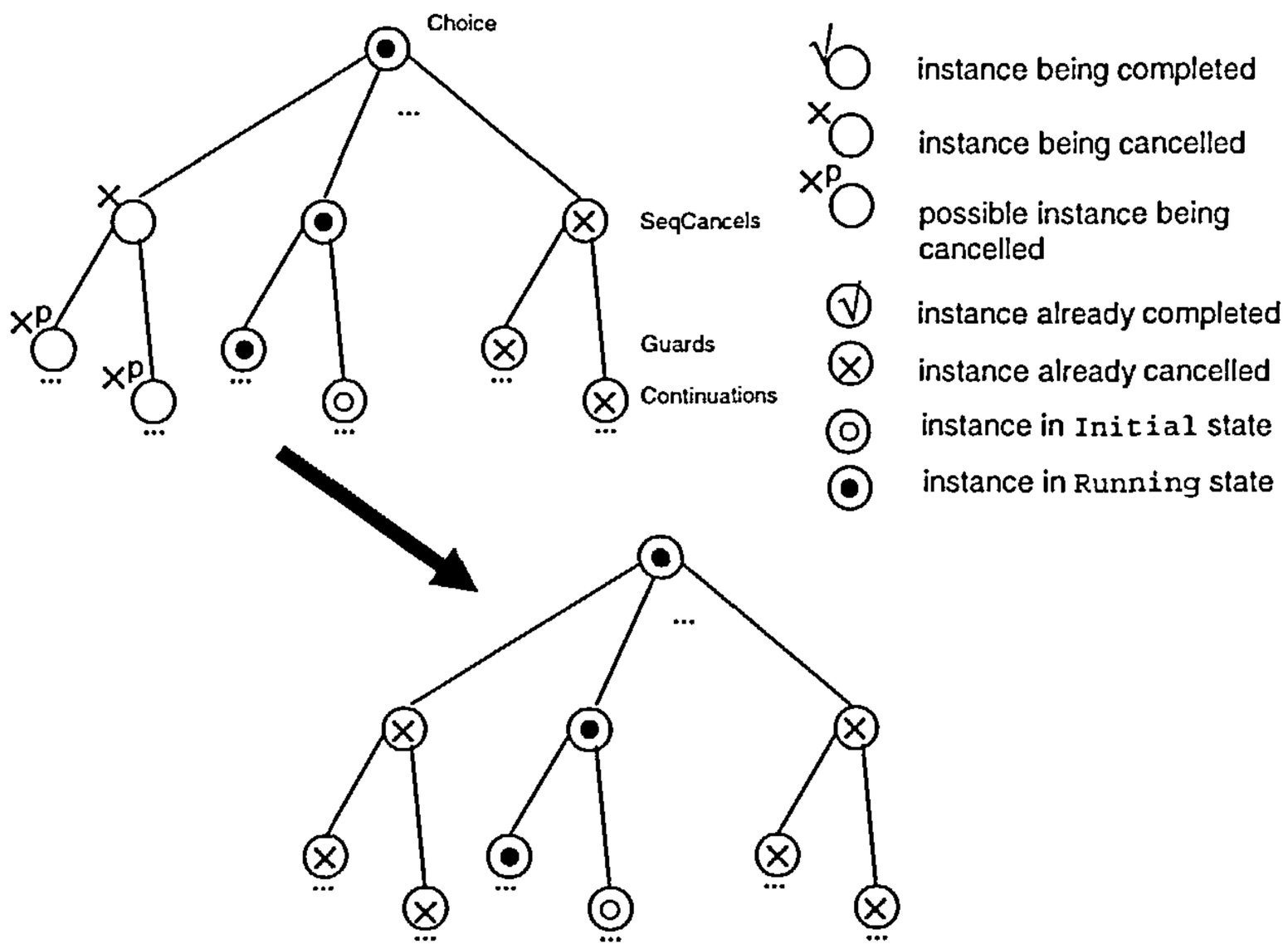


Figure 6.17: Depiction of Cancelling Guard/Continuation Instance in Choice Type I

In the top figure, a guard or continuation instance is being cancelled (x^p), which causes cancellation to be propagated to its SeqCancel parent container. In this case, all other child (guard, continuation) pairs have been cancelled.

As shown in the middle figure, cancellation is propagated upwards through the Choice type and attempted upwards thereafter. When propagation eventually stops, cancellation will be propagated back down through the Choice instance, as shown in the bottom figure, ensuring that instances which do not lie on the path along which cancellation was propagated upwards also get cancelled.




```
PropagateCancelUp(i',i,st,s) ≡
   ¬(∃p).Child(p,i',s) ∧ PropagateCancelDownInc(i',i,st,s) ∨
    (∃p).Child(p,i',s) ∧ ¬ GType(p,s)=GId_EXC ∧
       (AllRemGuardsCald(p,i',s) ∧ PropagateCancelUp(p,i,st,s) ∨
       ¬AllRemGuardsCald(p,i',s) ∧ PropagateCancelDownInc(i',i,st,s))
   ¬
```

Figure 6.18: Depiction of Cancelling Guard/Continuation Instance in `Choice` Type II

In the top figure, a guard or continuation instance is being cancelled (x^p), which causes cancellation to be propagated to its `SeqCancel` parent container. In this case, not all of the other child (guard, continuation) pairs have been cancelled.

As shown in the bottom figure, just cancellation is propagated through the originating (guard, continuation) pair.

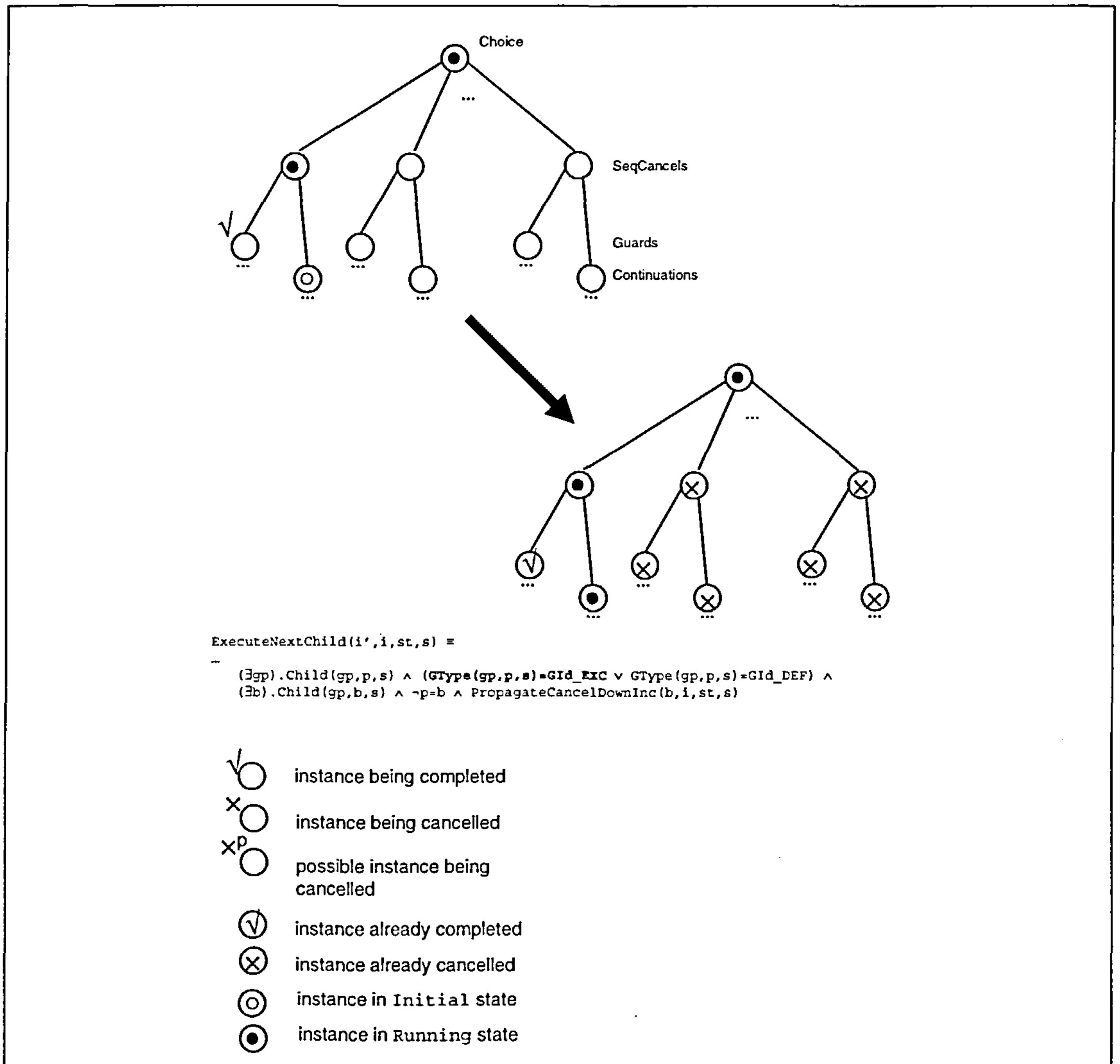


Figure 6.19: Depiction of Completing Guard in Choice Type

A running guard completing in a Choice type causes its pertaining continuation instance to be set running and cancellation of all other (guard, continuation) pairs to occur.

pairs in a SeqCancel container. Then, if a guard instance fails, its pertaining continuation instance is cancelled without the need for additional semantic machinery.

To facilitate the characterisation of Choice and DefaultChoice, we make the following augmentations to the definitions of PropagateCancelUp/4 and ExecuteNextChild/4 predicates. MultiChoice types are accommodated with no additional dispensation required.

The revised definition of PropagateCancelUp/4 is as follows.

```
PropagateCancelUp(i',i,st,s) ≡
  ¬(∃p).Child(p,i',s) ∧ PropagateCancelDownInc(i',i,st,s) ∨
  (∃p).Child(p,i',s) ∧
 (GType(p,s)=GId_SEC ∧ PropagateCancelUp(p,i,st,s) ∨
 GType(p,s)=GId_EXC ∧
   (AllRemGuardsCald(p,i',s) ∧ PropagateCancelUp(p,i,st,s) ∨
   ¬AllRemGuardsCald(p,i',s) ∧ PropagateCancelDownInc(i',i,st,s)) ∨
 GType(p,s)=GId_DEF ∧
  (Default(i',s) ∧ AllRemGuardsCald(p,i',s) ∧ PropagateCancelUp(p,i,st,s) ∨
  ¬Default(i',s) ∧ AllRemGuardsCald(p,i',s) ∧ (∃d).Default(d,s) ∧ Child(p,d,s) ∧
    (State(d,s)=Initial ∧ PropagateRunningDownInc(d,i,st,s) ∨
    ¬State(d,s)=Initial ∧ PropagateCancelUp(p,i,st,s)) ∨
  ¬AllRemGuardsCald(p,i',s) ∧ PropagateCancelDownInc(i',i,st,s)) ∨
¬GType(p,s)=GId_SEC ∧ ¬GType(p,s)=GId_EXC ∧ ¬GType(p,s)=GId_DEF ∧
  (PropagateCancelDownInc(i',i,st,s) ∨ PropagateCompleteUp(i',i,st,s))
```

For Choice (GId_EXC) types, with cancellation being propagated from a guard or continuation instance, if all remaining guards have been cancelled, we continue to propagate cancellation upwards. If it is not the case that all remaining guards have been cancelled, we just propagate cancellation through the (guard, continuation instance) sub-tree from which cancellation is being propagated, to ensure that the whole sub-tree is cancelled. The operation of Choice types, regarding cancellation of a guard or continuation instance, is depicted in Figures 6.17 and 6.18.

The definition of AllRemGuardsCald/3 is as follows. We simply check that all children of p, bar i and the default branch (applicable in the case of DefaultChoice), are cancelled.

```
AllRemGuardsCald(p,i,s) ≡ (∀b).Child(p,b,s) ∧ ¬i=b ∧ ¬Default(b,s) ⊃ State(b,s)=Cancelled
```

Instances of the fluent Default/2 are asserted to the BAT when default continuation instances of DefaultChoice types are added to the CWS, and persist thereafter. Specifically, whenever the parameter f in add_activity/7 is set to DEFAULT, the fluent instance Default(i,do(a,s)), where i is the identifier of the instance being added, will be asserted to the BAT.

Continuing with the description of PropagateCancelUp/4, for DefaultChoice (GId_DEF) types, if it is the default continuation instance (as determined by Default/2) from which cancellation is being propagated, and all guard instances have been cancelled, we continue to propagate cancellation upwards.

However, if cancellation is being propagated from a guard or (other) continuation instance, and all remaining guards have been cancelled, the following applies. If the default instance is

in the Initial state, we set it running; but, otherwise, we propagate cancellation through the DefaultChoice instance (as a whole) as the default cannot be executed.

Alternatively, if some branches of the DefaultChoice instance are yet to be cancelled, we just propagate cancellation through the whole (guard, continuation instance) sub-tree from which cancellation is being propagated.

We do not show the operation of DefaultChoice graphically, in order to save space.

For all other Liesbet types, (for now) we propagate cancellation through the sub-tree from which cancellation is being propagated, and then propagate completion upwards.

One aspect of propagating completion upwards is to attempt to advance an instance (other than completing it) through ExecuteNextChild/4 which has had completion/cancellation propagated to one of its children. We are able to advance an instance, in this event, if it has any of its other children left in a not finished state. Propagation of completion stops when we are able to advance an instance in this way. However, if the only action we can take is to complete an instance, on account of all of its children being finished, we continue to propagate completion upwards.

We need to modify ExecuteNextChild/4 for the occasion when a guard instance in a Choice or DefaultChoice is completed. When this happens, we need to cancel any guard instances which are still running. The augmented definition of ExecuteNextChild/4 is as follows. The operation of Choice types, regarding completion of a guard instance, is depicted in Figure 6.19.

```
ExecuteNextChild(i',i,st,s) ≡ (∃p,i'').Child(p,i',s) ∧
  (PropagateRunningDownInc(i'',i,st,s) ∧
   (GType(p,s)=GId_SEQ ∨ GType(p,s)=GId_SEC) ∧ NextInitialChild(p,i'',s)) ∨
   (∃gp).Child(gp,p,s) ∧ (GType(gp,s)=GId_EXC ∨ GType(gp,s)=GId_DEF) ∧
   (∃b).Child(gp,b,s) ∧ ¬p=b ∧ PropagateCancelDownInc(b,i,st,s))
```

6.2.5 Dynamic Adding of Activities by Multi/MultiSeq types

For Multi/MultiSeq types, it is interesting to note how we effect the dynamic addition of new instances of ExecAct types (see Section 3.1.15) to the CWS. Note that, as was done for choice types, the translator wraps (join condition, execution activity) pairs of all Multi* types in a containing SeqCancel type, which makes for a simpler characterisation.

We need templates for (join condition, execution activity) pairs, which can be instantiated whenever it is appropriate to add an instance of one of these pairs to the CWS. The templates are specified as instances of the ActivityTemplate/9 fluent. Instances of this fluent are asserted to the initial BAT (i.e. for situation S_0) and persist by inertia. There is a successor state axiom (ssa) for ActivityTemplate/9 which effects the inertia.

The statically-extant instances of a Liesbet model are represented in the initial BAT as instances of the Activity/8 fluent. Instances of the Activity/8 fluent may also be dynamically asserted to the BAT whenever the join condition of a Multi/MultiSeq instance completes. To reflect this, there is an augmentation to the definition of the ssa for Activity/8, as follows.

```
Activity(p,i,c,g,sc,f,j,do(a,s)) ≡
  (∃p',gp).CompletingMultiJoin(a,p',gp,s) ∧ (∃c').CType(p',s)=c' ∧
  (∃p'',i',j').ActivityTemplate(c',p'',i',c,g,sc,f,j',s) ∧
```

```
(∃n).GetNextInstNo(n,s) ∧ AssignActIds(i,i',j,j',p,p'',gp,n) ∨
¬a=add_activity(p,i,c,g,sc,f,j) ∧ Activity(p,i,c,g,sc,f,j,s)
```

According to this axiom, whenever a join condition in a `Multi/MultiSeq` is completing (`CompletingMultiJoin/4`), the following occurs. We create a fresh copy of the (join condition, execution activity) pair of the `Multi/MultiSeq`, contained within a `SeqCancel` instance having type `c'`, plus descendants, by creating an instance of `Activity/8` for each instance of `ActivityTemplate/9` whose first parameter is `c'` that exists within the BAT (for the current situation). For each such instance, the identifiers `p''` (parent), `i'` (instance) and `j'` are relative identifiers. Their absolute values are determined according to `AssignActIds/8`.

The definition of `CompletingMultiJoin/4` is as follows. It holds when the action `a` serves to complete `gu`, which is a join condition, running in `s`, in a `Multi/MultiSeq` instance, `gp`; where `p` is the `SeqCancel` container of the join condition instance.

```
CompletingMultiJoin(a,p,gp,s) ≡ (∃gu,c).Child(p,gu,s) ∧ Child(gp,p,s) ∧
  (GType(gp,s)=GId_MUL ∨ GType(p,s)=GId_MUS) ∧ Guard(gu,c,s) ∧ State(gu,s)=Running ∧
   (∃st').StateChange(gu,a,st',s) ∧ st'=Completed
```

The functional fluent `CType/2` records the customised type identifier of an activity instance, which is the parameter `c` in the action `add_activity/7`. The predicate `GetNextInstNo/2` gets the next "free" activity instance identification number, viz.

```
GetNextInstsNo(n,s) ≡ (∃c',n').CType(n',s)=c' ∧ n=n'+1 ∧
  ¬(∃c'',n'').(CType(n'',s)=c'' ∧ n''>n')
```

The predicate `AssignActIds/8` converts relative instance numbers into absolute ones.

```
AssignActIds(i,i',j,j',p,p',gp,n) ≡ i=i'+n ∧ j=j'+n ∧
  (p'=0 ∧ p=gp ∨ ¬p'=0 ∧ p=p'+n)
```

The `SitCalc`-based characterisation of `Liesbet` for the remaining constructs is presented in Appendix Section B.1.

6.3 Translation of `Liesbet` Models to `SitCalc`-based Characterisation

As we did for the CCS-based semantic characterisations of `Liesbet`, we provide a definition of a conceived translation function, $\mathcal{M}_{SitCalc}[\![-]\!]$, for `Liesbet` models, in order to fix the definition of the `SitCalc`-based characterisation of `Liesbet`.

The result of translating a `Liesbet` model using $\mathcal{M}_{SitCalc}[\![-]\!]$ is to assert a set of ground atoms to the BAT, which pertain to instances of fluents that hold in the initial situation, S_0. There are some other side-effects which are described in Appendix Section B.2.

We assume that a `Liesbet` model has been pre-processed in order to replace the use of defined types by *in situ* definitions, see Section 3.1. We also assume that, in doing so, the name specified in the definition of a customised activity type is copied to the `ctype` qualifier that exists when the

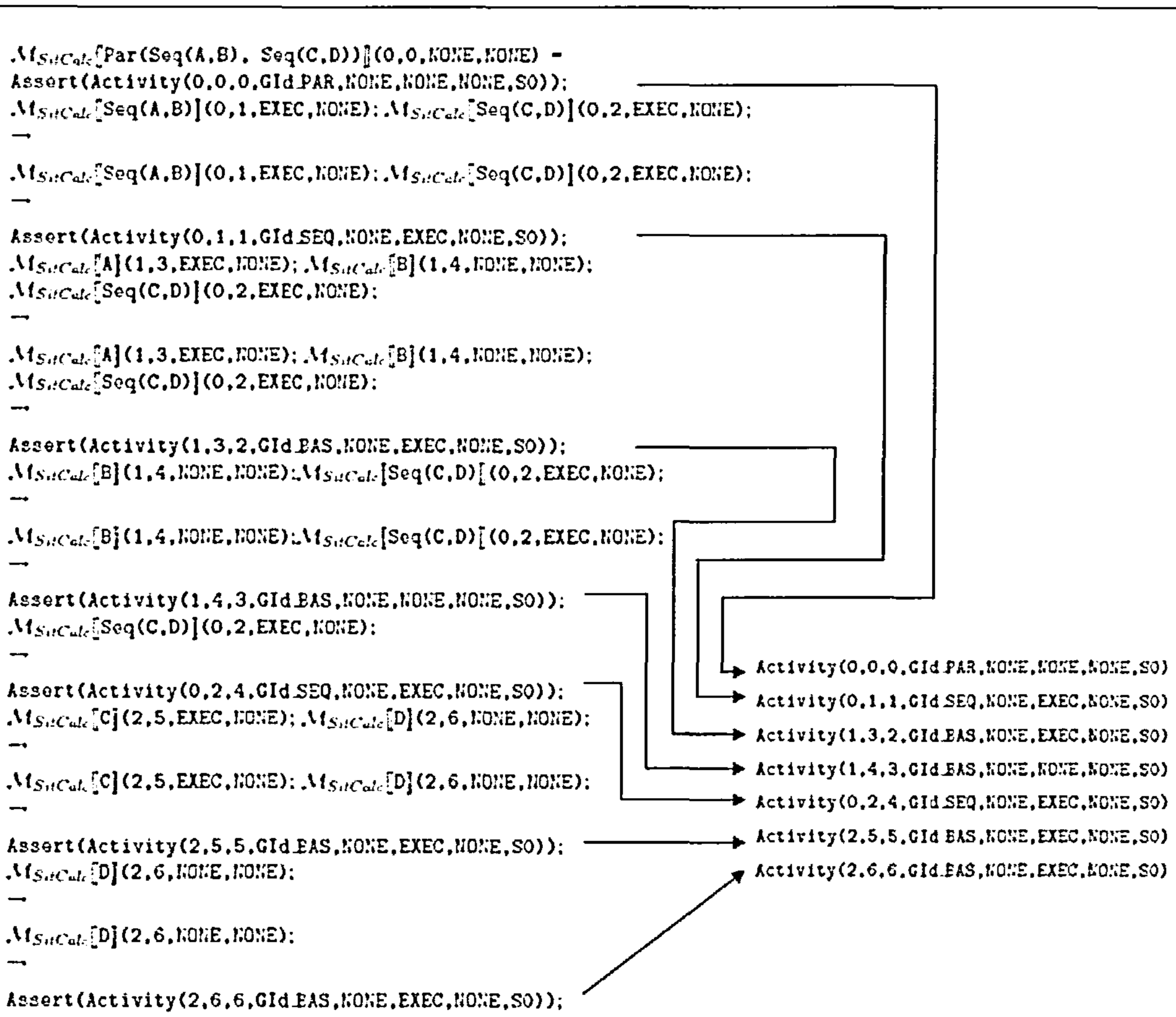


Illustration of the operation of $\mathcal{M}_{SitCalc}[\![-]\!]$ on the Liesbet model that we have used for illustrative purposes throughout this thesis, viz. `Par(Seq(A,B), Seq(C,D))`. $\mathcal{M}_{SitCalc}[\![-]\!]$ starts by processing the `Par` and works inwards in a depth-first, preorder manner. As $\mathcal{M}_{SitCalc}[\![-]\!]$ processes the model, it will assert ground atoms to the BAT, which is shown on the right of the figure.

Figure 6.20: Operation of $\mathcal{M}_{SitCalc}[\![-]\!]$ on `Par(Seq(A,B), Seq(C,D))`

type is used *in situ*.

In the following, we prescribe how the use of activity types within a (pre-processed) `Liesbet` model is to be mapped to instances of the `Activity/8` fluent to be asserted for the initial situation, S_0. Note that `GType(g)` indicates the generic type name for a macro identifier `g`, so `GType(GId_BAS)=Act`, `GType(GId_PAR)=Par` etc. So `GType(GId_PAR)(A,B,C)` is `Par(A,B,C)` where `A,B,C` are the child types of the `Par` customised type.

The body of a definition is a number of steps that must be carried out as part of the translation process for the particular activity type. The instruction `Assert(S)` asserts `S` to the BAT. When applying $\mathcal{M}_{SitCalc}[\![-]\!]$, the parent instance `p`, an id for the instance being translated `i`, the multi-purpose flag `f`, and possible join condition instance `j` are passed as arguments. The translation process is initiated by passing `(0, 0, NONE, NONE)` as the initial arguments to $\mathcal{M}_{SitCalc}[\![-]\!]$.

The global function `genTypeId(c)`, given a type name will generate a (natural number) iden-

tifier for the type – if it has already been used elsewhere, `genTypeId/1` returns the same id as previously. Otherwise it allocates a new (unused) id for the type name. The argument-free version, `genTypeId/0`, will simply generate a new type identifier, starting from `1`. `genInstId/0` returns a new (natural number) instance id, subsequent to the last.

In this section, we present the definition of $\mathcal{M}_{SitCalc}[\![-]\!]$ for `Act`, `Seq`, `SeqCancel`, `UnorderedSeq` and `Par` types, making a distinction between isolated and non-isolated scopes. The definition of $\mathcal{M}_{SitCalc}[\![-]\!]$ for the remaining `Liesbet` types is left to Appendix Section B.2.

The multi-purpose flag `f` for a child instance is set, according to $fval/2$, to `EXEC` if the parent is a `Par` instance, or if the parent is a `Seq` or `SeqCancel` instance and the child is the parent's first. Otherwise, the flag is set to `NONE`. The instance id, `i`, for an instance (which is passed in the arguments to $\mathcal{M}_{SitCalc}[\![-]\!]$) is generated when translating the parent instance.

- Not isolated

  $\mathcal{M}_{SitCalc}$[[GType(g)(Ch1,...,Chn)(ctype(ctype))]](p,i,f,j) =

  Assert(Activity(p,i,c,g,NONE,f,j,S0))

  where c=genTypeId(ctype)

  $\mathcal{M}_{SitCalc}$[[Ch1]](i,i1,$fval$(1,g),NONE); ...; $\mathcal{M}_{SitCalc}$[[Chn]](i,in,$fval$(n,g),NONE);

  where i1=genInstId(), ..., in=genInstId(), and

  $fval$(m,g) = EXEC if g=GId_PAR or (g=GId_SEQ or g=GId_SEC) and m=1;

  = NONE, otherwise.

- Isolated

  $\mathcal{M}_{SitCalc}$[[Isolated(GType(g)(Ch1,...,Chn)(ctype(ctype)))]](p,i,f,j) =

  Assert(Activity(p,i,c,g,ISCOPE,f,j,S0))

  $\mathcal{M}_{SitCalc}$[[Ch1]](i,i1,$fval$(1,g),NONE); ...; $\mathcal{M}_{SitCalc}$[[Chn]](i,in,$fval$(n,g),NONE);

The operation of $\mathcal{M}_{SitCalc}[\![-]\!]$ is illustrated in Figure 6.20. The definition of $\mathcal{M}_{SitCalc}[\![-]\!]$ for the remaining constructs of `Liesbet` is presented in Appendix Section B.2.

6.4 Completion Result

In Section 5.2, we presented a result concerning the completion of arbitrarily constructed `Liesbet1` models, characterised with CCS-based semantics. We proceed to do the same here for the `SitCalc`-based characterisation[4].

Result

A `Liesbet` model (constructed according to the syntactical constraints defined by the meta-model) is guaranteed to complete in a finite number of steps (that is, all instances report completion, or cancellation), with a finite situation tree, under the assumptions that:

[4]Note that the following result statement is different to that presented in the Section 5.2 for the CCS-based characterisation, because we include synchronisation types in the `SitCalc`-based characterisation of `Liesbet` presented here.

- All synchronisation activity instances (i.e. `Go` and `Stop` instances) eventually complete, or cancel.
- All join conditions used in `Multi/MultiSeq` types eventually fail (go to `Cancelled`).

These assumptions are necessary because it is possible for queries within synchronisation types never to be satisfied, possibly meaning that the pertaining synchronisation would never report completion or cancellation. A forever blocked synchronisation type instance is the only source of *model-deadlock* in `Liesbet` models. It is also possible that a join condition in a `Multi/MultiSeq` will always be satisfied. This represents the only source of *model livelock* in `Liesbet` models.

Proof. We restrict ourselves to the *primitive* `Liesbet` constructs contained within the set $\texttt{Liesbet}_{prim}$, defined in Section 3.4. As all other `Liesbet` constructs, as we shall show in the result subsequent to this one (see Section 6.5), may be considered to be abbreviations of `Liesbet` specifications using just primitive constructs, the following completion result for $\texttt{Liesbet}_{prim}$ will necessarily hold for $\texttt{Liesbet}_{abbrev}$.

We work inductively from the base case of a $\texttt{Liesbet}_{prim}$ model consisting of one activity instance. By definition, such an instance must be of a childless generic type, viz. a basic activity, `Go`, `Stop` or `CancelActivity`.

- *Base cases*:

  In the case of a **basic activity instance**, the BAT that the translator outputs would be the following, in respect of atoms.

```
Activity(0, 0, CId_A, GId_BAS, NONE, NONE, NONE, S0)
```

According to the axiom for executable situations (see Section 6.1), and the action precondition axioms (described in Section 6.2.2), the only executable next situation would be one where we add activity `0` to the CWS, using `add_activity/7`. There is, thus, a single following situation, which we shall call S_1, viz: S_1=*do*(`add_activity(0,0,CId_A,GId_BAS,NONE,NONE,NONE)`,S_0), which results from the occurrence of the action instance named in the preceding *do* term. The fluent state (i.e. instances of fluents which hold) in S_1 will be:

S_1:

```
GType(0, S1) = GId_BAS
CType(0, S1) = CId_A
State(0, S1) = Running
```

There are two executable situations that may follow S_1: one resulting from the completion of the single activity instance (`0`), by virtue of `comp_bas/1`, and the other resulting from cancelling the instance, by virtue of `canc_bas/1`.

Having completed (resp. cancelled) the instance, the fluent state will be the following, where S_2 (resp. S_3) is the situation *do*(`comp_bas(0)`,S_1) (resp. *do*(`canc_bas(0)`,S_1)).

S_2 (resp. S_3):

```
GType(0, S2) = GId_BAS      (resp.  GType(0, S3) = GId_BAS)
CType(0, S2) = CId_A        (resp.  CType(0, S3) = CId_A)
State(0, S2) = Completed    (resp.  State(0, S3) = Cancelled)
```

For a model consisting of a single basic instance, a model of the corresponding `SitCalc` basic action theory will be a tree of executable situations of size four: $S_0 \rightarrow S_1 \rightarrow \{S_2, S_3\}$, where, S_1 necessarily follows from S_0, and either S_2 or S_3 may follow S_1. For both enactment paths contained within the situation tree, the model completes, i.e. the single instance finishes in a completed or cancelled state. For the other bases case types, the argument (for completion in the context of single-instance `Liesbet` models) is identical, except that the instance will get completed/cancelled using `complete,cancel/1`.

- *Induction step*:

  We now proceed by taking each child-bearing generic activity type from $\texttt{Liesbet}_{prim}$ in turn and show that their introduction into a model preserves the completion result that we are seeking to prove.

  Note that:

  1) We shall show that the presence of a child-bearing structured instance in the model serves only to *eventually* propagate execution down to its children, and once all of its children have reached a finished state to propagate completion back up. These apart, it will have no other effect on workflow state. Accordingly, no child-bearing instance is ever a source of deadlock, or livelock (under the assumptions presented in the result premise, given above).

     Thus, each enactment step of a child-bearing instance, and each enactment step of a child-less instance (see the base cases), moves the instance (and model) closer to completion.
  2) There can only be a finite number of instances that are ever created. This is, fundamentally, a consequence of the second assumption, regarding satisfaction of `Multi/MultiSeq` join conditions, presented in the result premise.

  Because of 1) and 2), the completion of a model must take place in a finite number of steps.

  Moreover, for any one instance, there is only a finite number of actions that can involve the instance, in any given situation. All of the action schemas except `complete/3` and `cancel/3` have the property that only one instance of each will be executable for any particular activity instance. For `complete/3` and `cancel/3`, as any model has only a finite number of instances, there is only a finite number of possible ways of populating the `di` and `l` parameters of these actions. Further, most instances of actions will have deterministic effects – the only possibility for non-deterministic effects admitted in the `SitCalc` characterisation of `Liesbet` is for `UnorderedSeq`, where the next child to be executed is non-deterministically chosen (see Appendix Section B.1.3). But even in this case, there is only a finite number of "next possibilities", i.e. the number of yet-to-be-run children of the `UnorderedSeq` instance. Given the finite number of instances in a model, branching (from any situation) must itself be finite. Thus, given the finite length of paths and finite branching, there can only be a finite number of enactment paths and situations – the situation tree (i.e. state space) must be finite.

By the induction hypothesis (as part of what we are demonstrating here), each instance must eventually be set running. Once running:

- A `Par` instance will propagate execution to all of its children (as determined by `SetRunning/5` and `Executes/3`, used by `PropagateRunningDownInc/4`, see Figure 6.11, which query the value of the multi-purpose flag `f` being set to `EXEC`); and, once all of its child instances have finished, will itself complete and propagate completion upwards (`PropagateCompleteUp/4`).
- A `SeqCancel` instance will propagate execution to its first child instance (according to flag `f` being set to `EXEC`, for this instance). Once the given child instance has completed, it will execute its second child instance (`ExecuteNextChild/4`), and so on. Once all of its child instances have finished, it will itself complete and propagate completion upwards. If a child instance is cancelled, the `SeqCancel` instance will get cancelled and cancellation (at first, and completion thereafter) will be propagated upwards (`StateChange/4`, 6.2.3).
- A `Multi` instance will propagate execution to its first (and only) child instance (according to flag `f` being set to `EXEC`). The child will be a `SeqCancel` instance (see Section 6.3) containing a join condition and execution activity instance. The join condition is also set running as part of the propagation (according to `f=EXEC`). If the join condition completes successfully, the execution activity instance (being contained within a `SeqCancel` with the join condition) is executed. At the same time a fresh join condition, execution activity pair (plus all descendants) is added to the CWS, contained within a new `SeqCancel` instance (`Activity/8` ssa, B.1.6), whereon the new join condition is set running (according to flag `f` being set to `EXEC`). Many more join condition, execution activity pairs may be created in this way. Eventually (see second assumption of result premise), a join condition instance will get cancelled. This will cause its containing `SeqCancel` instance to get cancelled. When all execution activity instances have finished, completion of the `Multi` instance occurs and completion is propagated upwards.

This argument assumes that the propagation behaviour prescribed therein is actually realised by the foundational axioms, presented in Figure 6.12. In considering the possible models of these axioms, it is self-evident that their behaviour is precisely that prescribed.

Note also that an instance may be cancelled, by virtue of a `CancelActivity` instance, or because of dead-path elimination (see Section 3.1). In these cases, the instance hierarchy rooted at the given instance is cancelled (`PropagateCancelDownInc/4`), and thus completion of this sub-tree trivially occurs. On the sub-tree being cancelled, cancellation (at first, and completion thereafter) is propagated upwards.

□

6.5 Model Equivalence Result

In this section, we present a result concerning the equivalence of `Liesbet` models expressed using the `SitCalc`-based characterisation presented in the foregoing part of this chapter and (the same) models expressed using the described `SitCalc`-based characterisation for the set of primitive constructs, $\texttt{Liesbet}_{prim}$, together with the use of the constructs from $\texttt{Liesbet}_{abbrev}$ replaced by their definitions (in terms of primitive constructs), as presented in Section 3.4.

Note that we label the semantics presented in the foregoing part of this chapter, the *elaborated characterisation*; and the semantics which use the abbreviations for constructs in $\texttt{Liesbet}_{abbrev}$ as part of their definition, the *abbreviated characterisation.*

To proceed, we need to make some modifications to the `SitCalc` characterisation of `Liesbet` presented in this chapter. The single driver for all of these modifications is to remove certain aspects of the atomicity in propagating side-effects that are prescribed by the intended semantics for `Liesbet`, presented in Section 3.2.

An example of the necessary modifications for `Seq` is as follows. We need to remove the facility for propagative execution of the next child in `Seq`. Instead, we incorporate an explicit `execute/1` action, into our `SitCalc` semantics for `Liesbet`, that operates at the same level of priority as `complete,cancel/1` actions, according to an additional action precondition axiom (omitted here) for the action.

Doing this means that other model actions (pertaining to completion and cancellation actions) may get interleaved between propagating completion upwards (as a result of the finishing of an instance) and executing the/some next child instance in the `Seq` instance (once propagation reaches such an instance). In the regular, i.e. elaborated, `SitCalc` characterisation for `Liesbet` presented in the foregoing part of this chapter, this interleaving cannot occur. Consequently, keeping the original behaviour would mean that the abbreviated characterisation of `Seq` would not preserve model equivalence with respect to its elaborated counterpart.

For instance, in the original characterisation of `Seq`, its first instance finishing would necessarily cause its second instance to be executed as an implicit side-effect (see `ExecuteNextChild/4`, in Section 6.2.2). In this case, there is no opportunity for activities (namely, synchronisation type instances) in the rest of the model to see the momentary intermediate state between the first instance finishing and the second instance executing. That is, as far as the rest of the model is concerned, the two state changes are atomic. However, when this behaviour is modified, in the way proposed, any number of intermediate model actions may occur, making the intermediate state "visible" to the rest of the model. This latter state-of-affairs corresponds to the behaviour of the abbreviated characterisation for `Seq`.

The definition of `ExecuteNextChild/4` is cut down to the following.

```
ExecuteNextChild(i',i,st,s) = (∃p,i'').Child(p,i',s) ∧ GType(p,s)=GId_SEC ∧
   NextInitialChild(p,i'',s) ∧ PropagateRunningDownInc(i'',i,st,s)
```

We also augment `StateChange/4` to account for the new `execute/1` action, viz.

```
StateChange(i,a,st,s) ≡
  ...
  (∃i').a=execute(i') ∧ PropagateRunningDownInc(i',i,st,s)
  ...
```

For brevity, we omit a presentation of the necessary modifications for the remaining constructs in $\texttt{Liesbet}_{abbrev}$. We consider the description for `Seq` to be sufficient, for the purposes of this thesis, as this is the only construct for which we give our proof of model equivalence. We restrict our proof to this one construct for reasons of space. Information regarding the other necessary modifications for the remaining constructs, and proof of model equivalence with respect to these,

may be obtained from the author on request.

Result

The two sets of `SitCalc`-based semantics for `Liesbet`, viz. those

- presented in the foregoing part of this chapter, and those
- derived using a combination of the `SitCalc`-based characterisation (described in the foregoing) for $\texttt{Liesbet}_{prim}$, and the set of abbreviations for constructs in $\texttt{Liesbet}_{abbrev}$, presented in Section 3.4,

lead to characterisations of models which are *model equivalent*.

For model equivalence, a similar definition to that presented in Section 5.1.5 is proposed, whereby two `Liesbet` models are model equivalent iff

- Any progression of one of the models, through the advancement of *structured instances*, where the model is progressed to a state where no structured instance can be further advanced, is *matched* by some similar progression in the other model (i.e. structured instances are advanced as far as possible), such that the set of *basic instances* being offered for completion, or cancellation, is identical; *and* the two models resulting from a completion on the same basic instance in both (progressed) models, or a cancellation, are themselves model equivalent.
- When any progression of one of the models, through the advancement of structured instances leads to a state where no further progression can be made at all, the model reports successful completion; *and* this is matched by some, similar progression in the other model.

In the following proof, we need to show is that the *external* impact that an instance of a particular construct has, when present in a `Liesbet` model, is the same between characterisations. Notably, the characterisation of *basic activity instances* is the same between characterisations, as basic activities belong to $\texttt{Liesbet}_{prim}$. Given (as will be shown) that the external impact of all other activity types will be the same between characterisations, it will necessarily be the case that the behaviour, as the model evolves, of basic instances in the model, *which is the key to determining the model equivalence result*, will be the same. Moreover, completion (as an externally visible artefact) will necessarily be identically reported.

It is appropriate to elaborate what we mean by external impact; and, to do so, we define the following notions: *action trees*, *action classes*, *action windows* and *action sets*. An action tree, for a construct, is the tree of action occurrences that: (a) are externally generated, and may affect the evolution of the construct, (b) are internally generated, and may affect the evolution of the rest of the model, and (c) all other internally generated actions. These are the *action classes* that may concern a construct. In assessing external impact, we are interested solely in classes (a) and (b).

Notably, there is an orthogonal classification for actions, viz. explicit and implicit actions. Some actions in an action tree may refer to implicit side-effects of other actions, and do not actually occur explicitly (i.e. as actions that move the theory from one situation to another). For example, there may occur an action to complete an instance. This is an explicit action occurrence, but

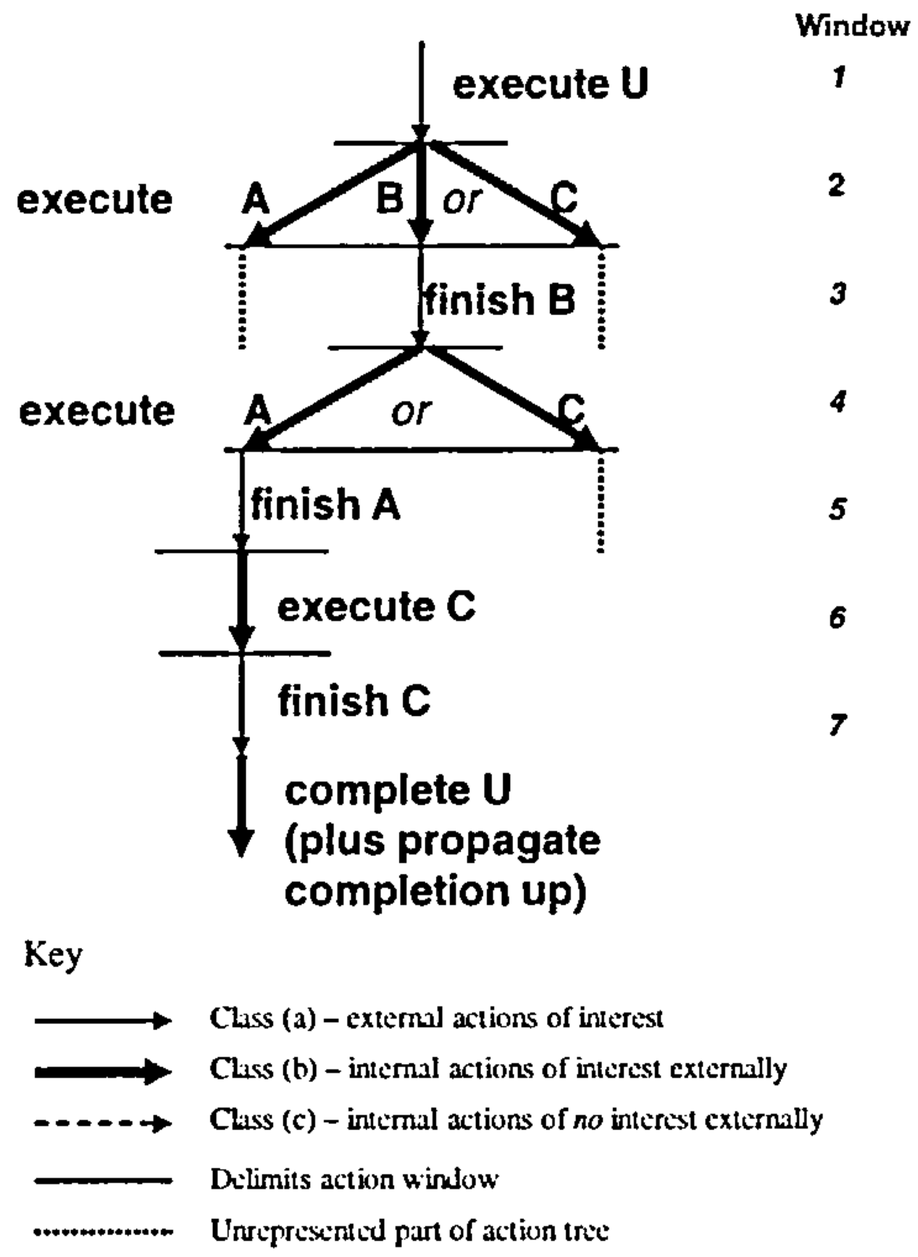


Figure 6.21: Action Tree for Elaborated Characterisation of `UnorderedSeq`.

---

may have a number of implicit side-effects (which may be conceptualised as implicit actions), e.g., propagating completion upwards.

Actions in action trees are grouped into *action windows*. *Action sets* map onto action windows, and there may be many sets that correspond to a single window. The action sets defined for a window represent a disjunction of action occurrences that may occur within the action window. The first action in an action set occurs as a result of the explicit execution of a domain theory action, which may belong to class (a) or class (b). The remaining actions within an action set must be implicit or class (c) actions. Two action sets *match* iff they have identical class (a) and class (b) actions, irrespective of their explicit or implicit nature.

In Figure 6.21, we show the action tree for the elaborated version of `UnorderedSeq`. Each action window is shown by a horizontal dividing line, and the evolution of the type occurs from top-to-bottom. Class (a) actions – external actions of interest – are shown by thin solid arrows. Class (b) actions – internal actions of external interest – are shown by thick solid arrows. And class (c) actions – internal actions of no external interest – are shown by dotted arrows. (There are no class (c) actions for this construct.)

Here, the action windows consist of a single action, apart from the seventh window which consists of two. The second and fourth windows of the tree contain three and two action sets, respectively. For these windows, just one of the action sets may be carried out. In the seventh window of the tree, an *explicit* (external) action occurs, which pertains to the last child finishing. As a side-effect, an *implicit* (internally-generated) occurrence to complete the `UnorderedSeq` instance occurs. This has external visibility, and its occurrence may cause further completions to be propagated up the `Liesbet` activity tree.

Two action trees are defined to have the same external impact iff an action set defined by the

immediate action window of one action tree matches a set present in the immediate action window of the other tree, *and* the pair of action trees that remain after carrying out the actions in the corresponding action sets also have the same external impact.

Proof.

The (elaborated and abbreviated) characterisations of constructs from $\texttt{Liesbet}_{prim}$ are the same, by definition. For constructs in $\texttt{Liesbet}_{abbrev}$, we prove the result for just one, Seq. Proofs for the remaining constructs follow in a similar way. The definition of Seq as an abbreviation is shown below (from Section 3.4).

```
S = Seq(A,B,C)
--------------

S  = Par(A,B',C')
B' = SeqCancel(Go(Finished_act(A in S)), B)
C' = SeqCancel(Go(Finished_act(B in S)), C)
```

We can prove equivalence for this three-argument Seq without loss of generality. That is, the proof for Seq types of different arity trivially follow. Moreover, the nature of Seqs children is immaterial. We argue the case for identical external impact on the basis of the operational semantics of the elaborated and abbreviated characterisations, i.e. on the basis of the respective action trees for Seq.

On the left of the figure, we show the evolution of Seq for the elaborated characterisation; and, on the right, we show it for the abbreviated characterisation. As can be seen, there is a single action set in each action window, for each characterisation. We discuss each window in turn, showing that their respective action sets match.

(1) When the Seq instance is set running, only the first instance A, according to the elaborated characterisation, is set running. This is determined by the value of the flag f being set to EXEC. In this characterisation, this is the only externally visible event that occurs, as a result of the Seq being executed. In the abbreviated characterisation, all three child instances of the Par are set running, according to f being EXEC, but only the first (A) is common to the elaborated characterisation. B' and C' should be considered to be type names which are not used in specifying Liesbet models. They are used solely here for the definition of the Seq as an abbreviation. As such, they do not have any external visibility, and the fact that their instances too are set running has no impact externally (i.e. in the rest of the model). So, the only externally visible effect of the Par being executed is, in the abbreviated characterisation also, the instance of A being set running.

(2) A finishing is matched in each characterisation.

(3) In the elaborated characterisation, the execute action (on B) will occur, in the next action window. In the abbreviated characterisation, the Go type (which is not visible to the rest of the model, and thus changes to its state have no external impact) will complete (because its GoQuery is now satisfied). When this occurs, the SeqCancel (immediately) executes B. The only action of interest, as indicated in the figure, is also the execution of B.

(4) Again, actions of interest match.

(5) As action window (3).

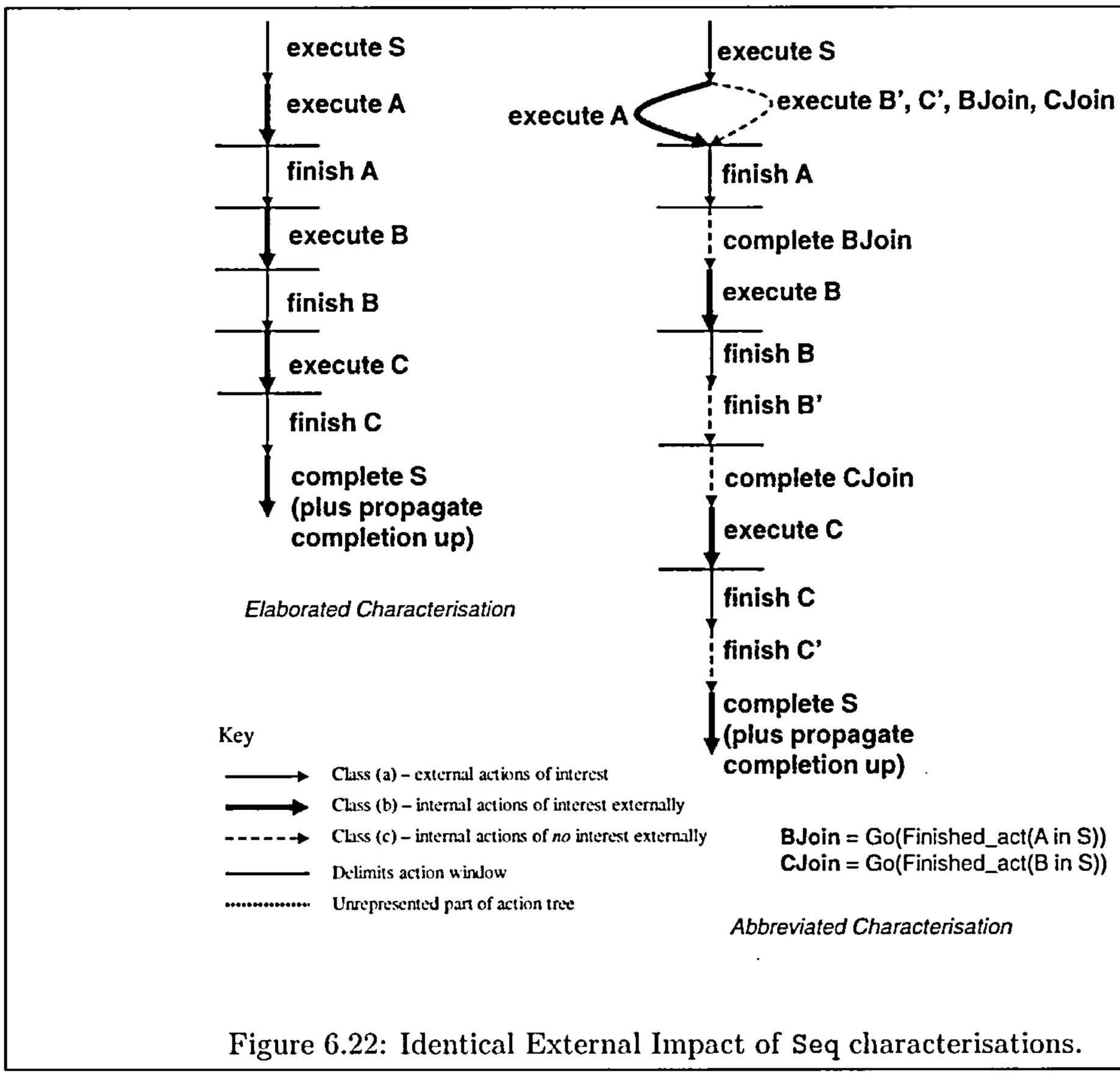


Figure 6.22: Identical External Impact of Seq characterisations.

(6) When C finishes, each characterisation (with respect to actions of interest) propagates completion up.

□

6.6 Concluding Remarks

In this chapter, we have presented a SitCalc-based characterisation of the Liesbet meta-model. An unequivocal advantage of using SitCalc for this purpose is that certain aspects of the intended semantics for Liesbet are captured quite straightforwardly such as: arbitrarily-complex synchronisation conditions, priority of structured instances over basic instances, and atomic propagation of side-effects through the activity instance hierarchy (see Section 3.2 for more information regarding the intended semantics).

Equally, a clear disadvantage is that, while the initial foundational axioms for workflow presented in Figure 6.12 are arguably clear enough, the augmented foundational axioms for generic activity types such as the choice types (e.g. Choice) and merge types (e.g. Multimerge, presented in Appendix Section B.1.4) are rather impenetrable. In contrast, the CCS/PCCS-based characterisations presented in Chapter Five are arguably a lot clearer in their meaning. As pointed out there, however, there is an apparent dichotomy between the clarity that comes from compositional, programming-like metaphors for characterising behaviour and the ability to model atomic arbitrary side-effects. It very much appears that, for the characterisation of Liesbet, the strengths of the logic-based approach (i.e. SitCalc) are the weaknesses of the process algebra- (i.e. CCS/PCCS) based and vice versa.

We have presented the definition of a translator function, $\mathcal{M}_{SitCalc}[\![-]\!]$, for Liesbet models which yields SitCalc basic action theories. We have also presented two results. The first relates to the guaranteed completion of Liesbet models in the context of assumptions relating to the absence of deadlock and livelock in a Liesbet model. The second demonstrates that the characterisations, presented in Section 3.4, of Liesbet constructs as abbreviations, in the set Liesbet_{abbrev}, are sound. They are shown to be so in that a particular Liesbet model characterised using a combination of the SitCalc semantics, described in this chapter, for Liesbet_{prim} and the set of abbreviations for constructs in Liesbet_{abbrev}, presented in Section 3.4, will necessarily be model equivalent to the same model characterised just with the SitCalc-based semantics presented here. This is an important result as it allows us to propose a core set of primitives for workflow. Being able to propose such a set enables us to articulate the true nature of workflow, and its fundamental representational requirements.

In the next chapter, we present our approach to verification of Liesbet workflows. Verification is an important tool to provide in a workflow modelling framework, as it is integral to the operation of the business that the definition of workflow models is sound.

Chapter 7

Verification of Liesbet Workflows

In this chapter, we provide details regarding the verification approach for Liesbet models that we have implemented in our work. We are able to prove both soundness and arbitrary temporal constraints, written in a language such as CTL*. We present a number of ways in which the complexity of verification may be ameliorated. Having presented these, we specify the algorithm that we use to perform verification and give an interesting characterisation of the complexity of our verification approach. Verification is an important tool to provide in a workflow modelling framework, as it is integral to the operation of businesses that the definition of workflow models is sound.

Note that our verification approach does not limit the specification of constraints to a temporal logic. In fact, constraints may be specified in any language for which a *progression function* (see Section 10.4) can be defined. To keep matters simple, however, we shall use the phrase "temporal constraints" in this and other chapters.

7.1 Soundness of Liesbet Models

Regarding the verification of Liesbet models, we are fundamentally concerned with the notion of model soundness, which is a property of the *control perspective*. Van der Aalst and colleagues have defined this property [120, 128]. We now present a definition of soundness, which is based on theirs, but adapted for our needs. A workflow model is sound (at the control perspective) iff it satisfies the following conditions:

- **Option to complete** – It should always be possible to complete a workflow instance
- **Proper completion** – A workflow instance should not signal completion while there is still work in progress
- **No dead activities** – For every activity instance that may be created in the enactment of a workflow model, there must exist at least one enactment path where the instance is run. This property ensures that every activity instance plays a meaningful role in the workflow model.

The first property, option to complete, stipulates that the workflow model should not be subject to locking along any of its enactment paths. We consider the possibility of two types of locking

– *deadlock* and *livelock*. The completion result from Section 6.4 states that the only source of deadlock in a `SitCalc`-characterised `Liesbet` model can be from instances of synchronisation types which never declare a result, i.e. go to `Completed`, or `Cancelled`. Also stated there is that the only source of livelock in a `SitCalc`-characterised `Liesbet` model can be from `Multi/MultiSeq` instances which forever spawn instances of their `ExecAct` types on account of their respective join conditions being forever satisfied. Note that our verification approach is such that we do not test for the eventuality of livelock.

According to the `SitCalc`-based semantics for `Liesbet`, a workflow instance is said to have completed, once the root instance has reached a finished state. As this may only occur once all of the root's descendant instances have themselves reached a finished state (by virtue of completion being propagated upwards, or cancellation being propagated downwards), a completed model necessarily entails a properly completed model.

As shown in Section 6.4, in the absence of any sources of locking, completion is guaranteed, i.e. completion is guaranteed *iff* there is an absence of locking. Thus, verifying `Liesbet` model soundness comes down to verifying an absence of model deadlock and an absence of dead activity instances, with the qualification that we do not test for the possibility of livelock.

7.2 Verification Runs and Options for Verification

If we are concerned solely with the verification of workflow soundness (as described in Section 7.1), and are not concerned with the verification of models against temporal constraints (as described in Section 7.3), we may split the verification of a `Liesbet` model into a number of *verification runs*, according to *isolated scopes*. Doing so may significantly improve the efficiency of verification – we do not need to consider the interleaving of enactment of instances between runs. We, thus, perform a partial-order reduction (POR) [38, 59] on the verification state space.

The process of splitting a `Liesbet` model into separate verification runs proceeds as follows. Starting from the root instance, we traverse the tree, in a depth-first fashion. Whenever we encounter an instance, which is an isolated scope, we replace it by an instance of the `Empty` type, and save the replaced instance as a distinct, new verification run. Eventually, the original model will constitute a single scope, with no "internal" isolated scopes. This will constitute the first verification run. For the "saved" instances, we take each in turn, and repeat the process. These become further verification runs. Once, we have no more saved instances to process, we stop – having generated a number of verification runs. These are then individually checked for workflow soundness.

Note that when splitting a `Liesbet` model into separate verification runs, instances of `Multi/MultiSeq` types are handled in a particular way, in order to ensure decidability of verification. Whenever an instance of a `Multi/MultiSeq` type is encountered (by the splitting process), we replace it by an instance of `Empty`. Then, we split its `ExecAct` type off into a separate verification run. This has the effect of the `ExecAct` type being an isolated scope. In doing this, we ignore the behaviour of join conditions in `Multi/MultiSeq` types, meaning that if there is an inherent source of livelock within such a condition, it will not be detected. This is the price to pay for decidability of verification.

On a practical note, this is not as restrictive as it may sound. For verification purposes, an

author could temporarily replace the use of a non-limited type by a limited multiple-instance (i.e. `MultiLimit*`) type, in order to assess the behaviour of the model. We do not place any prescriptions on the processing of limited types, for verification, meaning that their `ExecAct` instances are verified in the same run as ancestor instances of the containing `MultiLimit*` instance. If the type is susceptible to livelock, then this approach, assuming verification *tractability*, would detect it (given a large enough n – the limit on `ExecAct` instances).

In order to demonstrate workflow soundness, we need to show an absence of deadlock and dead activity instances. We may safely verify isolated scopes separately, as the visibility horizons of instances of synchronisation types – which are the only source of model deadlock – may not cross isolated scope boundaries.

Moreover, dead activity instances are those which are always cancelled – and thus never run – as a result of dead-path elimination (DPE), or explicit cancellation (by a `CancelActivity` instance). Firstly, the visibility horizon of a `CancelActivity` instance is not allowed to cross isolated scope boundaries. Thus, there is no need to be concerned with the effects of cancellation *within an isolated scope* where a `CancelActivity` instance *existing outside the scope* is initiating the cancellation. Secondly, for either DPE- or `CancelActivity`-initiated cancellation, if the *effects* of the cancellation (i.e. cancellation being propagated downwards) cross an isolated scope boundary, then there will be at least one instance in the parent model (which contains the scope) that will be affected. If cancellation of a particular sub-tree (crossing a scope boundary) occurs in all enactment paths (i.e. instances in the sub-tree are dead instances), this behaviour will be identified even when verifying the parent model separately from the isolated scope. Thus, the check for dead instances may safely be made in a number of runs, according to isolated scope boundaries.

We can help to improve the efficiency of verification some more, by performing further splits based on a similar notion to that of splitting on the basis of isolated scopes – viz. we separate those instances within a model, pertaining to an already-derived run, which fall within the visibility horizon of some other instance, from those instances which do not. We identify this as a second stage to the splitting process, already described for isolated scopes.

We identify the *reference instances*, which are used in determining the visibility horizons of synchronisation activities (i.e. instances of `Go` or `Stop` types), `CancelActivity` instances, and synchronisation rules.

- For synchronisation activities, when a query within a `Go` or `Stop` instance specifies a reference type, the instance to which the reference type resolves is counted as the reference instance for the query.

  If a query does not specify a reference type, and, instead, makes use of a global visibility horizon, then we ascertain the *least senior* instance that is a common ancestor (i.e. lowest common ancestor) of the querying instance and all instances of the customised type name, being queried. This common ancestor instance is counted as the reference instance for the query.

  If a query does not make use of any visibility horizon (as would be the case for the query `True`, for instance), the reference instance for the query is taken to be the `Go` or `Stop` instance itself.

  For any synchronisation activity, we collect together all of the reference instances for queries

used within the activity, discounting those which are descendants of others. If we are left with a single reference instance, then this becomes the reference instance for the activity. If there is more than one, the least senior common ancestor instance of the remaining reference instances is used as the reference instance for the activity.

- For `CancelActivity` instances that make use of a reference type, the instance to which the reference type resolves is counted as the reference instance for the `CancelActivity` instance.

  For `CancelActivity` instances that do not specify a reference type, and, instead, make use of a global visibility horizon, we ascertain the *least senior* instance that is a common ancestor of the `CancelActivity` instance and all instances, of the customised type name, being cancelled by the `CancelActivity` instance. This common ancestor instance is counted as the reference instance for the `CancelActivity` instance.

- For synchronisation rules, we consider the root instance of any sub-tree that may be affected by a synchronisation rule (i.e. an instance of `RType`) as a source instance. We consider any instances that are in the visibility horizons of the `CondQuery` or `GoQuery`, of the rule, to be the target instances of the source instance. For any source instance, the least senior instance that is a common ancestor of the source instance and all of its target instances is said to be a reference instance for the rule.

For any model, corresponding to an already-defined run, we identify all of the reference instances for synchronisation activities, `CancelActivity` instances and synchronisation rules, discounting those which are descendants of others. The run is then split into further runs at these reference instances. The model that is left is guaranteed to be sound with respect to deadlock-freedom, meaning that it just needs to be checked for (an absence of) dead instances. The runs that are split off are checked for deadlock-freedom and dead instances.

Moreover, as long as a model which has been determined to be sound with respect to deadlock-freedom does not use any choice, merge, or multiple-instance activity types (which are potential sources of cancellation in a model) then it is necessarily sound with respect to an absence of dead instances also.

The model `Par(Seq(A,B),Seq(B,C))` is an example of one that is necessarily sound. There are no reference instances that can be identified using the criteria for synchronisation activities, `CancelActivity` instances, and synchronisation rules. Thus, the model is necessarily sound with respect to deadlock-freedom. Moreover, it does not use any choice, merge, or multiple-instance activity types, which means that it is necessarily sound with respect to an absence of dead instances.

Note that, with regard to dead instance checking for a limited multiple-instance type, we consider that at least one instance of its `ExecAct` type should be run, along some enactment path of the containing `Liesbet` model. For unlimited multiple-instance types, we make no such prescription.

If we are concerned with the verification of a `Liesbet` model against a temporal constraint – see Section 7.3, then we must verify the model as a whole, with one exception. The exception is that we enforce the separation of `ExecAct` types for non-limited multiple-instance activity types (`Multi/MultiSeq`) into distinct verification runs, to ensure decidability. Constraint checking is applied to the individual verification runs separately. Given this convention, a workflow model author needs to ensure that the constraints that are checked make sense.

There are a number of verification options that are supported, relating to assumptions that can be made about the workflow engine that will, ultimately, enact a workflow model. These options are distinguished on the basis of what (structured, or basic) instances may be considered for progression next, in enacting a model.

Primarily, we are concerned with the verification of Liesbet models, which start life, and are enacted, as such. However, in our work, we also support the verification of WS-BPEL [87] models, which are translated to Liesbet, for the purposes of verification. Our verification approach is also potentially applicable to other workflow languages, assuming the existence of appropriate mappings to Liesbet. We have needed to take account of the likely enactment policies of these engines, in providing these verification options.

In brief, we support the following verification options:

- No prioritisation/prioritisation in enactment of structured instances over basic instances
- Non-determinism removal, in that at any point during the enactment of a model, there may be several instances that can be progressed. Either, we remove this non-determinism by saying that whenever there is such a choice, the instance with the lowest instance number will be progressed; or we say that any one of them may be progressed next, and thus the engine will make a non-deterministic choice between them. In the latter case, we should verify the effects of all possible choices.
- Just allow basic instance to complete versus allowing them to complete, or be cancelled.

In total, there are eight verification options from these possibilities alone. The implementation allows for alternative verification options to be implemented, if necessary.

7.3 Verification of Temporal Logic Constraints

In our work, we use the *Computation Tree Logic*, CTL*, for the description of constraints over the enactment of workflow models. CTL* formulas describe properties of *computation trees* – a SitCalc situation tree being an example of such a tree. A (possibly infinite) transition system, described by a CCS agent, may also be represented as a computation tree.

The logic CTL* subsumes the temporal logics LTL and CTL. Linear Temporal Logic (LTL) is useful for reasoning over properties of *individual paths*, which must hold true of *all paths*. For example, in LTL, we can say: *if "p" holds true at a state in the path, then "q" must hold true at some state* (Fp→Fq). There is no way of expressing such a property in CTL. CTL is useful for reasoning over properties of several (i.e. *all* or *some*) paths leaving a state. For example, in CTL, we can say: *if "p" holds true at some point along all enactment paths from the current state, then "q" must hold true at some point along some enactment paths from the current state* (AFp→EFq). There is no way of expressing this property in LTL.

In this section, we will be relating the verification of CTL* constraints to our SitCalc-based characterisation of Liesbet. In doing so, we have to be rather careful about using the terms *states* and *situations*. In most cases, when the term *state* is used, this may be read as *situation*. However, these terms are not synonymous. Two states are the same iff their fluent state is the same. Two situations are the same iff their action histories are the same. The occasion when this distinction matters is when two distinct situations have identical fluent state. This is what we call a *matched*

state, as the fluent states pertaining to these situations match. That is, a situation, in a situation tree (pertaining to a model of a `SitCalc` domain theory), is a matched state iff there is some other situation in the tree with the same fluent state.

The language of *Propositional CTL** is described by the following definitions (in Backus Naur Form – BNF, see [90]). We divide CTL* formulas into two classes, those which are evaluated in states, and those which are evaluated along paths.

- State formulas, where θ is any path formula

  ϕ ::= $\top$ | p | $(\neg\phi)$ | $(\phi_1 \vee \phi_2)$ | E[θ]

- Path formulas, where ϕ is any state formula

  θ ::= ϕ | $(\neg\theta)$ | $(\theta_1 \vee \theta_2)$ | $(\theta_1 \mathrm{U} \theta_2)$ | Xθ

We make use of a number of abbreviations in our presentation of CTL*. We use the term *primitive symbols* for those symbols that we have just used for the definition of state and path formulas. The following abbreviations all make use of just primitive symbols on the right-hand sides of their definitions:

- State formulas:
  - $\bot \equiv \neg\top$
  - $\phi_1 \wedge \phi_2 \equiv \neg(\neg\phi_1 \vee \neg\phi_2)$
  - A[θ] $\equiv \neg$E[$\neg\theta$]
- Path formulas:
  - $\theta_1 \wedge \theta_2 \equiv \neg(\neg\theta_1 \vee \neg\theta_2)$
  - $\theta_1 \mathrm{R} \theta_2 \equiv \neg(\neg\theta_1 \mathrm{U} \neg\theta_2)$
  - F$\theta \equiv \top \mathrm{U} \theta$
  - G$\theta \equiv \neg(\top \mathrm{U} \neg\theta)$

For *state formulas*:

- E – *somE paths*, or *there Exist paths* – requires that along some paths from the current state the property holds
- A – *All paths* – requires that along all paths from the current state the property holds

For *path formulas*:

- X – *neXt* – requires that a property holds in the second state (#1 if numbering from zero) of the path
- F – *eventually*, or *in the Future* – requires that a property will hold at some state on the path
- G – *Globally* – requires that a property will hold at all states on the path
- U – *Until* – requires that the first property holds on the path up to (but not including) the state where the second property holds. It also requires that the second property will eventually hold, on the path.

- R – *Release* – is the logical dual of U and requires that the second property holds up to and including the first state where the first property holds. Notably, the first property is *not* required to hold eventually.

The semantics of formulas that can be generated using the primitive symbols are as follows. Semantics for the other symbols follow from their definitions as abbreviations.

- $\mathcal{M}, s \models \top$
- $\mathcal{M}, s \models p$ **iff** ... *see below* ...
- $\mathcal{M}, s \models \neg\phi$ **iff** $\mathcal{M}, s \not\models \phi$
- $\mathcal{M}, s \models \phi_1 \vee \phi_2$ **iff** $\mathcal{M}, s \models \phi_1$ **or** $\mathcal{M}, s \models \phi_2$
- $\mathcal{M}, s \models$E[θ] **iff** there is a path π from (and including) s such that $\mathcal{M}, \pi \models \theta$
- $\mathcal{M}, \pi \models \phi$ **iff** s is the first state of π and $\mathcal{M}, s \models \phi$
- $\mathcal{M}, \pi \models \neg\theta$ **iff** $\mathcal{M}, \pi \not\models \theta$
- $\mathcal{M}, \pi \models \theta_1 \vee \theta_2$ **iff** $\mathcal{M}, \pi \models \theta_1$ **or** $\mathcal{M}, \pi \models \theta_2$
- $\mathcal{M}, \pi \models \theta_1 \mathrm{U} \theta_2$ **iff** there exists a $k \geq 0$ such that $\mathcal{M}, \pi^k \models \theta_2$ **and for all** $0 \leq j < k$ $\mathcal{M}, \pi^j \models \theta_1$, where π^i denotes the suffix of path π starting at state i
- $\mathcal{M}, \pi \models$ Xθ **iff** $\mathcal{M}, \pi^1 \models \theta$

In order to verify the satisfaction of constraints against Liesbet models, we use a progression algorithm, as described in the next section. This is essentially an *on-the-fly* model checking approach, and is similar in nature to automata-based approaches to model checking. In using a progression algorithm, we expose propositions that need to be satisfied in various states. For atomic propositions (replacing p in the CTL* syntax definition), we allow the use of atomic query formulas without reference types (so-called simple queries), from Section 3.1.4.

p ::= Completed_act(α) | Completed_all(α) |
Cancelled_act(α) | Cancelled_all(α) |
Finished_act(α) | Finished_all(α) |
Running_act(α) | Running_all(α) |
Initial_act(α) | Initial_all(α)

When a p proposition needs to be evaluated in a given state, we do so according to the following SitCalc-based characterisation. We show the appropriate evaluations of Completed queries. The evaluations of the remaining simple queries follow naturally.

- $\mathcal{M}, s \models$ Completed_act(c) iff ($\exists$i,c').[CType(i,s)=c' $\wedge$ IsType(c',c) $\wedge$ State(i,s)=Completed]
- $\mathcal{M}, s \models$ Completed_all(c) iff ($\forall$i,c').[CType(i,s)=c' $\wedge$ IsType(c',c) $\supset$ State(i,s)=Completed]

`s is current situation, starts at` S_0
`TC is initialised with CTL* constraint to verify`

(I) Make a "running list" comprising all activity instances present in the model. This list is *non-backtrackable.*
(II) LOOP
(1) (i) If s is a *matched state* (determined on the previous iteration, see step (4) – for first iteration, it will not be a matched state), then we re-use the same state identifier *id* used for the previous situation sharing the same fluent state. If, in progressing TC for *id*, we are able to establish a result for TC in s based on new or past results for *id*, then we propagate this result up and stop progression.
(ii) If s is not a *matched state*, we progress TC in s, and stop progression if we are able to establish a result for TC in s, having propagated the result up.
(2) If a result is established for TC in s, then after propagation, do the following based on the result for TC established against S_0:
(i) FALSE: REPORT FAILED VERIFICATION, and STOP
(ii) TRUE: there is no need to continue with verification of TC, as we have established its validity; but we continue verification for soundness unless s is a matched state, in which case we backtrack to last choice point
(iii) UNDEFINED: We continue verification unless s is a matched state, in which case we backtrack to last choice point
(3) Remove from the "running list" any instances which are in a Running state
(4) Select a next action, a, to do in s, according to the action pre-condition axioms. If the action results in a new situation s' which *state-wise* we have visited before, then this is a matched state (which we deal with appropriately on the next iteration of the loop). If no action is possible, and the workflow instance is not in a completed state then REPORT FAILED VERIFICATION and STOP else backtrack to last choice point.
(III) If the "running list" has instances left on it, then these are dead instances, so REPORT FAILED VERIFICATION. Otherwise REPORT SUCCESSFUL VERIFICATION.

Figure 7.1: Verification Algorithm for Liesbet

7.4 Algorithm for Verification of Liesbet Models

As stated previously, in Section 7.2, the verification of a workflow model is split into a number of verification runs. Whether we wish to verify a model against temporal constraints and/or verify model soundness, the algorithm is largely the same. The granularity of the runs, when verifying temporal constraints will be (typically) much larger than that when verifying model soundness. To verify model soundness, we may either use the same runs generated for the constraint verification, or we may use a set of runs which has been generated specifically for the purpose. This is provided as a verification option, as sometimes it will be more efficient to verify a separate set of finer-grained runs to check for model soundness, even if we are also verifying temporal constraints.

In Figure 7.1, we present the run-based algorithm for verification of model soundness and any presented temporal logic constraints. If a violation of model soundness (deadlock or dead

instances), or a violation of extant constraints, is identified, the verification run reports failure. For constraint verification, we *progress* constraints from one state to the next according to a progression function, prog/3, which is defined in Section 10.4.

7.5 Verification Complexity

The verification problem is inherently exponential in nature in terms of the *worst-case* size of the minimal activity instance state space that must be explored (see Section 5.7, for information concerning what is meant by "minimal state space"). This can be seen easily in considering the simple workflow model Par(A_1,...,A_n), where A_1,...,A_n are basic instances. If we consider just basic instances being able to complete, then the size of the instance state space will be 2^n+1. This may be computed simply by considering an n-bit binary number, with zero (say) indicating "not Completed", and one indicating Completed. We do not need to distinguish Initial from Running as execution, according to the SitCalc-based semantics, occurs as an implicit side-effect. The exception to this is that we count the state where all instances are in the Initial state, hence we add one to the complexity expression. The possible values of an n-bit binary number are 2^n. As such, the total complexity is 2^n+1. For basic instances being able to complete, or get cancelled, the complexity may be determined, similarly, as an n-bit ternary number, viz. 3^n+1.

In assessing verification complexity, the largest exponent is limited to the number of instances of childless types in a model; and thus the complexity of verification is $O(m^n)$ (or *big-oh* m^n), where n is the number of childless instances, and m is either two or three, depending on whether we have a model where childless instances may get cancelled as well complete. Models whose childless instances are all basic activity instances are the only candidates for which such a distinction may exist. For these, we may consider either that basic instances may just complete (in which case m is two), or that they may also get cancelled (in which case m is three). For all other models, some of the childless instances will have the possibility of being cancelled, and as such m is three. Child-bearing instances do not have any exponential impact on verification complexity, as their progression occurs as implicit side-effects of the progression of childless instances.

The mix of activity types in a model means that, although models will generally demonstrate some exponential characteristics in their verification complexity, the factors involved should often be considerably less than the number of childless instances involved in the model. For instance, a Seq of Pars will have p+1 complexity, where p is the sum of the individual Par complexities. The exponents involved in the complexity characterisation of such a model fragment will be bounded by the highest number of descendant childless instances of any of the Par instances. Moreover, descendant instances of the Par instances may further limit verification complexity.

For instance, one of the Par instances may contain two Seqs, which each have three basic instances as children. Again, considering it as an n-bit binary number, where n this time is six, corresponding to the number of basic instances, we have a bounded complexity of 2^6, which is 64. However, this may be reduced by $(4+2)\times 2^3$ because instances of the Seqs have to execute in a particular order. So, even though the Par prescribes an exponential factor which is determined by the total number of basic instances which it has as descendants, we are able to subtract a significant number (of impossible states), viz. 48, because of the criteria concerning the order in which completion of these instances may occur. Of course, if the initial exponent is too high then

any possible subtractions may lose their significance.

In any case, it is noteworthy that any POR-based improvements which can be made, such as removing model fragments for which soundness is guaranteed (see Section 7.2), will have a significant effect, in removing childless instances from (worst-case) exponents, in all but the simplest of models.

7.6 Concluding Remarks

In this chapter, we have presented our approach to design-time verification of properties of `Liesbet` models. We have described what it means for `Liesbet` models to be sound, and have elucidated various optimisations with respect to verification efficiency. We have also discussed the verification of `Liesbet` models against arbitrary constraints written in the temporal logic `CTL*`; and have presented an algorithm which effects verification of both soundness and such arbitrary constraints. We have concluded with a brief discussion of the complexity of the verification task, in terms of the state space that is explored.

In the next chapter, we shift focus somewhat by moving to a presentation of our work concerning the modelling of flexible workflow. We consider this to be of equal importance as our previous work. For instance, being able to support collaborative workflows like traditional workflows are typically of high value to a business.

Chapter 8

Flexible Workflow Modelling

In preceding chapters, we have considered the modelling of traditional workflow. A particularly significant characteristic of traditional workflow is their brittle nature in the face of exceptional behaviour. Furthermore, they are not well suited to the definition of collaborative workflow, where agents should have the facility to decide collectively how a prescribed task or process should be realised. In light of these issues, there is a need to consider the modelling of flexible workflows. Moreover, it is clear that more can be done to address this issue, and, in this chapter, we provide a contribution to its resolution.

We start with a description of flexible workflow modelling, including the presentation of a review of related works in the field. Then, we introduce our approach to flexible workflow modelling, which may be neatly summarised as *Flexible Workflow = Abstract Model + Policies for Refinement.* In this context, we identify a correspondence between refining an abstract workflow (specified for flexible enactment) into a concrete one, and the operation of an *Hierarchical Task Network* (HTN)-based planner, which refines abstract task networks into concrete ones. We present a brief overview of HTN-based planning, followed by the description of an HTN-based planner, called `Theodore`, which we have implemented. `Theodore` constitutes an additional contribution of our work. Finally, we present a description of how we have used `Theodore` for the modelling, verification and planning for enactment of flexible workflow models.

8.1 Flexible Workflow Modelling

In the introduction to this thesis (see Section 1.1), we mentioned how measures of flexibility might be introduced into workflow models so that they are better able to handle exceptional behaviour.

If the handling of exceptional behaviour is supported at all – in most commercial Workflow Management Systems (WfMSs) it is not [25] – it addresses the issue of exceptional behaviour which may be considered as erroneous, that is, *exceptions as errors.* Support for handling such behaviour, in academic contributions, has focussed on the use of Event Condition Action (ECA)-rules, or some similar artefact. In the context of workflow modelling, these will specify some combination of event occurrence, and condition on workflow state, which if satisfied causes the action specified in the rule to be effected. The action will serve to recover the workflow from its erroneous state so that execution may continue. The action may specify additional tasks that need to be carried out, for example.

Support for handling this kind of exceptional behaviour is essential in workflow modelling, and ECA-rules represent an effective means of such support. However, what we are concerned with in this chapter is another sort of exceptional behaviour, which is largely orthogonal, albeit in some circumstances it may provide a better, or more appropriate, alternative than considering certain behaviour to be erroneous. We would label this other sort of exceptional behaviour as *exceptions as alternatives.*

A workflow author may have in mind a preferred or default realisation of a workflow model. However, he or she may choose to make other alternatives available, which will also have the effect of realising the desired outcome. This notion enables greater flexibility in the enactment of workflow models, where models may be enacted differently according to current business priorities and objectives, which are codified as *operational policies.* These policies may reflect higher-level business objectives[1], or may be sourced from the need to meet Service Level Objectives, captured in customer Service Level Agreements [68].

Depending on the domain context, allowing the *as alternatives* sort of flexibility within workflow models may not even be considered as accommodating exceptional behaviour. That is, the emphasis may lie more heavily with the *alternative* aspect rather than the *exception* aspect. That said, the distinction between exceptions as errors and exceptions as alternatives is a useful one to make, as long as this caveat is kept in mind. Support for both sorts of exceptional behaviour is essential in workflow modelling. In this thesis, our interest lies solely with the *as alternatives* aspect, as this is where we are primarily motivated.

The flexibility that comes from support for exceptions *as alternatives* is essential, as workflow, with its roots in capturing manufacturing processes, is often too rigid and brittle an artefact for capturing the operation of business processes [96]. In traditional workflow modelling, we are concerned with a representation of the control flow perspective, where every possible enactment path through the model needs to be explicitly enumerated. Although flexibility can be captured by traditional approaches, the requirement to enumerate all ways of enacting a model quickly becomes laborious and impracticable. As a result, traditional workflow approaches almost always have no or little inherent flexibility, and are typically brittle to exceptional behaviour, with no room for flexible adaptation according to operational policies, as a result. In this sense, traditional modelling approaches may be seen as expressing what *should* be done. In contrast, flexible workflow modelling may be seen as expressing: what *could* be done.

The notion of flexibility is fundamentally addressed by putting measures in place which can capture flexibility without the need for explicit enumeration. Our support for flexible workflow modelling may be captured by the slogan: *Flexible Workflow = Abstract Model + Policies for Refinement.* That is, we recommend the definition of a somewhat abstract workflow model, which is refined into a concrete instance by the use of a number of operational policies.

Operational policies are typically coded as *business rules* [101]. Business rules are espoused as

[1]An example of such a policy may be one relating to maximising customer satisfaction (in insurance claims handling). For small insurance claims, we may seek to constrain the processing time for a claim to be no more than 5 days, at the possible expense of additional cost to the business. Normally for such insurance claims, damage to a motor vehicle would be fully investigated by an inspector. If, however, it is not possible for an inspector to assess the car in time to meet the turn-around time requirement, then it may be better, if the claim is small, to forgo the inspection in order to meet the objective of maximising customer satisfaction, even if it lays the company bare to the possibility that it will be defrauded.

a means of supporting agility within enterprises to react to, and proactively plan for, changes in market conditions. Rules are promoted as *separating the know from the flow* [101], i.e. they allow the knowledge that an enterprise has about itself to be externalised in a declarative form from workflows and procedural code. The idea is that the business logic captured by rules is easier to comprehend, change, and maintain by the individuals who have primary interest in their definition, namely business managers and analysts, rather than requiring the services of developers to effect the desired changes to the business logic.

Thus, our notion of flexibility would appear to fit in well with what is current practice in providing automation within the enterprise. A problem with current solutions is that rules and workflow engines are distinct artefacts which are integrated in an *ad hoc* fashion. A unified approach for the modelling of rules to constrain the enactment of workflow is thus lacking. This chapter addresses this point.

Another aspect of flexible workflow modelling is support for the modelling of *collaborative* workflows (introduced in Section 1.1). In collaborative workflows, agents decide collectively the way in which to enact a workflow. As will be described, many processes enacted within a business context will be of this nature; thus a means of modelling such workflows is also of importance.

We now describe a number of approaches in the literature which have primarily considered the *as alternatives* aspect of flexible workflow modelling. We will return to these works at the end of the chapter, in order to place our contribution into context.

8.1.1 Case Handling Systems CHSs

Case Handling Systems (CHSs) [127, 16, 96] have emerged, in recent years, with the expressed aim of offering what current Workflow Management Systems (WfMSs) lack: visibility of the entire case (i.e. process instance), and, of primary interest here, *flexibility.* Notably, in many enterprises, the practical application of WfMSs has been limited to the support of simple and well-defined business processes [96, 16]. According to [16], this limitation is caused mainly by WfMSs being founded on a *manufacturing* metaphor, where a WfMS effects a production line, rigidly routing work items to various agents in turn.

This approach is the diametrical opposite of everyday practice in many business scenarios, where flexibility in how a work case is processed is key. In CHSs, the logistical state of a particular work case (including its completion) is determined by the state of data objects, and not by routing (i.e. control flow) [127]. Workers have authority to complete data objects at various times, and have role-based authorisations assigned to them to view particular items of data associated with a case. The so-called *context tunnelling*, inherent in WfMSs, where the visibility that a role has on case data is determined by which work items it is currently working on, is thus removed by CHSs. Context tunnelling is a key inflexibility issue for WfMSs.

Data-driven business process enactment represents a shift in focus from the approach taken in traditional workflow modelling and enactment. CHSs define just the limits of what can be done, and thus follow the *could rather than should* tagline referred to above. However, as [127] points out it is not always desirable to lose the rigid control imposed by production workflows. Offering greater flexibility can tend to make the workflow specification less clear (depending on how it is expressed). Moreover, there are scenarios for which it is conceivable that a mixture of rigid control over some parts of a workflow specification, but with some measure of flexibility regarding others,

would be appropriate. As [16] describes, "flexibility is an essential condition .... This does not take away from the fact, however, that parts of the process can, and even should, be regarded as an actual production process. In this context we speak of production workflow. This also needs to be adequately supported." Providing a way of specifying such a mixture of production workflow *and* flexibility is a principal aspect of our work, as described in this chapter. It is also fundamentally facilitated in Case Handling Systems.

8.1.2 CrossFlow

CrossFlow [55] was a European project aimed at facilitating support for cross-organisational workflows in dynamic virtual enterprises (DVEs). Virtual enterprises aggregate skills and core competencies (packaged as services), from multiple organisations, to create composite services. Their dynamism stems from the fact that they may be composed spontaneously, and may be torn down with minimal impact. The lifespan of a DVE can thus be rather short. Enactment of composite services in Crossflow is realised by dynamically linking the WfMSs of the participating organisations. An aspect of the work was the requirement to support the definition of flexible workflows whose enactment could be tailored to maximise Quality-of-Service (QoS) metrics. To this end, Crossflow defined a workflow meta-model consisting of the usual workflow constructs such as OR-join, OR-split, AND-join, AND-split, as well as additional constructs, known as *flexible elements*, that allow the provision of enactment alternatives, viz, [55]:

- *Alternative activities* allowing the specification of different activities, of which exactly one may be chosen. For instance, a model may provide time-expensive high-quality options as well as quick low-cost options.

- *Non-vital activities* allowing activities to be omitted in enactment. For instance, in extreme situations, it may be beneficial to sacrifice an instance of such an activity in favour of other higher priority goals.

- *Optional execution order* allowing the specification of a preferred ordering of activities that can be overridden. Reordering may prove to be beneficial if other goals of higher priority are then achieved.

8.1.3 Collaboration Management Infrastructure (CMI)

The Collaboration Management Infrastructure (CMI) [107] was developed to manage *collaborative* workflows in both traditional and virtual enterprises. [51] describes an application of CMI in the context of *Crisis Mitigation*. Crisis Mitigation constitutes a challenging application for workflow technology. Its unpredictability forbids predefining a concrete crisis mitigation strategy and requires dynamic reaction of people involved in the mitigation. The authors argue that processes for Crisis Mitigation must *empower coordinators* and *experts* to deal with unexpected situations by permitting *coordination flexibility* and *dynamic change*, while providing enough structure to prevent chaotic response and increase mitigation effectiveness. It argues that this combination of structure and flexibility cannot be provided by current workflow-like technologies. Coordinators "determine the need of new activities and organisation structures, and delegate existing and new

activities to process participants", and, experts "perform specialised crisis mitigation activities and have the skills to decide the exact type or specialisation of these activities".

For flexible process definition, CMI provides:

- *Activity Placeholders* allowing for activities whose concrete types are left open at design-time. Coordinators and experts resolve the concretisation of placeholders using *Resolution Rules*, which provide policies for how they may be resolved.

- *Repeated Optional Dependencies* – While a milestone has been reached in the control flow, and has not yet expired, an activity may be repeated a number of times, which is not predetermined by the control flow specification.

8.1.4 Wainer and Colleagues

Wainer [131] puts forward the argument that processes in workflow-like applications should be represented in a logic language which allows for a *unified representation of processes, constraints and policies.* He argues that it is widely accepted that office procedures are much more creative and mutable than can be accommodated by traditional workflow applications; and, in dealing with real work cases, office workers creatively subvert the standard processes to get the job done. He asserts that there is a growing recognition that workflow applications should admit flexible and adaptable specifications, and should be able to cope effectively with wide-ranging exceptional behaviour. However, as noted, a principal concern for enterprises is that flexibility in process enactment is controlled so that organisational rules are not violated, and business objectives are achieved.

In [131], a workflow model is specified as a theory in a linear modal logic. Constraints and policies are added as further axioms. Exceptional behaviour is accommodated, as much as possible, by the notion that the (minimal) models of the theory specify what could, rather than should, be done. In this sense, the WfMS should be seen as more of a *querying mechanism*, where agents may query what they can do next. This is very similar to what we provide in (the implementation of) our flexible workflow modelling approach. The relation that is captured between the work case, and the model, is one of consistency rather than instantiation. Wainer also introduces the notion of soft constraints, where constraints may be prioritised in importance, and overridden if needed in the event of obtaining exceptions. The work also describes how it may be determined that a case, thus far, complies with a workflow model, as well as how it may be determined whether a case can be migrated to an alternative model. In the latter case, this would mean that the original case also, thus far, complies with the new model. A point made by Wainer, and of key importance, is that it is unclear whether such a logic-based representation would be an efficient one in practice.

In [132], Wainer and colleagues describe *Tucupi* – another flexible workflow system, based on overridable constraints. The flexibility is achieved through the definition of constraints on the execution of activities, which are pre- and post-conditions on the execution of other activities. There is no explicit control flow specified for a model. The authors propose a framework which includes a workflow server which effects the approach, and an access control model representing users having authority to execute activities and authority to override the constraints specified by activities. The framework is also able to help users decide which activity to choose to execute through what-if scenarios. Essentially, this approach facilitates the specification of what could be done by a domain expert, rather than what should be done, as is the case with traditional,

production workflow.

8.1.5 Organisational Modelling

It is worthwhile briefly mentioning the classification of workflow from yet another perspective, namely, the *organisational perspective*. Most of today's WfMSs focus on the process definition and oversimplify the organisational perspective [67]. Our conceptualisation of the organisational perspective is concerned with the following aspects.

- Management of Agents.
- Access Control to Enterprise Data.
- Operational Policies.

We proceed to describe the first two of these. We have already alluded to the use of operational policies in this chapter, and will not elaborate this further here.

8.1.6 Management of Agents

Specifying meta-models for the management of agents is given short shrift in most WfMSs. There has been some work to address this point, such as that presented in [141, 19, 20]. A WfMS should provide a capability to specify a dynamic and fine-grained model of agents that unifies the specification of role and authority structures.

A *role structure* is a partial ordering on roles within an organisation, where privileges propagate down the ordering. (More specific roles exist lower down the ordering.) For example, *c-programmer* is a more specific role than *programmer*, and should assume all of the privileges of programmer. An *authority structure* is a partial ordering specifying the organisational structuring of the enterprise to which the workflow pertains. The CEO of an enterprise has more authority (in the context of the enterprise) than any of the middle-managers, for example. Task assignment to agents, for instance, may be done on the basis of what positions agents occupy in both role and authority structures.

Role and authority structures should be dynamic and fine-grained; the relations that are described in a structure may change over time, and they may have temporal qualifications associated with them or may have exceptions.

8.1.7 Access Control to Enterprise Data

The management of (persistent) enterprise data, from the perspective of WfMSs, has had little attention in the literature. Whereas there have been a number of research efforts that have addressed the issue of task assignment (by means of access control models), such as [19, 20], there has been little research relating to the control of access to enterprise data from workflow models [138].

In traditional WfMSs, the data that is required for the execution of an activity is specified by the workflow designer. That apart, there is no, or little, flexibility in what can be accessed. In effect, therefore, there is no separation made between work distribution and authorisation. An agent is only authorised, in the processing of an activity, to see data that the author of the workflow

deems to pertain to that activity. However, the author of the workflow is unlikely to have the same level of expertise as a domain expert. Moreover, it is hard to prescribe at workflow build-time the exact data that will be required to process an activity of a case. The inability of an agent to view any data other than that which is prescribed at build-time is called context tunnelling (see Section 8.1.1).

A solution to context tunnelling is to detach authorisation from distribution by means of an independent, separate access control model. Such a model would be used to determine accesses to enterprise data from applications that are wholly unrelated to workflow enactment, as well as from applications that are. It would safeguard access to enterprise data according to enterprise-wide access strategies or constraints (such as those related to security). In the context of workflow enactment, such a model would possibly be augmented by additional or substitutive constraints to data access. This augmentation may occur for workflow enactment generally, or for specific workflow models, or instances thereof.

8.2 Flexible Workflow Modelling using Theodore

Our approach to flexible workflow modelling may be neatly summarised as *Flexible Workflow = Abstract Model + Policies for Refinement.* It is in this context, that we identify a correspondence between the refinement of an abstract workflow (through the use of policies) into a concrete workflow (to be enacted), *and* the refinement of abstract task networks into concrete ones (using similarly-conceived rules) in *Hierarchical Task Network* (HTN)-based Planning [85].

Fundamentally, our mechanism for facilitating the refinement of abstract workflow tasks according to policies is to make use of an HTN-based planner to guide the refinement process so that a concrete workflow is generated which conforms with:

- The business objectives of the enterprise, as represented in the *decomposition* rules specified in an HTN-based domain description (or *planning problem*)
- Subjective criteria that may be applied by the agents involved in the refinement process, such as in the context of collaborative workflows where a number of agents would agree on how an abstract workflow should be refined.

In explicating our approach to flexible workflow modelling, we start with an overview of HTN-based planning, and then continue with a description of our approach to HTN-based planning.

8.2.1 Hierarchical Task Network (HTN)-based Planning

The distinguishing features of a Hierarchical Task Network (HTN)-based planner over traditional (operator-based) approaches to planning [52] is what it plans for, and how it plans for it. A (purely) operator-based planner will work (regressively, progressively, or by a combination of the two) to find a (partially or totally) ordered set of actions that takes the world from an initial state specification to a goal state. The operation of such a planner is a search through a space of states, or space of partial plans. In contrast, HTN planners search through a space of decompositions, or refinements, of an initial task network. Note that we use the terms refinement and decomposition synonymously.

Task networks are much richer in structure than classical planning attainment goals [40]. In classical planning, any ordered set of actions, which, when applied in the initial state lead to a goal state, constitutes a plan. There is little control over which actions may be used in the plan, without going to an extremely fine-level of granularity in modelling the domain. On the other hand, HTN planning affords full control over the actions in a plan. Only those actions which are derived from the applications of operators to tasks, which themselves have been derived through successive refinements of the initial task network, may appear in the plan. This expressivity, afforded to HTN-planning domains, can be very useful in many planning applications. In fact, any sort of planning application where there is a notion of procedure for achieving a goal is likely to be a strong candidate for HTN planning. As there are typically procedures that underlie a domain, characterised by HTN planning, this sort of planning is sometimes called *template*-based planning to reflect the notion that plans follow a template corresponding to a procedure.

A good example of an HTN-based planner is SHOP2 [85, 10]. It is different from most other HTN-based planners in its use of *ordered task decomposition* (OTD). In planning by ordered-task decomposition, actions are added to the plan in the order that they will be executed. This means that the current state is known at each step of the planning process. This allows for greater expressive power in the planning system, such as the ability to use foreign agents, or *oracles*, because we are able to reason about what is true when applying an operator rather than constraining what has to be true (as in non-OTD HTN-based planning).

An HTN planning problem specifies a number of *methods* and *operators*, which are collectively known as domain constructs. An HTN *task* is a planning artefact that is meant to be *decomposed* by the application of these constructs. A method specifies sufficient conditions for the enactment of a task network to constitute the enactment of a *non-primitive* task. An operator specifies sufficient conditions for the enactment of an *action* to constitute the enactment of a *primitive* task.

Actions are physical artefacts that are meant to be performed by the plan enactor, and are not meant to be decomposed. An HTN task network is without loss of generality a partially-ordered set of ≥ 0 (non-primitive or primitive) tasks and ≥ 0 actions. An HTN planning task is concerned with decomposing an initial network of tasks and actions, and terminates successfully when a network of actions is reached.

Methods and operators may specify preconditions for their applicability. HTN planning assumes the use of a knowledge base, appropriately initialised, as well as a suitable language for querying and updating the knowledge base. Operators may also specify effects (i.e. updates) to be made to the knowledge base, as a consequence of the conceived execution of an action.

As a simple example, consider the `Liesbet` model that we have used for illustration throughout this thesis: `Par(Seq(A,B),Seq(C,D))`. In this form, the model would be fully-decomposed. We could alternatively cast this workflow as an HTN planning problem. The initial task network would be the non-primitive task `P`, say. Then, we would have three methods, viz.

- `P: true: Par(S1, S2)` – decomposes `P` into a network, i.e. `Liesbet` model, consisting of a `Par` as root, with two tasks, `S1` and `S2`, as its children. The precondition for application of the method is empty (or `true`), which is trivially satisfied.

- `S1: true: Seq(A',B')` – decomposes `S1` into a network consisting of a `Seq` as root, with two tasks, `A'` and `B'`, as its children.

- S2: `true`: `Seq(C',D')` – decomposes S2 into a network consisting of a `Seq` as root, with two tasks, `C'` and `D'`, as its children.

We would also have four operators, viz.

- `A'`: `true : true`: `A` – decomposes task `A'` into the action `A`, where the applicability is determined by a precondition (here, `true` – the first of them), and the effects of executing `A` are determined by an effects statement (here, also `true`, which signifies no updates to be made to the underlying knowledge base).

- `B'`: `true : true`: `B` – decomposes task `B'` into the action `B`.

- `C'`: `true : true`: `C` – decomposes task `C'` into the action `C`.

- `D'`: `true : true`: `D` – decomposes task `D'` into the action `D`.

Finally, there are four actions, which are physical activities that can be performed by the plan enactor. These are: `A`, `B`, `C` and `D`.

An `HTN` planner, starting with the task `P`, would select an appropriate method or operator to decompose it. There is only one such method, and the result of decomposition would be: `Par(S1,S2)`. The planner would then select one of `S1`, or `S2`, to decompose next. Let us arbitrarily pick `S1`. Alternatively, we may employ some heuristic that guides the selection. The resulting network is then: `Par(Seq(A',B'),S2)`. Note that whenever a method has been immediately previously applied, we need to select the next decomposition from the network specified by this method, and not from the entire task network being planned over. This ensures that when preconditions are evaluated in methods, they hold when the first action resulting (eventually) from the method decomposition – there may a number of further decompositions in between – is executed, thus maintaining soundness. In the example, we next need to choose a decomposition from `Seq(A',B')`. In effecting OTD, we respect the partial-ordering imposed by the task network, i.e. `Liesbet` constructs. As such, there is only one possible decomposition, which is to use the apposite operator to decompose `A'`. The decomposition of `A'` is `A` resulting in the network: `Par(Seq(A,B'),S2)`, and current plan: `[A]`. This is the first operator application. Whenever these occur, they get inserted into the plan, generated as a result of the planning exercise. `HTN` plans, at least traditionally, are sequential artefacts.

Whenever a task gets decomposed (by an operator) to some action, as well as being relabelled with the action name, the task is marked as being completed, so that dependent tasks in the network get enabled (if otherwise appropriate). In the example, we mark `A` as being completed so that the sequence, `S1`, may be progressed, making its next (leaf) task, `B'`, available for decomposition.

As the previously applied domain construct was not a method, we are at liberty to select the next decomposition from the entire network, in its current form: `Par(Seq(A,B'),S2)`. Let's say that we next select `S2` for decomposition (by method), followed by `C'` (by operator). The resulting task network is: `Par(Seq(A,B'),Seq(C,D'))`, and plan: `[A, C]`. Finally, let's say we choose to decompose `B'`, followed by `D'`. Planning stops when there are no more tasks to decompose – the leaves of the final network: `Par(Seq(A,B),Seq(C,D))` are all actions. The final plan is: `[A, C, B, D]`, which represents one possible way of enacting the given network. If the heuristic that we apply in selection is based on some objective function, then it may be that the plan is optimal according to this function.

8.2.2 The Theodore HTN-based Planner

We have chosen to implement our own planner rather than using an off-the-shelf planner, such as SHOP, as we wanted to make use of a number of features which are not available in any other OTD HTN-based planner. An example is the notion of a complex operator (which is described below), which greatly improves planning efficiency, and is particularly well suited to planning for Web Services Composition (WSC).

Our planning approach contains a range of other novel features that are useful in a number of domains. They are not described in this thesis. Details may be obtained from the author on request.

We are interested in a highly modularised approach to planning. While this approach is still formative in our work, we have been keen to get a better understanding of the issues involved in realising such an approach. One way of improving our understanding is the development of simple and quick prototypes that fit the modular mould. The first iteration of such an approach is embodied in the Theodore HTN-based Planner.

Features of the planner (non-exhaustively) include the following.

- As well as specifying *operators* and *methods* for a domain, *complex operators* may also be specified. Complex operators offer a combination of operator- and method-based decomposition. They decompose a non-primitive task into a network of *actions*. The use of a complex operator thus side-steps the need to refine a task by a number of method applications, followed by a number of operator applications.

  In the previous example of finding a plan for the network `P`, resulting in the network `Par(Seq(A,B),Seq(C,D)`, instead of having separate methods and operators for decomposing the sequences, for instance, we could have used complex operators. That is, we could recast the planning problem as using one method, with two complex operators, say:

  - Method: `P: true: Par(S1, S2)`.
  - Complex Operator: `S1: true: true: Seq(A,B)` – decomposes `S1` into a network consisting of actions `A` and `B`, in sequence, according to precondition: `true` (the first one) and effects: `true`.
  - Complex Operator: `S2: true: true: Seq(C,D)` – decomposes `S2` into a network consisting of actions `C` and `D`, in sequence.

When we decompose a task with a complex operator, the network of actions specified by the construct is inserted, in its entirety, into the plan. This means that the planner for this planning problem has just two possible plans: `[Seq(A,B), Seq(C,D)]` and `[Seq(C,D), Seq(A,B)]`.

Not only does this provision have the potential to improve planning efficiency significantly, it also enables a plan to have an aspect of concurrency in it. OTD HTN-based planners, like SHOP, produce sequential plans. We have identified complex operators to be particularly useful in improving the efficiency of planning for Web Services Composition (WSC) [86, 133]; where, using them, we may plan at the level of the service, rather than at the level of the service operation.

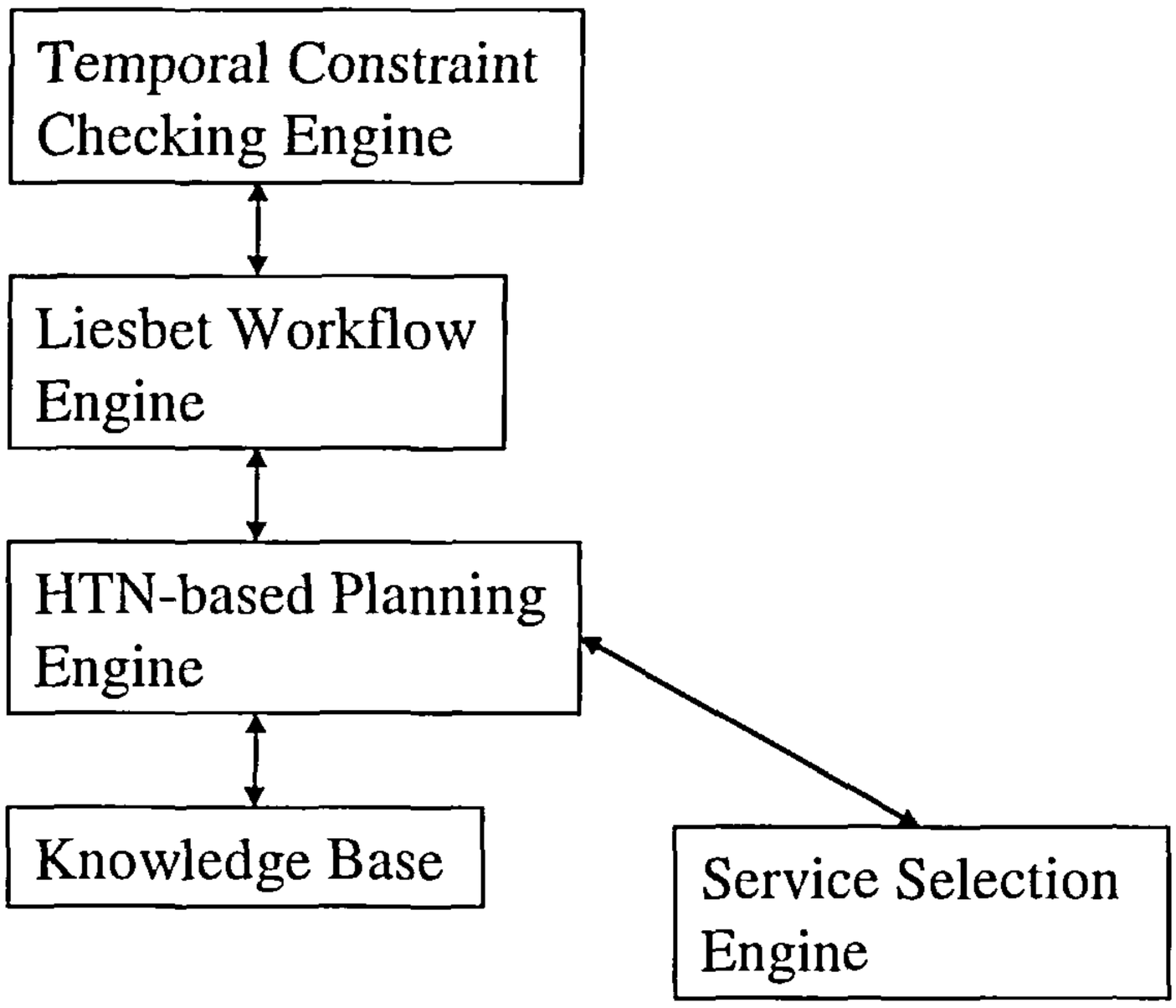


Figure 8.1: Theodore Planning Framework.

- Theodore supports the specification of *temporally-extended* constraints that plans should satisfy. These are constraints whose satisfaction is determined over successive states of enactment of a model. Typically, such constraints would be expressed using a temporal logic, such as CTL* [61, 38], although other constraint languages could be supported. Theodore uses a progression algorithm for constraint verification, which fits with the use of OTD-based planning.

In our work, we use Liesbet for the representation of task networks, although, generally, a fully-blown workflow language is not usually supported (for reasons of planning decidability) by HTN planners. In fact, our planning approach is sufficiently flexible, in principle, to incorporate arbitrary task network representation approaches. We place the burden of guaranteeing decidability on the planning-problem author, in order to obtain a greater flexibility, and additional expressive power, from using a workflow language. To counter the weight of responsibility that this places on a domain author, we may, in time, look at putting measures in place which relieve this burden, at least somewhat. However, we currently feel that the extra flexibility that is gained is worth the pain.

The architecture of the Theodore planner is shown in Figure 8.1. It consists of the following modules. As can be seen, even this very simple planner is highly modularised, allowing for any of the individual engines to be replaced, or sometimes omitted.

- Temporal Constraint Checking Engine – responsible for verifying the integrity of plans against temporal constraints.

- Service Selection Engine – responsible for selecting the next method, complex operator, or operator to apply while executing the planning procedure.

- Knowledge Base – responsible for maintaining the current state of the planning domain, as well as a history of previous states along the current path from the initial state.
- `Liesbet` Workflow Engine – responsible for maintaining the current task network.
- `HTN`-based planning engine – responsible for effecting `HTN`-based planning.

Formalisation of `Theodore`

We now formalise the description of `Theodore`, from the perspective of the `HTN`-based planning engine. In this simplified description, we assume that operators (and complex operators) have deterministic effects.

We will also present another example as we go along. The example is of robots `r1` and `r2`, moving two containers `c1` and `c2`, between two locations `l1` and `l2`. In the example, we have an initial task network consisting of a single task: `transfer-two-containers(c1,c2,l1,l2)`. We have a number of actions available: `move(_r,_la,_lb)` (for moving a robot `_r` from location `_la` to location `_lb`), `load(_c,_r)` (for loading a container `_c` onto a robot `_r`), and `unload(_c,_r)` (for unloading a container `_c` from a robot `_r`). Underscored-prefixed names indicate construct variables, or parameters. We assume the use of a STRIPS- [102] like knowledge base, which consists of a number of ground atoms. Further ground atoms may be inserted, and some removed, as planning takes place.

Definition 1. A Theodore Task t is a pair: (tn, $\mathbb{TP}$) where:

- tn is a name associated with the task.
- $\mathbb{TP}$ is an ordered list of the parameters associated with the task, some of which may be grounded. Those that are not are existentially quantified.

A non-primitive task is one that may only be decomposed by a method or a complex operator. A primitive task may only be decomposed by an operator.

Definition 2. A Theodore Action a is a pair: (an, $\mathbb{AP}$) where:

- an is a name associated with the action.
- $\mathbb{AP}$ is an ordered list of the parameters associated with the action, all of which must be grounded.

Definition 3. A Theodore Task Network n is a triple: ($\mathcal{T}$, $\mathcal{A}$, $\leq_{\mathcal{T}\cup\mathcal{A}}$) where:

- $\mathcal{T}$ is a set of tasks.
- $\mathcal{A}$ is a set of actions.
- $\leq_{\mathcal{T}\cup\mathcal{A}}$ is a partial-ordering over $\mathcal{T}\cup\mathcal{A}$.

Definition 4. A *Theodore Problem Domain tpd* is a triple: ($\mathbb{M}$, $\mathbb{C}$, $\mathbb{O}$), where:

- $\mathbb{M}$ is a set of *Methods.*
- $\mathbb{C}$ is a set of *Complex Operators.*

- $\mathbb{O}$ is a set of *Operators*.

Definition 5. A *Domain Construct*, *dc*, is a base type for methods, complex operators and operators, and is a 5-tuple: (*cn*, *tn*, $\mathbb{TP}$, *p*, $\mathbb{CP}$) where:

- *cn* is a name identifying the domain construct.
- *tn* is the name of the task to which the domain construct is applicable.
- $\mathbb{TP}$ is an ordered list of the parameters associated with the task named *tn* in the construct. *tn* and $\mathbb{TP}$, together, are known as the *head* of the construct.
- *p* is a pre-condition for the application of the domain construct (expressed in the language of the Knowledge Base, which is a module in the `Theodore` planner).
- $\mathbb{CP}$ is an ordered list of the parameters used in the construct, excluding those named in $\mathbb{TP}$.

Definition 6. A *Method m* is a pair: (*dc*, *mn*), where *dc* is a domain construct, and *mn* a network of tasks and actions, to be inserted into the current task network as a result of decomposition.

A method decomposes a non-primitive task into a network consisting of primitive and non-primitive tasks and actions. Note that we are able to distinguish between non-primitive and primitive tasks on the basis that tasks fall into two disjoint sets, namely, those that may be decomposed by operators and those that may be decomposed by methods and complex operators. We determine that a task is primitive (resp. non-primitive) by the existence of an operator (resp. method or complex operator) that decomposes it.

In our example of moving containers, we have a method which decomposes the initial task: `transfer-two-containers(c1,c2,l1,l2)`, viz.

Method: *cn*: transfer two containers
tn: `transfer-two-containers`
$\mathbb{TP}$ `_ca, _cb, _la, _lb`
p `container(_ca)∧container(_cb)∧location(_la)∧location(_lb)`
$\mathbb{CP}$
mn `Par(transfer-one-container(_ca,_la,_lb), transfer-one-container(_cb,_la,_lb))`

The method decomposes the task, `transfer-two-containers`, into a parallel composition of two tasks of transferring one container. There are a number of other methods, as follows.

Method: *cn*: transfer one container
tn: `transfer-one-container`
$\mathbb{TP}$ `_c, _la, _lb`
p `robot(_r)`
$\mathbb{CP}$ `_r`
mn `Seq(load'(_c,_r),move-robot(_r,_la,_lb),unload'(_c,_r))`

This method prescribes how we may decompose the task: `transfer-one-container(_c,_r,_la,_lb)` into a sequence of tasks: `load(_c,_r)`, `move-robot(_r,_la,_lb)` and `unload(_c,_r)`, where `_r` is bound in evaluating the precondition.

Method: *cn*: move _r from _la to _lb
tn: move-robot
$\mathbb{TP}$ _r,_la,_lb
p at(_r,_la)
$\mathbb{CP}$
mn move'(_r,_la,_lb)

This prescribes how we may decompose the task: move-robot(_r,_la,_lb) into a (primitive) task: move(r,la,lb).

Method: *cn*: move _r from _la to _lb, when _r is already at _lb
tn: move-robot
$\mathbb{TP}$ _r,_la,_lb
p at(_r,_lb)
$\mathbb{CP}$
mn

This method handles the possibility that we transfer both containers in the same execution of move, meaning that we need to trivially consume one instance of move-robot in the evolving task network. This would happen if we loaded both containers onto the same robot, prior to moving it.

Definition 7. An *Operator o* is a triple: (*dc*, *e*, *a*), where *dc* is a domain construct, *e* is an effects statement, and *a* is the action associated with the operator.

An operator decomposes a primitive task into a single action. In our example, we offer three operators for decomposing the primitive tasks: move', load' and unload', as follows.

Operator: *cn*: load container
tn: load'
$\mathbb{TP}$ _c, _r
p ¬on(_c,_r)∧at(_r,_l)∧at(_c,_l)
e on(_c,_r)∧¬at(_c,_l)
a load(_c,_r)
$\mathbb{CP}$ _l

Operator: *cn*: move robot
tn: move'
$\mathbb{TP}$ _r, _la, _lb
p true We already check at(_r,_la) in method decomposition
e ¬at(_r,_la)∧at(_r,_lb)
a move(_r,_la,_lb)
$\mathbb{CP}$

Operator:	*cn*:	unload container
	tn:	`unload'`
	$\mathbb{TP}$	`_c, _r`
	p	`on(_c,_r)∧at(_r,_l)`
	e	`¬on(_c,_r)∧at(_c,_l)`
	a	`unload(_c,_r)`
	$\mathbb{CP}$	`_l`

Definition 8. A *Complex Operator co* is a pair: (*o*, *con*), where *o* is an operator, and *con* a network of actions, to be inserted into the current task network as a result of decomposition.

A complex operator decomposes a non-primitive task into a network consisting solely of actions. In our example of moving containers, we could offer an alternative construct, namely, a complex operator, to effect a complete decomposition of `transfer-one-container` in one step, as follows.

For complex operators, the action *a* and the network of actions *con* are necessarily identical in definition. As a consequence, we usually just specify *con* when defining a complex operator, while omitting a specification of *a*.

Complex	*cn*:	transfer just one container
Operator:	*tn*:	`transfer-one-container`
	$\mathbb{TP}$	`_c, _la, _lb`
	p	`robot(_r)∧¬on(_c,_r)∧at(_r,_la)∧at(_c,_la)`
	e	`at(_r,_lb)∧at(_c,_lb)`
	$\mathbb{CP}$	`_r`
	con	`Seq(load(_c,_r),move(_r,_la,_lb),unload(_c,_r))`

Definition 9. A Theodore Planning Problem *tpp* is a triple: (*tpd*, *n*, *kb*) where:

- *tpd* is a Theodore Planning Domain.
- *n* is the initial task network, such as `transfer-two-containers(c1,c2,l1,l2)`, for which we wish to find a plan. The network consists of a number of primitive and non-primitive tasks and actions, constrained by some partial ordering.
- *kb* is the initial knowledge base state, maintained by the Knowledge Base engine.

In the example, the knowledge base *kb*, used for the planning problem, is initialised to assert the following atoms.

```
location(l1) location(l2)
robot(r1) robot(r2)
container(c1) container(c2)
at(c1,l1) at(c2,l1) at(r1,l1) at(r2,l1)
```

Theodore Ordered-Decomposition by Method

A network *n* may be decomposed into a network *n'*, by using a method to refine a non-primitive task in *n*, relative to a knowledge base *kb* and *initial task it*, according to the relation: $met(n, kb, it, t, n')$,

n' represents a method-realised decomposition, according to $met(n,kb,it,t,n')$ given n, t, kb and it, if and only if:

(mi) t is a task in n which has no immediate predecessors which have not been decomposed, i.e. there are no tasks in the network that must be completed prior to t, according to the partial-ordering specified by n.

(mii) t is it, or a descendant thereof, i.e. t is contained with the task network resulting from the decomposition of it.

(miii) m is a method in $\mathbb{M}$ whose task name tn matches that of t.

(miv) The task parameters associated with t and those specified by $\mathbb{TP}$ for m *unify* with a most general unifier μ. Unification is possible if the two lists of task parameters are of the same length, and values in the respective positions unify.

(mv) The precondition $p\mu$ in m holds, with substitution ν, according to kb.

(mvi) The decomposition n' is formed by attaching $mn\mu\nu$, in m, as the only child of t in n.

Figure 8.2: Criteria for Method-Realised Decomposition.

which specifies possible method-realised decompositions of n, on task t, given it and kb. In defining *met* (and *op* and *coop*, whose definitions are presented later), we assume the context of a particular Theodore planning problem, and its associated domain constructs: $\mathbb{M}$, $\mathbb{O}$ and $\mathbb{C}$.

When applying any sort of decomposition (method, operator, complex operator) if a method has previously been applied with no intervening operator or complex operator, we need to ensure that the task t (for which the decomposition step is to be applied) is a descendant of the task t' decomposed by the previous method application. By descendant, we mean contained within the task network that resulted from the decomposition of t'. We have alluded to this before. This satisfies a requirement in HTN planning that whenever a method is applied in decomposition, its pre-condition is satisfied just prior to the first execution of one of its (decomposed) actions. In order to safeguard this, we require that whenever we start a fresh round of method application – meaning an application of a method following an operator, or complex operator – on a task it, say, subsequent applications of methods, complex operators and operators (until we have applied a complex operator, or operator) will be on tasks which are descendants of it.

In Figure 8.2, we present the criteria for method-realised decomposition.

Referring to our example of moving containers, the initial task network for the planning problem is: `transfer-two-containers(c1,c2,l1,l2)`. The only applicable method for decomposing this task (t) is the one with the construct name (cn) "transfer two containers", viz.

Method:	*cn*:	transfer two containers
	tn:	`transfer-two-containers`
	$\mathbb{TP}$	`_ca, _cb, _la, _lb`
	p	`container(_ca)∧container(_cb)∧location(_la)∧location(_lb)`
	$\mathbb{CP}$	
	mn	`Par(transfer-one-container(_ca,_la,_lb), transfer-one-container(_cb,_la,_lb))`

Method cn is applicable on account of its task name tn matching t. In attempting the decompo-

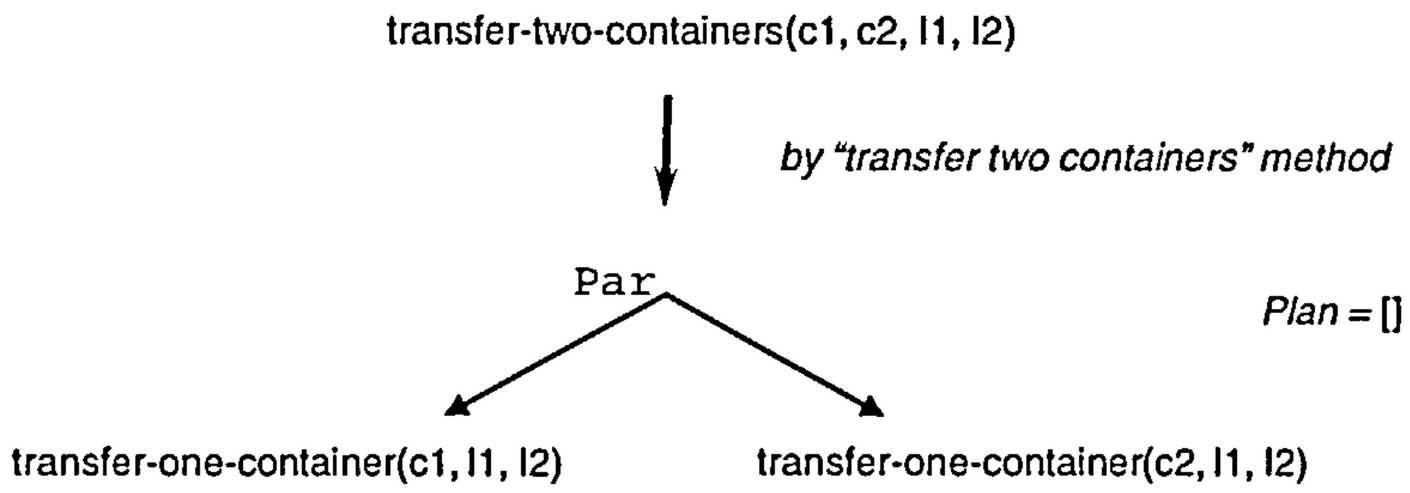


Figure 8.3: First Decomposition Step for `transfer-two-containers` Task.

sition, we try to unify the task parameters ($\mathbb{TP}$), that is, we attempt a member-wise unification of the lists: `[c1,c2,l1,lb]` and `[_ca,_cb,_la,_lb]`, where parameters prefixed with an underscore, _, are variables and unify with any constant, or other variable. The *mgu* μ is {`_ca=c1, _cb=c2, _la=l1, _lb=l2`}. The precondition is appropriately formed by substitution of its (free) variables, according to μ: `container(c1)`$\wedge$`container(c2)`$\wedge$`location(l1)`$\wedge$`location(l2)`. This holds according to the initial state of the *kb*, presented above, where ν is the empty substitution. The decomposition of `transfer-two-containers(c1,c2,l1,l2)` is thus: `Par(transfer-one-container(c1,l1,l2), transfer-one-container(c2,l1,l2))`, attached as a (single) child to the task: `transfer-two-containers(c1,c2,l1,l2)`. From now on, we omit, from the description of an evolving task network, tasks to which a decomposition has already been attached. In Figure 8.3, we present a graphical account of this decomposition.

Theodore Ordered-Decomposition by Complex Operator

A network n may be decomposed into a network n', by using a complex operator to refine a non-primitive task t in n, according to the relation: $coop(n, kb, it, t, n', e', a')$, which specifies possible complex operator-realised decompositions of n, given it and kb.

In contrast to *met*, *coop* (and *op*, see later) takes two additional parameters, namely: e', the applicable effects statement, appropriately substituted, and, a', the action, appropriately substituted, pertaining to the application of the complex operator.

In Figure 8.4, we present the criteria for complex operator-realised decomposition.

In our example, the current task network consisting of: `Par(transfer-one-container(c1,l1,l2), transfer-one-container(c2,l1,l2))` may be decomposed completely by just two further decompositions. That is, we may decompose both of the `transfer-one-container` tasks by applying a complex operator to effect their decomposition. The complex operator is "transfer just one container", and its definition is as follows.

The triple (n',e',a') represents a complex operator-realised decomposition, according to $coop(n,kb,it,t,n',e',a')$ given n, t, kb and it, if and only if:

(coi) As (mi): t is a task in n which has no immediate predecessors which have not been decomposed.

(coii) As (mii): t is it, or a descendant thereof.

(coiii) co is a complex operator in $\mathbb{C}$ whose task name tn matches that of t.

(coiv) As (miv): The task parameters associated with t and those specified by $\mathbb{TP}$ for co unify with a most general unifier μ.

(cov) As (mv): The precondition $p\mu$ in co holds, with substitution ν, according to kb.

(covi) As (mvi): The decomposition n' is formed by attaching $con\mu\nu$, in co, as the only child of t in n. All activities within $con\mu\nu$ are marked as being completed, as is the task t in n'.

(covii) The effects statement e' is $e\mu\nu$, where e is the effects statement specified in co.

(coviii) The action a' is $a\mu\nu$, where a is the action specified in co.

Figure 8.4: Criteria for Complex Operator-Realised Decomposition.

Complex	*cn*:	transfer just one container
Operator:	*tn*:	`transfer-one-container`
	$\mathbb{TP}$	`_c, _la, _lb`
	p	`robot(_r)∧¬on(_c,_r)∧at(_r,_la)∧at(_c,_la)`
	e	`at(_r,_lb)∧at(_c,_lb)`
	$\mathbb{CP}$	`_r`
	con	`Seq(load(_c,_r),move(_r,_la,_lb),unload(_c,_r)`

This domain construct is applicable on account of its task name *tn* matching the name of the task to be decomposed, in each case. For the first task, in attempting a unification of task parameters, the mgu μ is {`_c=c1, _la=l1, _lb=l2`}. The precondition holds, according to the current (still initial) *kb*, with possible substitutions: {`_r=r1`} and {`_r=r2`}. If we pick the first of these, the effects statement is grounded to: `at(r1,l2)∧at(c1,l2)`, meaning that the updated *kb* (regarding at) will be: `at(r1,l2), at(r2,l1), at(c1,l2), at(c2,l1)`. The network of actions: `Seq(load(c1,r1),move(r1,l1,l2),unload(c1,r1))` is inserted, as is, into the (currently empty) plan.

We can then apply another decomposition, using the same construct, to the second `transfer-one-container` task. In attempting a unification of task parameters, the mgu μ is {`_c=c2, _la=l1, _lb=l2`}. The precondition holds, according to the (new) *kb*, with single possible substitution: {`_r=r2`}. The effects statement is grounded to: `at(r2,l2)∧at(c2,l2)`, meaning that the updated *kb* (regarding at) will be: `at(r1,l2), at(r2,l2), at(c1,l2), at(c2,l2)`. The network of actions: `Seq(load(c2,r2),move(r2,l1,l2),unload(c2,r2))` is inserted, as is, into the plan. Planning is now successfully completed as there are no more tasks left to decompose, with the final plan: `[Seq(load(c1,r1),move(r1,l1,l2),unload(c1,r1)), Seq(load(c2,r2),move(r2,l1,l2),unload(c2,r2))]`, as can be seen in Figure 8.5.

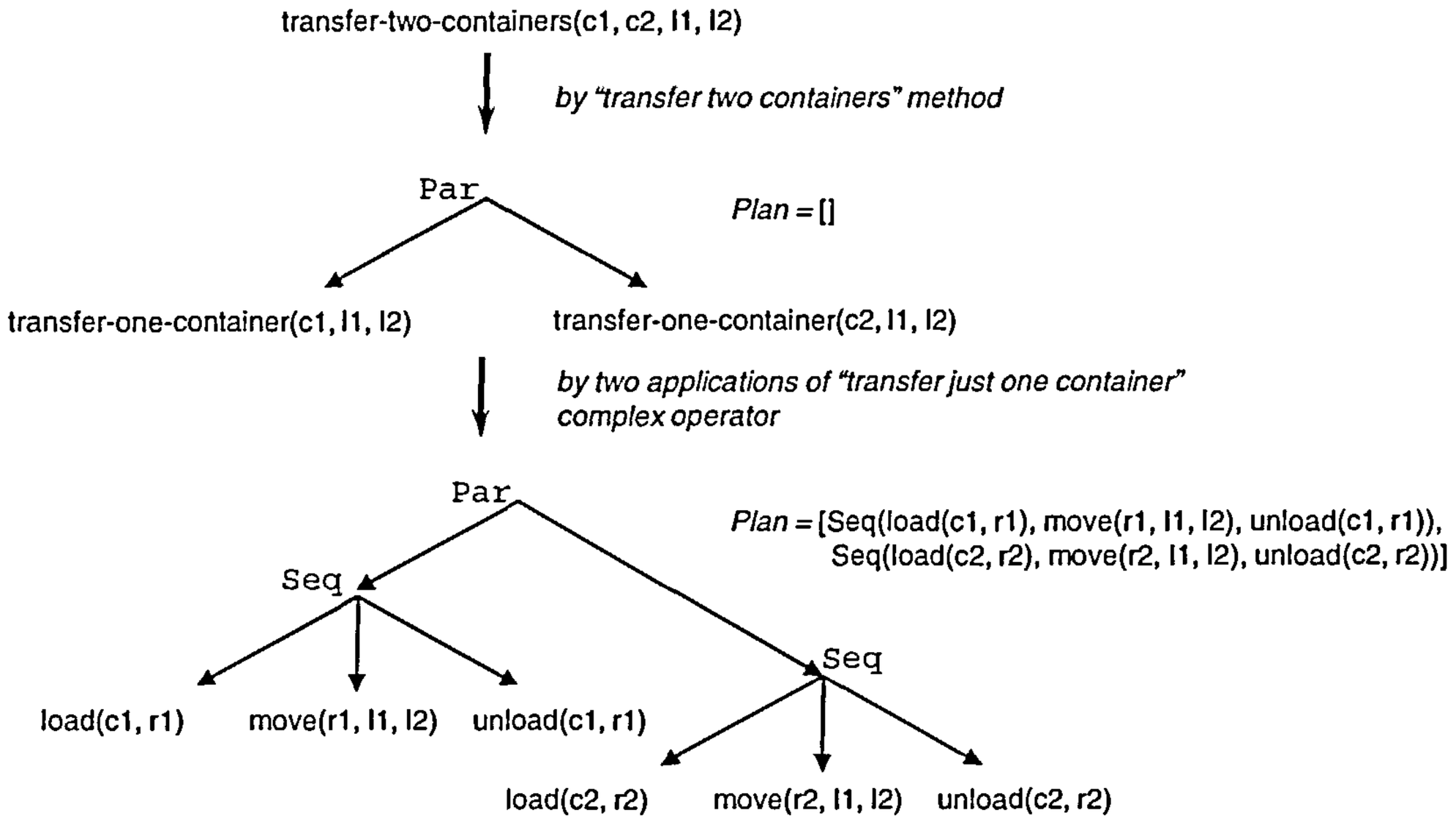


Figure 8.5: Further Decomposition Steps for `transfer-two-containers` Task.

Theodore Ordered-Decomposition by Operator

A network n may be decomposed into a network n', by using an operator to refine a primitive task t in n, according to the relation: $op(n, kb, it, t, n', e', a')$, which specifies possible operator-realised decompositions of n, given it and kb.

In Figure 8.6, we present the criteria for complex operator-realised decomposition.

In our example, the task network after effecting decomposition on the initial task network, using the method for `transfer-two-containers`, is: `Par(transfer-one-container(c1,l1,l2), transfer-one-container(c2,l1,l2))`. If we use the method "transfer one container" to decompose the first of these tasks, a possible resulting task network is: `Par(Seq(load'(c1,r1), move-robot(r1,l1,l2), unload'(c1,r1)), transfer-one-container(c2,l1,l2))`. At this stage of decomposition, we must decompose `load'` next. `load'` is a primitive task, as it has an operator relating to its decomposition, viz.

Operator:

cn:	load container
tn:	`load'`
$\mathbb{TP}$	`_c, _r`
p	`¬on(_c,_r)∧at(_r,_l)∧at(_c,_l)`
e	`on(_c,_r)∧¬at(_c,_l)`
a	`load(_c,_r)`
$\mathbb{CP}$	`_l`

This domain construct is applicable on account of its task name *tn* matching the name of the task to be decomposed. In attempting a unification of task parameters, the mgu μ is {`_c=c1, _r=r1`}. The grounded precondition, `¬on(c1,r1)∧at(r1,l1)∧at(c1,l1)`, holds, according to the

The triple (n',e',a') represents an operator-realised decomposition, according to $op(n,kb,it,t,n',e',a')$ given n, kb and it, if and only if:

(oi) As (mi): t is a task in n which has no immediate predecessors which have not been decomposed.

(oii) As (mii): t is it, or a descendant thereof.

(oiii) o is an operator in $\mathbb{O}$ whose task name tn matches that of t.

(oiv) As (miv): The task parameters associated with t and those specified by $\mathbb{TP}$ for o unify with a most general unifier μ.

(ov) As (mv): The precondition $p\mu$ in o holds, with substitution ν, according to kb.

(ovi) The decomposition n' is formed by applying the substitution $\mu\nu$ to t in n and changing the classification of t from a task to the action a' (see (oviii)), and marking it as being completed.

(ovii) As (cov): The effects statement e' is $e\mu\nu$, where e is the effects statement specified in o.

(oviii) As (covi): The action a' is $a\mu\nu$, where a is the action specified in o.

Figure 8.6: Criteria for Operator-Realised Decomposition.

current (still initial) kb. The effects statement is grounded to: `on(c1,r1)`$\wedge\neg$`at(c1,l1)`. The action `load(c1,r1)` is inserted, as is, into the (currently empty) plan, as shown in Figure 8.7.

If we next select the (remaining) `transfer-one-container(c2,l1,l2)` task for decomposition by the "transfer one container" method, where we are able to choose the same robot, `r1`, as used for the first of these tasks, and follow that by decomposing `load'(c2,r1)`, as above, then the resulting plan will be: `[load(c1,r1), load(c2,r1)]`, and resulting task network: `Par(Seq(load(c1,r1), move-robot(r1,l1,l2), unload'(c1,r1)), Seq(load(c2,r1), move-robot(r1,l1,l2), unload'(c2,r1))`. If we then move the robot, by using the method "move _r from _la to _lb'" to decompose the first `move-robot(r1,l1,l2)` task, and use the operator "move robot" to decompose the resulting `move(r1,l1,l2)` task, the resulting plan is: `[load(c1,r1), load(c2,r1), move(r1,l1,l2)]` and network: `Par(Seq(load(c1,r1), move(r1,l1,l2), unload'(c1,r1)), Seq(load(c2,r1), move-robot(r1,l1,l2), unload'(c2,r1))`, as shown in Figure 8.8.

At this point, we may, for instance, unload the first container `c1` by decomposing `unload'(c1,r1)` to the action `unload(c1,r1)`, resulting in current plan: `[load(c1,r1), load(c2,r1), move(r1,l1,l2), unload(c1,r1)]` and task network: `Par(Seq(load(c1,r1), move(r1,l1,l2), unload(c1,r1)), Seq(load(c2,r1), move-robot(r1,l1,l2), unload'(c2,r1))`. Then, all we can do is decompose the task `move-robot(r1,l1,l2)`. But, as the robot is already at `l2`, we can only use the method 'move _r from _la to _lb, when _r is already at _lb' to decompose the task into a empty network. The consequence of effecting this decomposition is to remove the task from the current network, viz: `Par(Seq(load(c1,r1), move(r1,l1,l2), unload(c1,r1)), Seq(load(c2,r1), unload'(c2,r1))`. Finally, we may decompose the final `unload(c2,r1)` task, resulting in a *final* plan: `[load(c1,r1), load(c2,r1), move(r1,l1,l2), unload(c1,r1), unload(c2,r1)]` and task network: `Par(Seq(load(c1,r1), move(r1,l1,l2), unload(c1,r1)), Seq(load(c2,r1), unload(c2,r1))`.

Definition 10. Theodore Plans and Solutions

A plan π for a Theodore Planning Problem is a sequence of actions, with bindings, resulting

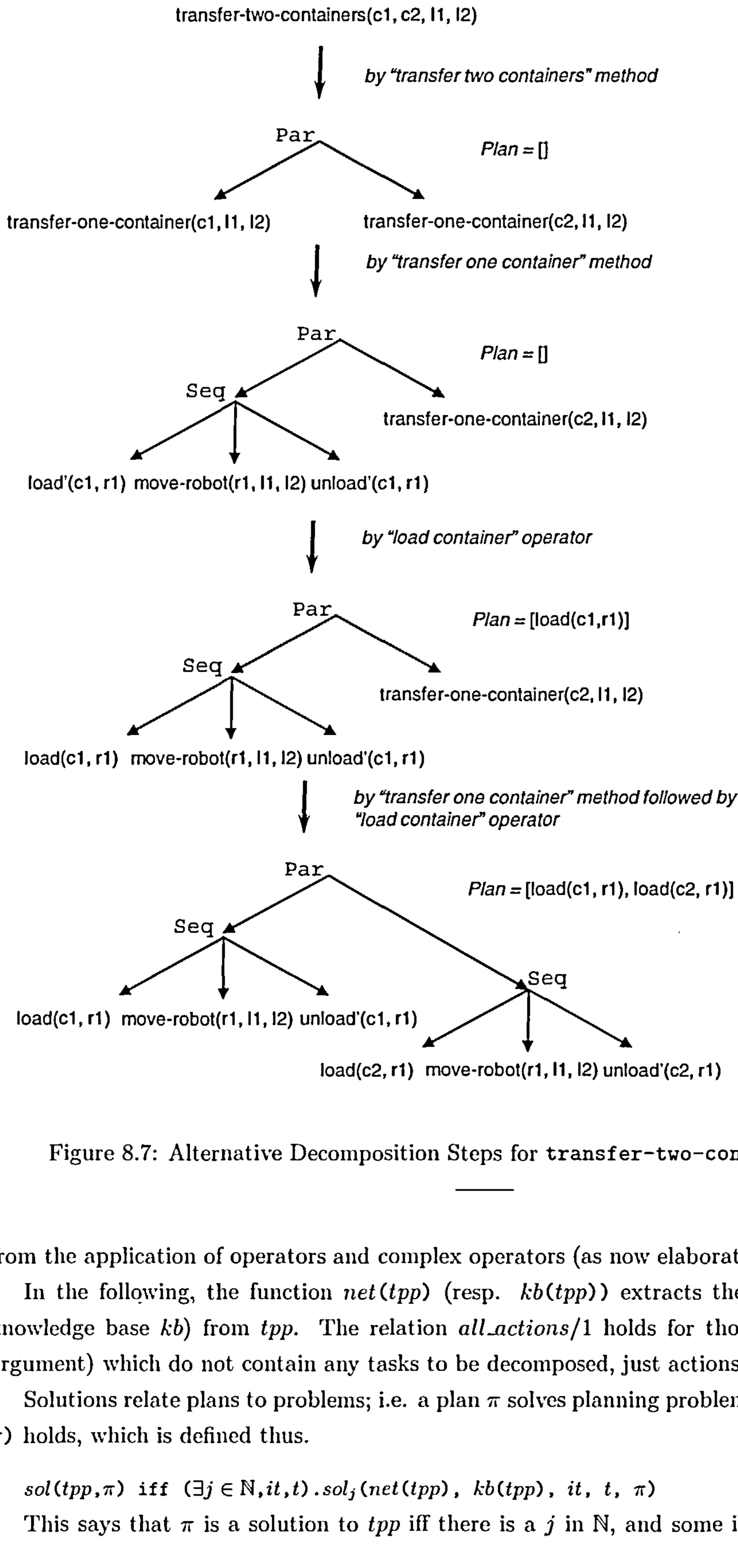


Figure 8.7: Alternative Decomposition Steps for `transfer-two-containers` Task (I).

from the application of operators and complex operators (as now elaborated).

In the following, the function *net*(*tpp*) (resp. *kb*(*tpp*)) extracts the network n (resp. the knowledge base *kb*) from *tpp*. The relation *all_actions*/1 holds for those networks (the single argument) which do not contain any tasks to be decomposed, just actions.

Solutions relate plans to problems; i.e. a plan π solves planning problem *tpp* whenever *sol*(*tpp*, π) holds, which is defined thus.

$sol(tpp,\pi)$ `iff` $(\exists j \in \mathbb{N}, it, t).sol_j(net(tpp),\ kb(tpp),\ it,\ t,\ \pi)$

This says that π is a solution to *tpp* iff there is a j in $\mathbb{N}$, and some initial task *it*, such that

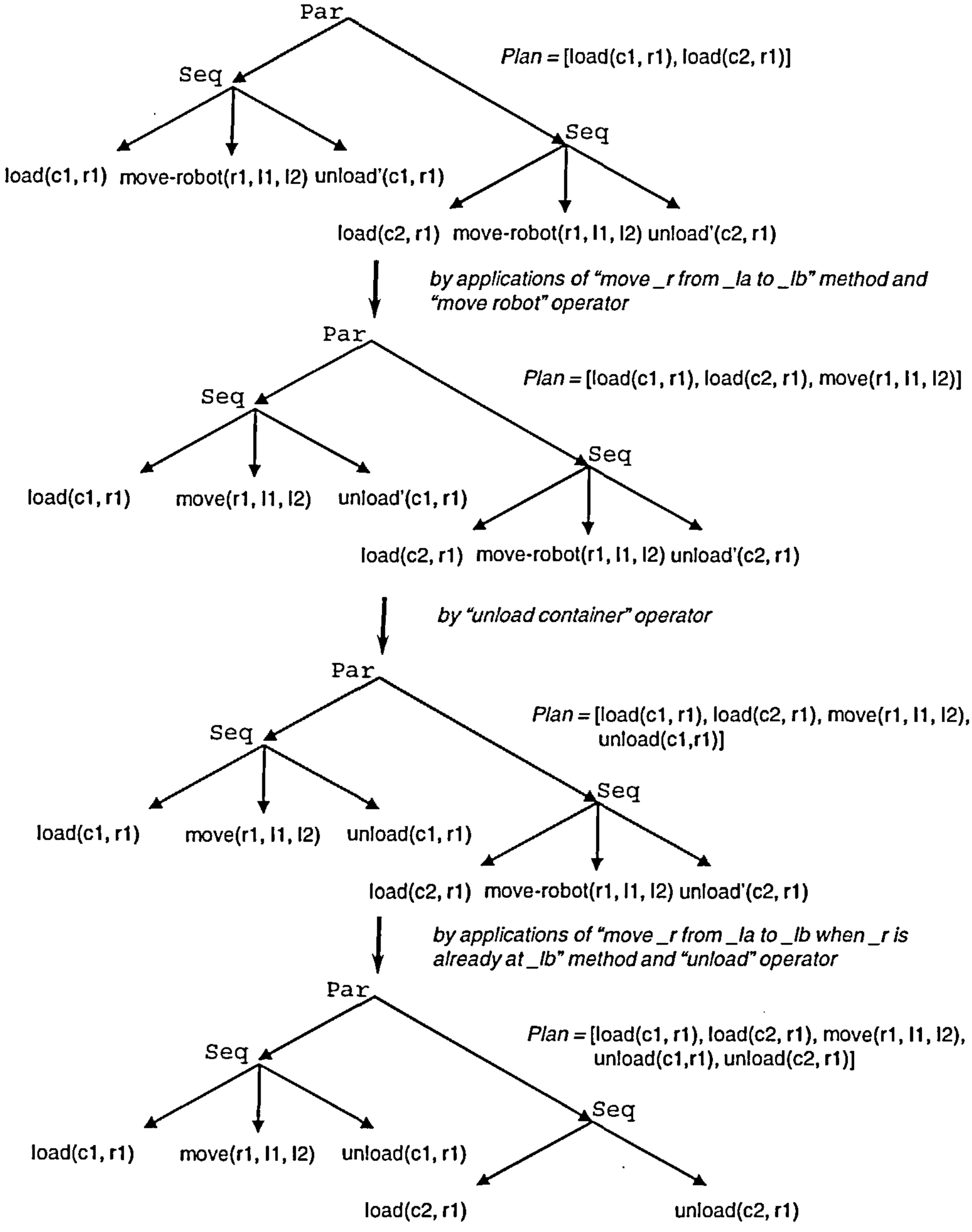


Figure 8.8: Alternative Decomposition Steps for `transfer-two-containers` Task (II).

π is generated after j decompositions, starting with task t. The definition of sol_j is presented in Figure 8.9.

Theodore Planning Algorithm

In Figure 8.10, we present the planning algorithm used by the `Theodore` engine. We assume that some mechanism is provided, by a Service Selection Engine (see previously), for selecting domain

$sol_0(n, kb, it, t, \pi)$ iff π=[] $\wedge$ $all\_actions(n)$
$sol_j(n, kb, it, t, \pi)$ iff
$(\exists n', t').met(n, kb, it, t, n') \wedge sol_{j-1}(n', kb, it, t', \pi) \vee$
$(\exists n', e', a').(coop(n, kb, it, t, n', e', a') \vee op(n, kb, it, t, n', e', a')) \wedge$
$(\exists kb', it', t', \pi').sol_j(n', kb', it', t', \pi') \wedge kb'=apply(kb, e') \wedge \pi=[a' | \pi']$

Figure 8.9: Definition of $sol_j(n, kb, it, t, \pi)$

constructs and tasks to decompose first. A requirement of such an engine is that it will (through backtracking) eventually cover all possible permutations of construct and task selection. It simply effects a preference service on constructs and tasks – selecting preferred ones first. As it does eventually return all possible selections, it does not affect the completeness result presented below. Any selection made using a Service Selection Engine is indicated, in the presentation below, by annotating the word 'Select' with asterisks, thus: '*Select*'. We also assume that the kb attached to Theodore is sound, as well as being practicably decidable in its inference procedures.

Theodore Planning Soundness and Completeness

To show soundness and completeness of the planning algorithm, we show that $sol(tpp, \pi)$ iff $sol_{THEO}(tpp, \pi)$, where $sol_{THEO}(tpp, \pi)$ holds when the plan π is generated by the Theodore planning algorithm for planning problem tpp. $sol_{THEO}(tpp, \pi)$ holds iff there is a j in $\mathbb{N}$, such that π is generated by Theodore, in j steps, given tpp. That is,

$sol_{THEO}(tpp, \pi)$ iff $(\exists j \in \mathbb{N}, t).sol_{THEO_j}(net(tpp), kb(tpp), t, \pi)$

We define the relation $sol_{THEO_j}(n, kb, t, \pi)$ to hold iff:

- For j=0, n is a fully-decomposed network, and π is the empty plan, [], where kb and t may each be any arbitrary value.

- For j >0, t is the task being decomposed, and $sol_{THEO_{j-1}}(n', kb', t', \pi')$ holds, where n' is the result of the decomposition performed in step j (which *precedes* step j−1), according to kb, t' is the task decomposed in step $j-1$, *and*:

  - if it is a complex operator/operator decomposition then t' may be chosen irrespective of the choice of t, kb' represents the modification of kb to account for the effects of the decomposition, and π' is the tail of π, whose head is the action prescribed by the complex operator/operator;

  - if it is a method decomposition then t' must be a descendant of t for j >1 and kb=kb' and π=π'.

The definition of this relation can be seen to characterise the behaviour of the Theodore planning algorithm, except with regard to how the relation is built up. The difference stems from the need to characterise the solutions to a planning problem (which the relation sol_j defines) from the bottom-up (that is, starting with a fully-decomposed network, and then "folding" decompositions into the network) versus describing a planning algorithm which works top-down (that is, starting with a partially-decomposed network, and decomposing it until a fully-decomposed network is reached). In order to show soundness and completeness of the planning algorithm, we need to

procedure find_plan(n,kb,tpd)
it:= null
π:= []
LOOP

Select an eligible task t in n. An eligible task is one with no predecessors which are yet to be decomposed, and which is a descendant of *it* if *it* is not null.

If no eligible task then:

- If tasks still exist in n, then backtrack to last choice point as partial plan is not valid.
  If further backtracking is not possible then FAIL PLANNING.
- If no tasks exist, then return π as plan.

If t is a primitive task then

- *Select* an *appropriate* operator o in *tpd* (i.e. whose parameterised task name unifies with that of t, and whose pre-condition is satisfied, (see step (ov), above)).
  If a selection is not possible then backtrack... (to last choice point)
- Mark task t as an action (a', from step (oviii)) in n, and set it to be completed
- Apply the effects of the operator (e', from step (ovii)) to *kb* yielding a new *kb*
- Concatenate the action of the operator (a', from step (oviii)) to the end of π yielding a new π
- Reset *it* to null

Else (t is not primitive)

- *Select* an *appropriate* complex operator, or method, in *tpd*.
  If a selection is not possible then backtrack...
- If a complex operator *co* is selected then
  - Attach the (customised – by substitution, see step (covi)) network $con\mu\nu$, in *co*, as a child of t in n.
  - Apply the effects of the complex operator (e', from step (covii)) to *kb* yielding a new *kb*
  - Concatenate the action of the complex operator (a', from step (coviii)) to the end of π yielding a new π
  - Reset *it* to null

  Else If a method m is selected then
  - Attach the (customised – by substitution, see step (mvi)) network $mn\mu\nu$, in m, as a child of t in n.
  - Set *it* to be t

Figure 8.10: Theodore Planning Algorithm.

characterise the algorithm (in defining the sol_{THEO_j} relation) in a similar bottom-up fashion, in order that we may be able to argue a correspondence between the two relations, sol_{THEO_j} and sol_j.

For *soundness*, we need to show that if $sol_{THEO_j}(n,\ kb,\ t,\ \pi)$ holds, with π generated in j steps, then $sol_j(n,\ kb,\ it,\ t,\ \pi)$ holds, for some *it*. For *completeness*, we show that if $sol_j(n,$

kb, it, t, π) holds for some it, then $sol_{THEO_j}(n,\ kb,\ t,\ \pi)$ holds.

For both results, the *base step* (j=0) is straightforward: $sol_{THEO_0}(n,\ kb,\ t,\ [])$ trivially implies $sol_0(n,\ kb,\ it,\ t,\ [])$, for any it, for a fully-decomposed network n; and, similarly, $sol_0(n,\ kb,\ it,\ t,\ [])$ trivially implies $sol_{THEO_0}(n,\ kb,\ t,\ [])$, for any it.

For the *induction step* (j=k+1), the induction hypothesis gives us that: $sol_{THEO_k}(n',\ kb',\ t',\ \pi')$ implies $sol_k(n',\ kb',\ it',\ t',\ \pi')$, for some it'; and, similarly, $sol_k(n',\ kb',\ it',\ t',\ \pi')$ implies $sol_{THEO_k}(n',\ kb',\ t',\ \pi')$, for some it'.

The Theodore algorithm, in step j (the first iteration), constructs kb' from kb, n' from n and π' from π in the same way that the relation sol_j prescribes:

- For operator application: n' is obtained from n by marking t as an action (as specified in Figure 8.10), kb' is obtained from kb by applying the (customised) effects e' of o, and, π' is the tail of π with a' as its head. These constructions mirror those prescribed in Figure 8.6, and in the definition of sol_j, presented in Figure 8.9.

- For complex-operator application, the same applies, except that the network is derived as specified in Figure 8.10, by attaching the task network specified by co, appropriately customised by substitution, as a child to t, which is mirrored in the definition of sol_j.

- For method-realised decomposition, the same applies again, except that kb is not modified.

Moreover, pre-condition evaluation, for the applicability of constructs, is handled the same in both the Theodore planning algorithm, and in the definitions of *met*, *coop* and *op* (in Figures 8.2, 8.4 and 8.6), which are used as a basis for the definition of sol_j.

The only complication that arises in proving both soundness and completeness results comes down to the role of the parameter it. We construct the proofs for soundness and completeness on the basis of what role this parameter plays in the definitions of sol_{THEO_j} and sol_j.

Theorem 1. Soundness: $sol_{THEO_j}(n,\ kb,\ t,\ \pi)$ implies $sol_j(n,\ kb,\ it,\ t,\ \pi)$, for some it.

Proof.

- For Method Decomposition: From $sol_k(n',\ kb,\ it',\ t',\ \pi)$, for step k, we may trivially derive $sol_j(n',\ kb,\ it,\ t,\ \pi)$ according to the definition of sol_j, where it' (for step k) is bound to t (the task decomposed in step j).

- Complex Operator/Operator: Follows similarly, except that kb and π differ between steps, and the value of it' is arbitrary.

□

Theorem 2. Completeness: $sol_j(n,\ kb,\ it,\ t,\ \pi)$ implies $sol_{THEO_j}(n,\ kb,\ t,\ \pi)$ for some it.

Proof.

- For Method decomposition:

  - For the simple case where j=1, from the base step, $sol_0(n',\ kb,\ it',\ t',\ [])$ implies $sol_{THEO_0}(n',\ kb,\ t',\ [])$ trivially holds for any it'. As j =1, for method decomposition, π =[]. From $sol_{THEO_0}(n',\ kb,\ t',\ [])$, the propositions $sol_{THEO_1}(n,\ kb,\ t,\ [])$ and $sol_{THEO_1}(n,\ kb,\ t,\ \pi)$ trivially obtain.

- For cases where $j > 1$, we observe that for step k, $sol_k(n',\ kb,\ it',\ t',\ \pi)$ implies $sol_{THEO_k}(n',\ kb,\ t',\ \pi)$, by the induction hypothesis, for some it', where t' is the task decomposed in step k. According to the definition of sol_j, it' must be the task t decomposed in step j. Thus, the task t', decomposed in step k, must be a descendant of t. If $sol_{THEO_k}(n',\ kb,\ t',\ \pi)$, then $sol_{THEO_j}(n',\ kb,\ t,\ \pi)$ holds as t' is a descendant of t, as established.

- Complex Operator/Operator: $sol_{THEO_j}(n,\ kb,\ t,\ \pi)$ trivially follows from $sol_{THEO_k}(n',\ kb',\ t',\ \pi')$.

□

8.3 Verification of and Planning over Flexible Workflow Models with Theodore

We specify flexible workflow models as Theodore planning problems. That is, we start with an abstract workflow and refine it into a concrete one using a number of decomposition relations[2] (i.e., methods, operators and complex operators), specified in the problem description.

The Theodore HTN-based planner may be used in two capacities for this purpose. The principal distinction between these capacities stems from the issue of whether the set of decomposition relations is fixed for the enactment lifetime of the pertaining model, or whether the set may be dynamically switched. That is, we make a distinction between *fixed* and *variable* flexible models, respectively.

An example of the context in which the set of decomposition relations may change in the course of enactment is the changing of operational policies used in an enterprise, which may be reflected (in part) by the set of available decomposition relations, as business objectives of the enterprise change.

For fixed models, we use the Theodore planner to verify that their definition (as a Theodore problem) is *sound.* A Theodore *fixed* flexible workflow model is defined to be sound iff every partial decomposition of a workflow leads to a full decomposition. This is called the *verification criterion.* If not, it will be possible for the workflow to reach a deadlocked state, where it is not possible to perform further decomposition based on the available decomposition relations. If a workflow model, specified as a Theodore planning problem, passes this criterion then it is guaranteed that its execution will complete properly.

The normal operation of Theodore, for example in the context of planning for Web Services Composition (WSC), is to search for a plan which effects the composition (and optionally satisfies some temporal constraints). When using Theodore for verification of flexible workflow models, we thus need to be a lot more thorough by checking all possible decompositions.

During verification, we may also check that certain constraints, expressed in some language or logic, are satisfied. This capability makes use of the approach that was presented in Chapter Seven. As documented there, we may use any language for which progression semantics can be specified. In our work, we have chosen the temporal logic CTL* for describing constraints.

[2]In the following text, the term *decomposition relation* is preferred over *domain construct* as a matter of taste. They should be treated as synonyms.

As an example, consider the task of verifying the simple planning problem (presented earlier) which results in the workflow: `Par(Seq(A,B),Seq(C,D))`. The output of verifying the flexible workflow, specified as a `Theodore` planning problem, using the `Theodore` engine is presented in Figure 8.11. As can be seen, verification succeeds with the input problem (model plus rules) declared as being sound. The temporal constraint that is checked confirms workflow soundness – we test the `CTL*` proposition: `AF Completed_act(P1)`, which expresses that the initial task network completes in every possible enactment path.

In Figure 8.11, every '.' after the word 'Planning' indicates where the planner has tried an alternative path in verification, and each 'c' indicates a constraint checking step. In the verification output, each action is prefixed by an action history. If an action is the first in a plan, it will be prefixed by a single number which indicates an index for the action in the collection of actions possible at this stage of the plan. In this output, there are two first actions, either `A` (the `0`th action, in the collection of first actions, as indicated by `0:A`), or `C` (the `1`st action, in the collection of first actions, as indicated by `1:C`). For `0:A`, there are two possible plan continuations `0:0:C` and `0:1:B`, the `0`th and `1`st actions, respectively, in the collection of actions that may follow `0:A`. The first of these may be extended by `0:0:0:B` or `0:0:1:D`, and so on. `END OF PATH` is a delimiter, and indicates that another plan has been found. As each partial plan must lead to a full plan, each prefixed action, output by the planner, must extend the previous one (unless it follows an `END OF PATH` delimiter), and the output must end with an `END OF PATH` delimiter.

We also draw the reader's attention to the '`TransferProperty`' examples that we give in Sections 9.4.5, 11.2.2 and 11.2.3, which further explicate our flexible modelling approach.

For *variable* flexible models, we need to work within the confines of the current set of decomposition relations whilst remaining aware that this set may be subject to change at any time. In this context, `Theodore` may be used to attempt to find a plan to effect an abstract workflow (which may have already been partly enacted) using the current set of decomposition relations. A domain controller or expert may be on hand to guide this planning procedure, so that the plan meets their subjective constraints. In this sense, the planner behaves as a "what may I do next" query-interpreter. Such a person may also perform "what-if" simulation, in order to understand what actions are available to them, and what the consequences of these actions are. Moreover, planning may be interleaved with enactment, so that enactment results may feed back into the planning process. Enactment will only fail if we are unable to complete the workflow given the current set of relations. In this scenario, the planner would look at alternative sets of decomposition relations, notwithstanding that they may be less favourable at that particular time. The idea of variable models is to allow a greater level of flexibility in modelling at the cost of statically ensuring workflow soundness.

For fixed flexible models, we also support the notions of a "what may I do next" query-interpreter and "what-if" simulation.

A key aspect of our approach to flexible workflow modelling, through both fixed and variable models, is its support for *collaborative* workflows. An example of the application of this kind of workflow (for crisis mitigation) has already been described in Section 8.1.3. In the enactment of collaborative workflows, parties decide on the best way to achieve the goals (prescribed by an abstract workflow specification), constrained only by the availability of decomposition relations for tasks in the workflow.

```
Simple workflow domain
Initialising planner...
Planning.cccc.cc.ccc.cccc.cc.ccc.
Workflow/Contract is SOUND with no constraint violations.
Planning details...
Time taken: 0(h),0(m),1(s),750(ms)
**************************************
0:A
0:0:C
0:0:0:B
0:0:0:0:D
END OF PATH
0:0:1:D
0:0:1:0:B
END OF PATH
0:1:B
0:1:0:C
0:1:0:0:D
END OF PATH
1:C
1:0:A
1:0:0:D
1:0:0:0:B
END OF PATH
1:0:1:B
1:0:1:0:D
END OF PATH
1:1:D
1:1:0:A
1:1:0:0:B
END OF PATH
*******************************
```

Figure 8.11: Output from Verifying a Theodore Planning Problem whose initial task network decomposes to the simple workflow model: `Par(Seq(A,B),Seq(B,C))`.

In our work, roles may be assigned to decomposition relations and tasks. When a decomposition relation refines a task, the role associated with the relation must be compatible with that described for the task, according to some model of roles. We briefly described role modelling in Section 8.1.6. We shall not elaborate any further here, for reasons of brevity.

8.4 Concluding Remarks

It is instructive to consider how our approach to flexible workflow modelling compares with the other similar approaches that we reviewed at the start of this chapter.

- In CrossFlow [55] some limited flexibility is allowed in an otherwise rigid process. The notion of a workflow containing multiple possible enactments to be constrained by organisational policies does exist. However, the nature of flexibility is limited in that the workflow is fully specified, and as a consequence the notion of collaborative workflows, which are completed through collaboration of the participating agents, is not supported.

  In fact, the nature of the flexibility is (arguably) more in keeping with that supported by Synchronisation Rules (see Section 3.3) in our work, for which we consider slogan *Flexible Workflow = Concrete Model + Policies for Constraint* to be an appropriate synopsis.

- In Wainer [131], anything that is compatible with a set of (temporal logic) domain axioms may be done. The notion of an explicit abstract workflow to provide at least some structure is absent. Such a workflow is typically beneficial as it can greatly aid the efficiency of verification. In Wainer and colleagues [132], any activity may be executed as long as its pre- and post- conditions regarding the execution of other acts are observed. There is a partial ordering implied by these pre-conditions, but no explicit workflow. Moreover, the actions are fully refined, meaning that there is an absence of control over which combinations of actions are allowable (as we have if we model tasks hierarchically), unless we encode these combinations implicitly within the activity dependencies. However, this would be quite impracticable for all but the simplest of domains.

  Both our work and the works of Wainer and colleagues support the notions of "what may I do next?" querying (as opposed to "what should I do next" in traditional workflow) and "what-if simulation".

- In Case Handling (CH) [127, 16, 96], model flexibility exists in how data objects may be completed, according to a constraining workflow. This is the diametric opposite to our notion of flexibility, where flexibility exists primarily in the control flow. In CH, a workflow is considered complete when its data is completed, not its tasks. In some contexts, control is the best driver, sometimes data. An example of the former is in collaborative workflows, an example of the latter is in form handling, where agent/s are required to complete forms to process a customer request.

- In the Collaboration Management Infrastructure (CMI) [51, 107], resolution rules are applied by domain experts to activity placeholders, which are quite similar to our use of decomposition relations to refine tasks. It is notable, however, that CMI provides no verification

support. We provide verification of workflow soundness for fixed models, and verification of arbitrary temporal constraints.

A key theme in our work in flexible workflow modelling is the notion that we combine structure with flexibility. That is, we start with an abstract workflow model which provides some initial structure. Furthermore, there is structure inherent within the policies for refinement, i.e., the decomposition relations – methods, operators and complex operators, in terms of them prescribing networks of actions which are acceptable refinements of tasks being decomposed. Moreover, complex operators prescribe structure from the bottom-up, in specifying complete refinements of tasks.

All of these dispensations, with respect to structuring, help reduce the complexity of verification. There is a trade-off here between flexibility in workflow specification, and complexity of verification. When we allow greater flexibility, the complexity will soar; but, as we allow less freedom, the complexity will drop. In the extreme of the latter case, we will have fully prescribed workflow models whose verification complexity will be that of `Liesbet` models.

In summary, we have proposed an approach to flexible workflow modelling, which is desirable to counter the significant issue of brittleness in traditional models of workflow. In doing so, we have been able to accommodate collaborative workflows, which are an important kind of workflow (as described in Section 1.1) where agents decide collectively how a workflow instance should be realised.

Our approach is based on the identification of a correspondence between what we seek to achieve in flexible workflow modelling, as epitomised by the slogan: *Flexible Workflow = Abstract Model + Policies for Refinement*, and the operation of an `HTN`-based planner. In identifying such a correspondence, we are able to propose a novel approach using `HTN`-based planning for the description, verification and planned enactment of flexible workflow models.

The expressivity of the planning language for describing domains is limited only be the expressivity of the knowledge base underwriting the problem description, together with the expressivity of the language used in pre-conditions and effects axioms, and the expressivity of the workflow language (such as `Liesbet`) that is used for the specification of abstract workflows. As our planner is modular, all of these provisions can easily be changed, and, thus, in principle, our approach does not limit workflow authors in what they would seek to express. This is a double-edged sword, however, with respect to decidability of an authored problem, and, as a consequence, some care must be take during the process of describing problems to ensure that decidability is maintained. This is perhaps a less than ideal consequence of making our planner wholly flexible. As already stated, we may at some time look at some constraints on what is allowed to be expressed, as other planners such as SHOP [85] do. We are minded, however, to prioritise flexibility at the possible detriment of usability for the time being.

Verification of *fixed* flexible workflow models, under the assumption that their planning domains are decidable, for soundness and for the satisfaction of arbitrary temporal constraints, is also a particularly desirable aspect and novel in the context of flexible workflow modelling.

At this point, an obvious question is how might we apply the work that we have done on both traditional and flexible workflow modelling in other contexts. A natural application is that of contract modelling, where contracts are often cast as protocols (i.e., workflows) of behaviour between two or more parties. In the next chapter, we explore the application of workflow modelling

to the modelling of contracts.

Chapter 9

Institutional Modelling for the Modelling of Contracts

We have been motivated to consider how the work presented in previous chapters may be reused in other contexts. This is an important issue in itself, as part of the utility of research comes from considering how it may be applied in different contexts. For our work, a somewhat evident application is that of contract modelling, where contracts are often cast as protocols (i.e., workflows) of behaviour between two or more parties. We have been motivated to look at the issue of contract modelling for its own sake as well, as this remains a somewhat formative research field in which there is ample scope to make a worthwhile contribution.

An aspect of contract modelling that is quite clear from existing research contributions is that approaches typically admit just a protocol-like view of contracts, or one where a contract is considered to be an aggregation of propositions capturing various concepts, such as obligation and permission. We believe that a hybrid approach, based on the two, is particularly useful for contract modelling.

In order to consider how our work on the modelling of workflow may be reused, it is instructive to consider workflow from new perspectives, other than just control, data and organisational ones, for example. One additional perspective that we have identified is, what we call, an *institutional perspective.* It is possible that there are other perspectives, but we have considered just this additional one for now.

Considering workflow from an institutional perspective entails identifying institutional concepts in workflow. This is particularly desirable as these have a strong overlap with concepts in the field of normative modelling. In turn, normative modelling is a good fit for the modelling of contracts.

We use the term 'normative modelling' to mean the modelling of communities, societies, and other kinds of collectives based on the identification of positions pertaining to *norms* that may hold between agents which operate or exist therein.

We start this chapter with an overview of *institutional modelling.* We then describe how we may consider workflow from an institutional perspective, to which we give the name *Institutional Workflow Modelling* (IWM). Following that, we give an overview of normative modelling (NM), and then proceed with a description of our approach to contract modelling (CM), which is based on both IWM and NM. We consider that IWM provides an invaluable foundational basis for NM,

and as a consequence CM.

9.1 Institutional Modelling for Workflow

We present details of an *institutional perspective* of workflow, that we have identified, and elaborate how our approaches to traditional and flexible workflow modelling may be viewed from this perspective. We use the term *Institutional Workflow Modelling* (IWM) for the modelling of workflow from an institutional perspective.

In the following section, we give an overview of *Institutional Modelling*, and then proceed with a description of IWM.

9.1.1 The Essence of Institutional Modelling

The essence of institutional modelling is that certain worldly facts, or actions, only manifest their stated significance according to an institutional context. That is, it is according to the context of a particular institution that these facts come into being. Searle makes a distinction between *institutional* and *brute* facts, to convey this point.

From [109]:

> Institutional facts are so called because they require human institutions for their existence. In order that this piece of paper should be a five dollar bill, for example, there has to be the human institution of money. Brute facts [such as the sun being ninety-three million miles from the earth] require no human institutions for their existence.

In order to explicate an institutional sense, Searle makes a distinction between two different kinds of rules, viz. *regulative* and *constitutive*. The former kind are concerned with regulating antecedently-existing forms of behaviour. For example, the rule "drive on the right-hand side of the road" regulates driving, but driving can exist prior to the existence of that rule [109]. Searle continues in [108]: "[s]ome rules on the other hand do not merely regulate but create or define new forms of behavio[u]r." These are the so-called constitutive rules, which prescribe what forms of behaviour, or facts, are constituted by the occurrence, or existence, of other forms of behaviour, or facts.

From [109]:

> [T]he rules of chess do not regulate an antecedently[-]existing activity. It is not the case that there were a lot of people pushing bits of wood around on boards, and in order to prevent them from bumping into each other all the time and creating traffic jams, we had to regulate the activity. Rather, the rules of chess create the very possibility of playing chess. The rules are *constitutive* of chess in the sense that playing chess is constituted in part by acting in accord with the rules. If you don't follow at least a large subset of the rules, you are not playing chess. The [constitutive] rules come in systems, and the rules individually, or sometimes the system collectively, characteristically have the form: 'X counts as Y' or 'X counts as Y in context [institution] C'. Thus, such and such counts as a checkmate, such and such a move counts as a legal pawn move, and so on.

Searle concludes that institutional facts exist only within systems of constitutive rules. "The systems of rules create the possibility of facts of this type, and specific instances of institutional facts such as the fact that I won at chess ... are created by the application of specific rules, rules for checkmate ..., for example" [109].

It would appear that Goldman's rules for *conventional generation* carry a similar sense to Searle's constitutive rules. From [54], "[c]onventional generation is characterized by the existence of rules, conventions, or social practices in virtue of which an act A' can be ascribed to an agent S, given his performance of another act, A."

It would appear sensible to make a distinction between brute and institutional actions, as much as between facts. It may be superficial to do so, given that the performance of action may be seen to establish a fact regarding its performance; but it is particularly useful in our work to maintain this distinction in its own right.

9.1.2 Institutional Workflow Modelling (IWM)

Principally, we assert that the ubiquitous hierarchy of a workflow model necessarily entails the manifestation of constitutive rules. That is, in a sequence, `Seq(A,B)`, carrying out actions `A` and `B` *counts as* carrying out the sequence. This is more than mere classification, which may, for example, prescribe subsumption, or so-called *isa* hierarchies, for classes of brute, or institutional, actions. For instance, filling out a form may count as processing a customer's application (constitutive); and, at the same time, may be a clerical task (in a classificative sense, i.e. filling out a form is a clerical task).

Furthermore, (typically) in workflow, the performance of tasks, i.e. basic activities, may only occur subject to an agent being *permitted* to do so, as exemplified in [19, 20], for instance. The fact that an agent `A` is permitted to carry out a task `T` may be considered to be an instance of a regulative rule, according to Searle's distinction.

In defining what we mean by *Institutional Workflow Modelling* (IWM), the two concepts of *counts as* and *permission* play a significant role. In the workflow context, *counts as* is (appropriately) transitive; viz. if performance of `A` *counts as* performance of `B`, and performance of `B` *counts as* performance of `C`, then performance of `A` *counts as* performance of `C`. In a workflow model, the basic activities would correspond to brute tasks, or actions, whereas the performance of a number of these may count as the performance of one or more institutional actions. Aggregating the performance of these institutional actions, in turn, may count as the performance of yet further distinct institutional actions. Ultimately, however, the performance of an institutional action at any level in the portrayed action hierarchy can be traced down to the performance of a number of brute actions, which exist at the leaves of the hierarchy.

Relating the IWM concepts of *counts as* and *permission* to our work on flexible workflow modelling, we note the following correspondences:

- An `HTN` Method may be seen as an embodiment of *counts as*. That is, their purpose is strongly similar to the purpose of *counts as*. *Counts as* may be considered to prescribe how institutional actions may be decomposed into some partial ordering of one or more institutional and brute actions. This notion is mirrored by `HTN` methods whose purpose is to decompose non-primitive `HTN` tasks into some partial ordering of one or more non-primitive and primitive tasks.

- An HTN Operator may be seen as an embodiment of *permission*. *Permission* may be considered to prescribe agents who are permitted to carry out brute actions. This notion is mirrored by HTN operators whose purpose is to prescribe how primitive tasks may be refined into actual actions to be carried out by (specific) agents.

- An HTN Complex Operator may be seen as an embodiment of both *counts as* and *permission* relations, as it serves to effect multiple method- and operator-based decompositions.

We define Institutional Workflow Modelling to be the sum of our Theodore-based approach to flexible workflow modelling and the presented correspondences of *counts as* and *permission* relations to workflow artefacts, on the one hand, and HTN-based planning constructs (i.e., methods, operators and complex operators), on the other.

As we have described, in institutional modelling, generally, notions and artefacts are given meaning according to a context, i.e. according to a pertaining institution. For IWM, the institution would be the particular instance of the workflow being enacted. More specifically, the institution is characterised by the extant decomposition relations defined in the instance. That is to say, "action α *counts as* action β according to the context of the workflow instance" really means "action α *counts as* action β according to the set of decomposition relations $\mathbb{M} \cup \mathbb{C} \cup \mathbb{O}$".

If we were to allow the aggregation of workflow instances (according to some assumed mechanism), the institution pertaining to the aggregated workflow instance would be the new workflow instance, with its set of decomposition relations formed by the union of the sets of decomposition relations of the instances being aggregated. It may be of some advantage, at some future time, to make some modifications to this view of institution. For instance, we may modify how decomposition relations from workflow instances should be combined. For example, there may be a partial ordering that is prescribed on institutions, so that one of a number of otherwise identical decomposition relations which only differ in the strength of their pre-conditions may remain in the new model, and so on.

We now consider in greater detail the question of which institutional relations may be considered to obtain in various Theodore-based flexible model artefacts.

- Basic Activities: When a basic activity becomes enabled, i.e. set to Running in Liesbet terminology, then two scenarios obtain. If the activity is marked, in HTN terms, as a *task*, then it is intended that it be further decomposed. That is to say, there should exist a suitable method (i.e. *counts as*), operator (i.e. permission), or complex operator (which corresponds to an aggregation of a number of instances of both *counts as* and permission relations) in order that a decomposition may be effected. Thus, a workflow model may be, in itself, an abstract artefact, in the sense that it is meant to be refined further by means of a set of constitutive and regulative rules.

An example which we discuss later (in work related to contract modelling, which uses IWM as a foundational basis) is that of `TransferProperty`. This is a Liesbet basic activity, and also a non-primitive HTN task, in the initial task network/workflow model of the given planning problem. It is meant to be decomposed by an application of an appropriate method. In the example, there is the following method:

Method: `Seq(MultiSeq(3)(Pay), TransferTitle) counts as TransferProperty.`

(Pay on vendee, TransferTitle on vendor)

This method (as a constitutive rule) indicates that three occurrences of a Pay action in sequence *counts as* effecting the institutional action TransferProperty. In this example, Pay itself is an institutional action which, as an HTN task, would be further decomposed. There is a brute action SendCheque whose performance *counts as* effecting Pay. In the example, this is specified as a complex operator. Alternatively, there could be a method which says that SendCheque' *counts as* Pay. The Liesbet basic activity SendCheque' would be a primitive HTN task, indicating that it is a brute action. An operator (as a regulative rule) would then be applied to determine a role for carrying out the brute action. Finally, the Liesbet basic activity SendCheque would be an HTN *action*. It is not possible to further decompose such a basic activity; and its existence in a task network necessarily implies the specification of a role to carry it out.

Note that there is also the possibility of a hierarchy of brute actions. The containers example elucidates this possibility nicely. The transfer-two-containers Liesbet basic activity is a non-primitive HTN task. It is meant to be further decomposed by methods and operators. In this example, a network of load, move and unload *brute* actions effect the action of transferring two containers. It is noteworthy that the transfer of two containers is itself a brute action, in the sense that Searle describes [108, 109]. That is, the transfer of two containers is an antecedently-existent artefact that does not need the context of an institution to be brought into being. For brute action hierarchies, the notion of *counts as* is absent. *Counts as* relations (and constitutive rules) are solely concerned with specifying how institutional actions may be effected.

HTN methods and complex operators (in our Theodore-based framework) may be used to express both constitutive rules and brute action hierarchies. In this sense, the relation between methods and (our embodiment of) constitutive rules is a subsumptive one. In our work, however, our principal concern is that of using Theodore to model *counts as* relations; and thus of using methods and complex operators as an embodiment of constitutive rules.

- Structured Activities: The root of a structured activity may or may not represent an institutional action, as exemplified in the Liesbet activities TransferProperty and transfer-two-containers, respectively, which we have just described. For a structured activity (pre-defined in an IWM-model) representing an institutional action, each of its children contribute to the exercising of the *counts as* relation associated with the activity.

  Strictly speaking, we consider the performance of the last brute action to count as exercising the *counts as* relation, the other brute actions that need to take place to exercise the *counts as* relation are seen as achieving a pre-requisite state for exercising the relation.

  Notably, as each child of a structured activity may itself be child-bearing, there may be a number of further *counts as* relations obtaining, associated with their performance, given that they too are institutional actions.

- Sub-workflows: We can use complex operators to specify pre-defined pieces of workflow logic. These *sub-workflows* specify arbitrarily-complex networks of actions. The use of these

constructs greatly simplifies the verification task, as they specify a single pre-condition for their applicability, and a single effects statement.

Complex operators used to decompose tasks corresponding to institutional actions correspond to an aggregation of a number of *counts as* and permission relations. Further, the enactment of the task network specified by the complex operator *counts as* fulfilling the task that it decomposes. The leaves of the task network specified by a complex operator are `Liesbet` activities which are necessarily `HTN` actions (with roles assigned to them), meaning that there is no further decomposition of the network that need take place.

9.2 Using IWM as a Foundation for Normative Modelling

Having explicated an institutional sense for workflow modelling, we now consider how it may be useful as a foundational basis for normative modelling, and latterly contract modelling. We start this section with an overview of the field of normative modelling, and thereafter proceed with a description of how we have reused IWM for the modelling of contracts.

We use the term 'normative modelling' to mean the modelling of communities, societies, and other kinds of collectives based on the identification of positions pertaining to *norms* that may hold between agents which operate or exist therein. The sense in which we use the term is not limited to a computer science context. Normative modelling, in the sense just expressed, has been extensively studied, for instance in works on legal and societal theory and analytical philosophy [100]. In contrast, applications of normative modelling in computer science remain largely formative (for instance, in the modelling of contracts for automated reasoning over them) with many questions and issues still to be answered.

9.2.1 Normative Modelling

Normative Modelling is concerned with the modelling of normative concepts, or *norms*. From [5], a norm may be defined as: "a principle of right action binding upon the members of a group and serving to guide, control, or regulate proper and acceptable behaviour". In our work, we consider the notion of a normative relation to be useful. We define a normative relation to be a *template for a relationship pertaining to a norm*. The 'template' aspect refers to the notion that such a relation may be parameterised, i.e. it may have arguments. The template would specify the types of these arguments. An example of a normative relation might be a two-argument *obligation*, where instances of this relation obtain according to specific role-action pairs; or a three-argument obligation, where instances obtain according to specific role-action-deadline triples.

A principal context in which Normative Modelling is considered is the domain of legal reasoning, where the normative concepts are legal ones. This is clearly of interest in the context of contract modelling, as a language of contracts would typically include many legal concepts. It should be noted, however, that the scope of normative modelling extends beyond that of domain of legal reasoning.

In what follows, we attempt to give just a flavour of some contributions that have been made within the field of legal analysis and reasoning. Wesley Newcomb Hohfeld (1879-1918) is one of the most acknowledged authors in this field. Hohfeld found the language used in judicial opinions and legal writings to be loose – for instance, concerning such fundamental terms such as rights, duties

and privileges. There was a tendency, Hohfeld believed, to conflate terms which stemmed from a confusion regarding the meaning of legal concepts. This "principle of linguistic contamination" as Hohfeld called it resulted in a oversimplification of complex legal problems [23]. From [58]:

> One of the greatest hindrances to the clear understanding, the incisive statement, and the true solution of legal problems frequently arises from the express or tacit assumptions that all legal relations may be reduced to 'rights' and 'duties', and that these latter categories are therefore adequate for the purpose of analysing even the most complex legal interests ... Even if the difficulty related merely to inadequacy and ambiguity of terminology, its seriousness would nevertheless be worthy of definite recognition and persistent effort toward improvement; for in any closed reasoned problem, whether legal or non-legal, chameleon-hued words are a peril both to clear thought and to lucid expression.

Ross [100] observes that:

> Hohfeld focussed on the relationships that law creates between actors – legal or jural relations. His analysis purports to tackle much of the confusion and ambiguity contained within bald claims like 'I have a right to do X'. Such claims can be interrogated: 'What sort of legal relationship do you claim to have, and with whom do you claim to have it?'.

Brady [23] notes that:

> He ... demonstrates in detail how the distinctions [that he sets out] can be used in solving actual legal problems. ... Hohfeld was after clarity, not for its own sake, but for the definite solution of legal problems.

Hohfeldian analysis may be compared with analytical methods of *social theory*, where the world is conceived as being composed of *social relationships*, which "points to a theoretical environment for Hohfeldian analysis that is of potentially greater explanatory power than analytical jurisprudence taken in isolation" [100]. Indeed, as Brady asserts: "an understanding of his distinctions of normative concepts is an essential starting point for anyone interested in the area of rights, legal or *moral*" [23]. As Jones and Sergot observe, in [63], the concept of power (for instance, as described by Hohfeld) should be considered in a wider context than just law.

Hohfeld, in his work [58], defines eight legal concepts, as presented in Figure 9.1. These are legal (i.e. jural), or more generally normative, relations that may hold between a *party* and a *co-party*. The concepts are organised into two sets. Each relation, which holds for a party, has a *correlate*, which is the same relation when viewed from the perspective of the co-party. Each relation also has an opposite, which conveys the opposite meaning. Ross notes that "[t]o understand the Hohfeldian jural relation is to understand how the mechanics of jural correlativity and jural opposition (as Hohfeld describes these) interact and coexist within the matrix of legal relationships" [100].

We start the discussion of Hohfeld's concepts with those depicted on the left of the figure, the *right-set*. Hohfeld's *right* (*stricto sensu*[1]) (or *claim*) and *duty* (or *obligation*) are *correlates*. They are different ways of viewing the same normative relation concerning a particular subject

[1] In the strict sense – in contrast the generally imprecise, and broad, (mis-)use of the term.

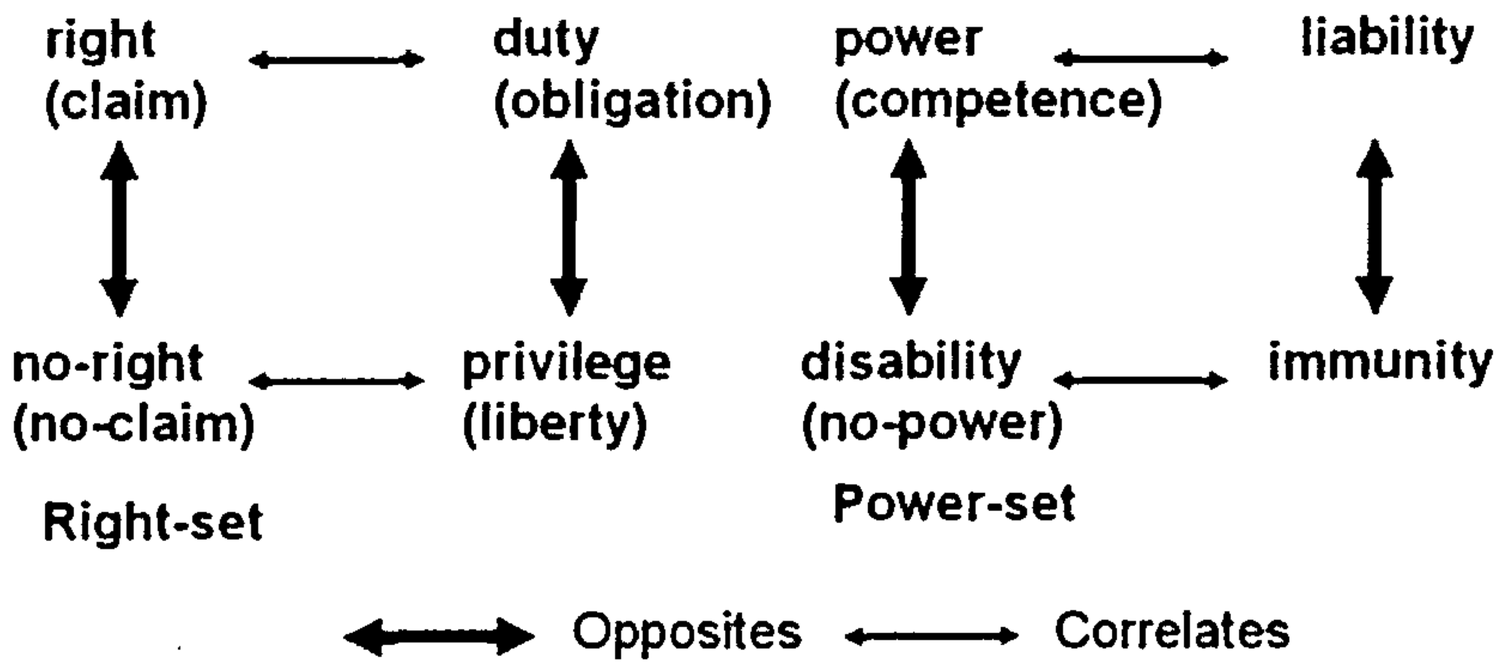


Figure 9.1: Hohfeld's Jural Relations.

matter [23]. Elaborating, if X has a right against Y with respect to subject matter S, then Y is duty-bound to X in respect of S. The relations *no-right* and *privilege* are also correlates.

The relations *duty* and *privilege* are *opposites*, or *contradictories*. If X has a duty to Y with respect to S, then X does not have a privilege to be delinquent with respect to S. Similarly, if X is privileged to Y to be delinquent with respect to S, then X is under no duty to Y with respect to S. It is worth reinforcing that, according to Hohfeld, *privilege* and *right* refer to two very distinct normative concepts. Where X has a privilege, he is free from the claim of another; where X has a right, he has a claim against another [23].

The 18th Century philosopher Bentham, as presented in [71], uses the term "right to a service" for a concept similar, in definition, to *right stricto sensu*. This terminology may be an insightful way of considering Hohfeld's concept – a party has a right to expect a service to be performed by the co-party. Bentham also uses the term *liberty* to convey a notion similar to that of Hohfeld's privilege, viz. "*you have a right to perform whatever you are not under obligation to abstain from the performance of*". Note that he uses the term *right*, here, not *stricto sensu* (i.e. not as a *right to a service*). Bentham further distinguishes two types of liberty: *naked liberty* to do action α – where others have the freedom to (attempt to) prevent α, and *vested liberty* – where others have an obligation not to prevent α. Hohfeld's privilege would appear to be closer to Bentham's naked liberty. As may be seen, Bentham uses the concept of obligation in a more primitive sense than liberty, defining liberty in terms of it. He provides a novel explanation of the concept of obligation, as being imposed by a legislator whenever a law of type *command* or *prohibition* is imposed:

> "...where the provision of the law is a command or a prohibition, [concerning the performance of an act], it creates an offence: if a command, it is the non-performance of the act that is the offence: if a prohibition, the performance ... Moreover the law, in constituting an act an offence, is said to impose thereby an obligation on the persons in question not to perform it."

From [33], we note that Hohfeld's right-set may be expressed, according to the following logical equivalences. Note that $\mathcal{E}_p\alpha$ is a relation, called an action modality in [92], which expresses that 'p brings about α', where α may be some action, or state-of-affairs. Each normative relation is expressed using a three-argument predicate; where the first argument is the bearer of the relation,

the second argument is the co-party, and the third argument is the action that is to be brought about, expressed using the action modality syntax.

$$\texttt{right}(\texttt{p}_1,\texttt{p}_2,\mathcal{E}_{p_2}\alpha) \equiv \texttt{duty}(\texttt{p}_2,\texttt{p}_1,\mathcal{E}_{p_2}\alpha)$$
$$\texttt{right}(\texttt{p}_1,\texttt{p}_2,\mathcal{E}_{p_2}\alpha) \equiv \neg\texttt{no-right}(\texttt{p}_1,\texttt{p}_2,\mathcal{E}_{p_2}\alpha)$$
$$\texttt{privilege}(\texttt{p}_1,\texttt{p}_2,\neg\mathcal{E}_{p_1}\alpha) \equiv \texttt{no-right}(\texttt{p}_2,\texttt{p}_1,\mathcal{E}_{p_1}\alpha)$$
$$\texttt{privilege}(\texttt{p}_1,\texttt{p}_2,\neg\mathcal{E}_{p_1}\alpha) \equiv \neg\texttt{duty}(\texttt{p}_1,\texttt{p}_2,\mathcal{E}_{p_1}\alpha)$$

This formulation of Hohfeld's right-set explicates the different senses of correlate and opposition between the various relations. Whereas right and duty are simple correlates of each other, no-right and privilege introduce a negation into the action modality. Similarly, whereas right and no-right are simple negations of each other, duty and privilege introduce a negation into the action modality as well.

The normative relations in Hohfeld's *power-set* explicate means by which the sum of legal relations that hold between parties may be *changed*. This is an important distinction with respect to the right-set. Ross [100] identifies three situations of change that may be brought about through *exercising* a power:

- the alteration, by virtue of the power, of the incidence, scope, application or effect of existing legal rights or legal powers.
- the annulment, by virtue of the power, of existing legal rights or powers.
- the creation, by virtue of the power, of entirely new legal rights or powers.

As was done for the right-set, Hohfeld pairs off relations into correlates, and opposites. When X has a *power* against Y to change a legal relation concerning subject matter S, then Y is under a liability (as correlate) to X with respect to S. An absence of power is captured by the relation *disability*, whose correlate is *immunity*. If X is disabled against Y with respect to S, then Y is immune from X with respect to S.

Having a liability, as noted by Brady [23], is not always disadvantageous. To be liable is to be subject to the possibility that one's legal relations with respect to a co-party may be changed, at the behest of the co-party. The change may be beneficial just as it may be detrimental. For instance, the owner of land may abandon (through a vested power) their legal entitlement to the land. The particular correlate of this power may be a liability on a party to have powers and privileges created; whereby the party may acquire the deeds of the land. Often, a liability will amount to the creation of a duties on the party, but it may equally pertain to the creation of relations which are of benefit to the party.

Ross [100] notes that:

> [L]egal power can ride a double-decker bus through any existing, settled legal arrangements or legal state. It can thus be said of legal powers that they have the potential to modify the whole gamut of legal states and entire range of legal relationships obtaining at any given time in relation to a particular person or class of persons and specific subject matter.

He continues:

> [N]o legally recognised change of any significance can occur unless the power is exercised. The exercise of the power is what induces a change in the legal situation of persons ... An unexercised legal power is merely a potential legal competence but it is of limited legal significance in so far as it remains unexercised.

The principal contribution of Jones and Sergot in [63] is to have proposed *counts as* as capturing the meaning of the Hohfeldian notion of power. That is to say, a power is defined by virtue of how it may be exercised, as expressed by a *counts as* relation or, in the language of Searle [108, 109], by a constitutive rule. Moreover, they do not consider power to be an exclusively legal phenomenon, using the term *institutionalised power* in order to emphasise as much. They assert that power "is a standard feature of any norm-governed organisation where selected agents are assigned specific roles (in which they are empowered to conduct the business of that organisation)."

It is the modelling of norm-governed organisations to which the notion of *normative modelling* (NM) pertains. We propose our work on *institutional modelling* with respect to *counts as*, and permission, as a foundational component of NM, where the modelling of contracts (i.e. instantiation in a legal context) is one possible application of NM.

The work of Jones and Sergot reinforces the clear separation between privilege and power made by Hohfeld. As described in [63], Hohfeld explicitly distinguishes between (i) legal power, (ii) the practical possibility to carry out the acts necessary for the exercise of the legal power and (iii) the privilege to carry out those acts. Jones and Sergot go on to cite an example proposed by Makinson which further exemplifies this distinction. Makinson [72] asks us to:

> ... consider the case of a priest of a certain religion who does not have *permission*, according to the instructions issued by the ecclesiastical authorities, to marry two people, only one of whom is of that religion, unless they both promise to bring up the children in that religion. He may nevertheless have the *power* to marry the couple even in the absence of such a promise, in the sense that if he goes ahead and performs the ceremony, it still *counts as* a valid act of marriage under the rules of the same church even though the priest may be subject to reprimand or more severe penalty for having performed it.

As Jones and Sergot note, "[t]his is clearly a case in which the priest is empowered to marry the couple, but not permitted to do so[2]".

In the modelling of contracts, normative concepts such as *obligation*, *power* and *privilege* prove to be useful. It is instructive to consider how IWM may be used to provide a basis for modelling such normative concepts. We address this point in the following section.

9.3 Contract Modelling

In Section 9.4, we describe how we have reused concepts identified for Institutional Workflow Modellinng (IWM) in the modelling of contracts. Before doing so, in this section, we give an overview of related work in the field of contract modelling. We firstly describe work that we have

[2]Indeed, they go on to note a report about "clandestine religious services conducted by former Roman Catholic priests who had left the priesthood to marry", but who still "retain their sacramental powers but are forbidden to exercise them".

carried out on a non-IWM based approach to contract modelling. We then review other related research contributions.

Note that our non-IWM based approach to contract modelling was realised as part of work contributing to this thesis, although for simplicity we choose not to enumerate it as a contribution in the introduction (see Section 1.2).

9.3.1 A Non-IWM Based Approach to Contract Modelling

In [42, 43, 41], we consider the modelling of contracts, so that we may monitor their performance at run-time. We ground our work by considering the modelling of *Service Level Agreements* [68] for *Utility Computing* (UC) [6].

UC offers an opportunity to corporate businesses to maximise the efficiency and efficacy of their IT service provision (both in-house and to customers). It allows them to out-source large areas of their IT service provision to UC-data centres, which will agree to provide computational resources, packaged as services to them. SLAs are essential for formalising the objectives of a UC service, and to manage expectations [68].

The levels of service that are agreed between a UC service-provider and customer are mandated by Quality-of-Service (QoS) guarantees, written as Service-Level Guarantees (SLGs) within Service-Level Agreements (SLAs). An example SLG might be:

- Service Availability should be greater or equal to 99%, weekdays 9a.m.-5p.m.
- Service Availability should be greater or equal to 95%, at all other times.
- Availability metric is measured over each calendar month; penalty for SLG violation: refund customer their monthly fee.

In [42, 43, 41], we define the state of a contract, at a particular time, to be the aggregation of instances of normative relations that hold between contract parties, plus the values of contract variables, at that time. A contract variable is a piece of numerical state whose value can change over the deployment lifetime of its containing contract. Its use will be normative in that it will have been agreed upon when the contract is formed. In this sense, a contract variable may also be considered to be a special kind of normative relation.

There are at least two functional aspects to the run-time performance monitoring of contracts: (i) Tracking the effect of events (pertinent to a contract) on contract state - the contractual (or, normative) relations that hold between contract parties - and informing interested parties of past, present and (possible) future contract states; and, (ii) Assessing the current state of the contract, in terms of its utility (that is, worth), and other metrics related to business intelligence [1]. The work that we have done is primarily concerned with the first of these, which is known as automated contract (state) tracking to distinguish it.

Notably, approaches to automated tracking of contract state, thus far, can be largely characterised in one of two ways [14]: (i) As general-purpose contract reasoning frameworks that (mainly) have not been applied in actual, deployed systems; or (ii) In the case of SLAs, as being fairly limited in capability. The work presented here is considered to be distinguished from such approaches in that: (i) It has been developed in the context of a 'real-world' deployment scenario (namely, SLAs for UC), while being generalised so to be applicable to other domains; and (ii) It represents

an advance (over many approaches) in what can be realised regarding automated state tracking for contracts.

We develop a general approach to the tracking of state based on a version of the Event Calculus (EC), originally presented in [99]. Simply put, EC allows the expression of domain axioms which characterise how propositional properties of a domain (*fluents* in the AI terminology) change according to the occurrence of domain events. Various forms of reasoning can be undertaken using such a set of domain axioms, such as *planning* (a sequence of actions that will take the domain from an initial state to a goal state), *prediction* (where given an initial domain state, and a sequence of domain events, an *event narrative*, we seek a resulting domain state), and *postdiction* (where given a current domain state, and a set of a domain events, we seek an initial domain state) [112]. In this terminology, state tracking is a special case of prediction, except that we shall also want to have access to all intermediate states as well as the initial and final ones. The Event Calculus is presented in a logic programming framework, and is usually implemented using a logic programming language such as Prolog or using techniques from deductive databases. In this work, for deployment in a business context, we have constructed a Java implementation of an EC reasoning component, called the *Event Calculus State Tracking Architecture* (ECSTA).

There have been many diverse research contributions that have utilised the Event Calculus (EC) for the purpose of reasoning over the effects of events on a logic theory. Some that are closely related to this work are now presented. In [15], Artikis describes the representation in EC of 'open' multi-agent systems viewed as societies of computational agents, including variations on a number of collaborative work protocols, among others. This work also explicitly employs the concepts of obligation, permission, and institutional power, and includes the specification of sanctions and penalties in the case of violations. It is also worth noting that Artikis and colleagues have also employed other action languages from AI as an alternative to the use of EC, and specifically the action language $\mathcal{C}/\mathcal{C}+$ [53]. $\mathcal{C}/\mathcal{C}+$ provides a high-level notation for defining laws specifying the effects of actions on domain fluents, and ways of characterising domain phenomena, such as the *common sense law of inertia.* It also has an explicit semantics in terms of labelled transition systems. Being able to describe contracts as transition systems is extremely useful for proving properties (using *model checking*) about the contracts. Also of note is an extended form of $\mathcal{C}/\mathcal{C}+$, called $(\mathcal{C}/\mathcal{C}+)^{++}$ [110], which is specifically defined for the representation of norms and institutional concepts. These extensions provide a treatment and formal semantics for institutionalised power, that is, *counts as* relations between actions, and for the specification of permitted (or acceptable, or legal) states of a transition system and its permitted (or acceptable or legal) transitions and histories.

In [12], Bandara and colleagues develop methods for performing analysis and refinement of policy specifications, employing an EC-based representation of both policy and system behaviour specifications. The resulting formalism is used in conjunction with abductive reasoning techniques to perform *a priori* analysis of policy specifications. In [104], Sadighi and colleagues develop an EC-based framework for issuing privileges to agents in a community, through *declaration* and *revocation* authority certificates. A distinction is made between the time a certificate is issued, or revoked, and the time for which the associated privilege is created, or discharged, enabling certificates to have prospective and retrospective effects.

Example Contract

We base the development of the approach described in [42, 43, 41] on the representation of a number of UC agreements. We use the following mail service agreement in order to ground our work.

- *The Service Provider (SP) will provide a mail service to the Service Customer (SC), which includes a mailbox with a quota of s GBytes. SC will be charged a fixed monthly fee of* $s * c_0$ *for the service.*

- *In the case that the mail service is unavailable, SP will pay p for every whole t minutes that it is unavailable. SP is obliged to pay any penalties to SC within a month of their accrual.*

- *Whenever u>s, where u is the mailbox utilisation in GBytes, SP will charge SC* c_1 *for each GByte over s, calculated daily.*

- *All billing of SC occurs monthly, and SC is given a month thereafter to pay. If SC fails to pay within the given time, SP may terminate the mailbox service without notice.*

In our work, a contract model is defined from a global perspective, as opposed to being an aggregation of a pair (in the case of two contract parties) of end-point perspectives. Live representations of a contract model may be replicated by contract monitoring engines belonging to each of the contract parties, and/or may be maintained by a single contract enforcement authority. In either case, in the following, we shall talk about events being routed to and from the environment. Events from the environment are known as exogenous events. Contract parties may perform actions which are observed by the contract model (however this is realised in a monitoring/enforcement context). Such actions are (the only source of) exogenous events. In response to these events, the contract model may push output events to interested parties in the environment. In the case of the centralised enforcer, the contract model will notify all contract parties of the event. In the case of local monitoring agents, some protocol is assumed such that all parties agree on output events as they are generated.

For the purpose of tracking the normative state of contracts, we are concerned with identifying events described in the contract that may have an effect on contract state. These may be exogenous events, as just defined, or events that are generated internally such as the expiration of a timer prescribed in the contract. Once identified, we need to express in our representation the effects on contract state of these events.

For example, the contract excerpt: "*All billing of SC occurs monthly*" indicates a *monthly billing event.* One effect of such an event is that *SC* receives an invoice for service. But this is not an effect on contract state, *per se.* We shall say that another effect of this event – this time, on the contract state – is to instantiate an instance of a normative relation, namely an obligation bearing on *SC* to pay *SP* for service within a month.

Another example is: "*If SC fails to pay within the given time, SP may terminate the mailbox service without notice*". This statement talks about another event, which occurs when the specified time period expires before *SC* fulfils its obligation (to pay for service) on time. We shall say that an effect of this event is to instantiate an instance of another normative relation, namely (vested) power of *SP* to terminate the mailbox service.

A Brief Introduction to the Event Calculus

From the perspective of what needs to be represented for contract state tracking, we need some way of representing the effects of events on contract state. For this, we use the Event Calculus (EC). In the following, we present a rather informal description of EC, and its use for the representation of contracts. The interested reader is referred to [99, 111] for a formal presentation.

We say that a contract in the Event Calculus is a conjunction of:

- A finite set of **initially** statements, which prescribe instances of normative relations that initially hold (i.e., are true).
- A finite set of **initiates** (resp. **terminates**) statements, which prescribe instances of normative relations which start (resp. cease) holding on an event occurrence.
- A finite set of **happens** statements, which record the occurrence of events as an *event narrative.*

There are also a collection of foundational axioms which we leave out of this presentation for reasons of brevity.

Our embodiment of EC also admits **timer** statements, for generating timer events which may be one-off or recurrent. One can view **timer** statements simply as a mechanism for inserting **happens** events into the event narrative.

Representation of Example Contract

We now give an informal presentation of the mail service agreement, represented using our EC-based approach.

- For the contract fragment: *Whenever u>s, where u is the mailbox utilisation in GBytes, SP will charge SC* c_1 *for each GByte over s, calculated daily*, it is assumed that an external event `daily_charge_event` is entered into the event narrative daily providing the daily charge that the customer has incurred, where this charge will be zero if the value of `u` has not gone above `s` for that day. The daily charge is accumulated in a contract variable `vDailyCharge`.

  In the EC-based representation, there is an **initially** statement, which indicates the initial value of `vDailyCharge`, viz.

  initially `vDailyCharge=0`.

  There is also a statement that says that when a `daily_charge_event` occurs, the value of the event's `Charge` parameter, corresponding to the charge for the day, is added to the current value of `vDailyCharge` to give the new value of this variable.

  event `daily_charge_event(`*Charge*`)` **initiates** `vDailyCharge=`*V* *if* `vDailyCharge=`*V*1 *and* *V*=*V*1+*Charge*

- The contract excerpt: *All billing of SC occurs monthly* is accommodated by a *timer* called `billing_timer`:

  timer `billing_timer monthly`.

This has the effect of inserting events, represented as instances of the **happens** relation, into the event narrative.

- The contract excerpt: *SC will be charged a fixed monthly fee of s*c0 for the service* is accommodated as follows. In response to the monthly `billing_timer` event, we create an instance of an obligation normative relation, which bears on SC to pay for service, viz.

  event `billing_timer` **initiates** `o(PayForService(`*Charge*`), SC, X, 1 month)` *if* `vDailyCharge=`$V1$ *and* *Charge*`=`$V1$`+sc0` *and* `new_id(X)`

  where `new_id(X)` allocates a unique identifier that has not been previously used (for recording instances of normative relations).

  The `Charge` parameter is assigned the value obtained by summing the current (accumulated) daily charge, given by the contract variable `vDailyCharge`, with the value (currently) assigned to the contract parameter sc_0. `PayForService` is the name of an action that needs to be carried out by SC, which is given as the second parameter of the obligation relation. There is also a time-limit of 1 month associated with the fulfilment of the obligation instance.

- The contract excerpt: "In the case that the mail service is unavailable, SP will pay p for every whole t minutes that it is unavailable" is, in fact, part of a Service-Level Guarantee (SLG), namely, the SLG pertaining to the provision of the mail service. Specifically, it describes what course of action is normative in the case that the SLG is violated by SP.

  We assume that some monitoring agent tells us when the SLG has been violated, that is that the mail service is unavailable. This agent will generate an event, *SLG*1*_violated* say, to this effect; and will generate an event, *SLG_restored* say, when the mail service has been restored.

  We define a timer for the SLG, thus:

  timer `SLG1_timer` t `minutes.`

  Also relation `o(RestoreService,` SP`)` is defined as an obligation that bears on SP to restore the service. This contract excerpt would then be represented as follows:

  - *event* `SLG1_violated` **initiates** `SLG1_timer`
  - *event* `SLG1_violated` **initiates** `o(RestoreService,` SP`)`
  - *event* `SLG1_restored` **terminates** `SLG1_timer`
  - *event* `SLG1_restored` **terminates** `o(RestoreService,` SP`)`
  - *event* `SLG1_timer` **initiates** `vPenalty=`V *if* `vPenalty=`$V1$ *and* V`=`$V1$`+`p

Event Calculus State Tracking Architecture (ECSTA) and Contract Visualiser

We have implemented, in Java, a reasoner for *EC*-based contracts, called the *Event Calculus State Tracking Architecture* (ECSTA), supporting: instantiation of contracts written in EC, assertion of event narratives including speculative narratives which can be unrolled, and querying of contract state. As well as the ECSTA reasoner, a tool called Contract Visualiser has been implemented

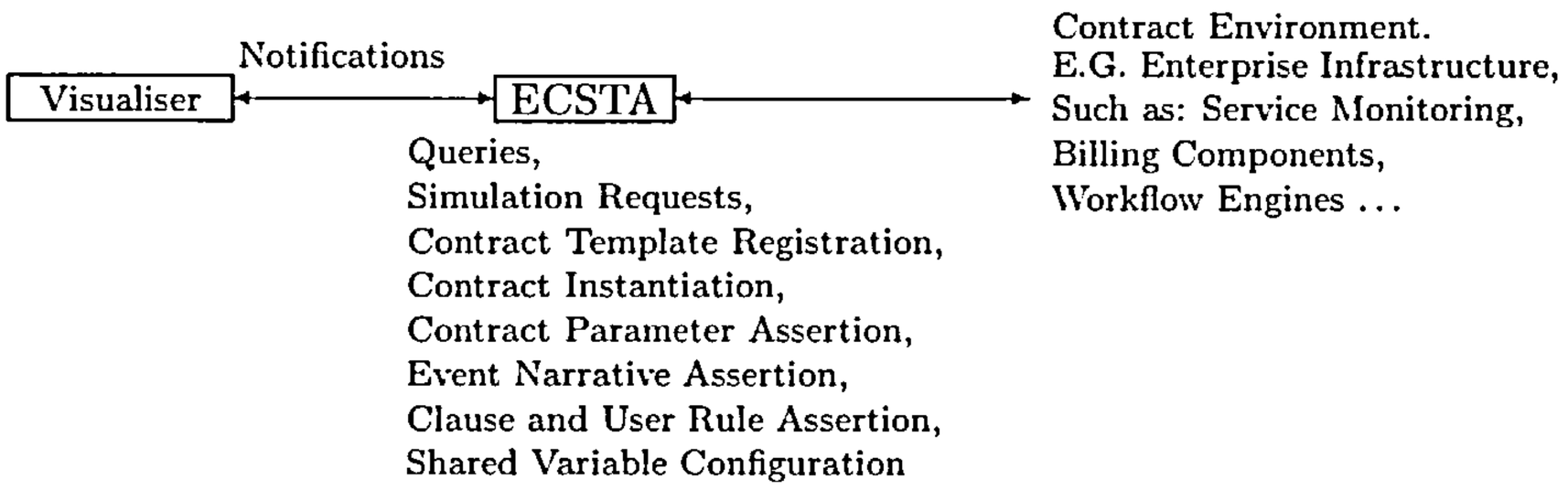


Figure 9.2: Relationship between ECSTA and Contract Visualiser

which allows for the deployment management of contracts. The relationship between ECSTA and Contract Visualiser is depicted in Figure 9.2.

In the following narrative, we present the evolution of a scenario pertaining to the mail service agreement as it would be captured within Visualiser. As screenshots may be hard to read, we present the scenario in the form of tables which capture the same information as would be presented by Visualiser. For illustration, an actual screenshot for the final stage of the scenario is shown in Figure 9.3.

In stage 1 of the scenario – see Table 9.1 – we see that the state of the mail service agreement contract instance is "OK" to begin with.

Occurrence	Date/Time
STATE: OK	Fri 3 Sep 2004 22-15-03

Table 9.1: Scenario Unfolds: Stage 1

In stage 2 – see Table 9.2 – we see that a "Service Violation" event occurs causing the state of the contract instance to change to "Service Violation" and an obligation to be initiated bearing on the provider to restore the service.

Occurrence	Date/Time
STATE: Service Violation	Mon 13 Sep 2004 22-15-03
INPUT EVENT: SERVICE VIOLATION with (id: slg1)	Mon 13 Sep 2004 22-15-03
OUTPUT EVENT: OBLIGATION with (id: o0, bearer: provider, actions: resolve breech with (id: slg1), deadline: not specified)	Mon 13 Sep 2004 22-15-03

Table 9.2: Scenario Unfolds: Stage 2

In stage 3 – see Table 9.3 – we see that a "Service Restoration" event occurs causing the state of contract instance to return to "OK". Also the obligation bearing on the provider to restore the service is fulfilled.

In stage 4 – see Table 9.4 – we see that two obligations are initiated (by timers that are specified in the contract instance representation and maintained by the reasoner) stipulating that: *SP* (a.k.a. "provider") must refund $25 to *SC* (a.k.a. "Mike Consulting") for poor service (before end of business day) and *SC* must pay $50 for service to *SP* (within 1 month). This causes the contract instance to move into state: "Provider Payment Outstanding" + "Customer Payment

Occurrence	Date/Time
STATE: OK	Mon 13 Sep 2004 22-45-03
INPUT EVENT: SERVICE RESTORATION with (id: slg1)	Mon 13 Sep 2004 22-45-03
INPUT EVENT: OBLIGATION with (id: o0, status: fulfilled)	Mon 13 Sep 2004 22-45-03

Table 9.3: Scenario Unfolds: Stage 3

Outstanding".

Occurrence	Date/Time
STATE: Provider Payment Outstanding, Customer Payment Outstanding	Tue 14 Sep 2004 22-15-03
OUTPUT EVENT: OBLIGATION with (id: o1, bearer: provider, actions: refund money with (amount: 25.00), deadline: end bus. day)	Tue 14 Sep 2004 22-15-03
OUTPUT EVENT: OBLIGATION with (id: o2, bearer: Mike Consulting, actions: pay for service with (amount: 50.00), deadline: 1 month)	Tue 14 Sep 2004 22-15-03

Table 9.4: Scenario Unfolds: Stage 4

In stage 5 – see Table 9.5 – we see that an input event saying that SP has fulfilled its obligation to refund $25 to the service customer occurs causing: the state of the contract instance moves from "Provider Payment Outstanding" + "Customer Payment Outstanding" to just "Customer Payment Outstanding". The fulfilment of the obligation bearing on SP occurs just 10 minutes after it was initiated and within the business day as stipulated – the manifestation of the fulfilment may be that the billing system sent the customer a cheque, or organised a fund transfer.

Occurrence	Date/Time
STATE: Customer Payment Outstanding	Tue 14 Sep 2004 22-25-13
INPUT EVENT: OBLIGATION with (id: o1, status: fulfilled)	Tue 14 Sep 2004 22-25-13

Table 9.5: Scenario Unfolds: Stage 5

In stage 6 – see Table 9.6 – we see that the 1 month timer for the obligation bearing on the service customer to pay for service has expired: this moves the contract instance into a "Terminable" state – SP is empowered to terminate the contract instance.

Occurrence	Date/Time
STATE: Terminable	Thu 14 Oct 2004 22-15-03
INPUT EVENT: OBLIGATION with (id: o2, status: timeout)	Thu 14 Oct 2004 22-15-03

Table 9.6: Scenario Unfolds: Stage 6

In stage 7 – see Figure 9.3 and Table 9.7 – we see that, in keeping with SP being empowered to terminate the service, they do so; the contract instance moves into a "Terminated" state.

It is worth noting that when SP becomes empowered to terminate the agreement, there is no mechanism within the contract for specifying how this may occur. This is where the notion of *counts as*, as used in our IWM-based approach, comes into play. That is, a contract party

Contract Visualiser

Zoom | Describe | Saturday 16 October 2004 18-19-32 | Help

State History. Contract #4, Customer:Mike Consulting, Template:Mail Service, Name:For Mike Maxwell.

Occurrence	Date/Time
STATE: OK	Friday 3 September 2004 22-15-3
STATE: Service Violation	Monday 13 September 2004 22-15-3
INPUT EVENT: SERVICE VIOLATION with (id: sig1)	Monday 13 September 2004 22-15-3
OUTPUT EVENT: OBLIGATION with (id: o0, bearer: provider, actions: resolve breech with (id: sig1), deadline: not specified)	Monday 13 September 2004 22-15-3
STATE: OK	Monday 13 September 2004 22-45-3
INPUT EVENT: SERVICE RESTORATION with (id: sig1)	Monday 13 September 2004 22-45-3
INPUT EVENT: OBLIGATION with (id: o0, status: fulfilled)	Monday 13 September 2004 22-45-3
STATE: Provider Payment Outstanding, Customer Payment Outstanding	Tuesday 14 September 2004 22-15-3
OUTPUT EVENT: OBLIGATION with (id: o1, bearer: provider, actions: refund money with (amount: 25.00), deadline: end bus. day)	Tuesday 14 September 2004 22-15-3
OUTPUT EVENT: OBLIGATION with (id: o2, bearer: Mike Consulting, actions: pay for service with (amount: 50.00), deadline: 1 month)	Tuesday 14 September 2004 22-15-3
STATE: Customer Payment Outstanding	Tuesday 14 September 2004 22-25-3
INPUT EVENT: OBLIGATION with (id: o1, status: fulfilled)	Tuesday 14 September 2004 22-25-3
STATE: Terminable	Thursday 14 October 2004 22-15-3
INPUT EVENT: OBLIGATION with (id: o2, status: timeout)	Thursday 14 October 2004 22-15-3
STATE: Terminated	Friday 15 October 2004 22-15-3
INPUT EVENT: TERMINATE AGREEMENT	Friday 15 October 2004 22-15-3

Figure 9.3: Final Stage of Mail Service Scenario.

Occurrence	**Date/Time**
STATE: Terminated	Fri 15 Oct 2004 22-15-03
INPUT EVENT: TERMINATE AGREEMENT	Fri 15 Oct 2004 22-15-03

Table 9.7: Scenario Unfolds: Stage 7

may ascertain how they may fulfil obligations through the query-interpreter available in our IWM-based framework. This mechanism decomposes the fulfilment of obligations using extant *counts as* relations (which in our work on contract modelling may be considered as an embodiment for the normative relation *power*).

Finally, we provide functionality which handles the management of contract instances, such as:

- Discovery of registered, and registration/deactivation/reactivation/annulment of, contract templates.
- Discovery of instantiated, and instantiation/reactivation/deactivation/annulment of, contract instances.
- Addition/annulment/activation/deactivation of contract clauses
- Changing of contract parameters.
- Assertion of contract events.
- Contract querying.
- Registration for/deactivate/reactivate notification of contract events.
- Registration for/deactivate/reactivate notification of contract clause application.

9.3.2 Other Related Work

There has been a good deal of research concerning the representation of contracts for monitoring their performance. In [81, 88] Milosevic and colleagues identify the scope for automated management of e-contracts, including contract drafting, negotiation and monitoring; and describe the design and implementation of a contract monitoring facility, for cross-organisational contract management. In [34], Daskalopulu discusses the use of Petri-nets for contract state tracking, and assessing contract performance. Her approach is best suited for contracts which can naturally be expressed as protocols, or workflows. One particular desirability of using Petri-nets is that they naturally facilitate analysis. In the context of contract representation, an example would be to show that a contract will always terminate in a favourable state for one, or more, contract parties. It is possible, however, to carry out analysis of this nature using our non-IWM-/IWM-based approach to contract modelling.

Molina-Jiminez and colleagues [82, 115] consider a framework by which contracts may be represented in machine form; and how they may be monitored and enforced at run-time. They advocate the use of Finite State Machines (FSMs) for the representation of contracts, which, specifically, capture *obligations* and *rights*[3] that may obtain between parties.

Their interest primarily concerns the use of contracts in a *business* and *cross-organisational* context. They make the reasonable assumption that business processes (unless atomic) can be decomposed into sub-processes, which are of lower complexity. The interaction that then takes place between business partners, pertaining to such sub-processes, (often) may be regulated by separate sub-contracts. In their approach, each sub-contract would prescribe the rights and obligations that may come into existence before, during and after the execution of the sub-process. An example given is a number of sub-contracts which, when aggregated by a parent contract, pertain to the provision of food-related items to a consumer – one sub-contract relating to the provision of tinned items, another to the provision of fresh items, and so on.

In discussing the requirements for a formal representation, the authors observe, in [82], that a fundamental requirement is the capability of validating *correctness requirements*. In [115], they present a list of what they consider to be common requirements, such as correct commencement and termination, absence of locking, and other properties relating to the soundness of contracts.

In [82], the authors define a *right* to be: "an action that a signing entity [may] do if it wishes". This notion corresponds most closely to a Hohfeldian privilege. It does not correspond to a Hohfeldian right, *stricto sensu*. An obligation is defined, in [82], in terms of what is usually considered to be a synonym, namely, duty: "an obligation [is] ... a duty that an entity is expected to perform". The sense of this is apparently that of Hohfeldian obligation.

A contract is represented as a pair of FSMs, one for each contracting party, that interact with each other [82]. A contract is, thus, represented as a pair of local views. For any state of a particular local view's FSM, there will be a number of events (which may be either locally generated, or generated by the foreign FSM) whose occurrence may change the state of the given FSM. A given FSM state will, thus, be a source to a number of output arcs, some of which pertain to (currently active) "rights" that the particular party may exercise, as well as obligations that the party may fulfil. Exercising a "right", or obligation, at one side of the contract may, or may not, have an effect at the other side.

[3] Not *stricto sensu*.

The approach documented in [82, 115] advocates the use of underlying middleware to enforce the rights and obligations that bear on contract parties, as tracked in each FSM. The model checker SPIN [59] is used, on FSM representations coded in SPIN's input language Promela, in order to verify arbitrary LTL-expressed [61, 38] constraints. SPIN may also be used to detect deadlock and livelock.

Some other research contributions which have considered the modelling of contracts are as follows. In [22], Grosof and colleagues have sought to address the representation of business rules for e-commerce contracts. For this purpose, they have developed the SWEET (Semantic WEb Enabling Technology) toolkit, which enables communication of, and inference with, e-business rules written in RuleML [9]. In contrast to our approach, Grosof and colleagues are not concerned with maintaining live representations of contracts for state tracking purposes. A facility for tracking contract state is (ostensibly) lacking in their work. Rather, they seek to represent contracts for the purpose of communicating contract rules.

Some similar work is that presented by Paschke, in [89]. Also based on RuleML, Paschke describes the language RBLSA, meant for the rule-based representation of Service Level Agreements. There is a significant overlap between the work of Paschke and that of Grosof, described above, such as the facility for specifying procedural attachments (predicates that are implemented by an external procedure, such as a Java method), and rule priorities. Unlike Grosof, however, Paschke's work does support the tracking of live contract state, through the use of *EC*-like rules, as well as explicating deontic concepts such as obligation and permission as distinct ontological constructs.

Finally, [75, 93] consider the modelling of the normative state that obtains between provider and consumer when an agreement for web service provision is agreed, or signed. Notable in both of these works is their rather primitive consideration of the normative relations that may obtain between parties. For instance, they do not pay any attention to the modelling of meta-level normative concepts, such as Hohfeldian power.

9.4 An Approach to Contract Modelling Based on Institutional Workflow Modelling

We now present how we have reused work that we have done on Institutional Workflow Modelling as a basis for the modelling of contracts. A key benefit of using IWM as a basis for normative and contract modelling is that we are able to establish emphatically the association of powers and privileges with the fulfilment of obligations. That is, in our non-IWM based work, obligations, powers and privileges may be asserted to hold, but we never put in place a means of decomposing the fulfilment of obligations using powers and privileges, because we did not define a mechanism for doing so. IWM gives us that mechanism.

In our IWM-based contract modelling (CM) work, we consider that a contract may generally be defined as a collection of *protocol fragments*, together with rules specifying how, and when, these fragments obtain, as well as rules for specifying how auxiliary (instances of) normative relations (specified within the contract) may be created or annulled, according to event occurrences. We consider contract variables, as described in our non-IWM based approach, to be a special kind of normative relation, albeit their purpose may be simply to record state that is needed for the correct operation of the contract, but is of no or little interest to the contract parties themselves.

An example may be a simple counter. These rules may be considered to be instances of *Hohfeldian* powers. From this description, we may also consider the rules to be akin to *Event Condition Action* (ECA) rules.

For our purposes, we view a protocol fragment as a partial ordering of tasks that need to be realised by contract parties. In our work on CM, not using IWM, we model protocol fragments in a rather awkward way. That is, we specify a number of ECA-like rules which control the creation and annulment of various normative relations, such as obligations, powers and privileges. In contrast, in our IWM-based work, we model protocol fragments in a contract as IWM-based (i.e. HTN-based Liesbet) workflows, consisting of tasks and actions. Using a task network-based (i.e. workflow) language is more natural for expressing protocols, given that protocols resemble task networks.

9.4.1 Legal Relations in a Theodore-based IWM Protocol Fragment

The legal relations that may be considered to exist in a Theodore-based IWM protocol fragment may be viewed from two directions, from legal to IWM concepts and from IWM to legal concepts. It is worth noting that the point of casting our work on flexible and traditional workflow modelling in an institutional sense is to establish the link between legal concepts such as *power*, say, and those explicated in our approach to workflow modelling, such as the *method* artefact in HTN-based planning. Power and method are related through the institutional concept of *counts as*, which Jones and Sergot [63] propose as an apposite means of characterising the exercising of legal power.

We map legal to IWM concepts as follows.

- We map Hohfeldian Power (i.e. exercising thereof) to *counts as*, and thus methods (and complex operators)
- We map Hohfeldian Privilege to permission, and thus operators (and complex operators)
- We map Hohfeldian Obligation to basic/leaf activities in protocol fragments

Note that, in this part of our work, we assume brute actions to be atomic (i.e. we do not admit the notion of brute action hierarchies, as previously described in Section 9.1.2), although this may not always be appropriate. This assumption simplifies the following discussion.

When considering the mapping in the other direction, the fundamental point of interest is leaf activities. As stated, these are considered to be obligations (when the activity is enabled, i.e. is Running). These are obligations either to do an institutional or brute action, which is determined by whether the activity is a non-primitive or primitive task, respectively.

As in IWM, the following applies in CM.

- Primitive tasks (i.e. obligations to carry out brute actions) demand the presence of operators (i.e. privileges) so that agents may be identified to carry out these actions.
- Non-primitive tasks (i.e. obligations to carry out institutional actions) demand the presence of methods (i.e. powers) to facilitate their refinement into networks of primitive and non-primitive tasks (i.e. brute and institutional actions, respectively), where these tasks are to be further refined using extant operators (i.e. privileges) and methods (i.e. powers).
- An HTN action represents a brute action which comes with an automatic privilege, assigned to a particular role.

- Complex operators combine the application of methods (i.e. powers) and operators (i.e. privileges).

Note that the *absence* of a power (resp. privilege) to perform an institutional (resp. brute) action is determined by the absence of a method or complex operator (resp. operator, or containing complex operator), to carry out the action. The absence of any such a decomposition relation does not automatically imply the presence of some disability (resp. prohibition) relation, however. It is possible to model such a *closed policy* [41], if desired, using an auxiliary theory.

The entity which controls the availability of decomposition relations (i.e. methods, complex operators and operators), and thus controls empowerment and assigning of privileges is the contract model. We describe this entity in some detail in the following presentation.

Note that we consider that obligations to perform brute actions bear on particular roles. In contrast, obligations to perform institutional actions may bear on particular roles, but may also be described as being anonymous where the intent is that further decompositions of the pertaining HTN task prescribe specific roles. Similarly, methods, operators and complex operators may be role-specific, or anonymous. We consider that anonymous normative relations and decomposition relations bear on the institution that is the contract itself.

9.4.2 Event Handling Logic

The specification of a contract in our IWM-based framework comprises a contract model, along with a number of IWM-specified protocol fragments. The contract model essentially specifies the effects of contract-related events (both exogenous and internally-generated) on contract state, creating and annulling protocol fragments in response to these events, as well as creating and annulling instances of normative relations.

The underlying event handling logic, in our contract modelling framework, is an evolution of that used in our non-IWM based work. It is still based on the Event Calculus. The framework stores both exogenous events and events from IWM-based protocol fragments (such as changes in the state of activities) in an event narrative, which is a similar artefact to that described previously. Rules within a contract model dictate the effects of these events. As well as causing the state of (instances of) normative relations recorded in the *kb* to change, these events may cause the assertion of state signifying that instances of particular workflow fragments should be created or annulled. The contract model may also determine that certain events should be pushed to subscribers in the environment, in response to events in the event narrative.

As an example, the contract model may contain the rule:

```
event E initiates create(TerminateService)
```

where `TerminateService` is the name of a workflow fragment. According to this rule, an instance of the `create(TerminateService)` relation should be asserted to *kb* in response to the occurrence of `E`. There is a corresponding `annul` relation, which indicates the annulling of workflow fragments. The assertion of `create` and `annul` instances are transparent to the IWM engine, which will act on them accordingly.

9.4.3 Verification of Contract Fragments

For verification, we make a distinction between *fixed* and *variable* contract models. A necessary condition for a model to be fixed is that the set of decomposition relations, described therein, is fixed. This is a condition that carries over from our work on flexible workflow modelling. Another necessary condition is that a power may not be exercised in the absence of an obligation that prescribes the institutional action to which the power applies.

As our approach to contract modelling is based on our work on flexible workflow modelling, we reuse a lot of the components implemented in the verification and enactment engine for Theodore flexible workflows. For fixed contract models, a contract author or contract party (in enactment) may make use of the IWM-based verification facilities for soundness (i.e. completion along all enactment paths) and arbitrary properties expressed in a constraint language, such as a temporal logic like CTL*.

Note that properties are *not* verified for a contract as a whole, rather, just for individual protocol fragments, i.e. IWM-based workflows. As a result, there is a notion of independent *sub-contracts* that is imposed, where each IWM fragment is such a sub-contract. This is not necessarily as restrictive as it may sound. As described in the work of Molina-Jiminez [82, 115], it is often the case that contracts will naturally be composed of independent sub-contracts. The notion of contract soundness carries over from the notion of soundness defined for IWM models. A contract is sound iff each protocol fragment defined therein is sound, according to the IWM-based criteria for soundness. Our verification approach also ensures that the presence of an obligation always entails the presence of sufficient powers and/or privileges so that the obligation can be fulfilled.

Often, sub-contracts will specify fully-prescribed protocols of behaviour that should take place between contract parties. Complex operators are ideal candidates for such protocols. An example of such a protocol might be a buyer-seller protocol for buying goods and having them delivered. This could be expressed as the sub-workflow `Seq(Pay, Deliver)`, with pre-specified decompositions of `Pay` and `Deliver` into finer structured and basic activities. These activities will represent further power and privilege relations, respectively, which hold by prescription of the complex operator.

Note also that, for both fixed and variable models, a contract party may perform "what-if" simulation and "what may I do next" querying.

9.4.4 Derivation of Obligation Fulfilment

One aspect of our IWM-based approach to contract modelling is the mechanism it affords for decomposing the fulfilment of obligations using powers and permissions. Such a facility in contract modelling and enactment frameworks is typically overlooked, although some such as Molina and colleagues [82, 115] do make provision for reconciling privileges with obligation fulfilment. That is, their work does not consider the distinction between institutional and brute actions in contracts, but they do consider that the presence of an obligation must entail the presence of a privilege (which they call a right).

Note that our verification framework always ensures that an active obligation entails the availability of appropriate powers and privileges to fulfil it. That is, $O \Rightarrow \{R, P\}$, where R abbreviates power and P abbreviates privilege.

Moreover, powers may be used by a contract enactor independently of there being an active

obligation prescribing the (institutional) action associated with the power. We consider this to be a wholly appropriate notion. (Note that we disallow this dispensation for fixed models, for reasons of decidability in verification). Arguably, for every active R, there should be active P relations (and possibly further R relations) that enable the exercising of the power. That is, $R \Rightarrow \{R, P\}$. This is a moot point, philosophically. Our verification approach does not check it, although it could be made to do so.

In contrast, we would not consider it appropriate to check for the presence of active R relations for any active P relations so that, in some sense, every P relation would have some meaning institutionally. Rather, we consider it to be appropriate that agents are permitted to carry out brute actions which are not necessitated to have an institutional effect.

9.4.5 TransferProperty – A Simple Example Pertaining to the Transfer of Property

We now present a simple example of our IWM-based approach to contract modelling. We consider two versions of a contract and, for each, a single protocol fragment with no auxiliary normative relations suffices for its representation. The example is based on an excerpt from [23], where Brady seeks to exemplify the distinction between power and right. Brady writes:

> [I]t is a mistake to think, as some have done, of a power as a lesser or limited right. In some cases, it is *more* advantageous to have a power rather than a right. As an example, take the position of a vendee in regard to a conditional sales contract of personal property. Suppose that all but the last instalment has been paid. When the last instalment becomes due what is the vendee's legal interest in regard to the property? There is a significant difference in analysing his interest as a power to have title to the property passed as opposed to a right to have title passed. If the vendee *only* has a right to ownership of the property, the vendor is under a duty to confer title. Thus the vendor could return the previous instalments, renege on the contract, and the vendee would have to sue the vendor for breach of duty in order to get the title. On the other hand if the vendee has the power, by paying the last instalment, to acquire title, the vendor can do nothing to prevent the title from passing. In this example, the power to acquire title by paying the last instalment is a much more advantageous legal relation than having only the right to have title passed.

For this example, we can represent both scenarios using the same Theodore-based planning domain. The purpose of the contract in either scenario is to transfer ownership of the title for the given property. Let's call the initial task in the planning domain `TransferProperty`. There is a power involved in both scenarios relating to this task, otherwise, the transfer would not be effected. But what *counts as* realising this legal change?

In the case that the vendee lacks power to acquire the contract, it is the vendor who has power to dispense it. Let's say that three instalments need to be made (by the vendee) for the vendor to be under a duty to transfer the title. The planning domain (in this case) might contain a method which decomposes the task `TransferProperty` into the network: `Seq(MultiSeq(3)(Pay), TransferTitle)`. This network *counts as* the vendor transferring (the title) of the property. Instances of the `Pay` task are obligations bearing on the vendee to pay. Each `Pay` instance would

be decomposed according to one or more domain constructs. These could be methods/complex operators such as SendCheque *counts as* Pay, or EFT *counts as* Pay, say. The task TransferTitle is an obligation bearing on the vendor to transfer the title and would be decomposed by a number of domain constructs.

The Theodore domain for this scenario might thus be constructed as follows[4]. We call this contract, the *no power* contract[5], for convenience.

- Initial task: TransferProperty.

- Method: Seq(MultiSeq(3)(Pay), TransferTitle) counts as TransferProperty.

  (Pay on vendee, TransferTitle on vendor)

- Complex Operator: SendCheque counts as Pay.

  (Pay on vendee, SendCheque on vendee)

- Complex Operator: EFT counts as Pay.

  (Pay on vendee, EFT on vendee)

- Complex Operator: SendSignedTransfer counts as TransferTitle.

  (SendSignedTransfer on vendor, TransferTitle on vendor)

In the scenario that the vendee is empowered to acquire the title to the property, the Theodore domain might be constructed as follows. Note that there is no actual action required of the vendor. As soon as the vendee has made the three payments, (the title to) the property is transferred. We call this contract, the *power* contract, for convenience.

- Initial task: TransferProperty.

- Method: MultiSeq(3)(Pay) counts as TransferProperty.

  (Pay on vendee)

- Complex Operator: SendCheque counts as Pay.

  (Pay on vendee, SendCheque on vendee)

- Complex Operator: EFT counts as Pay.

  (Pay on vendee, EFT on vendee)

In Sections 11.2.2 and 11.2.3, we show examples of verifying the soundness of these contracts using the Theodore verification, planning and enactment engine. For instance, we show that in the first contract, completion of the MultiSeq, containing three occurrences of the Pay activity, is not enough to effect transfer of the property, whereas for the second contract it is.

---

[4]In the following, MultiSeq(3)(Pay) effects three instances of the Pay activity, *in sequence*.

[5]I.e. no power on the vendee.

9.4.6 Further Comments Regarding Example of Mail Service Agreement

In our previous representation of the example mail service agreement, presented in Section 9.3.1, we proposed `PayForService` as the name of an action that needs to be carried out by *SC*. In an IWM-based representation of this agreement, we could characterise the action `PayForService` as follows.

Firstly, we would model it as a workflow fragment corresponding to a `Theodore` model/planning problem. The model would specify just this task as its initial task, and, when enabled (i.e. set `Running` in `Liesbet`-speak), would correspond to an obligation obtaining on *SC* to pay for service. We consider that the task would be an institutional action, meaning that there would exist powers (as methods and complex operators) and privileges (as operators, and contained within complex operators), specified as part of the `Theodore` model, bearing on *SC*, so that *SC* may fulfil the given obligation.

Also consider the contract excerpt: *If SC fails to pay within the given time, SP may terminate the mailbox service without notice.* The term *may* here implies a power on the part of *SP* to terminate the agreement. We could model this power as a decomposition relation (i.e. method) that becomes available for use (i.e. is enabled) once *SC* has failed to pay.

This is not an obligation on the part of *SP*; indeed, *SP* may elect not to exercise it. As already described, we allow contract enactors to apply powers – they may query their existence through a "what may I do next?" query-interpreter – in the absence of a task within the contract model that prescribes an associated obligation to which the power would apply.

Within the contract model, there may exist the definition of a method:

Method: `Seq(MultiSeq(3)(SendWarnings), TerminateService) counts as TerminateAgreement.` `TerminateAgreement` on *SP*.

When such a method becomes enabled (i.e. is available for use), within the enactment of a contract model, *SP* (as specified in the definition of the method) may terminate the agreement, but not before sending three warnings to *SC* first. Thus, to exercise this power, *SP* must further decompose and enact the workflow given on the left-hand side of the method.

9.5 Concluding Remarks

We have shown in this chapter how the work that we have carried out concerning the modelling of traditional and flexible workflow may be reused, by explicating an *institutional perspective* for workflow. In defining the notion of *Institutional Workflow Modelling* (IWM), we identify the institutional concepts of *counts as* and *permission*, and the related classification of actions into *institutional* and *brute* classes of action, to be pertinent to the characterisation of workflow. These concepts are also pertinent in normative and contract modelling (NCM), and our experience shows IWM to be useful as a foundational basis for NCM.

Thus, it is through IWM that we link artefacts inherent in workflow to normative concepts used in contracts, and propose a means of reusing our work on workflow. We define IWM to be the sum of our `Theodore`-based approach to flexible workflow modelling and the presented correspondences of *counts as* and *permission* relations to workflow artefacts, on the one hand, and `HTN`-based planning constructs (i.e., methods, operators and complex operators), on the other.

When IWM is applied in the modelling of contracts, *counts as* provides a means of modelling power, and permission provides a means of modelling privilege (in the terminology of Hohfeld). Jones and Sergot [63] identify the correspondence between *counts as* and power, in respect of *counts as* relations prescribing ways in which powers may be exercised. The question of how powers should be exercised is arguably the most important aspect of this normative concept. Obligation is modelled by leaf activities within IWM model fragments, which may pertain to institutional or brute actions that demand the presence of powers and privileges (as methods and operators, respectively) to refine them.

A particularly interesting aspect of our approach to contract modelling is it relates the fulfilment of obligations directly to the existence of powers and privileges, in providing a mechanism by which contract enactors may query and plan obligation fulfilment using these relations. The distinction between institutional and brute actions in the modelling of contracts, and thus the distinction between power and privilege, is often overlooked in the modelling of contracts (see, for example, [82, 115]).

For verification, we make a distinction between *fixed* and *variable* contract models. A necessary condition for a model to be fixed is that the set of decomposition relations, described therein, is fixed. This is a condition that carries over from our work on flexible workflow modelling. Another necessary condition is that a power may not be exercised in the absence of an obligation that prescribes the institutional action to which the power applies.

For fixed contract models, a contract author or contract party (in enactment) may make use of the IWM-based verification facilities for soundness (i.e. completion along all enactment paths) and arbitrary properties expressed in a constraint language, such as a temporal logic like CTL*. For both fixed and variable models, the framework enables a contract party to plan obligation, fragment and contract fulfilments according to subjective constraints and to perform what-if simulation.

In neither of our (non-IWM and IWM-based) approaches on contract modelling, do we include any built-in support for specifying a theory of normative concepts and their inter-relationships, other than that which is a by-product of Institutional Workflow Modelling, which gives a means by which the fulfilment of obligations may be specified and derived. For IWM-based models, we verify that the existence of an obligation entails the existence of sufficient powers and permissions for their fulfilment.

The utility of an IWM-based approach to contract modelling is evident from both examples given in the chapter, namely, the `TransferProperty` and mail service agreement examples. We propose a hybrid approach to contract modelling, where a contract is modelled as a number of IWM-based workflow fragments along with a set of auxiliary normative relations. Other research contributions focus on one or the other, whereas we argue that a hybrid approach such as ours is a more natural way of viewing and modelling contracts. Although we have not particularly emphasised the role of auxiliary normative relations in the discussion in this thesis, it is clear that some means of supporting the modelling of additional normative concepts such as *prohibition*, *entitlement* [92], and others, would be of utility. As part of accommodating the definition and representation of auxiliary normative relations, we could extend our framework to account for theories of normative concepts that may be identified as being desirable.

In future work, we need to give our approach to contract modelling a comprehensive road-test against a number of different sorts of contracts in order to identify any weaknesses in our modelling,

verification and planning approach. However, we feel that our IVVM-based approach is a significant improvement over our previous work on contract modelling in directly supporting the modelling of protocol-like artefacts in contracts. In the next chapter, we give an overview of how we have implemented the authoring, verification and enactment frameworks for `Liesbet` and `Theodore`.

Chapter 10

Implementation

In the following chapter, we present a concise overview of what we have implemented in the course of our work. As van der Aalst and colleagues argue [123] "any proposed language should be supported by at least a running prototype in addition to a formal definition". We are of the same opinion, and thus considered it essential to provide such a framework.

We start with a brief presentation of the *Eclipse Modelling Framework* which provides persistence functionality, and follow that with an overview of the structure of our implementation. Then, we go through each of the components in our framework, in turn: the `Liesbet` verification and enactment engine, the `CTL*` constraint checking engine, and the `Theodore` verification, planning and enactment engine.

10.1 Eclipse Development Platform and Eclipse Modelling Framework

We have used the *Eclipse Development Platform* (EDP) [3] to facilitate a Java-based implementation of verification and enactment engines for `Liesbet` and `Theodore`. It is an Integrated Development Environment (IDE), which provides a number of useful features, including support for test writing.

The *Eclipse Modelling Framework* (EMF) [4] is a modelling and code generation facility, which comes as part of the EDP. The key features of EMF that motivated its use are: its support for UML-like class and relationship modelling, its code-generation capability and its support for persistence. We have used the facility within EMF for the definition of class models to define a number of meta-models for `Liesbet` and `Theodore`. From these class models, EMF is able to generate a collection of Java-based APIs (Application Programming Interfaces) which may be used to traverse instances of these models stored in memory. The persistence support within EMF makes it possible to save models which have been created using the generated APIs, and load them back in for traversal using the APIs. It thus provides a model-specific way of loading and storing data to/from file, in a way that is hidden from the programmer. EMF also provides (extensible) support for the graphical authoring of instances of class models.

10.2 Structure of Liesbet/Theodore Framework

The structure of the Liesbet/Theodore verification, planning and enactment framework is modular in nature. The following modules exist in the complete framework.

- Liesbet Workflow Verification and Enactment Engine.
- Theodore HTN-based Planner.
- CTL* Temporal Constraint Engine.
- Service Selection Engine.
- Knowledge Base.

Most of these modules have a class model associated with them for describing configuration instances of the given modules. For example, the Liesbet module has a class model for describing Liesbet models, the Theodore module has a class model for describing Theodore planning problems and the CTL* module has a class model for specifying CTL* formulas.

The Theodore class model, shown in Figure 10.1, is the core class model in the framework, inherited by all other class models. It serves two purposes. Firstly, it enables the specification of Theodore planning domains; and thus specifies a number of interfaces that (typically) need to be implemented, such as: Workflow (for the specification of the task network – e.g., Liesbet-based – used in planning), ConstraintChecker (for the progression-based constraint engine), ServiceSelector (for the service selection engine, responsible for prioritising the use of HTN domain constructs) and KnowledgeBase (for the particular knowledge base instance, primed with its initial state). Secondly, it serves as a repository for these interfaces, in the event that planning is not used. For instance, we can run a Liesbet verification instance, without the use of Theodore per se; but some aspects of the problem will be described using classes that are a part of the Theodore class model, such as Workflow.

We now describe the implementation of each of these modules in detail, starting with the Liesbet verification and enactment engine.

10.3 Liesbet Workflow Verification and Enactment Engine

The class model for the Liesbet Workflow Engine is presented in Figure 10.1. The LiesbetWorkflow class extends the Workflow class, defined in Theodore's class model. This means that it inherits the capability of specifying a collection of ConstraintChecker engines. LiesbetWorkflow specifies the root activity (Activity) of the workflow model and specifies a set of AbstractActionType (i.e. ISA) hierarchies, collects together all of the activity and query definitions in the workflow model, and facilitates the specification of a number of synchronisation rules pertaining to the model. Activity is a base class to many other Liesbet activity-related classes. It captures the join and transition condition types of an activity type, whether the type is isolated or not, and the customised type name, or ctype, of an activity. As can be seen, there are a number of other classes, which are largely self-describing.

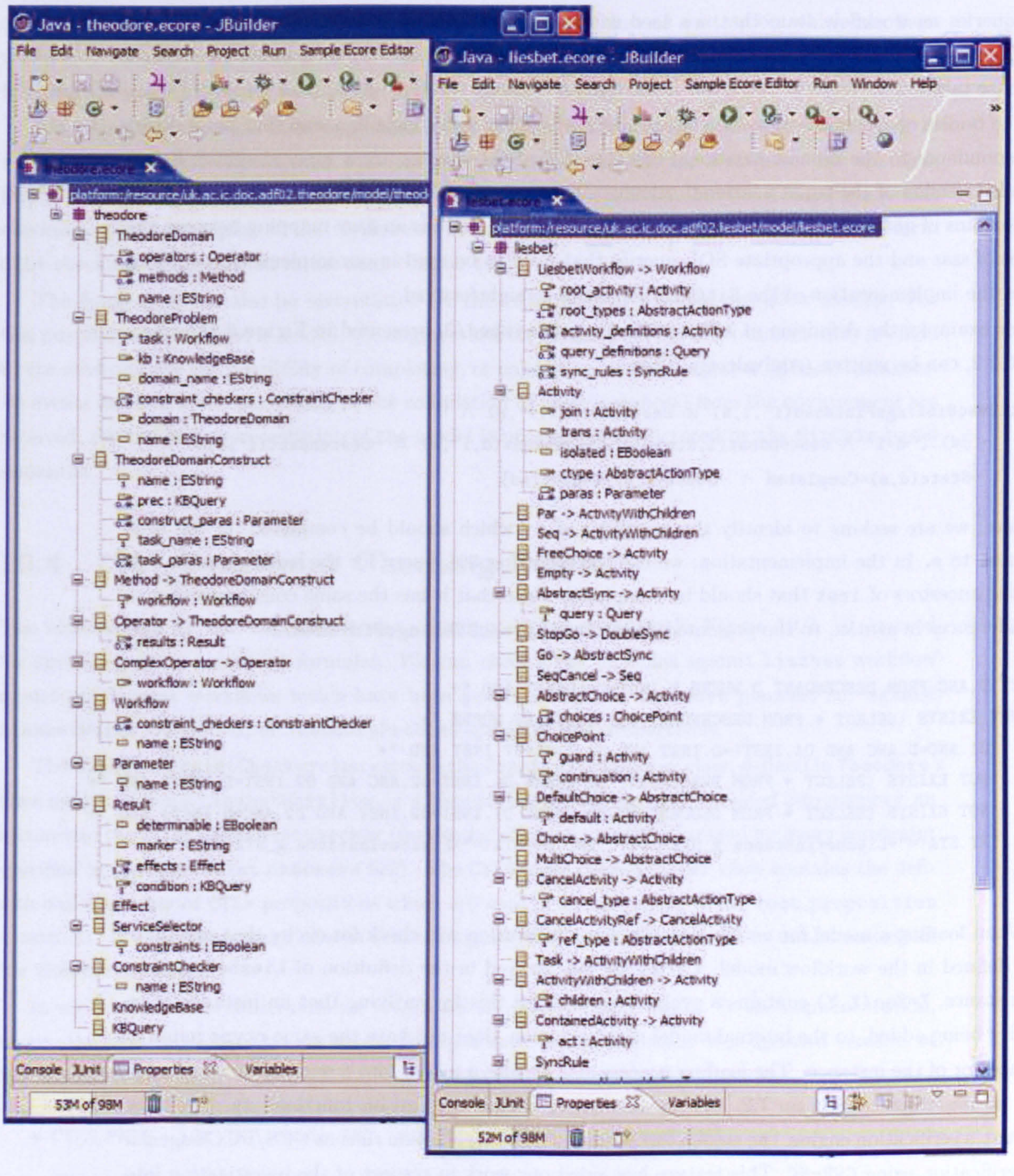


Figure 10.1: `Theodore` (left) and `Liesbet` Class Models.

For verification, a Liesbet model is loaded into a number of *MySQL* database tables. Our SitCalc-based characterisation for Liesbet naturally lends itself to a relational database implementation. The relational and functional fluents, which are updated according to the successor-state axioms (presented in Section 6.2), are implemented directly as a number of database tables. The queries on workflow state that are used within synchronisation activity types, and also the activity (and optional reference) types used within CancelActivity* types, are also stored in a database table.

The bodies of successor-state axioms are specified in first-order logic in a way that has a direct correspondence to the domain relational calculus [31]. *SQL* queries are a sugared-syntax for an extended version of the tuple relational calculus. The two calculi are very similar in nature – just the domains of quantification are different. This means that there is an easy mapping between the bodies of ssas and the appropriate SQL queries that need to be used in our implementation. This makes the implementation of the SitCalc semantics straightforward.

For example, the definition of AllDescSiblingsFinished/3, presented in Figure 6.11 of Section 6.2.2, can be written (equivalently) thus:

```
AllDescSiblingsFinished(i',i,s) ≡ Descendant(i,i',s) ∧
   ¬(∃d).[¬d=i' ∧ Descendant(i,d,s) ∧ ¬Descendant(d,i',s) ∧ ¬Descendant(i',d,s) ∧
    ¬State(d,s)=Completed ∧ ¬State(d,s)=Cancelled]
```

Here, we are seeking to identify the ancestors of i' which should be completed, in the next situation to s. In the implementation, we use the following SQL query for the same purpose – to find the ancestors of inst that should be completed. Note that it has the same construction, save for differences in syntax, to the presented definition AllDescSiblingsFinished/3.

```
SELECT D.ANC FROM DESCENDANT D WHERE D.INST="+inst+" AND "+
   "NOT EXISTS (SELECT * FROM DESCENDANT D1, STATE ST WHERE "+
      "D1.ANC=D.ANC AND D1.INST!=D.INST AND D1.INST=ST.INST AND "+
      "NOT EXISTS (SELECT * FROM DESCENDANT D2 WHERE D1.INST=D2.ANC AND D2.INST=D.INST) AND "+
      "NOT EXISTS (SELECT * FROM DESCENDANT D2 WHERE D1.INST=D2.INST AND D2.ANC=D.INST) AND "+
      "ST.STA!="+LiesbetInstance.g_STA_CAN+" AND ST.STA!="+LiesbetInstance.g_STA_COM+
   ")"
```

When loading a model for verification, the implementation will check for cycles that may have been defined in the workflow model. Cycles are not allowed in the definition of Liesbet models. For instance, X=Seq(X,Y) contains a cycle. We check for this by verifying that an instance of an activity being added, to the internal model representation, does not have the same ctype name as an ancestor of the instance. The loading process also divides a model into a number of verification runs, as described in Section 7.2. As an alternative to the verification functionality offered by Liesbet's verification engine, the engine can instead output verification runs as CCS/PCCS agents, for verification using CWB-NC. This feature has aided our work in respect of the investigation into the utility of CCS/PCCS for providing semantics, and verification support, to Liesbet.

In performing verification, we maintain a number of state tables – one for each number of activity instances in an evolving model. If the number of instances in a model stays the same throughout its enactment, which will be the case if the model does not make use of non-limited multiple activity instance types, a single state table will be used.

Each row of a state table pertains to a single state of the workflow model being verified. Each field corresponds to the state of an individual instance in the model. The state is represented by a numerical value (e.g. 0 for `Ready`, 1 for `Running`, and so on). When we come to evaluate whether a new situation (according to the verification algorithm, presented in Section 7.4) is a *matched* state (i.e. a situation whose activity instance state is the same as that of a situation which has been previously visited in the verification process), we simply check whether there is a row in the table that has the same values for each of the activity instance fields.

The algorithm that has been implemented for verification is that presented in Section 7.4. A verification run can be configured with one or more constraint checker instances. For the time being, the only constraint checker that the framework supports is one for CTL*-based constraint checking. Many examples of the verification of `Liesbet` models, including runs which perform CTL* checking, are presented in the next chapter (11).

The framework can also be instantiated for the enactment of `Liesbet` workflow models. For this purpose, once a model is loaded, the engine waits for events from the environment and/or offers to the environment the possibility of completing, or cancelling, one of a number of basic instances. As events (such as those pertaining to the completion of basic instances) from the environment are received, the internal representation of the model is progressed, as dictated by the `SitCalc`-based semantics.

10.4 CTL* Constraint Checking Engine

The class model for the CTL* constraint checking engine is shown in Figure 10.2. It is used for specifying CTL* constraint formulas. We can check CTL* formulas against `Liesbet` workflow models, or against workflows which have been generated by the `Theodore` planner, i.e. against `LiesbetWorkflow` models, or `TheodoreProblem` specifications, respectively.

The `CTLSConstraintChecker` class extends the `ConstraintChecker` class, defined in `Theodore`'s class model. When a `LiesbetWorkflow`, or a `TheodoreProblem`, instance is loaded into memory, an instance of the `CTLSConstraintChecker` (implementation) class will be created for every constraint specified in its `constraint_checkers` field. The `CTLSConstraintChecker` class contains the definition of a number of CTL* propositions which are used in the definition of the `root_proposition` constraint. It is this constraint that is checked against the `LiesbetWorkflow`, or in the context of the `TheodoreProblem`.

In verification, CTL* constraints are *progressed* through workflow states. In the implementation, we use the following (*MySQL*) database tables for maintaining the state of progressed versions of the original constraint:

- `CTLS_PROP – idx INT, code INT, nt INT, prop1 INT, prop2 INT`

  This table maintains the propositions that make up the composite temporal constraint, to be verified. `idx` is a table index, `code` captures the type of proposition (*some paths*, *or*, *next*, etc.), `nt` determines whether the proposition is under negation, `prop1` and `prop2` are indices of sub-propositions.

- `CTLS_EVAL_REC – idx INT, prop INT, res INT, st_idx INT`

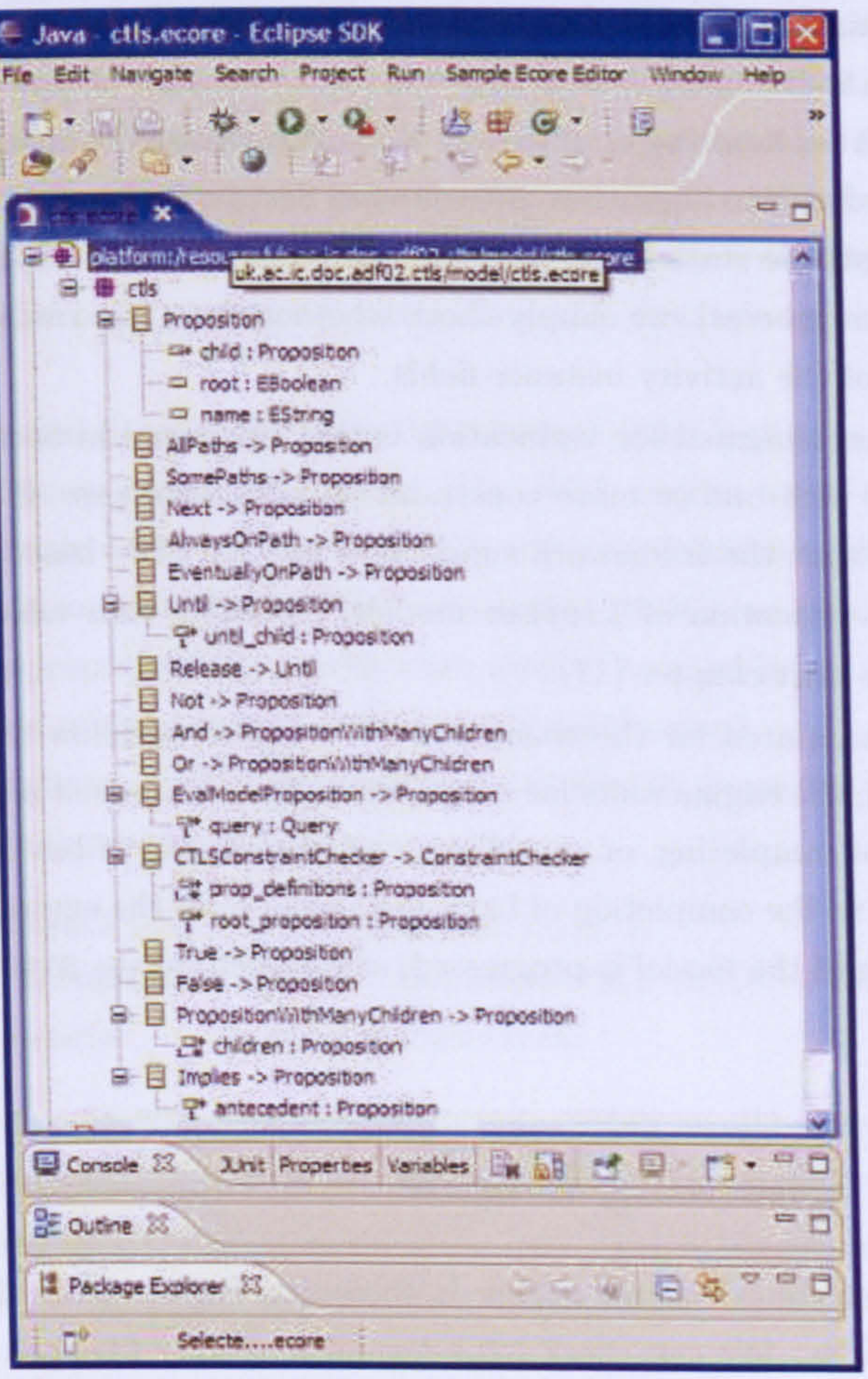


Figure 10.2: CTL* Constraint Checker Class Model.

This table maintains the progression of the composite temporal constraint for various states of enactment. `idx` is a table index, `prop` is the pertaining proposition in `CTLS_PROP`, `res` captures the current result of the proposition (e.g. `TRUE`, `FALSE`, `UNDEF`, ...), `st_idx` captures the state to which the proposition pertains.

- `CTLS_EVAL_CHILD` – `pidx INT, idx INT, st_idx INT`

  This table maintains the parent, child relationship between progressed formulas; where `idx` is the index within `CTLS_EVAL_REC` of the formula representing the progression of the formula (given by `pidx` in `CTLS_EVAL_REC`) to the current state (`st_idx`).

As part of the initialisation of the verification process, we translate a given CTL* constraint into one which uses only the primitive symbols that we have decided for CTL*, as described in Section 7.3. That is to say, we replace any abbreviations with their respective expansions; and repeatedly do so until the formula no longer uses any abbreviations. The formula, after this processing, should only make use of: $\neg$, $\vee$, $\top$, $\bot$[1], `E`, `X` and `U`.

[1] We allow $\bot$ as it is trivially supported in the implementation.

A CTL* constraint may be conceptualised as a tree of the atomic propositions, as leaves, which are recursively combined using logical symbols, to form the inner-nodes of the tree. In pre-processing a CTL* constraint, we start at the root of this tree and work out to the leaves. At any inner-node within the tree, we process the formula at that node according to the logic symbol being used to combine the children of the node, or if it is a leaf, according to the nature of the atomic proposition.

We load the translated CTL* constraint into the CTLS_PROP table, starting at the root node of the conceptualised proposition tree. A node (leaf, or internal) may specify a proposition which has already been loaded into CTLS_PROP, as part of pre-processing another node. In this circumstance, the same CTLS_PROP entry will be used, and further processing along the branch of the proposition tree stopped.

In pre-processing a node, we strip off leading negations. For every negation, we toggle whether the proposition is *under negation or not.* We maintain a record *in memory* of the indices of records, in CTLS_PROP, corresponding to propositions both not under, and under, negation. For the proposition that is exposed after stripping off leading negations, we check the memory records for whether the proposition has been previously loaded (given whether it is not under/under negation). If it has already been processed, we return the index of the pertaining record in CTLS_PROP to the logic processing the parent node. If it has not already been processed, we insert a new record into CTLS_PROP, as now described. Having processed the proposition, we update the memory record for the proposition, by assigning the index (of the new table record) to the appropriate "not under/under negation" field. We distinguish the processing of a proposition on the basis of its type.

- Atomic query formula – The new record contains: a code stipulating that it is an *atomic query formula*, whether it is not under/under negation (as nt), the particular query formula used (as prop1), e.g. Completed_act, and the customised activity type name of interest to the query (as prop2).

  The atomic queries that are supported for Liesbet verification are those presented in Section 7.3. These are queries on activity state, with the general forms: *State*_act and *State*_all; where *State* is one of Completed, Cancelled, Finished, Running, or Initial. Queries making use of reference types are not allowed.

  In the context of planning with Theodore, we also support arbitrary querying against current knowledge base state, within the limits of bounded quantification [17].

- Some paths (EΦ) (resp. Next, XΦ) – We process the proposition Φ, which will yield an index into CTLS_PROP for the proposition. The new record, then, has a code stipulating that it is a "some paths" (resp. "next") proposition, whether it is not under/under negation (nt), and the index of the child proposition Φ (prop1).

- Or $(\Phi_1 \vee \ldots \vee \Phi_n)$ – We process the propositions $\Phi_1 \ldots \Phi_n$, which will yield indices into CTLS_PROP for the propositions. The new record, then, has a code stipulating that it is an "or" proposition, and whether it is not under/under negation (nt). We then also create a number of supplementary records to record the child indices. These have their code fields set to reflect their association with an "or" proposition, with nt set to FALSE, prop1 set to the index of the parent "or" record, and prop2 set to one of the child indices.

$prog(s,\Phi,\top) = \neg prog(s,\Phi)$

$prog(s,\Phi,\bot) = prog(s,\Phi)$

$prog(s,\bot) = \bot$

$prog(s,\top) = \top$

$prog(s,\phi) = eval(p,s)$

$prog(s,E\Phi) = prog(s,\Phi)$

$prog(s,X\Phi) = next(s,\Phi)$

$prog(s,\Phi_1 \vee \ldots \vee \Phi_n) = prog(s,\Phi_1) \vee \ldots \vee prog(s,\Phi_1)$

$prog(s,\Phi_1 U \Phi_2) = \text{if } final(s) \text{ then } prog(s,\Phi_2)$
$\text{else } prog(s,\Phi_2) \vee prog(s,\Phi_1) \wedge next(s,\Phi_1 U \Phi_2)$

where `final(`s`)` determines whether s corresponds to a state where all activity instances in the workflow model have finished, `next(`s`,` Φ`)` stipulates that the proposition Φ should be progressed through any *subsequent* state to s and `eval` evaluates p, being an atomic query formula, against the current state s.

Table 10.1: Definition of `prog/3` and `prog/2`, for Progression of CTL* Propositions Through States.

- Until ($\Phi_1 U \Phi_2$) – We process the propositions Φ_1 and Φ_2, which will yield indices into `CTLS_PROP` for the propositions. The new record, then, has a `code` stipulating that it is an "until" proposition, whether it is not under/under negation (`nt`), and the indices of the child propositions Φ_1 (`prop1`), and Φ_2 (`prop2`).

An exception to this processing is if the child proposition that is processed is simply $\top$ (or $\bot$) prefixed with zero or more negations. In this case, we strip off the negations, toggling the truth value. We then use special index codes, which are taken to mean `TRUE`, or `FALSE`, in the parent table record being inserted.

We also check that the temporal constraint being checked is a state formula, and not a path formula, which is a necessary constraint as we wish to verify CTL* state formulas against the initial state of `Liesbet` models.

When carrying out the verification process, we use the function `prog/3`, as defined in Table 10.1, to progress constraints from the last state. If the current state is the initial state of the workflow model, we create a new entry in `CTLS_EVAL_REC` for the initial temporal constraint. We then apply `prog/3`, which will have the side-effect of creating further entries in `CTLS_EVAL_REC`, with their dependencies reflected in `CTLS_EVAL_CHILD`. Note that, if we come across a proposition that has already been progressed in the current state, we do not progress this particular instance of the proposition any further, but we do update `CTLS_EVAL_CHILD` to reflect that a further parent index is to be related to any evaluation result for the (already progressed) proposition. Also, if the said proposition has already had a result assigned to it, this result is propagated upwards, as described in subsequent paragraphs.

The function `prog/3` is used during the verification procedure, where the last argument indicates whether the second argument, Φ, is under negation. If it is (indicated by $\top$), then `prog/3` negates the progression of Φ, determined by `prog/2`.

In applying `prog/2`, when we reach:

- $\bot$, or $\top$, we propagate the result up the proposition tree, as explained below.
- An application of `eval`, the proposition will be an atomic query formula. We evaluate the formula and propagate the result up the proposition tree.
- An application of `next`, the proposition needs to be evaluated against the next state so we stop progression (along this branch of the proposition tree) for the current state.

For states other than the initial state, we apply `prog` on the propositions that were under an application of the function `next` in progressing the previous state.

In progressing a proposition, in a state, whenever we evaluate an atomic query formula, or reach a simple truth value ($\top$, or $\bot$), we need to propagate the result back up the proposition tree, according to the reverse of the progression function `prog/2`, taking into account whether propositions are under negation, or not. To get the index of a parent in `CTLS_EVAL_REC`, in order to record a result for it, we inspect the record for the child index in `CTLS_EVAL_CHILD`.

When we backtrack, according to the verification algorithm described in Section 7.4, we may come across *matched states*, where the instance state (or, domain state of interest) in the current situation is identical to that of a previous situation that we have already visited. In this case, any propositions that are to be progressed (in the current situation), or that are elicited through further progression, and which have already reported a result for the particular state, previously, may simply have that result propagated upwards.

When $\top$ (resp. $\bot$) is propagated to the initial state, we are able to declare a result for the constraint checking, i.e. that the constraint is satisfied (resp. violated).

Also note that when we have finished traversing all of the paths which lead out of a state, which has a "some paths" proposition associated with it, if the proposition is yet to record a result, then this means that its contained proposition has not been satisfied. In this case, we need to record a result of `FALSE` against this proposition, and propagate results upwards, accordingly.

10.5 Theodore Verification, Planning and Enactment Engine

The class model that we have defined for `Theodore` is presented in Figure 10.1. It is largely self-describing and corresponds closely to the formalisation of the `Theodore` HTN planner, described in Section 8.2.2.

A `Theodore` planning problem is described as an instance of `TheodoreProblem`, which defines the initial task network of the problem (`Workflow`), knowledge base (`KnowledgeBase`), constraint checkers (`ConstraintCheckers`) and the planning domain (`TheodoreDomain`). A planning domain consists of a number of domain constructs, namely instances of `Operator` (including `ComplexOperator`) and `Method`. The common base class for constructs is `TheodoreDomainConstruct`, which defines a precondition (`KBQuery`) on the use of the construct, and parameters used in the construct. `Method` additionally specifies a `Workflow`, to which a task may be decomposed. An `Operator` instance may specify a number of effects (`Results`). A `ComplexOperator` extends `Operator`, and also defines a workflow.

The planning algorithm repeatedly seeks to apply constructs until it finds a plan which effects the initial task network (or, in the case of verification, until it has identified that all partial

decompositions successfully complete). A `ServiceSelector` instance determines which constructs are preferred, at a particular point in the decomposition process. The `Theodore` engine also provides an enactment mechanism, where a plan may be enacted and *what-if* simulation may be performed. What-if simulation allows a domain controller or expert (see Section 8.1.3) to try out particular decompositions, in order to see what plans are available to complete a given a workflow. In doing this, they help to guide the planning process.

10.6 Service Selection Engine

An instance of `ServiceSelector` must be specified for a `Theodore` planning problem. Its purpose is to control how domain constructs are applied in planning, determining a preference order over their application. The default class (`BasicServiceSelector`), implemented for the `Theodore` framework, simply applies constructs in the order they are specified in a `TheodoreDomain`. This behaviour may be changed by implementing a different `ServiceSelector`.

10.7 Knowledge Base

An instance of `KnowledgeBase` effects the knowledge base associated with a `Theodore` planning problem, if extant. It is responsible for evaluating the preconditions of domain constructs given parameter bindings, returning any new bindings, and for applying the effects of using constructs (specifically, operators and complex operators). It must also be capable of being backtracked.

When a `Theodore` problem is specified not to use an explicit knowledge base, then domain state – just pertaining to the enablement of domain constructs – is maintained within the `Theodore` planning implementation itself. Finally, the default implementation for the `KnowledgeBase` is SQL-based, which means that precondition and effects clauses must be specified in SQL, by default.

The `Liesbet` verification and enactment engine does not make use of a separate `KnowledgeBase` component, in its operation. Instead it necessarily makes use of a *MySQL* database, as described in Section 10.3.

In the next chapter, we present some examples of using our implemented frameworks for the verification of `Liesbet` (i.e. traditional) and `Theodore` (i.e. flexible) workflow models.

Chapter 11

Examples of Verification

In this chapter, we show some examples of verification of `Liesbet` and `Theodore`-based workflow and contract models. We show screen dumps of some of the authored models and of the output from the pertaining verification engine. We start with some `Liesbet` examples, and then present some `Theodore`-based examples.

11.1 Liesbet Examples

11.1.1 A Simple Workflow

We start with a simple `Liesbet` model (that we have used throughout this thesis), viz. `Par(Seq(A,B),Seq(C,D))`. This may also be written in an elaborated form, as follows.

```
Par(S1, S2)
S1=Seq(A,B)
S2=Seq(C,D)
```

A screenshot of this model as authored (using the Eclipse Modelling Framework (EMF), see Section 10.1) is shown in Figure 11.1.

The model is trivially sound according to the criteria specified in Section 7.2. If we disable the `por` verification option (which will cause the verification engine to check workflow fragments for soundness even if they are necessarily sound), we see that the model has ten distinct states (which agrees with the diagnosis presented in Section 6.2.1). As can be seen, from Figure 11.2, the verification engine finds the workflow sound. Note that a dot '.' appears after the output "`Verifying model run`" for every progressive verification step. Later on, when we present examples which include the verification of constraints, a small 'c' will appear for every progressive constraint checking step.

If we enable the `por` option, we see (in Figure 11.3) that the model is trivially passed as being sound, with no explicit verification (as indicated by a single state). The number of "por'd acts" indicates how many activity instances in the model were "ignored" – in this case, all of them are ignored.

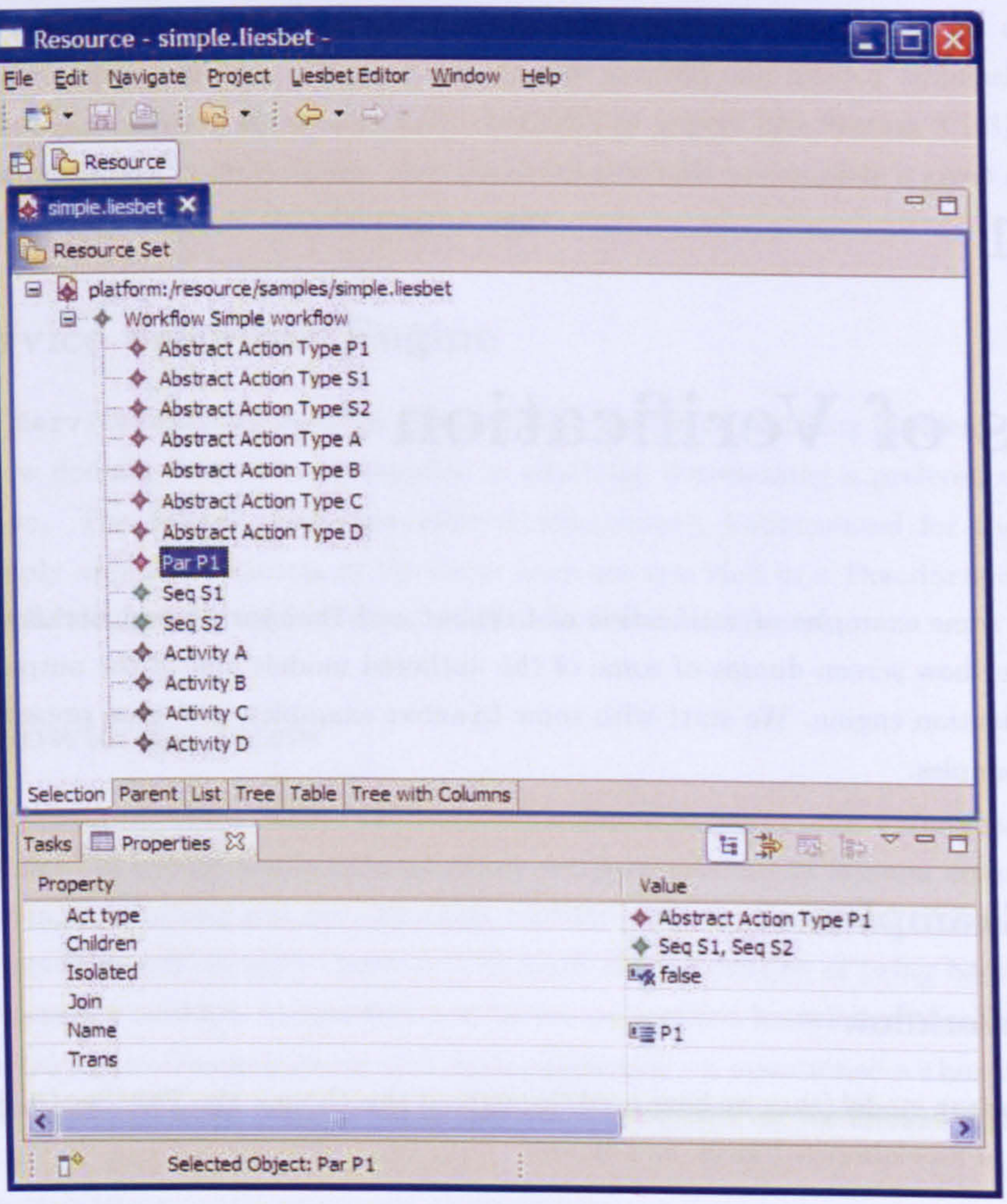


Figure 11.1: `Par(Seq(A,B),Seq(C,D))` as Authored in EMF.

```
Simple liesbet workflow...
Parsing model...
Setting up verification...
Verifying model run ...........
Filing report...
Checking for deadlock...
Checking for dead instances...
*****************************
Time taken: 0(h),0(m),0(s),410(ms)
Liesbet run 0 success (no locking, no dead insts, no violated constraints) with...
    stored states: 10, matched states: 4, por'd acts: 0.
*****************************
```

Figure 11.2: `Par(Seq(A,B),Seq(C,D))` Verified Using `Liesbet` Verification Engine.

```
Simple liesbet workflow, with por...
Parsing model...
Setting up verification...

Filing report...
Checking for deadlock...
Checking for dead instances...
*****************************
Time taken: 0(h),0(m),0(s),100(ms)
Liesbet run 0 success (no locking, no dead insts, no violated constraints) with...
    stored states: 1, matched states: 0, por'd acts: 7.
*****************************
```

Figure 11.3: `Par(Seq(A,B),Seq(C,D))` Verified Using `Liesbet` Verification Engine, with `por` Enabled.

```
Simple liesbet workflow, with sync rule, plus constraint check...
Parsing model...
Setting up verification...
Verifying model run cc.c.c.c.c.c.c.c.c.c.
Filing report...
Checking for deadlock...
Checking for dead instances...
*****************************
Time taken: 0(h),0(m),0(s),490(ms)
Liesbet run 0 success (no locking, no dead insts, no violated constraints) with...
    stored states: 10, matched states: 2, por'd acts: 0.
*****************************
```

Figure 11.4: `Par(Seq(A,B),Seq(C,D))` Verified Against a Given Constraint, with Synchronisation Rule.

```
Simple liesbet workflow, with no sync rule, plus constraint check...
Parsing model...
Setting up verification...
Verifying model run cc.c.c.c.c.
Filing report...
Failed Temporal Constraint...
******************************
Time taken: 0(h),0(m),0(s),380(ms)
Liesbet run 0 FAILED with violated constraints with...
    stored states: 7, matched states: 0, por'd acts: 0.

Failing State...
Instance: 0, State: 1, Cid:P1
Instance: 1, State: 1, Cid:S1
Instance: 2, State: 2, Cid:A
Instance: 3, State: 1, Cid:B
Instance: 4, State: 1, Cid:S2
Instance: 5, State: 1, Cid:C
Instance: 6, State: 0, Cid:D

******************************
```

Figure 11.5: `Par(Seq(A,B),Seq(C,D))` Verified Against a Given Constraint, Without Synchronisation Rule.

11.1.2 Synchronisation Rules and Constraints

We augment the previous example with a synchronisation rule, viz. `SyncRule(S2, Completed_act(A), Completed_act(S1))`. This stipulates (for the previously presented workflow model) that as soon as the (only) instance of `A` is in the `Completed` state, descendants of `S2` (namely, the instances of `C` and `D`), and `S2` itself, may not advance until the sequence `S1` (containing `A` and `B`) has completed.

In Figure 11.4, we show the verification output for the model `Par(Seq(A,B),Seq(C,D))`, augmented with the said synchronisation rule. Note the reduction in the number of matched states. In this verification run, we also verify the constraint: `AG (Completed_act(A)` $\rightarrow$ `Completed_act(C)` $\vee$ `Completed_act(S1)` $\vee$ `AX`$\neg$`Completed_act(C))`. This constraint says that in all states it must be the case that, if `A` has been completed, then `C` has completed, `S1` has completed, or in all next states from the state of interest `C` should not have completed. This constraint captures the requirement that, once `A` is completed, execution of `S1` takes precedence over completion of `C`. (We could verify that it takes precedence over execution of `S2`, and its descendants; but this constraint suffices for this example). Because of the presence of the synchronisation rule, this constraint should not be violated, as can be seen from the verification output, presented in Figure 11.4.

If we remove the synchronisation rule from the model, we should see that the constraint is violated. When a constraint violation occurs, we dump the instance state for the preceding state, from which we are progressing constraints. In the output shown, in Figure 11.5, `A` has already completed and the failure comes about in completing `C`, while `S1` has not completed.

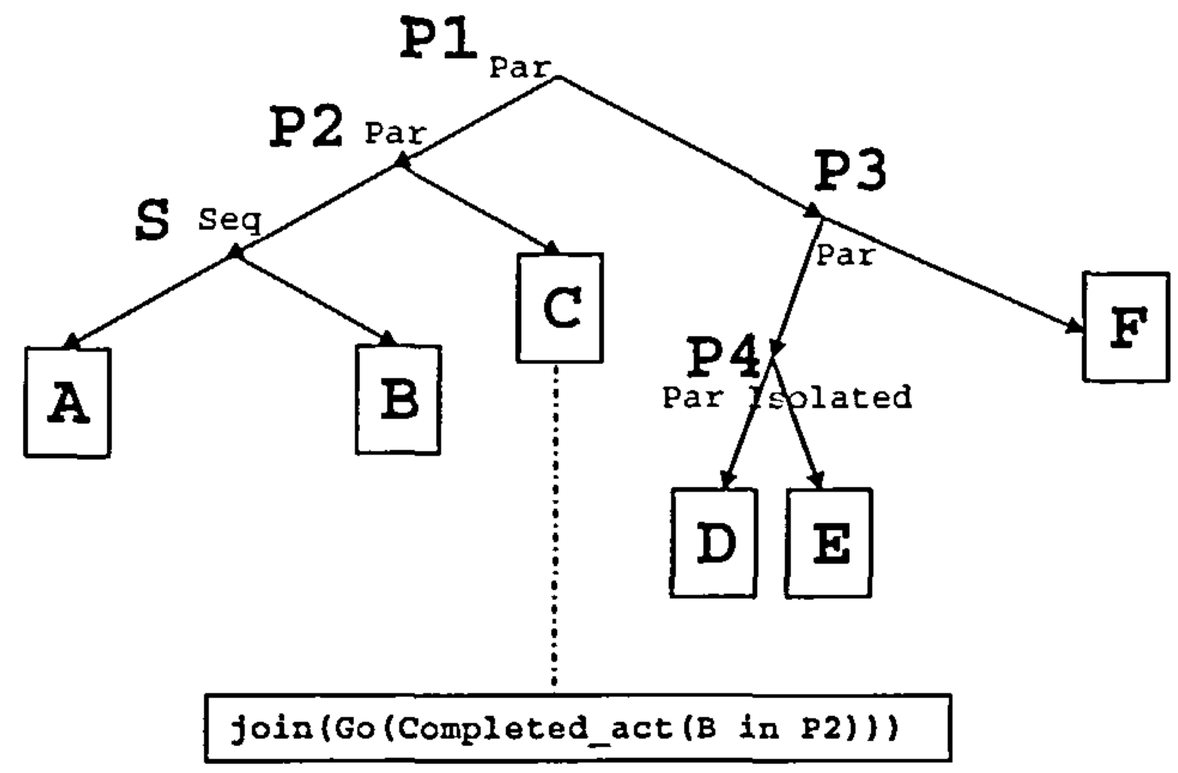


Figure 11.6: A Liesbet Model with Isolated Scope, and Potential for POR in Verification.

```
Parsing model...
Setting up verification...
Verifying model run ....
Filing report...
Checking for deadlock...
Checking for dead instances...
*******************************
Time taken: 0(h),0(m),0(s),300(ms)
Liesbet run 0 success (no locking, no dead insts, no violated constraints) with...
    stored states: 6, matched states: 0, por'd acts: 4.
*******************************
Setting up verification...

Filing report...
Checking for deadlock...
Checking for dead instances...
*******************************
Time taken: 0(h),0(m),0(s),190(ms)
Liesbet run 1 success (no locking, no dead insts, no violated constraints) with...
    stored states: 1, matched states: 0, por'd acts: 3.
*******************************
```

Figure 11.7: Output from Verifying the Model Presented in Figure 11.6, Using Liesbet Verification Engine, with por Enabled.

```
Dead activity instances model...
Parsing model...
Setting up verification...
Verifying model run ......
Filing report...
Checking for deadlock...
Checking for dead instances...
*****************************
Time taken: 0(h),0(m),0(s),280(ms)
Liesbet run 0 FAILED with dead activity instances detected with...
    stored states: 8, matched states: 0, por'd acts: 0.
Dead Instances Report Details
Instance: 8, CId: C

*****************************
```

Figure 11.8: Output from Verifying a Model with a Dead Activity Instance.

11.1.3 Simple POR Example

In this example, we show the verification output for a model which has an isolated scope and which demonstrates the possibility of having further POR applied in its verification. From Figure 11.6, we see that `P4` is an isolated scope. The verification of this activity type should occur as a separate run. We also note that the activity type `C` has a join condition on activity `B` having completed, where this condition is qualified with a reference type rooted at `P2` (see Section 3.1.3, for information about reference types). The remainder of the workflow model (save for `P4`) may be ignored, in verification, as it is necessarily sound.

Referring to Figure 11.7, we see for run 0 that there are six stored states corresponding to the evolution of `P2`, and four "por'd acts" corresponding to the rest of the model (save for `P4`), which is ignored. The whole sub-tree rooted at `P4` is necessarily sound, according to the criteria described in Section 7.2. Consequently, as the output from run 1 shows, the verification engine trivially passes it as being sound – the single state is the initial state of the model, which is always stored irrespective of whether POR is applicable. Notably, the number of "por'd" acts is three, corresponding to `P4` and its two children.

11.1.4 Dead Activity Instances

In Appendix Section A.4, we present a model which has an activity which can never be executed – a dead activity instance. The model is:

```
Par(Choice(Empty, A, Empty, B), C)
C = Act(join(Go(Finished_act(A) | Finished_act(B),
                Completed_act(A) | Completed_act(B))))
```

In the model, only activity `A` OR activity `B` may be executed. However, for the join condition on `C` to succeed, both instances must be executed and reach a `Completed` state. Thus, activity `C`

```
Deadlock check.  Should exhibit deadlock.
Parsing model...
Setting up verification...
Verifying model run ..
Filing report...
Checking for deadlock...
*******************************
Time taken: 0(h),0(m),0(s),350(ms)
Liesbet run 0 FAILED with deadlock detected with...
    stored states: 4, matched states: 0, por'd acts: 0.
Instance: 0, State: 1, Cid:P1
Instance: 1, State: 1, Cid:S1
Instance: 2, State: 2, Cid:A
Instance: 3, State: 1, Cid:JOIN_SEC_B
Instance: 4, State: 1, Cid:BJoin
Instance: 5, State: 0, Cid:B
Instance: 6, State: 0, Cid:C
Instance: 7, State: 1, Cid:S2
Instance: 8, State: 2, Cid:D
Instance: 9, State: 1, Cid:JOIN_SEC_E
Instance: 10, State: 1, Cid:EJoin
Instance: 11, State: 0, Cid:E
Instance: 12, State: 0, Cid:F

*******************************
Deadlock check.  Should NOT exhibit deadlock.
Parsing model...
Setting up verification...
Verifying model run ................
Filing report...
Checking for deadlock...
Checking for dead instances...
*******************************
Time taken: 0(h),0(m),0(s),580(ms)
Liesbet run 0 success (no locking, no dead insts, no violated constraints) with...
    stored states: 15, matched states: 5, por'd acts: 0.
*******************************
```

Figure 11.9: Output from Verifying a Model with a Source of Deadlock, and a Variant with the Deadlock Removed.

```
Travel agent example...
Parsing model...
Setting up verification...
Verifying model run ..........................................................
..............................................................................
..............................................................................
..............................................................................
.............................................................
Filing report...
Checking for deadlock...
Checking for dead instances...
*******************************
Time taken: 0(h),0(m),42(s),540(ms)
Liesbet run 0 success (no locking, no dead insts, no violated constraints) with...
    stored states: 286, matched states: 88, por'd acts: 0.
*******************************
```

Figure 11.10: Output from Verifying a Liesbet Representation of the 3rd Travel Agent Example from Section 4.5.

---

will never (under any enactment path) be executed. It is an example of a dead activity instance. In Figure 11.8, we see that the Liesbet verification engine identifies this structural flaw.

11.1.5 Deadlock

In Appendix Section A.4, we present two models, one having a source of deadlock and a variant which is free from deadlock, viz.

```
Par(Seq(A, B, C), Seq(D, E, F)
B = Act(join(Go(Completed_act(E))))
E = Act(join(Go(Completed_act(B))))

Par(Seq(A, B, C), Seq(D, E, F)
B = Act(join(Go(Completed_act(E))))
```

The Liesbet verification engine detects the deadlock appropriately, as shown in Figure 11.9.

11.1.6 Travel Agent Example, with Cancellation

In Section 4.5, we present a Travel Agent example (#3), which we repeat here.

```
PayDecision = Stop(
   (Cancelled_act(Flight) | Completed_act(BookFlightDecision)) +
   (Cancelled_act(Hotel) | Completed_act(BookHotelDecision)) +
   (Cancelled_act(Car) | Completed_act(BookCarDecision)) +
   (Cancelled_act(Flight) | Cancelled_act(Hotel) | Cancelled_act(Car)),
```

```
    (Completed_act(Flight) + Cancelled_act(BookFlightDecision)) |
    (Completed_act(Hotel) + Cancelled_act(BookHotelDecision))   |
    (Completed_act(Car) + Cancelled_act(BookCarDecision))
  )

PayCancelChoice = DefaultChoice(PayDecision,Pay; Exit)

Book = MultiChoice(BookFlightDecision,Flight;
   BookHotelDecision,Hotel;
   BookCarDecision,Car)

Par(Seq(Register,Book),PayCancelChoice)
```

In Figure 11.10, we show the output from verifying this model for soundness, where we allow basic activities (specifically, `Flight`, `Hotel` and `Book`) to also be cancelled (as well as complete). All other `Liesbet` examples, presented in this section, have been verified on the basis that basic instances may only complete.

The verification output shows a number of matched states. Identifying matched states greatly increases the efficiency of verification. However, it is necessary to ensure that identification of these states is implemented in an efficient way, because, as the state space grows, it could represent a significant bottleneck. We have succeeded in realising an efficient implementation, as documented in Section 10.3. We identify a matched state by means of a single query on a database table, which represents the limit of how efficient this identification could be.

11.2 Theodore Examples

11.2.1 A Simple Workflow

We now turn to verification of workflow and contract models using `Theodore`. We start with a simple example, which is a flexible workflow model, described as an initial task, `P`, together with a number of methods (or, "count as" rules) and operators for how the task may be decomposed. We have previously described this example in Section 8.2.1.

Our `TheodoreProblem` defines a `TheodoreDomain` having three methods, viz.

- `P: true: Par(S1, S2)` – decomposes `P` into a `Par` containing two tasks `S1` and `S2`.
- `S1: true: Seq(A',B')` – decomposes `S1` into a `Seq` containing two tasks `A'` and `B'`.
- `S2: true: Seq(C',D')` – decomposes `S2` into a `Seq` containing two tasks `C'` and `D'`.

The domain also defines four operators, viz.

- `A': true : true: A` – decomposes task `A'` into the action `A`.
- `B': true : true: B` – decomposes task `B'` into the action `B`.
- `C': true : true: C` – decomposes task `C'` into the action `C`.
- `D': true : true: D` – decomposes task `D'` into the action `D`.

```
Simple workflow domain
Initialising planner...
Planning.cccc.cc.ccc.cccc.cc.ccc.
Workflow/Contract is SOUND with no constraint violations.
Planning details...
Time taken: 0(h),0(m),1(s),750(ms)
**************************************
0:A
0:0:C
0:0:0:B
0:0:0:0:D
END OF PATH
0:0:1:D
0:0:1:0:B
END OF PATH
0:1:B
0:1:0:C
0:1:0:0:D
END OF PATH

1:C
1:0:A
1:0:0:D
1:0:0:0:B
END OF PATH
1:0:1:B
1:0:1:0:D
END OF PATH
1:1:D
1:1:0:A
1:1:0:0:B
END OF PATH
*******************************
```

Figure 11.11: Output from Verifying a `Theodore` Representation whose Initial Task Network Decomposes to the Simple Workflow Model: `Par(Seq(A,B),Seq(B,C))`.

The domain constructs that are present mean that eventually the model `Par(Seq(A,B), Seq(C,D))` will result from all possible decompositions.

The Theodore planner, used as a verification tool, identifies all possible enactments of the initial task; and, for soundness, establishes that all partial decompositions are further decomposable into models which represent complete enactments of the initial task (as described in Section 8.3). In Figure 11.11, we see that there are six possible ways of enacting the initial task network, which is the appropriate result, and that the workflow, as described, is sound. The temporal constraint that is checked confirms the soundness result – we test the CTL* proposition: `AF Completed_act(P1)`, which asserts that the initial task network completes in every possible enactment path.

In Figure 11.11, every '.' after the word 'Planning' indicates where the planner has tried an alternative path in verification, and each 'c' indicates a constraint checking step. In the verification output, each action is prefixed by an action history. If an action is the first in a plan, it will be prefixed by a single number which indicates an index for the action in the collection of actions possible at this stage of the plan. In this output, there are two first actions, either `A` (the `0`th action, in the collection of first actions, as indicated by `0:A`), or `C` (the `1`st action, in the collection of first actions, as indicated by `1:C`). For `0:A`, there are two possible plan continuations `0:0:C` and `0:1:B`, the `0`th and `1`st actions, respectively, in the collection of actions that may follow `0:A`. The first of these may be extended by `0:0:0:B` or `0:0:1:D`, and so on. `END OF PATH` is a delimiter, and indicates that another plan has been found. As each partial plan must lead to a full plan, each prefixed action, output by the planner, must extend the previous one (unless it follows an `END OF PATH` delimiter), and the output must end with an `END OF PATH` delimiter.

11.2.2 TransferProperty Contract with Power (on Vendee)

In this subsection, we present output from performing verification on a Theodore model corresponding to the example presented in Section 9.4.5, where a contract between two parties is described. The contract consists of a *power* held by a vendee, such that on paying three instalments, they acquire the title to a property. This is prescribed by a method which stipulates how this power may be exercised. The contract also specifies a number of other rules, i.e. methods and (complex) operators, which prescribe how payments may be made.

From Section 9.4.5, the contract might look as follows.

- Initial task: `TransferProperty`.
- Method: `MultiSeq(3)(Pay) counts as TransferProperty`.

  (`Pay` on vendee)
- Complex Operator: `SendCheque counts as Pay`.

  (`Pay` on vendee, `SendCheque` on vendee)
- Complex Operator: `EFT counts as Pay`.

  (`Pay` on vendee, `EFT` on vendee)

The authoring of this contract as a Theodore planning problem is shown in Figure 11.12. We use Theodore to verify that all partial decompositions may be completed, and also to verify the constraint: `AG (Completed_act(Payments)` $\rightarrow$ `Completed_act(TransferProperty))`. This

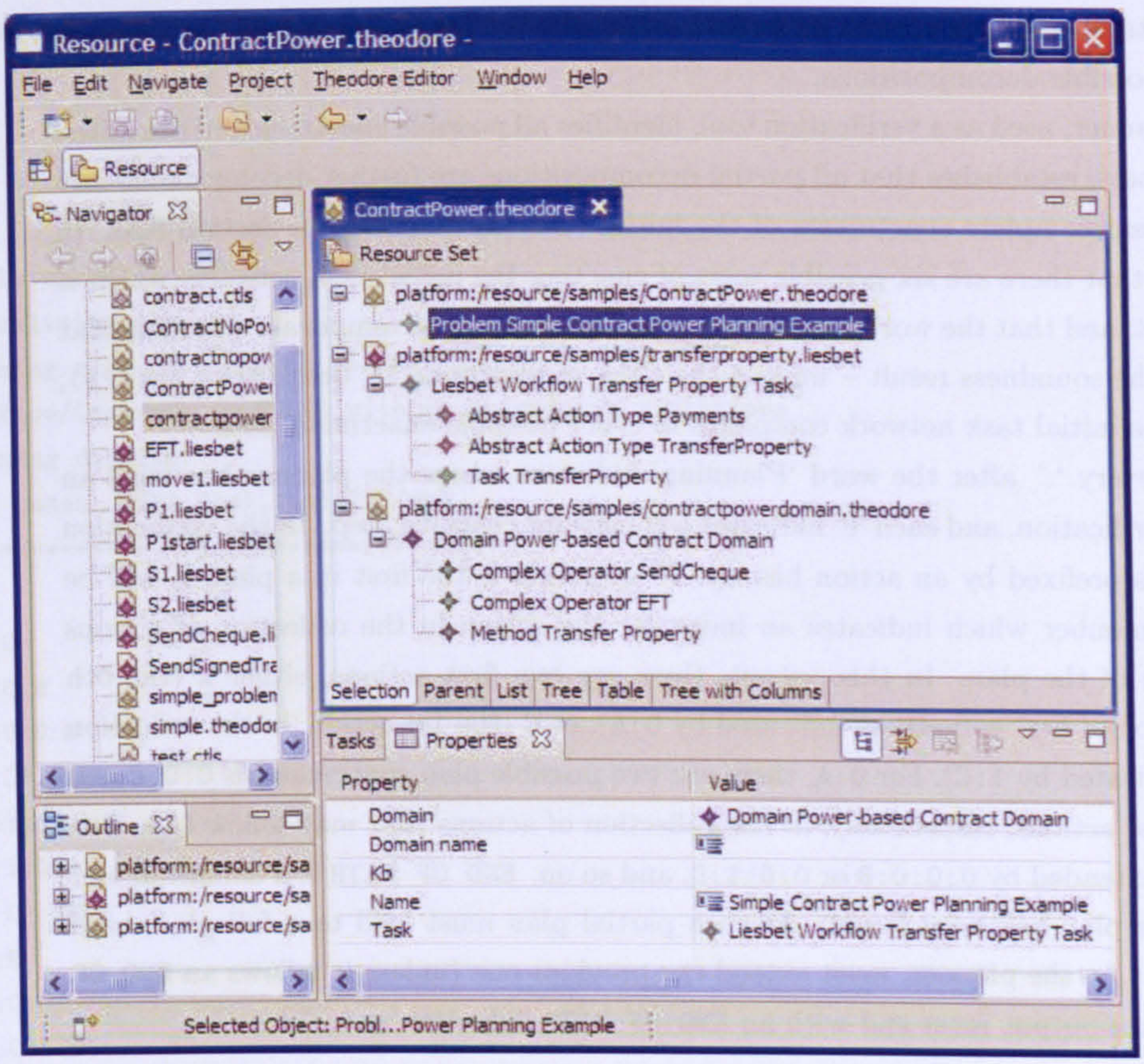


Figure 11.12: `Theodore` Representation of the `TransferProperty` Contract, Containing the Power on the Vendee.

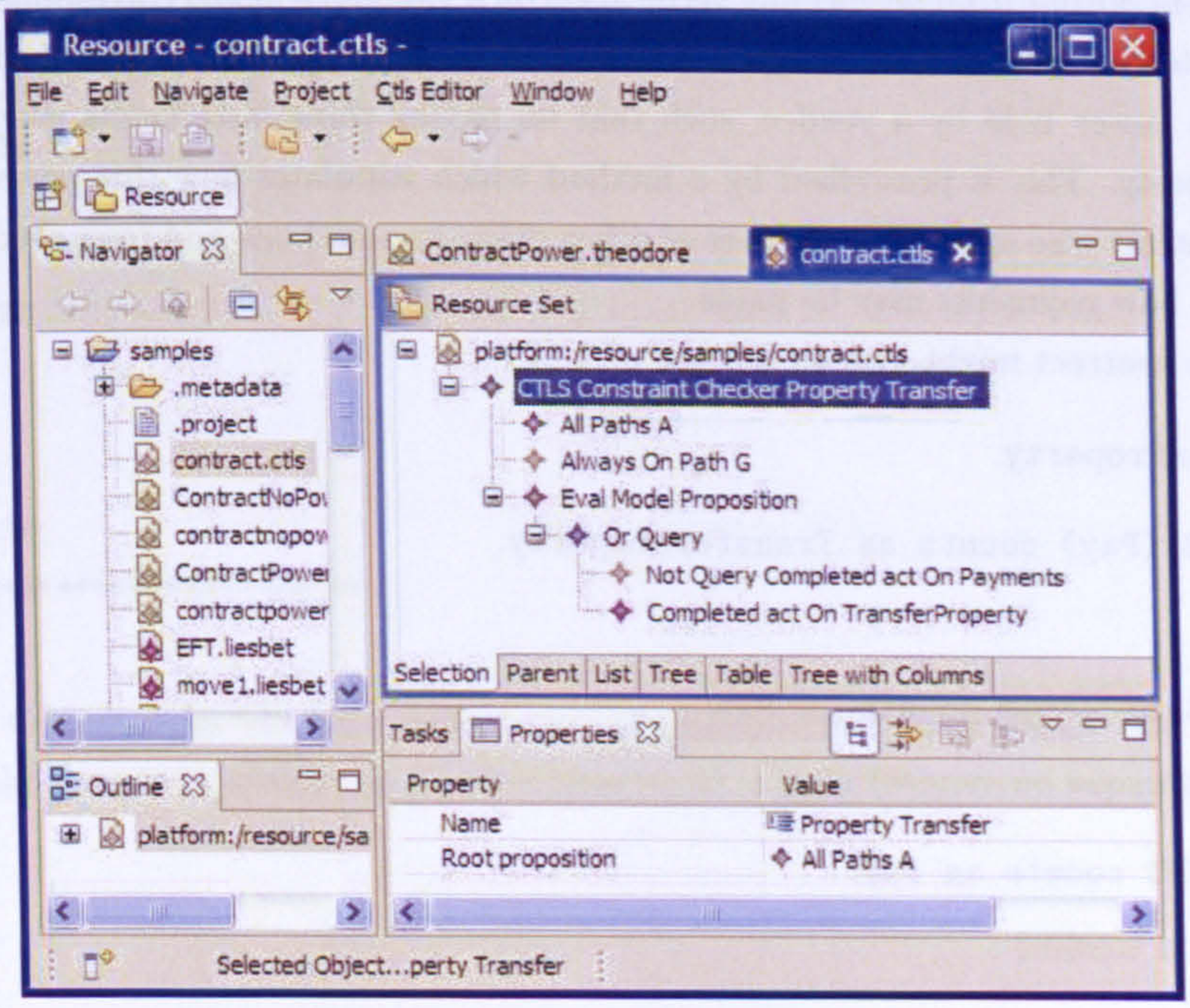


Figure 11.13: `AG (Completed_act(Payments)` → `Completed_act(TransferProperty))`, as Authored in EMF

```
Power Contract with particular constraint checking...
Initialising planner...
Planning.ccc.c.cc.c.ccc.c.cc.c.
Workflow/Contract is SOUND with no constraint violations.
Planning details...
Time taken: 0(h),0(m),0(s),470(ms)
**************************************
0:SendCheque
0:0:SendCheque
0:0:0:SendCheque
END OF PATH
0:0:1:EFT
END OF PATH
0:1:EFT
0:1:0:SendCheque
END OF PATH
0:1:1:EFT
END OF PATH
1:EFT
1:0:SendCheque
1:0:0:SendCheque
END OF PATH
1:0:1:EFT
END OF PATH
1:1:EFT
1:1:0:SendCheque
END OF PATH
1:1:1:EFT
END OF PATH
******************************
```

Figure 11.14: Output from Verifying the Theodore Representation of the TransferProperty Contract, Containing the Power on the Vendee.

```
No Power Contract.  No constraint checking...
Initialising planner...
Planning.........
Workflow/Contract is SOUND.
Planning details...
Time taken: 0(h),0(m),0(s),540(ms)
**************************************
0:SendCheque
0:0:SendCheque
0:0:0:SendCheque
0:0:0:0:SendSignedTransfer
END OF PATH
0:0:1:EFT
0:0:1:0:SendSignedTransfer
END OF PATH
0:1:EFT
0:1:0:SendCheque
0:1:0:0:SendSignedTransfer
END OF PATH
0:1:1:EFT
0:1:1:0:SendSignedTransfer
END OF PATH
1:EFT
1:0:SendCheque
1:0:0:SendCheque
1:0:0:0:SendSignedTransfer
END OF PATH
1:0:1:EFT
1:0:1:0:SendSignedTransfer
END OF PATH
1:1:EFT
1:1:0:SendCheque
1:1:0:0:SendSignedTransfer
END OF PATH
1:1:1:EFT
1:1:1:0:SendSignedTransfer
END OF PATH
******************************
```

Figure 11.15: Output from Verifying the `Theodore` Representation of the `TransferProperty` Contract, NOT Containing the Power on the Vendee.

```
No Power Contract with particular constraint checking...
Initialising planner...
Planning.ccc
Constraint check failed.  Dumping last record.
Instance: 0, State: 1, Cid:TransferProperty
Instance: 1, State: 1, Cid:TransferProperty Seq
Instance: 2, State: 1, Cid:Payments
Instance: 3, State: 1, Cid:MLS_SEQ_Payments
Instance: 4, State: 2, Cid:Pay
Instance: 5, State: 2, Cid:Pay
Instance: 6, State: 1, Cid:Pay
Instance: 7, State: 0, Cid:TransferTitle
```

Figure 11.16: Output from Verifying the Theodore Representation of the TransferProperty Contract, NOT Containing the Power on the Vendee (ii)

This time, we check the constraint that as soon as Payments has been completed, this *counts as* TransferProperty completing.

---

says that the moment the Payments MultiSeq activity has completed, the transfer of property is effected. The authoring of this constraint is shown in Figure 11.13. The output from Theodore is shown in Figure 11.14, where we see that the contract is sound with respect to completion along all possible enactment paths and no constraint violations.

In the output from Theodore, we note the different ways in which payments may be made (either sending a cheque, or by *EFT*).

11.2.3 TransferProperty Contract with No Power (on Vendee)

Finally, we show output from the verification of the variant contract which no power on the vendee. Instead, the power to transfer the property lies with the vendor. The contract, from Section 9.4.5, is represent in Theodore, thus.

- Initial task: TransferProperty.
- Method: Seq(MultiSeq(3)(Pay), TransferTitle) counts as TransferProperty.

  (Pay on vendee, TransferTitle on vendor)
- Complex Operator: SendCheque counts as Pay.

  (Pay on vendee, SendCheque on vendee)
- Complex Operator: EFT counts as Pay.

  (Pay on vendee, EFT on vendee)
- Complex Operator: SendSignedTransfer counts as TransferTitle.

  (SendSignedTransfer on vendor, TransferTitle on vendor)

The output from not performing any constraint checking, as shown in Figure 11.15, is that the contract is sound. Notably, as shown in Figure 11.16, if we include checking for the same constraint as before (namely, that as soon as `Payments` has been completed, this *counts as* `TransferProperty` completing), the verification tool reports the expected violation of this constraint.

In the next chapter, we present some conclusions for this thesis.

Chapter 12

Conclusions and Future Work

In the introduction to this thesis (Section 1.2), we enumerated the aims of this work to be concerned with addressing the issues of:

- Providing a formal grounding of workflow.
- A more flexible approach to workflow.
- How workflow concepts might apply in the modelling of contracts, and looking at the modelling of contracts generally.

We also listed ten contributions of this work that we consider to have gone a significant way to meeting these aims.

1) We address the issue of providing a formal grounding for traditional workflow. We define a meta-model called `Liesbet` as a point of reference for our formalisation of workflow.
2) We have provided an authoring, verification and enactment framework for workflow based on our formalisation.
3) We have identified a reduced set of workflow patterns, using which (we show) all others may be represented.
4) We have demonstrated a number of important results using our formal characterisations of traditional workflow.
5) We have proposed a characterisation of workflow to be: *Flexible Workflow = Abstract Model + Policies for Refinement*, in order that we might support a more flexible view of workflow, including support for *collaborative workflows.*
6) We have implemented our own planner, `Theodore`, which in itself is a useful contribution as it provides many novel features.
7) We have provided an authoring, verification and planned enactment framework for flexible workflow.
8) We have proposed a new perspective of workflow, namely an *institutional perspective.* We call our institutional account of workflow *Institutional Workflow Modelling* (IWM).
9) By drawing out institutional concepts inherent in workflow, we have been able to propose how workflow may be used in the modelling of contracts.
10) We have provided an IWM-based framework for contract authoring, verification and (planned) enactment.

We structure the following discussion around the presented three aims, while providing some further insight regarding the ten contributions as pertinent to do so.

12.1 Formal Grounding of (Traditional) Workflow, through Liesbet

This section primarily concerns **Contribution #1**. That is, the formalisation of workflow in order to address the lack of robust semantics that is typical of many workflow languages [121].

12.1.1 Approach

In defining the Liesbet meta-model, we have sought to understand the *true nature* of workflow, and thus the fundamental concepts that need to be represented with Liesbet. In the overview of constructs, presented in Section 3.1, and through the additional constraints imposed on the intended semantics, presented in Section 3.2, we have defined a clear and succinct point of reference for any formal characterisation at the computational view.

In Section 3.5, we document how Liesbet, at the information view, supports all of the YAWL [125, 126, 123] workflow patterns. The representational requirements for Liesbet were primarily sourced from the need to be able to represent the YAWL patterns [125, 126, 123, 64], as well as the control flow perspective of the Web Service Composition (WSC) language, WS-BPEL. In Section 3.6, we briefly present details of a mapping of (the control flow perspective) of WS-BPEL to Liesbet.

We have defined functions which map Liesbet models to their characterisations at the computational view. For any of these mapping functions, it is the definition of the mapping function, as well as the semantics of the corresponding formalism, that define the particular characterisation. For instance, our SitCalc characterisation of Liesbet is the sum of the mapping function $\mathcal{M}_{SitCalc}[\![-]\!]$ and the semantics of SitCalc, as defined in [98].

12.1.2 A Minimal View of Workflow

Through the definition of Liesbet, we are able to propose a *minimal view of workflow* which may be used to understand what is *fundamentally* required from any computational view formalism used to characterise Liesbet. Our definition of a minimal view is composed of the definition of a reduced (or primitive) set of patterns (**Contribution #3**), with which we show that all others may be represented (as described below), as well as a number of semantic artefacts for workflow that need to be observed. The two together constitute the intended semantics for Liesbet, and our view on a minimal semantics for workflow generally (according to the representational requirements set out for capturing YAWL and WS-BPEL).

Elaborating, we consider a minimal view of workflow to be a collection of activities (operating in parallel threads) with states, thus defining a transition system, whose transitions are constrained by:

- Synchronisation conditions on the states of activities.
- Progression of certain (i.e. structured) instances over others (i.e. basic instances).

- Atomic propagation of all side-effects of completion/cancellation and execution through activity hierarchy.
- Some other phenomena, namely:
  - Join conditions for activities (i.e. support for `SeqCancel`).
  - Unlimited multiple-instance activities (i.e. `Multi`).
  - Activities effecting cancellation of others (i.e. `CancelActivity`).

We have been able to show that all other representational requirements for workflow reduce to this minimal view, through our proposal of a primitive set of workflow patterns. We show this reduction to be sound in Section 6.5. From this minimal view, we are able to conclude that:

- Workflow is little more than: Parallel Composition + Arbitrary Synchronisation.
- The expressivity of workflow rests primarily with the choice/suitability of the synchronisation language.

Note that our minimal view of workflow does place some bias as to suitability of any computational view formalism that we might choose to characterise `Liesbet`. However, this is exactly the point – we have wanted to identify a minimal set of concepts that we feel characterise the true nature of workflow, and this bias is a legitimate by-product of this process.

12.1.3 Comparison of Formalisms for Characterising `Liesbet`

In our work, we have principally used three formal tools for characterising the intended semantics of `Liesbet`. These are:

- Milner's Calculus of Communicating Systems (CCS) [78, 80].
- Cleaveland et al's Prioritised CCS (PCCS) [30, 29], which we shall call PCCS for convenience.
- Situation Calculus (`SitCalc`) [76, 77, 98], based on First-Order Logic (FOL).

We discuss the utility of these various formalisms for characterising `Liesbet` in the next few sub-sections.

12.1.4 CCS/PCCS-based Characterisations

We selected CCS/PCCS as appropriate formalisms to investigate for two reasons:

1) There has been quite a lot of talk within the BPM community as to whether Petri nets or CCS/π-calculus is better suited for the characterisation of workflow, and specifically the `YAWL` patterns [122]. While we do not seek to compare these two formalisms at length, by characterising `YAWL` with CCS we are able to provide a contribution to this debate from one perspective. Note that we do present some points regarding their respective suitability at the end of Chapter Five.
2) The operational semantics of CCS/PCCS (in terms of facilitating compositional specifications of behaviour) should lend themselves quite well to the representation of workflow, and this is a point we seek to investigate.

In this thesis, we have presented a comprehensive formalisation of the Liesbet meta-model using PCCS. The formalisation represents a contribution to the Business Process Management community. We argue that it trivially follows from this that a full CCS characterisation of the Liesbet meta-model is possible. The principal motivation for using PCCS over CCS was lower verification complexity, as well as it being a particularly amenable language, through its in-built support for the specification of priorities, for capturing that the priority of internal workflow activity (i.e. progression of structure instances) over external activity (i.e. progression of basic instances).

There is an interesting dichotomy at play in our PCCS-based characterisation of Liesbet. We could make the verification complexity of PCCS-characterised Liesbet models even better by using further priority levels to achieve an even better partial-order reduction (POR) on the state space. However, these are not strictly necessary to capture the intended semantics of Liesbet, which is sufficiently captured without their use, and they would greatly obscure the clarity of the PCCS-based characterisation of Liesbet. For instance, in the characterisation of synchronisation types, presented in Appendix Section A.3, we use many handshaking actions. These could be mutually-differently prioritised to effect better POR, but, the order in which they occur is not important for the characterisation to be sound. In fact, we could remove some of the use of priorities in the current characterisation, and still have a sound characterisation. Again, the handshaking actions occur at a distinct level of priority from all other actions. We could soundly remove this dispensation, which arguably would make for better clarity in specification but at the cost of increased verification complexity.

Notably, even when we opt for maximising POR in order to reduce verification complexity as much as we can, the performance of verification under CWB-NC is still painfully slow for all but the simplest PCCS-characterised examples. An example is that of the Travel Agent model, presented in Section 4.5, which took several hours to return a result for checking whether the model completes along all enactment paths. The principal reason for this is the inability of the CCS-based characterisations to capture the intended semantics for Liesbet practicably, as explicated by our minimal view of workflow.

Weaknesses

Our PCCS-based characterisation of Liesbet exposes the real weaknesses of using process algebra, such as CCS/PCCS, for the representation of workflow. Formalisms such as these suffer on at least two principal counts:

- It is not possible to arrive at the intended semantics for Liesbet without a lot of abstraction. It is only through abstraction that we may count more than one transition occurring at a time to be atomic, which is a key requirement of the intended semantics (in propagating effects of completing/cancelling childless instances up the tree, for instance). Although CCS/PCCS has a notion of abstraction in distinguishing internal (τ) transitions from external ones, it is not possible to instruct CWB-NC to take account of this difference in constructing the state space of models. The lack of such a capability is hardly surprising: a CCS/PCCS model is fundamentally characterised by all of its transition types, and the distinction between external and internal transition types is purely cosmetic. As such, to perform model checking on a CCS/PCCS, as CWB-NC does, it would always be necessary to construct the state space for a model accounting for all transition types, at least initially. It is the construction of the entire

space that kills CWB-NC when used for verification of PCCS-characterised Liesbet models.

- The efficiency (and clarity) of performing queries as part of progressing synchronisation types is not good. In order to carry out a single atomic query, there is no limit to the number of instances that may be need to be queried as to their state. All of these individual queries themselves require several transitions. The state space for querying alone quickly explodes. Again, this is behaviour that needs to be captured as atomic, together with the consequences of completing/cancelling synchronisation instances being atomically propagated.

It is worth noting, purely subjectively, that the specification of semantics for the generic type agents is quite clear and succinct, using CCS/PCCS. It is evidently appealing to be able to express the semantics using the programming-like, compositional constructs of CCS/PCCS.

The down-side of using such a language is that we would want its operational semantics to admit the notion that multiple transitions may occur atomically, as we have stated. We would imagine that this would be quite difficult to achieve in a process algebra such as CCS/PCCS. Thus, we have some clarity (especially when compared with the Situation Calculus characterisation, presented in Chapter Six) at *the cost of atomicity, which is another apparent dichotomy.*

CCS/PCCS and Petri nets

Interestingly, it is quite evident that Petri nets would not fare any better in characterising Liesbet than our CCS-based characterisations do. The principal reason lies in our making the recording of the state of activities explicit. Because of this, Petri nets would handle the characterisation of Liesbet in largely the same way in having tracker, generic type, and scheduler agents. Moreover, the same shortcomings in the expression and evaluation of synchronisation conditions would exist.

It is also notable that none of the problems asserted (in Section 2.3.2) for Petri net-based characterisations of the YAWL patterns would exist in a Petri net-based characterisation of Liesbet. These problems were concerned with: tracking multiple-instances, advanced synchronisation, and cancellation. This is because we resolve these issues at the information view (i.e. in defining Liesbet) prior to any characterisation using Petri nets/CCS/PCCS. This is a point that is discussed further below.

The need for abstraction described in the discussion of the weaknesses of using CCS/PCCS does highlight an important argument that we seek to make in this thesis, viz.

> General-purpose languages for the description and modelling of process dynamics (such as Petri-nets and CCS) are, necessarily by their nature, too low-level for the description of workflow. In modelling workflow, we are able to make a number of prescriptions, regarding the way in which processes must evolve, as embodied by the definition of a minimal view of workflow here. As such, we are able to describe workflow using artefacts that are much more coarsely-grained than those offered by these general-purpose languages.

Consequently, when looking to characterise workflow, at the computational view, such languages are not ideal choices, as already described. Rather, we need a language in which we are able to capture the espoused minimal view of workflow cleanly. In some regards, as we discuss below, SitCalc is better suited for this purpose, but it is not without its shortcomings either.

CCS/PCCS versus π-calculus

It is notable that both [37] and [94] suggest the use of the π-calculus for the modelling of the YAWL patterns, the latter making particular use of a primary aspect of π-calculus: *mobility* – where communication channels may be passed between agents. Both our work, and that of [117], show that the use of mobility is not essential when modelling the YAWL patterns. In fact, it is hard to see many applications, in the context of workflow modelling, where it is necessary, or particularly desirable. One exception is in the modelling of *sessions*, see for example [60], where dedicated communication channels are passed between agents.

12.1.5 SitCalc-based Characterisation

A motivation for investigating the use of the Situation Calculus was that, as a logic-based formalism, it is quite different to a process algebra-based approach for characterising the behaviour of dynamic systems. Moreover, we felt that certain aspects in which CCS/PCCS may be deficient may be better addressed using the Situation Calculus, and vice-versa, making the investigation of using the Situation Calculus to characterise Liesbet complementary to the investigation of using CCS/PCCS.

Strengths

An unequivocal advantage of using SitCalc for the characterisation of Liesbet is that certain aspects of the intended semantics for Liesbet are captured quite straightforwardly, such as: arbitrarily-complex synchronisation conditions, priority of structured instances over basic instances, and atomic propagation of side-effects through the activity instance hierarchy (see Section 3.2 for more information regarding the intended semantics). Atomic propagation of effects is particularly important when it comes to verification, as it greatly reduces the complexity of verification (in terms of the state space generated). In our CCS/PCCS-based characterisations of Liesbet, we fail to capture this notion in the absence of abstraction, and as a result verification complexity soars.

The specification of semantics for synchronisation conditions (both as queries in Go and Stop types – see Section 3.1.4, and in synchronisation rules – see Section 3.3) is naturally accommodated by logic-based formalisms, such as SitCalc, where we can straightforwardly access current workflow state. That is, we can write the conditions as fluent-based assertions that must currently hold, as described in Section 6.3. Token-based formalisms (such as Petri nets), or process-based formalisms (such as CCS/PCCS), appear to be less suitable for the purpose of capturing synchronisations conditions, because of the need to consume many tokens, or make many transitions, in order to ascertain the result of a query. As well as being inefficient from a verification perspective, it also tends to be undesirably verbose. This does not mean to say that it is not possible to represent such conditions using these other formalisms, as we have demonstrated in Appendix Section A.3, for PCCS.

A Weakness

A weakness of using SitCalc (albeit subjective) is that, while the initial foundational axioms for workflow presented in Figure 6.12 are arguably clear enough, the augmented foundational

axioms for generic activity types such as the choice types (e.g. `Choice`) and merge types (e.g. `Multimerge`, presented in Appendix Section B.1.4) are rather impenetrable. In contrast, the CCS/PCCS-based characterisations presented in Chapter Five are arguably a lot clearer in their meaning. As pointed out there, however, there is an apparent dichotomy between the clarity that comes from programming-like metaphors for characterising behaviour, on the one hand, and the ability to model atomic arbitrary side-effects, on the other. It very much appears that, for the characterisation of `Liesbet`, the strengths of the logic-based approach (i.e. `SitCalc`) are the weaknesses of the process algebra- (i.e. CCS/PCCS) based and vice versa.

12.1.6 Shoe-horning

We do not consider it to be appropriate, as an alternative approach to ours, to *shoe-horn* specifications of workflow artefacts directly into some general-purpose formal language (as people have done, for example, when considering the application of Petri nets to workflow – see `WF-nets` in Section 2.3.2). The problem with a shoe-horning approach is when it recommends the underlying formalism in its entirety for, in this context, the specification of workflow. This is often inappropriate because it allows the use of the underlying language in an unconstrained way. It *is* useful to consider the applicability of general-purpose formalisms (such as CCS and Petri nets), given the tool-support and the wealth of results that exist for them, but to do so in the absence of defining an abstract model of what needs to be modelled, and using such a model to *constrain* the use of the underlying formalism, would appear to be folly. In our view, the approach should be top-down, rather than bottom-up.

An example of this point can be found in an issue described in the `YAWL` (Yet Another Workflow Language) work [125], and further investigated in [140], concerning the use of OR-joins in the context of arbitrary cycles. OR-joins are meant to synchronise (possibly) multiple threads of enactment. Arbitrary cycles are unstructured cycles in that they may contain arbitrary entry and exit points. In the Petri net-like, token-based, characterisation of `YAWL`, it is not clear when an OR-join should be considered to be satisfied, i.e. when it has received all pertinent input tokens. The issue is exacerbated by the use of arbitrary cycles because, according to the token-based semantics, the question of how tokens will be recycled has a non-trivial answer.

We would consider that issues such as when to synchronise an OR-join should be answered at the information view, without consideration of any particular computational view tool or formalism. This issue manifests itself in `YAWL` because of the chosen computational view formalism dictating the ontological commitments of workflow artefacts, rather than cleanly defining these separately, and then only using the computational formalism to describe their meaning. In `YAWL`, workflow artefacts are shoe-horned into the machinery of the underlying computational formalism, rather than defining them cleanly, in a suitably abstract way. In doing this, there is no mechanism (i.e. the information view) constraining the use of the underlying computational formalism, which leads to the creation of a representational problem for workflow which need not exist. In our information view model for workflow, `Liesbet`, synchronisation occurs when queries on workflow state are satisfied. Query satisfaction is easily computable according to its informal semantics, and this remains the case when formalising the semantics of `Liesbet`, at the computational view, using CCS/PCCS, `SitCalc`, or, indeed, Petri nets. The problem concerning synchronisation (for OR-joins) does not arise.

A similar issue obtains with respect to the cancellation of activities, which constitutes YAWL patterns #19 (Cancel Activity) and #20 (Cancel Case). These patterns cause problems when modelled using Petri nets, if cancellation is modelled as the withdrawal of tokens. This is because it is not possible to anticipate, generally speaking, how many tokens to remove from appropriate places within a Petri net in order to effect a cancellation. This led the authors of [125] to introduce special "vacuum-cleaner"-like artefacts, as part of a transition-system based semantics for workflow, to model these YAWL patterns.

12.1.7 An Appropriate Expressivity for Workflow

In considering the definition of a workflow language, it is clearly important to decide an appropriate expressivity for the language. By this, we do not mean whether the language is Turing-complete, i.e. whether it has the computational power of a Universal Turing Machine [114]. A language may be Turing-complete, but it does not mean that it is *suitable* for writing workflow models in a succinct and clear way. Moreover, there are definite benefits, in defining special-purpose languages, for them not to be Turing-complete, such as for decidability reasons. Rather, suitability, in the sense conveyed, is the key. It is important at both information and computational views.

Ultimately, an appropriate expressivity at the information view meta-model is going to depend largely on the set of patterns that we seek to capture, i.e. the YAWL patterns. Whenever the set of patterns and thus the representational requirements for workflow grows, we would need to take account of the additional requirements within the information view meta-model, Liesbet. As it currently stands, Liesbet supports all of the patterns that are prescribed by the representational requirements.

There is another source of uncertainty regarding the expressivity of Liesbet, which lies in the expressivity of the language for expressing synchronisation conditions. It is unclear whether the language used for synchronisation conditions is sufficient for capturing all conceivable workflow-based scenarios.

In fact, the language is rather simple: a synchronisation constraint is made up of *queries* on state that need to be satisfied, where any particular *atomic* query has an associated *visibility horizon.* The key feature of the language is how the visibility horizon of a query is specified; and this would be a principal issue when deciding whether the language is sufficiently expressive. Currently, a visibility horizon, in the absence of the use of isolated scopes, may be either unconstrained, or constrained according to the use of reference types. We have devised a means of constraining the visibility horizon of a query on the basis that querying instances will be principally interested in the state of instances that share a common, local ancestor instance – hence, the use of plain reference types. Sometimes, we are interested in satisfying queries (for instance, when used within multiple-instance activity types) in a distinct way – hence the use of distinct reference types. The use of isolated scopes further constrains a query's visibility horizon.

A key difference in our work, to that of [37, 94, 117], reviewed in Section 2.3.2, lies in the capability for arbitrary synchronisation on workflow state. These other approaches only support very primitive querying against workflow state, in order to facilitate the Milestone (#18) YAWL workflow pattern. In our approach, a model author can, in both synchronising the performance of activity instances and in cancelling activity instances, gain a fine level of control over how activity instances are synchronised or what instances are cancelled.

One advantage of the use of an information view meta-model to fix the representational requirements is that it is possible to consider the most desirable way of expressing artefacts without concern for any particular computational view formalism that may be used to characterise the semantics of workflow. We would argue that our synchronisation language is particularly succinct and intuitive, even when the synchronisation condition that needs to be expressed is quite complex. This is a function of being able straightforwardly to combine complex queries on workflow state. In contrast, when using CCS and Petri nets it is far from straightforward to do so. An example of this was presented at the end of Chapter Four, where the third travel agency scenario is in fact quite complex to characterise using Petri net-based networks.

12.1.8 Bespoke Formalism

It might prove beneficial to define a bespoke formalism for Liesbet to capture its semantics more directly. We would make the following comments in this regard. We could either:

- Directly characterise the information view, in which case we might attempt the definition of a (structured) operational semantics applied to Liesbet compositions. It is by no means clear how attractive such a characterisation would be, for instance, in terms of its clarity and understandability.

  OR

- Define a language which raises the level of abstraction closer to that of the intended semantics of Liesbet, but does not provide a semantics directly to Liesbet. That is, the language could support some notions of the intended semantics as first-class artefacts, such as the notion of hierarchy.

  The idea would be that the mapping from the meta-model to the intermediate language, and the semantics of the intermediate language itself, are clear and easy-to-understand. Thus, the sum of the semantic characterisations is expressed in a natural way, avoiding the weaknesses that have been highlighted in both SitCalc and CCS/PCCS characterisations.

  In light of the (apparent) dichotomy expressed regarding the characterisation of generic-activity types using programming-like constructs, and the need to support atomicity, it is less than clear how successful an attempt at a bespoke formalism in this way would be.

12.1.9 Results Demonstrated for Characterisations of Liesbet

We have presented two principal results in this work (**Contribution #4**), viz.

1) For our CCS and SitCalc-based characterisations, we have proved that completion of Liesbet models is guaranteed (in the context of assumptions relating to the absence of deadlock and livelock in a Liesbet model).
2) For SitCalc models, we show that the characterisations, presented in Section 3.4, of Liesbet constructs as abbreviations, in the set $\texttt{Liesbet}_{abbrev}$, are sound.

These are particularly useful results; the latter confirms our minimal view of workflow to be correct.

12.1.10 Authoring, Verification and Enactment Framework for Traditional Workflow

We have implemented an authoring, verification and enactment framework for `Liesbet` models (**Contribution #2**). Regarding authoring, we have a simple GUI for describing models, as shown in Chapter Ten. For enactment, our `Liesbet` engine provides a Java-based API in order that the workflow engine can be integrated with other application logic. Our main interest in these conclusions is in discussing verification.

For our CCS/PCCS-based characterisations, we have used the Concurrency Workbench for the New Century (`CWB-NC`) [11], as a direct route to verification. However, we have found verification of CCS/PCCS-characterised `Liesbet` models using `CWB-NC` to be punitively inefficient, given the wastefulness in terms of states and transitions of the CCS-based characterisations, as described in Section 5.7.

We have sought to make verification under `CWB-NC` practicable by ensuring that the CCS/PCCS characterisations are as efficient as possible in their semantic characterisation of `Liesbet`. Unfortunately, both characterisations do still lead to inflated state spaces. This is due to the lack of atomicity in effecting propagation of side-effects, as described previously, and also the inefficiency of evaluating artefacts such as synchronisation conditions, when these are represented in CCS/PCCS.

The simple example: `Par(Seq(A,B),Seq(C,D))`, presented in Section 5.1.3, when characterised using CCS, had a state space of 833 states under `CWB-NC`. A significant improvement is made in the PCCS characterisation of the same model. It generates 53 states under `CWB-NC`. It is notable that the state space, as described in Section 6.2.1 according to the `SitCalc`-based semantics, for this particular model is 10 states.

An advantageous aspect of `CWB-NC` is that we have been able to use it to provide quick validation of our CCS-based characterisations of `Liesbet`. A similar approach could be undertaken in using a logic programming language like Prolog to quickly validate our `SitCalc`-based characterisation of `Liesbet`.

Our principal framework for verification of `Liesbet` models is implemented in Java and runs considerably more efficiently in verification than `CWB-NC` on the examples that we have presented because it operates according to the intended semantics for `Liesbet` – described above, which has the consequence of minimising the verification state space. It has been implemented against the `SitCalc`-based characterisation of `Liesbet`, but, as it is realised using Java, it is not a direct implementation of the `SitCalc` axioms for `Liesbet`, which would be the case if were to express them in Prolog, for example.

However, as noted in Chapter Ten, use of `SitCalc` provides a natural path to implementation using a relational database. The database query language SQL is a sugar-syntax for the relational calculus, which means `SitCalc` successor-state axioms easily map to SQL queries. We decided against the use of Prolog for reasons of efficiency. Not only would Prolog be quite inefficient, relational databases are contrastingly very efficient at manipulating database tables and returning results from the queries captured on the right-hand side of `SitCalc` successor state axioms.

Our verification approach for `Liesbet` is capable of verifying workflow soundness as well as checking `Liesbet` models against constraints expressed in the temporal logic `CTL*`. In principle, any constraint language whose semantics can be characterised by a *progression function*, such as that presented in Chapter Ten, would be suitable. The verification engine divides the verification

```
pan: invalid end state (at depth 17)
pan: wrote test3.xml.prm.trail
(Spin Version 4.2.5 -- 2 April 2005)
Warning: Search not completed
        + Compression

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 100 byte, depth reached 18, errors: 1
       5 states, stored
       0 states, matched
       5 transitions (= stored+matched)
      14 atomic steps
hash conflicts: 0 (resolved)
...truncated
```

Figure 12.1: SPIN Output for Example Liesbet Model.

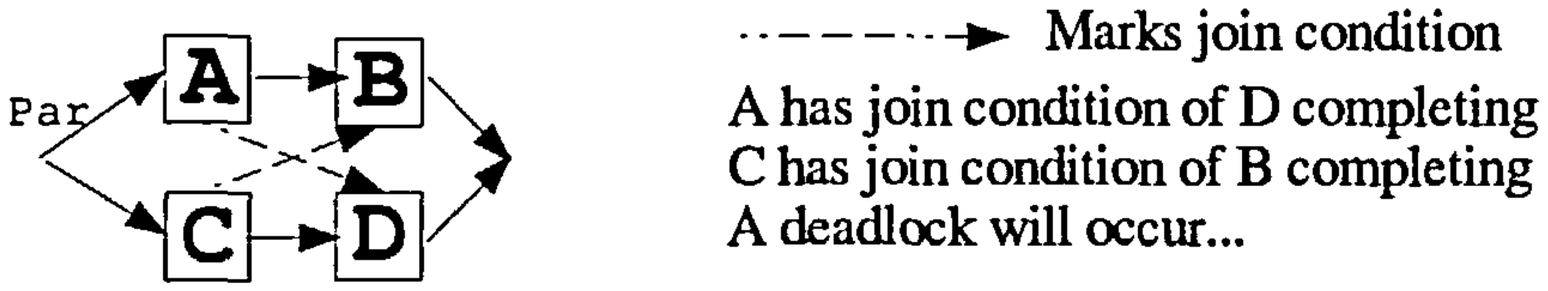


Figure 12.2: Graphical Representation of Example Liesbet Model.

of Liesbet models into a number of runs, whilst maintaining verification soundness. This has the effect of reducing the complexity of verification further still.

It is worth mentioning that we have experimented with the SPIN [59] model checker for the verification of Liesbet models. We have represented the semantics of Liesbet in Promela (the input language for SPIN) and found it to be punitively inefficient, as was the case with verifying CCS-based models with CWB-NC. This is because of the inability of Promela to capture the intended semantics of Liesbet in an efficient way. The principal problem, which is common to our experience of using CWB-NC with CCS/PCCS, is the inability to prescribe arbitrary side-effects of actions as being atomic.

It is possible to use SPIN as a model-checking wrapper for models written in the imperative programming language *C*. We have also implemented a verification approach for Liesbet based on this approach. This is particularly desirable because it means that we can capture minimal models, while using the highly-optimised implementation of SPIN to *drive* the verification process.

In Figure 12.1, we present an excerpt from the output of SPIN when detecting the deadlock in the Liesbet model presented (graphically) in Figure 12.2. The deadlock is identified by the verification run reaching an "invalid end state" – a state which is not one that pertains to proper completion of the Liesbet model, but, in being an "end state", is one that cannot be progressed. The model has the following *Easy Syntax* definition.

```
Par(Seq(A,B),Seq(C,D))
A=Act(join(Go(Completed_act(D))))
```

```
C=Act(join(Go(Completed_act(B))))
```

12.1.11 Synchronisation Rules – A First Attempt at Flexibility

We have taken our first step towards greater flexibility in workflow models through the proposal of *Synchronisation Rules*, which may be used to provide a notion of flexibility that may be captured as: *Flexible Workflow = Concrete Model + Policies for Constraint.* We have described how such rules may be useful. For instance, we are able to capture the behaviour of `Liesbet`'s `PriPar` construct using such a rule.

12.1.12 Strengths and Weaknesses

We consider the use of an information view meta-model to constrain the scope of the semantics of the underlying computational view formalism not only to be essential but also to be a real strength of our work compared with other contributions, such as [125]. The entailment (from `Liesbet`) of a minimal view of workflow is also significant in allowing us to understand the fundamentals of workflow for representation at the computational view.

The question of the adequacy or sufficiency (in expressiveness) of the synchronisation language remains open, and can only be effectively addressed through a comprehensive study of typical workflow scenarios. Another weakness of our work is that neither CCS/PCCS- nor `SitCalc`-based characterisations are wholly suitable for the characterisation of `Liesbet`. It is unclear, however, whether a bespoke formalism would necessarily improve matters. The apparent dichotomy between clarity and atomicity would need some significant thought to address.

12.2 A Flexible Approach To Workflow, through `Theodore`

This section primarily concerns **Contribution #5.** We have proposed an approach to flexible workflow modelling, which is desirable to counter the significant issue of brittleness in traditional models of workflow. In doing so, we have been able to accommodate collaborative workflows, which are an important kind of workflow (as described in Section 1.1) where agents decide collectively how a workflow instance should be realised.

12.2.1 Correspondence to `HTN`-based Planning

Our approach to flexible workflow modelling is based on the identification of a correspondence between what we seek to achieve in flexible workflow modelling, as epitomised by the slogan: *Flexible Workflow = Abstract Model + Policies for Refinement*, i.e. refining an abstract workflow, specified for flexible enactment, into a concrete one, and the operation of an `HTN`-based planner, which refines abstract task networks into concrete ones. In identifying such a correspondence, we are able to propose a novel approach using `HTN`-based planning for the description, verification and planned enactment of flexible workflow models.

We have implemented our own `HTN`-based planner, called `Theodore` (**Contribution #6**). We implemented our own planner rather than using an off-the-shelf planner such as SHOP [85], as we wanted features (such as complex operator-like artefacts) not available in any other planner.

12.2.2 Providing Structure with Flexibility

A key theme in our work in flexible workflow modelling is the notion that we combine structure with flexibility. That is, we start with an abstract workflow model which provides some initial structure. Furthermore, there is structure inherent within the policies for refinement, i.e., the decomposition relations – methods, operators and complex operators, in that they prescribe networks of actions which are acceptable refinements of tasks being decomposed. Moreover, complex operators prescribe structure from the bottom-up, in specifying complete refinements of tasks.

All of these dispensations, with respect to structuring, help reduce the complexity of verification. There is a trade-off here between flexibility in workflow specification, and complexity of verification. When we allow greater flexibility, the complexity will soar; but, as we allow less freedom, the complexity will drop. In the extreme of the latter case, we will have fully prescribed workflow models whose verification complexity will be that of `Liesbet` models.

12.2.3 Expressivity

The expressivity of the planning language for describing domains is limited by the expressivity of the knowledge base underwriting the problem description, together with the expressivity of the language used in pre-conditions and effects axioms, and the expressivity of the workflow language (such as `Liesbet`) that is used for the specification of abstract workflows. As our planner is modular, all of these provisions can easily be changed, and, thus, in principle, our approach does not limit workflow authors in what they would seek to express.

This is a double-edged sword, however, with respect to decidability of an authored problem, and, as a consequence, some care must be take during the process of describing problems to ensure that decidability is maintained. This is perhaps a less than ideal consequence of making our planner wholly flexible. As already stated, we may at some time look at some constraints on what is allowed to be expressed, as other planners such as SHOP [85] do. We are minded, however, to prioritise flexibility at the possible detriment of usability for the time being.

12.2.4 Meaning Assignable to a `Theodore` Flexible Workflow Model

The meaning that may be assigned to a workflow model expressed with `Theodore` is simply the set of full decompositions that may result from planning over the initial abstract workflow using the decomposition relations specified in the `Theodore` model. The meaning of a `Theodore` model may in this sense be considered as being mutiplicious. This is in contrast to `Liesbet` models, which may be considered to be singular in meaning – that is, for any `Liesbet` model, its meaning is the single network specified therein.

12.2.5 Authoring, Verification and Planned Enactment Framework for Flexible Workflow

We have implemented an authoring, verification and enactment framework for flexible (i.e. `Theodore`) workflow models (**Contribution #7**). As before, regarding authoring, we have a simple GUI for describing models, as shown in Chapter Ten.

In our approach to flexible workflow modelling, we make a distinction between fixed and variable models. Fixed (resp. variable) models are those for which the set of decomposition relations for HTN tasks is (resp. is not) fixed. For fixed models, we define a notion of soundness which is embodied as the *verification criterion*. This criterion prescribes that every partial decomposition of a Theodore model leads to a full decomposition.

The terms verification and enactment amount to: (i) verification and flexible enactment for fixed models, and (ii) planned, flexible enactment for variable models. For fixed models, it is also imperative that their planning domains be practicably decidable. Under the assumption that this is the case for a particular model, verification of fixed flexible workflow models for soundness and for the satisfaction of arbitrary temporal constraints is a particularly desirable aspect of our framework and novel in the context of flexible workflow modelling.

For variable models, the options are based around finding a plan to realise the (possibly partially enacted) abstract workflow. Here, we may perform "what may I do next?" querying, as well as "what-if" simulation. These facilities are also available for fixed models. In performing planning, a domain expert is able to make choices of which decomposition steps to take based on his or her subjective constraints, as well as doing on-line planning which mixes planning with enactment.

12.2.6 Strengths and Weaknesses

The strengths of our approach to flexible workflow modelling are as follows:

- The capability for expressing structure in the definition of a workflow from the bottom-up as well as the top-down provides additional power to domain authors in controlling the degree of flexibility in a model. This bottom-up structuring is provided by complex operators, which are a novel aspect of our work.

- As described at the end of Chapter Eight, our approach compares favourably in terms of the modelling capability, and verification and planned enactment facilities against other approaches to flexible workflow modelling. No other approach that we have been able to identify in the literature provides the range of support that we do. We also naturally capture the notion of collaborative workflows in our approach.

Similar to the weakness identified for the synchronisation language for Liesbet, the only weakness that we currently identify in our work on flexible workflow modelling is that we are not sure whether our approach is powerful enough to cover the range of possible scenarios that might obtain. We are only going to be able to gain insight into resolving this matter through a comprehensive study of typical workflow scenarios.

12.3 Workflow as a Basis for Contract Modelling, through Institutional Modelling

This section primarily concerns **Contribution #8**. We have been motivated to consider how our work on workflow modelling might be reused in other contexts. We consider this to be an important issue in itself, as part of the utility of research comes from considering how it may be applied in different contexts. An immediately-apparent context was that of contract modelling,

where contracts are often cast as protocols (i.e., workflows) of behaviour between two or more parties. We have been motivated to look at the issue of contract modelling for its own sake as well, as this remains a somewhat formative research field in which there is ample scope to make a worthwhile contribution.

12.3.1 Institutional Workflow Modelling (IWM) as a Foundational Basis for Normative and Contract Modelling

In order to explicate how our previous work may be reused, we have identified a new perspective for workflow, namely an *institutional perspective* (**Contribution #9**). We define *Institutional Workflow Modelling* (IWM) as an embodiment of an institutional perspective for workflow. In IWM, we identify the institutional concepts of *counts as* and *permission*, and the related classification of actions into *institutional* and *brute* classes of action, to be pertinent to the characterisation of workflow.

These concepts are also pertinent in normative and contract modelling (NCM), and our experience shows IWM to be useful as a foundational basis for NCM. The utility of IWM, in this regard, is evident from both examples given Chapter Nine, namely, the `TransferProperty` and mail service agreement examples.

We define IWM to be the sum of our `Theodore`-based approach to flexible workflow modelling and the presented correspondences of *counts as* and *permission* relations to workflow artefacts, on the one hand, and `HTN`-based planning constructs (i.e., methods, operators and complex operators), on the other.

When IWM is applied in the modelling of contracts, *counts as* provides a means of modelling power, and permission provides a means of modelling privilege (in the terminology of Hohfeld). Jones and Sergot [63] identify the correspondence between *counts as* and power, in respect of *counts as* relations prescribing ways in which powers may be exercised. Obligation is modelled by leaf activities within IWM model fragments, which may pertain to institutional or brute actions that demand the presence of powers and privileges (as methods and operators, respectively) to refine them.

12.3.2 Mechanism for Relating Obligation Fulfilment to Extant Power and Privilege

A particularly interesting aspect of our approach to contract modelling is that it relates the fulfilment of obligations directly to the existence of powers and privileges, in providing a mechanism by which contract enactors may query and plan obligation fulfilment using these relations. The distinction between institutional and brute actions in the modelling of contracts, and thus the distinction between power and privilege, is often overlooked in the modelling of contracts (see, for example, [82, 115]).

12.3.3 Authoring, Verification and Planned Enactment Framework for Contracts

We have implemented an authoring, verification and enactment framework for IWM-based contracts (**Contribution #10**), which builds on our IWM framework.

We make a distinction between *fixed* and *variable* contract models. A necessary condition for a model to be fixed is that the set of decomposition relations, described therein, is fixed. This is a condition that carries over from our work on flexible workflow modelling. Another necessary condition is that a power may not be exercised in the absence of an obligation that prescribes the institutional action to which the power applies.

As our approach to contract modelling is based on our work on flexible workflow modelling, we reuse a lot of the components implemented in the verification and enactment engine for `Theodore` flexible workflows. For fixed contract models, a contract author or contract party (in enactment) may make use of the IWM-based verification facilities for soundness and arbitrary constraints. For both fixed and variable models, a party may perform "what-if" simulation and "what may I do next" querying.

12.3.4 Strengths and Weaknesses

The strengths of our approach to contract modelling may be enumerated (non-exhaustively) as follows.

- The use of workflow artefacts to model contracts in a hybrid approach with auxiliary normative relations is at least uncommon if not novel. It is a strength because it provides a natural means of modelling protocol fragments inherent in contracts.
- Accounting for the normative concept of *power* in contracts, which (as stated) is often overlooked in approaches to contract modelling. Not only do we model it, but we also provide a mechanism by which a contract author/party can simulate and reason over the fulfilment of obligations using powers (as well as privileges).

  Power is very important contract modelling both in itself but also to give structure to a contract model. Without it, contracts would be specified at the level of brute actions instead of institutional and brute actions, serving to remove the possibility of specifying abstraction hierarchies in contracts.

Again the weaknesses of our work lie in the breath and depth of coverage that we have been able to give to different scenarios in which contracts may be used. In future work, we need to give our approach to contract modelling a comprehensive road-test against a number of different sorts of contracts in order to identify any weaknesses in our modelling, verification and planned enactment approach. It is not until we do so that we can be certain that it is a sufficient approach to contract modelling. However, it is certainly clear from comparable works that have been carried out by the research community that it is a significant and useful contribution.

12.4 Future Work

We intend to continue working on our flexible/IWM-based approach for workflow modelling, verification and planned enactment. One key area in which we intend to apply our work is template-based planning for Web Service Composition [86, 133]. Our planner `Theodore` provides some nice features that would be useful in this domain. For example, the complex operator artefact would be useful in representing complete service orchestrations enabling us to plan over services rather than service operations, thus speeding up planning.

We shall also look at how we may practically integrate the use of other planners and tools into our Theodore-based planning framework. For example, in HTN-based planning, the ability to perform hybrid planning is frequently desirable where HTN-based planning is combined with operator-based planning (the latter "filling in the gaps" when the former has no applicable decomposition relation, for example). Another example tool that we would seek to integrate is a scheduler so that we may combine the planning of compositions with the scheduling of their enactment. This would make an important contribution to the area of *Business-Driven IT Management* [1], for example, where an important issue is effective *Change Management* (ChM). In ChM, there is a need to plan and schedule *changes* to underlying IT infrastructure in ways which serve to best meet current business objectives, codified as business rules. We also need to look at a number of issues relating to what the notion of institution means in the context of workflows and contracts. We shall also continue to evolve our IWM-based approach to contract modelling including the maturing of tool-support.

We intend to continue to evaluate the suitability of our workflow language Liesbet for the specification of the control flow perspective of workflow, particularly the language for the expression of synchronisation queries. We shall continue to make adjustments to Liesbet and its SitCalc/FOL-based characterisation, as this our preferred characterisation given that it more naturally captures the intended semantics of Liesbet. We are also in the process of developing another model of orchestration. It augments Liesbet with the notion that activities may have arbitrarily complex lifecycles, providing for a more natural and intuitive way of authoring certain notions of orchestration.

Furthermore, we will mature the implementation of our verification and enactment engine for Liesbet and Theodore. In looking at all of these things, it will be necessary to identify a stock of representative use-cases which can be used to ground and contextualise the work.

Bibliography

[1] Business Driven IT Management, at: http://businessdrivenitmanagement.org. URL last verified: 2008-22-01.

[2] Business Process Trends at: http://www.bptrends.com. URL last verified: 2008-22-01.

[3] Eclipse – an Open Development Platform at: http://www.eclipse.org. URL last verified: 2008-22-01.

[4] Eclipse Modelling Framework, EMF, at: http://www.eclipse.org/emf. URL last verified: 2008-22-01.

[5] Mirriam Webster Dictionary at: http://www.m-w.com. URL last verified: 2008-22-01.

[6] On-Demand, Grid and Utility Computing at: http://www.utilitycomputing.com. URL last verified: 2008-22-01.

[7] Proceedings of First Workshop of Process Modelling Group, Eindhoven, June 2005. Available at: http://www.bptrends.com. URL last verified: 2008-22-01.

[8] Process Modelling Group, now defunct, see [7].

[9] RuleML: The Rule Markup Initiative, at: http://www.ruleml.org. URL last verified: 2008-22-01.

[10] Simple Hierarchical Ordered Planner (SHOP), Automated Planning at the University of Maryland at: http://www.cs.umd.edu/projects/shop/index.html. URL last verified: 2008-22-01.

[11] The Concurrency Workbench of the New Century at: http://www.cs.sunysb.edu/~cwb/. URL last verified: 2008-22-01.

[12] A.K.Bandara, E.C.Lupu, and A.Russo. Using Event Calculus to Formalise Policy Specification and Analysis. *Fourth IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2003), Lake Como, Italy*, 2003.

[13] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services, ISBN: 3540440089.* Springer, 2004.

[14] Andrew D H Farrell. Logic-based Formalisms for the Representation of Service Level Agreements for Utility Computing. Master's thesis, Imperial College, London, 2003.

[15] Alexander Artikis. *Executable Specification of Open Norm-Governed Computational Systems.* PhD thesis, Imperial College, London, 2003.

[16] Pallas Athena. Case Handling with FLOWer. Beyond Workflow, Pallas AthenaBV, Apeldoorn, The Netherlands. 2002.

[17] Fahiem Bacchus and Froduald Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[18] Falko Bause and Pieter S. Kritzinger. *Stochastic Petri Nets – An Introduction to the Theory, ISBN: 3528155353.* Vieweg, 2002.

[19] James B.D.Joshi, E. Bertino, Usman Latif, and Arif Ghafoor. Generalised Temporal Role-Based Access Control Model (GTRBAC) Part I: Specification and Modeling. *CERIAS TR 2001-47*, 2001.

[20] James B.D.Joshi, E. Bertino, Usman Latif, and Arif Ghafoor. Generalised Temporal Role-Based Access Control Model (GTRBAC) Part II: Expressiveness and Design Issues. *CERIAS TR 2003-01*, 2003.

[21] Khalid Belhajjame, Christine Collet, and Genoveva Vargas-Solar. A Flexible Workflow Model for Process-Oriented Applications. In M. Tamer Özsu, Hans-Jörg Schek, Katsumi Tanaka, Yanchun Zhang, and Yahiko Kambayashi, editors, *Proceedings of the Second International Conference on Web Information Systems Engineering (WISE'01), Organized by WISE Society and Kyoto University, Kyoto, Japan, 3-6 December 2001, Volume 1 (Main program).* IEEE Computer Society, 2001.

[22] B.N.Grosof, Y.Labrou, and H.Y.Chan. A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML. In M.P.Wellman, editor, *Proceedings of First ACM Conference on Electronic Commerce (EC-99), Denver, Colorado, USA.* ACM Press, New York, NY, USA, November 1999.

[23] James B. Brady. Law, Language and Logic: The Legal Philosophy of Wesley Newcomb Hohfeld. *Transactions of the Charles S. Peirce Society*, 8:246–263, 1972.

[24] Business Modeling & Integration Domain Task Force. Business Process Modelling Notation (BPMN) Specification, at: `http://www.omg.org/technology/documents/br_pm_spec_catalog.htm`. URL last verified: 2008-22-01.

[25] Fabio Casati, Stefano Ceri, Stefano Paraboschi, and Guiseppe Pozzi. Specification and Implementation of Exceptions in Workflow Management Systems. *ACM Trans. Database Syst.*, 24(3):405–451, 1999.

[26] Fabio Casati, Stefano Ceri, Barbara Pernici, and Giuseppe Pozzi. Workflow Evolution. In *Proceedings of the 15th International Conference on Conceptual Modeling*, pages 438–455, London, UK, 1996. Springer-Verlag.

[27] Fabio Casati and Giuseppe Pozzi. Modeling Exceptional Behaviors in Commercial Workflow Management Systems. In *COOPIS '99: Proceedings of the Fourth IECIS International Conference on Cooperative Information Systems*, Washington, DC, USA, 1999. IEEE Computer Society.

[28] S. Ceri, P. Grefen, and G. Sanchez. WIDE – a Distributed Architecture for Workflow Management. In *RIDE '97: Proceedings of the Seventh International Workshop on Research Issues in Data Engineering (RIDE '97) High Performance Database Management for Large-Scale Applications*, Washington, DC, USA, 1997. IEEE Computer Society.

[29] R. Cleaveland, G. Luettgen, V. Natarajan, and S. Sims. Modeling and Verifying Distributed Systems using Priorities: A Case Study. *Software Concepts and Tools*, 17:50–62, 1996.

[30] Rance Cleaveland and Matthew Hennessy. Priorities in Process Algebras. *Information and Computation*, 87(1-2):58–77, 1990.

[31] Thomas Connolly and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation and Management, ISBN:0321210255.* Addison-Wesley, 2004.

[32] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unravelling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, March 2002.

[33] A. Daskalopulu. *Logic-Based Tools for the Analysis and Representation of Legal Contracts.* PhD thesis, Imperial College, London, 1999.

[34] A. Daskalopulu. Modelling Legal Contracts as Processes. In *Proceedings of 11th International Conference and Workshop on Database and Expert Systems Applications*, pages 1074–1079. IEEE C.S. Press, 2000.

[35] ShuiGuang Deng, Zhen Yu, ZhaoHui Wu, and LiCan Huang. Enhancement of Workflow Flexibility by Composing Activities at Run-time. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 667–673, New York, NY, USA, 2004. ACM Press.

[36] Paulo Dias, Pedro Vieira, and Antonio Rito-Silva. Dynamic Evolution in Workflow Management Systems. In *DEXA '03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, Washington, DC, USA, 2003. IEEE Computer Society.

[37] Yang Dong and Zhang Shensheng. Modeling Workflow Patterns with π-calculus, unpublished, available from `http://www.workflow-research.de`. URL last verified: 2008-22-01.

[38] Edmund M. Clarke, Jr. and Orna Grumberg and Doron A. Peled. *Model Checking, ISBN: 0-262-03270-8.* MIT Press, Cambridge, MA, USA, 1999.

[39] Clarence A. Ellis and Karim Keddara. A Workflow Change Is a Workflow. In *Business Process Management, Models, Techniques, and Empirical Studies*, pages 201–217, London, UK, 2000. Springer-Verlag.

[40] Kutluhan Erol. *Hierarchical Task Network Planning: Formalization, Analysis and Implementation.* PhD thesis, The University of Maryland, 1995.

[41] Andrew D. H. Farrell, Marek J. Sergot, Mathias Sallé, and Claudio Bartolini. Using the Event Calculus for Tracking the Normative State of Contracts. *International Journal of Cooperative Information Systems*, 14(2-3):99–129, 2005.

[42] Andrew D. H. Farrell, Marek J. Sergot, David Trastour, and Athena Christodoulou. Performance Monitoring of Service-Level Agreements for Utility Computing Using the Event Calculus. In *WEC '04: Proceedings of the First IEEE International Workshop on Electronic Contracting (WEC'04)*, pages 17–24, Washington, DC, USA, 2004. IEEE Computer Society.

[43] Andrew D. H. Farrell, Marek J. Sergot, David Trastour, and Athena Christodoulou. Using the Event Calculus for the Performance Monitoring of Service-Level Agreements for Utility Computing. In *CoALa '04: Proceedings of the First IEEE International Workshop of Contract Architectures and Languages (CoALa 04)*, 2004.

[44] Howard Foster. *A Rigorous Approach To Engineering Web Service Compositions.* PhD thesis, Imperial College, London, 2006.

[45] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. LTSA-WS: A Tool for Model-Based Verification of Web Service Compositions and Choreography. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 771–774, New York, NY, USA, 2006. ACM Press.

[46] Martin Fowler. *UML Distilled, A Brief Guide to the Standard Object Modelling Language. ISBN: 0321193687.* Addison-Wesley, 2004.

[47] Xiang Fu, Tevfik Bultan, and Jianwen Su. WSAT: A Tool for Formal Analysis of Web Services. *Proceedings of 16th International Conference on Computer Aided Verification (CAV 2004).*

[48] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of Interacting BPEL Web Services. In *WWW '04: Proceedings of the 13th International Conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM.

[49] Bruns G. *Distributed Systems Analysis with CCS, ISBN: 0-13-398389-7.* Prentice-Hall, 1997.

[50] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, April 1995.

[51] Dimitrios Georgakopoulos, Hans Schuster, Donald Baker, and Andrzej Cichocki. Managing Escalation of Collaboration Processes in Crisis Mitigation Situations. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, Washington, DC, USA, 2000. IEEE Computer Society.

[52] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning : Theory & Practice, ISBN: 1558608567.* Morgan Kaufmann, 2004.

[53] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic Causal Theories. *Artificial Intelligence*, 153:49–104, 2004.

[54] Alvin I Goldman. *A Theory Of Human Action, ISBN: 0139144404.* Prentice-Hall, Englewood Cliffs, NJ, 1970.

[55] Paul W. P. J. Grefen, Karl Aberer, Heiko Ludwig, and Yigal Hoffner. CrossFlow: Cross-Organizational Workflow Management for Service Outsourcing in Dynamic Virtual Enterprises. *IEEE Data Engineering Bulletin*, 24(1):52–57, 2001.

[56] Michael Havey. *Essential Business Process Modeling, ISBN: 0-596-00843-0.* O'Reilly, 2005.

[57] Jan Hidders, Marlon Dumas, Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, and Jan Verelst. When Are Two Workflows the Same? In Mike Atkinson and Frank Denhe, editors, *Proceedings Computing: The Australasian Theory Symposium, Newcastle, NSW, Australia*, pages 3–11, 2005.

[58] Wesley N. Hohfeld. *Fundamental Legal Conceptions as Applied in Judicial Reasoning, ISBN: 185521668X.* Dartmouth Pub Co, 2002.

[59] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual, ISBN: 0-321-22862-6.* Addison-Wesley, 2004.

[60] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP '98: Proceedings of the Seventh European Symposium on Programming*, pages 122–138, London, UK, 1998. Springer-Verlag.

[61] Michael Huth and Mark Ryan. *Logic in Computer Science, ISBN: 0-521-65602-8.* Cambridge University Press, 2000.

[62] S. Jablonski and C. Bussler. *Workflow Management - Modeling Concepts, Architecture and Implementation, ISBN: 1850322228.* International Thomson Computer Press, September 1996.

[63] Andrew Jones and Marek Sergot. A Formal Characterisation of Institutionalised Power. *Logic Journal of the IGPL*, 4(3), 1996.

[64] B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows.* PhD thesis, Queensland University of Technology, Brisbane, Australia, 2003.

[65] Mariya Koshkina and Franck van Breugel. Verification of Business Processes for Web Services, CS-2003-11. Technical report, Department of Computer Science, York University, Toronto, 2003.

[66] Mariya Koshkina and Franck van Breugel. Modelling and Verifying Web Service Orchestration by means of the Concurrency Workbench. *SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[67] Akhil Kumar, Wil M P van der Aalst, and Eric M W Verbeek. Dynamic Work Distribution in Workflow Management Systems: How to Balance Quality and Performance. *Journal of Management Information Systems*, 18(3):157–194, 2002.

[68] John Lee and Ron Ben-Natan. *Integrating Service Level Agreements: Optimizing Your OSS for SLA Delivery, ISBN: 0471210129.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[69] F. Leymann. Web Services Flow Language (WSFL 1.0), IBM (2001), at: `http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`. URL last verified: 2008-22-01.

[70] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques, ISBN: 0130217530.* Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.

[71] Lars Lindahl. *Position And Change, A Study in Law and Logic; Synthese Library Volume 122.* D. Reidel Publishing Company, 1977.

[72] David Makinson. On the Formal Representation of Rights Relations: Remarks on the Work of Stig Kanger and Lars Lindahl. *The Journal of Philosophical Logic*, 15:403–425, 1986.

[73] Peter Mangan and Shazia Sadiq. A Constraint Specification Approach to Building Flexible Workflows. *Journal of Research and Practice in Information Technology*, 34(3), 2002.

[74] Mike Marin. Business Process Technology: From EAI and Workflow to BPM. In Layna Fischer, editor, *The Workflow Handbook 2002, ISBN:0-9703509-2-9.* Future Strategies.

[75] Olivera Marjanovic. Managing the Normative Context of Composite E-services. In *Proceedings of the International Conference on Web Services, (ICWS-Europe'2003), Erfurt, Germany, 2003*, pages 24–36.

[76] John McCarthy. Situations, Actions and Causal Laws. Technical Report, Stanford University, 1963. *Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pp. 410-417.*

[77] John McCarthy and Patrick J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[78] Robin Milner. *Communication and Concurrency, ISBN: 0-13-115007-3.* Prentice Hall, 1989.

[79] Robin Milner. Operational and Algebraic Semantics of Concurrent Processes in *Handbook of Theoretical Computer Science (Vol. B): Formal Models and Semantics, ISBN:0-444-88074-7.* pages 1201–1242, 1990.

[80] Robin Milner. *Communicating and Mobile Systems: The π-Calculus, ISBN:0-521-64320-1.* Cambridge University Press, 1999.

[81] Z. Milosevic, S. Gibson, P. F. Linington, J. Cole, and S. Kulkarni. On Design and Implementation of a Contract Monitoring Facility. In *WEC '04: Proceedings of the First IEEE International Workshop on Electronic Contracting (WEC'04)*, pages 62–70, Washington, DC, USA, 2004. IEEE Computer Society.

[82] Carlos Molina-Jiménez, Santosh K. Shrivastava, Ellis Solaiman, and John P. Warne. Runtime Monitoring and Enforcement of Electronic Contracts. *Electronic Commerce Research and Applications*, 3(2):108–125, 2004.

[83] S. Nakajima. Verification of Web Service Flows with Model-Checking Techniques. *CW '02: Proceedings of the First International Symposium on Cyber Worlds (CW'02)*, pages 3–78, 2002.

[84] Shin Nakajima. Model-Checking Behavioral Specification of BPEL Applications. *Electronic Notes on Theoretical Computer Science*, 151(2):89–105, 2006.

[85] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[86] Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services, ISBN: 0-321-18086-0.* Addison-Wesley, 2005.

[87] OASIS. Web Services Business Process Execution Language Version 2.0, OASIS Standard, 11th April 2007, at: `http://www.oasis-open.org/apps/org/workgroup/wsbpel`. URL last verified: 2008-22-01.

[88] O.Marjanovic and Z.Milosevic. Towards Formal Modelling of e-Contracts. In *Proceedings of Fifth International Enterprise Distributed Object Computing Conference (EDOC 2001), 4-7 September 2001, Seattle, WA, USA*, pages 59–68. IEEE Computer Society, 2001.

[89] Adrian Paschke. RBSLA A Declarative Rule-based Service Level Agreement Language Based on RuleML. In *CIMCA '05: Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce Vol-2 (CIMCA-IAWTIC'06)*, pages 308–314, Washington, DC, USA, 2005. IEEE Computer Society.

[90] Richard E. Pattis. EBNF: A Notation to Describe Syntax, at: `http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf`. URL last verified: 2008-22-01.

[91] Marco Pistore, Marco Roveri, and Paolo Busetta. Requirements-Driven Verification of Web Services. *Electronic Notes in Theoretical Computer Science*, 105:95–108, 2004.

[92] Jeremy Pitt, Lloyd Kamara, Marek Sergot, and Alexander Artikis. Formalization of a Voting Protocol for Virtual Organizations. In *AAMAS '05: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 373–380, New York, NY, USA, 2005. ACM Press.

[93] David Portabella Clotet, Vincenzo Pallotta, and Martin Rajman. Systematic Definition and Assent to eContracts for Web Services. In *CoAla 2005 Workshop on Contract Architectures and Languages*, Enschede, The Netherlands, September 2005.

[94] Frank Puhlmann and Mathias Weske. Using the π-calculus for Formalizing Workflow Patterns. In W.M.P. van der Aalst et al, editor, *Business Process Management (BPM) 2005*, volume 3649 of *Lecture Notes in Computer Science*. Springer, 2005.

[95] Manfred Reichert and Peter Dadam. Adept-flex – Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.

[96] H.A. Reijers, J. Rigter, and W. van der Aalst. The Case Handling Case. *International Journal of Cooperative Information Systems*, 12(3):365–391, 2003.

[97] Wolfgang Reisig and Grzegorz Rozenberg. *Lectures on Petri Nets I: Basic Models, ISBN:3-540-65307-4.* Springer, 1998.

[98] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, ISBN: 0-262-18218-1.* The MIT Press, 2001.

[99] R.Kowalski and M.Sergot. A Logic-Based Calculus of Events. *New Generation Computing,* 4:67–95, 1986.

[100] Hamish Ross. Hohfeld and the Analysis of Rights (Chapter 13). In James Penner, David Schiff, and Richard Nobles, editors, *Jurisprudence & Legal Theory: Commentary and Materials, ISBN: 0-406-94678-7.* 2002.

[101] Ronald G. Ross. *Principles of the Business Rule Approach (Paperback), ISBN: 0201788934.* Addison-Wesley Professional; First Edition, 2003.

[102] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, ISBN: 0130803022.* Prentice-Hall, Englewood Cliffs, NJ, second edition, 2003.

[103] S. Thatte. XLANG: Web Services for Business Process Design, now defunct, see: `http://en.wikipedia.org/wiki/Xlang`. URL last verified: 2008-22-01.

[104] F. B. Sadighi, M. J. Sergot, and O. Bandemann. Using Authority Certificates to Create Management Structures. In Bruce Christianson, Bruno Crispo, James A. Malcolm, and Michael Roe, editors, *Security Protocols. Ninth International Workshop, Cambridge, April 2001,* LNCS 2467, pages 134–145. Springer, 2002.

[105] Shazia W. Sadiq, Maria E. Orlowska, and Wasim Sadiq. Specification and Validation of Process Constraints for Flexible Workflows. *Information Systems,* 30(5):349–378, 2005.

[106] Gwen Salaun, Lucas Bordeaux, and Marco Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04),* Washington, DC, USA, 2004. IEEE Computer Society.

[107] Hans Schuster, Donald Baker, Andrzej Cichocki, Dimitrios Georgakopoulos, and Marek Rusinkiewicz. The Collaboration Management Infrastructure. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering,* Washington, DC, USA, 2000. IEEE Computer Society.

[108] John R Searle. What is a Speech Act? In *The Philosophy of Language,* pages 130–141. 1965.

[109] John R Searle. *The Construction of Social Reality, ISBN: 0-02-928045-1.* The Free Press, 1995.

[110] Marek Sergot. The language $(C+)^{++}$. In J. Pitt, editor, *The Open Agent Society.* Wiley, 2005. (In press). Extended version: Technical Report 2004/8. Department of Computing, Imperial College, London.

[111] Murray Shanahan. *Solving the Frame Problem. A Mathematical Investigation of the Common Sense Law of Inertia, ISBN: 0-262-19384-1.* The MIT Press, 1997.

[112] Murray Shanahan. The Event Calculus Explained. In M.J.Wooldridge and M.Veloso, editors, *Artificial Intelligence Today, Lecture Notes in Artificial Intelligence*, volume 1660, pages 409–430. Springer, 1999.

[113] S. Sims. The Process Algebra Compiler User's Manual, at: `http://www.reactive-systems.com/pac`. URL last verified: 2008-22-01. 1999.

[114] Michael Sipser. *Introduction to the Theory of Computation, ISBN:0619217642.* Thomson Course Technology, 2006.

[115] Ellis Solaiman, Carlos Molina-Jiménez, and Santosh K. Shrivastava. Model Checking Correctness Properties of Electronic Contracts. In *Proceedings of First International Conference Service Oriented Computing (ICSOC 2003), Trento, Italy, December 15–18*, pages 303–318, 2003.

[116] Christian Stefansen. A SMAll Workflow Language based on CCS, TR-06-05. In *Proceedings of 17th Conference on Advanced Information Systems Engineering, CAiSE05, to appear*, 2005.

[117] Christian Stefansen. A SMAll Workflow Language based on CCS, TR-06-05. Technical report, Harvard University, Division of Engineering and Applied Sciences, Cambridge, MA 02138, March 2005.

[118] Kishor Shridharbhai Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications, Second Edition, ISBN: 0471333417.* Wiley-Interscience (October 26, 2001).

[119] W. M. P. van der Aalst and S. Jablonski. Dealing with Workflow Change: Identification of Issues and Solutions. *International Journal of Computer Systems Science and Engineering*, 15(5):267–276, September 2000.

[120] W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management, BPM Center Report BPM-04-03. Technical report, BPMcenter.org, 2004.

[121] W.M.P. van der Aalst. Don't Go With the Flow: Web Services Composition Exposed. In *Trends and Controversies. Web Services: Been there, Done that? IEEE Intelligent Systems*, pages 72–76, Jan–Feb 2003.

[122] W.M.P. van der Aalst. π-calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "π hype". *BPTrends*, 3(5):1–11, May 2005.

[123] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and Implementation of the YAWL system. In *Proceedings of The 16th International Conference on Advanced Information Systems Engineering (CAiSE 04), Riga, Latvia.* Springer Verlag, June 2004.

[124] W.M.P. van der Aalst, A. Hofstede, and M. Weske. Business Process Management: A Survey. *Business Process Management (BPM)*, 2003.

[125] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language (Revised Version). *Information Systems*, 30(4):245–275, 2005.

[126] W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages. In K. Jensen, editor, *Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002), volume 560 of DAIMI, Aarhus, Denmark*, pages 1–20, August 2002.

[127] W.M.P. van der Aalst and Mathias Weske. Case Handling: a New Paradigm for Business Process Support. *Data Knowledge Engineering*, 53(2):129–162, 2005.

[128] H. M. W. Verbeek, T. Basten, and W. M. P. van der Aalst. Diagnosing Workflow Processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.

[129] W3C. Web Service Choreography Interface (WSCI) 1.0, at: `http://www.w3.org/TR/wsci/`. URL last verified: 2008-22-01.

[130] W3C Recommendation. Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C Recommendation 16 August 2006, edited in place 29 September 2006, at: `http://www.w3.org/TR/2006/REC-xml-20060816/`. URL last verified: 2008-22-01.

[131] Jacques Wainer. Logic Representation of Processes in Work Activity Coordination. In *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, pages 203–209, New York, NY, USA, 2000. ACM Press.

[132] Jacques Wainer, Fabio Bezerra, and Paulo Barthelmess. Tucupi: a Flexible Workflow System Based on Overridable Constraints. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 498–502, New York, NY, USA, 2004. ACM Press.

[133] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture, ISBN: 0-13-148874-0.* Prentice Hall, 2005.

[134] M. Weske. Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 7*, Washington, DC, USA, 2001. IEEE Computer Society.

[135] Workflow Management Coalition. XML Process Definition (XPDL) Language, at: `http://www.wfmc.org/standards/xpdl.htm`. URL last verified: 2008-22-01.

[136] Workflow Management Coalition. Workflow Management Coalition Terminology & Glossary. Document Number: WFMC-TC-1011. Document Status: Issue 3.0. February 1999.

[137] WS-CDL W3C Working Group. Web Services Choreography Description Language Version 1.0 W3C Working Draft 17 December 2004, at: `http://www.w3.org/TR/ws-cdl-10`. URL last verified: 2008-22-01.

[138] Shengli Wu, Amit Sheth, John Miller, and Zongwei Luo. Authorization and Access Control of Application Data in Workflow Systems. *Journal of Intelligent Information Systems*, 18(1):71–94, 2002.

[139] WWW Consortium. Web Services Architecture Requirements at (October 2002): `http://www.w3c.org/TR/wsa-reqs`. URL last verified: 2008-22-01.

[140] Moe Thandar Wynn, David Edmond, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Achieving a General, Formal and Decidable Approach to the OR-join in Workflow using Reset nets. In Gianfranco Ciardo and Philippe Darondeau, editors, *Applications and Theory of Petri Nets 2005, 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005, Proceedings*, volume 3536 of *Lecture Notes in Computer Science*. Springer, 2005.

[141] Michael zur Meuhlen. Organisational Management in Workflow Applications – Issues and Perspectives. *Information Technology and Management*, 5(4), 2004.

Appendix A

PCCS Characterisation – Additional Information

In this appendix, we present the rest of our PCCS-based characterisation of `Liesbet`.

A.1 Cancellation of Basic Instances

According to the description of `Liesbet`, presented in Section 3.1.2, basic activity instances may be cancelled, as well as being completed. Dispensation for this is easily introduced into our PCCS-based characterisation of `Liesbet`. All that is required is a modification to the definition of $\texttt{Basics}^b$ agents, to also allow for cancellation of basic instances. We present the following PCCS-based definition of $\texttt{Basics}^b$, for the case where b is 3. We simply offer the choice of cancelling an instance, as well as offering the choice of completing it.

```
proc Basics3 =
   comp1:20.Basics3 + canc1:20.Basics3 +
   comp2:20.Basics3 + canc2:20.Basics3 +
   comp3:20.Basics3 + canc3:20.Basics3
```

A.2 SeqCancel

The characterisation of `SeqCancel` is tricky because we need to ensure that, when a child instance of such a type is cancelled, the parent `SeqCancel` instance is also cancelled. The most straightforward way of doing this is to have a distinct set of tracker agents for *children* of `SeqCancel` types, viz. $\texttt{InitialStateSC}^n$ and $\texttt{RunningStateSC}^n$. The definition of $\texttt{InitialStateSC}^n$ is the same as $\texttt{InitialState}^n$, except that it evolves to $\texttt{RunningStateSC}^n$, instead of $\texttt{RunningState}^n$. The crucial difference lies in the definition of $\texttt{RunningStateSC}^n$, where the line relating to accepting synchronisations on canc is changed, thus.

```
proc RunningStateSCn =
   ...
   canc:3.'pcanc:5.CancelledState + canc:10.'pcanc:5.CancelledState
```

Here, instead of effecting a synchronisation on `prec` (which would effect a `precompletion` step in the parent instance), we cancel the parent instance. The generic type agent for `SeqCancel` is the same as that for `Seq`; in fact, the translator just outputs `Seq` for `SeqCancel` types. Examples of the translation of `SeqCancel` types are given in the listings contained within Appendix Section A.4.

A.3 Synchronisation Types

We present support for synchronisation types, for now, with the caveat that just monotonic querying (see Section 5.1.1) is allowed, with no support for distinct queries. We relax this restriction later. It is useful to present two separate cases, as the one presented here is simpler to understand and may be sufficient.

The characterisation of synchronisation activity types is a non-trivial task. This is because any state-querying channels belonging to (tracker agents of) activity instances within the visibility horizons of queries within a synchronisation activity instance may (potentially) be used in these queries. Thus, we must build custom agents, effecting the logic of synchronisation instances, where the apposite channels are made available to the mapped queries. This is carried out in an additional translation step, which we shall call *Step 3*.

For use in *Step 1* of the translation process, $\mathcal{M}_{pccs}[\![-]\!]$ is defined as follows for synchronisation types. These types are: `Stop(StopQuery, GoQuery)`, `Stop(StopQuery)`, `Go(StopQuery,GoQuery)` and `Go(GoQuery)`.

- $\mathcal{M}_{pccs}[\![\texttt{Stop(StopQuery,GoQuery)}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i, \texttt{pcald}_i) =$
  $\texttt{InitialState}^0[SC_i, {}^{\texttt{pprec}_i}/_{\texttt{pprec}}, {}^{\texttt{pcald}_i}/_{\texttt{pcald}}]$
  |
  $STOP(\mathcal{QT}[\![\texttt{StopQuery}]\!](st\_chs_i \rightarrow), \mathcal{QT}[\![\texttt{GoQuery}]\!](st\_chs_i \rightarrow))[SC_i]$
- $\mathcal{M}_{pccs}[\![\texttt{Stop(StopQuery)}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i, \texttt{pcald}_i) =$
  $\texttt{InitialState}^0[SC_i, {}^{\texttt{pprec}_i}/_{\texttt{pprec}}, {}^{\texttt{pcald}_i}/_{\texttt{pcald}}]$
  |
  $STOP(\mathcal{QT}[\![\texttt{StopQuery}]\!](st\_chs_i \rightarrow))[SC_i]$
- $\mathcal{M}_{pccs}[\![\texttt{Go(StopQuery,GoQuery)}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i, \texttt{pcald}_i) =$
  $\texttt{InitialState}^0[SC_i, {}^{\texttt{pprec}_i}/_{\texttt{pprec}}, {}^{\texttt{pcald}_i}/_{\texttt{pcald}}]$
  |
  $GO(\mathcal{QT}[\![\texttt{StopQuery}]\!](st\_chs_i \rightarrow), \mathcal{QT}[\![\texttt{GoQuery}]\!](st\_chs_i \rightarrow))[SC_i]$
- $\mathcal{M}_{pccs}[\![\texttt{Go(GoQuery)}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i, \texttt{pcald}_i) =$
  $\texttt{InitialState}^0[SC_i, {}^{\texttt{pprec}_i}/_{\texttt{pprec}}, {}^{\texttt{pcald}_i}/_{\texttt{pcald}}]$
  |
  $GO(\mathcal{QT}[\![\texttt{GoQuery}]\!](st\_chs_i \rightarrow))[SC_i]$

The auxiliary functions, *STOP* and *GO*, take one or two arguments, which are the translated queries. Note that when they are used, as part of *Step 1* of the translation process, the arguments are placeholders. These placeholders are filled in as part of *Step 3* of the translation process. The purpose of *STOP* and *GO* is to construct the customised agent effecting the logic of the translated synchronisation type, and are defined as follows.

- *STOP*(qtStopQuery, qtGoQuery)=

  (qtStopQuery[$^{done_s}/_{done}$] | dones:4.'lose:5.nil | qtGoQuery[$^{done_g}/_{done}$] | doneg:4.'win:6.nil
  |
  lose:5.'canc:10.nil + win:6.'comp:10.nil)\{doneg, dones, win, lose}[> 'find:4.nil

- *STOP*(qtStopQuery)=

  (qtStopQuery[$^{done_s}/_{done}$] | dones:4.'canc:10.nil)\{dones}[> 'find:4.nil

- *GO*(qtStopQuery, qtGoQuery)=

  (qtGoQuery[$^{done_g}/_{done}$] | doneg:4.'win:5.nil | qtStopQuery[$^{done_s}/_{done}$] | dones:4.'lose:6.nil
  |
  win:5.'comp:10.nil + lose:6.'canc:10.nil)\{doneg, dones, win, lose}[> 'find:4.nil

- *GO*(qtGoQuery)=

  (qtGoQuery[$^{done_g}/_{done}$] | doneg:4.'comp:10.nil)\{doneg}[> 'find:4.nil

Step 3 of the translation process is concerned with filling in the queries, qtStopQuery and qtGoQuery, in the customised agents that we have built with *STOP/GO* in *Step 1* of the translation process. The translation function, $\mathcal{QT}[\![-]\!]$, is responsible for translating these queries. Its definition makes use of four relations that are constructed during *Step 1* of the translation process. These relations are as follows.

- CotdInScope – gives the Completed state querying channels which are in a particular visibility horizon of a querying instance.

- CaldInScope – gives the Cancelled channels ....

- FindInScope – gives the *Finished* channels ....

- NInitInScope – gives the *Not Initial* channels ....

The arguments of CotdInScope are:

- cotds – the Completed state querying channel of the querying instance (the *source* instance).

- cotdt – the Completed channel of the *target* instance, which would be in some visibility horizon of the source instance.

- rtype – the reference customised activity type (see Section 3.1.3). This is the type of a common ancestor instance of the source and target instances.

- ctype – the customised activity type of the target instance.

The presented relations are updated as we move through the workflow model, translating nodes with $\mathcal{M}_{pccs}[\![-]\!]$, as part of *Step 1*. The semantics of these relations exactly matches those of the InScope relation, presented in Appendix Section B.1.1. The prescription for updating CotdInScope is as follows. Note that $\mathcal{M}_{pccs}[\![-]\!]$ also records the activity types and parent/ancestor/descendant information of instances, as they are translated.

- If we are adding an instance with Completed channel cotdi (which is passed into $\mathcal{QT}[\![-]\!]$ with all of the instance's state channels), and parent Completed channel cotdp, then we may assert CotdInScope(cotdi,cotdt,rtype,ctype) IF

  - There is an instance with `Completed` channel `cotdt` within the visibility horizon of the parent instance such that `CotdInScope(cotdp,cotdt,rtype,ctype)` is already asserted OR
  - The parent instance itself is of customised activity type `rtype` and `cotdt` is a descendant of `cotdp`, where `cotdt` is of customised activity type `ctype`.

- If we are adding an instance with `Completed` channel `cotdi`, customised activity type `ctype`, and parent `Completed` channel `cotdp`, then we may assert `CotdInScope(cotds,cotdi,rtype,ctype)` IF `CotdInScope(cotds,cotdp,rtype,ctype')` is already asserted, for some `ctype'`.

The first of the two alternatives for asserting a new instance of the `CotdInScope` relation extend the visibility horizon of the parent down to the newly added instance. The second alternative adds the newly added instance to the visibility horizons of all instances that already exist. Note that if the scope of the instance being added is *isolated* then there will not exist any instances of the `CotdInScope` relation for that instance. More information concerning the treatment of isolated scopes is presented, for the `SitCalc`-based characterisation, in Appendix Section B.1.1.

Identical definitions exist for the other three relations, `CaldInScope`, `FindInScope`, and `NInitInScope`, based on `cald`, `find` and `ninit` channel types, respectively.

The definition of $\mathcal{QT}[\![-]\!]$, which acts on the queries of `Liesbet` synchronisation types, is now presented. It is inductively defined, as queries may be composite. Note, $\sum_{c\in\{c_1,\ldots,c_n\}} f(c)$ is the summation $f(c_1)+\ldots+f(c_n)$, $\prod_{c\in\{c_1,\ldots,c_n\}} f(c)$ is the prefix sequence $f(c_1).\ \ldots\ .f(c_n)$. Regarding atomic queries, we present definitions for the `Completed` state only. In these definitions, we make use of the channel `cotdi`, which is the `Completed` state querying channel for the querying instance. It is passed into $\mathcal{QT}[\![-]\!]$ along with all of the state channels for the instance. The definitions of $\mathcal{QT}[\![-]\!]$ for queries relating to other states easily follow.

- $\mathcal{QT}[\![\texttt{True}]\!](st\_chs_i \rightarrow) =$

  `'done:4.nil`

- $\mathcal{QT}[\![\texttt{False}]\!](st\_chs_i \rightarrow) =$

  `nil`

- $\mathcal{QT}[\![\texttt{Completed\_act}(\phi)]\!](st\_chs_i \rightarrow) =$

  $\sum_{c\in C}$ `'c:5.'done:4.nil`

  where, for ϕ being `qtype`,

  C = { `cotdt | ∃rtype.CotdInScope(cotdi, cotdt, rtype, qtype)` }

  and for ϕ being `qtype IN rtype`

  C = { `cotdt | CotdInScope(cotdi, cotdt, rtype, qtype)` }

- $\mathcal{QT}[\![\texttt{Completed\_all}(\phi)]\!](st\_chs_i \rightarrow) =$

  $(\prod_{c\in C}$ `'c:5).'done:4.nil`

  where, for ϕ being `qtype`,

  C = { `cotdt | ∃rtype.CotdInScope(cotdi, cotdt, rtype, qtype)` }

  and for ϕ being `qtype IN rtype`

  C = { `cotdt | CotdInScope(cotdi, cotdt, rtype, qtype)` }

- $\mathcal{QT}[\![\texttt{Q}_1|\ldots|\texttt{Q}_n]\!](st\_chs_i \rightarrow) =$

  $(\mathcal{QT}[\![\texttt{Q}_1]\!](st\_chs_i \rightarrow)[^{\texttt{done}_1}/_{\texttt{done}}] \mid \ldots \mid \mathcal{QT}[\![\texttt{Q}_n]\!](st\_chs_i \rightarrow)[^{\texttt{done}_1}/_{\texttt{done}}]$

|

done1:4.$^{n\ copies}$....done1:4.'done:4.nil)\{done1}

- $\mathcal{QT}[\![\mathtt{Q}_1+\ldots+\mathtt{Q}_n]\!](st\_chs_i \rightarrow) =$
  $(\mathcal{QT}[\![\mathtt{Q}_1]\!](st\_chs_i \rightarrow)[^{\mathtt{done}_1}/_{\mathtt{done}}] \mid \ldots \mid \mathcal{QT}[\![\mathtt{Q}_n]\!](st\_chs_i \rightarrow)[^{\mathtt{done}_1}/_{\mathtt{done}}]$
  |
  done1:4.'done:4.nil)\{done1}

A.4 Model Checking Example

We now present examples of model checking a `Liesbet2` model for the two key properties related to soundness for `Liesbet` models, described in Section 7.1, viz. absence of dead activity instances, and an absence of deadlock. For `Liesbet2`, an absence of deadlock guarantees completion along all enactment paths. We start with an example showing model checking for an absence of dead instances.

A.4.1 Dead Activity Instance Detection

Consider the following `Liesbet2` model.

```
Par(Choice(Empty, A, Empty, B), C)
C = Act(join(Go(Finished_act(A) | Finished_act(B),
  Completed_act(A) | Completed_act(B))))
```

In this model, activity `C` is never executed, as its join condition will always fail. As such, it counts as a dead instance. This is because either activity `A` or activity `B` will be executed by the `Choice` but not both, where the requirement for `C` to run is that instances of both `A` and `B` have previously completed successfully.

In order to detect the occurrence of dead activity instances, we add an output on an unrestricted channel, `dead`, in the definitions of `InitialState`^{n} agents. The output occurs once the model has finished (as indicated by a synchronisation on `find_0`), if the instance went straight from an `Initial` state to a `Cancelled` state. The appropriate definition of `InitialState0` would be as follows.

```
proc InitialState0 =
   'pcald:5.('find_0:10.'dead:10.nil | CancelledState) +
   canc:3.'pprec:5.('find_0:10.'dead:10.nil | CancelledState) +
   canc:10.'pprec:5.('find_0:10.'dead:10.nil | CancelledState) +
   exec:3.RunningState0
```

In this approach, we may only test a single instance at a time as to whether it is a dead instance. This is not, typically, much of a disadvantage, as it is often clear which instances are likely to be susceptible to being dead instances.

We test the model against a proposition which is a slight modification to the `cotd` proposition used in Section 5.6.1. Instead of testing for the root instance finishing along all enactment paths, we test for the occurrence of a transition on `dead`, appropriately relabelled, along all enactment paths.

In the following example, we wish to check whether the instance C is a dead instance; as such, we relabel its dead channel (to something like deadc), and check for its occurrence along all enactment paths.

```
prop deadc =
   min X = <->tt ∧ [-'deadc:10]X
```

The model translated by $\mathcal{M}_{pccs}[\![-]\!]$ yields the following PCCS source, where we omit the definitions of certain tracker and generic agent types for brevity.

```
****************************************
* PCCS Verification Run **************
* # 0
* Generated from: file:samples/LiesbetDeadInsts.liesbet
* On: Fri Jul 14 12:31:18 BST 2006

proc InitialState0 =
   'pcald:5.('find_0:10.'dead:10.nil | CancelledState) +
   canc:3.'pprec:5.('find_0:10.'dead:10.nil | CancelledState) +
   canc:10.'pprec:5.('find_0:10.'dead:10.nil | CancelledState) +
   exec:3.RunningState0

...appropriate tracker and generic type agents...

proc Workflow0 =
   (
***Instance:0:P1
   InitialState2[runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, nread_0/nread, comp_0/comp, canc_0/canc, exec_0/exec,
      prec_0/prec, cald_0/pcald] |

   Par2[runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, nread_0/nread, comp_0/comp, canc_0/canc, exec_0/exec,
      exec_1/exec1, exec_6/exec2] |

***Instance:1:CH
   InitialState4[runn_1/runn, cald_1/cald, cotd_1/cotd,
      find_1/find, nread_1/nread, comp_1/comp, canc_1/canc, exec_1/exec,
      prec_1/prec, prec_0/pprec, cald_0/pcald] |

   Choice2[runn_1/runn, cald_1/cald, cotd_1/cotd,
      find_1/find, nread_1/nread, comp_1/comp, canc_1/canc, exec_1/exec,
      exec_2/execg1, exec_3/execc1, canc_3/cancc1, canc_2/cancg1, cotd_2/cotdg1, cald_2/caldg1,
      exec_4/execg2, exec_5/execc2, canc_5/cancc2, canc_4/cancg2, cotd_4/cotdg2, cald_4/caldg2] |

***Instance:2:Em1
   InitialState0[runn_2/runn, cald_2/cald, cotd_2/cotd,
```

```
      find_2/find, nread_2/nread, comp_2/comp, canc_2/canc, exec_2/exec,
      prec_1/pprec, cald_1/pcald] |

   Empty[runn_2/runn, cald_2/cald, cotd_2/cotd,
      find_2/find, nread_2/nread, comp_2/comp, canc_2/canc, exec_2/exec] |

***Instance:3:A
   InitialState0[runn_3/runn, cald_3/cald, cotd_3/cotd,
      find_3/find, nread_3/nread, comp_3/comp, canc_3/canc, exec_3/exec,
      prec_1/pprec, cald_1/pcald, deada/dead] |

***Instance:4:Em2
   InitialState0[runn_4/runn, cald_4/cald, cotd_4/cotd,
      find_4/find, nread_4/nread, comp_4/comp, canc_4/canc, exec_4/exec,
      prec_1/pprec, cald_1/pcald] |

   Empty[runn_4/runn, cald_4/cald, cotd_4/cotd,
      find_4/find, nread_4/nread, comp_4/comp, canc_4/canc, exec_4/exec] |

***Instance:5:B
   InitialState0[runn_5/runn, cald_5/cald, cotd_5/cotd,
      find_5/find, nread_5/nread, comp_5/comp, canc_5/canc, exec_5/exec,
      prec_1/pprec, cald_1/pcald, deadb/dead] |

***Instance:6:JOIN_SEC_C
   InitialState2[runn_6/runn, cald_6/cald, cotd_6/cotd,
      find_6/find, nread_6/nread, comp_6/comp, canc_6/canc, exec_6/exec,
      prec_6/prec, prec_0/pprec, cald_0/pcald] |

   Seq[runn_6/runn, cald_6/cald, cotd_6/cotd,
      find_6/find, nread_6/nread, comp_6/comp, canc_6/canc, exec_6/exec,
      exec_7/exec2, find_7/find2, exec_8/exec1] |

***Instance:7:CJoin

   InitialStateSC0[runn_4/runn, cald_4/cald, cotd_4/cotd,
      find_4/find, nread_4/nread, comp_4/comp, canc_4/canc, exec_4/exec,
      prec_1/pprec, cald_1/pcald, canc_6/pcanc] |

 (
   (
***GoQuery
    (
     ('cotd_5:5.'done1:4.nil  | 'cotd_3:5.'done1:4.nil  | done1:4.done1:4.'done0:4.nil)\{done1} |
        done0:4.'win:5.nil)\{done0}  |
***StopQuery
    (
```

```
    ('find_5:5.'done3:4.nil  | 'find_3:5.'done3:4.nil  | done3:4.done3:4.'done2:4.nil)\{done3} |
       done2:4.'lose:6.nil)\{done2}  |
***Go: GoQuery takes priority
   lose:6.'canc_2:10.nil +win:5.'comp_2:10.nil
  )\{win, lose} [> 'find_2:5.nil
 ) |

***Instance:8:C
   InitialStateSC0[runn_8/runn, cald_8/cald, cotd_8/cotd,
      find_8/find, nread_8/nread, comp_8/comp, canc_8/canc, exec_8/exec,
      prec_6/pprec, cald_6/pcald, canc_6/pcanc, deadc/dead] |

Basics3[comp_3/comp1, comp_5/comp2, comp_8/comp3] |

'exec_0:3.pprec:5.nil | 'find_0:10.'rfind:10.nil

)\{
   runn_0, cald_0, cotd_0, find_0, nread_0, comp_0, canc_0, exec_0, prec_0,
   runn_1, cald_1, cotd_1, find_1, nread_1, comp_1, canc_1, exec_1, prec_1,
   runn_2, cald_2, cotd_2, find_2, nread_2, comp_2, canc_2, exec_2, prec_2,
   runn_3, cald_3, cotd_3, find_3, nread_3, comp_3, canc_3, exec_3, prec_3,
   runn_4, cald_4, cotd_4, find_4, nread_4, comp_4, canc_4, exec_4, prec_4,
   runn_5, cald_5, cotd_5, find_5, nread_5, comp_5, canc_5, exec_5, prec_5,
   runn_6, cald_6, cotd_6, find_6, nread_6, comp_6, canc_6, exec_6, prec_6,
   runn_7, cald_7, cotd_7, find_7, nread_7, comp_7, canc_7, exec_7, prec_7,
   runn_8, cald_8, cotd_8, find_8, nread_8, comp_8, canc_8, exec_8, prec_8,
   dead, pprec, pcald}
```

The output of the test, under CWB-NC, reveals that C is indeed a dead activity instance.

```
cwb-nc> chk Workflow0 deadc
Invoking alternation-free model checker.
Building automaton...
.....
States: 526
Transitions: 830
Done building automaton.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(11.375,0.000,0.015,11.375)
```

It is also instructive to highlight the translation of the Go instance for this model, which is instance 7 in the presented source. Here, we seek to ascertain that either:

- Instances 3 and 5 have completed, in which case we win. Or, that
- Instances 3 and 5 have finished, in which case we lose.

As `GoQuerys` take precedence over `StopQuerys` in `Go` types, as realised by the differing priorities on `win` and `lose`, if the first of these scenarios holds (i.e. we win) then we complete the synchronisation instance. If the second scenario holds (i.e. we lose), but not the first, we cancel the synchronisation instance. The disabling operator is used to garbage-collect the residual logic, once one of these eventualities occurs.

A.4.2 PCCS Example of Deadlock Detection

Consider the following two `Liesbet` workflow models.

```
Par(Seq(A, B, C), Seq(D, E, F)
B = Act(join(Go(Completed_act(E))))
E = Act(join(Go(Completed_act(B))))

Par(Seq(A, B, C), Seq(D, E, F)
B = Act(join(Go(Completed_act(E))))
```

The first of these contains an obvious source of deadlock. That being, the execution of `B` may only commence once the (single) instance of `E` has completed. But, the execution of `E` may only commence once the (single) instance of `B` has completed. The second model removes the latter constraint and should complete normally.

The PCCS source for the first model follows, where we omit the definitions of certain tracker and generic agent types for brevity.

```
****************************************
* PCCS Verification Run **************
* # 0
* Generated from: file:samples/LiesbetDeadTestDead.liesbet
* On: Fri Jul 14 12:28:01 BST 2006

...appropriate tracker and generic type agents...

proc Workflow0 =
   (
***Instance:0:P1
   InitialState2[runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, ninit_0/ninit, comp_0/comp, canc_0/canc, exec_0/exec,
      prec_0/prec, cald_0/pcald] |

   Par2[runn_0/runn, cald_0/cald, cotd_0/cotd,
      find_0/find, ninit_0/ninit, comp_0/comp, canc_0/canc, exec_0/exec,
      exec_1/exec1, exec_7/exec2] |

***Instance:1:S1
   InitialState3[runn_1/runn, cald_1/cald, cotd_1/cotd,
      find_1/find, ninit_1/ninit, comp_1/comp, canc_1/canc, exec_1/exec,
      prec_1/prec, prec_0/pprec, cald_0/pcald] |
```

```
   Seq3[runn_1/runn, cald_1/cald, cotd_1/cotd,
      find_1/find, ninit_1/ninit, comp_1/comp, canc_1/canc, exec_1/exec,
      exec_2/exec3, find_2/find3, exec_3/exec2, find_3/find2, exec_6/exec1] |

***Instance:2:A
   InitialState0[runn_2/runn, cald_2/cald, cotd_2/cotd,
      find_2/find, ninit_2/ninit, comp_2/comp, canc_2/canc, exec_2/exec,
      prec_1/pprec, cald_1/pcald] |

***Instance:3:JOIN_SEC_B
   InitialState2[runn_3/runn, cald_3/cald, cotd_3/cotd,
      find_3/find, ninit_3/ninit, comp_3/comp, canc_3/canc, exec_3/exec,
      prec_3/prec, prec_1/pprec, cald_1/pcald] |

   Seq2[runn_3/runn, cald_3/cald, cotd_3/cotd,
      find_3/find, ninit_3/ninit, comp_3/comp, canc_3/canc, exec_3/exec,
      exec_4/exec2, find_4/find2, exec_5/exec1] |

***Instance:4:BJoin
   InitialStateSC0[runn_4/runn, cald_4/cald, cotd_4/cotd,
      find_4/find, ninit_4/ninit, comp_4/comp, canc_4/canc, exec_4/exec,
      prec_3/pprec, cald_3/pcald, canc_3/pcanc] |
 (
***GoQuery
  ('cotd_11:5.'done0:4.nil | done0:4.'comp_4:10.nil)\{done0}
  [> 'find_4:5.nil
 ) |

***Instance:5:B
   InitialStateSC0[runn_5/runn, cald_5/cald, cotd_5/cotd,
      find_5/find, ninit_5/ninit, comp_5/comp, canc_5/canc, exec_5/exec,
      prec_3/pprec, cald_3/pcald,  canc_3/pcanc] |

***Instance:6:C
   InitialState0[runn_6/runn, cald_6/cald, cotd_6/cotd,
      find_6/find, ninit_6/ninit, comp_6/comp, canc_6/canc, exec_6/exec,
      prec_1/pprec, cald_1/pcald] |

***Instance:7:S2
   InitialState3[runn_7/runn, cald_7/cald, cotd_7/cotd,
      find_7/find, ninit_7/ninit, comp_7/comp, canc_7/canc, exec_7/exec,
      prec_7/prec, prec_0/pprec, cald_0/pcald] |

   Seq3[runn_7/runn, cald_7/cald, cotd_7/cotd,
      find_7/find, ninit_7/ninit, comp_7/comp, canc_7/canc, exec_7/exec,
      exec_8/exec3, find_8/find3, exec_9/exec2, find_9/find2, exec_12/exec1] |
```

```
***Instance:8:D
   InitialState0[runn_8/runn, cald_8/cald, cotd_8/cotd,
      find_8/find, ninit_8/ninit, comp_8/comp, canc_8/canc, exec_8/exec,
      prec_7/pprec, cald_7/pcald] |

***Instance:9:JOIN_SEC_E
   InitialState2[runn_9/runn, cald_9/cald, cotd_9/cotd,
      find_9/find, ninit_9/ninit, comp_9/comp, canc_9/canc, exec_9/exec,
      prec_9/prec, prec_7/pprec, cald_7/pcald] |
   Seq[runn_9/runn, cald_9/cald, cotd_9/cotd,
      find_9/find, ninit_9/ninit, comp_9/comp, canc_9/canc, exec_9/exec,
      exec_10/exec2, find_10/find2, exec_11/exec1] |

***Instance:10:EJoin
   InitialState0[runn_10/runn, cald_10/cald, cotd_10/cotd,
      find_10/find, ninit_10/ninit, comp_10/comp, canc_10/canc, exec_10/exec,
      prec_9/pprec, cald_9/pcald, canc_9/pcanc] |
 (
***GoQuery
  ('cotd_5:5.'done1:4.nil | done1:4.'comp_10:10.nil)\{done1}
  [> 'find_10:5.nil
 ) |

***Instance:11:E
   InitialState0[runn_11/runn, cald_11/cald, cotd_11/cotd,
      find_11/find, ninit_11/ninit, comp_11/comp, canc_11/canc, exec_11/exec,
      prec_9/pprec, cald_9/pcald, canc_9/pcanc] |

***Instance:12:F
   InitialState0[runn_12/runn, cald_12/cald, cotd_12/cotd,
      find_12/find, ninit_12/ninit, comp_12/comp, canc_12/canc, exec_12/exec,
      prec_7/pprec, cald_7/pcald] |

   Basics6[comp_2/comp1, comp_5/comp2, comp_6/comp3, comp_8/comp4, comp_11/comp5, comp_12/comp6] |

   'exec_0:3.pprec:5.nil | 'find_0:10.'rfind:10.nil

   )\{
   runn_0, cald_0, cotd_0, find_0, ninit_0, comp_0, canc_0, exec_0, prec_0,
   runn_1, cald_1, cotd_1, find_1, ninit_1, comp_1, canc_1, exec_1, prec_1,
   runn_2, cald_2, cotd_2, find_2, ninit_2, comp_2, canc_2, exec_2, prec_2,
   runn_3, cald_3, cotd_3, find_3, ninit_3, comp_3, canc_3, exec_3, prec_3,
   runn_4, cald_4, cotd_4, find_4, ninit_4, comp_4, canc_4, exec_4, prec_4,
   runn_5, cald_5, cotd_5, find_5, ninit_5, comp_5, canc_5, exec_5, prec_5,
   runn_6, cald_6, cotd_6, find_6, ninit_6, comp_6, canc_6, exec_6, prec_6,
```

```
runn_7, cald_7, cotd_7, find_7, ninit_7, comp_7, canc_7, exec_7, prec_7,
runn_8, cald_8, cotd_8, find_8, ninit_8, comp_8, canc_8, exec_8, prec_8,
runn_9, cald_9, cotd_9, find_9, ninit_9, comp_9, canc_9, exec_9, prec_9,
runn_10, cald_10, cotd_10, find_10, ninit_10, comp_10, canc_10, exec_10, prec_10,
runn_11, cald_11, cotd_11, find_11, ninit_11, comp_11, canc_11, exec_11, prec_11,
runn_12, cald_12, cotd_12, find_12, ninit_12, comp_12, canc_12, exec_12, prec_12,
pprec, pcald}
```

Under CWB-NC, the proposition `find` (see Section 5.6.1) is found to be FALSE, as appropriate.

```
cwb-nc> load test.pccs
Execution time (user,system,gc,real):(0.047,0.000,0.000,0.047)
cwb-nc> load testp.mu
Execution time (user,system,gc,real):(0.015,0.000,0.000,0.015)
cwb-nc> chk Workflow0 find
Invoking alternation-free model checker.
Building automaton...
States: 35
Transitions: 36
Done building automaton.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(6.844,0.000,0.062,6.844)
cwb-nc>
```

For the other workflow model, there should be no deadlock – we have removed one of the join conditions responsible for the cyclic dependency. Its PCCS source is the same as that above with activities 9 (`SeqCancel`) and 10 (`EJoin`) removed, and activity 11 (`E`) promoted to being a direct child of activity 7 (`S2`).

The CWB-NC output when testing `find` on this model is as follows, correctly indicating an absence of deadlock.

```
cwb-nc> load test.pccs
Execution time (user,system,gc,real):(0.031,0.000,0.000,0.031)
cwb-nc> chk Workflow0 find
Invoking alternation-free model checker.
Building automaton...
States: 99
Transitions: 106
Done building automaton.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(11.656,0.000,0.403,11.656)
cwb-nc>
```

A.5 Support for Non-monotonic and Distinct Reference Queries

In order to support non-monotonic and distinct reference queries, we need to change the characteristic of our PCCS-based characterisation that instances of synchronisation types are evaluated

between instances of other types grabbing execution rights. Instead, we need to move to a characterisation where synchronisation instances compete for these rights; and, once such an instance has them, its respective `GoQuery` and/or `StopQuery` is fully evaluated to determine satisfiability, prior to releasing them. This will ensure a sound characterisation for non-monotonic and distinct queries. We also may support negated queries, and queries directly on `Initial` and `Running` states – as opposed to supporting queries on `NotInitial`. We do not provide further details here for non-monotonic querying – although, it should be quite evident how this would be achieved.

With regard to distinct reference queries, it is important that they be evaluated atomically, so that if all component sub-queries of a `GoQuery`, or `StopQuery`, for a type may be satisfied at some time point, it will not be the case that another querying instance is able to "steal" the use of the candidate instances. This is not relevant if we are not using distinct queries, because candidate instances may be used without limit in satisfying such queries.

In the following, we give just a flavour of the PCCS-based support for distinct querying. We present a more detailed overview of our support for distinct querying in Appendix Section B.1.2, which describes our `SitCalc`-based support for it.

Distinct queries are satisfied against target instances, just as non-distinct queries are satisfied. An instance `t` may be used to satisfy a distinct query just once per instance `di`, pertaining to the distinct reference type of a query. (See Section 3.1.3 for more information regarding distinct reference types.) For every *pair* `(t,di)` *which could be used in the satisfaction of a specific query*, we make use of an agent `ProxyInit` (for satisfying queries pertaining to the `Initial` state), which may evolve into agents `ProxyRunn`, `ProxyCotd`, or `ProxyCald` as the pertaining tracker agent for instance `t` evolves.

Each of these agents will make use of the following channels:

- Incoming:
  - Proxy channels for querying each of the states: `Completed`, `Cancelled`, `Finished`, `Running` and `Initial` – specifically, `pcotd:5`, `pcald:5`, `pfind:5`, `prunn:5` and `pinit:5`.
  - A channel for marking the proxy agent as expended: `exp:3`
- Outgoing:
  - Querying channels to be connected to `t`'s tracker agent's state-querying channels – specifically, `cotd:5`, `cald:5`, `find:5`, `runn:5` and `init:5`.

The `ProxyInit` agent would have the following PCCS definition:

```
proc ProxyInit =
   'cald:3.ProxyCald + 'runn:3.ProxyRunn + pinit:5.ProxyInit + exp:3.nil
```

While the proxy agent has not been used to satisfy a query for its pertaining `(t,di)` pair, the agent:

- May evolve into `ProxyRunn`, `ProxyCotd` or `ProxyCald`, as appropriate, in response to changes in the state of the pertaining tracker agent for the instance.
- Allows querying instances to ascertain that the target instance is in the `Initial` state, using `pinit`, and facilitates the marking of the agent as expended, meaning that it can no longer be used to satisfy queries against the instance pair.

A similar definition is appropriate for ProxyRunn, viz.

```
proc ProxyRunn =
   'cald:3.ProxyCald + 'cotd:3.ProxyCotd + prunn:5.ProxyRunn + exp:3.nil
```

For ProxyCotd and ProxyCald, we also allow 'finished' queries, as shown.

```
proc ProxyCald =
   pcald:5.ProxyCald + pfind:5.ProxyCald + exp:3.nil

proc ProxyCotd =
   pcotd:5.ProxyCotd + pfind:5.ProxyCotd + exp:3.nil
```

When we make use of a distinct query within a GoQuery, or a StopQuery, we use the proxy channels in place of cotdt etc., as presented in the definition of $\mathcal{QT}[\![-]\!]$ in Appendix Section A.3. The proxy agents and associated channels are constructed as part of the translation process as needed. Then, if such a query is satisfied, there will be a residual piece of logic for the query which marks (by 'exp:3) the specific target instance pairs, used in satisfying the query, as expended.

A.6 CancelActivity and Exit

For Liesbet2, we translate synchronisation activity types by outputting PCCS agents which have been customised for the visibility horizons of the pertaining instances (see Appendix Section A.3). For CancelActivity, we adopt a similar approach.

The translation of CancelActivity types is defined by the following extension to $\mathcal{M}_{pccs}[\![-]\!]$.

$\mathcal{M}_{pccs}[\![\texttt{CancelActivity}(\phi)]\!](st\_chs_i \rightarrow, \texttt{pprec}_i, \texttt{pcald}_i) =$
$\quad \texttt{InitialState}^0\,[SC_i,\ {}^{\texttt{pprec}_i}/_{\texttt{pprec}},\ {}^{\texttt{pcald}_i}/_{\texttt{pcald}}]$
|
$\quad CT(CT[\![\phi]\!](st\_chs_i \rightarrow))\,[SC_i]$

The auxiliary function $CT[\![-]\!]$ translates (as part of *Step 3*) the cancellation reference, which, syntactically, will be of the form qtype, or qtype IN rtype, into an agent which effects cancel on all instances within the visibility horizon of the cancellation instance. Similarly to $\mathcal{QT}[\![-]\!]$, which is used in the translation of queries for synchronisation types, $CT[\![-]\!]$ relies on the existence of a relation, namely, CancInScope, which is built in *Step 1* of the translation process. The definition of CancInScope follows from that of CotdInScope, presented in Appendix Section A.3. The definition of $CT[\![-]\!]$ is, then, as follows.

$CT[\![\phi]\!](st\_chs_i \rightarrow) =$
$(\prod_{c \in C}$ 'c:3).'comp:3.nil
where, for ϕ being qtype,
C = { canct | ∃rtype.CancInScope(canci, canct, rtype, qtype) }
and for ϕ being qtype IN rtype
C = { canct | CancInScope(canci, canct, rtype, qtype) }

Then, the definition of CT, which is responsible for constructing the definition of the customised agent pertaining to the translated CancelActivity type is as follows. We pass in the output from $CT[\![-]\!]$.

CT(tCanc_Ref) = 'runn:10.tCanc_Ref + 'cald:5.nil

The translation of Exit is defined by the following extension to $\mathcal{M}_{pccs}[\![-]\!]$, where we assume that the cancellation channel of the root instance, rcanc, is set aside by $\mathcal{M}_{pccs}[\![-]\!]$ for use in translating Exit types.

$\mathcal{M}_{pccs}[\![\text{Exit}]\!](st\_chs_i \rightarrow, \text{pprec}_i, \text{pcald}_i) =$
$\text{InitialState}^0[SC_i, {}^{\text{pprec}_i}/_{\text{pprec}}, {}^{\text{pcald}_i}/_{\text{pcald}}]$
|
CT('rcanc:3.nil)$[SC_i]$

A.7 MultiLimit^n and MultiLimitSeq^n

The MultiLimit^n and MultiLimitSeq^n multiple-instance activity types are represented in our PCCS characterisation, in a straightforward way. In section 3.5, we note that these types represent a possibility for satisfying the representational requirements epitomised by the YAWL workflow patterns, relating to multiple-instance activity types.

We have presented characterisations for Multi and MultiSeq in Section 5.5. It is worth noting that, for verification, it is better efficiency-wise to use the limited-instance (MultiLimit/MultiLimitSeq) types, rather than the Multi/MultiSeq types. This is because the auxiliary counter that is used in the characterisation of an unlimited-instance type, to keep a track of the number of outstanding child instances, is quite costly from the perspective of the size of the verification state-space.

The translation of MultiLimit^n and MultiLimitSeq^n is defined by the following extensions to $\mathcal{M}_{pccs}[\![-]\!]$.

$\mathcal{M}_{pccs}[\![\text{MultiLimit(n)(ExecAct(join(ExecActJoin)))}]\!](st\_chs_i \rightarrow, \text{pprec}_i, \text{pcald}_i) =$
$\text{let } st\_chs_{ij1} \rightarrow \text{ in } \ldots st\_chs_{ijn} \rightarrow \text{ in let } st\_chs_{ie1} \rightarrow \text{ in } \ldots st\_chs_{ien} \rightarrow \text{ in let prec}_i \text{ in}$
$\text{MultiLimit}^n[SC_i, SC_{ij1,j1}, \ldots, SC_{ijn,jn}, SC_{ie1,e1}, \ldots, SC_{ien,en}]$
|
$\text{InitialState}^{2n}[SC_i, {}^{\text{pprec}_i}/_{\text{pprec}}, {}^{\text{pcald}_i}/_{\text{pcald}}, {}^{\text{prec}_i}/_{\text{prec}}]$
|
$\mathcal{M}_{pccs}[\![\text{ExecActJoin}]\!](st\_chs_{ij1} \rightarrow, \text{prec}_i, \text{cald}_i) \mid \ldots \mid$
$\mathcal{M}_{pccs}[\![\text{ExecActJoin}]\!](st\_chs_{ijn} \rightarrow, \text{prec}_i, \text{cald}_i)$
|
$\mathcal{M}_{pccs}[\![\text{ExecAct}]\!](st\_chs_{ie1} \rightarrow, \text{prec}_i, \text{cald}_i) \mid \ldots \mid \mathcal{M}_{pccs}[\![\text{ExecAct}]\!](st\_chs_{ien} \rightarrow, \text{prec}_i, \text{cald}_i)$

$\mathcal{M}_{pccs}[\![\text{MultiLimitSeq(n)(ExecAct(join(ExecActJoin)))}]\!](st\_chs_i \rightarrow, \text{pprec}_i, \text{pcald}_i) =$
$\text{let } st\_chs_{ij1} \rightarrow \text{ in } \ldots st\_chs_{ijn} \rightarrow \text{ in let } st\_chs_{ie1} \rightarrow \text{ in } \ldots st\_chs_{ien} \rightarrow \text{ in let prec}_i \text{ in}$
$\text{MultiLimitSeq}^n[SC_i, SC_{ij1,j1}, \ldots, SC_{ijn,jn}, SC_{ie1,e1}, \ldots, SC_{ien,en}]$
|
$\text{InitialState}^{2n}[SC_i, {}^{\text{pprec}_i}/_{\text{pprec}}, {}^{\text{pcald}_i}/_{\text{pcald}}, {}^{\text{prec}_i}/_{\text{prec}}]$
|
$\mathcal{M}_{pccs}[\![\text{ExecActJoin}]\!](st\_chs_{ij1} \rightarrow, \text{prec}_i, \text{cald}_i) \mid \ldots \mid$
$\mathcal{M}_{pccs}[\![\text{ExecActJoin}]\!](st\_chs_{ijn} \rightarrow, \text{prec}_i, \text{cald}_i)$
|

$$\mathcal{M}_{pccs}[\![\texttt{ExecAct}]\!](st\_chs_{\texttt{ie1}} \rightarrow, \texttt{prec}_i, \texttt{cald}_i) \mid \ldots \mid \mathcal{M}_{pccs}[\![\texttt{ExecAct}]\!](st\_chs_{\texttt{ien}} \rightarrow, \texttt{prec}_i, \texttt{cald}_i)$$

The definition of the PCCS agents for MultiLimit^n and MultiLimitSeq^n, for the case where n is 3, are now presented.

```
proc MultiLimit3 =
    'runn:10.'execj3:3.MultiLimit3f + 'cald:5.nil

proc MultiLimit3f =
    'cotdj3:10.'execе3:3.'execj2:3.MultiLimit2f +
    'caldj3:10.'cance3:3.'cancj2:3.'cance2:3.'cancj1:3.'cance1:3.nil +
    'cald:5.nil

proc MultiLimit2f =
    'cotdj2:10.'execе2:3.'execj1:3.MultiLimit1f +
    'caldj2:10.'cance2:3.'cancj1:3.'cance1:3.nil +
    'cald:5.nil

proc MultiLimit1f =
    'cotdj1:10.'execе1:3.nil + 'caldj1:10.'cance1:3.nil + 'cald:5.nil

proc MultiLimitSeq3 =
    'runn:10.'execj3:3.MultiLimitSeq3fj + 'cald:5.nil

proc MultiLimitSeq3fj =
    'cotdj3:10.'execе3:3.MultiLimitSeq3fe +
    'caldj3:10.'cance3:3.'cancj2:3.'cance2:3.'cancj1:3.'cance1:3.nil +
    'cald:5.nil

proc MultiLimitSeq3fe =
    'finde3:10.'execj2:3.MultiLimitSeq2fj + 'cald:5.nil

proc MultiLimitSeq2fj =
    'cotdj2:10.'execе2:3.MultiLimitSeq2fe +
    'caldj2:10.'cance2:3.'cancj1:3.'cance1:3.nil +
    'cald:5.nil

proc MultiLimitSeq2fe =
    'finde2:10.'execj1:3.MultiLimit1f + 'cald:5.nil
```

In the presented PCCS characterisation of MultiLimit and MultiLimitSeq, we create n instances of the execution activity, ExecAct, and its associated join condition. Similarly to the definition of Seq^n agents, the first (join condition, execution activity) pair to be executed are those with the highest index. This makes for more simple definitions of the MultiLimit^n and MultiLimitSeq^n agents.

We start by executing the first join condition instance. If it completes successfully then this triggers the execution of its corresponding execution activity instance. If it gets cancelled, however, all remaining execution and join condition instances get cancelled. For MultiLimit^n, as soon as

an execution activity instance has been set running, we initiate the execution of the next join condition instance. This continues until we run out of instances. For $\texttt{MultiLimitSeq}^n$ types, we need to wait for the execution activity instance that we have just set running to finish before we initiate the execution of the next join condition instance.

A.8 $\texttt{MultiMerge}^{m,n}$

The translation of $\texttt{MultiMerge}^{m,n}$ is defined by the following extension to $\mathcal{M}_{pccs}[\![-]\!]$.

$\mathcal{M}_{pccs}[\![\texttt{MultiMerge(Chg1,...,Chgn,Chc1,...,Chcm)}]\!](st\_chs_i \rightarrow, \texttt{pprec}_i, \texttt{pcald}_i) =$
$\quad \texttt{let } st\_chs_{\texttt{ig1}} \rightarrow \texttt{ in } \ldots\ st\_chs_{\texttt{ign}} \rightarrow \texttt{ in let } st\_chs_{\texttt{ic1}} \rightarrow \texttt{ in } \ldots\ st\_chs_{\texttt{icm}} \rightarrow \texttt{ in let prec}_i \texttt{ in}$
$\quad\ \texttt{MultiMerge}^{m,n}[\mathcal{SC}_i,\ \mathcal{SC}_{\texttt{ig1,g1}},\ \ldots,\ \mathcal{SC}_{\texttt{ign,gn}},\ \mathcal{SC}_{\texttt{ic1,c1}},\ \ldots,\ \mathcal{SC}_{\texttt{icm,cm}}]$
|
$\quad \texttt{InitialState}^{m+n}[\mathcal{SC}_i,\ {}^{\texttt{pprec}_i}/_{\texttt{pprec}},\ {}^{\texttt{pcald}_i}/_{\texttt{pcald}},\ {}^{\texttt{prec}_i}/_{\texttt{prec}}]$
|
$\quad \mathcal{M}_{pccs}[\![\texttt{Chg1}]\!](st\_chs_{\texttt{ig1}} \rightarrow, \texttt{prec}_i, \texttt{cald}_i) \mid \ldots \mid \mathcal{M}_{pccs}[\![\texttt{Chgn}]\!](st\_chs_{\texttt{ign}} \rightarrow, \texttt{prec}_i, \texttt{cald}_i)$
|
$\quad \mathcal{M}_{pccs}[\![\texttt{Chc1}]\!](st\_chs_{\texttt{ic1}} \rightarrow, \texttt{prec}_i, \texttt{cald}_i) \mid \ldots \mid \mathcal{M}_{pccs}[\![\texttt{Chcm}]\!](st\_chs_{\texttt{icm}} \rightarrow, \texttt{prec}_i, \texttt{cald}_i)$

The definition of the PCCS agent for $\texttt{MultiMerge}^{m,n}$, for the case where n is 4 and m is 2, is now presented. n is the number of guard instances of the Multimerge type, and m is the number of continuation instances.

```
proc MultiMerge2_4
=
   'runn:10.'execg1:3.'execg2:3.'execg3:3.'execg4:3.MultiMerge2_4f + 'cald:5.nil

proc MultiMerge2_4f
=
   (
      'cotdg1:10.'go:3.'used:3.nil + 'caldg1:10.'used:3.nil
    |
      'cotdg2:10.'go:3.'used:3.nil + 'caldg2:10.'used:3.nil
    |
      'cotdg3:10.'go:3.'used:3.nil + 'caldg3:10.'used:3.nil
    |
      'cotdg4:10.'go:3.'used:3.nil + 'caldg4:10.'used:3.nil
    |
      go:3.'execc1:3.(go:3.'execc2:3.(go:3.(go:3.stop:3.nil + stop:3.nil) + stop:3.nil) +
                    stop:3.'cancc2:3.nil) +
      stop:3.'cancc1:3.'cancc2:3.nil
    |
      used:3.used:3.used:3.used:3.'stop:3.nil
   )\{go,stop,used}
```

In $\texttt{MultiMerge}^{m,n}$, all of the guard instances are set running. Then, the first guard instance to complete successfully triggers the execution of the first continuation instance. This is facilitated

in MultiMerge$^{m\text{-}n}$f by signalling on go. The second guard to complete successfully triggers the execution of the second continuation instance, and so on. Notably, once m guard instances have completed successfully, and the execution of m continuation instances has been initiated, there will not be any more continuation instances to execute, notwithstanding the fact that more guard instances may complete successfully. Whenever, guard instances get completed *or* cancelled, a synchronisation is made on used. This occurs so that when all guard instances have finished, we may cancel the unused continuation instances (by signalling on stop).

A.9 Discriminator$^{m\text{-}n}$

The translation of Discriminator$^{m\text{-}n}$ is defined by the following extension to $\mathcal{M}_{pccs}[\![-]\!]$.

$\mathcal{M}_{pccs}[\![$Discriminator(m)(Chg1,...,Chgn,Chc)$]\!](st_chs_i \rightarrow$, pprec$_i$, pcald$_i$)=
let $st\_chs_{ig1} \rightarrow$ in ... $st\_chs_{ign} \rightarrow$ in let $st\_chs_{ic} \rightarrow$ in let prec$_i$ in
 Discriminator$^{m\text{-}n}$[$\mathcal{SC}_i$, $\mathcal{SC}_{ig1,g1}$, ..., $\mathcal{SC}_{ign,gn}$, $\mathcal{SC}_{ic,c}$]
|
 InitialState^{n+1}[$\mathcal{SC}_i$, $^{\text{pprec}_i}/_{\text{pprec}}$, $^{\text{pcald}_i}/_{\text{pcald}}$, $^{\text{prec}_i}/_{\text{prec}}$]
|
 $\mathcal{M}_{pccs}[\![$Chg1$]\!](st_chs_{ig1} \rightarrow$,prec$_i$, cald$_i$) | ... | $\mathcal{M}_{pccs}[\![$Chgn$]\!](st_chs_{ign} \rightarrow$,prec$_i$, cald$_i$) |
  $\mathcal{M}_{pccs}[\![$Chc$]\!](st_chs_{ic} \rightarrow$,prec$_i$, cald$_i$)

The definition of the PCCS agent for Discriminator$^{m\text{-}n}$, for the case where n is 4 and m is 2, is now presented. n is the number of guard instances of the Discriminator type, and m is its completion threshold, for executing the continuation instance.

```
proc Discriminator2_4
=
   'runn:10.'execg1:3.'execg2:3.'execg3:3.'execg4:3.Discriminator2_4f + 'cald:5.nil

proc Discriminator2_4f
=
   (
      'cotdg1:10.'win:3.nil + 'caldg1:10.'lose:3.nil
    |
      'cotdg2:10.'win:3.nil + 'caldg2:10.'lose:3.nil
    |
      'cotdg3:10.'win:3.nil + 'caldg3:10.'lose:3.nil
    |
      'cotdg4:10.'win:3.nil + 'caldg4:10.'lose:3.nil
   |
     lose:3.lose:3.lose:3.'canc:3.nil
   |
     win:3.win:3.execc:3.nil

 )\{win,lose}

[> 'find:5.nil
```

In `Discriminator`^{m,n}, all of the guard instances are set running. Whenever one of them completes (resp. gets cancelled), a synchronisation on `win` (resp. `lose`) occurs. If sufficient synchronisations on `win` occur (i.e. the completion threshold is met), the continuation instance is executed. If sufficient synchronisations on `lose` occur (i.e. the failure threshold is met), the `Discriminator` instance, as a whole, is cancelled. The failure threshold corresponds to the number of guard instances which must fail (i.e. get cancelled) in order that the completion threshold can never be reached. Its value is $(n - m) + 1$. Once the `Discriminator` instance has finished, the residual logic of the generic type agent is garbage-collected.

Appendix B

SitCalc Characterisation – Additional Information

In this appendix, we complete the presentation of the SitCalc characterisation for Liesbet, and the presentation of the translation function $\mathcal{M}_{SitCalc}[\![-]\!]$.

B.1 Remaining SitCalc Characterisation of Liesbet

In this section, we present the SitCalc-based characterisation of the Liesbet types omitted from the presentation in Chapter Six.

B.1.1 Completion and Cancellation Actions on Childless Structured Instances

Childless structured instances may be explicitly completed (or cancelled). (Notably, child-bearing structured instances are completed/cancelled implicitly as a side-effect of some action occurrence on another instance. For instance, a child-bearing instance may be completed through propagation as a side-effect of a descendant instance finishing.) The childless types in question are: FreeChoice, Empty, Go, Stop, CancelActivity and Exit.

There are four action schemas that are concerned with the completion and cancellation of childless structured instances. We concentrate on the most general two for the time being – the other two are concerned with something very specific, namely, the completion or cancellation of Go or Stop synchronisation types which make use of distinct querying (see Appendix Section B.1.2). The two general action schemas are complete/1 and cancel/1. The action precondition axioms for these actions are now presented. Note that the CType/2 (resp. GType/2) fluent records the customised (resp. generic) type of an instance i in situation s.

```
Poss(complete(i),s) ≡ State(i,s)=Running ∧ (GType(i,s)=GId_FRE ∨ GType(i,s)=GId_EMP ∨
  GType(i,s)=GId_CAN ∨ GType(i,s)=GId_CAR ∨ GType(i,s)=GId_EXI ∨
  (CType(i,s) = CUSTOMISED_SYNC_TYPE ∧ CUSTOMISED_COMPLETION_CONDITION) ∨ ...) ∧
  ¬(∃p,i',c,g,sc,f,j).Poss(add_activity(p,i',c,g,sc,f,j),s)

Poss(cancel(i),s) ≡ State(i,s)=Running ∧ (GType(i,s)=GId_FRE ∨
```

```
(CType(i,s) = CUSTOMISED_SYNC_TYPE ∧ CUSTOMISED_CANCELLATION_CONDITION) ∨ ...) ∧
¬(∃p,i',c,g,sc,f,j).Poss(add_activity(p,i',c,g,sc,f,j),s)
```

The first of these axioms (for `complete/1`) says that it is possible to complete a `FreeChoice` (`GId_FRE`), `Empty` (`GId_EMP`), `CancelActivity` (`GId_CAN` or `GId_CAR`), or customised synchronisation (`Go` or `Stop`) instance iff the instance is running, and it is not possible to add another instance (via `add_activity/7`) to the CWS. Further information regarding the facilitation of `CancelActivity` in the `SitCalc` semantics for `Liesbet` is presented in Appendix Section B.1.5.

Note that these axioms are, for the most part, domain-independent, but they may be customised for a particular model with respect to the use of synchronisation types. For any occurrence of a customised synchronisation type, in the precondition axiom for `complete/1`, its corresponding completion condition (which must also hold for the action to be possible) will be the `GoQuery` of the pertaining synchronisation instance. We present an example of this at the end of this section.

The second of these axioms is similar to the first, except that `Empty`, `CancelActivity`, and `Exit` instances may not be (explicitly, at least) cancelled. These possibilities are thus removed from the axiom for `cancel/1`. Note that occurrences of a customised cancellation condition in an instance of the `cancel/1` action precondition axiom correspond to the `StopQuery` of the pertaining synchronisation instance. Clearly, if a synchronisation type only has one type of query then it will only appear in one of the `complete,cancel/1` axioms (i.e. `complete/1` for `GoQuery` only, and `cancel/1` for `StopQuery` only).

For `complete/1` actions, we need to modify the definition of `Completing/3`, and `CompletingAction/2`, viz.

```
Completing(i,a,st) ≡ (a=comp_bas(i) ∨ a=complete(i)) ∧ st=Completed

CompletingAction(i,a) ≡ a=comp_bas(i) ∨ complete(i)
```

For `cancel/1` actions, we need to modify the definition of `CancellingAction/2`, viz.

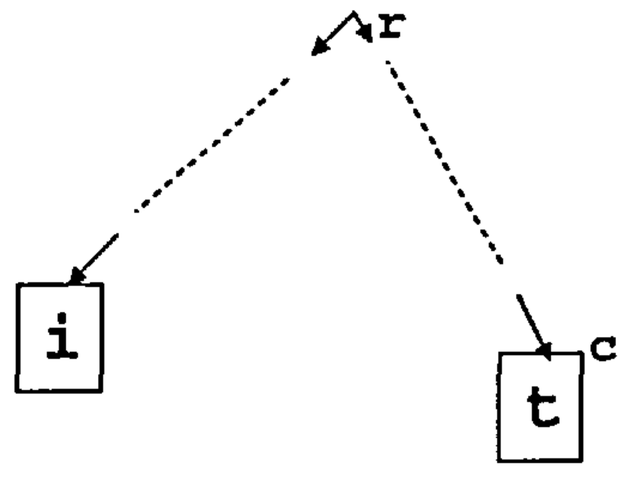
```
CancellingAction(i,a) ≡ a=canc_bas(i) ∨ cancel(i)
```

We also modify the action precondition axioms for `comp_bas,canc_bas/1` to say that these actions are only possible if a `complete,cancel/1` on an instance is not possible. This is a straightforward extension.

Finally, the customised completion/cancellation conditions, in these precondition axioms, will make use of (instances of) the `InScope/5` predicate. This predicate determines the visibility horizon for instances, and has the following definition.

```
InScope(i,t,r,c,do(a,s)) ≡
  (∃p,g,sc,f,j).sc=NONE ∧ (((∃c').a=add_activity(p,i,c',g,sc,f,j) ∧
  (InScope(p,t,r,c,s) ∨ (CType(p,s)=r ∧ CType(t,s)=c ∧ Descendant(p,t,s)))) ∨
  (a=add_activity(p,t,c,g,sc,f,j) ∧ (∃c').InScope(i,p,r,c',s))) ∨
  InScope(i,t,r,c,s)
```

Referring to Figure B.1, the successor state axiom for `InScope/5` says that a target instance



An instance i has target t (of customised type c) in its visibility horizon iff there is a reference instance (of type r) which is ancestral to both instances, and there is no intervening isolated scope (see Section 3.1.3). t may not be an ancestor of i.

Figure B.1: InScope/5, Defining Visibility Horizons.

t is in the visibility horizon of an instance i, with reference type r, and customised type c (in situation do(a,s)) iff

- the instance i is being added to the CWS (via action a) and (i) i is not isolated (sc=NONE), and (ii) its parent p has t (with respect to r and c) in its visibility horizon OR its parent p is itself of type r and t (of customised type c) is a descendant of p

  OR

- the target t is being added to the CWS (via a) and (i) t is not isolated, and (ii) t's parent is in the visibility horizon of i

  OR

- InScope for i, with t/r/c, holds in the previous situation. (Once an instance of this fluent is asserted to the BAT, it persists thereafter.)

As an example of the dispensation made for synchronisation types within the precondition axioms for complete,cancel/1, say we have a Go instance with GoQuery Completed_act(q), and customised type name CId_G. Then, according to the definition of the translator (for Liesbet models) presented below in Appendix Section B.2, the pertaining fragment of the precondition axiom for complete/1 would look as follows, albeit presented in an abridged form here.

CType(i,s) = CId_G ∧ (∃t,r).InScope(i,t,r,q,s) ∧ State(t,s)=Completed It is worth noting that, in Section 6.2.2, we made the assertion that most of the SitCalc-based characterisation of a Liesbet model instance may be considered as foundational axioms, as they are domain-independent in nature. It is trivial to specify versions of the axioms presented here for cancel, complete/1 which are also domain-independent. They may instead refer, in a domain-independent way, to auxiliary fluents, whose instances would be used to represent the domain-dependent information.

B.1.2 Distinct Querying

Distinct queries are supported by means of customisations to action precondition axioms for two further action schemas, viz. `complete,cancel/3`. Actions pertaining to these schemas take two additional arguments compared with actions for synchronisation types which do not make use of distinct querying.

To recap from Section 3.1.4, a `GoQuery` or `StopQuery`, within a synchronisation type, may be a composite query, meaning that it may contain a number of sub-queries which are composed into boolean expressions, where some of the sub-queries may be queries involving distinct reference types, i.e. *distinct queries*. There are some restrictions on the use of distinct queries which make their semantic characterisation much simpler:

- They are not allowed to be under the scope of a negation at any level of nesting.
- In any one `GoQuery`, or `StopQuery`, the same distinct reference type should be used, which will necessarily resolve to the same instance.
- Disjunction exists only at the outer-most query level of a `GoQuery` or `StopQuery`.
- The target instances that may be used to satisfy each distinct query within a conjunct (of the top-level disjunction – see previous point) must fall into disjoint sets.

In satisfying a composite query involving distinct queries, we must mark as expended the target instances used to satisfy the query against the common instance of the distinct reference type. The two additional arguments for the action schemas, `complete,cancel/3`, are: `di`, which is the instance of the distinct reference type used to satisfy the `Go/StopQuery`, and `l` which is a list of targets to mark as expended against `di`.

As disjunction exists at the outer querying level only, we can construct the customisation of the pertinent action precondition axiom (for `GoQuerys`, this will be the axiom for `complete/3`, and for `StopQuerys`, `cancel/3`), as a disjunction where we assign the target instances to mark as expended in each of the conjuncts, by assigning the action argument `l`.

For example, we may have the following query, used to complete a `Go` type, `CId_G`.

```
Finished_act(CId_A dist in CId_P) | Finished_act(CId_B dist in CId_P) |
  Finished_act(CId_C in CId_P) +
   Finished_act(CId_D dist in CId_P) | Finished_act(CId_C in CId_P)
```

Here, the query is satisfied either by satisfying distinct queries on `A` and `B` and a non-distinct query on `C` or by satisfying a distinct query on `D` and a non-distinct query on `C`. Note that the disjunction appears at the outer-most level.

The action precondition for `complete(i,di,l)` is customised to include the case of completing `CId_G` using this query. An abridged version follows.

```
Poss(complete(i,di,l),s) ≡
  ...
  CType(i,s)=CId_G ∧
  (∃ti1,ti2,ti3).DistInScope(i,di,CId_P,s) ∧
   (l=[ti1,ti2] ∧ ¬DistQuery(ti1,di,s) ∧ ¬DistQuery(ti2,di,s) ∧
   InScope(i,ti1,CId_P,CId_A,s) ∧ InScope(i,ti2,CId_P,CId_B,s) ∧
```

```
    (∃ti).InScope(i,ti,CId_P,CId_C,s) ∨
    l=[ti3] ∧ ¬DistQuery(ti3,di,s) ∧ InScope(i,ti3,CId_P,CId_D,s) ∧
    (∃ti).InScope(i,ti,CId_P,CId_C,s))
  ...
```

One or more instances of the fluent `DistInScope(i,di,d,s)` are asserted to the BAT whenever an activity instance is added to the CWS. The fluent asserts `di` to be the instance of distinct reference type `d` to be used for querying instance `i`, which is the instance being added. `DistQuery(t,di,s)` records expended target instances `t` against distinct (reference type) instances `di`.

Finally, for `complete/3` actions, we need to modify the definition of `Completing/3`, and `CompletingAction/2`, viz.

```
Completing(i,a,st) ≡
  (a=comp_bas(i) ∨ a=complete(i) ∨ (∃di,l).a=complete(i,di,l)) ∧ st=Completed

CompletingAction(i,a) ≡ a=comp_bas(i) ∨ complete(i) ∨ (∃di,l).a=complete(i,di,l)
```

For `cancel/3` actions, we need to modify the definition of `CancellingAction/2`, viz.

```
CancellingAction(i,a) ≡ a=canc_bas(i) ∨ cancel(i) ∨ (∃di,l).a=cancel(i,di,l)
```

B.1.3 UnorderedSeq

To support `UnorderedSeq`, we firstly augment the definition of `SetRunning/5`, to include a case for `GId_UOS`, viz.

```
SetRunning(p,i,f,st,s) ≡ p=i ∧ st=Running ∨
  State(p,s)=Running ∧
   (GType(p,s)=GId_UOS ∧ (st=Running ∨ st=Initial) ∨
   ¬GType(p,s)=GId_UOS ∧ (st=Running ∧ f=EXEC ∨ st=Initial ∧ ¬f=EXEC)) ∨
  ¬State(p,s)=Running ∧ st=Initial
```

Here, we allow the children of `UnorderedSeqs` to (non-deterministically) be set to an `Initial` or `Running` state. We also change the definition of `ExecuteNextChild/4` to handle the case of completion being propagated to `UnorderedSeq` instances which still have children to run, in order to make a similar dispensation. In this case, we simply allow execution to be propagated to some yet-to-be-run child of the `UnorderedSeq`.

```
ExecuteNextChild(i',i,st,s) = (∃p,i'').Child(p,i',s) ∧
  (PropagateRunningDownInc(i'',i,st,s) ∧
   ((GType(p,s)=GId_SEQ ∨ GType(p,s)=GId_SEC) ∧ NextInitialChild(p,i'',s) ∨
    GType(p,s)=GId_UOS ∧ Child(p,i'',s) ∧ State(i'',s)=Initial) ∨
   (∃gp).Child(gp,p,s) ∧ (GType(gp,s)=GId_EXC ∨ GType(gp,s)=GId_DEF) ∧
    (∃b).Child(gp,b,s) ∧ ¬p=b ∧ PropagateCancelDownInc(b,i,st,s))
```

We augment these measures with two state constraint axioms, which are added to the BAT.

- The first says that if an UnorderedSeq is running, then at least one child should also be running.

  (∀i).GType(i,s)=GId_UOS ∧ State(i,s)=Running ⊃ (∃c).Child(i,c,s) ∧ State(c,s)=Running

- The second says that no more than one child should run at any one time.

  (∀i).GType(i,s)=GId_UOS ∧ State(i,s)=Running ⊃

  ((∀c,c').Child(i,c,s) ∧ Child(i,c',s) ∧

  State(c,s)=Running ∧ State(c',s)=Running ⊃ c=c')

B.1.4 Merge Types

Merge types, i.e. MultiMerge and Discriminator, also need dispensations to be made for them in the definitions of PropagateCancelUp/4 and ExecuteNextChild/4.

We firstly consider the case where cancellation has been propagated to a guard or continuation instance of a MultiMerge (GId_MUM) or Discriminator (GId_DIS) type. In the event that a *guard* is cancelled, the following applies. In a Multimerge, if the guard is the last running, we need to cancel remaining continuation instances (i.e. those which have not been executed); and if all other continuation instances have finished, propagate completion upwards (including the Multimerge instance). In a Discriminator, we check whether the guard being cancelled is sufficient for the failure threshold to have been reached (i.e. the minimum number of guard instances that need to fail to signify that the completion threshold can never be reached). If the threshold has been reached, we cancel the single continuation instance, cancel the remaining guards, and propagate completion upwards (including the Discriminator instance).

In the case that a continuation instance has been cancelled, the following applies. In a Multimerge, we check whether it is the last one to have been in an Initial state; if so, cancel any remaining guards that are still running. We also check whether all other continuation instances are now finished; if so, propagate completion upwards (including the Multimerge instance). In a Discriminator, we cancel any guards that are still running, and propagate completion upwards (including the Discriminator instance).

The appropriate augmentation to PropagateCancelUp/4 is as follows.

```
PropagateCancelUp(i',i,st,s) ≡
... as above for the case where i' is the root instance ...
  (∃p).Child(p,i',s) ∧
... as above for GId_SEC, GId_EXC, GId_DEF ...
   GType(p,s)=GId_MUM ∧ (PropagateCancelDownInc(i',i,st,s) ∨
    (Guard(i',s) ∧ AllGuardsFinished(p,i',s) ∧
   (CancelRemainingConts(p,i,st,s) ∨
    NoContsRunning(p,i',s) ∧ PropagateCompleteUpInc(p,i,st,s)) ∨
  Cont(i',s) ∧ (NoContsInitial(p,i',s) ∧ CancelRemainingGuards(p,i,st,s) ∨
                AllContsFinished(p,i',s) ∧ PropagateCompleteUpInc(p,i,st,s)))) ∨
GType(p,s)=GId_DIS ∧ (PropagateCancelDownInc(i',i,st,s) ∨
 (Guard(i',s) ∧ DiscThreshFailed(p,s) ∨ Cont(i',s)) ∧
  (CancelRemainingGuards(p,i,st,s) ∨ CancelRemainingConts(p,i,st,s) ∨
   PropagateCompleteUpInc(p,i,st,s))) ∨
```

```
... as above for all other cases ...
```

Instances of the fluent `Guard/2` are asserted to the BAT when guard instances of merge types are added to the CWS, and persist thereafter. Specifically, whenever the parameter `f` in `add_activity/7` is set to `EXEC`, and the parent instance is a `Multimerge` or a `Discriminator` type, an instance of the fluent `Guard(i,do(a,s))`, where `i` is the identifier of the instance being added, will be asserted to the BAT. Instances of `Cont/2` are asserted to the BAT whenever the parameter `f` in `add_activity/7` is set to `CONT`, and persist thereafter.

The predicate `PropagateCompleteUpInc(i',i,st,s)` is the same as `PropagateCompleteUp/4`, except that it also sets the state of `i'` to be `Completed`. Its variant does not do this. The predicate `AllGuardsFinished(p,gu,s)` holds just when all guards of `p`, bar `gu` (which is being cancelled), have finished (in situation `s`), viz.

```
AllGuardsFinished(p,gu,s) ≡ (∀gu').Guard(gu',s) ∧ Child(p,gu',s) ∧ ¬gu=gu' ⊃
  (State(gu',s)=Completed ∨ State(gu',s)=Cancelled)
```

The predicate `AllContsFinished(p,gu,s)` has an identical definition, except that it applies to instances for which `Cont/2` holds (i.e. continuation instances).

The predicate `NoContsInitial(p,i,s)` (resp. `NoContsRunning/3`) holds iff no continuation instance, bar `i`, of the merge instance, `p`, is in an `Initial` (resp. `Running`) state. The definition of `NoContsInitial/3` follows. The definition of `NoContsRunning/3` is a trivial variation.

```
NoContsInitial(p,i,s) ≡ (∀c).Cont(c,s) ∧ Child(p,c,s) ∧ ¬c=i ⊃ ¬State(c,s)=Initial
```

The predicate `CancelRemainingGuards/4` propagates cancellation through those guard instances which are still running, viz.

```
CancelRemainingGuards(p,i,st,s) ≡ (∃gu).Guard(gu,s) ∧ Child(p,gu,s) ∧
  State(gu,s)=Running ∧ PropagateCancelDownInc(gu,i,st,s)
```

The definition of `CancelRemainingConts/4` is trivially different – it just applies to those *continuation* instances which are in the `Initial` state.

Another dispensation that needs to be made concerns `ExecuteNextChild/4`. We need to modify the definition of this predicate for the occasion when a guard instance in a `Multimerge` or `Discriminator` is completed. The appropriate augmentation to `ExecuteNextChild/4` is as follows.

```
ExecuteNextChild(i',i,st,s) = (∃p,i'').Child(p,i',s) ∧
... as above for GId_SEQ, GId_UOS, GId_SEC, GId_EXC, GId_DEF ...
  (GType(p,s)=GId_MUM ∨ GType(p,s)=GId_DIS) ∧ Guard(i',s) ∧
    FirstInitialContinuation(p,c,s) ∧
    (GType(p,s)=GId_MUM ∧
     (PropagateRunningDownInc(c,i,st,s) ∨
      AllGuardsFinished(p,i',s) ∧ CancelRemainingConts(p,c,i,st,s) ∨
      NoContsInitial(p,c,s) ∧ CancelRemainingGuards(p,i',i,st,s)) ∨
     GType(p,s)=GId_DIS ∧ DiscThreshReached(p,s) ∧
```

```
(PropagateRunningDownInc(c,i,st,s) ∨ CancelRemainingGuards(p,i',i,st,s)))
```

In the foregoing, `FirstInitialContinuationAction/3` holds for the first continuation instance of a merge instance that is yet to be run – the instance is in an `Initial` state, viz.

```
FirstInitialContinuation(p,c,s) ≡ Cont(c,s) ∧ Child(p,c,s) ∧ State(c,s)=Initial ∧
  ¬(∃c').(c'<c ∧ Cont(c',s) ∧ Child(p,c',s) ∧ State(c',s)=Initial)
```

The predicate `CancelRemainingGuards/5` (resp. `CancelRemainingConts/5`) is a variant of its four-arity counterpart. It takes an additional argument (#2), which gives the guard being completed (resp. continuation being started), so that cancellation is effected on all other running guards (resp. yet-to-start continuations). The definition of `CancelRemainingGuards/5` is now presented. The definition of `CancelRemainingConts/5` is a simple variant.

```
CancelRemainingGuards(p,i',i,st,s) ≡ (∃i'').Guard(i'',s) ∧ Child(p,i'',s) ∧ ¬i''=i' ∧
  State(i'',s)=Running ∧ PropagateCancelDownInc(i'',i,st,s)
```

For `Discriminator`, if the guard instance completing means that the (completion) threshold for guards completing has now been reached (see Section 3.1.14, for more information), as determined by `DiscThreshReached/2`, then the continuation instance is executed and remaining guards are cancelled.

The definitions of `DiscThreshReached/2` (and `DiscThreshFailed/2`, from above) are as follows.

```
DiscThreshReached(i,s) ≡ (∃t,f,c).DiscThresh(i,t,s)=c ∧ t=c+1

DiscThreshFailed(i,s) ≡ (∃t,f,c).DiscFailThresh(i,f,s)=c ∧ f=c+1
```

Each `Discriminator` instance maintains instances of the `DiscThresh(i,t,s)=c` and `DiscFailThresh(i,f,s)=c` fluents, which are initially asserted to the BAT by the translator, see Section 6.3. The parameter `d` is the `Discriminator` instance, `t` is the completion threshold, `f` is the threshold for failed (i.e. cancelled) guards, and `c` is the count of completions in `DiscThresh/3` and the count of failures in `DiscFailThresh/3`.

There are the following successor-state axioms for these fluents:

```
DiscThresh(d,t,do(a,s))=c ≡ CompletingDiscAction(a,d,s) ∧
    (∃c').DiscThresh(d,t,s)=c' ∧ c=c'+1 ∨
  ¬CompletingDiscAction(a,d,s) ∧ DiscThresh(d,t,s)=c

DiscFailThresh(d,f,do(a,s))=c ≡ CancellingDiscAction(a,d,s) ∧
    (∃c').DiscFailThresh(d,f,s)=c' ∧ c=c'+1 ∨
  ¬CancellingDiscAction(a,d,s) ∧ DiscThresh(d,f,s)=c
```

Here, `CompletingDiscAction(a,d,s)` (resp. `CancellingDiscAction(a,d,s)`) holds when the action `a` causes one of the guards of `d` to be completed (resp. cancelled). The definition of `CompletingDiscAction(a,d,s)` is now presented. The definition of `CancellingDiscAction(a,d,s)` is trivially similar – we test for a state change to `Cancelled`, instead of `Completed`.

```
CompletingDiscAction(a,d,s) ≡ (∃i,st).Child(d,i,s) ∧ Guard(i,s) ∧
  StateChange(i,a,st,s) ∧ st=Completed
```

B.1.5 CancelActivity and Exit Types

When a CancelActivity instance is completed, it is likely that there will be a number of instances in its visibility horizon which should be cancelled. In order to support this, we need to augment the definition of the predicate StateChange/4. The updated definition of this predicate follows; in it, we have added that a completing cancel activity, which has i' in its visibility horizon, causes cancellation to be propagated up from i'.

```
StateChange(i,a,st,s) ≡
  (∃p,c,g,sc,f,j).a=add_activity(p,i,c,g,sc,f,j) ∧ SetRunning(p,i,f,st,s) ∨
  Completing(i,a,st) ∨
  (∃i').CompletingAction(i',a) ∧ PropagateCompleteUp(i',i,st,s) ∨
  (∃i').((CancellingAction(i',a) ∨ CompletingCancelActivity(a,i',s)) ∧
   PropagateCancelUp(i',i,st,s))
```

The predicate CompletingCancelActivity/3 holds just when a is a complete/1 action on a CancelActivity instance, which causes the cancellation of target instance i, viz.

```
CompletingCancelActivity(a,i,s) ≡ (∃i').a=complete(i') ∧
  (∃q,q',r,r').(CancelAct(i',q,s) ∨ CancelAct(i',q,r,s)) ∧
  InScope(i',i,r',q',s) ∧ IsType(r',r) ∧ IsType(q',q) ∧
  (State(i,s) = Running ∨ State(i,s) = Initial)
```

The fluents CancelAct/3 and CancelAct/4 record the target customised type, qtype, and plain reference type (if applicable), rtype, of CancelActivity types – see Section 3.1.16. Instances of these fluents are asserted to the BAT whenever a CancelActivity is added to the CWS, using add_activity/7, and thereafter persist. The use of IsType/2 is explained in Section 6.3.

The effects of completing a (running) Exit (GId_EXI) instance are to cancel the whole model, as determined by the following modified definition of StateChange/4 – see the last two lines. This simply says that each instance i for which CType is defined (which is just a mechanism for enumerating all instances) should be cancelled.

```
StateChange(i,a,st,s) ≡
  (∃p,c,g,sc,f,j).a=add_activity(p,i,c,g,sc,f,j) ∧ SetRunning(p,i,f,st,s) ∨
  Completing(i,a,st) ∨
  (∃i').CompletingAction(i',a,s) ∧ PropagateCompleteUp(i',i,st,s) ∨
  (∃i').((CancellingAction(i',a) ∨ CompletingCancelActivity(a,i',s)) ∧
   PropagateCancelUp(i',i,st,s)) ∨
  (∃i').a=complete(i') ∧ GType(i',s) = GId_EXI ∧ (∃c).CType(i,s)=c ∧
   ¬State(i,s)=Completed ∧ st=Cancelled
```

We also need to say that the completion of Exit instances should not be propagated upwards. To this end, we migrate to a version of CompletingAction of arity three; its additional argument

is the situation term s. In the new definition, we except complete/1 actions on Exit instances from being a "completing action".

```
CompletingAction(i,a,s) ≡ a=comp_bas(i) ∨ (complete(i) ∧ ¬GType(i,s)=GId_EXI) ∨
  (∃di,l).a=complete(a,di,l)
```

B.1.6 Multiple-Instance Types

The SitCalc-based characterisation of Multi* activity types is now presented. There is a significant overlap between how limited and non-limited Multi* types are treated. There are also important differences.

As was done for choice types, the translator wraps (join condition, execution activity) pairs of all Multi* types in a containing SeqCancel type, which makes for a simpler characterisation.

For MultiLimit (GId_MLI) and MultiLimitSeq (GId_MLS), the translator will specify the creation of n (join condition, execution activity) pairs, where n is the limit, or threshold, of the type, see Section 3.1.15. For Multi (GId_MUL) and MultiSeq (GId_MUS), the definition of the translator specifies that just one (join condition, execution activity) pair be created initially. When the join condition of the given pair completes successfully, another such pair is created. For Multi, its join condition is immediately set running. For MultiSeq, we wait until the execution activity instance from the previous pair finishes before setting the join condition of the new pair running. Pairs continue to be created, in this way, until a join condition fails.

We support the Multi* types, as we do merge and choice types, by dispensations within the definitions of PropagateCancelUp/4 and ExecuteNextChild/4. The modified definition of PropagateCancelUp/4 is now presented (in full). The changes from its previous definition are localised to the case where the parent, p, (of the instance, i', from which cancellation is being propagated) is an instance of a SeqCancel type. In this case, we need to discern whether its respective parent (if extant) is an instance of a Multi* type; and, if so, act appropriately, as will be described.

```
PropagateCancelUp(i',i,st,s) ≡
  ¬(∃p).Child(p,i',s) ∧ PropagateCancelDownInc(i',i,st,s) ∨
  (∃p).Child(p,i',s) ∧
   GType(p,s)=GId_EXC ∧
    (AllRemGuardsCald(p,i',s) ∧ PropagateCancelUp(p,i,st,s) ∨
    ¬AllRemGuardsCald(p,i',s) ∧ PropagateCancelDownInc(i',i,st,s)) ∨
   GType(p,s)=GId_DEF ∧
    (Default(i',s) ∧ AllRemGuardsCald(p,i',s) ∧ PropagateCancelUp(p,i,st,s) ∨
    ¬Default(i',s) ∧ AllRemGuardsCald(p,i',s) ∧ (∃d).Default(d,s) ∧ Child(p,d,s) ∧
     (State(d,s)=Initial ∧ PropagateRunningDownInc(d,i,st,s) ∨
     ¬State(d,s)=Initial ∧ PropagateCancelUp(p,i,st,s)) ∨
    ¬AllRemGuardsCald(p,i',s) ∧ PropagateCancelDownInc(i',i,st,s)) ∨
  GType(p,s)=GId_MUM ∧ (PropagateCancelDownInc(i',i,st,s) ∨
   (Guard(i',s) ∧ AllGuardsFinished(p,i',s) ∧
    (CancelRemainingConts(p,i,st,s) ∨
     NoContsRunning(p,i',s) ∧ PropagateCompleteUpInc(p,i,st,s)) ∨
```

```
  Cont(i',s) ∧ (NoContsInitial(p,i',s) ∧ CancelRemainingGuards(p,i,st,s) ∨
                AllContsFinished(p,i',s) ∧ PropagateCompleteUpInc(p,i,st,s)))) ∨
GType(p,s)=GId_DIS ∧ (PropagateCancelDownInc(i',i,st,s) ∨
 (Guard(i',s) ∧ DiscThreshFailed(p,s) ∨ Cont(i',s)) ∧
  (CancelRemainingGuards(p,i,st,s) ∨ CancelRemainingConts(p,i,st,s) ∨
   PropagateCompleteUpInc(p,i,st,s))) ∨
GType(p,s)=GId_SEC ∧
 ((∃gp).Child(gp,p,s) ∧
  ((GType(gp,s)=GId_MLI ∨ GType(gp,s)=GId_MLS ∨
   GType(gp,s)=GId_MUL ∨ GType(gp,s)=GId_MUS) ∧
   ((∃e).Guard(i',e,s) ∧
    (CancelRemainingPairs(gp,p,i,st,s) ∨
     CompleteOnExecActsFinished(gp,p,i,st,s)) ∨
    (∃gu).Guard(gu,i',s) ∧ (PropagateCancelDownInc(p,i,st,s) ∨
     PropagateCompleteUp(p,i,st,s))) ∨
   ¬GType(p,s)=GId_MLI ∧ ¬GType(gp,s)=GId_MLS ∧
   ¬GType(gp,s)=GId_MUL ∧ ¬GType(gp,s)=GId_MUS ∧
   PropagateCancelUp(p,i,st,s)) ∨
  ¬(∃gp).Child(gp,p,s) ∧ PropagateCancelDownInc(p,i,st,s)) ∨
¬GType(p,s)=GId_SEC ∧ ¬GType(p,s)=GId_EXC ∧ ¬GType(p,s)=GId_DEF ∧
¬GType(p,s)=GId_MUM ∧ ¬GType(p,s)=GId_DIS ∧
 (PropagateCancelDownInc(i',i,st,s) ∨ PropagateCompleteUp(i',i,st,s))
```

Instances of `Guard(j,e,s)` are asserted to the BAT when an execution activity instance, `e` (of a `Multi*` type), is added to the CWS, and persist thereafter. The `j` parameter of `add_activity/7` specifies the identifier of the pertaining join instance, `j`, and is assigned by the translator (see Section 6.3).

According to the foregoing, whenever a join condition (of a `Multi*` type) is being cancelled (given by the `(∃e).Guard(i',e,s)` case), we propagate cancellation down to any (join condition, execution activity) pairs, which are yet-to-run. These will only exist for limited types, as these are created by the translator *a priori*. This is effected by the `CancelRemainingPairs/5`, which has the following definition.

```
CancelRemainingPairs(gp,p,i,st,s) ≡ (∃b).Child(gp,b,s) ∧ b≥p ∧
  PropagateCancelDownInc(b,i,st,s)
```

We also propagate cancellation down through the (join condition, execution activity) pair whose join condition is being cancelled. Moreover, if all of the execution activity instances (which may have previously been set running) have finished, we complete the `Multi*` instance and propagate completion upwards. This is effected by the `CompleteOnExecActsFinished/5`, which has the following definition.

```
CompleteOnExecActsFinished(gp,p,i,st,s) ≡
  ((∀b,gu,c).Child(gp,b,s) ∧ b<p ∧ Child(b,c,s) ∧ Guard(gu,c,s) ⊃
                                                  ¬State(c,s)=Running) ∧
  PropagateCompleteUpInc(gp,i,st,s)
```

Whenever an execution activity instance is cancelled (given by the (∃gu).Guard(gu,i',s) case), we propagate cancellation throughout the (join condition, execution activity) pair, just in case the execution activity instance was cancelled (i.e. externally) prior to the join condition finishing. We also propagate completion upwards from the pair itself, which will have the effect (by virtue of ExecuteNextChild/4) of executing another (join condition, execution activity) pair (specifically, the join condition would be set running), for MultiLimitSeq/MultiSeq types, if extant. For MultiLimitSeq, it may be the case that there is no further pair to be set running. This would happen if all n pairs have been executed. When all (join condition, execution activity) pairs have finished, in a Multi* instance, propagating completion upwards (from a cancelled execution activity instance) will complete the Multi* instance. In this case, we continue to propagate completion further upwards.

The modified version of ExecuteNextChild/4 is as follows.

```
ExecuteNextChild(i',i,st,s) = (∃p,i'').Child(p,i',s) ∧
  (PropagateRunningDownInc(i'',i,st,s) ∧
   ((∃gp).((gp=p ∧ (GType(p,s)=GId_SEQ ∨ GType(p,s)=GId_SEC ∨
                    GType(p,s)=GId_MUS ∨ GType(p,s)=GId_MLS) ∨
            Child(gp,p,s) ∧ GType(gp,s)=GId_MLI ∧ (∃e).Guard(i',e,s)) ∧
           NextInitialChild(gp,i'',s)) ∨
    GType(p,s)=GId_UOS ∧ Child(p,i'',s) ∧ State(i'',s)=Initial) ∨
  (∃gp).Child(gp,p,s) ∧ (GType(gp,s)=GId_EXC ∨ GType(gp,s)=GId_DEF) ∧
    (∃b).Child(gp,b,s) ∧ ¬p=b ∧ PropagateCancelDownInc(b,i,st,s) ∨
  (GType(p,s)=GId_MUM ∨ GType(p,s)=GId_DIS) ∧ Guard(i',s) ∧
    FirstInitialContinuation(p,c,s) ∧
   (GType(p,s)=GId_MUM ∧
    (PropagateRunningDownInc(c,i,st,s) ∨
    AllGuardsFinished(p,i',s) ∧ CancelRemainingConts(p,c,i,st,s) ∨
    NoContsInitial(p,c,s) ∧ CancelRemainingGuards(p,i',i,st,s)) ∨
   GType(p,s)=GId_DIS ∧ DiscThreshReached(p,s) ∧
    (PropagateRunningDownInc(c,i,st,s) ∨ CancelRemainingGuards(p,i',i,st,s))))
```

When an execution activity instance in a MultiLimitSeq/MultiSeq has finished (and, thus, its containing SeqCancel completed), we execute the next (join condition, execution activity) pair, if extant. For MultiLimit types whose join condition is completing, we execute the next (join condition, execution activity) pair, if extant.

B.2 Augmentations to $\mathcal{M}_{SitCalc}[\![-]\!]$

In the following, we present the definition of $\mathcal{M}_{SitCalc}[\![-]\!]$ for those Liesbet types not covered in Section 6.3. Note that the result of translating a Liesbet model, using $\mathcal{M}_{SitCalc}[\![-]\!]$, is to assert a set of ground atoms to the BAT, which pertain to instances of fluents that hold in the initial state, S_0. Additionally, four of the action precondition axioms, presented in the previous section, i.e. those for complete,cancel/1,3, may be customised.

- The definition of $\mathcal{M}_{SitCalc}[\![-]\!]$ for synchronisation types is as follows. We present the translation of a Stop type, with both StopQuery and GoQuery queries, whose unique generic type

identifier is `GId_DST` (the convention being `D` for double query and `ST` for `Stop`). We show just the translation of a non-isolated type – the isolated case follows as above. For `Stop` with just `StopQuery` (`GId_SST`), we remove the `AssertStopGoQuery` instruction from the following. For `Go` with both queries (`GId_DGO`), we change `AssertStopStopQuery` to `AssertGoStopQuery` and `AssertStopGoQuery` to `AssertGoGoQuery`, and remove `AssertGoStopQuery` for the single-queried case (`GId_SGO`), which uses just `GoQuery`.

- $\mathcal{M}_{SitCalc}[\![$`Stop(StopQuery,GoQuery)(ctype(ctype))`$]\!]$`(p,i,f,j)` =
  `Assert(Activity(p,i,c,GId_DST,NONE,f,j))`
  where `c=genTypeId(ctype)`
  `AssertStopStopQuery(`$\mathcal{QT}_{SC}[\![$`StopQuery`$]\!]$`(i),c);` `AssertStopGoQuery(`$\mathcal{QT}_{SC}[\![$`GoQuery`$]\!]$`(i),c);`

The helper translation function, $\mathcal{QT}_{SC}[\![-]\!]$, translates a compound `Liesbet` synchronisation query into a query made against the fluent state of the basic action theory, taking the instance identifier of the synchronisation instance (which makes use of the query) as its single argument. Its definition is as follows. We omit details of translating distinct queries (i.e. queries which make use of distinct reference types, see Section 3.1.3), as these are more involved. We have previously given a flavour of how distinct queries are constructed in `SitCalc` in Appendix Section B.1.2.

- $\mathcal{QT}[\![$`True`$]\!]$`(i)` = $\top$
- $\mathcal{QT}[\![$`False`$]\!]$`(i)` = $\bot$
- $\mathcal{QT}[\![$`Completed_act(qtype)`$]\!]$`(i)` =
  `(∃t,r,q').InScope(i,t,r,q',s)` ∧ `IsType(q',q)` ∧ `State(t,s)=Completed`, where `q=genTypeId(qtype)`
  and `IsType(q',q)` ≡ `q=q'` ∨ `ISA(q',q)` ∨ `(∃q'').ISA(q',q'')` ∧ `IsType(q'',q)`
- $\mathcal{QT}[\![$`Completed_all(qtype)`$]\!]$`(i)` =
  `(∀t,r,q').InScope(i,t,r,q',s)` ∧ `IsType(q',q)` ⊃ `State(t,s)=Completed`, where ...
- $\mathcal{QT}[\![$`Completed_act(qtype in rtype)`$]\!]$`(i)` =
  `(∃t,r',q').InScope(i,t,r',q',s)` ∧ `IsType(q',q)` ∧ `IsType(r',r)` ∧ `State(t,s)=Completed`,
  where ... and `r=genTypeId(rtype)`
- $\mathcal{QT}[\![$`Completed_all(qtype in rtype)`$]\!]$`(i)` =
  `(∀t,r',q').InScope(i,t,r,q,s)` ∧ `IsType(q',q)` ∧ `IsType(r',r)` ⊃ `State(t,s)=Completed`,
  where ...
- For `Cancelled`, `Initial` and `Running` queries, replace occurrences of `Completed` accordingly in the foregoing.
- For `Finished` queries, we construct a disjunction of the pertinent `Completed` and `Cancelled` queries. For instance, $\mathcal{QT}[\![$`Finished_act(qtype)`$]\!]$`(i)` = $\mathcal{QT}[\![$`Completed_act(qtype)`$]\!]$`(i)` ∨ $\mathcal{QT}[\![$`Cancelled_act(qtype)`$]\!]$`(i)`
- $\mathcal{QT}[\![\neg$`Q`$]\!]$`(i)` = $\neg\mathcal{QT}[\![$`Q`$]\!]$`(i)`
- $\mathcal{QT}[\![$`Q`$_1|\ldots|$`Q`$_n]\!]$`(i)` = $\mathcal{QT}[\![$`Q`$_1]\!]$`(i)`$\wedge\ldots\wedge\mathcal{QT}[\![$`Q`$_n]\!]$`(i)`
- $\mathcal{QT}[\![$`Q`$_1+\ldots+$`Q`$_n]\!]$`(i)` = $\mathcal{QT}[\![$`Q`$_1]\!]$`(i)`$\vee\ldots\vee\mathcal{QT}[\![$`Q`$_n]\!]$`(i)`

The instruction AssertStopStopQuery(q,c) adds: CType(i,s)=c ∧ q within the body of the action precondition axiom for cancel(i), constituting one of the replacements for the conjunct: CType(i,s) = CUSTOMISED_SYNC_TYPE ∧ CUSTOMISED_COMPLETION_CONDITION, described in Appendix Section B.1.1. Note that c is textually replaced by the given actual parameter, as is q.

Similarly, the instruction AssertStopGoQuery(q,c) adds: CType(i,s) = c ∧ q ∧ ¬Poss(cancel(i),s) within the body of the action precondition axiom for complete(i). Note the extra condition, requiring that it is not possible to cancel the synchronisation instance, which enforces the priority of StopQuerys over GoQuerys for Stop synchronisation types.

The instructions for AssertGoStopQuery(q,c), and AssertGoGoQuery(q,c), similarly add the conjunct CType(i,s) = c ∧ q to the action precondition axioms for cancel(i), and complete(i), respectively, with the difference that the condition ¬Poss(complete(i),s) is asserted for AssertGoStopQuery(q,c); this time, no additional condition is asserted for AssertGoGoQuery(q,c), thus enforcing the appropriate priority in this case.

- For CancelActivity types:

  - $\mathcal{M}_{SitCalc}$[[CancelActivity(qtype)(ctype(ctype))]](p,i,f,j) =
    Assert(Activity(p,i,c,GId_CAN,NONE,f,j))
     where c=genTypeId(ctype)
    AssertCancelAct(i,q,c)
     where q=genTypeId(qtype)
  - $\mathcal{M}_{SitCalc}$[[CancelActivity(qtype in rtype)(ctype(ctype))]](p,i,f,j) =
    Assert(Activity(p,i,c,GId_CAR,NONE,f,j))
     where c=genTypeId(ctype)
    AssertCancelActRef(i,q,c,r)
     where q=genTypeId(qtype) and r=genTypeId(rtype)

The instruction AssertCancelAct(i,q',c') (resp. AssertCancelActRef(i,q',c',r')) inserts c=c' ∧ q=q' (resp. c=c' ∧ q=q' ∧ r=r') as a conjunct of the pertinent disjunction in the successor state axiom for CancelAct (resp. CancelActRef), which now follows.

```
CancelAct(i,q,do(a,s)) ≡
  (∃p,c,g,sc,f,j).(a=add_activity(p,i,c,g,sc,f,j) ∧
   (c=CUSTOMISED_CANCEL_ACT_TYPE ∧ q=CUSTOMISED_QUERY_TYPE ∨ ...)) ∨
  CancelAct(i,q,s)

CancelActRef(i,q,r,do(a,s)) ≡
  (∃p,c,g,sc,f,j).(a=add_activity(p,i,c,g,sc,f,j) ∧
   (c=CUSTOMISED_CANCEL_ACT_TYPE ∧ q=CUSTOMISED_QUERY_TYPE ∧
     r=CUSTOMISED_REF_TYPE ∨ ...)) ∨
  CancelAct(i,q,r,s)
```

- Choice(Chg1,...,Chgn,Chc1,...,Chcn)(ctype(ctype))⟧(p,i,f,j) =

  Assert(Activity(p,i,c,GId_EXC,NONE,f,j,SO))

  where c=genTypeId(ctype)

  Assert(Activity(i,i1,sec,GId_SEC,NONE,EXEC,NONE,SO))

  where i1=genInstId() and sec=genTypeId();

  $\mathcal{M}_{SitCalc}$⟦Chg1⟧(i1,g1,EXEC,NONE);

  where g1=genInstId()

  $\mathcal{M}_{SitCalc}$⟦Chc1⟧(i1,c1,NONE,NONE);

  where c1=genInstId()

  ...

  Assert(Activity(i,in,sec,GId_SEC,NONE,EXEC,NONE,SO))

  where in=genInstId() and sec=genTypeId();

  $\mathcal{M}_{SitCalc}$⟦Chgn⟧(in,gn,EXEC,NONE);

  where gn=genInstId()

  $\mathcal{M}_{SitCalc}$⟦Chcn⟧(in,cn,NONE,NONE);

  where cn=genInstId()

- $\mathcal{M}_{SitCalc}$⟦DefaultChoice(Chg1,...,Chgn,Chc1,...,Chcn,Chd)(ctype(ctype))⟧(p,i,f,j) =

  Assert(Activity(p,i,c,GId_DEF,NONE,f,j,SO))

  where c=genTypeId(ctype)

  Assert(Activity(i,i1,sec,GId_SEC,NONE,EXEC,NONE,SO))

  where i1=genInstId() and sec=genTypeId();

  $\mathcal{M}_{SitCalc}$⟦Chg1⟧(i1,g1,EXEC,NONE);

  where g1=genInstId()

  $\mathcal{M}_{SitCalc}$⟦Chc1⟧(i1,c1,NONE,NONE);

  where c1=genInstId()

  ...

  Assert(Activity(i,in,sec,GId_SEC,NONE,EXEC,NONE,SO))

  where in=genInstId() and sec=genTypeId();

  $\mathcal{M}_{SitCalc}$⟦Chgn⟧(in,gn,EXEC,NONE);

  where gn=genInstId()

  $\mathcal{M}_{SitCalc}$⟦Chcn⟧(in,cn,NONE,NONE);

  where cn=genInstId()

  $\mathcal{M}_{SitCalc}$⟦Chcd⟧(i,d,DEFAULT,NONE);

  where d=genInstId()

- $\mathcal{M}_{SitCalc}$[[MultiChoice(Chg1,...,Chgn,Chc1,...,Chcn)(ctype(ctype))]](p,i,f,j) =

  Assert(Activity(p,i,c,GId_MUC,NONE,f,j,SO))

  where c=genTypeId(ctype)

  Assert(Activity(i,i1,sec,GId_SEC,NONE,EXEC,NONE,SO))

  where i1=genInstId() and sec=genTypeId();

  $\mathcal{M}_{SitCalc}$[[Chg1]](i1,g1,EXEC,NONE);

  where g1=genInstId()

  $\mathcal{M}_{SitCalc}$[[Chc1]](i1,c1,NONE,NONE);

  where c1=genInstId()

  ...

  Assert(Activity(i,in,sec,GId_SEC,NONE,EXEC,NONE,SO))

  where in=genInstId() and sec=genTypeId();

  $\mathcal{M}_{SitCalc}$[[Chgn]](in,gn,EXEC,NONE);

  where gn=genInstId()

  $\mathcal{M}_{SitCalc}$[[Chcn]](in,cn,NONE,NONE);

  where cn=genInstId()

- $\mathcal{M}_{SitCalc}$[[MultiMerge(Chg1,...,Chgn,Chc1,...,Chcm)(ctype(ctype))]](p,i,f,j) =

  Assert(Activity(p,i,c,GId_MUM,NONE,f,j,SO))

  where c=genTypeId(ctype)

  $\mathcal{M}_{SitCalc}$[[Chg1]](i,g1,EXEC,NONE);

  where g1=genInstId()

  ...

  $\mathcal{M}_{SitCalc}$[[Chgn]](i,gn,EXEC,NONE);

  where gn=genInstId()

  $\mathcal{M}_{SitCalc}$[[Chc1]](i,c1,CONT,NONE);

  where c1=genInstId()

  ...

  $\mathcal{M}_{SitCalc}$[[Chcm]](i,cn,CONT,NONE);

  where cn=genInstId()

- $\mathcal{M}_{SitCalc}$[[Discriminator(m)(Chg1,...,Chgn,Chc)(ctype(ctype))]](p,i,f,j) =

  Assert(Activity(p,i,c,GId_DIS,NONE,f,j,SO))

  where c=genTypeId(ctype)

  Assert(DiscThresh(i,m,SO)=0); Assert(DiscFailThresh(i,f,SO)=0)

  where f=*eval*(n-m+1)

  $\mathcal{M}_{SitCalc}$[[Chg1]](i,g1,EXEC,NONE);

where g1=genInstId()

...

$\mathcal{M}_{SitCalc}$[[Chgn]](i,gn,EXEC,NONE);

where gn=genInstId()

$\mathcal{M}_{SitCalc}$[[Chc]](i,c,CONT,NONE);

where c=genInstId()

- $\mathcal{M}_{SitCalc}$[[MultiLimit(n)(ExecAct(join(ExecActJoin))(ctype(ctype))]](p,i,f,j) =

Assert(Activity(p,i,c,GId_MLI,NONE,f,j,SO))

where c=genTypeId(ctype)

Assert(Activity(i,i1,sec,GId_SEC,NONE,EXEC,NONE,SO))

where i1=genInstId() and sec=genTypeId();

$\mathcal{M}_{SitCalc}$[[ExecActJoin]](i1,j1,EXEC,NONE);

where j1=genInstId()

$\mathcal{M}_{SitCalc}$[[ExecAct]](i1,e1,NONE,j1);

where e1=genInstId()

Assert(Activity(i,i2,sec,GId_SEC,NONE,NONE,NONE,SO)) *do not EXEC all but first join*

where i2=genInstId() and sec=genTypeId();

$\mathcal{M}_{SitCalc}$[[ExecActJoin]](i2,j2,EXEC,NONE);

where j2=genInstId()

$\mathcal{M}_{SitCalc}$[[ExecAct]](i2,e2,NONE,j2);

where e2=genInstId()

...

Assert(Activity(i,in,sec,GId_SEC,NONE,NONE,NONE,SO))

where in=genInstId() and sec=genTypeId();

$\mathcal{M}_{SitCalc}$[[ExecActJoin]](in,jn,EXEC,NONE);

where jn=genInstId()

$\mathcal{M}_{SitCalc}$[[ExecAct]](in,en,NONE,jn);

where en=genInstId()

- $\mathcal{M}_{SitCalc}$[[MultiLimitSeq(n)(ExecAct(join(ExecActJoin))(ctype(ctype))]](p,i,f,j) =

Assert(Activity(p,i,c,GId_MLS,NONE,f,j,SO))

where c=genTypeId(ctype)

Assert(Activity(i,i1,sec,GId_SEC,NONE,EXEC,NONE,SO))

where i1=genInstId() and sec=genTypeId();

$\mathcal{M}_{SitCalc}$[[ExecActJoin]](i1,j1,EXEC,NONE);

where j1=genInstId()

$\mathcal{M}_{SitCalc}$[[ExecAct]](i1,e1,NONE,j1);

where e1=genInstId()

Assert(Activity(i,i2,sec,GId_SEC,NONE,NONE,NONE,S0)) *do not EXEC all but first join*

where i2=genInstId() and sec=genTypeId();

$\mathcal{M}_{SitCalc}$[[ExecActJoin]](i2,j2,EXEC,NONE);

where j2=genInstId()

$\mathcal{M}_{SitCalc}$[[ExecAct]](i2,e2,NONE,j2);

where e2=genInstId()

...

Assert(Activity(i,in,sec,GId_SEC,NONE,NONE,NONE,SO))

where in=genInstId() and sec=genTypeId();

$\mathcal{M}_{SitCalc}$[[ExecActJoin]](in,jn,EXEC,NONE);

where jn=genInstId()

$\mathcal{M}_{SitCalc}$[[ExecAct]](in,en,NONE,jn);

where en=genInstId()

- $\mathcal{M}_{SitCalc}$[[Multi(ExecAct(join(ExecActJoin))(ctype(ctype))]](p,i,f,j) =

Assert(Activity(p,i,c,GId_MUL,NONE,f,j,SO))

where c=genTypeId(ctype)

Assert(Activity(i,i',sec,GId_SEC,NONE,EXEC,NONE,SO))

where i'=genInstId() and sec=genTypeId();

$\mathcal{M}_{SitCalc}$[[ExecActJoin]](i',j',EXEC,NONE);

where j'=genInstId()

$\mathcal{M}_{SitCalc}$[[ExecAct]](i',e',NONE,j');

where e'=genInstId()

Assert(ActivityTemplate(sec,ROOT,0,sec,GId_SEC,NONE,EXEC,NONE,SO))

$\mathcal{M}_{SitCalc_{template}}$[[ExecActJoin]](sec,0,jt,EXEC,NONE);

where jt=genInstId()

$\mathcal{M}_{SitCalc_{template}}$[[ExecAct]](sec,0,et,NONE,j);

where et=genInstId()

Note that the translation function $\mathcal{M}_{SitCalc_{template}}[\![-]\!]$ is identical to $\mathcal{M}_{SitCalc}[\![-]\!]$, except that it asserts ActivityTemplate/9 formulas to the basic action theory for the initial state, S_0, rather than Activity/8 formulas. It also uses a distinct copy of the genInstId/0 function so that instance

numbers are generated from the value one (inclusively). These will be relative ids, which will be made absolute when join and execution activity instances are added. $\mathcal{M}_{SitCalc_{template}}[\![-]\!]$ takes an extra parameter, which is the customised activity type of the SeqCancel used to contain (join condition, execution activity instance) pairs of the Multi type.

- $\mathcal{M}_{SitCalc}[\![$MultiSeq(ExecAct(join(ExecActJoin))(ctype(ctype))$]\!]$(p,i,f,j) =

Assert(Activity(p,i,c,GId_MUS,NONE,f,j,SO))

where c=genTypeId(ctype)

Assert(Activity(i,i',sec,GId_SEC,NONE,EXEC,NONE,SO))

where i'=genInstId() and sec=genTypeId();

$\mathcal{M}_{SitCalc}[\![$ExecActJoin$]\!]$(i',j',EXEC,NONE);

where j'=genInstId()

$\mathcal{M}_{SitCalc}[\![$ExecAct$]\!]$(i',e',NONE,j');

where e'=genInstId()

Assert(ActivityTemplate(sec,ROOT,0,sec,GId_SEC,NONE,NONE,NONE,SO)) *Don't execute MultiSeq join,exec pair immediately*

$\mathcal{M}_{SitCalc_{template}}[\![$ExecActJoin$]\!]$(sec,0,jt,EXEC,NONE);

where jt=genInstId()

$\mathcal{M}_{SitCalc_{template}}[\![$ExecAct$]\!]$(sec,0,et,NONE,j);

where et=genInstId()

We also need to process the ISA specifications, for customised activity types, within a Liesbet model. These are handled by $\mathcal{M}_{SitCalc}[\![-]\!]$ in a separate translation pass. For every ISA definition in a Liesbet model, $\mathcal{M}_{SitCalc}[\![-]\!]$ inserts them into the BAT, (almost) as is, as situation-independent atoms. That is, if ctype(q) ISA ctype(q') exists in the Liesbet model, then ISA(q,q') is asserted to the BAT. Note that the translator ensures that there are no cycles engendered by the type definitions.

Finally, we discuss the processing of synchronisation rules, described in Section 3.3, by $\mathcal{M}_{SitCalc}[\![-]\!]$. The use of these rules is naturally accommodated in our SitCalc characterisation of Liesbet, by means of a straightforward augmentation of the action precondition axioms. The rules, which will have the schema: SyncRule(RType, CondQuery, GoQuery), are handled by $\mathcal{M}_{SitCalc}[\![-]\!]$ in a separate pass. For each synchronisation rule instance that exists, $\mathcal{M}_{SitCalc}[\![-]\!]$ will modify all completion and cancellation action precondition axioms, by inserting an additional necessary condition on the right-hand side of each of these axioms, viz.

$(\forall$i',c').(Descendant(i',i,s) $\vee$ i'=i) $\wedge$

IsType(c',c) $\wedge$ CType(i',s)=c' $\wedge$ $\mathcal{QT}[\![$CondQuery$]\!]$(i) $\rightarrow$ $\mathcal{QT}[\![$GoQuery$]\!]$(i)

where c=genTypeId(RType)

This says that for a completion or cancellation action to occur, concerning instance i, if i is a descendant of an instance i' (in s), or i' is i, then if i' is of customised type RType (or some sub-type thereof) and CondQuery holds for i then GoQuery must hold for i.

For the 4-argument synchronisation rule variant, `SyncRule(Ref, RType, CondQuery, GoQuery)`, we simply tag all atomic queries within `CondQuery` and `GoQuery` with the `Ref` argument, which is a plain reference type (see Section 3.1.3). For instance, `Completed_act(A)` would become `Completed_act(A in Ref)`.