(I) A Declarative Framework for ERP Systems(II) Reactors: A Data-Driven Programming Model for Distributed Applications

Stefansen, Christian Oskar Erik

Publication date:
2008

Document version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Stefansen, C. O. E. (2008). *(I) A Declarative Framework for ERP Systems(II) Reactors: A Data-Driven Programming Model for Distributed Applications*. København: Department of Computer Science, University of Copenhagen.

Ph.D. Dissertation

(I) A Declarative Framework for ERP Systems
(II) Reactors: A Data-Driven Programming Model for Distributed Applications

Ph.D. Dissertation

Christian Stefansen

September 20th 2008

Department of Computer Science
Faculty of Science
University of Copenhagen
Denmark

To those who can be swayed by argument
and those who know they do not have all the answers

Abstract

This dissertation is a collection of six adapted research papers pertaining to two areas of research.
(I) A Declarative Framework for ERP Systems:

- *POETS: Process-Oriented Event-driven Transaction Systems.* The paper describes an ontological analysis of a small segment of the enterprise domain, namely the general ledger and accounts receivable. The result is an event-based approach to designing ERP systems and an abstract-level sketch of the architecture.
- *Compositional Specification of Commercial Contracts.* The paper describes the design, multiple semantics, and use of a domain-specific language (DSL) for modeling commercial contracts.
- *SMAWL: A SMAll Workflow Language Based on CCS.* The paper shows how workflow patterns can be encoded in CCS and proceeds to design a macro language, SMAWL, for workflows based on those patterns. The semantics of SMAWL is defined via translation to CCS.
- *Using Soft Constraints to Guide Users in Flexible Business Process Management Systems.* The paper shows how the inability of a process language to express *soft constraints*—constraints that can be violated occasionally, but are closely monitored—leads to a loss of intentional information in process descriptions. This in turn makes it difficult for a process execution engine to help its users adhere to established practices. The paper then describes how soft constraints can be used to capture preferred practices explicitly in process descriptions.
- *A Work Allocation Language with Soft Constraints.* Based on the idea of soft constraints the paper explains the design, semantics, and use of a language for allocating work in business processes. The language lets process designers express both hard constraints and soft constraints.

(II) The *Reactors* programming model:

- *Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications.* The paper motivates, explains, and defines a distributed data-driven programming model. In the model a *reactor* is a stateful unit of distribution. A reactor specifies constructive, declarative constraints on its data and the data of other reactors in the style of Datalog. An attempt to update the data of a reactor starts a *reaction* during which other reactors may be affected. The reaction ends when all constraints of all affected reactors are satisfied or when it is clear that this is not possible (conflict).

Contents

Preface

This dissertation is submitted in partial fulfillment of the requirements for the Ph.D. degree at the Department of Computer Science (DIKU), University of Copenhagen. The work has been supervised by Professor Fritz Henglein, Ph.D. The disseration contains six selected and adapted research papers and a short introduction to each of them. Table 1 provides an overview; a full list of publications including position papers and technical reports can be found in Appendix A.

The dissertation contains two distinct areas of research: Chapter 1, *A Declarative Framework for ERP Systems*, enumerates some widespread problems in current Enterprise Resource Planning (ERP) systems and proposes a new architecture as a partial solution. The architecture relies on formalizations of *data* (for accounting and event tracking), *contracts* (to handle business obligations), *workflows* (to support the execution of work), and *reports* (to monitor the state of the company). After an overview of the architecture, the following sections introduce the papers on *contracts* and *workflows.* Due to space concerns some work related to *data* (specifically, accounting models) in the architecture had to be left out (see the list of publications in Appendix A). *Reports* are not covered in this work, but interested readers are referred to Brixen [5], who applied finite differencing [7] on a small report language, FunSETL, in a manner suitable for the architecture presented here. The reader should be advised that much work still needs to be done before the architecture can be built. The work at hand represents a set of papers that have one thing in common: they are all intended as the bits and pieces that one day, when all parts are done, will come together in a unified architecture.

Chapter 2, *Reactors*, presents a programming model for synchronous and asynchronous distributed applications based on stateful units of distribution called *reactors.* Computation is *data-driven* in the sense that it is controlled by declarative constraints on the state of reactors.

The dissertation is intended for people who have a technical interest in developing ERP systems or in Internet programming models. A background in computer science is strongly recommended.

Acknowledgments

I would like to thank above all Fritz Henglein, *mein akademischer Vater*, who has been the most understanding, dedicated, flexible, energetic, and visionary advisor I can possibly imagine. He has shown more confidence in me than I have ever warranted, and he, more than anyone, has permitted my years in graduate school to be an adventure rather than a confinement.

Table 1 Work by topic. Non-peer-reviewed work is only listed if included as part of the dissertation; for a full list see Appendix A. The included papers have been adapted slightly. (**J** = journal, **C** = conference, **S** = short paper, **W** = workshop, **TR** = technical report)

Article	**Venue (Type)**	**Included**
Architecture		
POETS: Process-Oriented Event-driven Transaction Systems, with F. Henglein, K. F. Larsen, J. G. Simonsen	JLAP (**J**)	√
Accounting		
Evaluating the REA Enterprise Ontology from an Operational Perspective, with S. E. Borch	INTEROP'04 (**W**)	
Contracts		
Compositional Specification of Commercial Contracts, J. Andersen,E. Elsborg, J. G. Simonsen, F. Henglein	STTT **(J)**	√
Compositional Specification of Commercial Contracts, J. Andersen,E. Elsborg, J. G. Simonsen, F. Henglein	ISoLA'04 (**C**)	
Workflow		
SMAWL*: A SMAll Workflow Language Based on CCS* (adapted version included here)	(**TR**)	√
SMAWL: A SMAll Workflow Language Based on CCS	CAiSE'05 (**S**)	
On Controlled Flexibility, with S. E. Borch	BPMDS'06 (**W**)	
Using Soft Constraints to Guide Users in Flexible Business Process Management Systems (BPMS), with S. E. Borch	IJBPIM (**J**)	√
A Work Allocation Language with Soft Constraints, with S. Rajamani, P. Seshan	CAiSE'08 (**S**)	
A Work Allocation Language with Soft Constraints, with S. Rajamani, P. Seshan (extended version included here)	ICWS'08 (**C**)	√
Reactors		
Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications, with J. Field, M.-C. Marinescu	Coordination'07 (**C**)	
Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications, with J. Field, M.-C. Marinescu	TCS (**J**)	√

If it takes a village to raise a child, then it takes an academic community to raise a Ph.D. student. In addition to Fritz, I would like to thank all my unofficial advisors who have given me their time for valuable research discussions. I remain very grateful to Greg Morrisett for hosting me at Harvard University and to Parameswaran Seshan for hosting me at Infosys Technologies, Bangalore, India. Also to Sriram Rajamani, a man of incredible talent and an admirably positive spirit, for hosting me at Microsoft Research in Bangalore, India. Thanks also to John Field for hosting my internship at IBM Research in New York and welcoming me again and again for our continued research together. I want to thank both John Field and Maria-Cristina Marinescu for making the research on the Reactors model fun, fruitful, and pleasant thanks to their impressive competence and fantastic sense of humor.

I would like to warmly thank Microsoft Development Center Copenhagen for having sponsored my Ph.D. and for being a flexible, committed, and colloquial research partner. In particular I would like to thank my designated mentor, Kim Ibfelt, for useful discussions and insights. Also thanks to our various other sponsors, including the Danish National Advanced Technology Foundation, for supporting and believing in the 3gERP project.

Thanks to Nils Andersen and Eric Jul who—in markedly different ways—helped things get started. Thanks to Espen for starting everything by referencing me. To Jakob Grue Simonsen for encouragement that worked and for being a fun and fiercely competent research associate. Signe Ellegård Borch for getting me started writing papers. My other co-authors who have agonized and laughed with me: Jesper Andersen, Ebbe Elsborg, and Ken Friis Larsen. The folks in my research group from Microsoft Development Center Copenhagen, DIKU, and ITU. My officemate and great friend at Harvard, Peter Pietzuch. And my friends and fellow students at Harvard University, the University of Hyderabad, and the University of Copenhagen, who are too numerous to list. To my colleagues at IBM Research and Microsoft Research India, in particular Camilo Tellez, Indrani Medhi, Prasad Naldurg, Debajyoti Ray, Lucia Specia, and Teofilo de Campos. I would like to thank my driver, Raaju, for making our long drives in the suffocating Bangalore traffic entertaining and for teaching me Hindi.

I remain humbly grateful to the Fulbright Commission for supporting me. Also thanks to Anton Hansen & Hustrus Mindelegat for its continuing support. Thanks to Jacob Riff for running our company in my absence and to Karin Outzen and Marianne Henriksen for always being helpful with a smile.

I would like to thank Peter Møller Neergaard for a great friendship, Helga for letting me call her that despite her name being Julija Lavrac, Kristoffer Hauskov Andersen for moral support, and Sebastian Skalberg who responded to my enrollment with the encouraging words “welcome to four years in hell”. Thank you, my dear Cambridge roommates, Maria and Taras Gapotchenko, for taking good care of me when I was bogged down by work. Thanks also to Maria’s family for providing a loving and welcoming home away from home in Connecticut, when New York got too hectic or New Haven mind-numbingly boring. I would like to thank my parents for being my friends rather than merely my parents; and I pay my deepest respect to my sweet grandmothers of a combined 189 years of age. Thanks also to the rest of my family.

Thanks to those who were.

Thank you, Irene.

Chapter 1

A Declarative Framework for ERP Systems

The objective of the work introduced in this chapter is to design

> *a declarative framework for writing process-oriented ERP systems.*

We begin by sketching the preliminaries to that objective, namely: what are ERP (Enterprise Resource Planning) systems, how did they come about, and what challenges do they face now and in the future. We then review the objective in more detail and introduce each of the papers that contribute to this objective.

1.1. A Brief History of ERP Systems

Throughout the history of computers, business systems have always been among the first to establish a foothold in industry when new technological advances were made. The financial sector was among the first to utilize mainframe computers for critical business applications, and with the introduction of the personal computer pioneered by IBM in the 80s, software packages for small business financials were among the first to reach a broader market. The foundation of all financial systems was the principle of double-entry bookkeeping, a system first employed widely by merchants in Venice in the 15th century. The system was described in 1458 by the merchant Benedetto Cotrugli in *Il libro dell'arte di mercatura*, and the first printed account was given by Luca Pacioli in 1494 in *Summa de Arithmetica, Geometria, Proportioni et Proportionalita.* Double-entry bookkeeping subsequently became the authoritative method of accounting.

Double-entry bookkeeping in its pure form, however, handles only financial accounting. Even before computers entered the realm of accounting, one had to resort to auxiliary systems outside the ledger. There were paper-based systems such as customer address books, inventory lists, payroll records, and contracts. In the days before computers the development of such *ad hoc* systems did not matter much because ideas such as interoperability and distribution were only theoretical. When the first systems were developed, they too were developed as stand-alone systems specific to one department or function of the company (e.g. project management, sales or payroll). Integration between these system was time-consuming if possible at all.

ERP (*Enterprise Resource Planning*) systems sought to change this by integrating several systems to share data and better connect processes across departments. A typical sales process, for instance, might take the concerted effort sales, inventory, and accounting to complete. ERP systems were designed in modules, and those modules were integrated with each other so as to support the flow of data mandated by common business practices. This was very reasonable in that the systems kept strict boundaries between their subsystems.

Processes. The 80s and 90s saw a significant shift from data-centric to process-centric management, and the introduction of a large set of new jargon: *Value Chain*, *Business Process Reengineering*, *Supply-Chain Management*, etc. [3]. In essence, these regard the enterprise as a collection of processes designed with the purpose of generating added value to its customer from its input to its output. If some enterprises, processes or parts of processes do not generate added value, they should be trimmed or they will be out-competed, the theory goes. To this end *Business Process Reengineering*—accused of being a reincarnation of *Taylorism/Scientific Management* that had fallen from grace earlier—prescribed continuous measurement and process improvement in a style similar to the Japanese management principle of continuous improvement*, Kaizen*, part of the *Toyota Production System* (TPS).

Whereas ERP systems had been designed with the specific intent of supporting business processes better, the common business processes were often tied intimately with the system design; fundamentally changing the processes and still having the ERP system support them was often cumbersome or even economically untenable. ERP system vendors responded by building dedicated business process modules. However, the system architecture remained modular based on the historic division of the company. The challenge of adapting the systems to support frequently changing processes gave rise to the term *business-to-IT alignment.* It remains a pivotal issue to design systems that can accommodate changes in business practice with the least possible delay.

Reporting/costing. The shared data in ERP systems made reporting faster and made it possible to report directly on a larger set of data. At the same time, the process perspective suggested new ways of reporting costs. The shortcomings of current accounting had been discussed for a while, and a host of ideas under the general label of *Management Accounting* emerged. In this vein, Robert Kaplan and Robin Cooper proposed a new cost allocation method, *activity-based costing* (ABC). In activity-based costing one allocates costs to products and services, rather than to traditional functional divisions of the enterprise. This makes apparent the cost of maintaining a product line, not just in actual production, but throughout the entire enterprise from requisition to post-sale support—it gives an approximate allocation of costs to business processes. Activity-based costing—not a profound idea on the surface of things—was made possible in part by ERP systems, which allowed a much finer-grained registration of business activities than what had been possible before. As a result there was a continuing effort to improve reporting in order to react to problems early, get the big picture by integrating data from several modules and identify cost reduction opportunities.

Supply-chain integration. Another management trend was that of *Just-In-Time* management (JIT), also part of the *Toyota Production System* (TPS). Toyota Motor Corporation, the Japanese car manufacturer, sought ways of reducing the capital tied up in inventory at any given time. To this end, parts would be produced not to stock, but just in time to be finished for their immediate assembly and delivery. This gave rise to the idea of *Supply Chain Integration.* Not only would the enterprise's own production be just in time, so would (recursively) ordering parts from the suppliers. The impact was clear in the ERP market: enterprises needed systems that would integrate with the inventories and order systems of their suppliers—and with the procurement systems of their customers. Fluctuations in market demand, one hoped, would automatically propagate back through the supply-chain to avoid excessive stock-piling or supply shortages.

Compliance and globalization. A series of serious corporate scandals—notably Enron whose downfall resulted in the dissolution of Arthur Andersen, one of the world's five largest accounting firms—lead to the introduction in the United States of the hotly contended Sarbanes–Oxley Act[1]. But this was just one example of legislation on the national level. Companies operating globally or multi-nationally found themselves spending significant resources on compliance with ever-changing local legislation. Furthermore, companies (and hence ERP system vendors) were increasingly pressured to adhere to international standards and recommendations such as BASEL II, International Financial Reporting Standards (IFRS), and International Accounting Standards (AIS).

Interweaving. Since all companies were different, or at least conceived of themselves as different, ERP were designed to be highly customizable. Except perhaps SME-market systems (Small and Medium Enterprises) or specialized solutions, all ERP systems today have their own development platforms using which companies can add more functionality using proprietary domain-specific languages (DSLs). This raises serious interweaving issues when a system consists of interacting pieces of logic and code expressing standard procedures, national legislation, industry practices, and company customizations—and the situation becomes particularly taxing when existing customization need to be merged with a system upgrade from the vendor.

Interoperability/system integration. The emphases on outsourcing, supply-chain integration, and business processes applied pressure on current ERP systems: they needed a higher degree of interoperability, and they needed, it was posited, to accommodate a process-oriented view of the world. In the first iteration, these observations lead to a surge of ERP system vendor interest in interoperable data formats (e.g. XML), automating business processes over the Internet (cf. WS-CDL, WS-BPEL), and *Service-Oriented Architecture* (SOA).

[1] Officially known as the "Public Company Accounting Reform and Investor Protection Act of 2002". In particular the aptly named section 404(a) has an impact on internal business processes in stating: *(a) Rules Required. The Commission shall prescribe rules requiring each annual report required by section 13(a) or 15(d) of the Securities Exchange Act of 1934 to contain an internal control report, which shall—(1) state the responsibility of management for establishing and maintaining an adequate internal control structure and procedures for financial reporting; and (2) contain an assessment, as of the end of the most recent fiscal year of the issuer, of the effectiveness of the internal control structure and procedures of the issuer for financial reporting.*

Vendors. The last few years have seen a concentration in the market for ERP systems, and after a series of acquisitions the biggest five vendors in terms of market share (2005) are SAP, *Oracle*, *The Sage Group*, *Microsoft Dynamics*, and SSA *Global Technologies* (later acquired by *Infor Global Solutions*). A few open source systems have appeared, most notably *Compiere*, but their impact remains negligible.

1.2. Objective

To summarize the previous section we are seeing (at least) six areas of significant challenges:

1. Business processes and business–IT alignment
2. Supply-chain integration
3. Regulatory compliance (nationally and internationally)
4. Interoperability
5. Customization weaving and evolution
6. Integrated and instantaneous reporting

Rather than soldiering on and adhering to the historic design of ERP systems, it is enticing to attempt to address these challenges by designing an ERP system architecture completely from first principles—even if this means confronting the dogma that accounting must be based on double-entry bookkeeping. This work seeks to do so with a particular focus on processes. Whereas current ERP systems have both designated process modules and integrated best practices, no ERP system to our knowledge is based directly on processes. We therefore aim to design

a declarative framework for writing process-oriented ERP systems.

Our hypothesis is that a process-based architecture will make it easier to: (i) have changes to processes reflected immediately in the running system, (ii) provide reminders, schedule tasks, perform on obligations, and exercise due diligence, (iii) know what to do next and what is urgent, (iv) report on running processes, (v) get early warnings in running processes, (vi) perform *what-if* analysis on processes, and (vii) derive a graphical user interface that adapts to changes in processes.

All the papers introduced in the following sections contribute to the goal of a unified architecture, but more research is required before the architecture can be built convincingly and used in practice. In particular, reporting and data definitions—important parts of the architecture—are outside the scope of this dissertation. Readers interested in reporting are referred to Brixen [5] and Larsen and Nissen [6], who have implemented domain-specific languages (DSLs) for reporting for the architecture.

This work considers neither supply-chain integration, regulatory compliance, interoperability nor customization, but it is certainly possible—and, indeed, the intension—that problems in those areas can be alleviated by the architecture presented here. Note, though, that it would be fully possible to build the architecture without concern for those challenges. For the architecture and its associated DSLs we do not consider performance, security or formal verification for now.

1.3. Architecture

The first paper, *POETS: Process-Oriented Event-driven Transaction Systems* (Section 3.1), presents a high-level system architecture for enterprise systems. Its point of departure is *reporting*, in other words, the desire to know the status of the company. Employees, managers, stakeholders, owners, and authorities require to know parts of the state of the company at various times to act according to their respective commissions. Starting from some of the most common reports, the paper proceeds to derive what real-world events must be captured in order to eventually produce those reports.

The first outcome is an ontological argument to support the architecture shown in Figure 1.1. The elements of the architecture are then defined formally and coded in a prototype in F#. In doing so the paper demonstrates that it is possible and, indeed, promising to construct an accounting system without using double-entry bookkeeping. The architecture comprises the following components:

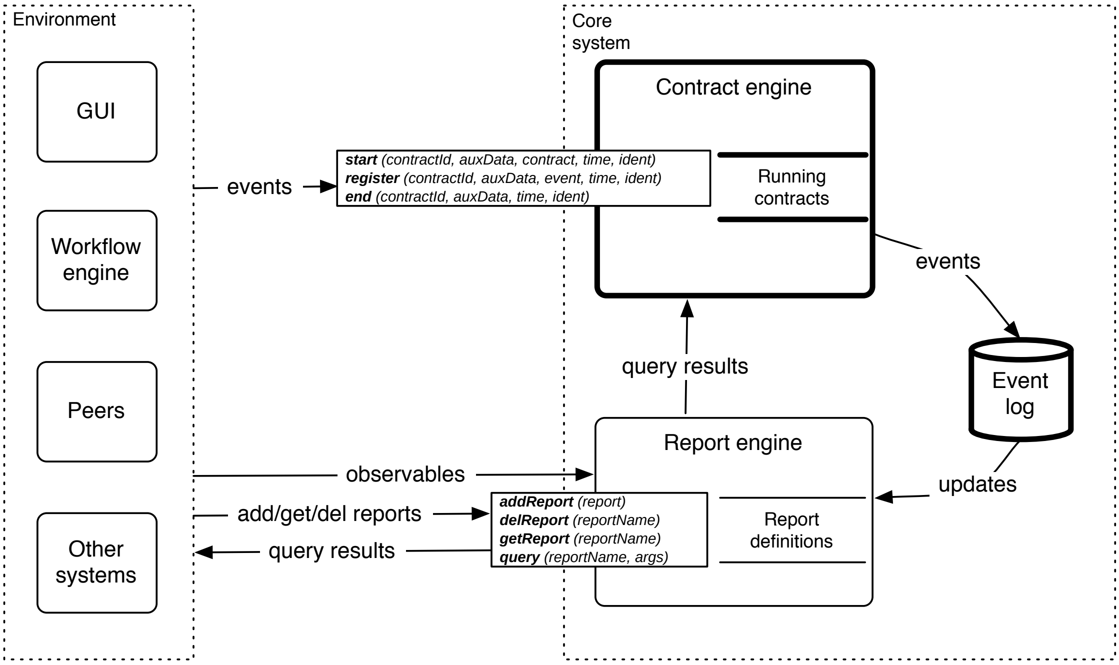
Data The fundamental data entities are agents, resources, information, and events.

Events The basic observation is that if one tracks all relevant changes to the state of the world, it is always possible to generate the desired reports. Crucially, the word "relevant" is defined as exactly that which is necessary to *produce* the desired reports. This is simple, to the point of being inane, but an important difference is made here: events are registered *as-is* with the least possible amount of interpretation. Contrary to double-entry bookkeeping, this enables us to separate what has unequivocally happened (the events) from how we choose to understand what has happened (our interpretation).

Log The log can be thought of conceptually as a list that allows events to be added, but never removed or modified. This is consistent with the definition of events above: they represent observable changes in the state of the world and cannot be undone. Should we wish to undo an event, we facilitate another (inverse) event, and the reports should then be defined accordingly to understand those two evenst as cancelling each other out. This is similar to the venerable feature of double-entry bookeeping that transactions can be cancelled out by inverse transactions but never edited or removed. The log can be thought of informally as a "flight recorder" or a "time machine" because it enables users to go back to any point in time and investigate the state of the company at that time. This is contrary to many existing ERP systems that use destructive updates for some data.

Reports Reports are functions on the log. To know some part of the state of the company, e.g. the revenue for the fiscal year, invoices that are overdue or average employee retention time, one must sift through the log to extract and accumulate the relevant events. An important part of the paper is to show that even if the events look nothing like double-entry transactions, the income statement, the balance sheet, and other typical fiscal reports can be expressed as reports on the log of events.

Figure 1.1 System architecture



Contracts Contracts are explicit representations of processes. They guard the system and specify what is allowed in the sense that events must be valid according to one of the ongoing contracts to be admitted in the system log. Having an explicit representation of (running) contracts enables us to analyze, formally verify, monitor, expose, persist, and move running instances.

The remaining sections treat selected elements of the architecture. Note again, that reporting and data definitions are not part of this dissertation. The last section presents research on workflows. Workflows are not part of the core architecture, but essential as an interface between the core system and human workers.

1.4. Business Data: Accounting Models

An important part of the architecture is how business data are represented. Since the desire is to have a uniform data model for ERP systems, double-entry bookkeeping must either be replaced or subsumed. Due to space concerns my work in this area is not included here. Interested readers are referred to the qualification report [8] for a formalization of double-entry bookeeping and a critique of the *Resources/Events/Agents* (REA) model (also published separately [4]).

1.5. Business Processes: Contracts

In the paper *Compositional Specification of Commercial Contracts* (Section 3.2) we consider how contracts can be expressed in a domain-specific language designed as part of the overall architecture. The language design is based on an ontological analysis of a pool of more than twenty actual contracts. The paper explains the properties and multiple semantics of the contract language and shows how selected contracts are written in the language.

1.6. Business Processes: Workflows

Workflows are not part of our core ERP system architecture, but they act as significant producers of events (as proxies to human workers), and they are closely related to contracts. Three papers on workflow have been included:

1. *SMAWL: A SMAll Workflow Language Based on CCS* (Section 3.3) begins by investigating if a process calculus, namely CCS (*Calculus of Communicating Systems*), can be used to model workflows. A popular, albeit somewhat inaccurate, litmus test is the suite of *workflow patterns* published by van der Aalst *et al.* [9]. Unsurprisingly, the patterns can, indeed, be modeled in CCS, but the resulting code easily becomes crufty. The paper therefore proceeds to define via translation the semantics of a small macro language, SMAWL.
2. *Using Soft Constraints to Guide Users in Flexible Business Processes Management Systems* (Section 3.4) represents a departure from the idea that workflows are strict. It argues that workflows must be pliable and sketches how this might be done.
3. *A Work Allocation Language with Soft Constraints* (Section 3.5) implements the ideas of the previous paper. It develops a language for expressing flexible work allocation rules. It functions orthogonally to any process language in the sense that the allocation aspect can be toggled on and off. When toggled off, all allocations are permitted. The language is based on requirements from both academia and industry representatives, and its semantics is defined with particular care that the language can be used in a wide range of settings.

Chapter 2

Reactors

The *Reactors* programming model came about from the desire to write declarative, constructive constraints on distributed data. Reactors are stateful units of distribution that communicate with each synchronously and asynchronously to maintain their data constraints. The paper, *Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications* (Section 3.6), explains the model, defines its semantics, and makes the case that it is applicable to Internet programming in general. The model is inspired by Datalog (see Abiteboul [1] for a thorough introduction) and to a limited extent by the Actor model (see e.g. Agha [2]).

Chapter 3

Research Papers

POETS:
Process-Oriented Event-driven Transaction Systems[⋆]

Fritz Henglein, Ken Friis Larsen, Jakob Grue Simonsen,
Christian Stefansen

Department of Computer Science, University of Copenhagen (DIKU)

Abstract

We present a high-level enterprise system architecture that closely models the domain ontology of resource and information flows in enterprises. It is:

Process-oriented: formal, user-definable specifications for the expected exchange of resources (money, goods, and services), notably contracts, are represented explicitly in the system state to reflect expectations on future events;

Event-driven: events denote relevant information about real-world transactions, specifically the transfer of resources and information between economic agents, to which the system reacts by matching against its portfolio of running processes/contracts in real time;

Declarative: user defined reporting functions can be formulated as declarative functions on the system state, including the representations of contractual residual obligations.

We introduce the architecture and demonstrate how analyses of the standard reporting requirements for companies—the income statement and the balance sheet—can be used to drive the design of events that need registering for such reporting purposes. We then illustrate how the multi-party obligations in trade contracts (sale, purchase), including pricing and VAT payments, can be represented as formal contract expressions that can be subjected to analysis.

To the best of our knowledge this is the first architecture for enterprise resource accounting that demonstrably maps high-level process and information requirements directly to executable specifications.

Key words: process, contract, type, event, financial system, enterprise resource planning, ERP

Preprint submitted to Elsevier *20 September 2008*

1 Introduction

Enterprise Resource Planning (ERP) systems integrate several information systems of an organization into one system. Financials, manufacturing, project management, supply chain management, human resource management, and customer relationship management are typical components of an ERP system [1] .

Enterprise Resource Planning (ERP) systems do in principle a simple thing: they model activities in an enterprise and register relevant information about them so they can be queried and interpreted for whatever is deemed important or required to run the company, ranging from high-level strategy to down-on-the-floor operations.

It may be surprising then to learn that even ERP systems targeted at small and medium-sized enterprises such as Microsoft Dynamics NAV [2] or AX [3] comprise several million lines of code and thousands of tables in a relational database management system. The software architecture of such systems typically consists of a decomposition of the system in terms of tables, code units, (user interface) forms specifications, data mappers for input/output, etc., built as a three-tier client-server system running on centralized database servers for their data repository. They do not reflect the "architecture" of enterprises, which consists of resources (goods, services, money) being bought, processed, sold, and moved around between different parts, whether physical, functional or organizational, of a company. As a result the translation of business requirements into running code has to span a large and costly semantic and architectural divide.

Despite the widespread use and business reliance on ERP systems little effort has been spent applying sound theoretical principles to designing an ERP system from first principles. We argue that using well-known principles from process algebra and functional programming, as we do here, gives an elegant and much cleaner architecture for ERP systems.

⋆ This work has been supported by the Danish National Advanced Technology Foundation under Project *3rd generation enterprise resource systems (3gERP)*. See `http://www.3gERP.org`.

Email addresses: `henglein@diku.dk` (Fritz Henglein), `kflarsen@diku.dk` (Ken Friis Larsen), `simonsen@diku.dk` (Jakob Grue Simonsen), `cstef@diku.dk` (Christian Stefansen).

[1] `http://en.wikipedia.org/wiki/Enterprise_resource_planning` retrieved on June 10th 2008.

[2] `http://www.microsoft.com/dynamics/nav` retrieved on June 10th 2008.

[3] `http://www.microsoft.com/dynamics/ax` retrieved on June 10th 2008.

Taking as our fundamental goal that the *ontological* architecture for requirements also be the architecture (our motto is: the (formalized) requirements *are* the system.), we develop an event-driven architecture aimed at directly reflecting the domain-oriented requirements. The key motivation is shortening the distance between requirements and their formal expression for rapid system prototyping, implementation, and continuous system adaptation to changing processes and information needs.

Given the size of ERP systems, it is not possible to cover all functionality. Hence this paper restricts itself to only consider some of the functionality typically contained in the finance module of ERP systems.

1.1 Contributions

The paper contributes the following:

An ERP system model An ERP system model that
- directly and declaratively reflects *enterprise domain concepts*, notably resources, events, agents, information/reports and processes
- does not encumber the system with non-enterprise concepts such as "database management system", "client-server", memory management, etc.
- separates interpretation and registration of (business) events
- separates (monetary) valuation from resources and thus enables re-valuation
- supports user-defined contract specifications, which can be executed and analyzed

Design methodology An enterprise design methodology based on identifying relevant events to be registered and, consequently, processes to be modeled from reporting requirements

Formal semantics A formal semantics (architecture reduction semantics) that is:
- event-trace based, yet
- orthogonal to the contract (choreography) language, rendering the concrete choice of language independent of the architecture (we have used the contract language by Andersen *et al.* [AEH$^+$06], but, e.g., WS-CDL could be used instead).

Prototype Illustrative parts of a text-based prototype in *F#*.

1.2 Overview

In Section 2 we address the question of what to model, i.e., represent as data, and what not. We start with the premise that only the information required for reporting purposes need be modeled. We take the standard reports that every company must supply as our starting point. Section 3 formalizes the entities discovered in this process: agents, resources, valuations of resources (prices), and events that must me registered for the given reporting requirements. Section 4 illustrates how declarative reports can be mapped in a natural fashion to the functional programming language F#.

Section 5 discusses the need for representing not only *ex post* events, but also processes, specifically *contracts*. Section 5.2 describes the resulting architecture: incoming events modeling real-world activities are matched against process/contract states expressing current expected/legal events and then put into a log. In Section 5.3 we describe an example of a contract specification language and show how it integrates with the overall architecture.

We discuss related work in Section 6 and conclude in Section 7.

2 Domain model of resource accounting

In this section we derive a stringent description of the functionality of any system that models the *economic status* of a company. Initially, we notice that at any given point in time, such a status can be derived if we have registered with sufficient granularity the *events* that have occurred up to that point. Events are any atomic, observable change in the state of the world. The challenge lies in selecting what events to register and what events to ignore—and in particular in doing so without a bias to existing methods of accounting.

Receiving an amount into the company's bank account seems inherently *relevant*, whereas the acquisition of a cup of coffee from the machine on the second floor by the clerk may keep the coffee-drinking accountant happy for a while, but is unlikely to have a direct, causal, unequivocal, and important effect on the economic status of the company. But this distinction is vague. A better first approximation to the requirements of relevant events for any accounting system is the union of all events that are mentioned in (accounting) legislation and current accounting practice. However, this approach imports exactly the unfortunate bias towards existing accounting practices which we seek to avoid. For example, many simple accounting systems do not register when a customer accepts a quote and it becomes an order. This is because such an event has no *direct* effect on any account or in a traditional ledger or on the income statement. Granted, it has the *indirect* effect of starting a process that generally leads to invoicing, and invoicing has a direct effect on an A/R (*Accounts receivable*) account and a revenue account.

In this distinction lies the key to the definition of a *relevant* event: the receipt of an order was not relevant because it had no effect on the income statement (assume for the sake of argument that the income statement is our only company status report of interest). Therefore receipt of an order is not registered. This leads to a pleasant and quite obvious definition of relevant: an event is relevant if and only if it has a direct effect on any of the reports of the company status that we want the system to produce.

The first step is to determine what reports will be needed. Once these are established, we can proceed to find the event types that affect those reports.

2.1 Reports

We assume that the company needs to produce the following five reports: an income statement, a balance sheet, a cash flow statement, a list of open (not yet paid) invoices and a VAT (value-added tax, somewhat similar to sales tax) report. These are chosen because they constitute the core functionality of the traditional accounting system that is our benchmark—as well as the legal requirements faced by any registered company. For brevity of exposition, however, we shall concern ourselves with only two reports: the income statement and the balance sheet.

We now consider as an example a company that sells goods. That is, the company sustains itself by buying goods and selling them with a profit.

For ease of exposition we shall ignore taxes (other than VAT), interests, mortgages, and other advanced accounting phenomena. We argue that this is without loss of generality, as these are similar to the basic accounting phenomena outlined here.

What follows are bare-bones definitions of the reports. For a more thorough exposition the reader is referred to a standard text on accounting [WKK04].

	Revenue (Gross income)
−	Cost of goods sold
=	**Contribution margin**
−	Fixed costs
−	Depreciation
=	**Net operating income**

Fig. 1. **Income Statement** The Income Statement summarizes the profits and losses of the company over a given period (hence also the name *Profit & Loss Statement*).

Assets	*Liabilities and owners' equity*
Fixed assets	**Liabilities**
Current assets	Accounts payable
Inventory	VAT payable
Accounts receivable	**Owners' equity**
Cash and cash equivalents	*Total liabilities and owners' equity*
Total assets	

Fig. 2. **The Balance Sheet** The Balance Sheet summarizes assets, liabilities, and owners' equity at a particular point in time. The balance sheet should always satisfy the fundamental invariant known as the *Accounting Equation*, which states that *Assets = Liabilities + Owners' equity.*

2.2 Events

To be able to generate the reports outlined above we must identify the changes in the state of the world that affect each item that report. Such changes in the state of the world are reported as *events*, and we will assume—based on *The Theory of the Firm* [Cre75]—that the events of interest are transfers of economic resources or information between self-interested agents (economic entities).

All that happens can be expressed in terms of a few basic types of events:

- Transmit a resource or money from one agent to another
- Convey information from one agent to another
- Transform a set of resources into another set of resources

These events can (and should) also be further refined, which is what we will do next. Receiving a resource can be something for the company (land, property, paper clips) or something intended for selling with a profit. Some resources are put in stock for later consumption or sale, whereas other resources are consumed the moment they are received (e.g., a session with a business consultant).

Recall that we are trying to mimic only the functionality of a standard accounting system. For this reason the events we need to register—although chosen based on what the desired reports mandate rather than just what events we *think* might be necessary—look strikingly similar to that which a traditional accounting system registers in its ledger. However, the point remains: we do not randomly decide what events are relevant and made such a decision subject to biased models or paradigms; we simply choose what gives the necessary granularity for what we eventually desire to report.

Events that affect the Income Statement

Revenue Affected by sending an invoice for normal sale (not fixed assets, for instance) to a customer.

Cost of goods sold Affected by making an inventory requisition relating to a customer order. Notice that the requisition event does not inherently contain information about the purchase price, and thus the purchase price must be looked up or computed. The time of registration varies, but commonly the cost of goods sold is registered at the time where the sale is invoiced to the customer.

Fixed costs Affected by receiving an invoice for a fixed cost.

Depreciation Not an event, but a continuous process. Here depreciation is *computed* (based on the events describing purchases and sales of assets). That is, depreciation is computed as a report (an interpretation) which can be invoked per need. If depreciation were registered as discrete phantom events—as typical accounting practice mandates—it would be difficult to change the depreciation method retro-actively.

Events that affect the Balance Sheet

Fixed assets Affected (a) by receiving an invoice for a fixed asset and (b) by sending an invoice for a fixed asset

Raw materials Affected (a) by receiving an invoice for raw materials or (b) by making a requisition for raw materials from the inventory for production.

Finished goods Affected by (a) sending an invoice to a customer for a good or (b) by receiving finished goods from the production process.

Accounts receivable Affected by (a) sending an invoice or credit note to anyone and (b) receiving payment pertaining to an invoice into the money bin or the bank account

Cash Affected by (a) money being put in the money bin and (b) money being taken from the money bin

Bank account Affected by (a) money being deposited into our bank account and (b) money being withdrawn from out bank account

Accounts payable Affected by (a) receiving an invoice or credit note from anyone and (b) sending payment pertaining to an invoice from the money bin or the bank account

VAT payable Affected by issuing or receiving an invoice containing items on which VAT is due.

Owners' equity Affected by transferring money or resource to and from the owners.

2.3 An Example

Figure 3 shows an example of the events that need to be registered by a company over a period of time.

3 Formal model of resource accounting

3.1 Agents

Agents represent whole companies as well as categorizations within a company such as organizational unit, location, etc. They can be thought of as partitioning a company, possibly along multiple dimensions for any suitable purpose. To be able to distinguish resources determined for (re)sale, for use inside the company, but with long-term depreciation or instantaneous depreciation, we require conceptual *resource containers* such as *operations*, *fixed assets*, *losses*. In our model agents are thus not restricted to modeling only legal persons, organizational units, roles, and

1 Receive 3 iPhones and 2 MacBooks from supplier X

2 Receive 2 iPhones and 1 MacBooks from supplier Y

3 Receive an invoice from X for 3 iPhones (3 * 400 USD incl. VAT) and 2 MacBooks (2 * 2000 USD incl. VAT) and rush delivery charge (20 USD – VAT exempt)

4 Receive invoice from Y for 3 iPhones (3 * 420 USD incl. VAT) and 2 MacBooks (2 * 1940 USD incl. VAT) and shipping (100 USD incl. VAT)

5 Deposit 5220 USD into X's bank account

6 Send check to Y to the amount of 5240 USD

7 Observe on our bank account that check has been cashed

8 Receive order from A of 1 MacBook and 1 iPhone priced at 3000 USD incl. VAT

9 Deliver 1 MacBook and 1 iPhone to A

10 Receive from A 3000 USD into our bank account

11 Pay VAT due

12 *A year passes*

13 Deliver, invoice, and receive payment for 1 MacBook worth 800 USD incl. VAT to Z

Fig. 3. An example of events relevant to a company

actual persons in the real world as in the REA-model.

We have seen that for the reporting purposes presented and analyzed in Section 2 it is sufficient to have an agent representing a company and a set of internal agents, where each internal agent has a unique company that it belongs to. This can be captured by defining

$$\begin{aligned} \mathit{Agent} &= \mathit{CompanyName} \times \mathit{InternalAgentName} \\ \mathit{CompanyName} &= \mathit{String} \\ \mathit{InternalAgentName} &= \mathit{String} \end{aligned}$$

where the empty string ϵ, also written as "me", is the designated internal name for the company itself. We write $C.I$ instead of (C, I) and C instead of $C.\epsilon$. We write $A \leq C$ if $A = (C, I)$ for some $I \in \mathit{InternalAgentName}$.

3.2 Resources

A resource is either: empty; a unit of a resource identified by a unique resource name such as "iPhone", "water", "Picasso's *Guernica* painting"; a scaled resource; or the formal sum of two resources, which models taking their union.

Resource types are usually categorized into uncountable, countable or binary (unique, one either has that particular resource or not), corresponding to allowing Real_0^+, $\mathbb{N}$ or $\{0, 1\}$ as scaling factors. We shall not distinguish between resource types here, but model all resource types as being uncountable. To simplify the presentation of the semantic resource model we shall not dwell on the handling of unique and countable resources, but treat all resources as uncountable. This is without loss of expressive power since $\mathbb{N}$ and $\{0, 1\}$ can be embedded into the nonnegative reals, and we can maintain a mapping from resource types to their category and referring to it during computations on resources to ensure that the corresponding invariant is satisfied. Finally, we also allow "negative" resources, which ensures that the difference between resources is always defined. (compound) resource is a finite from resource. Since resources can be added a compound resource thus is represented by a finite map from resource types (such as "iPhone", "water", etc.) to the real numbers *Real*.

Formally, this amounts to resources being modeled by the *infinite coordinate space* Real^∞ over the field *Real* of real numbers, an infinite dimensional vector space whose elements are infinite vectors of reals $(k_1, k_2, ...)$ with finitely many nonzero elements. We shall also write them as $\sum_{i=0}^{\infty} k_i X_i$ to emphasize their treatment as elements of a vector space with scalar multiplication and addition defined by:

$$k(\sum_{i=0}^{\infty} k_i X_i) = \sum_{i=0}^{\infty} (k k_i) X_i$$

$$(\sum_{i=0}^{\infty} k_i X_i) + (\sum_{i=0}^{\infty} k'_i X_i) = \sum_{i=0}^{\infty} (k_i + k'_i) X_i$$

For convenience we also use descriptive strings as names for the different resource types instead of formals X_i; such as 2 iPhone +3 MacBook denotes the compound resource consisting of the 2.0 times one unit of the resource type denoted by “iPhone” plus 3.0 times one unit of the resource type denoted by “MacBook”. This is instead of using $2X_{234} + 3X_{4117}$ or $(0, \ldots, 0, 2, 0, \ldots, 0, 3, 0, \ldots)$ where the 2 occurs in the 234th component and the 3 in the 4117th component of the infinite vector and 234 is the “item number” of one resource (presumably iPhones) and 4117 of another (presumably MacBooks—the strings do not mean anything by themselves).

To summarize, we have the following spaces for modeling resource names and resources

$$\begin{aligned} \mathit{ResourceName} &= \mathit{String} \\ \mathit{Resource} &\cong \mathit{Real}^{\infty} \end{aligned}$$

where we assume the existence of numbering function that maps resource names to natural numbers for indexing resource vectors. (Such a function corresponds to an product catalog with item numbers serving as indexes of the resource vectors.)

Apart from the vector operations scaling and addition (including subtraction), we shall define an additional vector operation, which is used in connection with costing.

We say resource $R = \sum_{i=0}^{\infty} k_i X_i$ is *nonnegative* and write $R \geq 0$ if $k_i \geq 0$ for all $i \in \mathbb{N}$. We write $R \leq R'$ if $R' - R \geq 0$.

We define the operation $\mathit{Subtract} : \mathit{Resource} \times \mathit{Resource} \rightarrow \mathit{Resource} \times \mathit{Resource}$ as follows: For $R_1, R_2 \geq 0$ we define $\mathit{Subtract}(R_1, R_2) = (R'_1, R'_2)$ if: (1) $R'_1, R'_2 \geq 0$; (2) $R_1 + R'_2 = R'_1 + R_2$, and (3) R'_1, R'_2 are least with respect to $\leq$ amongst vectors that have the first two properties.

Subtract models the situation where we would like to subtract a compound resource R_2 from another resource R_1. If there are more units of a particular resource type in R_1 than in R_2, after subtracting those in R_2 from those in R_1 the remaining units are returned as part of R'_1, with R'_2 receiving 0. If there are fewer, the roles of R'_1 and R'_2 are reversed. E.g.,

$$\begin{aligned} &\mathit{Subtract}(2\,\mathrm{iPhone} + 1\,\mathrm{MacBook} + 1\,\mathrm{Guernica}, 1\,\mathrm{iPhone} + 5\,\mathrm{MacBook}) = \\ &\qquad\qquad\qquad\qquad (1\,\mathrm{iPhone} + 1\,\mathrm{Guernica}, 4\,\mathrm{MacBook}). \end{aligned}$$

3.3 Valuations

Valuations are an integral part of resource accounting. A valuation is a linear map from resources to a subspace of the resources, normally a *single* resource type that represents some designated currency (say USD). They are used for a number of purposes: The most obvious is as a *price list*, that is, the basis of a sales contract. Another is to express the result of reappraisal of certain resources a company may have (e.g., as a basis for extraordinary write-downs of assets).

Valuations distribute over scaling of resources and taking their union (sum):

$$\begin{aligned} \mathit{Value}(R_1 + R_2) &= \mathit{Value}(R_1) + \mathit{Value}(R_2) \\ \mathit{Value}(kR_1) &= k\,\mathit{Value}(R_1) \end{aligned}$$

In other words, the total value of two resources R_1 and R_2 is the sum of their values; and the value of k copies of a resource is k times the individual cost. This is tantamount to saying that valuations are *linear functions (homomorphisms)* between the vector spaces *Resource* and *Real*.

$$\mathit{Valuation} = \mathit{Hom}(\mathit{Resource}, \mathit{Real})$$

The linearity requirement does not limit pricing in sales contracts to only applying the “same” unit price in every trade. Larger volumes may just let the buyer negotiate a lower price, corresponding to a *different* valuation as the basis for a sales contract. Linearity rather models *distributing* value of resources when they are split up as part of movement to and from inventory, production units (modeled as other agents), etc. Here the basic assumption is always that items acquired in the same transaction are all equally priced. So, if 8 iPhones were bought for stock and 4 of them are transferred to operations later on, the original 4 and the transferred ones are given the same unit price, whatever that is.

3.4 Events

As stated in Section 2.2 we need events to model transmission of resources, transformation of resources, and conveyance of information. We will refer to these collectively as *transactional events*—or in the definition *TransactEvent*. Each event is equipped with a *time stamp* and an *identifier* for correlating different events to each other. Time-stamps are represented by real numbers that model the time difference to some given base point (say, January 1, 1970, 00:00:00). We use strings for identifiers.

We arrive at the following definitions:

$$\begin{aligned}
\mathit{Event} &= \mathit{LogEvent} \times (\mathit{Time} \times \mathit{Ident}) \\
\mathit{LogEvent} &= \mathit{TransactEvent} \\
\\
\mathit{TransactEvent} &= \mathit{TransmitEvent} \uplus \mathit{TransformEvent} \uplus \mathit{InformEvent} \\
\mathit{Time} &= \mathit{Real} \\
\mathit{Ident} &= \mathit{String} \\
\mathit{TransmitEvent} &= \mathit{Agent} \times \mathit{Agent} \times \mathit{Resource} \\
\mathit{TransformEvent} &= \mathit{Agent} \times \mathit{Resource} \times \mathit{Resource} \\
\mathit{InformEvent} &= \mathit{Agent} \times \mathit{Agent} \times \mathit{Information}
\end{aligned}$$

Note that $\uplus$ denotes disjoint union.

We are left with modeling the information conveyed in information events. The only information events that need be captured for the given reporting purposes is the transmission of *invoices*. An invoice carries a lot of information in practice: sender and receiver, their contact information, company tax code information (for VAT purposes), which resources are delivered, and expenses for delivering them such as shipping and handling. The resources are usually also split up into resource names, number of units (scale), and what each unit costs.

Sender, receiver, time and an identification are conveyed as part of the event itself. The remaining information needed we represent in the information part. Note that we only capture information that is required for our reporting purposes! This consists of the resources delivered/to be delivered and price information: [4]

$$\mathit{Information} = \mathit{Resource} \times \mathit{PriceInformation}$$

The price information, in turn, consists of a *price* of the resources and their *value*

[4] We simplify matters somewhat by not modeling here expenses for facilitating the delivery. For convenience we assume their cost is already amortized over the actually resources delivered.

added tax (VAT).

$$PriceInformation = Price \times VAT$$

The price is a valuation. To simplify matters we disregard the rather complex conglomerate of legislation on VAT calculation, and represent VAT as a valuation as well.

$$\begin{aligned} Price &= Valuation \\ VAT &= Valuation \end{aligned}$$

Note that we can model bundled prices. A bundle is a compound resource, but with its own price different from the valuation of the bundle in terms of its constituent resources. This can be modeled by introducing the bundles as an "abstract" new resource resource type, whose "implementation" (definition) is the compound resource it is made up of. This allows pricing of that resource independently of its constituent resources.

Beyond the above information an invoice also contains payment terms and other information. Following our principle of capturing information that is required to produce our target reports and no more, this is not represented. In practice, a reference to the actual invoice from the information event will give access to all such information.

With the above formalization the events of the example in Figure 3 can be rendered as shown in Figure 4.

3.5 Reports

Let us reconsider our reports from Section 2. They are parameterized over:

- a time period, with starting date and end date,
- a time-stamped set of events,

The first requirement also holds for the balance sheet: the difference is that the period of interest for it usually covers the starting date of the company until a particular period end date, whereas the other reports use a more recent start date.

1	Receive 3 iPhones and 2 MacBooks from supplier X	transmit(X, C, 3 iPhone + 2 Mac, 2008-01-15)
2	Receive 2 iPhones and 1 MacBooks from supplier Y	transmit(Y, C, 2 iPhone + 1 Mac, 2008-01-19)
3	Receive an invoice from X for 3 iPhones (3 * 400 USD incl. VAT) and 2 MacBooks (2 * 2000 USD incl. VAT) and rush delivery charge (20 USD – VAT exempt)	inform(X, C, invoice (iPhone, 3, 400 USD, 25%), (Mac, 2, 2000 USD, 25%), (fee, 1, 20 USD, 0%), 2008-01-19)
4	Receive invoice from Y for 3 iPhones (3 * 420 USD incl. VAT) and 2 MacBooks (2 * 1940 USD incl. VAT) and shipping (100 USD incl. VAT)	inform(Y, C, invoice (iPhone, 3, 1680 USD, 25%), (Mac, 2, 7760 USD, 25%), (shipping, 1, 400 USD, 25%), 2008-01-19)
5	Deposit 5220 USD into X's bank account	transmit(C.bank, X.bank, 26100 USD, 2008-01-22)
6	Send check to Y to the amount of 5240 USD	transmit(C, Y, right to draw 26.300 USD from C.bank, 2008-01-22)
7	Observe on our bank account that check has been cashed	transmit(C.bank, Y.bank, 26300 USD)
8	Receive order from A of 1 MacBook and 1 iPhone priced at 3000 USD incl. VAT	enter contract for exchange of (1 iPhone + 1 Mac, 1, 12000 USD, 25%)
9	Deliver 1 MacBook and 1 iPhone to A	transmit (C, A, 1 Mac, 2008-01-26)
10	Receive from A 3000 USD into our bank account	transmit (A.bank, C.bank, 15000 USD 2008-01-30)
11	Pay VAT due	transmit (C.bank, IRS, VAT_due())
12	*A year passes*	
13	Deliver, invoice, and receive payment for 1 MacBook worth 800 USD incl. VAT to Z	transmit (C.ops, Z, 1 Mac, 2009-01-30); inform (C, Z, invoice (Mac, 1, 3200 USD, 25%), 2009-01-30); transmit (Z.bank, C.bank, 4000 USD, 2009-02-06);

Fig. 4. Example with formal events

$$
\begin{aligned}
\mathit{InvoicesReceived} &= \{(i,(A,R,(\mathit{price},t)))\\
&\qquad : (\mathit{inform}(A,B,(R,\mathit{price})),t,i) \in \mathit{Events} \mid B \leq me, A \not\leq me\}\\
\mathit{InvoicesSent} &= \{(i,(B,R,(\mathit{price},t)))\\
&\qquad : (\mathit{inform}(A,B,(R,\mathit{price})),t,i) \in \mathit{Events} \mid A \leq me, B \not\leq me\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{InvAcq} = \mathit{Sort}\{(R,(\mathit{price},t))\\
&\qquad : (\mathit{transmit}(me, me.\mathit{Inventory}, R),t,i) \in \mathit{Events},\\
&\qquad\quad (A,R',\mathit{price}) = \mathit{InvoicesReceived}(i)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{GoodsSold} = \sum\{R : (i,(A,R,(\mathit{price},t))) \in \mathit{InvoicesSent}\}\\
&\mathit{FIFOCost} = \mathit{foldl}(\mathit{accumCost},(0,\mathit{GoodsSold}),\mathit{InvAcq})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{accumCost}((R,((p,m),t)),(\mathit{total},Q)) =\\
&\quad \text{let } (R',Q') = \mathit{Subtract}(R,Q) \text{ in}\\
&\qquad (\mathit{total} + p(R-R'),Q')\\
&\quad \text{end}
\end{aligned}
$$

Fig. 5. Definitions of selected subreports

Note that the time period is solely used to filter events out that are outside the designated period. Once that is done, each report can then be defined as a function on the remaining events.

In other words, we can define each report as a function from a set of time-stamped events, represented as an event sequence obeying the the events' timestamps, to the designated information of the report.

3.5.1 Subreports

In this section we consider some subreports that are necessary to create the income statement and the balance sheet, which are then defined in the next section.

Below we give the definitions of some of the reports. We use set-comprehension notation, as it first appeared in the SETL programming language [SDDS86]. These are simplified specifications for reasons of exposition. We assume that all goods purchased and transferred into inventory are for eventual sale only, and fixed assets are written as if acquired and sold the first day of the accounting period (usually year).

We would like to emphasize that we use the term "report" here to denote any computable function on sets of events. This is in contrast to accounting system practice, where the term usually conveys an expectation that the result be rendered in some graphical format (as a printable document), and where functions computed as part of other systems (such as data warehouses and OLAP engines) may carry other designations even though they are computable functions on business events.

Figure 5 contains the mathematical definition of the subreports necessary to define the report *FIFOCost*; all of which will be described shortly. The remaining subreports can be found in Appendix A.

The *InvoicesSent* and *InvoicesReceived* reports As a basic report we need a map from identifiers to corresponding invoice information. Payment and goods/service delivery events are correlated to invoices via their identifiers that they share with their invoice.

The *InvAcq* report We call a set of resources, where each is associated with a time-stamped price and VAT valuation, *priced resources.*

The *inventory acquisitions* are the priced resources that have been transferred to internal agent *Inventory*, sorted according to their time-stamp. The identifier of an internal transmit event is used to indicate from which original purchase the price information comes.

The *GoodsSold* report The most interesting reports are *costing* because they reflect accounting decisions as to attributing a cost (valuation) to goods sold. For unique resources we can uniquely associate a valuation by looking up the price information in the invoice received for it. For non-unique resources, however, many purchases may contain the same resource. Costing is about allocating a valuation to goods sold that is normally derived from their purchase prices. There are several generally accepted methods for inventory valuation: first-in-first-out (FIFO) costing, last-in-first-out (LIFO) costing, average costing. For illustration purposes we use FIFO costing here.

	Revenue	$\triangleq \sum\{p(R) : (i,(A,R,((p,m),t))) \in \mathit{InvoicesSent}\}$
–	Cost of goods sold	$\triangleq \#1(\mathit{FIFOCost})$
=	**Contrib. margin**	$\triangleq \mathit{Revenue} - \mathit{Cost\ of\ goods\ sold}$
–	Fixed costs	$\triangleq \sum\{p(R) : (R,((p,m),t)) \in \mathit{Expenses}\}$
–	Depreciation	$\triangleq \mathit{Depreciation}$
=	**Net op. income**	$\triangleq \mathit{Contrib.\ margin} - \mathit{Fixed\ costs} - \mathit{Depreciation}$

Fig. 6. **Income Statement** The Income Statement summarizes the profits and losses of the company over a given period (hence also the British term *Profit & Loss Account*).

The goods sold in the period are the resources invoiced to another company, which means that they have been or are committed to being delivered. We assume that all such resources must be moved out of inventory in connection with the sale, and that the identifier of the move indicates which sale (invoice) it relates to.

The *FIFOCost* **report** *FIFOCost* returns the value of goods removed from inventory for sale, combined with any remaining goods that could not be found in inventory. Ordinarily the latter is always 0 for a reporting period. However, the function is general enough to handle cases where items have been sold before they have been moved into inventory.

The function *foldl* is defined as:

$$\mathit{foldl}(f, e, [x_1, x_2, ..., x_n]) = f(x_n, \ldots f(x_2, f(x_1, e)) \ldots)$$

3.5.2 Financial statements

With the basic reports of the previous section it is possible to define the income statement and the balance sheet. These are shown in Figures 6 and 7.

4 Prototyping in F#

Many details are easy to overlook without a machine-checkable and executable model. For this reason we have produced a proof-of-concept implementation in

Assets	
Fixed assets	$\triangleq \mathit{FAssetAcq} - \mathit{Depreciation}$
Current assets	$\triangleq \mathit{Inv.} + \mathit{Acc.\ rec.} + \mathit{Cash} + \mathit{equiv.}$
Inventory	$\triangleq \sum\{p(R) : (R, ((p, m), t)) \in \mathit{InvAcq}\}$ $-\#1(\mathit{FIFOCost})$
Acc. receivable	$\triangleq \sum\{p(R) : (i, (A, R, ((p, m), t))) \in \mathit{InvoicesSent}\}$ $-\sum\{a : (i, a) \in \mathit{PaymentsReceived}\}$
Cash+equiv.	$\triangleq \sum\{a : (i, a) \in \mathit{PaymentsReceived}\}$ $-\sum\{a : (i, a) \in \mathit{PaymentsMade}\}$
Total assets	$\triangleq \mathit{Fixed\ assets} + \mathit{Current\ assets}$

Liabilities and owners' equity	
Liabilities	
Acc. payable	$\triangleq \sum\{p(R) : (i, (A, R, ((p, m), t))) \in \mathit{InvoicesReceived}\}$ $-\sum\{a : (i, a) \in \mathit{PaymentsMade}\}$
VAT payable	$\triangleq \mathit{VATOutgoing} - \mathit{VATIncoming}$
Owners' eq.	$\triangleq \mathit{Total\ assets} - \mathit{Liabilities}$
Total liab.+eq.	$\triangleq \mathit{Liabilities} + \mathit{Owners'\ equity}$

Fig. 7. **The Balance Sheet** The Balance Sheet summarizes assets, liabilities, and owners' equity at a particular point in time. The balance sheet should always satisfy the fundamental invariant known as the *Accounting Equation*, which states that *Assets* = *Liabilities* + *Owners' equity.*

F# [SGC07], an ML dialect similar to O'Caml. An important point of this section is to show that once we have the rigorous formal model, the actual coding of the system is simple, and even reports that are considered complex in standard ERP systems can be implemented in a succinct way.

We start with some type definitions for modeling agent specifications and resources:

```
type company_name = string
type internal_agent_name = string
type agent_spec = {company: company_name;
                   agent: internal_agent_name option
                  }

type resource_name = string
type resource = (resource_name * float) list
```

The definitions are straightforward as one would expect. The only thing worth noticing is that we represent the elements of $Real^{\infty}$ by association lists.

We model information in invoices as:

```
type invoice_line = {no_items: float;
                     resource: resource_name;
                     price: int; (* per item in std. currency *)
                     vat: int; (* VAT per item in std. currency *)
                    }
type invoice = invoice_line list
```

This does not directly correspond to the definition of *Information*, but it contains the same information and reflects more directly the known line item structure of invoices.

Finally, we model events, log entries, and the log with the following definitions:

```
type ident = string
type log_event =
  | Transmit  of agent_spec * agent_spec * resource
  | Transform of agent_spec * resource * resource
  | Inform    of agent_spec * agent_spec * invoice
type log_entry =
  | Event of log_event * date * ident
type log = log_entry = log_entry list
```

Here we use dates (year–month–day) as timestamps and strings as identifiers.

In the proof-of-concept implementation F# doubles as the preliminary report language, i.e., the implementation of the architecture itself is in F#, and to avoid introducing a separate report language at this stage F# is also used to write reports. Although F# is quite suitable for that purpose, a complete system would most likely use a report language designed specifically for enterprise reporting rather than a general-purpose language. Reports, for now, are simply F# functions that take the log as an argument (and possibly additional context arguments).

We present two subreports from Section 3.5.1 of varying complexity: one for listing the received invoices, the other for computing the accumulated cost of inventory requisitions using the FIFO method.

Invoices Received The set of invoices received can be found by simple inspection of the log. The following function `invoices_received` runs through the log, finds the relevant `Inform` events, filters out the resources (using the function `choose_informs_where`), and builds an association list for all invoice ids:

```
let choose_informs_where f log =
  let match_trans = function
    | Event(Inform(sender, receiver, inv), d, id) ->
        f sender receiver inv d id
    | _ -> None in
  List.choose match_trans log

let invoices_received me log =
  choose_informs_where (fun sender receiver inv d id ->
    if sender.company = me && receiver.company <> me
    then Some (id, (inv, d))
    else None) log
```

The function `invoices_received` takes two arguments: the name of my company, `me`, and the log. Notice that the function ignores the internal agent specification by only looking at the `company` attribute. Analogously to the utility function `choose_informs_where` we define the utility function `choose_transmits_where` that is used in the following reporting function.

FIFO Inventory Costing To find the cost of what has been taken out of the inventory using FIFO ordering as described in Section 3.5, we define the following function `fifo`:

```
let addf key map f = Map.add key (Map.tryfind key map |> f) map

let fifo inventory time log me =
  let stuff_in_inv =
    choose_transmits_where (fun sender receiver res d id ->
      if receiver = inventory && d <= time then Some(res, d, id)
      else None) log in
  (* Assumes that log is time-sorted, thus inList is also sorted *)
  let inList = List.map (fun (r,t,id) -> lookup_price log me r id)
             stuff_in_inv in
  let price_map =
    List.fold_right (fun inv map ->  (* use fold_left for LIFO *)
      List.fold_left (fun map (name, no, price) ->
        addf name map
          (function
            | Some s -> (no, price) :: s
            | None -> [no,price])) map inv) inList Map.empty in

  let outList =
    choose_transmits_where (fun sender _ res d _ ->
      if sender = inventory && d <= time then Some res
      else None) log in
  (* Outflows *)
  let outSum = List.fold_left add_res null_resource outList in
  let price_atomar (name, total) =
    let prices = Map.find name price_map in
    let rec loop remaining ((n,p)::prices) =
      if remaining > n then (n*p) + loop (remaining - n) prices
      else remaining * p in
    loop total prices in
  let total_price = List.sumByFloat price_atomar outSum in
  total_price
```

It uses the following helper functions to lookup the prices of a resource in the corresponding invoices.

```
let find_in_invoice resource invoice =
  List.map (fun (name, no) ->
    let line = List.find (fun line -> line.resource = name) invoice
    in name, no, line.price) resource
let lookup_price log me r id =
  let match_invoice = function
    | Event(Inform(sender, receiver, lines), _, ident)
      when receiver = me && ident = id -> Some lines
    | _ -> None in
  let invoice = List.first match_invoice log |> Option.get in
  find_in_invoice r invoice
```

5 A contract-oriented event-based architecture

In this section we describe the formal semantics of a contract-oriented event-based architecture. The architecture is deliberately designed to allow any contract language to be used. We begin by providing background, proceed to describe the architecture, and afterwards give a concrete example of a contract language.

5.1 Background

The most basic form of economic interaction is that of an *exchange* of resources between agents. If we take || to be the basic composition operator, we could imagine describing an exchange by writing:

```
    transmit (X, Y, 1 apple) || transmit (Y, X, 1 USD)
```

This represents a particular *contract* between X and Y that, if and when it is *entered*, obliges `X` to transfer an apple to `Y` and `Y` to give `X` a dollar in consideration. It says very little else. It sets no time limits and it mandates no particular ordering. It does, however, say that until an apple has been transmitted and a dollar has been transmitted in consideration hereof, *obligations remain.*

If the event occurs that `X` transmits an apple to `Y`, the event should be logged, and, moreover, the state of the contract should now reflect that only one obligation remains. We say that the event *matched* (i.e., satisfied) an obligation in the contract, and the result is a *residual contract* representing the remaining obligations.

In general, it is clear that the state of a company requires representation of the processes that it is committed to following, either for contractual reasons or for non-legal reasons. We suggestively call all such processes contracts, even though they may also represent processes that have no legal significance, such as internal processes.

The portion of the contract *life cycle*, that we need to model is sketched in Figure 8. The steps in the contract life cycle are as follows:

NEGOTIATE The terms are negotiated between two or more parties (independent agents). We do not model this stage.

START The contract is *started* (i.e., entered). At this point the contract describes potential different series of resource and information transfer steps that must

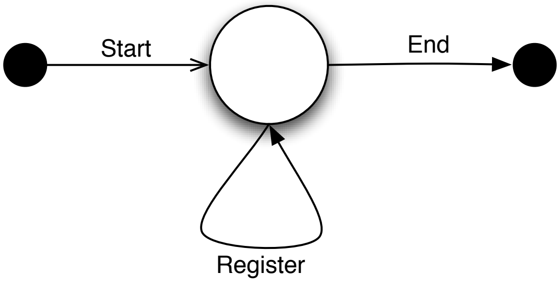


Fig. 8. State diagram showing the life cycle of contracts

happen over time

REGISTER A transmit or inform event is *matched* against a contract; that is, it is checked to see whether it is a valid step according to the contract. If it is, the event is *registered*, and the contract is updated to represent only the remaining obligations. If it is not, both the offending event and a representation of the residual obligations in the contract at that time are returned for error processing.

END The contract is *ended*. This can happen for a variety of reasons, most commonly because no obligations remain. If a breach of contract has occurred, we might choose to end it, albeit unsuccessfully; what happens thereafter is decided outside of the system.

This means that in addition to logging all transaction events (transmit, transform, and inform), we must log whenever a contract is *started* or *ended*.

All of this leads to an architecture consisting of a contract engine, a log, and a report engine.

5.2 Architecture

Figure 9 shows a birds-eye view of the architecture. We assume that there is an environment that takes care of collecting and buffering events. These are then matched, manually or automatically, with an ongoing contract. The environment can be a GUI, a workflow engine or other systems that interact with the contract engine or the report engine.

Since the report engine has been developed in a related paper [LN07], we will concentrate on a formal model of the log and the running contracts. The log, L, is a set containing elements of the type *Event*. The precise structure of the abstract representation, C, of the running contracts depends on the concrete contract language being used. Each of the running contracts can be identified uniquely via a

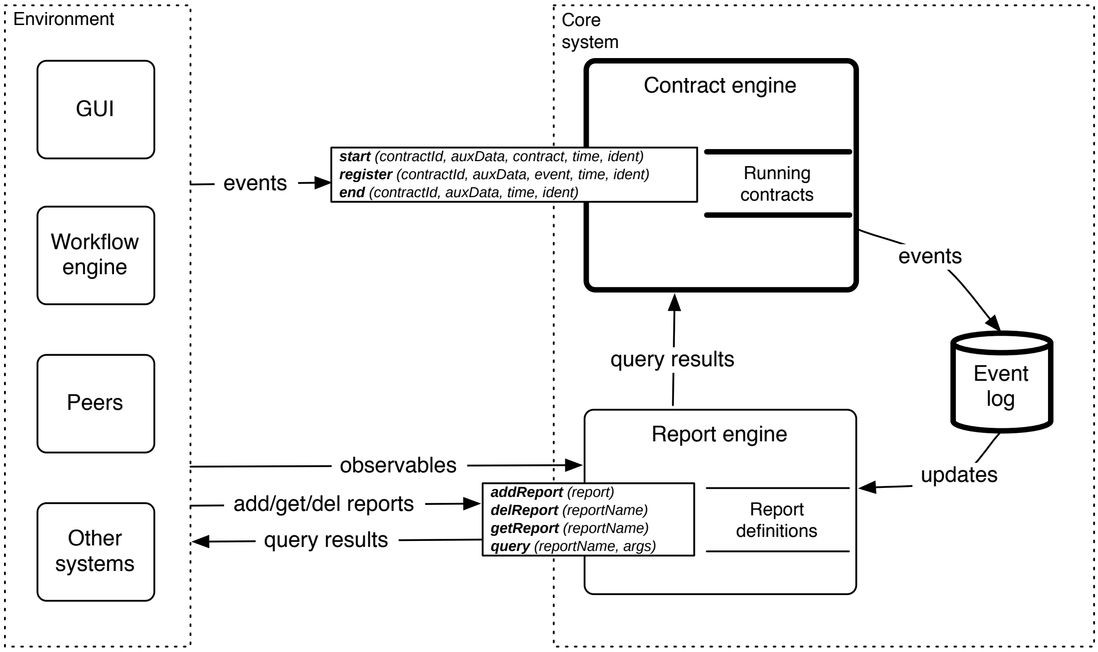


Fig. 9. Event-driven, contract-based architecture. Only the components in bold are treated in this paper; for reports readers are referred to Nissen and Larsen [NL08].

contract identifier, *cid*.

The system changes over time via a sequence of events. Given $e \in \mathit{Event}$, the transition relation on the state of the core system looks as follows

$$\langle L, C\rangle \xrightarrow{e} \langle L', C'\rangle$$

The definition of *Event* from Section 3.4 must be extended (a) to accommodate events that start and end contracts and (b) to allow for any auxiliary data that the contract language may need:

$$\begin{aligned}
\mathit{Event} &= \mathit{LogEvent} \times (\mathit{Time} \times \mathit{Ident}) \\
\mathit{LogEvent} &= (\mathit{StartEvent} \uplus \mathit{RegisterEvent} \uplus \mathit{EndEvent})
\end{aligned}$$

$$\begin{aligned}
\mathit{StartEvent} &= \texttt{Start} \ \times \mathit{ContractID} \times \mathit{AuxData} \times \mathit{Contract} \\
\mathit{RegisterEvent} &= \texttt{Register} \ \times \mathit{ContractID} \times \mathit{AuxData} \times \mathit{TransactEvent} \\
\mathit{EndEvent} &= \texttt{End} \ \times \mathit{ContractID} \times \mathit{AuxData}
\end{aligned}$$

$$\mathit{TransactEvent} = \mathit{TransmitEvent} \uplus \mathit{TransformEvent} \uplus \mathit{InformEvent}$$

$$\textsc{start}\ \frac{cid \text{ does not exist in } C \qquad C' = C \oplus (cid, x, c)}{\langle L, C\rangle \xrightarrow{\mathbf{start}(cid,x,c)@(t,id)} \langle (\mathbf{start}(cid, x, c)@(t, id)) :: L, C'\rangle}$$

$$\textsc{register}\ \frac{C \xrightarrow{(cid,x,e)} C'}{\langle L, C\rangle \xrightarrow{\mathbf{reg}(cid,x,e)@(t,id)} \langle (\mathbf{reg}(cid, x, e)@(t, id)) :: L, C'\rangle}$$

$$\textsc{end}\ \frac{C \xrightarrow{(cid,x)} C' \qquad C'' = C' \ominus cid}{\langle L, C\rangle \xrightarrow{\mathbf{end}(cid,x)@(t,id)} \langle (\mathbf{end}(cid, x)@(t, id)) :: L, C''\rangle}$$

Fig. 10. Transition relation for the core architecture

$$\begin{aligned}
\mathit{Time} &= \mathit{Real}\\
\mathit{Ident} &= \mathit{String}\\
\mathit{TransmitEvent} &= \mathit{Agent} \times \mathit{Agent} \times \mathit{Resource}\\
\mathit{TransformEvent} &= \mathit{Agent} \times \mathit{Resource} \times \mathit{Resource}\\
\mathit{InformEvent} &= \mathit{Agent} \times \mathit{Agent} \times \mathit{Information}
\end{aligned}$$

A *LogEvent* can now be either a start event, a register event or an end event. All of these three carry a contract identifier to indicate the contract being started, matched against or ended. They also carry auxiliary data, which are any data specific to the contract language. *StartEvent* additionally contains the body of the contract being inserted into the system, and *RegisterEvent* has a *TransactEvent* as part of its payload. As before all events contain a time stamp of type *Time* as well as an event identifier, *Ident*, chosen by the environment to be able to refer to the event later.

5.2.1 State transitions

We can now begin to consider the state transitions of the core architecture. The transitions are described by the inference rules displayed in Figure 10.

START The START rule inserts a new contract into the system state and logs it. $\mathbf{start}(cid, x, c)@(t, id)$ denotes a start event with contract identifier *cid*, auxiliary data x, contract c, time stamp t, and event ID *id*. The contract language-specific operation $C \oplus (cid, x, c)$ adds the contract c with identifier *cid* and auxiliary data x to the running contracts represented by C. The rule applies only if the chosen identifier, *cid*, is, indeed, previously unused.

REGISTER In this rule e is the *TransactEvent* payload of the register event. The operational semantics of the REGISTER rule relies on the operational semantics of the contract language: if the contract language permits the transition $C \overset{(cid,x,e)}{\longrightarrow} C'$, the register event is logged, and C' is the new state of the running contracts.

END The END rule removes a contract, *cid*, from the running contracts, provided that the contract language permits the transition $C \overset{(cid,x)}{\longrightarrow} C'$. The removal is written as $C \ominus cid$ where $\ominus$ is a contract language-specific operator.

5.3 An example contract language

In this section we show how to describe contracts as compositional specifications in the language of Andersen *et al.* [AEH$^+$06]:

$$\begin{aligned} c ::= & \text{Success} \mid \text{Failure} \mid f(\vec{a}) \mid c_1 + c_2 \mid c_1 \parallel c_2 \mid c_1; c_2 \\ \mid & \text{ transmit}(A_1, A_2, R, T \mid P).\, c \\ \mid & \text{ transform}(A, R_1, R_2, T \mid P).\, c \\ \mid & \text{ inform}(A_1, A_2, I, T \mid P).\, c \end{aligned}$$

Success denotes the *trivial* or *(successfully) completed* contract: it carries no obligations on anybody. Failure denotes the *inconsistent* or *failed* contract; it signifies breach of contract or a contract that is impossible to fulfill. The contract expression

$$\text{transmit}(A_1, A_2, R, T \mid P).\, c$$

represents the *commitment* transmit$(A_1, A_2, R, T \mid P)$ followed by the contract c. The commitment must be matched by a transmit event

$$e = \text{transmit}(v_1, v_2, r, t)$$

of resource r from agent v_1 to agent v_2 at time t where the predicate

$$P[A_1 \mapsto v_1, A_2 \mapsto v_2, R \mapsto r, T \mapsto t]$$

holds. If the event matches the commitment, the residual contract is c with A_1, A_2, R, T bound to v_1, v_2, r, t, respectively. In other words, A_1, A_2, R, T are binding variable occurrences whose scope is P and c. In this fashion the subsequent contractual obligations expressed by c may depend on the actual values in the event e.

The *contract combinators* $\cdot + \cdot$, $\cdot \parallel \cdot$ and $\cdot ; \cdot$ are used to express choice, parallelism, and sequence, respectively. E.g., the contract

```
     transmit (vendor, customer, Y, T | T < deadline)
||  (  transmit (customer, vendor, $100, T | T < deadline)
    + (transmit (customer, vendor, $55, T | T < deadline) ;
       transmit (customer, vendor, $55, T | T < deadline + 60 days)))
```

expresses a sale of resource Y. The customer is given a choice between paying \$100 before a given deadline (line 2) or just paying \$55 before the deadline (line 3) and then paying \$55 before 60 days after the original deadline (line 4). Both the delivery and the initial payment (whichever is chosen) must occur before the deadline, but because $\cdot \parallel \cdot$ is used, no particular order is mandated.

The language also provides facilities for defining and instantiating contract templates. The construct $f(\vec{a})$ is an instantiation of a previously defined contract template, f, with actual parameters $\vec{a}$. Contract templates definitions can be recursive, enabling us to express repetition using this construct. The mechanics of contract template definition and instantiation are outside the scope of this paper, but interested readers are referred to Andersen *et al.* [AEH$^+$06] for a complete description.

Example: sales contract A somewhat more realistic contract for simple exchanges is captured in the following contract template:

```
Sale (vendor, customer, resource, pinfo as (p, m), deadline) =
   transmit (vendor, customer, resource, T | T <= deadline) ||
   (inform (vendor, customer, (resource, pinfo), T').
     (transmit (Tax, vendor, -m(resource) DKK, _ ) ||
      transmit (customer, vendor, (p + m)(resource) DKK, T''
                | T'' <= T' + 8 days)))
```

Once instantiated with a particular vendor, a customer, a resource to be delivered, the pricing of those resources, and a deadline for delivery, the contract expresses a set of legal executions: the first transmit expresses an obligation on the vendor to deliver the resource to the customer by the given deadline. The vendor must also send an invoice to the customer, which then results in an obligation by the tax autorities to collect the VAT amount for the invoiced resources and by the customer to pay the vendor the agreed-upon price, plus VAT.

Example: internal processes The term contract is suggestive of modeling certain multi-party commitments with mutual consideration, specifying who the parties are, which resources are involved, and by when they are to be transmitted.

Formally, though, our contract specifications just specify sets of event sequences: they can also be used to structure and express *internal processes* within a company and then *monitor* their execution. We can define a *universal process* as a contract that can be matched by *any* transmit, transform or inform event, as long as the agents involved are both internal agents of the subject company:

```
UniversalProcess() =
 ( transmit (A, B, R, T | A <= Me, B <= Me) +
   inform (A, B, info, T | A <= Me, B <= Me) +
   transform (A, R1, R2, T | A <= Me)
 );
   UniversalProcess()
```

5.3.1 Routing information

Consider a variation of the contract for the sale of Y:

```
     transmit (vendor, customer, Y, T | T < deadline)
||  (  transmit (customer, vendor, $100, T | T < deadline)
    + (transmit (customer, vendor, $55, T | T < deadline) ||
       transmit (customer, vendor, $55, T | T < deadline + 60 days)))
```

Lines 3 and 4 are now conjoined using || rather than ; . If the customer transmits \$55 to the vendor before the deadline, this event can match both line 3 and line 4. Although clumsily written, the contract illustrates the need for a way to disambiguate between several possible matches. We will call such disambiguation *routing information.*

The basic idea is that all nondeterminism can be reduced to a series of routing decisions to identify the particular commitment the event is to be matched with.

We can express such a series as a sequence of routing decisions of $R = \{f, s, l, r\}$, where f (first) and s (second) indicate what choice to make when a $+$ construct is encountered, and l (left) and r (right) indicate what side of a $\|$ construct to continue on. E.g., to ensure that the early payment of \$55 is, indeed, matched to line 3, the routing information would be rsl.

5.3.2 Integration with the main architecture

With the contract language in place we can provide the remaining definitions of *ContractID*, *Contract*, and *AuxData* to integrate it with the architecture:

$$\begin{aligned} \mathit{ContractID} &= \mathit{String} \\ \mathit{Contract} &= c \\ \mathit{AuxData} &= \mathit{RoutingInformation} \\ \mathit{RoutingInformation} &= \{f, s, l, r\}^* \end{aligned}$$

Here c denotes the contract body in the syntax of the contract language.

Some contract language-specific definitions remain, namely the running contracts, C, the transition relation, $C \longrightarrow C'$, and the operators $\oplus$ and $\ominus$. These require some care to ensure that the contract language's function and variable environments are handled properly. Since these have been omitted here for the sake of simplicity, we do not delve into the details, but instead refer readers to Andersen *et al.* [AEH+06].

6 Related work

6.1 Accounting Models

Since Paccioli's Quaderno work dating back to the 15th century the dominant tradition in all financial accounting has been Double-Entry Bookkeeping (DEB) (see [WKK04] for a standard financial accounting text). However, as pointed out by McCarthy in his seminal paper introducing the Resources-Events-Agents (REA) accounting model [McC82] there are possible advantages to be reaped from other approaches. We have adopted the use of Resources, Events and Agents from REA and added our own contracts as formal, structured processes. REA is sometimes described as being in contrast with DEB. Event-based accounting does not preclude the use of the ideas presented here in conjunction with DEB, however.

In the last two decades new management accounting methods have been proposed; notably activity-based costing (ABC) [ABKY01]. A distinctive feature of our model is the separation of events and interpretation – something which is not found in DEB – and this facilitates management accounting, because one does find oneself get locked into a specific method of interpretation (say, FIFO valuation). A key aspect of our architecture is that it, accordingly, separates events representing real-world (ordinarily incontrovertible) events from interpretation (such as the various valuation method employed), which are defined as report functions. Multiple interpretations can coexist, such as tax-based depreciation and internal depreciation schemes. New interpretations can be added as report functions at any time. Conversely, a report function that is no longer of interest leaves no garbage data behind.

6.2 Contract management

Contract management is a term broadly applied to concepts, models and systems for managing contractual agreements throughout their lifecycle, from negotation through creation to execution and termination There are numerous papers that investigate organizational, system integration and, to a lesser degree, semantic aspects of contract management; see [Bou02,SLO06].

There are also a good number of commercial IT-applications that support contract management. [5]

[5] Here is a sample of contract management software in arbitrary order, without prejudice and without any claim as to completeness or representation: TotalContracts (www.procuri.com), Livelink ECM–eDOCS for Contract Management (www.opentext.com), Meridian (www.meridiansystems.com), CompleteSource Contract Management (www.moai.com), UpsideContract (www.upsidesoft.com), StatsLog4 (www.statslog.com) for construction contracts, Contraxx (www.ecteon.com), Salesforce.com (www.salesforce.com), SAP xApp Contract Lifecycle Management (www.sap.com), 8over8.com (www.8over8.com) for oil and gas contracts, IntelliContract (www.intellicontract.com), Softrax (www.softrax.com), On Demand Contract Management (www.ketera.com), Contract Assistant (www.blueridgesoftware.bz), Autotask (www.autotask.com), Contract Web (cobblestonesystems.com), Accruent cmSuite (www.accruent.com), Memba Context (www.memba.com), Emptoris Enterprise Contract Management (www.emptoris.com), Contract Advantage (www.greatminds-

Judging by their descriptions these systems are primarily aimed at supporting the reliable production of *contracts as natural-language documents* and maintaining some key information about them. They do not seem to contain an expressive, yet declarative *formal* language for user-defined contract templates, nor a theory (or tools for) correct transformation and analysis. In particular, they are generally advertised as *integrating* with ERP systems, but not, as proposed here, as *being* at their core.

6.3 Process languages

A core component of the architecture is the explicit representation of contracts or, more generally, processes that model legal/acceptable sequences of events, which are time-stamped transfers of resources and information between companies and their actual or virtual parts.

Since the seminal publications on *business process reengieering* in the early 90s [Ham90,DS90] there has been a marked shift towards a process-oriented view of the world in management science. This has naturally induced an interest in *process-aware information systems* [DvdAtH05] and enterprise process modeling [DKKS04]. A large part of this interest was devoted to various ways of expressing business processes, or—since they are commonly seen as an instance hereof—workflows. This has lead to efforts to express workflows in Petri nets [vdAHKB03], π-calculus [Ste05a,Ste05b], and a variety of other formalisms. A significant other strand of research was that of integrating processes—however they may be represented formally—with existing information systems, most saliently ERP systems. The *ARIS* framework is an important example of this [Sch00a]. Both commercial and open source ERP system (such as, SAP and *Compiere*, respectively) have introduced process concepts. As of now, however, no other ERP system has been based on processes from *first principles* to our knowledge. In other words there has been significant research of business process reengineering, workflow systems, process-aware information systems, and how to build process *on top of* ERP systems. This has lead to a consensus that processes are a useful way to think about how business operate, and a further consensus that information systems need to closely mirror the business—this is often referred to as *business/IT-alignment.* However, no publications have attempted to revise the standard ERP system architecture to directly accommodate a process-oriented view of the world as we have done here.

As said numerous formalisms exist for specifying processes, and a number of them

software.com). Please note that trademark notices have been omitted for readability.

have been applied to modeling contracts in the business domain. Timed finite state systems such as timed automata [AD94] enhance the corresponding finite state system with deadline constraints on state transitions. Careful limitation of the expressive power of the timing constraints combined with the finite-state nature enable powerful model checking techniques.

Daskalopulu demonstrates how model checking can be applied to a sales contract whose interactions are modeled as a timed Petri Net [Das00]. Molina-Jimenez *et al.* show how contracts represented as finite state machines can be monitored during execution [MJSSW03]. To ensure the finite-state property of the space of control states, their *data* components—the actual *resources* exchanged—are removed, however. Control dependency of interactions on data must be abstracted in the model when rendering it as timed finite state system. In particular, simple data-dependent protocols such as payment by installment—pay as often as necessary until the amount due is paid up as long as they all occur within a certain deadline—must be approximated in some fashion in the model. The same argument seems to apply to Event-driven Process Chains (EPC) [vdA99,Kin03] and other workflow languages with event-driven transitions on a finite set of control states. Again, at that level of modeling, they factor out the data into a separate part and treat events as atomic data with no internal structure.

An interesting recent development is the explicit declarative representation of the Deontic notions [McN06] of permissions and obligations by Pace, Priscariu, and Schneider [PS07,PPS07] for declaratively representing contractual relations. What makes this work engaging is that they demonstrate an "isomorphic" translation of contractual stipulations to the formalization in their language CL; and that CL still can be subjected to model checking.

In contrast to the above finite state systems the contract language of Andersen et al. [AEH$^+$06] employed here models completely all the relevant data uncovered in our analysis of what is relevant for reporting purposes: the actions in our state transitions are events that carry full data representations of resources and agents and constitute thus, by themselves, an infinite domain. The data part is not factored into unspecified off-process database updates with no data-dependent control state transitions. Unsurprisingly this makes the complexity of semantically faithful contract analysis hard: Equivalence with Fail, the impossible-to-satisfy contract, is NP-complete for contracts without recursion [Nis05], and it is undecidable [Nis07] with full recursion, even when restricted to a very simple predicate language with deadlines as in timed automata. We expect the paucity of the control constructs—sequential and parallel composition; recursion— however to enable practically useful analyses that include precise analysis of the resource flows. The contract language has been developed as a match for ERP systems. An empirical evaluation in that domain is future work. We expect to find the need for variations on and extensions to it, specifically support for parallel composition of contractual commitments whose interdependencies are expressed declaratively by constraints, which is why it has deliberately been designed to be a minimal core language.

It is important to observe that the contract language specifies process *types* in the sense of protocols or behavioral types, rather than executable systems. As such it is more basic than, but analogous to, the "global" Web Services Choreography Description Language (WS-CDL[6]) rather than a "local" orchestration (executable process) language such as Web Services Business Process Execution Language (WS-BPEL[7]). The global communication perspective in our contract language is motivated by and inherited from the application domain, specifically the REA accounting model (see above); it constitutes, as such, a "natural" way of formulating processes in that domain. Such a global language with a *well-defined formal semantics* enables an automatic, provably correct transformation to the (parallel) subprocesses of the individual agents (partners, roles) in a process, as has been demonstrated by Carbone, Honda and Yoshida [CHY07] for an expressive WS-CDL-like language. We believe this to be an important enabling step in generating process-specific role-based user interfaces, which are expected to be important in future ERP systems.

6.4 Event-driven architectures

An event-driven architecture is any architecture that is built on the notion of components reacting to events and generating events.

As such any *run-time monitoring/verification* system can be thought of as an event-driven (sub)system. This includes active databases [RG03, Section 5.8], security automata [Sch00b,SMH01], policy engines [GRF06], access control/resource monitors, etc., whether based on automata specifications, temporal logics such as LTL [KPA03,BLS08] or, for that matter, low-level code that implements state transitions.

What makes our architecture a *process-oriented* event-driven *architecture* is that each process (contract specification) is a denotation for a set of expected event sequences. It furthermore offers a syntax for composing and subsequently automat-

[6] `http://www.w3.org/TR/ws-cdl-10` retrieved on June 10th 2008.

[7] `http://www.oasis-open.org/committees/wsbpel` retrieved on June 10th 2008.

ically manipulating/transforming such denotations: run-time events are matched against events and transformed to represent the residual process, and residual processes can be input as data to—in principle arbitrary— analysis functions.

Complex Event Processing [LF98] also relies on the event view of the world, but is primarily intended for the purpose of monitoring events from several layers in a network by installing predicates and aggregators. In contrast, in our architecture events are matched against contracts that *prescribe* the expected arrival of events and act as run-time monitors. Simultanously their syntactic representation can be used as inputs to analyses for generating information, including new events for matching.

7 Conclusions

We have presented an event-driven architecture consisting of an event processing engine that matches economic and information events against their process specifications, and user-definable functions providing information on the state of the system. As we have seen, these functions can be specified compactly in set notation, and the specifications map closely to the actual reports in F# code.

At any given point in time the state of the system consists of the logged events and residual contracts modeling expected future events. Reports can be defined as arbitrary functions. In this fashion derived data, expressed as report functions, are strictly separated from base data, which model real-world events. Theory and technology exist for turning naively formulated functions that read the whole state each time they are executed into efficient on-line algorithms [Bri05,NL08].

The explicit representation of contracts enables defining reporting functions, ranging from useful to-do lists to, as demonstrated by Peyton-Jones and Eber [JE03], sophisticated valuations. By formulating analyses for all possible (specifiable) contracts, it is possible to break the binding time dependencies that normally require that a process be coded up first before a corresponding set of specific reports can be (hand-)coded for it. Notably, we believe that role-specific user interfaces can be generated from process specifications that always reflect the present state of a process, even if changed as the result of case (process instance) specific changes at run-time.

Formal contract specifications thus build the core of a *process-oriented event-driven architecture*: Contracts function as behavioral types for event traces. In the architecture an event is matched against a user-specified contract specification to validate the contractual validity of the event and compute the residual obligations as an explicit contract specification in its own. Contract specifications cannot only used in this passive fashion of matching and flagging errors, but are likely to be most useful in enabling functions to be formulated on them, ranging from to-do lists via automatically generated user interfaces to sophisticated stochastic analyses for valuation purposes.

Acknowledgments

The above reflects ongoing work within the 3rd generation Enterprise Resource Planning Systems Project (3gERP.org), a collaboration between Copenhagen Business School, University of Copenhagen and Microsoft Development Center Copenhagen made possible by a grant by the Danish National Advanced Technology Foundation.

The section on contracts is excerpted from Andersen, Elsborg, Henglein, Jakobsen, Stefansen [AEH+06]; Figure 9 is from Stefansen [Ste08]; both with permission by the authors.

References

[ABKY01] Anthony A. Atkinson, Rajiv D. Banker, Robert S. Kaplan, and S. Mark Young. *Management Accounting.* Prentice Hall, third internation edition edition, 2001.

[AD94] Rajeev Alur and David Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[AEH$^+$06] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):485–516, November 2006.

[BLS08] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly—but how ugly is ugly? Technical Report TUM-I0803, Institut für Informatik, Technische Universität München, February 2008.

[Bou02] Abdel Boulmakoul. Integrated contract management. Technical report, Hewlett-Packard Laboratories

Bristol, July 2002. http://www.hpl.hp.com/techreports/2002/HPL-2002-183.pdf.

[Bri05] Daniel Brixen. Incremental methods for REA-based reporting. M.S. thesis, Department of Computer Science, University of Copenhagen, January 2005. In Danish.

[CHY07] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *Proc. 16th European Symposium on Programming (ESOP), Braga, Portugal*, volume 4421 of *Lecture Notes in Computer Science (LNCS)*, pages 2–17. Springer, 2007.

[Cre75] Michael A. Crew. *Theory of the Firm.* Longman, 1975.

[Das00] Aspassia Daskalopulu. Model checking contractual protocols. In Winkels Breuker, Leenes, editor, *13th Annual Conference on Legal Knowledge and Information Systems (JURIX)*, Frontiers in Artificial Intelligence and Applications, pages 35–47. IOS Press, 2000.

[DKKS04] Nikunj P. Dalal, Manjunath Kamath, William J. Kolarik, and Eswar Sivaraman. Toward an integrated framework for modeling enterprise processes. *Commun. ACM*, 47(3):83–87, 2004.

[DS90] Th. H. Davenport and J. E. Short. The new industrial engineering: Information technology and business process reengineering. *Sloan Management Review*, 31(4), 1990.

[DvdAtH05] Marlon Dumas, Wil M. van der Aalst, and Arthur H. ter Hofstede. *Process Aware Information Systems: Bridging People and Software Through Process Technology.* Wiley-Interscience, 2005.

[GRF06] Pedro Gama, Carlos Ribeiro, and Paulo Ferreira. A scalable history-based policy engine. In *7th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 100–112. IEEE Computer Society, 2006.

[Ham90] M. Hammer. Reengineering work: Don't automate, obliterate. *Harvard Business Review*, 68(4):104ff, 1990.

[JE03] Simon Peyton Jones and Jean-Marc Eber. How to write a financial contract. In Gibbons and de Moor, editors, *The Fun of Programming.* Palgrave Macmillan, 2003.

[Kin03] E. Kindler. On the semantics of EPCs: A framework for resolving the vicious circle. Technical report, Reihe Informatik, University of Paderborn, Paderborn, Germany, August 2003.

[KPA03] Kåre J. Kristoffersen, Christian Pedersen, and Henrik R. Andersen. Runtime verification of timed ltl using disjunctive normalized equation systems. In *Proc. 3d Workshop on Runtime Verification (RV), Boulder, Colorado*, volume Electronic Notes in Theoretical Computer Science, vol. 9, no. 2, 2003.

[LF98] David C. Luckham and Brian Frasca. Complex event processing in distributed systems. Technical report, Computer Systems Lab, Stanford University, August 1998.

[LN07] Ken Friis Larsen and Michael Nissen. FunSETL—functional reporting for ERP systems. In *Proc. 19th International Symposium on Implementation and Application of Functional Languages (IFL)*, Freiburg, Germany, October 2007.

[McC82] William E. McCarthy. The REA accounting model: A generalized framework for accounting systems in a shared data environment. *The Accounting Review*, LVII(3):554–578, July 1982.

[McN06] Paul McNamara. Deontic logic. Stanford Encyclopedia of Philosophy, 2006.

[MJSSW03] Carlos Molina-Jimenez, Santosh Shrivastava, Ellis Solaiman, and John Warne. Contract representation for run-time monitoring and enforcement. In *Proc. 2003 IEEE International Conference on E-Commerce Technology (CEC*, page 103 ff., 2003. Technical Report CS-TR-810, School of Computing Science, University of Newcastle upon Tyne.

[Nis05] Michael Nissen. B.S. thesis, January 2005. Department of Computer Science, University of Copenhagen (DIKU).

[Nis07] Michael Nissen. Contract analysis. TOPPS D-Report D-564, Department of Computer Science, University of Copenhagen (DIKU), June 2007.

[NL08] Michael Nissen and Ken Friis Larsen. FunSETL—functional reporting for ERP systems. In *Proceedings of The Ninth Symposium on Trends in Functional Programming (TFP)*, The Netherlands, May 2008.

[PPS07] Gordon Pace, Cristian Prisacariu, and Gerardo Schneider. Model checking contracts –a case study. In *Proc. 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, volume 4762 of *Lecture Notes in Computer Science (LNCS)*, pages 82–97, Tokyo, Japan, October 2007. Springer-Verlag.

[PS07] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In *Proc. 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *Lecture Notes in Computer Science (LNCS)*, pages 174–189, Paphos, Cyprus, June 2007. Springer.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 3d edition, 2003.

[Sch00a] August-Wilhelm Scheer. *Aris-Business Process Modeling*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.

[Sch00b] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[SDDS86] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[SGC07] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F#*. Apress, 2007.

[SLO06] Jan W. Schemm, Christine Legner, and Hubert Österle. E-contracting: Towards electronic collaboration processes in contract management. In Franz Lehner, Holger Nösekabel, and Peter Kleinschmidt, editors, *Proc. Multikonferenz Wirtschaftsinformatik 2006*, volume Tagungsband 1, pages 255–272, 2006. http://www.gbv.de/dms/tib-ub-hannover/508826985.pdf.

[SMH01] Fred Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In *Informatics—10 years back, 10 years ahead*, volume 2000 of *Lecture Notes in Computer Science (LNCS)*, pages 86–101. Springer, 2001.

[Ste05a] Christian Stefansen. SMAWL: A SMAll Workflow Language based on CCS. Technical Report TR-06-05, Harvard University, Division of Engineering and Applied Sciences, Cambridge, MA 02138, March 2005.

[Ste05b] Christian Stefansen. SMAWL: A SMAll Workflow Language based on CCS. In *Proceedings of the CAiSE Forum of the 17h Conference on Advanced Information Systems Engineering*, June 2005.

[Ste08] Christian Stefansen. *(I) A Declarative Framework for ERP Systems; (II) Reactors: A Data-Driven Programming Model for Distributed Applications.* PhD thesis, Department of Computer Science, University of Copenhagen, June 2008.

[vdA99] W.M.P. van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639–650, July 1999.

[vdAHKB03] W. M. P. van der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[WKK04] Jerry J. Weygandt, Donald E. Kieso, and Paul D. Kimmel. *Financial Accounting, with Annual Report*. Wiley, 2004.

$$\begin{aligned}
\mathit{PaymentsReceived} &= \{(i,\, \mathit{USD}(R)) \\
&\qquad : (\mathit{transmit}(A,B,R),t,i) \in \mathit{Events} \mid B \leq \mathit{me}\} \\
\mathit{CashReceived} &= \sum\{k : (i,k) \in \mathit{PaymentsReceived}\} \\
\mathit{PaymentsMade} &= \{(i,\, \mathit{USD}(R)) \\
&\qquad : (\mathit{transmit}(A,B,R),t,i) \in \mathit{Events} \mid A \leq \mathit{me}\} \\
\mathit{CashPaid} &= \sum\{k : (i,k) \in \mathit{PaymentsMade}\} \\
\mathit{NetCashFlow} &= \mathit{CashReceived} - \mathit{CashPaid}
\end{aligned}$$

$$\begin{aligned}
\mathit{FAssetAcq} &= \mathit{Sort}\{(R, (\mathit{price}, t)) \\
&\quad : (\mathit{transmit}(\mathit{me}, \mathit{me.FixedAssets}, R), t, i) \in \mathit{Events}, \\
&\quad\ (A, R', \mathit{price}) = \mathit{InvoicesReceived}(i)\} \\
\mathit{Expenses} &= \mathit{Sort}\{(R, (\mathit{price}, t)) \\
&\quad : (\mathit{transmit}(\mathit{me}, \mathit{me.Expenses}, R), t, i) \in \mathit{Events}, \\
&\quad\ (A, R', \mathit{price}) = \mathit{InvoicesReceived}(i)\}
\end{aligned}$$

$$\begin{aligned}
\mathit{VATOutgoing} &= \textstyle\sum\{m(R) : (i, (A, R, ((p, m), t))) \in \mathit{InvoicesSent}\} \\
\mathit{VATIncoming} &= \textstyle\sum\{m(R) : (i, (A, R, ((p, m), t))) \in \mathit{InvoicesReceived}\}
\end{aligned}$$

Fig. A.1. Definitions of more subreports

A Subreport descriptions

Depreciation Another interesting aspect is depreciation of fixed assets, again because it reflects certain accounting assumptions, whether mandated by law or to reflect realistic wear-and-tear or resale considerations. For example, in the *straight line depreciation method* the depreciation per time is the same over the useful lifetime of an asset; in other words, the value of a resource is a linear function of time over its depreciation period, thereafter it is 0. The *declining balance depreciation method* writes down value of priced resources by a certain percentage after each equally long period, with special write-down to 0 once the value has become sufficiently small; in other words, it is a discrete (periodicized) approximation to an exponential decay function.

We can model such depreciation as report functions or even higher-order functions on valuations. Given a priced resource $(R, ((p, m), t))$, and a depreciation period d, valuation $p_{t'}$ of R at time t' according to the straight line method such that $t \leq t' \leq t + d$ is $\frac{t'-t}{d}p$. The declining balance method can also be modeled as a time-dependent scaling of a resource's purchase valuation, with the notable exception of the final write-down to zero, since that step does not distribute over set union of priced resources. This can be handled by associating the write-down-to-0 threshold with the internal agent containing the resources, such as *FixedAssets*: The value of the priced resources at time t is computed according the depreciation formula in use (linear or exponential), and then the total value is compared to the threshold value associated with the internal agent whose resources are being value. If it is above the threshold value, its value is returned; otherwise 0 is returned.

The *NetCashFlow* report Cash-flow can be simply given by gathering up all money flows in the events.

Here *USD* is the base vector (as a linear function) for a currency resource. Multi-currency cash flow can be performed by taking the sum of all such base vectors; for instance $\mathit{USD} + \mathit{Euro} + \mathit{DKK}$.[8]

The *FAssetAcq* and *Expenses* reports Analogously to the *inventory acquisitions* we define *fixed asset acquisitions* and *operational expenses*. Note that the internal agent *Expenses* stands for resources that are consumed instantaneously.

VATOutgoing* and *VATIncoming The incoming and outgoing VAT amounts are computed from the invoices by employing their VAT valuation components.

[8] Recall that currencies such as USD, Euro, DKK are treated as regular resource names.

Compositional Specification of Commercial Contracts

Jesper Andersen [a] Ebbe Elsborg [b] Fritz Henglein [a]
Jakob Grue Simonsen [a] Christian Stefansen [a]

[a] *Department of Computer Science, University of Copenhagen (DIKU), Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark*

[b] *The IT University of Copenhagen (ITU), Rued Langgards Vej 7, DK-2300 Copenhagen S, Denmark*

Abstract

We present a declarative language for compositional specification of contracts governing the exchange of resources. It extends Eber and Peyton Jones's declarative language for specifying financial contracts [JE03] to the exchange of money, goods and services amongst multiple parties, and it complements McCarthy's Resources, Events and Agents (REA) accounting model [McC82] with a view-independent formal contract model that supports definition of user-defined contracts, automatic monitoring under execution, and user-definable analysis of their state before, during and after execution. We provide several realistic examples of commercial contracts and their analyses. A variety of (real) contracts can be expressed in such a fashion as to support their integration, management and analysis in an operational environment that registers events.

The language design is driven by both domain considerations and semantic language design methods: A contract denotes a set of traces of events, each of which is an alternative way of concluding the contract successfully, which gives rise to a CSP-style [BHR84,Hoa85] denotational semantics. The denotational semantics drives the development of a sound and complete small-step operational semantics, where a partially executed contract is represented as a (full) contract that represents the remaining contractual commitments. This operational semantics is then systematically refined in two stages to an instrumented operational semantics that reflects the bookkeeping practice of identifying the specific contractual commitment a particular event matches at the time the event occurs, as opposed to delaying this matching until the contract is concluded.

1 Introduction

When entrepreneurs enter contractual relationships with a large number of other parties, each with possible variations on standard contracts, they are confronted with the interconnected problems of *specifying* contracts, *monitoring* their execution for performance[1], *analyzing* their ramifications for planning, pricing and other purposes prior to and during execution, and *integrating* this information with accounting, workflow management, supply chain management, production planning, tax reporting, decision support *etc.*

1.1 Contract Management and Information Systems

Judging by publically available information, support for contracts in most present-day enterprise resource planning (ERP) systems is delegated to *functional silos*, specialized (sub)systems supporting a fixed catalogue of predefined contracts for specific application domains; e.g. creditor/debitor modules in ERP systems such as Microsoft Business Solutions' Navision 3.60 and Axapta [nav] for simple commercial contracts, SAP's specialized contract management subsystems for particular industries such as the beverage industry [sap], or independent systems for managing portfolios of financial contracts such as Simcorp's IT/2 system for managing treasuries [sim]. Common to these systems seems to be that they support a fixed and limited set of contract templates specialized to a particular application domain and lack flexible integration with other (parts of) enterprise systems. A notable exception is LexiFi [lex] whose products for complex financial derivatives incorporate some of the ideas pioneered in Peyton Jones, Eber, and Seward's research in financial engineering [JES00,JE03].

In the absence of support for user-definable (custom) contracts users are forced to adhere to stringent business processes or end up engaging in "off-book" activities, which are not easily tracked or integrated; e.g. oral or written contracts in natural language. Furthermore, development of new specialized contract modules incurs considerable development costs with little possibility for supporting efficient division of labor in a multi-stage development model where a software *vendor* produces a solution *framework*, *partners* with domain expertise *specialize (instantiate)* the framework to particular industries, and *customers* (individual companies) *configure* and *deploy* specialized systems for their *end users*.

[1] *Performance* in contract lingo refers to *compliance* with the *promises* (contractual commitments) stipulated in a contract; nonperformance is also termed *breach of contract*.

1.2 Problems with Informal Contract Management

Typical problems that can arise in connection with informal modeling and representation of contracts and their execution include the following:

(1) Disagreement on what a contract actually requires. Many contract disputes involve a disagreement between the parties about what the contract requires, and many rules of contract law pertain to interpretation of terms of a contract that are vague or ambiguous.
(2) Agreement on contract, but disagreement on what events have actually happened (event history); e.g. buyer of goods claims that payment has been made, but seller claims not to have received it ("check is in the mail" phenomenon).

(3) Agreement on contract and event history, but disagreement on remaining contractual obligations; e.g., seller applies payment by buyer to one of several commitments buyer has, but buyer intends it for another commitment.
(4) Breach or malexecution of contract: A party overlooks a deadline on a commitment and is in breach of contract (missed payment deadline) or incurs losses (deadline on lucrative put or call option overlooked).
(5) Entering bad or undesirable contracts/missed opportunities; e.g., a company enters a contract or refrains from doing so because it cannot quickly analyze its value and risk.
(6) Coordination of contractual obligations with production planning and supply chain management; e.g., company enters into an otherwise lucrative contract, but overlooks that it does not have the requisite production capacity due to other, preexisting contractual obligations.
(7) Impossibility, slowness or costliness in evaluating state of company affairs; e.g., bad business developments are detected late, or high due diligence costs affect chances and price of selling company.

Anecdotal evidence suggests that costs associated with these problems can be considerable. Eber estimates that a major French investment bank has costs of about 50 mio. Euro per year attributable to 1 and 4 above, with about half due to legal costs in connection with contract disputes and the other half due to malexecution of financial contracts [Ebe02].

In summary, capturing contractual obligations precisely and managing them conscientiously is important for a company's planning, evaluation, and reporting to management, shareholders, tax authorities, regulatory bodies, potential buyers, and others.

ERP systems used today capture the activities of an enterprise based on the principles of double-entry bookkeeping. Since the integration of this with subsystems for handling contract execution is characterized by *ad hoc*, makeshift solutions, it is interesting to consider if a specification language can be designed and integrated with the data model in which historic activities of the enterprise are collected. We argue that a declarative *domain-specific (specification) language (DSL)* for compositional specification of commercial contracts (defining contracts by combining subcontracts in various, well-defined ways) with an associated precise *operational semantics* is ideally suited to alleviating the above problems. [2]

Note that contracts are not only put to a single use as programs are, whose sole use usually consists of *execution.* They are subjected to monitoring, which can be considered to be the standard semantics for contracts, plus various user-defined *analyses.*

In this sense contract specifications are more like *intelligent data* that are subjected to various uses. This is in contrast to *programs* that are exclusively executed.

As a consequence, both the syntactic structure of contract specifications and the ability of limiting their expressive (programming) power are of particular significance in their design.

We believe the DSL facilitates multi-stage development as the central interface between framework developer and partner:

(1) The *framework developer* provides the DSL, which allows specification of an infinity of contracts in a domain-oriented fashion, but without (too much) prejudice towards specific industries; delivers a run-time environment for managing execution of all definable contracts; and provides a number of useful general-purpose standard contracts. Furthermore, the framework developer provides a language (or library) and run-time system for defining contract analyses, and defines a number of standard analyses applicable to all definable contracts; e.g., next-point-of-interest computation for alerting users – human or computer – to commitments that require action (sending payment, making deliveries) or computation of accounts receivable and accounts payable for

[2] Please note that our language is rendered in ordinary linear syntax, but we do not intend to limit the scope of the term 'language' to specifying linear sequences of characters only, but to include graphical objects and the like.

financial reporting.

(2) The *partner* defines a collection of contract templates using the DSL for use in a particular industry and adds relevant industry-specific analyses using the vendor's analysis language. No general-purpose low-level programming expertise is required, but primarily domain knowledge and the ability to formalize it in the DSL and to express specialized analysis functions in the vendor's analysis language. The partner may leave some aspects (parameters) of the specialized system open for final configuration at the end user company.

(3) The *customer organization* receives its system from the partner and configures and deploys it for use by its end users.

Note that the DSL provides encapsulation and division of labor in this pipeline: Discussions between end users and partners are performed in terms of domain concepts close to the DSL, but the end user does not need to know the DSL itself. Discussions between partners and the framework provider on design, functionality, limitations are in terms of the design and semantics of the DSL, not in terms of its underlying (general-purpose) implementation language; in particular, specific implementation choices by the framework developer are unobservable by the partners. The DSL encapsulates its implementation and thus facilitates upgrading of software throughout the pipeline.

1.4 Contributions

We make the following contributions in this article:

- We define a contract language for multi-party commercial contracts with iteration and first-order recursion. They involve explicit agents and transfers of arbitrary resources (money, goods and services, or even pieces of information), not only currencies. Our contract language is stratified into a pluggable base language for atomic contracts (commitments) and a combinator language for composing commitments into structured contracts.
- We provide a natural contract semantics based on an inductive definition for when a trace—a finite sequence of events—constitutes a successful ("performing") completion of a contract. This induces a trace-based denotational semantics, which compositionally maps contracts to trace sets.
- We systematically develop three operational semantics in a stepwise fashion, starting from the denotational semantics:

(1) A (sound and complete) reduction semantics for monitoring contract execution during arrival of events. It represents the residual obligations of a contract after

an event as a *bona fide* (full) contract specification and defers matching of events to specific commitments until the whole contract has completed. It can be implemented by backtracking where events are tentatively matched to the first suitable commitment and backtracking is performed if that choice turns out to be wrong later on.

(2) A nondeterministic reduction semantics for *eager matching*, where matching decisions are made as events arrive and cannot be backtracked. Eager matching corresponds to bookkeeping practice, but leads to nondeterminacy in the case multiple commitments in a contract can be matched by the same event; in particular, the parties to a contract may perform different matches and may end up disagreeing on the contract's residual obligations.

(3) An instrumentation of the eager matching semantics that equips events with explicit control information that *routes* the event unambiguously to the particular commitment it is to be matched with. This yields an eager matching semantics with a deterministic reduction semantics and thus ensures that all parties to a contract agree on the residual contract if they agree on the prior contract state and on which event (including its routing information) has happened.

- We validate applicability of our language by encoding a variety of existing contracts in it, and illustrate analyzability of contracts by providing examples of compositional analysis.

The denotational semantics has been an instrumental methodological tool in deriving a small-step semantics.

Our work builds on a previous language design by Andersen and Elsborg [AE03] and is inspired by:

- Peyton-Jones and Eber's language for compositional specification of financial contracts [JES00], which has been the original impetus for the language design approach we have taken;
- McCarthy's Resources-Events-Agents (REA) accounting model [McC82], which has provided the ontological justification for modeling commercial contracts as being built from atomic commitments stipulating transfers (economic *events*) of scarce *resources* between between *agents* (and nothing else);
- Hoare's Calculus of Sequential Processes (CSP), specifically its view-independent event synchronization model, and its associated trace theoretic semantics [BHR84,Hoa85].

See Section 7 for a more detailed comparison with this and other related work.

2 Modeling Commercial Contracts

A *contract* is an agreement between two or more parties which creates obligations to do or not do the specific things that are the subject of that agreement. A *commercial contract* is a contract whose subject is the exchange of scarce *resources* (money, goods, and services). Examples of commercial contracts are sales orders, service agreements, and rental agreements. Adopting terminology from the REA accounting model [McC82] we shall also call obligations *commitments* and parties *agents*.

It is worth noticing that contracts may be *express* or *implied*. When two parties decide to exchange goods, more often than not there is no express contract. There is, however, an implied contract of the form of "Party A expects to pay X in exchange for party B's provision of goods Y". Usually when no express contract is present, the contractual obligations are taken from common practice, general terms of trade, or legislation. Thus the term *contract* should be understood in a broader sense as a structure that governs any trade or production even if it is not verbal.

2.1 Contract Patterns

In its simplest form a contract commits two contract parties to an exchange of resources such as goods for money or services for money; that is to a pair of *transfers* of resources from one party to the other, where one transfer is in *consideration* of the other.

The sales order *template* in Figure 1 commits the two parties (`seller`, `buyer`) to a pair of transfers, of `goods` from `seller` to `buyer` and of `money` from `buyer` to `seller`. Note that both commitments are predicated on when they must be satisfied: `seller` *may* deliver *any time*, but *must* do so by a given `date`, and `buyer` *must* pay at the time delivery happens. We can think of the sales order as being *composed sequentially* of two *atomic contracts*: the `seller`'s commitment to deliver goods, followed by the `buyer`'s commitment to pay for them. If goods are not delivered there is no commitment by `buyer` to pay anything, and only `seller` is in breach of contract. In a barter (goods for goods or goods for services) the commitments on each party may be *composed concurrently*; that is, both commitments are unconditional and must be satisfied independently of each other. If no party delivers on time and no explicit provision for this is made in the contract, *both* parties may be in breach of contract. Many commercial contracts are of this simple *quid-pro-quo* kind, but far from all. Consider the legal services agreement template in Figure 2.

Fig. 1 Agreement to Sell Goods

Section 1. (Sale of goods) Seller shall sell and deliver to buyer (description of goods) no later than (date).

Section 2. (Consideration) In consideration hereof, buyer shall pay (amount in dollars) in cash on delivery at the place where the goods are received by buyer.

Section 3. (Right of inspection) Buyer shall have the right to inspect the goods on arrival and, within (days) business days after delivery, buyer must give notice (detailed-claim) to seller of any claim for damages on goods.

Fig. 2 Agreement to Provide Legal Services

Section 1. The attorney shall provide, on a non-exclusive basis, legal services up to (n) hours per month, and furthermore provide services in excess of (n) hours upon agreement.

Section 2. In consideration hereof, the company shall pay a monthly fee of (amount in dollars) before the 8th day of the following month and (rate) per hour for any services in excess of (n) hours 40 days after the receival of an invoice.

Section 3. This contract is valid 1/1-12/31, 2004.

Here commitments for rendering of a monthly legal service are *repeated*, and each monthly service consists of a standard service part and an *optional* service part. More generally, a contract may allow for *alternative* executions, any one of which satisfies the given contract.

We can discern the following basic *contract patterns* for composing commercial contracts from subcontracts (a subcontract is a contract used as part of another contract):

- a *commitment* stipulates the transfer of a resource or set of resources between two parties; it constitutes an *atomic contract*;
- a contract may require *sequential* execution of subcontracts;
- a contract may require *concurrent* execution of subcontracts, that is execution of all subcontracts, where individual commitments may be interleaved in arbitrary order;
- a contract may require execution of one of a number of *alternative* subcontracts;
- a contract may require *repeated* execution of a subcontract.

Furthermore, commitments and, more generally, contracts usually carry *temporal constraints*, which stipulate when the actual resource transfers must happen.

In the remainder of this report we shall explore a declarative contract specification language based on these contract patterns.

3 Compositional Contract Language

In this section we present a core contract specification language and its properties. All proofs are relegated to Appendix A.

The language should satisfy the following design criteria:

- Contracts should be specifiable compositionally, reflecting the contract composition patterns of Section 2.1.
- The language should separate contract composition (contract language) from definition of the atomic commitments (base language), including their temporal constraints; this is to make sure that the design can accommodate changes and extensions to the base language without simultaneously forcing substantial changes in the contract language.
- The language should obey good language design principles such as naming and parameterization, orthogonality and compositional semantics.
- The language should be expressive enough to represent partially executed contracts as (full) contracts and have a reduction semantics that reduces a contract under arrival of an event to a contract that represents the residual obligations. By representing partially executed contracts as contracts any contract analysis will also be applicable to partially executed contracts.
- The reduction semantics should be a good basis for 'control' of execution; in particular, for *matching* of events against the specific (intended) commitment in a contract that it satisfies.

3.1 Syntax

Our contract language $\mathcal{C}^{\mathcal{P}}$ is defined inductively by the inference system for deriving judgements of the forms $\Gamma; \Delta \vdash c : \text{Contract}$ and $\Delta \vdash D : \Gamma$. Here Γ and Δ range over maps from identifiers to *contract template types* and to *base types*, respectively. The *map extension operator* on maps is defined as follows:

$$(m \oplus m')(x) = \begin{cases} m'(x) \text{ if } x \in \text{domain}(m') \\ m(x) \;\; \text{otherwise} \end{cases}$$

The language is built on top of a a *base structure* of domains $(\mathcal{A}, \mathcal{R}, \mathcal{T})$ of *agents*, *resources*, *time* where $(\mathcal{T}, \leq_{\mathcal{T}})$ is totally ordered. It consists of a typed *base lan-*

guage of expressions $\mathcal{P}$, for which we assume the existence of a set of valid typing judgements $\Delta \vdash a : \tau$ for expressions a, which include variables X and constants for each element in the base structure. Types τ include $\text{Agent}, \text{Resource}, \text{Time}$, which denote $(\mathcal{A}, \mathcal{R}, \mathcal{T})$, respectively, as well as Boolean for predicates (Boolean expressions). The expression language has a notion of substitution $b[\vec{a}/\vec{X}]$ [3] and a denotation function $\mathcal{Q}[\![\Delta \vdash a : \tau]\!]$ that maps valid typing judgements to elements of domains $Dom[\![\Delta \to \tau]\!]$. (See Figure 6 for a brief description of the thus denoted domains.) The only properties we shall assume are that substitution is compatible with judgements: if $\Delta \oplus \vec{X} : \vec{\tau} \vdash b : \tau_b$ and $\Delta \vdash \vec{a} : \vec{\tau}$ then $\Delta \vdash b[\vec{a}/\vec{X}] : \tau_b$ where $\vec{a} = a_1 \dots a_n$ and $\vec{X} = X_1 \dots X_n$ for some $n \geq 0$; and that the denotation function is compositional; that is, $\mathcal{Q}[\![\Delta \vdash b[\vec{a}/\vec{X}] : \tau]\!]^\delta = \mathcal{Q}[\![\Delta \vdash b : \tau]\!]^{\delta \oplus \{X_i \mapsto \mathcal{Q}[\![\Delta \vdash a_i : \tau_i]\!]^\delta\}_i}$.

We use metavariable P for Boolean expressions and abbreviate $\Delta \vdash P : Bool$ to $\Delta \vdash P$. For brevity and readability, we also abbreviate $\mathcal{Q}[\![\Delta \vdash a : \tau]\!]$ to $\mathcal{Q}[\![a]\!]$, leaving Δ and τ to be understood from the context. Finally, we write $\delta \models P$ for $\mathcal{Q}[\![P]\!]^\delta = \text{true}$.

The language $\mathcal{P}$ provides the possibility of referring to *observables* [JES00,JE03]. We shall introduce suitable base language expressions on an *ad hoc* basis in our examples for illustrative purposes.

The context-free structure of contracts directly reflects the contract patterns we discussed in Section 2.1:

$$\begin{aligned} c ::= {} & \text{Success} \mid \text{Failure} \mid f(\vec{a}) \mid \text{transmit}(A_1, A_2, R, T \mid P).\, c \\ \mid {} & c_1 + c_2 \mid c_1 \parallel c_2 \mid c_1 ; c_2 \end{aligned}$$

Success denotes the *trivial* or *(successfully) completed* contract: it carries no obligations on anybody. Failure denotes the *inconsistent* or *failed* contract; it signifies breach of contract or a contract that is impossible to fulfill. The environment $\mathrm{D} = \{f_i[\vec{X_i}] = c_i\}_{i=1}^m$ contains named *contract templates* where $\vec{X_i}$ is a vector of formal parameters for use in the embedded contract c_i. A contract template needs to be instantiated with actual arguments from the base language. (The n_i on the τ indicates that different contracts may have a different number of formal parameters.) For a Boolean predicate P the contract expression $\text{transmit}(A_1, A_2, R, T \mid P).\, c$ represents a contract where the *commitment* $\text{transmit}(A_1, A_2, R, T \mid P)$ must be satisfied first. Note that A_1, A_2, R, T are binding variable occurrences whose scope

[3] We use the general convention that metavariables in boldface denote vectors (sequences) of what the metavariable denotes.

Fig. 3 Syntax for contract specifications

$$\Gamma;\Delta \vdash \text{Success} : \text{Contract} \qquad \Gamma;\Delta \vdash \text{Failure} : \text{Contract}$$

$$\frac{\Gamma(f) = \vec{\tau} \rightarrow \text{Contract} \quad \Delta \vdash \vec{a} : \vec{\tau}}{\Gamma;\Delta \vdash f(\vec{a}) : \text{Contract}}$$

$$\frac{\begin{array}{c}\Delta' = \Delta \oplus \{A_1 : \text{Agent}, A_2 : \text{Agent}, R : \text{Resource}, T : \text{Time}\} \\ \Gamma;\Delta' \vdash c : \text{Contract} \\ \Delta' \vdash P : \text{Boolean}\end{array}}{\Gamma;\Delta \vdash \text{transmit}(A_1, A_2, R, T \mid P).\, c : \text{Contract}}$$

$$\frac{\Gamma;\Delta \vdash c_1 : \text{Contract} \quad \Gamma;\Delta \vdash c_2 : \text{Contract}}{\Gamma;\Delta \vdash c_1 + c_2 : \text{Contract}}$$

$$\frac{\Gamma;\Delta \vdash c_1 : \text{Contract} \quad \Gamma;\Delta \vdash c_2 : \text{Contract}}{\Gamma;\Delta \vdash c_1 \parallel c_2 : \text{Contract}}$$

$$\frac{\Gamma;\Delta \vdash c_1 : \text{Contract} \quad \Gamma;\Delta \vdash c_2 : \text{Contract}}{\Gamma;\Delta \vdash c_1 ; c_2 : \text{Contract}}$$

$$\frac{\begin{array}{c}\Gamma = \{f_i \mapsto \tau_{i1} \times \ldots \times \tau_{in_i} \rightarrow \text{Contract}\}_{i=1}^{m} \\ \Gamma;\Delta \oplus \{X_{i1} : \tau_{i1}, \ldots, X_{in_i} : \tau_{in_i}\} \vdash c_i : \text{Contract}\end{array}}{\Delta \vdash \{f_i[\vec{X_i}] = c_i\}_{i=1}^{m} : \Gamma}$$

$$\frac{\Delta \vdash \{f_i[\vec{X_i}] = c_i\}_{i=1}^{m} : \Gamma \quad \Gamma;\Delta \vdash c : \text{Contract}}{\Delta \vdash \text{letrec}\, \{f_i[\vec{X_i}] = c_i\}_{i=1}^{m} \text{ in } c : \text{Contract}}$$

is P and c. The commitment must be *matched* by a *(transfer) event*

$$e = \text{transmit}(v_1, v_2, r, t)$$

of resource r from agent v_1 to agent v_2 at time t where $P(v_1, v_2, r, t)$ holds. After matching, the residual contract is c in which A_1, A_2, R, T are bound to v_1, v_2, r, t, respectively. In this fashion the subsequent contractual obligations expressed by c may depend on the actual values in event e. The *contract combinators* $\cdot + \cdot$, $\cdot \parallel \cdot$ and $\cdot ; \cdot$ compose subcontracts according to the contract patterns we have discerned: by alternation, concurrently, and sequentially, respectively. A (contract) context is a finite set of named contract template declarations of the form $f(\vec{X}) = c$. By using the *contract instantiation* (or *contract application*) construct $f(\vec{a})$ contract templates may be (mutually) recursive, which, in particular, lets us capture repetition of subcontracts. Contract template definitions occur only at top level.

Since the contract language $\mathcal{C}^{\mathcal{P}}$ is statically typed its syntax is formally defined by the inference system in Figure 3. If top-level judgement $\Delta \vdash \mathrm{letrec}\,\mathrm{D}$ in $c : \mathrm{Contract}$ is derivable we shall say that c is well-formed in context D. Henceforth we shall assume that all contracts are well-defined, where D may be implicitly understood.

What we call contracts should justly be called *precontracts* as they do not necessarily satisfy the legal requirement for validity. In particular, Success, Failure and any expression that obligates only one agent are not judicially valid contracts. Following [JES00,JE03], we shall freely use the term contract, however. Note that consideration (*reciprocity* in REA terms) is not built into our language as a syntactic construct. This allows flexible definitions of contracts where commitments are not in a simple, syntactically evident one-to-one relation, and it allows different, user-defined notions of consideration to be applied as *analyses* to the same language.

In the following we shall adopt the convention that A_1, A_2, R, T must not be bound in environment Δ. If a variable from Δ or any expression a only involving variables bound in Δ occurs as an argument of a transmit, we interpret this as an abbreviation; for example, $\mathrm{transmit}(a, A_2, R, T \mid P).\, c$ abbreviates

$$\mathrm{transmit}(A_1, A_2, R, T \mid P \wedge A_1 = a).\, c$$

where A_1 is a new (agent-typed) variable not bound in Δ and different from A_2, R and T.
We abbreviate $\mathrm{transmit}(A_1, A_2, R, T \mid P).\,\mathrm{Success}$ to $\mathrm{transmit}(A_1, A_2, R, T \mid P)$.

The contract from Figure 1 is encoded in Figure 4, and the contract in Figure 2 is treated in depth in Sections 4 and 5.

3.2 Event Traces and Contract Satisfaction

A contract specifies a set of alternative performing event sequences (contract executions), each of which satisfies the obligations expressed in the contract and concludes it. In this section we make these notions precise for our language.

Fig. 4 Specification of Agreement to Sell Goods

```
letrec
  nonconforming [seller, buyer, goods, payment, days, t1, notice] =
      transmit (buyer, seller, notice, T |
               T < t1 + days d and #(goods,broken,t1) = 1).
      transmit (seller, buyer, payment/2, T' | T' < T + days d).

  sale [seller, buyer, goods, payment, t1, days, notice] =
      transmit (seller, buyer, goods, T | T < t1).
      transmit (buyer, seller, payment, T' | T' < t1).
      (Success + nonconforming (seller, buyer, goods, days, T', notice))
in
 sale ("Furniture maker", "Me", "Chair", 40, 2004.7.1, 8, "Chair broken")
```

Recall that our *base structure* is a tuple $(\mathcal{R}, \mathcal{T}, \mathcal{A})$ of sets of resources $\mathcal{R}$, agents $\mathcal{A}$ and a totally ordered set $(\mathcal{T}, \leq_{\mathcal{T}})$ of *dates* (or *time points*). Whenever convenient, we will extend base structures with other sets for other types, as needed. A *(transfer) event* e is a term $\mathrm{transmit}(v_1, v_2, r, t)$, where $v_1, v_2 \in \mathcal{A}, r \in \mathcal{R}$ and $t \in \mathcal{T}$. An *(event) trace* s is a finite sequence of events that is chronologically ordered; that is, for $s = e_1 \ldots e_n$ the time points in $e_1 \ldots e_n$ occur in nondescending order. We adopt the following notation: $\langle\rangle$ denotes the empty sequence; a trace consisting of a single event e is denoted by e itself; concatenation of traces s_1 and s_2 is denoted by juxtaposition: $s_1 s_2$; we write $(s_1, s_2) \leadsto s$ if s is an interleaving of the events in traces s_1 and s_2; we write $\vec{X}$ for the vector $X_1, \ldots, X_k$ with $k \geq 0$ and where k can be deduced from the context; we write $c[\vec{v}/\vec{X}]$, where $\vec{v} = v_1 \ldots v_n$ and $\vec{X} = X_1 \ldots X_n$ for some $n \geq 0$, for the result of simultaneously *substituting* elements v_i for the all free occurrences of the corresponding X_i in c. (Free and bound variables are defined as expected.)

We are now ready to specify when a trace *satisfies* a contract, i.e. gives rise to a performing execution of the contract. This is done inductively by the inference system for judgements $\delta' \vdash^{\delta}_{\mathrm{D}} s : c$ in Figure 5, where $\mathrm{D} = \{f_i[\vec{X_i}] = c_i\}_{i=1}^{m}$ is a finite set of named *contract templates* and δ is a finite set of bindings of variables to elements (values of a domain) of the given base structure. A derivable judgement $\delta' \vdash^{\delta}_{\mathrm{D}} s : c$ expresses that event sequence s satisfies—successfully executes and concludes—contract c in an environment where contract templates are defined as in D, δ is the top-level environment for both D and c, and δ' is a local environment for additional free variables in c. Conversely, if $\delta' \vdash^{\delta}_{\mathrm{D}} s : c$ is not derivable then s does not satisfy c for given $\mathrm{D}, \delta, \delta'$. The condition $\delta \oplus \delta'' \models P$ in the third rule stipulates that P, with free variables bound as in $\delta \oplus \delta'$, must be true in the base language for an event to match the corresponding commitment.

Fig. 5 Contract satisfaction

$$\delta' \vdash^{\delta}_{\mathrm{D}} \langle\rangle : \mathrm{Success} \qquad \frac{\vec{X} \mapsto \vec{v} \vdash^{\delta}_{\mathrm{D}} s : c \quad (f(\vec{X}) = c) \in \mathrm{D}, \vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta \oplus \delta'}}{\delta' \vdash^{\delta}_{\mathrm{D}} s : f(\vec{a})}$$

$$\frac{\delta \oplus \delta'' \models P \quad \delta'' \vdash^{\delta}_{\mathrm{D}} s : c \qquad (\delta'' = \delta' \oplus \{\vec{X} \mapsto \vec{v}\})}{\delta' \vdash^{\delta}_{\mathrm{D}} \mathrm{transmit}(\vec{v})\, s : \mathrm{transmit}(\vec{X}|P).\, c}$$

$$\frac{\delta' \vdash^{\delta}_{\mathrm{D}} s_1 : c_1 \quad \delta' \vdash^{\delta}_{\mathrm{D}} s_2 : c_2 \quad (s_1, s_2) \rightsquigarrow s}{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_1 \parallel c_2} \qquad \frac{\delta' \vdash^{\delta}_{\mathrm{D}} s_1 : c_1 \quad \delta' \vdash^{\delta}_{\mathrm{D}} s_2 : c_2}{\delta' \vdash^{\delta}_{\mathrm{D}} s_1 s_2 : c_1 ; c_2}$$

$$\frac{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_1}{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_1 + c_2} \qquad \frac{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_2}{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_1 + c_2}$$

3.3 Denotational Semantics

A denotational semantics maps contract specifications compositionally into a domain of mathematical objects; that is, by induction on the syntax (inference tree) of contract expressions as given by the inference rules of Figure 3. A denotational semantics supports reasoning by structural induction on the syntax. In particular, any subcontract of a contract can be replaced by any other subcontract with the same denotation without changing the behavior of the whole contract.

The satisfaction relation relates each contract to a set of traces. We can use that to define the *extension* of a contract c to be the set of its performing executions: $\mathcal{E}[\![\mathrm{letrec}\, \mathrm{D} \text{ in } c]\!]^{\delta} = \{s : \ \vdash^{\delta}_{\mathrm{D}} s : c\}$. This, however, is not a denotational semantics since it is not compositional. Turning it into a compositional definition we arrive at the semantics given in Figure 7. Note that each contract denotes a trace set, and the meaning of a compound contract can be explained in terms of a mathematical operation on the trace sets denoted by its constituent subcontracts without any reference to the actual syntax of the latter.

The presence of recursive contract definitions requires domain theory; see e.g. Winskel [Win93]. Briefly, each type in our language is mapped to a *complete partial order (cpo)*; that is, a set equipped with a partial order where each directed subset has a least upper bound (in the set). A *pointed complete partial order (pcpo)* is a cpo that has a least element. All our domains in Figure 6 are cpos since we can

Fig. 6 Domains for $\mathcal{C}^{\mathcal{P}}$

$$\begin{aligned}
\mathit{Dom}[\![\text{Boolean}]\!] &= (\{\text{true}, \text{false}\}, =)\\
\mathit{Dom}[\![\text{Agent}]\!] &= (\mathcal{A}, =)\\
\mathit{Dom}[\![\text{Resource}]\!] &= (\mathcal{R}, =)\\
\mathit{Dom}[\![\text{Time}]\!] &= (\mathcal{T}, =)\\
\mathcal{E} &= \mathcal{A} \times \mathcal{A} \times \mathcal{R} \times \mathcal{T}\\
\mathit{Tr} &= (\mathcal{E}^*, =)\\
\mathit{Dom}[\![\text{Contract}]\!] &= (2^{\mathit{Tr}}, \subseteq)\\
\mathit{Dom}[\![\tau_1 \times \ldots \times \tau_n \to \text{Contract}]\!] &= \mathit{Dom}[\![\tau_1]\!] \times \ldots \times \mathit{Dom}[\![\tau_n]\!] \to \mathit{Dom}[\![\text{Contract}]\!]\\
\mathit{Dom}[\![\Gamma]\!] &= \{\{f_i \mapsto v_i\}_{i=1}^m \mid v_i \in\\
&\qquad \mathit{Dom}[\![\tau_{i1}]\!] \times \ldots \times \mathit{Dom}[\![\tau_{in_i}]\!] \to \mathit{Dom}[\![\text{Contract}]\!]\}\\
&\qquad \text{where } \Gamma = \{f_i \mapsto \tau_{i1} \times \ldots \times \tau_{in_i} \to \text{Contract}\}_{i=1}^m\\
\mathit{Dom}[\![\Delta]\!] &= \{\{X_i \mapsto v_i\}_{i=1}^m \mid v_i \in \mathit{Dom}[\![\tau_i]\!]\}\\
&\qquad \text{where } \Delta = \{X_i : \tau_i\}_{i=1}^m\\
\mathit{Dom}[\![\Gamma; \Delta \vdash c : \text{Contract}]\!] &= \mathit{Dom}[\![\Gamma]\!] \times \mathit{Dom}[\![\Delta]\!] \to \mathit{Dom}[\![\text{Contract}]\!]
\end{aligned}$$

choose equality for the base domains $\mathcal{A}, \mathcal{R}, \mathcal{T}$. Furthermore, 2^{Tr}, the powerset of all finite event sequences, is a pcpo under $\subseteq$, and the function space $D \to D'$ is a pcpo under pointwise ordering if D' is a pcpo. A function between cpos is *continuous* if the result of applying it to the least upper bound of a directed set is the same as the least upper bound of applying it to each element of the directed set individually. It is well-known that each continuous function from a pcpo to the same pcpo has a least (unique minimal) fixed point. It is a routine matter to check that $\mathcal{C}[\![.]\!]^{\cdot}$, $\mathcal{E}[\![.]\!]^{\cdot}$ and $\mathcal{D}[\![.]\!]^{\cdot}$ map contracts under function environments, contract specifications, and contract function environments, respectively, to continuous functions. Consequently the least fixed point in line 3.3 of Figure 7 always exists.

We say c *denotes* a trace set S in context D, δ, if $\mathcal{C}[\![c]\!]^{\mathrm{D};\delta} = S$. The following theorem states that the denotational semantics characterizes the satisfaction relation.

Fig. 7 Denotational semantics

$$\begin{aligned}
\mathcal{C}\llbracket \mathrm{Success} \rrbracket^{\gamma;\delta} &= \{\langle\rangle\} \\
\mathcal{C}\llbracket \mathrm{Failure} \rrbracket^{\gamma;\delta} &= \emptyset \\
\mathcal{C}\llbracket f(\vec{a}) \rrbracket^{\gamma;\delta} &= \gamma(f)(\mathcal{Q}\llbracket \vec{a} \rrbracket^{\delta}) \\
\mathcal{C}\llbracket \mathrm{transmit}(\vec{X} \mid P).\, c \rrbracket^{\gamma;\delta} &= \{\mathrm{transmit}(\vec{v})\, s : \vec{v} \in \mathcal{E}, s \in \mathit{Tr} \mid \\
&\qquad \mathcal{Q}\llbracket P \rrbracket^{\delta \oplus \vec{X} \mapsto \vec{v}} = \mathrm{true} \wedge s \in \mathcal{C}\llbracket c \rrbracket^{\gamma;\delta \oplus \vec{X} \mapsto \vec{v}}\} \\
\mathcal{C}\llbracket c_1 + c_2 \rrbracket^{\gamma;\delta} &= \mathcal{C}\llbracket c_1 \rrbracket^{\gamma;\delta} \cup \mathcal{C}\llbracket c_2 \rrbracket^{\gamma;\delta} \\
\mathcal{C}\llbracket c_1 \parallel c_2 \rrbracket^{\gamma;\delta} &= \{s : s \in \mathit{Tr} \mid \\
&\qquad \exists s_1 \in \mathcal{C}\llbracket c_1 \rrbracket^{\gamma;\delta}, s_2 \in \mathcal{C}\llbracket c_2 \rrbracket^{\gamma;\delta}.\, (s_1, s_2) \rightsquigarrow s\} \\
\mathcal{C}\llbracket c_1; c_2 \rrbracket^{\gamma;\delta} &= \{s_1 s_2 : s_1, s_2 \in \mathit{Tr} \mid s_1 \in \mathcal{C}\llbracket c_1 \rrbracket^{\gamma;\delta} \wedge s_2 \in \mathcal{C}\llbracket c_2 \rrbracket^{\gamma;\delta}\} \\
\mathcal{D}\llbracket \{f_i[\vec{X}_i] = c_i\}_{i=1}^{m} \rrbracket^{\delta} &= \mathit{least}\, \gamma : \gamma = \{f_i \mapsto \lambda \vec{v}_i . \mathcal{C}\llbracket c_i \rrbracket^{\gamma;\delta \oplus \vec{X}_i \mapsto \vec{v}_i}\}_{i=1}^{m} \\
\mathcal{E}\llbracket \mathrm{letrec}\, \{f_i[\vec{X}_i] = c_i\}_{i=1}^{m} \text{ in } c \rrbracket^{\delta} &= \mathcal{C}\llbracket c \rrbracket^{\mathcal{D}\llbracket \{f_i[\vec{X}_i] = c_i\}_{i=1}^{m} \rrbracket^{\delta};\delta}
\end{aligned}$$

Theorem 1 (Denotational characterization of contract satisfaction)

$$\mathcal{C}\llbracket c \rrbracket^{\mathcal{D}\llbracket \mathrm{D} \rrbracket^{\delta};\delta \oplus \delta'} = \{s \mid \delta' \vdash^{\delta}_{\mathrm{D}} s : c\}$$

3.4 Contract Monitoring by Residuation

Extensionally, contracts classify traces (event sequences) into performing and nonperforming ones. We are not only interested in classifying complete event sequences once they have happened, though, but in *monitoring* contract execution as it unfolds in time under the arrival of events.

We say a trace is *consistent* with a trace set S if it is a prefix of an element of S; it is *inconsistent* otherwise.

Given a trace set S denoted by a contract c and an event e, the *residuation function* $\cdot\backslash\cdot$ captures how c can be satisfied if the first event is e. It is defined as follows: [4]

[4] Conway [Con71] calls $e\backslash S$ the *e-derivative* for a *language* S and *alphabet symbol* e. We use the term *residuation* instead to emphasize that $e\backslash S$ represents the *residual* obligations of a contract after execution of event e.

$$e\backslash S = \{s' \mid \exists s \in S : es' = s\}$$

Conceptually, we can map contracts to trace sets and use the residuation function to monitor contract execution as follows:

(1) Map a given contract c_0 to the trace set S_0 that it denotes. If $S_0 = \emptyset$, stop and output "inconsistent".
(2) For $i = 0, 1, \ldots$ do:
Receive message e_i.
    (a) If e_i is a transfer event, compute $S_{i+1} = e_i\backslash S_i$. If $S_{i+1} = \emptyset$, stop and output "breach of contract"; otherwise continue.
    (b) If e_i is a "conclude contract" message, check whether $\langle\rangle \in S_i$. If so, all obligations have been fulfilled and the contract can be terminated. Stop and output "successfully completed". If $\langle\rangle \notin S_i$, output "cannot be concluded now", let $S_{i+1} = S_i$ and continue to receive messages.

To make the conceptual algorithm for contract life cycle monitoring from Section 3.4 *operational*, we need to represent the residual trace sets and provide methods for deciding tests for emptiness and failure. In particular, we would like to use contracts as representations for trace sets. Not all trace sets are denotable by contracts, however. In particular, given a contract c that denotes a trace set S_c it is not *a priori* clear whether $e\backslash S_c$ is denotable by a contract c'. If it is, we call c' the *residual contract of c after e*.

Let us momentarily extend contract specifications with a *residuation operator*, which is the syntactic analogue of residuation, but for contracts instead of trace sets:

$$\mathcal{C}[\![e\backslash c]\!]^{\gamma;\delta} = \{s' \mid \exists s \in \mathcal{C}[\![c]\!]^{\gamma;\delta} : es' = s\}.$$

Let us write $\mathrm{D}, \delta \models c = c'$ if $\mathcal{C}[\![c]\!]^{\gamma;\delta\oplus\delta'} = \mathcal{C}[\![c']\!]^{\gamma;\delta\oplus\delta'}$ for all δ', where $\gamma = \mathcal{D}[\![\mathrm{D}]\!]^{\delta}$; analogously for $\mathrm{D}, \delta \models c \subseteq c'$. To elide parentheses we use the following operator precedence order in contract expressions (highest precedence first): residuation $\cdot\backslash\cdot$, concurrent composition $\cdot \parallel \cdot$, alternation $\cdot + \cdot$, sequential composition $\cdot ; \cdot$.

Lemma 2 (Correctness of residuation) *The reduation equalities in Figure 8 are true.*

For the proof of this lemma we need an auxiliary lemma that extends the compositionality of the base language to the contract language:

Lemma 3 (Agreement of substitution and environments) *For all c, γ and*

Fig. 8 Residuation equalities

$$\mathrm{D}, \delta \models e \backslash \mathrm{Success} = \mathrm{Failure}$$

$$\mathrm{D}, \delta \models e \backslash \mathrm{Failure} = \mathrm{Failure}$$

$$\mathrm{D}, \delta \models e \backslash f(\vec{a}) = e \backslash c[\vec{v}/\vec{X}] \text{ if } (f(\vec{X}) = c) \in \mathrm{D}, \vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta}$$

$$\mathrm{D}, \delta \models \mathrm{transmit}(\vec{v}) \backslash (\mathrm{transmit}(\vec{X} \mid P).\, c) = \begin{cases} c[\vec{v}/\vec{X}] \text{ if } \delta \oplus \{\vec{X} \mapsto \vec{v}\} \models P \\ \text{Failure otherwise} \end{cases}$$

$$\mathrm{D}, \delta \models e \backslash (c_1 + c_2) = e \backslash c_1 + e \backslash c_2$$

$$\mathrm{D}, \delta \models e \backslash (c_1 \parallel c_2) = e \backslash c_1 \parallel c_2 + c_1 \parallel e \backslash c_2$$

$$\mathrm{D}, \delta \models e \backslash (c_1; c_2) = \begin{cases} (e \backslash c_1; c_2) + e \backslash c_2 \text{ if } \mathrm{D}, \delta \models \mathrm{Success} \subseteq c_1 \\ e \backslash c_1; c_2 \qquad \text{otherwise} \end{cases}$$

δ:

$$\mathcal{C}[\![c]\!]^{\gamma;\delta \oplus \vec{X} \mapsto \vec{v}} = \mathcal{C}[\![c[\vec{v}/\vec{X}]]\!]^{\gamma;\delta}$$

Executing the residuation equations as left-to-right rewrite rules eliminates the residuation operator in $e \backslash c$, assuming c is residuation operator free to start with. That computation does not always terminate, however. Consider, e.g.,

$$\mathrm{letrec}\, f(N) = (\mathrm{transmit}(a_1, a_2, r, T \mid T \leq N) \parallel f(N+1)) \text{ in } f(0)$$

and event transmit$(a_1, a_2, r, 0)$. Applying the rewrite rules will not terminate. Intuitively, this is because transmit$(a_1, a_2, r, 0)$ can be matched against any one of the infinitely many commitments

$$\mathrm{transmit}(a_1, a_2, r, T_0 | T_0 \leq 0) \parallel \cdots \parallel \mathrm{transmit}(a_1, a_2, r, T_i | T_i \leq i) \parallel \cdots$$

since transmit$(a_1, a_2, r, 0)$ satisfies the match condition of each one of them. Note that, semantically, $f(N) = \mathrm{transmit}(a_1, a_2, r, T \mid T \leq N) \parallel f(N+1), \emptyset \models f(0) =$ Failure, but left-to-right rewriting according to Figure 8 does not rewrite $f(0)$ to Failure.

3.5 Nullable and Guarded Contracts

In this section we characterize *nullability* of a contract and introduce *guarding*, which is a sufficient condition on contracts for ensuring that residuation can be performed by reduction on contracts.

Fig. 9 Nullable contracts

$$\frac{\mathrm{D} \vdash c \text{ nullable} \quad (f(\vec{X}) = c) \in \mathrm{D}}{\mathrm{D} \vdash f(\vec{a}) \text{ nullable}} \qquad \frac{\mathrm{D} \vdash c \text{ nullable}}{\mathrm{D} \vdash c + c' \text{ nullable}}$$

$$\frac{\mathrm{D} \vdash c' \text{ nullable}}{\mathrm{D} \vdash c + c' \text{ nullable}} \qquad \mathrm{D} \vdash \text{Success nullable}$$

$$\frac{\mathrm{D} \vdash c \text{ nullable} \quad \mathrm{D} \vdash c' \text{ nullable}}{\mathrm{D} \vdash c \parallel c' \text{ nullable}} \qquad \frac{\mathrm{D} \vdash c \text{ nullable} \quad \mathrm{D} \vdash c' \text{ nullable}}{\mathrm{D} \vdash c; c' \text{ nullable}}$$

Definition 4 (Nullability) *(1) We write* $\mathrm{D} \models c$ nullable *if* $\mathrm{D}, \delta \models \text{Success} \subseteq c$ *for some* δ*; that is,* $\langle\rangle \in \mathcal{C}[\![c]\!]^{\mathrm{D};\delta}$*.*

(2) We say c *is* nullable *(or* terminable*) in context* D *if* $\mathrm{D} \vdash c$ nullable *is derivable by the inference system in Figure 9.*

A nullable contract can be concluded successfully, but may possibly also be continued. E.g., the contract Success + transmit$(a_1, a_2, r, t|P)$ is nullable, as it may be concluded successfully (left choice). Note however, that it may also be continued (right choice). It is easy to see that nullability is independent of δ and δ': $\langle\rangle \in \mathcal{C}[\![c]\!]^{\gamma;\delta\oplus\delta'}$ if and only if $\langle\rangle \in \mathcal{C}[\![c]\!]^{\gamma;\hat{\delta}\oplus\hat{\delta}'}$ for any other $\hat{\delta}$ and $\hat{\delta}'$, where $\gamma = \mathcal{D}[\![\mathrm{D}]\!]^{\delta}$. Deciding nullability is required to implement Step 2b in contract monitoring. The following proposition expresses that nullability characterizes semantic nullability.

Proposition 5 (Syntactic characterization of nullability)

$$\mathrm{D} \models c \text{ nullable} \iff \mathrm{D} \vdash c \text{ nullable}.$$

Definition 6 (Guarded contract, guarded declarations) *Let* $\mathrm{D} = \{f_i[\vec{X_i}] = c_i\}_{i=1}^{m}$ *be contract template declarations.*

A contract c *is* guarded *in context* D *if* $\mathrm{D} \vdash c$ guarded *is derivable from Figure 10. We say* D *is guarded if* c_i *is guarded in context* D *for all* i *with* $1 \leq i \leq m$*.*

Intuitively, guardedness ensures that we do not have (mutual) recursions such as $\{f(\vec{X}) = g(\vec{X}), g(\vec{X}) = f(\vec{X})\}$ that cause the residuation algorithm to loop infinitely. Guarded declarations ensure that all contracts built from them are guarded:

Lemma 7 (Guardedness of contracts using guarded declarations) *For all* D, c*, if* D *is guarded then* $\mathrm{D} \vdash c$ guarded*.*

Fig. 10 Guarded contracts

$$\mathrm{D} \vdash \text{Success guarded} \qquad \mathrm{D} \vdash \text{Failure guarded}$$

$$\mathrm{D} \vdash \text{transmit}(\vec{X} \mid P).\, c \text{ guarded} \qquad \frac{\mathrm{D} \vdash c \text{ guarded} \quad (f(\vec{X}) = c) \in \mathrm{D}}{\mathrm{D} \vdash f(\vec{a}) \text{ guarded}}$$

$$\frac{\mathrm{D} \vdash c \text{ guarded} \quad \mathrm{D} \vdash c' \text{ guarded}}{\mathrm{D} \vdash c + c' \text{ guarded}} \qquad \frac{\mathrm{D} \vdash c \text{ guarded} \quad \mathrm{D} \vdash c' \text{ guarded}}{\mathrm{D} \vdash c \parallel c' \text{ guarded}}$$

$$\frac{\mathrm{D} \vdash c \text{ nullable} \quad \mathrm{D} \vdash c \text{ guarded} \quad \mathrm{D} \vdash c' \text{ guarded}}{\mathrm{D} \vdash c; c' \text{ guarded}}$$

$$\frac{\mathrm{D} \not\vdash c \text{ nullable} \quad \mathrm{D} \vdash c \text{ guarded}}{\mathrm{D} \vdash c; c' \text{ guarded}}$$

As we shall see, guardedness is key to ensuring termination of contract residuation and thus that every (guarded) contract has a residual contract under any event in the reduction semantics of Figure 11.

3.6 Operational Semantics I: Deferred Matching

The denotational semantics tells us what trace set is denoted by a contract, and residuation on trace sets tells us how to turn the denotational semantics conceptually into a *monitoring* semantics. In this section we present a *reduction semantics* for contracts, which lifts residuation on trace sets to contracts and is derived systematically from the residuation equalities of Figure 8.

The ability of representing residual contract obligations of a partially executed contract and thus any state of a contract as a *bona fide* contract carries the advantage

Fig. 11 Deterministic reduction (delayed matching)

$$\mathrm{D}, \delta \vdash_D \text{Success} \xrightarrow{e} \text{Failure} \qquad \mathrm{D}, \delta \vdash_D \text{Failure} \xrightarrow{e} \text{Failure}$$

$$\frac{\delta \oplus \{\vec{X} \mapsto \vec{v}\} \models P \quad (\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta})}{\mathrm{D}, \delta \vdash_D \text{transmit}(\vec{X}|P).\, c \overset{\text{transmit}(\vec{v})}{\longrightarrow} c[\vec{v}/\vec{X}]}$$

$$\frac{\delta \oplus \{\vec{X} \mapsto \vec{v}\} \not\models P \quad (\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta})}{\mathrm{D}, \delta \vdash_D \text{transmit}(\vec{X}|P).\, c \overset{\text{transmit}(\vec{v})}{\longrightarrow} \text{Failure}}$$

$$\frac{\mathrm{D}, \delta \vdash_D c[\vec{v}/\vec{X}] \xrightarrow{e} c' \quad (f(\vec{X}) = c) \in \mathrm{D}, \vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta}}{\mathrm{D}, \delta \vdash_D f(\vec{a}) \xrightarrow{e} c'}$$

$$\frac{\mathrm{D}, \delta \vdash_D c \xrightarrow{e} d \quad \mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d'}{\mathrm{D}, \delta \vdash_D c + c' \xrightarrow{e} d + d'}$$

$$\frac{\mathrm{D}, \delta \vdash_D c \xrightarrow{e} d \quad \mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d'}{\mathrm{D}, \delta \vdash_D c \parallel c' \xrightarrow{e} c \parallel d' + d \parallel c'}$$

$$\frac{\mathrm{D} \vdash c \text{ nullable} \quad \mathrm{D}, \delta \vdash_D c \xrightarrow{e} d \quad \mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d'}{\mathrm{D}, \delta \vdash_D c; c' \xrightarrow{e} (d; c') + d'}$$

$$\frac{\mathrm{D} \not\vdash c \text{ nullable} \quad \mathrm{D}, \delta \vdash_D c \xrightarrow{e} d}{\mathrm{D}, \delta \vdash_D c; c' \xrightarrow{e} d; c'}$$

that any analysis that is performed on “original” contracts automatically extends to partially executed contracts as well. E.g., an investment bank that applies valuations to financial contracts before offering them to customers can apply their valuations to their portfolio of contracts under execution; e.g., to analyze its risk exposure under current market conditions.

Likewise, a company that analyzes production and capacity requirements of a contract before offering it to a customer can apply the same analysis to the contracts it has under execution; e.g., to adjust planning based on present capacity requirements. The reduction semantics is presented in Figure 11. The basic *matching rule* is

$$\frac{\delta \oplus \{\vec{X} \mapsto \vec{v}\} \models P \quad (\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta})}{\mathrm{D}, \delta \vdash_D \mathrm{transmit}(\vec{X}|P).\, c \xrightarrow{\mathrm{transmit}(\vec{v})} c[\vec{v}/\vec{X}]}$$

It *matches* an event with a specific commitment in a contract. There may be multiple commitments in a contract that match the same event. The semantics captures the possibilities of matching an event against multiple commitments by applying all possible reductions in alternatives and concurrent contract forms and forming the sum of their possible outcomes (some of which may actually be Failure).

The rule

$$\frac{\mathrm{D}, \delta \vdash_D c \xrightarrow{e} d \quad \mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d'}{\mathrm{D}, \delta \vdash_D c + c' \xrightarrow{e} d + d'}$$

thus reduces both alternatives c and c' and then forms the sum of their respective results d, d'.

Likewise, the rule

$$\frac{\mathrm{D}, \delta \vdash_D c \xrightarrow{e} d \quad \mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d'}{\mathrm{D}, \delta \vdash_D c \parallel c' \xrightarrow{e} c \parallel d' + d \parallel c'}$$

for concurrent subcontracts expresses that the match could be in either one of c or c' and represents the result as the sum of those two possibilities.

Finally, the rule

$$\frac{\mathrm{D} \vdash c \text{ nullable} \quad \mathrm{D}, \delta \vdash_D c \xrightarrow{e} d \quad \mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d'}{\mathrm{D}, \delta \vdash_D c; c' \xrightarrow{e} (d; c') + d'}$$

captures that e can be matched in c or, if c is nullable, in c'. Note that, if c is not nullable, e can only be matched in c, not c', as expressed by the rule

$$\frac{\mathrm{D} \not\vdash c \text{ nullable} \quad \mathrm{D}, \delta \vdash_D c \xrightarrow{e} d}{\mathrm{D}, \delta \vdash_D c; c' \xrightarrow{e} d; c'}.$$

In this fashion the semantics keeps track of the results of all possible matches in a reduction sequence as explicit *alternatives* (summands) and *defers* the decision as to *which specific* commitment is matched by a particular event during contract execution until the very end: By selecting a particular summand in a residual contract after a number of reduction steps that represents Success (and the contract is thus terminable) a particular set of matching decisions is chosen *ex post*. As presented, the reduction semantics gives rise to an implementation in which the multiple reducts of previous reduction steps are reduced in parallel, since they are represented as summands in a single contract, and the rule for reduction of sums reduces both summands. It is relatively straightforward to turn this into a backtracking semantics by an asymmetric reduction rule for sums, which delays reduction of the right summand.

The operational semantics fully and faithfully implements residuation (when the residuation equalities are oriented):

Theorem 8 (Residuation by deferred matching) *(1) For any* c, c', δ, e *and* D*: if* $\mathrm{D}, \delta \vdash_D c \xrightarrow{e} c'$ *then* $\mathrm{D}, \delta \models e \backslash c = c'$*.*

(2) For all c, δ *and guarded* D*, there exists a unique* c' *such that* $\mathrm{D}, \delta \vdash_D c \xrightarrow{e} c'$*; furthermore,* $\mathrm{D} \vdash c'$ guarded*.*

Using Theorem 8 we can turn our conceptual contract monitoring algorithm into a real algorithm.

(1) Let contract c_0 be given. If c_0 is inconsistent, stop and output “inconsistent”.

(2) For $i = 0, 1, \ldots$ do:
Receive message e_i.
   (a) If e_i is a transfer event, let c_{i+1} be such that $\vdash_D c_i \xrightarrow{e_i} c_{i+1}$. If c_{i+1} is inconsistent, stop and output “breach of contract”; otherwise continue.
   (b) If e is a “terminate contract” message, check whether c_i is nullable. If so, all obligations have been fulfilled and the contract can be terminated. Stop and output “successfully completed”. If c_i is not nullable, output “cannot be terminated now”, let $c_{i+1} = c_i$ and continue to receive messages.

Proposition 5 provides a syntactic characterization of nullability, which can easily be turned into an algorithm. Deciding $\mathrm{D}, \delta \models c = \mathrm{Failure}$, that is whether a contract has actually failed, is a much harder problem. See Figure 21 for a sketch for a conservative approximation (some failed contracts may not be identified as such) to this.

3.7 Operational Semantics II: Eager Matching

The deferred matching semantics of Figure 11 is flexible and faithful to the natural notion of contract satisfaction as defined in Figure 5. But from an accounting practice view it is weird because matching decisions are deferred. In bookkeeping standard *modus operandi* is that events are matched against specific commitments *eagerly*; that is online, as events arrive.[5]

We shall turn the deferred matching semantics of Figure 11 into an eager matching semantics (Figure 12). The idea is simple: Represent here-and-now choices as alternative *rules* (meta-level) as opposed to alternative contracts (object level). Specifically, we split the rules for reducing alternatives and concurrent subcontracts into multiple rules, and we capture the possibility of reducing in the second component of a sequential contract by adding τ-transitions, which "spontaneously" (without a driving external event) reduce a contract of the form $\mathrm{Success}; c$ to c. For this to be sufficient we have to make sure that a nullable contract indeed can be reduced to Success, not just a contract that is *equivalent* to Success, such as Success $\|$ Success. This is done by ensuring that τ-transitions are strong enough to guarantee reduction to Success as required.

Based on these considerations we arrive at the reduction semantics in Figure 12, where meta-variable λ ranges over events e and the internal event τ. Note that it is nondeterministic and not even confluent: A contract c can be reduced to two different contracts by the same event. Consider e.g., $c = a; b + a; b'$ where a, b, b' are commitments, no two of which match the same event. For event e matching a we have $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} b$ and $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} b'$, but neither b nor b' can be reduced to Success or any other contract by the same event sequence. In reducing c we have not only resolved it against e, but also made a *decision*: whether to apply it to the first alternative of c or to the second. Technically, the reduction semantics is not closed under residuation: Given c and e it is not always possible to find c' such that $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} c'$ and $\mathrm{D}; \delta \models e \backslash c = c'$. It is sound, however, in the sense that the reduct always denotes a subset of the residual trace set. It is furthermore complete in the sense that the set of all reductions do preserve residuation.

Theorem 9 (Soundness of eager matching)

(1) If $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} c'$ *then* $\mathrm{D}, \delta \models c' \subseteq e \backslash c$*.*
(2) If $\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau} c'$ *then* $\mathrm{D}, \delta \models c' \subseteq c$*.*

Even though individual eager reductions do not preserve residuation, the set of all reductions does so:

[5] There are standard accounting practices for changing such decisions, but both default and standard conceptual model are that matching decisions are made as early as possible. In general, it seems representing and deferring choices and applying *hypothetical* reasoning to them appears to be a rather unusual phenomenon in accounting.

Fig. 12 Nondeterministic reduction (eager matching)

$$\mathrm{D}, \delta \vdash_N \mathrm{Success} \xrightarrow{e} \mathrm{Failure} \qquad \mathrm{D}, \delta \vdash_N \mathrm{Failure} \xrightarrow{e} \mathrm{Failure}$$

$$\frac{\delta \oplus \{\vec{X} \mapsto \vec{v}\} \models P, \vec{v} = \mathcal{Q}[\![\vec{a}]\!]^\delta}{\mathrm{D}, \delta \vdash_N \mathrm{transmit}(\vec{X} \mid P).\, c \xrightarrow{\mathrm{transmit}(\vec{v})} c[\vec{v}/\vec{X}]}$$

$$\frac{\delta \oplus \{\vec{X} \mapsto \vec{v}\} \not\models P, \vec{v} = \mathcal{Q}[\![\vec{a}]\!]^\delta}{\mathrm{D}, \delta \vdash_N \mathrm{transmit}(\vec{X} | P).\, c \xrightarrow{\mathrm{transmit}(\vec{v})} \mathrm{Failure}}$$

$$\frac{(f(\vec{X}) = c) \in \mathrm{D}, \vec{v} = \mathcal{Q}[\![\vec{a}]\!]^\delta}{\mathrm{D}, \delta \vdash_N f(\vec{a}) \xrightarrow{\tau} c[\vec{v}/\vec{X}]} \qquad \mathrm{D}, \delta \vdash_N c + c' \xrightarrow{\tau} c \qquad \mathrm{D}, \delta \vdash_N c + c' \xrightarrow{\tau} c'$$

$$\frac{\mathrm{D}, \delta \vdash_N c \xrightarrow{\lambda} d}{\mathrm{D}, \delta \vdash_N c \parallel c' \xrightarrow{\lambda} d \parallel c'} \qquad \frac{\mathrm{D}, \delta \vdash_N c' \xrightarrow{\lambda} d'}{\mathrm{D}, \delta \vdash_N c \parallel c' \xrightarrow{\lambda} c \parallel d'}$$

$$\mathrm{D}, \delta \vdash_N \mathrm{Success} \parallel c \xrightarrow{\tau} c \qquad \mathrm{D}, \delta \vdash_N c \parallel \mathrm{Success} \xrightarrow{\tau} c$$

$$\mathrm{D}, \delta \vdash_N \mathrm{Success}; c' \xrightarrow{\tau} c'$$

$$\frac{\mathrm{D}, \delta \vdash_N c \xrightarrow{\lambda} d}{\mathrm{D}, \delta \vdash_N c; c' \xrightarrow{\lambda} d; c'} \qquad \frac{\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau} c' \quad \mathrm{D}, \delta \vdash_N c' \xrightarrow{e} c''}{\mathrm{D}, \delta \vdash_N c \xrightarrow{e} c''}$$

$$\frac{\mathrm{D}, \delta \vdash_N c \xrightarrow{e} c'}{\delta \vdash_N \mathrm{letrec}\, \mathrm{D} \text{ in } c \xrightarrow{e} \mathrm{letrec}\, \mathrm{D} \text{ in } c'}$$

Theorem 10 (Completeness of eager matching)
If $\mathrm{D}, \delta \vdash_D c \xrightarrow{e} c'$ *then there exist contracts* $c_1, \ldots, c_n$ *for some* $n \geq 1$ *such that* $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} c_i$ *for all* $i = 1 \ldots n$ *and* $\mathrm{D}, \delta \models c' \subseteq \sum_{i=1}^n c_i$*.*

As a corollary, Theorems 9 and 10 combined yield that the object-level nondeterminism (expressed as contract alternatives) in the deferred matching semantics is faithfully reflected in the meta-level nondeterminism (expressed as multiple applicable rules) of the eager matching semantics.

3.8 Operational Semantics III: Eager Matching with Explicit Routing

Consider the following execution model for contracts: Two or more parties each have a copy of the contract they have previously agreed upon and monitor its execution under the arrival of events. Even if they agree on prior contract state and the next event, the parties may arrive at different residual contracts and thus different expectations as to the future events allowed under the contract. This is because of nondeterminacy in contract execution with eager matching; e.g., a payment of \$50 may match multiple payment commitments, and the parties may make different matches. We can remedy this by making *control* of contract reduction with eager matching explicit in order to make reduction deterministic: events are accompanied by control information that unambiguously prescribes how a contract is to be reduced. In this fashion parties that agree on what events have happened and on their associated control information, will reduce their contract identically. [6]

The basic idea is that all nondeterminism in our reduction semantics (see Figure 12) can be reduced to a series of choices and routing decisions to identify the particular commitment the event is to be matched with; in particular, we can express such a series as an element of I^* where $I = \{f, s, l, r\}$; see below. A control-annotated event then is an element of $I^*\mathcal{E}$. (Recall that $\mathcal{E}$ denotes the set of transfer events.) In Figure 13 we note that $\mathbf{d} \in I^*$.

The τ-reductions in Figure 13 rewrite a contract into a simplified form while preserving its semantics faithfully:

Proposition 11 (Soundess of τ-reduction) *For all* $\mathrm{D}, \delta, c, c'$, *if* $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} c'$ *then* $\mathrm{D}, \delta \models c = c'$.

Furthermore, they are strong enough to guarantee that any contract equivalent to Success actually reduces to Success.

[6] The question of which party has the right of generating control information is very important, of course. It will be discussed only briefly later, as it is beyond the scope of this paper. We only require that a consensus on the events and their associated control information has been achieved, whether dictated by one party or the other having the (contractual) right to do so or by an actual consensus process.

Fig. 13 Eager matching with explicit reduction control

$$\mathrm{D}, \delta \vdash_C \mathrm{Success} \xrightarrow{e} \mathrm{Failure} \quad \mathrm{D}, \delta \vdash_C \mathrm{Failure} \xrightarrow{e} \mathrm{Failure}$$

$$\frac{\delta \oplus \{\vec{X} \mapsto \vec{v}\} \models P \quad (\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta})}{\mathrm{D}, \delta \vdash_C \mathrm{transmit}(\vec{X} \mid P).\, c \xrightarrow{\mathrm{transmit}(\vec{v})} c[\vec{v}/\vec{X}]}$$

$$\frac{\delta \oplus \{\vec{X} \mapsto \vec{v}\} \not\models P \quad (\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta})}{\mathrm{D}, \delta \vdash_C \mathrm{transmit}(\vec{X}|P).\, c \xrightarrow{\mathrm{transmit}(\vec{v})} \mathrm{Failure}}$$

$$\frac{(f(\vec{X}) = c) \in \mathrm{D} \quad (\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta})}{\mathrm{D}, \delta \vdash_C f(\vec{a}) \xrightarrow{\tau} c[\vec{v}/\vec{X}]}$$

$$\frac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} d}{\mathrm{D}, \delta \vdash_C c + c' \xrightarrow{\tau} d + c'} \qquad \frac{\mathrm{D}, \delta \vdash_C c' \xrightarrow{\tau} d'}{\mathrm{D}, \delta \vdash_C c + c' \xrightarrow{\tau} c + d'}$$

$$\mathrm{D}, \delta \vdash_C \mathrm{Success} + \mathrm{Success} \xrightarrow{\tau} \mathrm{Success}$$

$$\frac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\vec{de}} c'}{\mathrm{D}, \delta \vdash_C c + d \xrightarrow{\mathrm{f}\vec{de}} c'} \qquad \frac{\mathrm{D}, \delta \vdash_C d \xrightarrow{\vec{de}} d'}{\mathrm{D}, \delta \vdash_C c + d \xrightarrow{\mathrm{s}\vec{de}} d'}$$

$$\frac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} d}{\mathrm{D}, \delta \vdash_C c \parallel c' \xrightarrow{\tau} d \parallel c'} \qquad \frac{\mathrm{D}, \delta \vdash_C c' \xrightarrow{\tau} d'}{\mathrm{D}, \delta \vdash_C c \parallel c' \xrightarrow{\tau} c \parallel d'}$$

$$\frac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\vec{de}} d}{\mathrm{D}, \delta \vdash_C c \parallel c' \xrightarrow{\mathrm{l}\vec{de}} d \parallel c'} \qquad \frac{\mathrm{D}, \delta \vdash_C c' \xrightarrow{\vec{de}} d'}{\mathrm{D}, \delta \vdash_C c \parallel c' \xrightarrow{\mathrm{r}\vec{de}} c \parallel d'}$$

$$\mathrm{D}, \delta \vdash_C \mathrm{Success} \parallel c \xrightarrow{\tau} c \qquad \mathrm{D}, \delta \vdash_C c \parallel \mathrm{Success} \xrightarrow{\tau} c$$

$$\frac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} d}{\mathrm{D}, \delta \vdash_C c; c' \xrightarrow{\tau} d; c'} \qquad \frac{\mathrm{D}, \delta \vdash_C c \xrightarrow{e} d}{\mathrm{D}, \delta \vdash_C c; c' \xrightarrow{e} d; c'} \qquad \mathrm{D}, \delta \vdash_C \mathrm{Success}; c' \xrightarrow{\tau} c'$$

Proposition 12 (Completeness of τ-reduction for concluded contracts) *For all* $\mathrm{D}, \delta, c, c'$*:* $\mathrm{D}, \delta \models c = \mathrm{Success}$ *if and only if* $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau^*} \mathrm{Success}$.

Finally, τ-rewriting is strongly normalizing and confluent, which means that each contract has a unique τ-normal form, which can be computed by applying the τ-rewriting rules exhaustively in arbitrary order.

Lemma 13 (Unique normalization of τ-reduction) *For all* δ *and guarded* D *there is a unique* c' *such that*

(1) $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau^*} c'$ *and*
(2) for no c'' *do we have* $\mathrm{D}, \delta \vdash_C c' \xrightarrow{\tau} c''$.

We say c' in Lemma 13 is *τ-normalized* or simply *normalized* and we call it the *τ-normalized* form of c. We can observe that a contract is nullable if and only if its τ-normalized form has the form $\ldots + \mathrm{Success} + \ldots$; that is, has a Success-summand.

The following theorem expresses that sequences of labels $\mathrm{f}, \mathrm{s}, \mathrm{l}, \mathrm{r}$ preceding an economic event unambiguously determine how a contract should be reduced.

Theorem 14 (Correctness of eager matching with routing) *For each* δ*,* D*, normalized* c *and event* e *we have that* $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} c'$ *if and only if there exists* $\vec{d} \in \{\mathrm{f}, \mathrm{s}, \mathrm{l}, \mathrm{r}\}^*$ *such that* $\mathrm{D}, \delta \vdash_C c \xrightarrow{\vec{d}e} c'$*. Furthermore, for all* c'' *such that* $\mathrm{D}, \delta \vdash_C c \xrightarrow{\vec{d}e} c''$ *we have* $c' = c''$*; that is, given* c *and control-annotated event* $\vec{d}e$ *the residual contract* c'' *is uniquely determined.*

Intuitively, a control-annotated event $\vec{d}e$ conveys an event e and information $\vec{d}$ that unambiguously *routes* the event to the particular commitment it is to be matched with: f, s determine which branch of a $. + .$-contract is to be chosen, and l, r identify in which subcontract of a $. \parallel .$-contract the economic event is to be matched. This routing information ensures that all trading partners in a contract, each maintaining their own state of the contract, match events to the same atomic commitment and thus can be assured that they will also be in agreement on the residual contract. Other methods for controlling reduction in an eager matching semantics are discussed by Andersen and Elsborg [AE03].

Some of these left/right choices may be further eliminated in practice (that is, inferred automatically) where they are “forced” (no other choice allows successful completion of contract).

4 Example Contracts

We previously saw an encoding of the Agreement to Sell Goods (Figure 4). In this section, two additional real-life example contracts are considered.

First, the previously presented abbreviated version of the natural language Legal Services Agreement (Figure 2) is encoded in our contract specification language. Second, we present a natural language contract for software development (Figure 15) and provide its encoding in our language (Figure 16).

Before it is possible to express real-life contracts, however, the predicate language and the arithmetic language must be defined. For the purpose of demonstration we will afford ourselves a fairly advanced language that has multiple datatypes (e.g. integers and dates), common arithmetic operators, logical connectives, lists and a number of built-in functions. The syntax is common and straightforward, and hence we shall not delve into the technical details here. Later, in Section 5, we will define the language and consider possible restrictions that ameliorate contract analysis.

Fig. 14 Specification of Agreement to Provide Legal Services

```
letrec
extra (att, com, invoice, pay) =
  ( Success
  + transmit (att, com, invoice, T2).
    transmit (com, att, pay, T3 | T3 <= T2 + 45d))

legal (att, com, fee, invoice, pay, n, m, end) =
  transmit (att, com, H, T | n < T and T <= m).
    (   extra (att, com, invoice, pay)
    ||  transmit (com att, fee, T | T <= m + 8d)
    ||  ( legal (att, com, fee, invoice, pay, m, min(m + 30d,end), end)
        + transmit (att, com, end, T | end <= T)))
in
  legal ("Attorney","Company",10000,invoice,pay,0,30,360)
```

Writing the formal specification of the Legal Services Agreement (Figure 2) is fairly straightforward, bar two points: Consider the validity period specified in Section 3 of the contract. Taken literally, it would imply, that the attorney shall render services in the month of December, but receive no fee in consideration since January 2005 is outside the validity period. Surely, this is not the intention; in fact, consideration will defeat most deadlines as is clearly the intent here and this is avoided in the encoding of the contract (Figure 14). This weakness in the informal

Fig. 15 Software Development Agreement

Section 1. The Developer shall develop software as described in Exhibit A (Requirements Specification) according the schedule set forth in Exhibit B (Project Schedule and Deliverables). Specifically, the Developer shall be responsible for the timely completion of the deliverables identified in Exhibit B.

Section 2. The Client shall provide written approval upon the completion of each deliverable identified in Exhibit B.

Section 3. In the event of any delay by the Client, all the Developer's remaining deadlines shall be extended by the greater of the two following: (i) five working days, (ii) two times the delay induced by the Client. The Client's deadlines shall be unchanged.

Section 4. In consideration of services rendered the Client shall pay USD $100.000 due on 7/1.

Section 5. If the Client wishes to add to the order, or if upon written approval of a deliverable, the Client wishes to make modifications to the deliverable, the Client and the Developer shall enter into a Change Order. Upon mutual agreement the Change Order shall be attached to this contract.

Section 6. The Developer shall retain all intellectual rights associated with the software developed. The Client may not copy or transfer the software to any third party without the explicit, written consent of the Developer.

Exhibit A. (omitted)

Exhibit B. Deadlines for deliverables and approval: (i) 1/1, 1/15; (ii) 3/1, 3/15, (final deadline) 7/1, 7/15.

contract is revealed, which is a good thing, when encoding it formally.

The Agreement to Provide Legal Services fails to specify who decides if legal services should be rendered. In the encoding it is simply assumed that the attorney is the initiator and that all services rendered over a month can be modelled as one event. Based on the hours of services rendered, the attorney has a choice to invoice extra hours at the hourly rate. Furthermore, the attorney is assumed to give the notice `end` to allow contract termination. This is introduced to make sure that the contract is not nullable between every recursion.

Now consider the more elaborate Software Development Agreement in Figure 15. When coding the contract, one notices that the contract fails to specify the ramifications of the client's non-approval of a deliverable. One also sees that the contract does not specify what to do if due to delay, some approval deadline comes before the postponed delivery date. In the current code, this is taken to mean further delay on the client's part even if the client gave approval at the same time as the deliverable was transmitted. It seems that contract coding is a healthy process

Fig. 16 Specification of Software Development Agreement – note that we assume (easily defined) abbreviations for `max(x,y)` and allow subtraction on the domain Time.

```
letrec
  deliverables (dev, client, payment, deliv1, deadline1, approv1,
                                    deliv2, deadline2, approv2,
                                    delivf, deadlinef, approvf) =
    transmit(dev, client, deliv1, T1 | T1 <= deadline1)).
    transmit(client, dev, "ok", T).
    transmit(dev, client, deliv2, T2 |
             T2 <= deadline2 + max(5d, (T - approv1) * 2)).
    transmit(client, dev, "ok", T).
    transmit(dev, client, delivf, Tf |
             Tf <= deadlinef + max(5d, (T - approv2) * 2)).
    transmit(client, dev, "ok", T).
    transmit(dev, client, "done", T).
    Success

  software (dev, client, payment, paymentdeadline, ds) =
    deliverables (dev, client, deliv1, deadline1, approv1,
                               deliv2, deadline2, approv2,
                               delivf, deadlinef, approvf) ||
    transmit(client, dev, payment, T | T <= paymentdeadline)
in
 software ("Me", "Client", 100000, 2004.7.1, d1, 2004.1.1, 2004.1.15,
          d2, 2004.3.1, 2004.3.15, final, 2004.7.1, 2004.7.15)
```

in the sense that it will often unveil underspecification and errors in the natural language contract being coded. The Change Order described in Section 5 of the contract and the intellectual rights described in Section 6 are not coded due to certain limitations in our language. We will postpone the discussion of this until Section 6.

4.1 Example Reduction

We now demonstrate how the Legal Services Agreement behaves under our three reduction strategies: deferred matching, eager matching, and eager matching with explicit control. All three derivations assume that we invoke the contract as

```
legal (att, com, fee, invoice, pay, 0, 30, 60)
```

i.e. we would like the contract to run for two months. Of course, the parameters `att`, `com`, `fee`, `invoice`, and `pay` should be bound to values, but we leave them as is for readability since none of them have an impact on the control flow of the contract. This yields the contract body:

```
transmit (att, com, H, T | 0 < T and T <= 30).
(   transmit (com att, fee, T | T <= 30 + 8d)
||  ( legal (att, com, fee, invoice, pay, 30, min(30 + 30d,60), 60)
    + transmit (att, com, end, T | 60 <= T)))
```

The sub-contract `extra` has been taken out to reduce the size of the reductions. To facilitate comparison we will use the same basic event trace for all three reduction strategies:

$\xrightarrow{(att,com,h1,20)}$ Services rendered first month
$\xrightarrow{(att,com,h2,37)}$ Services rendered second month
$\xrightarrow{(com,att,fee,38)}$ Fee for first month
$\xrightarrow{(com,att,fee,62)}$ Fee for second month
$\xrightarrow{(att,com,end,64)}$ Attorney signals end-of-contract

The trace will be furnished with reduction controls and interspersed with τ when mandated by the concrete semantics in question. Consider Figure 17 for a juxtaposition of the two eager matching strategies (with and without explicit control) on the Legal Services Agreement and Figure 18 for a demonstration of the deferred matching strategy.

5 Contract Analysis

The formal groundwork in order, we can begin to ask ourselves questions about contracts such as: What is my first order of business? When is the next deadline? How much of a particular resource will I gain from my portfolio and at what times? What is the monetary value of my portfolio? Is the contract I just wrote “safe” and “fair”? Will contract fulfillment require more than the x units I currently have in stock?

The attempt to answer such questions is broadly referred to as *contract analysis*. Some analyses, notably “safeness”, will primarily be of interest during contract

Fig. 17 Eager matching without and with explicit control on the legal services agreement

```
transmit (att, com, H, T | 0 < T and T <= 30).
(  transmit (com, att, fee, T | T <= 30 + 8d)
|| ( legal (..., 30, min(30 + 30d,60), 60)
   + transmit (att, com, end, T | 60 <= T)))
```

Services rendered first month:

$$\xrightarrow{(att,com,h1,20)}$$

```
(  transmit (com, att, fee, T | T <= 30 + 8d)
|| ( legal (..., 30, min(30 + 30d,60), 60)
   + transmit (att, com, end, T | 60 <= T)))
```

Take the first branch in $+$ and unfold 'legal':

$$\xrightarrow{\tau}$$

```
(  transmit (com,att,fee,T | T <= 30 + 8d)
|| (transmit (att,com,H,T | 30<T and T<=60).
     (  transmit (com,att,fee,T | T<=60+8d)
     || ( legal (..., 60, min(60 + 30d,60), 60)
         + transmit (att,com,end,T | 60<=T)))))
```

Services rendered second month:

$$\xrightarrow{(att,com,h2,37)}$$

The non-determinism is not constrained to viable options, but will allow any obviously wrong reduction to go wrong at any point. Assuming the desired outcome:

```
(  transmit (com,att,fee,T | T <= 30 + 8d)
|| (  transmit (com,att,fee,T | T <= 60 + 8d)
    || ( legal (...,60,min(60 + 30d,60), 60)
       + transmit (att,com,end,T | 60 <= T))))
```

The next event matches a transmit in the first iteration and a transmit in the second iteration. The contract could reduce properly or fail. We demonstrate the latter. Fee for first month:

$$\xrightarrow{(com,att,fee,38)}$$

```
(  transmit (com,att,fee,T | T <= 30 + 8d)
|| (  Success
    || ( legal (...,60,min(60 + 30d,60), 60)
       + transmit (att,com,end, T | 60 <= T))))
```

At time 39 the whole contract can terminate, because the $30 + 8d$ condition becomes unsatisfiable. Assume that this possibility is exploited. Fee for second month:

$$\xrightarrow{(com,att,fee,62)}$$

Now, there is a serious problem. The choice of matching the first fee was unwise, and the limits of the eager matching semantics shows. The contract can now only fail.

```
(  Failure
|| (  Success
    || ( legal (...,60,min(60 + 30d,60), 60)
        + transmit (att,com,end,T | 60 <= T))))
```

$$\xrightarrow{\tau}$$

```
Failure
```

```
transmit (att, com, H, T | 0 < T and T <= 30).
(   transmit (com, att, fee, T | T <= 30 + 8d)
||   ( legal (..., 30, min(30 + 30d,60), 60)
    + transmit (att, com, end, T | 60 <= T)))
```

Services rendered first month:

$$\xrightarrow{(att,com,h1,20)}$$

```
(   transmit (com, att, fee, T | T <= 30 + 8d)
||   ( legal (..., 30, min(30 + 30d,60), 60)
    + transmit (att, com, end, T | 60 <= T)))
```

We now take the first branch in $+$ and unfold 'legal'

$$\xrightarrow{\tau}$$

```
(  transmit (com,att,fee,T | T <= 30 + 8d)
|| (transmit (att,com,H,T | 30<T and T<=60).
    (  transmit (com,att,fee,T | T <= 60+8d)
    || ( legal (...,60,min(60 + 30d,60), 60)
       + transmit (att,com,end,T | 60<=T)))))
```

Services rendered second month:

$$\xrightarrow{r(att,com,h2,37)}$$

We use explicit directives to point out the transmit we wish to match. Probably, the runtime system already suggested the options available and we picked one leaving the details to the system.

```
(  transmit (com,att,fee,T | T <= 30 + 8d)
|| (  transmit (com,att,fee,T | T <= 60 + 8d)
   || ( legal (...,60,min(60 + 30d,60), 60)
      + transmit (att,com,end,T | 60 <= T))))
```

Fee for first month

$$\xrightarrow{l(com,att,fee,38)}$$

$$\xrightarrow{\tau}$$

This event matches two different transmits, but the decision is taken "by" the directives:

```
(   transmit (com, att, fee, T | T <= 60 + 8d)
||   ( legal (..., 60, min(60 + 30d,60), 60)
    + transmit (att, com, end, T | 60 <= T)))
```

Fee for second month:

$$\xrightarrow{l(com,att,fee,62)}$$

$$\xrightarrow{\tau}$$

```
( legal (..., 60, min(60 + 30d,60), 60)
+ transmit (att, com, end, T | 60 <= T))
```

Attorney signals end-of-contract:

$$\xrightarrow{s(att,com,end,64)}$$

```
Success
```

development, whereas other analyses apply to running contracts. The residuation property allows a contract analysis to be applied at any time (i.e. to any residual contract), and we can thus continuously monitor the execution of the contracts in our portfolio.

Fig. 18 Deferred matching on the legal services agreement

```
transmit (att, com, H, T | 0 < T and T <= 30).
(  transmit (com, att, fee, T | T <= 30 + 8d)
|| ( legal (..., 30, min(30 + 30d,60), 60)
   + transmit (att, com, end, T | 60 <= T)))
```

Services rendered first month:

$$\xrightarrow{(att,com,h1,20)}$$

```
(  transmit (com, att, fee, T | T <= 30 + 8d)
|| ( legal (..., 30, min(30 + 30d,60), 60)
   + transmit (att, com, end, T | 60 <= T)))
```

Services rendered second month:

$$\xrightarrow{(att,com,h2,37)}$$

```
 (  Failure
 || ( legal (..., 30, min(30 + 30d,60), 60)
    + transmit (att, com, end, T | 60 <= T)))
+
 (  transmit (com, att, fee, T | T <= 30 + 8d)
 || ( (  transmit (com,att,fee,T | T<=60+8d)
      || ( legal (...,60,min(60 + 30d,60),60)
           + transmit(att,com,end,T | 60<=T)))
    + Failure))
```

Let us remove the failed parts, i.e.
$C +$ Failure $\rightarrow C$ and $C \parallel$ Failure $\rightarrow$ Failure:

$$\xrightarrow{\tau}$$

```
(   transmit (com,att,fee,T | T <= 30 + 8d)
||  (   transmit (com,att,fee,T | T<=60+8d)
    ||  ( legal (...,60,min(60 + 30d,60),60)
        + transmit(att,com,end,T | 60<=T))))
```

Fee for first month:

$$\xrightarrow{(com,att,fee,38)}$$

```
 (  Success
 || (  transmit (com,att,fee,T | T <= 60 + 8d)
    || ( legal (...,60,min(60 + 30d,60),60)
       + transmit (att,com,end,T | 60 <= T))))
+
 (  transmit (com,att,fee,T | T <= 30 + 8d)
 || (  Success
    || ( legal (...,60,min(60 + 30d,60),60)
       + transmit (att,com,end,T | 60 <= T))))
+
 (  transmit (com,att,fee,T | T <= 30 + 8d)
 || (  transmit (com,att,fee,T | T <= 60 + 8d)
    || ( Failure
       + Failure)))
```

And some more housecleaning, also Success $\parallel C \rightarrow C$:

$$\xrightarrow{\tau}$$

```
 (  transmit (com, att, fee, T | T <= 60 + 8d)
 || ( legal (..., 60, min(60 + 30d,60), 60)
    + transmit (att, com, end, T | 60 <= T)))
+
 (  transmit (com, att, fee, T | T <= 30 + 8d)
 || ( legal (..., 60, min(60 + 30d,60), 60)
    + transmit (att, com, end, T | 60 <= T)))
```

Two continuations are valid at time $T \leq 38$. The first has matched the first month's fee with the first iteration. The second represents matching the first fee with the second iteration. At time 39 the second branch can be rewritten to failure if our algorithm is able to decide that the condition $30 - 8d$ becomes unsatisfiable. But let us leave both branches for now and see what happens. Fee for second month:

$$\xrightarrow{(com,att,fee,62)}$$

This time let us skip the step where all non-matching branches get their own continuation, which is then removed immediately afterwards. Assume that we only attempt a match on the two transmits mentioning the fee:

```
 (  Success
 || ( legal (..., 60, min(60 + 30d,60), 60)
    + transmit (att, com, end, T | 60 <= T)))
+
 (  Failure
 || ( legal (..., 60, min(60 + 30d,60), 60)
    + transmit (att, com, end, T | 60 <= T)))
```

$$\xrightarrow{\tau}$$

```
( legal (..., 60, min(60 + 30d,60), 60)
+ transmit (att, com, end, T | 60 <= T)))
```

At this time a good failure algorithm would detect that the invocation of legal can be reduced to failure. Unfolding 'legal' gives predicates of the form $60 < T$ and $T \leq 60$ on all transmits, hence no event can match in 'legal'.

Attorney signals end-of-contract:

$$\xrightarrow{(att,com,end,64)}$$

```
Success
```

Technically, this is `Success || extra || extra` because we left out 'extra' during reduction. Events can still match the invoices (i.e. the attorney retains the right to invoice any extra hours of service previously rendered). The contract is terminable (nullable) at this point.

Recall that our contract specification language is parameterized over the language of predicates and arithmetic. There is a clear trade-off in play here: a sophisticated language buys expressiveness, but renders most of the analyses undecidable. There is another source of difficulties. Variables may be bound to components of an event that is unknown at the time of analysis. An expression like transmit$(a_1, a_2, R, T \mid \text{true})$. offers little insight into the nature of R unless furnished with a probability vector over all resources.

Here we will circumvent these problems by making do with a restricted predicate language and accepting that analyses may not give answers on all input (but will give correct answers).

The predicate language is plugged in at two locations. In function application $f(\vec{a})$ where all components of the vector $\vec{a}$ must be checked according to the rules of the predicate language, and in $\text{transmit}(a_1, a_2, r, t|P)$ where P must have the type Boolean. As previously we require that a_1, a_2, r, and t are either variables (bound or unbound) or constants. If some components are bound variables or constants, they must be equal to the corresponding components of an incoming event (a'_1, a'_2, r', t') for a match to occur.

Consider the syntax provided in figure 19. In addition to the types Agent, Resource, and Time, the language has the fundamental types Int and Boolean. Take ρ to range over $\{\text{Int}, \text{Time}\}$, take σ to range over $\rho \cup \{\text{Agent}, \text{Resource}\}$, and assume that constants can be uniquely typed (e.g. time constants are in ISO format, and agent and resource constants are disjoint and known).

The language allows arithmetic on integers, simple propositional logic, and manipulation of the two abstract types Resource and Time. Given a time (date) t we may add an integral number of years, months or days. For example $2004.1.1 + 3d + 1y$ yields 2005.1.4. Resources permit a projection on a named component (field) and all fields are of type Int. E.g. to extract the total amount from an information resource named *invoice* we write $\#(\mathit{invoice}, \mathit{total}, t)$ where t is some date[7]. The fields of resources may change over time; hence the third parameter of type Time.

Observables can now be understood simply as fields of a ubiquitous resource named `obs`. An Int may double for a Resource in which case the Int is understood to be a currency amount.

For the denotational semantics of the predicate language we define the following functions mapping syntactic expressions to mathematical objects:

[7] When a resource is introduced into the system through a match, it must be dynamically checked that it possesses the required fields. The set of required fields can be statically determined by a routine type check annotating resources with field names à la $\{\mathit{date}, \mathit{total}, \mathit{paymentdeadline}\}$ Resource. To keep things simple we omit this type extension here.

Fig. 19 Example syntax for predicate language

$$\frac{\Delta(var) = \sigma}{\Delta \vdash var : \sigma} \qquad \frac{\text{type}(const) = \sigma}{\Delta \vdash const : \sigma}$$

$$\frac{\Delta \vdash e_1 : \text{Int} \quad \Delta \vdash e_2 : \text{Int} \quad \text{op} \in \{+, -, *, /\}}{\Delta \vdash e_1 \text{ op } e_2 : \text{Int}}$$

$$\frac{\Delta \vdash t : \text{Time} \quad \Delta \vdash e : \text{Int} \quad \text{f} \in \{\texttt{y}, \texttt{m}, \texttt{d}\} \quad \text{op} \in \{+, -\}}{\Delta \vdash t \text{ op } e \text{ f} : \text{Time}}$$

$$\frac{\Delta \vdash e : \text{Time} \quad \text{f} \in \{\texttt{y}, \texttt{m}, \texttt{d}\}}{\Delta \vdash e\#\text{f} : \text{Int}} \qquad \frac{\Delta \vdash e : \text{Int}}{\Delta \vdash e : \text{Resource}}$$

$$\frac{\Delta \vdash r : \text{Resource} \quad \Delta \vdash t : \text{Time} \quad f \in fields(r)}{\Delta \vdash \#(r, f, t) : \text{Int}} \qquad \frac{\Delta \vdash b : \text{Boolean}}{\Delta \vdash \texttt{not } b : \text{Boolean}}$$

$$\frac{\Delta \vdash e_1 : \rho \quad \Delta \vdash e_2 : \rho}{\Delta \vdash e_1 < e_2 : \text{Boolean}} \qquad \frac{\Delta \vdash e_1 : \sigma \quad \Delta \vdash e_2 : \sigma}{\Delta \vdash e_1 = e_2 : \text{Boolean}}$$

$$\frac{\Delta \vdash b_1 : \text{Boolean} \quad \Delta \vdash b_2 : \text{Boolean} \quad \text{op} \in \{\texttt{and}, \texttt{or}\}}{\Delta \vdash b_1 \text{ op } b_2 : \text{Boolean}}$$

$$\begin{aligned}
&\mathcal{E} : Exp \to \nabla \to (\text{Agent} \cup \text{Resource} \cup \text{Int} \cup \text{Time})\\
&\mathcal{B} : Bexp \to \nabla \to \{t, f\}
\end{aligned}$$

where we assume the following mathematical environment:

- ∇ is the set of all possible bindings δ of variables to values.
- *Exp* is the set of all possible expressions of type Int, Time, Resource or *Agent* in the language.
- *Bexp* is the set of all possible expressions of type Boolean in the language.
- Resource and Agent are the sets of resources and agents respectively.
- Int $= \mathbb{Z}$
- Time $= \{\ldots, -2_t, -1_t, 0_t, 1_t, 2_t, \ldots\}$ where operators $+$ and $-$ have the obvious interpretations, and we have the map $(\cdot)_t : \mathbb{Z} \to$ Time defined by $(n)_t = n_t$.
- Int $\subseteq$ Resource

- Agent, Resource, and Time are pairwise disjoint.
- (Agent $\cup$ Resource $\cup$ Int $\cup$ Time) is equipped with an (non-total) order $<$ that is the union of the orders of the participating sets. Assume that Int and Time have the usual orderings.
- $\wedge$, $\vee$, and $\neg$ serve as logical operators with the usual meaning over the set $\{t, f\}$.
- If a and b are integers, $a \div b$ gives the the largest integer c such that $c \cdot b \leq a$. mod is the corresponding modulo function so that $c \cdot b + a \bmod b = a$.
- φ : Resource $\times$ Field $\times$ Time $\rightarrow$ Int is a projection function on resources, and Field is a set of static field identifiers.

A contract analysis is a map from a syntactic description of a contract and some auxiliary information to a domain of our choice. The auxiliary information is often an agent or a point in time that the analysis should be relative to or an estimate of the probabilities associated with an underlying process. Ideally, a contract analysis can be performed *compositionally.* This section contains two simple analyses with this property. Space considerations prevent a walkthrough of more involved examples, but the basic idea should be clear. We will assume for simplicity that recursively defined contracts are *guarded.* The analyses are presented using inference systems defined by induction on syntax, emphasizing the declarative and compositional nature of the analyses.

5.1 Example: Failed Contracts

A contract may accept a sequence of one of more events that is not a prefix of a performing trace. Thus the residual contract is failed and its denotation is the empty set – the contract is in an inconsistent state. The inference rules provided in Figure 21 sketch how one could go about detecting this. The focal point is being able to decide if a predicate P can not hold true for any future values of its parameters. In practice, this often amounts to a simple argument: A deadline has been passed.

We have referred to the *failed* analysis numerous times in the example reductions. In section 4 we saw that eager matching made a bad choice, which was not detected until much later. The failure analysis seeks to alleviate such situations as early as possible. Consider the scenario in Figure 22 for an example under the eager matching regime. The failure of the contract is detected as soon as there is no remedy, i.e. at $T = 39$.

Fig. 20 Denotational semantics for predicate language

$$
\begin{aligned}
\mathcal{E}[\![const]\!] &= \lambda\delta \in \nabla.const \\
\mathcal{E}[\![var]\!] &= \lambda\delta \in \nabla.\delta(var) \\
\mathcal{E}[\![e_1 + e_2]\!] &= \lambda\delta \in \nabla.\mathcal{E}[\![e_1]\!]\delta + \mathcal{E}[\![e_2]\!]\delta \\
\mathcal{E}[\![e_1 - e_2]\!] &= \lambda\delta \in \nabla.\mathcal{E}[\![e_1]\!]\delta - \mathcal{E}[\![e_2]\!]\delta \\
\mathcal{E}[\![e_1 * e_2]\!] &= \lambda\delta \in \nabla.\mathcal{E}[\![e_1]\!]\delta \cdot \mathcal{E}[\![e_2]\!]\delta \\
\mathcal{E}[\![e_1 / e_2]\!] &= \lambda\delta \in \nabla.\mathcal{E}[\![e_1]\!]\delta \div \mathcal{E}[\![e_2]\!]\delta \\
\mathcal{E}[\![e\#\mathtt{d}]\!] &= \lambda\delta \in \nabla.\mathcal{E}[\![e]\!]\delta \bmod 30 \\
\mathcal{E}[\![e\#\mathtt{m}]\!] &= \lambda\delta \in \nabla.\mathcal{E}[\![e]\!]\delta \div 30 \bmod 12 \\
\mathcal{E}[\![e\#\mathtt{y}]\!] &= \lambda\delta \in \nabla.\mathcal{E}[\![e]\!]\delta \div 360 \\
\mathcal{E}[\![e + f\ \mathtt{d}]\!] &= \lambda\delta \in \nabla.\mathcal{E}[\![e]\!]\delta + (\mathcal{E}[\![f]\!]\delta)_t \\
\mathcal{E}[\![e + f\ \mathtt{m}]\!] &= \lambda\delta \in \nabla.\mathcal{E}[\![e]\!]\delta + (\mathcal{E}[\![f]\!]\delta \cdot 30)_t \\
\mathcal{E}[\![e + f\ \mathtt{y}]\!] &= \lambda\delta \in \nabla.\mathcal{E}[\![e]\!]\delta + (\mathcal{E}[\![f]\!]\delta \cdot 360)_t \\
\mathcal{E}[\![e - f\ \mathtt{d}]\!] &= \lambda\delta \in \nabla.\mathcal{E}[\![e]\!]\delta - (\mathcal{E}[\![f]\!]\delta)_t \\
\mathcal{E}[\![e - f\ \mathtt{m}]\!] &= \lambda\delta \in \nabla.\mathcal{E}[\![e]\!]\delta - (\mathcal{E}[\![f]\!]\delta \cdot 30)_t \\
\mathcal{E}[\![e - f\ \mathtt{y}]\!] &= \lambda\delta \in \nabla.\mathcal{E}[\![e]\!]\delta - (\mathcal{E}[\![f]\!]\delta \cdot 360)_t
\end{aligned}
$$

$$
\mathcal{E}[\![\#(r, f, t)]\!] = \lambda\delta \in \nabla.\varphi(\mathcal{E}[\![r]\!]\delta, f, \mathcal{E}[\![t]\!]\delta)
$$

$$
\begin{aligned}
\mathcal{B}[\![e_1 < e_2]\!] &= \lambda\delta \in \nabla.\begin{cases} t & \text{if } \mathcal{E}[\![e_1]\!]\delta < \mathcal{E}[\![e_2]\!]\delta \\ f & \text{otherwise} \end{cases} \\
\mathcal{B}[\![e_1 = e_2]\!] &= \lambda\delta \in \nabla.\begin{cases} t & \text{if } \mathcal{E}[\![e_1]\!]\delta = \mathcal{E}[\![e_2]\!]\delta \\ f & \text{otherwise} \end{cases} \\
\mathcal{B}[\![b_1 \mathtt{and}\ b_2]\!] &= \lambda\delta \in \nabla.\mathcal{B}[\![b_1]\!]\delta \wedge \mathcal{B}[\![b_2]\!]\delta \\
\mathcal{B}[\![b_1 \mathtt{or}\ b_2]\!] &= \lambda\delta \in \nabla.\mathcal{B}[\![b_1]\!]\delta \vee \mathcal{B}[\![b_2]\!]\delta \\
\mathcal{B}[\![\mathtt{not}\ b]\!] &= \lambda\delta \in \nabla.\neg\mathcal{B}[\![b]\!]\delta
\end{aligned}
$$

Fig. 21 Failed contracts

$$\frac{\forall \delta', \forall t' \geq t : (\delta \oplus \delta' \oplus T \mapsto t' \models \neg P)}{\mathrm{D}, \delta, t \vdash \text{transmit}(\vec{X}T \mid P).\, c \text{ failed}}$$

$$\frac{\mathrm{D}, \delta, t \vdash c \text{ failed}}{\mathrm{D}, \delta, t \vdash \text{transmit}(\vec{X}T \mid P).\, c \text{ failed}}$$

$$\mathrm{D} \vdash \text{Failure failed} \qquad \frac{\mathrm{D}, \delta, t \vdash c \text{ failed} \quad \mathrm{D}, \delta, t \vdash c' \text{ failed}}{\mathrm{D}, \delta, t \vdash c + c' \text{ failed}}$$

$$\frac{\mathrm{D}, \delta, t \vdash c \text{ failed}}{\mathrm{D}, \delta, t \vdash c \parallel c' \text{ failed}} \qquad \frac{\mathrm{D}, \delta, t \vdash c' \text{ failed}}{\mathrm{D}, \delta, t \vdash c \parallel c' \text{ failed}}$$

$$\frac{\mathrm{D}, \delta, t \vdash c \text{ failed}}{\mathrm{D}, \delta, t \vdash c; c' \text{ failed}} \qquad \frac{\mathrm{D}, \delta, t \vdash c' \text{ failed}}{\mathrm{D}, \delta, t \vdash c; c' \text{ failed}}$$

$$\frac{\mathrm{D}, \delta, t \vdash c \text{ failed} \quad (f(\vec{X}) = c) \in \mathrm{D}}{\mathrm{D}, \delta, t \vdash f(\vec{a}) \text{ failed}}$$

Fig. 22 Example: Failed legal services agreement under eager matching (non-deterministic)

```
transmit (att,com,H,T | 0 < T and T<=30).   (att, com, h1, 20),   (  transmit (com,att,fee,T | T<=30+8d)
( transmit (com,att,fee,T | T <= 30+8d)     (att, com, h2, 37),   || (  Success
|| ( legal (...,30,min(30 + 30d,60),60)     (com, att, fee, 38)      || (legal(...,60,min(60 + 30d,60),60)
   + transmit (att,com,end,T | 60<=T)))             -->                 + transmit(att,com,end,T|60<=T))))
```

We would rather not wait for the next event $\xrightarrow{(com,att,fee,62)}$ before realizing that the situation is not working. As soon as $T = 39$, `transmit (com att, fee, T | T <= 30 + 8d)` can transition to `Failure`. The relevant part of the derivation looks like this:

$$\dfrac{\dfrac{D, d, 39 \vdash 39 \leq 30 + 8d}{D, d, 39 \vdash \texttt{transmit (com att, fee, T | T <= 30 + 8d)}\ \mathit{failed}}}{D, d, 39 \vdash \begin{array}{l} \texttt{(\ \ transmit (com att, fee, T | T <= 30 + 8d)} \\ \texttt{|| (\ \ \ Success} \\ \texttt{\ \ \ ||\ \ (legal (..., 60, min(60 + 30d,60), 60)} \\ \texttt{\ \ \ \ \ \ \ \ + transmit (att, com, end, T | 60 <= T)))} \end{array}\ \mathit{failed}}$$

5.2 *Example: Task List*

Given a contract or a portfolio of contracts it is tremendously important for an agent to know when and how to act. To this end we demonstrate how a very simple *task list* can be compiled.

Consider the definition given in Figure 23. The function gives returns a list of outstanding commitments that can be carried out at time t. We only admit interval conditions of the form $a \leq T$ `and` $T \leq b$ with T being the time variable in the enclosing transmit, since in "real" contracts hardly anything else is used. It is important to notice that the result of the analysis may be incomplete. A task is only added if the agents agree (i.e. `a = a1`), but if `a1` is not bound at the time t of analysis, the task is simply skipped. A more elaborate dataflow analysis might reveal that in fact `a1` is always bound to `a`.

Also notice the case for application $f(\vec{a})$. We expand the body of the named contract `f` given arguments `a` but only once (assuming f is guarded). This measure ensures termination of the analysis, but reduces the function's look-ahead horizon. Hence, any task or point of interest more than one recursive unfolding away is not detected. This is unlikely to have practical significance for two reasons: (1) recursively defined contracts are guarded and so a `transmit` must be matched before a new unfold can occur. This `transmit` therefore is presumably more relevant than any other `transmits` further down the line; (2) it would be utterly unidiomatic if some `transmit` t_1 was required to be matched before another `transmit` t_2, but nevertheless had a later deadline than that of t_2.

For an example of the task list analysis, we return to the Legal Services Agreement. The task list works best with eager matching with explicit reduction control. Eager matching alone is too careless, and deferred matching represents many states, which are all assumed valid, but may confuse the user when he or she sees overlapping tasks for every hypothetical state of the contract. Consider Figure 24 for an example of how the task list evolves under reduction of the Legal Services Agreement.

The examples given above, in their simplicity, may be extended given knowledge of the problem domain. In particular, knowledge of or forecasting about probable event sequences may be used in a manner orthogonal to the coding of analyses by appropriate function calls.

Analyses possible to implement in this way include:

- Resource flow forecasting (supply requirements).
- Terminability by agent, latest termination, earliest termination.
- Valuation, or simply put: What is the value to an agent of a given contract? The analysis is fairly intricate and requires knowledge of financial models and stochastic processes. Interested readers are referred to Peyton Jones and Eber [JES00,JE03] who provide a very readable introduction targeted at computer

Fig. 23 Task list analysis

$$\mathrm{D}, \delta, \mathrm{a}, t \vdash \text{Success} \ : \ [] \qquad \mathrm{D}, \delta, \mathrm{a}, t \vdash \text{Failure} \ : \ []$$

$$\frac{\models a \neq a_1 \qquad \vec{X} = (a_1, a_2, R, T)}{\mathrm{D}, \delta, \mathrm{a}, t \vdash \text{transmit}(\vec{X} \mid x \leq T \ \textbf{and} \ T \leq y).\, c \ : \ []}$$

$$\frac{\models \neg(x \leq t \ \textbf{and} \ t \leq y)}{\mathrm{D}, \delta, \mathrm{a}, t \vdash \text{transmit}(\vec{X} \mid x \leq T \ \textbf{and} \ T \leq y).\, c \ : \ []}$$

$$\frac{\models a = a_1 \qquad \vec{X} = (a_1, a_2, R, T) \qquad \models x \leq t \ \textbf{and} \ t \leq y}{\begin{array}{c}\mathrm{D}, \delta, \mathrm{a}, t \vdash \text{transmit}(\vec{X} \mid x \leq T \ \textbf{and} \ T \leq y).\, c \ : \\ {[\text{transmit}(\vec{X} \mid x \leq T \ \textbf{and} \ T \leq y).\, c]}\end{array}}$$

$$\frac{\mathrm{D}, \delta, \mathrm{a}, t \vdash c_1 \ : \ l_1 \quad \mathrm{D}, \delta, \mathrm{a}, t \vdash c_2 \ : \ l_2}{\mathrm{D}, \delta, \mathrm{a}, t \vdash c_1 + c_2 \ : \ l_1 \ @ \ l_2}$$

$$\frac{\mathrm{D} \vdash c_1 \text{ nullable} \qquad \mathrm{D}, \delta, \mathrm{a}, t \vdash c_1 \ : \ l_1 \qquad \mathrm{D}, \delta, \mathrm{a}, t \vdash c_2 \ : \ l_2}{\mathrm{D}, \delta, \mathrm{a}, t \vdash c_1; c_2 \ : \ l_1 \ @ \ l_2}$$

$$\frac{\mathrm{D} \nvdash c_1 \text{ nullable} \qquad \mathrm{D}, \delta, \mathrm{a}, t \vdash c_1 \ : \ l_1}{\mathrm{D}, \delta, \mathrm{a}, t \vdash c_1; c_2 \ : \ l_1}$$

$$\frac{\mathrm{D}, \delta, \mathrm{a}, t \vdash c_1 \ : \ l_1 \qquad \mathrm{D}, \delta, \mathrm{a}, t \vdash c_2 \ : \ l_2}{\mathrm{D}, \delta, \mathrm{a}, t \vdash c_1 \parallel c_2 \ : \ l_1 \ @ \ l_2}$$

$$\frac{(f(\vec{X}) = c) \in \mathrm{D} \qquad \mathrm{D}, \delta, \mathrm{a}, t \vdash c \ : \ l}{\mathrm{D}, \delta, \mathrm{a}, t \vdash f(\vec{a}) \ : \ l}$$

scientists.

- General model checking for business rules: (a) static (b) dynamic/runtime (Timed LTL checking), cf. [KPA04].

Fig. 24 Task list for the Legal Services Agreements under eager matching with explicit control

```
transmit (att,com,H,T | 0 < T and T <= 30).
(   transmit (com,att,fee,T | T <= 30 + 8d)
||  ( legal (...,30,min(30 + 30d,60),60)
    + transmit (att,com,end,T | 60 <= T)))
```

Services rendered first month:
$\xrightarrow{(att,com,h1,20)}$
$\xrightarrow{\tau}$

$T = 0$:
```
att: transmit(att,com,H,T | 0<T and T<=30)
```

```
(  transmit (com,att,fee,T | T<=30+8d)
|| (transmit (att,com,H,T | 30<T and T<=60).
    (  transmit (com,att,fee,T | T<=60+8d)
    || ( legal (...,60,min(60 + 30d,60),60)
       + transmit(att,com,end,T | 60<=T)))))
```

Services rendered second month:
$\xrightarrow{r(att,com,h2,37)}$

$T = 20$:
```
com: [transmit(com,att,fee,T | T<=30+8d)]
```

$T = 31$:
```
att: transmit(att,com,H,T | 30<T and T<=60)
com: transmit(com,att,fee,T | T<=30+8d)
```

```
(  transmit (com,att,fee,T | T<=30+8d)
|| (  transmit (com,att,fee,T | T<=60+8d)
   || ( legal (...,60,min(60 + 30d,60),60)
      + transmit(att,com,end,T | 60<=T))))
```

Fee for first month:
$\xrightarrow{l(com,att,fee,38)}$
$\xrightarrow{\tau}$

$T = 37$:
```
com: transmit (com,att,fee,T | T<=30+8d)
com: transmit (com,att,fee,T | T<=60+8d)
```

If the system was unable to decide predicates, two additional tasks would have been shown for `att`:

```
att: transmit (att,com,H,T | 60<T and T<=60)
att: transmit (att,com,end,T | 60<=T)
```

```
(   transmit (com,att,fee,T | T <= 60 + 8d)
||  ( legal (...,60,min(60 + 30d,60),60)
    + transmit (att,com,end,T | 60 <= T)))
```

Fee for second month:
$\xrightarrow{l(com,att,fee,62)}$
$\xrightarrow{\tau}$

$T = 38$:
```
com: transmit (com,att,fee,T | T<=60+8d)
```

$T = 60$:
```
att: transmit (att,com,end,T | 60<=T)
com: transmit (com,att,fee,T | T<=60+8d)
```

```
( legal (...,60,min(60 + 30d,60),60)
+ transmit (att,com,end,T | 60 <= T))
```

Attorney signals end-of-contract:
$\xrightarrow{s(att,com,end,64)}$

$T = 62$:
```
att: transmit (att, com, end, T | 60 <= T)
```

```
Success
```

$T \geq 64$: `No tasks!`

6 Discussion and Future Work

Our definition of contracts focuses on contracts as classifiers of event traces into performing and nonperforming ones. This is coarse, and many real-world issues are left out—not for good, but for now.

The basic idea is to develop these notions within a general framework that may require specifications of runtime environment and protocols for event transmission. The inclusion of explicit operators in the language to mimic many standard steps in the contract lifecycle—say checking a contract for potential problems with current law—would not facilitate easy contract coding without both static ("does this contract conform to standard practice?") and dynamic ("is this sequence of events and their handling proper?") checks appealing to some enclosing structures.

We decided to pursue compositionality—hierarchical specification—from the outset as a central notion and thus follow a process algebra approach, basically to evaluate how far that would take us in the given domain. This can be contrasted to a network-oriented approach supported by suitable diagramming to appeal to visual faculties, which appears to be the preferred modeling approach for workflow systems (Petri nets) [vdAvH02] and in object-oriented analysis (UML diagramming). Note that hierarchical specification is also needed in a network-oriented approach to achieve modular description and reuse of specification components. Furthermore, powerful specification mechanisms such as functional abstraction and (non-tail) recursion have no simple visual representations.

The Software Development Agreement (Figure 15) provides a good setting to observe the limitations to our approach and the ramifications of the design choices made.

The Change Order is not coded. It might be cleverly coded in the current language, again using constraints on the events passed around, but a more natural way would be using higher-order contracts, i.e. contracts taking contracts as arguments. Thus, a Change Order would simply be the passing back and forth of a contract followed by an instantiation upon agreement.

The transmission of rights can easily be coded, but the prohibition to transmit a particular resource affects all other contracts. Currently, we have no construct available to handle this situation.

Contracts often specify certain things that are not to be done (e.g. not copying the software). Such restrictions should intersect all other outstanding contracts and limit them appropriately. A higher-order language or predicates that could guard all `transmit`s of an entire subcontract might ameliorate this in a natural way.

A fuller range of language constructions that programmers are familiar with is also desirable; in the present incarnation of the contract language, several standard constructions have been left out in order to emphasize the core event model. In practice, conditionals and various sorts of lambda abstractions would make the language easier to use, though not strictly more expressive, as they can be encoded through events, albeit in a non-intuitive way. A conditional that is *not* driven by events (i.e. an if-then-else) seems to be needed for natural coding in many real-world contracts. Also, a catch-throw mechanism for unexpected events would make contracts more robust.

Conversely, certain features of the language appear to be almost too strong for the domain; the inclusion of full recursion means that contracts active for an unlimited period of time, say leases, are easy to code, but make contract analysis significantly harder. In practice, contracts running for “unlimited” time periods often have external constraints (usually local legislation) forcing the contract to be reassessed by its parties, and possibly government representatives, from time to time. Having only a restricted form of recursion that suffices for most practical applications should simplify contract analysis.

The expressivity of the contract language and indeed the feasibility of non-trivial contract analysis depends heavily on the predicate language used. Predicates restricted to the form $[a; b]$ are surely too limited, and further investigation into the required expressiveness of the predicate language is desirable.

While the language is parametrized over the predicate language used, almost all real-world applications will require some model of time and timed events to be incorporated*vis-à-vis* the examples using interval in Section 5 . The current event model allows for encoding through the predicate language, but an extended set of events, with companion semantics, would make for easier contract programming; timer (or “trigger”) events appear to be ubiquitous when encoding contracts.

7 Related Work

The impetus for this work comes from two directions: the REA accounting model pioneered by McCarthy [McC82] and Peyton Jones, Eber and Seward’s seminal article on specification of financial contracts [JES00]. Furthermore, given that contracts specify protocols as to how parties bound by them are to interact with each other there are links to process and workflow models.

7.1 Composing Contracts

Peyton Jones, Eber and Seward [JES00] present a compositional language for specifying financial contracts. It provides a decomposition of known standard contracts such as zero coupon bonds, options, swaps, straddles, etc., into individual payment commitments that are combined declaratively using a small set of contract combinators. All contracts are two-party contracts, and the parties are implicit. The combinators (taken from [JE03], revised from [JES00]) correspond to $\text{Success}, \cdot \parallel \cdot, \cdot + \cdot, \text{transmit}(\cdot)$ of our language $\mathcal{C}^{\mathcal{P}}$; it has no direct counterparts to $\text{Failure}, \cdot ; \cdot$ nor, most importantly, recursion or iteration. On the other hand, it provides conditionals and predicates that are applicable to arbitrary contracts, not just commitments as in $\mathcal{C}^{\mathcal{P}}$, something we have found to be worthwhile also for specifying commercial contracts. Furthermore, their language provides an `until`-operator that allows a party to terminate a contract successfully at a particular time, even if not all commitments have been satisfied. Using `until` for contract specification seems difficult, however, since it may—legally—cut off contract execution before all reciprocal commitments have been satisfied, e.g., the requirement to pay for a service that has been rendered.

Our contract language generalizes financial payment commitments to arbitrary transfers of resources and information, provides explicit agents and thus provides the possibility of specifying multi-party contracts.

We have provided a denotational semantics for $\mathcal{C}^{\mathcal{P}}$ and developed operational semantics for contract monitoring from it, whereas Peyton Jones, Eber and Seward focus on *valuation*, a sophisticated contract analysis based on stochastic analysis for pricing contracts.

7.2 Resources/Events/Agents (REA)

McCarthy [McC82] pioneered REA, an accounting model that focuses on the basic transaction patterns of the enterprise, the exchange of scarce goods and the transformation of resources by production, and separates it from phenomena that can be derived by aggregation or other means. Geerts and McCarthy [GM00] complement REA's entity-relationship model of basic *ex-post* notions of *events*, in which *agents* transmit scarce *resources*, with *ex-ante* notions: commitments and sets of commitments making up contracts. [8] Contracts, however, are only modeled as sets of commitments whose concrete terms and constraints are usually described in natural language and as such live outside the scope of the entity-relationship model. Our work provides a formalization for contracts and their (performing) executions and thus complements the REA's data-centered notions with a well-defined process perspective.

[8] This is a highly simplified description of key parts of REA.

Disregarding the structure of events and their temporal properties, $\mathcal{C}^{\mathcal{P}}$ is basically a process algebra. It corresponds to Algebra of Communicating Processes (ACP) with deadlock (Failure), free merge ($\cdot \parallel \cdot$) and recursion, but without encapsulation [BW90]. Note that contracts are to be thought of as exclusively *reactive* processes, however: they respond to externally generated events, but do not autonomously generate them. This leads naturally to contracts classifying event traces, making CSP [BHR84,Hoa85] and its trace-theoretic semantics a natural conceptual framework for our view-independent approach to contract specification. This is in contrast to CCS-like process calculi [Hen88,Mil89,Mil99], which take a rather operational process-as-machine view; they treat communication as dual pairs of send and receive messages and allow observation of branching decisions in processes. Note that $\mathcal{C}^{\mathcal{P}}$, as presented here, contains no synchronization between concurrently executing subcontracts. A previous version of $\mathcal{C}^{\mathcal{P}}$ contained the contract conjunction operator $c \ \& \ c'$, whose denotational semantics is

$$\mathcal{C}[\![c \ \& \ c']\!]^{\mathrm{D};\delta} = \mathcal{C}[\![c]\!]^{\mathrm{D};\delta} \cap \mathcal{C}[\![c']\!]^{\mathrm{D};\delta}.$$

This is the parallel composition operator of CSP with synchronization at each step. A trace satisfies $c \ \& \ c'$ if it satisfies both c and c'. This makes it possible to specify a contract by providing a *basic* specification, c (sales order), and refining it by conjoining it with an additional *policy*, c' (no alcohol must be sold to minors), that a correct contract execution must satisfy. Our language can be extended to include contract conjunction. We have not included it here to keep the theoretical treatment of $\mathcal{C}^{\mathcal{P}}$ simple. Furthermore, it is our impression that the above asymmetry of c specifying the fundamental protocol for contract execution and c' filtering illegal executions may be better captured by formulating policies logically, e.g., in Linear-Time Logic (LTL), possibly enforced by run-time verification [KPA03].

There are numerous timed variants of process algebras and temporal logics; see e.g. Baeten and Middelburg [BM02] for timed process algebras. It should be noted that our contract language is fundamentally deterministic to avoid misunderstanding between contract partners: by design, nondeterministic implicit control decisions as in CCS-based process calculi are avoided. Indeed the eager matching semantics presented can be considered a process language with implicit control decisions (a process may evolve nondeterministically and autonomously). Since this is considered undesirable in our context (though realistic as it reflects the matching ambiguities common in bookkeeping), its events (actions in process terminology) are beefed up with control ("routing") information to control process/contract evolution deterministically.

Note that, in contrast to conventional process calculi, we have included both sequential composition and parameterized recursion to support a separation of data (the base language) and control (the contract language).

Also, our base language is not fixed, but a parameter of the contract language so as to accommodate expressing temporal (and other) constraints modularly and "naturally". Indeed, the basic structure of events can be entirely encapsulated in the base language, making the technical development of the contract language (the "control part") independent of REA or other data models for that matter.

Timed process calculi tend to build on rudimentary models of time. These appear to be insufficient for expressing contract constraints naturally, but may turn out to be viable as core languages. Clearly, studying timing more closely as well as other connections to process calculi constitutes requisite future work.

Finally, most of the extant process algebras apparently do not consider the approach of contract monitoring by residuation. In this paper, the need for considering (prefixes of) *event traces* leads to the problem of allowing only contracts that ensure that the arrival of any event leads to a well-defined residual contract. Calculi such as CCS do not have a notion of *event* traces, and do not encounter the problem, since the (structural) operational semantics turns out to be sound and complete for the set of structural equivalences defining a "program" in CCS. The main difference seems to be the liberal recursion operator employed in our language which admits mutual recursion, unlike CCS where the constructs of equal strength only admit transitions that are *syntactically* guarded in the sense that if an operator has a transition to a new term, the root of that term contains an operator of "lower" strength (e.g. the "replication" operator is guarded by the "parallel" operator in CCS).

7.4 Work flow and business process languages

In [SMTA95] an *event algebra* is developed which is used to monitor a discrete event system. The terms of the algebra contain the equivalent of Success, Failure, $\cdot \parallel \cdot, \cdot + \cdot, \cdot ; \cdot$ while the atomic contract transmit$(\cdot).\cdot$ is replaced by an enumerated set of unique atomic constructs with no free variables. Iteration is stated to be done by instantiating terms such that atomic terms are relabeled to ensure uniqueness of all atomic terms. A trace semantics is given for terms as well as *residuation equations*. The equations allow monitoring of terms by a syntactic method like in $\mathcal{C}^{\mathcal{P}}$. Guardedness (in the sense of $\mathcal{C}^{\mathcal{P}}$) is guarenteed by excluding recursion from the language. It is not entirely clear how iteration is included in the language as no formal description of it is given. The residuation equations given, essentially implement the *eager* semantics of $\mathcal{C}^{\mathcal{P}}$.

Another branch of research has focused on the specification and modelling of business processes. In this vein, the *Business Process Modelling Language* (BPML) is an XML-inspired specification language defined by a consortium of agents from industry and reported in several white papers and technical reports [Ark02,vdADtHW02]. A "program" in the language is, essentially, an XML schema containing process specifications, including temporal and conditional statements, as well as a restricted iteration construct ("repeat"). The scope of entities that can reasonably be modelled by BPML is conceptually larger than the one considered in this paper, since arbitrary (internal or external) processes and commitments can be modelled – hence also contractual obligations. However, while the language operates with an execution model loosely based on π-calculus [MPW92], a proper (and formal) semantics for process execution, performance and monitoring is lacking. The semantics of the framework is currently described only in terms of natural language, and any kind of *safe* automated or formal analysis of execution of processes specified in the language thus cannot be performed at present.

8 Acknowledgements

This work has been partially funded by the NEXT Project, which is a collaboration between Microsoft Business Solutions, The IT University of Copenhagen (ITU) and the Department of Computer Science at the University of Copenhagen (DIKU). See `http://www.itu.dk/next` for more information on NEXT.

We would like to thank Simon Peyton Jones and Jean-Marc Eber for valuable discussions on modeling financial contracts. Kasper Østerbye, Jesper Kiehn and the members of the NEXT Working Group have provided helpful comments and feedback on extending the work of Peyton Jones and Eber to commercial contracts based on the REA accounting model. Indeed, Kasper has worked out similar ideas on representing contracts as ours, but in an object-oriented setting.

A Full Proofs

Theorem 1 Let $\mathrm{D} = \{f_i[\vec{X_i}] = c_i\}_{i=1}^{m}$ and δ be given. We prove

$$\mathcal{C}\llbracket c \rrbracket^{\gamma;\delta\oplus\delta'} = \{s : \delta' \vdash^{\delta}_{\mathrm{D}} s : c\}$$

where $\gamma = \mathcal{D}\llbracket \mathrm{D} \rrbracket^{\delta}$.

"$\supseteq$": Define $\delta' \models^{\delta}_{\mathrm{D}} s : c \iff s \in \mathcal{C}\llbracket c \rrbracket^{\gamma;\delta\oplus\delta'}$. We prove by induction on the derivation of $\delta' \vdash^{\delta}_{\mathrm{D}} s : c$ that $\delta' \models^{\delta}_{\mathrm{D}} s : c$.

$\boxed{\delta' \vdash^{\delta}_{\mathrm{D}} \langle\rangle : \text{Success}}$ We need to show that $\delta' \models^{\delta}_{\mathrm{D}} \langle\rangle : \text{Success}$. This follows immediately from $\mathcal{C}\llbracket \text{Success} \rrbracket^{\gamma;\delta\oplus\delta'} = \{\langle\rangle\}$.

$\boxed{\dfrac{\vec{X} \mapsto \vec{v} \vdash^{\delta}_{\mathrm{D}} s : c \quad (f(\vec{X}) = c) \in \mathrm{D}, \vec{v} = \mathcal{Q}\llbracket \vec{a} \rrbracket^{\delta\oplus\delta'}}{\delta' \vdash^{\delta}_{\mathrm{D}} s : f(\vec{a})}}$ Assume $\vec{X} \mapsto \vec{v} \models^{\delta}_{\mathrm{D}} s : c$ (induction hypothesis) with $\vec{v} = \mathcal{Q}\llbracket \vec{a} \rrbracket^{\delta\oplus\delta'}$ and $(f(\vec{X}) = c) \in \mathrm{D}$. We need to show that $\delta' \models^{\delta}_{\mathrm{D}} s : f(\vec{a})$.

By definition we have

$$\begin{aligned} \mathcal{C}\llbracket f(\vec{a}) \rrbracket^{\gamma;\delta\oplus\delta'} &= \gamma(f)(\mathcal{Q}\llbracket \vec{a} \rrbracket^{\delta\oplus\delta'}) \\ (\text{by def. of } \vec{v}) &= \gamma(f)(\vec{v}) \\ (\text{by def. of } \gamma) &= \mathcal{C}\llbracket c \rrbracket^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}} \end{aligned}$$

and thus, since $\vec{X} \mapsto \vec{v} \models^{\delta}_{\mathrm{D}} s : c$ by induction hypothesis, we can conclude that $\delta' \models^{\delta}_{\mathrm{D}} s : f(\vec{a})$.

$\boxed{\dfrac{\delta \oplus \delta'' \models P \quad \delta'' \vdash^{\delta}_{\mathrm{D}} s : c \qquad (\delta'' = \delta' \oplus \{\vec{X} \mapsto \vec{v}\})}{\delta' \vdash^{\delta}_{\mathrm{D}} \text{transmit}(\vec{v})\, s : \text{transmit}(\vec{X}|P).\, c}}$ Assume $\delta \oplus \delta'' \models P$ and $\delta'' \models^{\delta}_{\mathrm{D}} s : c$ where $\delta'' = \delta' \oplus \{\vec{X} \mapsto \vec{v}\}$. We need to show that $\delta' \models^{\delta}_{\mathrm{D}} \text{transmit}(\vec{v})\, s : \text{transmit}(\vec{X}|P).\, c$.

Since $\delta \oplus \delta'' \models P$ and $\delta'' \models^{\delta}_{\mathrm{D}} s : c$ it follows immediately from the definition of $\mathcal{C}\llbracket \text{transmit}(\vec{X} \mid P).\, c \rrbracket^{\gamma;\delta\oplus\delta'}$ that $\delta' \models^{\delta}_{\mathrm{D}} \text{transmit}(\vec{v})\, s : \text{transmit}(\vec{X}|P).\, c$.

$\boxed{\dfrac{\delta' \vdash^{\delta}_{\mathrm{D}} s_1 : c_1 \quad \delta' \vdash^{\delta}_{\mathrm{D}} s_2 : c_2 \quad (s_1, s_2) \rightsquigarrow s}{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_1 \parallel c_2}}$ Assume $\delta' \models^{\delta}_{\mathrm{D}} s_1 : c_1$, $\delta' \models^{\delta}_{\mathrm{D}} s_2 : c_2$ and $(s_1, s_2) \rightsquigarrow s$. We need to show that $\delta' \models^{\delta}_{\mathrm{D}} s : c_1 \parallel c_2$.

From the assumptions and the definition of $\mathcal{C}\llbracket c_1 \parallel c_2 \rrbracket^{\gamma;\delta\oplus\delta'}$ it follows immediately that $\delta' \models^{\delta}_{\mathrm{D}} s : c_1 \parallel c_2$.

$\boxed{\dfrac{\delta' \vdash^{\delta}_{\mathrm{D}} s_1 : c_1 \quad \delta' \vdash^{\delta}_{\mathrm{D}} s_2 : c_2}{\delta' \vdash^{\delta}_{\mathrm{D}} s_1 s_2 : c_1; c_2}}$ Immediate from the definition of $\mathcal{C}\llbracket c_1; c_2 \rrbracket^{\gamma;\delta\oplus\delta'}$.

$\boxed{\dfrac{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_1}{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_1 + c_2}}$ Immediate from the definition of $\mathcal{C}\llbracket c_1 + c_2 \rrbracket^{\gamma;\delta\oplus\delta'}$.

$\boxed{\dfrac{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_2}{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_1 + c_2}}$ Immediate from the definition of $\mathcal{C}\llbracket c_1 + c_2 \rrbracket^{\gamma;\delta\oplus\delta'}$.

"$\subseteq$": We prove $\mathcal{C}\llbracket c \rrbracket^{\gamma;\delta\oplus\delta'} \subseteq \{s \mid \delta' \vdash^{\delta}_{\mathrm{D}} s : c\}$.

Define $\gamma'(f_i) = \lambda\vec{v}.\{s \mid \vec{X}_i \mapsto \vec{v} \vdash^{\delta}_{\mathrm{D}} s : c_i\}$ for $1 \leq i \leq m$. (Recall that δ is fixed.)

Claim: $\mathcal{C}\llbracket c \rrbracket^{\gamma';\delta\oplus\delta'} = \{s \mid \delta' \vdash^{\delta}_{\mathrm{D}} s : c\}$ for all δ'.

Proof of claim:

The proof is by structural induction on c.

- Consider $f(\vec{a})$ for some $(f(\vec{X}) = c) \in \mathrm{D}$. Let $\vec{v} = \mathcal{Q}\llbracket \vec{a} \rrbracket^{\delta\oplus\delta'}$. We need to show that $\mathcal{C}\llbracket f(\vec{a}) \rrbracket^{\gamma';\delta\oplus\delta'} = \{s \mid \delta' \vdash^{\delta}_{\mathrm{D}} s : f(\vec{a})\}$.

  We have:

$$\begin{aligned}
\mathcal{C}\llbracket f(\vec{a}) \rrbracket^{\gamma';\delta\oplus\delta'} &= \gamma'(f)(\mathcal{Q}\llbracket \vec{a} \rrbracket^{\delta\oplus\delta'}) \\
&= \gamma'(f)(\vec{v}) \\
&= \{s \mid \vec{X} \mapsto \vec{v} \vdash^{\delta}_{\mathrm{D}} s : c\} \\
&= \{s \mid \delta' \vdash^{\delta}_{\mathrm{D}} s : f(\vec{a})\}
\end{aligned}$$

  which concludes this case.

- Consider $\mathrm{transmit}(\vec{X} \mid P).\, c$. We may assume $\mathcal{C}\llbracket c \rrbracket^{\gamma';\delta\oplus\delta'} = \{s \mid \delta' \vdash^{\delta}_{\mathrm{D}} s : c\}$ for all δ'. We need to show that $\mathcal{C}\llbracket \mathrm{transmit}(\vec{X} \mid P).\, c \rrbracket^{\gamma';\delta\oplus\delta'} = \{s \mid \delta' \vdash^{\delta}_{\mathrm{D}} s : \mathrm{transmit}(\vec{X} \mid P).\, c\}$.

  We have:

$$\begin{aligned}
&\mathcal{C}\llbracket \mathrm{transmit}(\vec{X} \mid P).\, c \rrbracket^{\gamma';\delta\oplus\delta'} \\
&= \{\mathrm{transmit}(\vec{v})\, s \mid \mathcal{Q}\llbracket P \rrbracket^{\delta\oplus\delta'\oplus\vec{X}\mapsto\vec{v}} = \mathrm{true} \wedge s \in \mathcal{C}\llbracket c \rrbracket^{\gamma';\delta\oplus\delta'\oplus\vec{X}\mapsto\vec{v}}\} \\
&= \{\mathrm{transmit}(\vec{v})\, s \mid \delta \oplus \delta' \oplus \vec{X} \mapsto \vec{v} \models P \wedge \delta' \oplus \delta'' \vdash^{\delta}_{\mathrm{D}} s : c\} \\
&= \{s' \mid \delta' \vdash^{\delta}_{\mathrm{D}} s' : \mathrm{transmit}(\vec{X} \mid P).\, c\}
\end{aligned}$$

  which concludes this case.

- The remaining cases are straightforward.

From $\mathcal{C}\llbracket c \rrbracket^{\gamma';\delta\oplus\delta'} = \{s \mid \delta' \vdash^{\delta}_{\mathrm{D}} s : c\}$ for all δ' follows immediately that $\gamma'(f) = \lambda\vec{v}.\mathcal{C}\llbracket c \rrbracket^{\gamma';\delta\oplus\vec{X}\mapsto\vec{v}}$ for all $(f(\vec{X}) = c) \in \mathrm{D}$. Since $\gamma = \mathcal{D}\llbracket \mathrm{D} \rrbracket^{\delta}$ is the least function with this property, it follows that $\gamma \sqsubseteq \gamma'$ and thus $\mathcal{C}[\![c]\!]^{\gamma;\delta\oplus\delta'} \subseteq \mathcal{C}[\![c]\!]^{\gamma';\delta\oplus\delta'} = \{s \mid \delta' \vdash^{\delta}_{\mathrm{D}} s : c\}$ and we are done.

Lemma 3 The proof proceeds by structural induction on c assuming (A) for our base language:

$$\mathcal{Q}[\![\Delta \vdash b[\vec{v}/\vec{X}] : \tau]\!]^{\delta} = \mathcal{Q}[\![\Delta \vdash b : \tau]\!]^{\delta\oplus\{X\mapsto\vec{v}}$$

.

We use figure 7 and abbreviate $\mathcal{D}[\![\mathrm{D}]\!]^{\delta}$ by γ where appropriate.

$\boxed{c \equiv \text{Success}}$ To show: $\mathcal{C}[\![\text{Success}]\!]^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}} = \mathcal{C}[\![\text{Success}[\vec{v}/\vec{X}]]\!]^{\gamma;\delta}$.

We have $\mathcal{C}[\![\text{Success}]\!]^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}} = \{\langle\rangle\} = \mathcal{C}[\![\text{Success}[\vec{v}/\vec{X}]]\!]^{\gamma;\delta}$.

$\boxed{c \equiv \text{Failure}}$ This case proceeds exactly as the previous except that both sides denote $\emptyset$.

$\boxed{c \equiv f(\vec{b})}$ To show: $\mathcal{C}[\![f(\vec{b})[\vec{v}/\vec{X}]]\!]^{\gamma;\delta} = \mathcal{C}[\![f(\vec{b})]\!]^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}}$.

We have:

$$\begin{aligned}
\mathcal{C}[\![f(\vec{b})[\vec{v}/\vec{X}]]\!]^{\gamma;\delta} &= \mathcal{C}[\![f(\vec{b}[\vec{v}/\vec{X}])]\!]^{\gamma;\delta}\\
&= \gamma(f)(\mathcal{Q}[\![\vec{b}[\vec{v}/\vec{X}]]\!]^{\delta}\\
&= \gamma(f)(\mathcal{Q}[\![\vec{b}]\!]^{\delta\oplus\vec{X}\mapsto\vec{v}})\text{(by (A))}\\
&= \mathcal{C}[\![f(\vec{b})]\!]^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}}
\end{aligned}$$

$\boxed{c \equiv \text{transmit}(\vec{X}' \mid P).\, c'}$ To show:

$\mathcal{C}[\![\text{transmit}(\vec{X}' \mid P).\, c'[\vec{v}/\vec{X}]]\!]^{\gamma;\delta} = \mathcal{C}[\![\text{transmit}(\vec{X}' \mid P).\, c']\!]^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}}$.

We allow α-conversion and may thus assume that $\vec{X}'$ is chosen such that $\vec{X} \cap \vec{X}' = \emptyset$. We have:

$$\begin{aligned}
&\mathcal{C}[\![\text{transmit}(\vec{X}' \mid P).\, c'[\vec{v}/\vec{X}]]\!]^{\gamma;\delta} =\\
&\mathcal{C}[\![\text{transmit}(\vec{X}' \mid P[\vec{v}/\vec{X}]).\, c'[\vec{v}/\vec{X}]]\!]^{\gamma;\delta} =\\
&\{\text{transmit}(\vec{v'})\ s \mid \mathcal{Q}[\![P[\vec{v}/\vec{X}]]\!]^{\delta\oplus\vec{X}'\mapsto\vec{v'}} = \text{true} \wedge s \in \mathcal{C}[\![c'[\vec{v}/\vec{X}]]\!]^{\gamma;\delta\oplus\vec{X}'\mapsto\vec{v'}}\} =\\
&\{\text{transmit}(\vec{v})\ s \mid \mathcal{Q}[\![P]\!]^{\delta\oplus\vec{X}'\mapsto\vec{v'}\oplus\vec{X}\mapsto\vec{v}} = \text{true} \wedge s \in \mathcal{C}[\![c']\!]^{\gamma;\delta\oplus\vec{X}'\mapsto\vec{v'}\oplus\vec{X}\mapsto\vec{v}} =\\
&\{\text{transmit}(\vec{v})\ s \mid \mathcal{Q}[\![P]\!]^{\delta\oplus\vec{X}\mapsto\vec{v}\oplus\vec{X}'\mapsto\vec{v}'} = \text{true} \wedge s \in \mathcal{C}[\![c']\!]^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}\oplus\vec{X}'\mapsto\vec{v}'} =\\
&\mathcal{C}[\![\text{transmit}(\vec{X}' \mid P).\, c']\!]^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}}
\end{aligned}$$

$\boxed{c \equiv c_1 + c_2}$ To show: $\mathcal{C}\llbracket c_1 + c_2[\vec{v}/\vec{X}]\rrbracket^{\gamma;\delta} = \mathcal{C}\llbracket c_1 + c_2\rrbracket^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}}$. We have:

$$\begin{aligned}\mathcal{C}\llbracket c_1 + c_2[\vec{v}/\vec{X}]\rrbracket^{\gamma;\delta} &= \mathcal{C}\llbracket c_1[\vec{v}/\vec{X}] + c_2[\vec{v}/\vec{X}]\rrbracket^{\gamma;\delta}\\ &= \mathcal{C}\llbracket c_1[\vec{v}/\vec{X}]\rrbracket^{\gamma;\delta} \cup \mathcal{C}\llbracket c_2[\vec{v}/\vec{X}]\rrbracket^{\gamma;\delta}\\ &= \mathcal{C}\llbracket c_1\rrbracket^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}} \cup \mathcal{C}\llbracket c_2\rrbracket^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}}\\ &= \mathcal{C}\llbracket c_1 + c_2\rrbracket^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}}\end{aligned}$$

$\boxed{c \equiv c_1 \parallel c_2}$ Similar to $\cdot + \cdot$ case.

$\boxed{c \equiv c_1; c_2}$ Similar to $\cdot + \cdot$ case.

□

Lemma 2 We verify that each equation in Figure 8 holds. Note that $\gamma = \mathcal{D}\llbracket \mathrm{D}\rrbracket^{\delta}$ in the following.

$\boxed{\mathcal{C}\llbracket e\backslash\text{Failure}\rrbracket^{\gamma;\delta} = \mathcal{C}\llbracket\text{Failure}\rrbracket^{\gamma;\delta}}$ By definition of the residuation operator we have $\mathcal{C}\llbracket e\backslash\text{Failure}\rrbracket^{\gamma;\delta} = e\backslash\emptyset = \emptyset = \mathcal{C}\llbracket\text{Failure}\rrbracket^{\gamma;\delta}$.

$\boxed{\mathcal{C}\llbracket e\backslash\text{Success}\rrbracket^{\gamma;\delta} = \mathcal{C}\llbracket\text{Failure}\rrbracket^{\gamma;\delta}}$ By definition of the residuation operator we have $\mathcal{C}\llbracket e\backslash\text{Success}\rrbracket^{\gamma;\delta} = e\backslash\{\langle\rangle\} = \emptyset = \mathcal{C}\llbracket\text{Failure}\rrbracket^{\gamma;\delta}$.

$\boxed{\mathcal{C}\llbracket e\backslash f(\vec{a})\rrbracket^{\gamma;\delta} = \mathcal{C}\llbracket e\backslash c[\vec{v}/\vec{X}]\rrbracket^{\gamma;\delta}}$ This follows from $\mathcal{C}\llbracket f(\vec{a})\rrbracket^{\gamma;\delta} = \mathcal{C}\llbracket c\rrbracket^{\gamma;\delta\oplus\vec{X}\mapsto v} = \mathcal{C}\llbracket c[\vec{v}/\vec{X}]\rrbracket^{\gamma;\delta}$.

We assume $(f(\vec{X}) = c) \in \mathrm{D}$ and $\vec{v} = \mathcal{Q}\llbracket\vec{a}\rrbracket^{\delta}$.

We have $\mathcal{C}\llbracket f(\vec{a})\rrbracket^{\gamma;\delta} = \mathcal{C}\llbracket c\rrbracket^{\gamma;\delta\oplus\vec{X}\mapsto v}$ by definition of γ and assumption for $\vec{v}$. By Lemma 3 this can be rewritten to $\mathcal{C}\llbracket c[\vec{v}/\vec{X}]\rrbracket^{\gamma;\delta}$, and we are done.

$\boxed{\mathcal{C}\llbracket\text{transmit}(\vec{v})\backslash(\text{transmit}(\vec{X} \mid P).\, c')\rrbracket^{\gamma;\delta}}$ There are two cases to consider:

- $\delta \oplus \{\vec{X} \mapsto \vec{v}\} \models P$.

  To show: $\mathcal{C}\llbracket\text{transmit}(\vec{v})\backslash(\text{transmit}(\vec{X} \mid P).\, c')\rrbracket^{\gamma;\delta} = \mathcal{C}\llbracket c'[\vec{v}/\vec{X}]\rrbracket^{\gamma;\delta}$.

  Again we unfold the left-hand side of the equation and the goal is then:

$$\{s' \mid \exists s \in \mathcal{C}\llbracket\text{transmit}(\vec{X} \mid P).\, c'\rrbracket^{\gamma;\delta} : \text{transmit}(\vec{v})s' = s\} = \mathcal{C}\llbracket c'[\vec{v}/\vec{X}]\rrbracket^{\gamma;\delta}.$$

From the denotational semantics we see that

$$\mathcal{C}\llbracket\text{transmit}(\vec{X} \mid P).\, c'\rrbracket^{\gamma;\delta} = \{\text{transmit}(\vec{v})\, s' \mid s' \in \mathcal{C}\llbracket c'\rrbracket^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}}.$$

What we need to show is then that $\mathcal{C}[\![c']\!]^{\gamma;\delta\oplus\vec{X}\mapsto\vec{v}} = \mathcal{C}[\![c'[\vec{v}/\vec{X}]]\!]^{\gamma;\delta}$, which follows immediately from Lemma 3.

- $\delta \oplus \{\vec{X} \mapsto \vec{a}\} \not\models P$.

  To show: $\mathcal{C}[\![\mathrm{transmit}(\vec{a})\backslash(\mathrm{transmit}(\vec{X} \mid P).\, c')]\!]^{\gamma;\delta} = \mathcal{C}[\![\mathrm{Failure}]\!]^{\gamma;\delta}$.

  We unfold the left-hand side of the equation using the denotational semantics and the goal is now to show:

$$\{s' \mid \exists s \in \mathcal{C}[\![\mathrm{transmit}(\vec{X} \mid P).\, c']\!]^{\gamma;\delta} : \mathrm{transmit}(\vec{v})s' = s\} = \emptyset.$$

  Since $\delta \oplus \{\vec{X} \mapsto \vec{a}\} \not\models P$ we know that $\mathcal{C}[\![\mathrm{transmit}(\vec{X} \mid P).\, c']\!]^{\gamma;\delta} = \emptyset$, and we are done.

$\boxed{\mathcal{C}[\![e\backslash(c_1 + c_2)]\!]^{\gamma;\delta} = \mathcal{C}[\![e\backslash c_1 + e\backslash c_2]\!]^{\gamma;\delta}}$ Unfolding the left-hand side gives $\{s' \mid \exists s \in \mathcal{C}[\![c_1 + c_2]\!]^{\gamma;\delta} : es' = s\}$. The denotation of a choice contract is given by

$$\mathcal{C}[\![c_1 + c_2]\!]^{\gamma;\delta} = \mathcal{C}[\![c_1]\!]^{\gamma;\delta} \cup \mathcal{C}[\![c_2]\!]^{\gamma;\delta}.$$

Any s' will thus be a trace of c_1 or a trace of c_2 with a prefix of e removed. The denotation of the right-hand side is $\mathcal{C}[\![e\backslash c_1]\!]^{\gamma;\delta} \cup \mathcal{C}[\![e\backslash c_2]\!]^{\gamma;\delta}$ which unfolds to $\{s'_1 \mid \exists s \in \mathcal{C}[\![c_1]\!]^{\gamma;\delta} : es'_1 = s\} \cup \{s'_2 \mid \exists s \in \mathcal{C}[\![c_2]\!]^{\gamma;\delta} : es'_2 = s\}$. Thus any s'_1 or s'_2 is a trace of c_1 or c_2 with the prefix e removed. We can now conclude that in any case $s' = s'_i$ for $0 < i \leq 2$ as required.

$\boxed{\mathcal{C}[\![e\backslash(c_1 \parallel c_2)]\!]^{\gamma;\delta} = \mathcal{C}[\![e\backslash c_1 \parallel c_2 + c_1 \parallel e\backslash c_2]\!]^{\gamma;\delta}}$ Rewriting the left-hand side of the equation by definition of the residuation operator we arrive at the following equation:

$$\{s' \mid \exists s \in \mathcal{C}[\![c_1 \parallel c_2]\!]^{\gamma;\delta} : es' = s\} = \mathcal{C}[\![e\backslash c_1 \parallel c_2 + c_1 \parallel e\backslash c_2]\!]^{\gamma;\delta}.$$

Using the definition of the denotational semantics to rewrite the right-hand side we arrive at:

$$\{s' \mid \exists s \in \mathcal{C}[\![c_1 \parallel c_2]\!]^{\gamma;\delta} : es' = s\} = \mathcal{C}[\![e\backslash c_1 \parallel c_2]\!]^{\gamma;\delta} \cup \mathcal{C}[\![c_1 \parallel e\backslash c_2]\!]^{\gamma;\delta}.$$

From the denotational semantics, we note that the trace set of a parallel contract is an interleaving of the events from *both* subcontracts:

$$\{s' \mid \exists s \in \{s'' \mid s_1 \in \mathcal{C}[\![c_1]\!]^{\gamma;\delta}, s_2 \in \mathcal{C}[\![c_2]\!]^{\gamma;\delta} : (s_1, s_2) \rightsquigarrow s''\} : es' = s\} = \ldots$$

If e is a prefix of s_1 we have the trace set $\mathcal{C}[\![e\backslash c_1 \parallel c_2]\!]^{\gamma;\delta}$ and if e is a prefix of s_2 we have the traceset $\mathcal{C}[\![c_1 \parallel e\backslash c_2]\!]^{\gamma;\delta}$. Combining these two sets we conclude what was required.

$$\boxed{\mathcal{C}[\![e\backslash(c_1;c_2)]\!]^{\gamma;\delta} = \begin{cases} (e\backslash c_1;c_2) + e\backslash c_2 \ \textbf{if}\ \mathrm{D},\delta \models \text{Success} \subseteq c_1 \\ e\backslash c_1;c_2 \qquad\qquad \textbf{otherwise} \end{cases}}$$

- $\mathrm{D},\delta \models \text{Success} \subseteq c_1$.

  We unfold the left-hand side and the goal becomes:

  $$\{s' \mid \exists s \in \{s_1s_2 \mid \exists s_1 \in \mathcal{C}[\![c_1]\!]^{\gamma;\delta}, s_2 \in \mathcal{C}[\![c_2]\!]^{\gamma;\delta}\} : es' = s\} = \mathcal{C}[\![e\backslash c_1;c_2 + e\backslash c_2]\!]^{\gamma;\delta}.$$

  Unfold the right-hand side

  $$\{s' \mid \exists s \in \{s_1s_2 \mid \exists s_1 \in \mathcal{C}[\![c_1]\!]^{\gamma;\delta}, s_2 \in \mathcal{C}[\![c_2]\!]^{\gamma;\delta}\} : es' = s\}$$
  $$=$$
  $$\{s'_1s'_2 \mid \exists s'_1 \in \mathcal{C}[\![e\backslash c_1]\!]^{\gamma;\delta}, s'_2 \in \mathcal{C}[\![c_2]\!]^{\gamma;\delta}\} \cup \mathcal{C}[\![e\backslash c_2]\!]^{\gamma;\delta}$$

  - In case $s_1 = \langle\rangle$, we get that $es' = \mathcal{C}[\![c_2]\!]^{\gamma;\delta}$ and $\mathcal{C}[\![e\backslash c_1]\!]^{\gamma;\delta} = \emptyset$. Thus we need to show that: $\{s' \mid \exists s \in \mathcal{C}[\![c_2]\!]^{\gamma;\delta} : es' = s\} = \mathcal{C}[\![e\backslash c_2]\!]^{\gamma;\delta}$, which is immediate from the definition of residuation.
  - If $s_1 \neq \langle\rangle$ there is some s_1 in which e occurs as the first event. Thus $s = es'_1s'_2$, which means $s' = s'_1s'_2$ as required. The added $\mathcal{C}[\![e\backslash c_2]\!]^{\gamma;\delta}$ are accounted for by the previous case.

- $\mathrm{D},\delta \models \text{Success} \not\subseteq c_1$.

  We unfold the left-hand side and the goal becomes:

  $$\{s' \mid \exists s \in \{s_1s_2 \mid \exists s_1 \in \mathcal{C}[\![c_1]\!]^{\gamma;\delta}, s_2 \in \mathcal{C}[\![c_2]\!]^{\gamma;\delta}\} : es' = s\} = \mathcal{C}[\![e\backslash c_1;c_2]\!]^{\gamma;\delta}$$

  Unfold the right-hand side

  $$\{s' \mid \exists s \in \{s_1s_2 \mid \exists s_1 \in \mathcal{C}[\![c_1]\!]^{\gamma;\delta}, s_2 \in \mathcal{C}[\![c_2]\!]^{\gamma;\delta}\} : es' = s\}$$
  $$=$$
  $$\{s'_1s'_2 \mid \exists s'_1 \in \mathcal{C}[\![e\backslash c_1]\!]^{\gamma;\delta}, s'_2 \in \mathcal{C}[\![c_2]\!]^{\gamma;\delta}\}$$

  Since $\langle\rangle \notin \mathcal{C}[\![c_1]\!]^{\gamma;\delta}$, we know that $e \in s_1$ from which we immediately see that $s_2 = s'_2$; thus we just need to show that

  $$\{s' \mid \exists s \in \{s_1 \mid \exists s_1 \in \mathcal{C}[\![c_1]\!]^{\gamma;\delta}\} : es' = s\} = \{s'_1 \mid \exists s'_1 \in \mathcal{C}[\![e\backslash c_1]\!]^{\gamma;\delta}\}$$

  . That is,

  $$\{s' \mid \exists s \in \mathcal{C}[\![c_1]\!]^{\gamma;\delta}\} : es' = s\} = \mathcal{C}[\![e\backslash c_1]\!]^{\gamma;\delta}.$$

  Which is exactly the definition of the residuation operator.

Proposition 5 We show $\forall \mathrm{D}, \delta, \delta', c : \delta' \vdash^{\delta}_{\mathrm{D}} \langle\rangle : c \iff \mathrm{D} \vdash c$ nullable. From this the proposition follows by Theorem 1.

"$\Longrightarrow$": To show $\forall \mathrm{D}, \delta, \delta', c : \delta' \vdash^{\delta}_{\mathrm{D}} \langle\rangle : c \Longrightarrow \mathrm{D} \vdash c$ nullable we proceed by induction on derivations of $\delta' \vdash^{\delta}_{\mathrm{D}} s : c$.

$$\boxed{\delta' \vdash^{\delta}_{\mathrm{D}} \langle\rangle : \text{Success}}$$

We need to show that $\mathrm{D} \vdash$ Success nullable. This follows immediately from the nullability axiom for Success.

$$\boxed{\dfrac{\vec{X} \mapsto \vec{v} \vdash^{\delta}_{\mathrm{D}} s : c \quad (f(\vec{X}) = c) \in \mathrm{D}, \vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta \oplus \delta'}}{\delta' \vdash^{\delta}_{\mathrm{D}} s : f(\vec{a})}}$$

Assume $\mathrm{D} \vdash c$ nullable (induction hypothesis). We need to show that $\mathrm{D} \vdash f(\vec{a})$ nullable, which follows from the nullability inference rule for $f(\vec{a})$.

$$\boxed{\dfrac{\delta \oplus \delta'' \models P \quad \delta'' \vdash^{\delta}_{\mathrm{D}} s : c \qquad (\delta'' = \delta' \oplus \{\vec{X} \mapsto \vec{v}\})}{\delta' \vdash^{\delta}_{\mathrm{D}} \mathrm{transmit}(\vec{v})\, s : \mathrm{transmit}(\vec{X}|P).\, c}}$$

We need to show $\mathrm{D} \vdash \mathrm{transmit}(\vec{X}|P).\, c$ nullable if $\mathrm{transmit}(\vec{v})\, s = \langle\rangle$. This implication is vacuously true since the assumption $\mathrm{transmit}(\vec{v})\, s = \langle\rangle$ is false.

$$\boxed{\dfrac{\delta' \vdash^{\delta}_{\mathrm{D}} s_1 : c_1 \quad \delta' \vdash^{\delta}_{\mathrm{D}} s_2 : c_2 \quad (s_1, s_2) \rightsquigarrow s}{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_1 \parallel c_2}}$$

Assume $(\langle\rangle, \langle\rangle) \rightsquigarrow s$; that is, $s = \langle\rangle$. Assume furthermore $\mathrm{D} \vdash c_1$ nullable and $\mathrm{D} \vdash c_2$ nullable. We need to show that $\mathrm{D} \vdash c_1 \parallel c_2$ nullable, which follows from the nullability inference rule for $c_1 \parallel c_2$.

$$\boxed{\dfrac{\delta' \vdash^{\delta}_{\mathrm{D}} s_1 : c_1 \quad \delta' \vdash^{\delta}_{\mathrm{D}} s_2 : c_2}{\delta' \vdash^{\delta}_{\mathrm{D}} s_1 s_2 : c_1; c_2}}$$

Immediate from nullability inference rule for $c_1; c_2$.

$$\boxed{\dfrac{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_1}{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_1 + c_2}}$$

Immediate from first nullability inference rule for $c_1 + c_2$.

$$\boxed{\dfrac{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_2}{\delta' \vdash^{\delta}_{\mathrm{D}} s : c_1 + c_2}}$$

Immediate from second nullability inference rule for $c_1 + c_2$.

"$\Longleftarrow$": To show $\forall \mathrm{D}, c : \mathrm{D} \vdash c$ nullable $\Longrightarrow \forall \delta, \delta'. \delta' \vdash^{\delta}_{\mathrm{D}} \langle\rangle : c$ we proceed by induction on derivations of $\mathrm{D} \vdash c$ nullable.

$$\boxed{\dfrac{\mathrm{D} \vdash c \text{ nullable} \quad (f(\vec{X}) = c) \in \mathrm{D}}{\mathrm{D} \vdash f(\vec{a}) \text{ nullable}}}$$

Assume $\forall \delta, \delta'. \delta' \vdash^{\delta}_{\mathrm{D}} \langle\rangle : c$ (induction hypothesis). We need to show $\forall \delta, \delta'. \delta' \vdash^{\delta}_{\mathrm{D}} \langle\rangle : f(\vec{a})$. Let δ, δ' be arbitrary environments for D and $f(\vec{a})$. From the induction hypothesis it follows that $\vec{X} \mapsto \vec{v} \vdash^{\delta}_{\mathrm{D}} \langle\rangle : c$ where $\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta\oplus\delta'}$. And, using the satisfaction inference rule for contract application, we arrive at $\delta' \vdash^{\delta}_{\mathrm{D}} \langle\rangle : f(\vec{a})$.

$\boxed{\dfrac{\mathrm{D} \vdash c \text{ nullable}}{\mathrm{D} \vdash c + c' \text{ nullable}}}$ Immediate.

$\boxed{\dfrac{\mathrm{D} \vdash c' \text{ nullable}}{\mathrm{D} \vdash c + c' \text{ nullable}}}$ Immediate.

$\boxed{\mathrm{D} \vdash \text{Success nullable}}$ Let δ, δ' be arbitrary environments. Using the satisfaction rule for Success we obtain $\delta' \vdash^{\delta}_{\mathrm{D}} \langle\rangle : \text{Success}$.

$\boxed{\dfrac{\mathrm{D} \vdash c \text{ nullable} \quad \mathrm{D} \vdash c' \text{ nullable}}{\mathrm{D} \vdash c \parallel c' \text{ nullable}}}$ Immediate.

$\boxed{\dfrac{\mathrm{D} \vdash c \text{ nullable} \quad \mathrm{D} \vdash c' \text{ nullable}}{\mathrm{D} \vdash c; c' \text{ nullable}}}$ Immediate.

Lemma 7 This is proved by straightforward structural induction on the definition of contracts.

The only interesting cases are the cases of a contract application $f(\vec{a})$, where $(f(\vec{X}) = c) \in \mathrm{D}$, and sequential composition.

In the first case, we can use the assumption of the Lemma that $\mathrm{D} \vdash c$ guarded, which, by rule application, immediately implies that $\mathrm{D} \vdash f(\vec{a})$ guarded.

In the second case, we have the induction hypotheses $\mathrm{D} \vdash c_1$ guarded and $\mathrm{D} \vdash c_2$ guarded. Now, either $\mathrm{D} \vdash c_1$ nullable or $\mathrm{D} \nvdash c_1$ nullable. In either case, we have a rule for concluding that $\mathrm{D} \vdash c_1; c_2$ guarded.

Theorem 8 (Sketch)

(1) We show $\mathrm{D}, \delta \vdash_D c \xrightarrow{e} c' \Longrightarrow \mathrm{D}, \delta \models e \backslash c = c'$ by induction on derivations of $\mathrm{D}, \delta \vdash_D c \xrightarrow{e} c'$. Each case follows immediately from Lemma 2. In the case of sequential composition we also require Proposition 5.

(2) Note that, by Lemma 7, $\mathrm{D} \vdash c$ guarded if D is guarded. It is sufficient to show $\mathrm{D} \vdash c$ guarded $\Longrightarrow \forall\delta\forall e\exists c'.\mathrm{D}, \delta \vdash_D c \xrightarrow{e} c'$. The fact that c' is guarded in context D follows from Lemma 7, and it is a routine matter to extend the proof cases with a check of uniqueness of c'.

We cannot prove $\mathrm{D} \vdash c$ guarded $\Longrightarrow \forall\delta\forall e\exists c'.\mathrm{D}, \delta \vdash_D c \xrightarrow{e} c'$ by induction on the definition of guaraded contracts, however, since the induction hypothesis is not strong enough in the case of contract application: We would require that $c[\vec{v}/\vec{X}]$ has a residual contract for arbitrary δ, e, but the induction hypothesis only yields that that holds for c. Consequently, we strengthen the lemma and prove $\mathrm{D} \vdash c$ guarded $\Longrightarrow \forall \delta, e, \vec{X}, \vec{v} \exists c'.\mathrm{D}, \delta \vdash_D c[\vec{v}/\vec{X}] \xrightarrow{e} c'$.

All cases are straightforward except the second rule for sequential composition: To make the induction proof go through we require $\mathrm{D} \not\vdash c[\vec{/}\vec{X}]$ nullable the last deterministic reduction rule, but the case only carries the assumption $\mathrm{D} \not\vdash c$ nullable. Consequently, if we can show that $\mathrm{D} \vdash c[\vec{/}\vec{X}]$ nullable $\Longrightarrow \mathrm{D} \vdash c$ nullable, we are done.

Claim: $\mathrm{D} \vdash c[\vec{/}\vec{X}]$ nullable $\Longrightarrow \mathrm{D} \vdash c$ nullable.

Proof of claim: By structural induction on c. All cases are straightforward except the rule for contract application. In that case we need to show $\mathrm{D} \vdash f(\vec{a}[\vec{v}/\vec{X}])$ nullable $\Longrightarrow \mathrm{D} \vdash f(\vec{a})$ nullable. Assume $\mathrm{D} \vdash f(\vec{a}[\vec{v}/\vec{X}])$ nullable. By inspection of the rules for nullability we can see that this must have been concluded from $\mathrm{D} \vdash c$ nullable where $(f(\vec{Y}) = c) \in \mathrm{D}$. By the same rule we can infer, however, $\mathrm{D} \vdash f(\vec{a})$ nullable, and we are done.

Theorem 9 We prove the two statements

(1) If $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} c'$ then $\mathrm{D}, \delta \models c' \subseteq e \backslash c$
(2) If $\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau} c'$ then $\mathrm{D}, \delta \models c' \subseteq c$

by induction on the height of the derivation of $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} c'$ and $\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau} c'$, respectively. We use definitions in Figures 7, 8 and 12. "Assume..." is used as shorthand for "Assume a derivation with the conclusion...". Finally, γ abbreviates $\mathcal{D}[\![\mathrm{D}]\!]^\delta$ in the following.

Proving 1:

- Assume $\mathrm{D}, \delta \vdash_N \text{Success} \xrightarrow{e} \text{Failure}$. To show $\mathcal{C}[\![\text{Failure}]\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![e \backslash \text{Success}]\!]^{\gamma;\delta} = \mathcal{C}[\![\text{Failure}]\!]^{\gamma;\delta}$. Done.
- Assume $\mathrm{D}, \delta \vdash_N \text{Failure} \xrightarrow{e} \text{Failure}$. To show $\mathcal{C}[\![\text{Failure}]\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![e \backslash \text{Failure}]\!]^{\gamma;\delta} = \mathcal{C}[\![\text{Failure}]\!]^{\gamma;\delta}$. Done.
- Assume $\mathrm{D}, \delta \vdash_N \text{transmit}(\vec{X} \mid P).\, c \xrightarrow{\text{transmit}(\vec{v})} c[\vec{v}/\vec{X}]$ and also $\delta \oplus \vec{X} \mapsto \vec{v} \models P$ where $\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^\delta$. To show $\mathcal{C}[\![c[\vec{v}/\vec{X}]]\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![(\text{transmit}(\vec{v}) \backslash \text{transmit}(\vec{X} \mid P).\, c)]\!]^{\gamma;\delta} = \mathcal{C}[\![c[\vec{v}/\vec{X}]]\!]^{\gamma;\delta}$. Done.
- Assume $\mathrm{D}, \delta \vdash_N \text{transmit}(\vec{X} \mid P).\, c \xrightarrow{\text{transmit}(\vec{v})} \text{Failure}$ and also $\delta \oplus \vec{X} \mapsto \vec{v} \not\models P$ where $\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^\delta$. To show $\mathcal{C}[\![\text{Failure}]\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![(\text{transmit}(\vec{v}) \backslash \text{transmit}(\vec{X} \mid P).\, c)]\!]^{\gamma;\delta} = \mathcal{C}[\![\text{Failure}]\!]^{\gamma;\delta}$. Done.

- Assume $\mathrm{D}, \delta \vdash_N c \parallel c' \xrightarrow{e} d \parallel c'$ and $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} d$. To show $\mathcal{C}[\![d \parallel c']\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![e \backslash (c \parallel c')]\!]^{\gamma;\delta} = \mathcal{C}[\![e \backslash c \parallel c' + c \parallel e \backslash c']\!]^{\gamma;\delta} = \mathcal{C}[\![e \backslash c \parallel c']\!]^{\gamma;\delta} \cup \mathcal{C}[\![c \parallel e \backslash c']\!]^{\gamma;\delta}$. By the IH, $\mathcal{C}[\![d]\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![e \backslash c]\!]^{\gamma;\delta}$ so in particular $\mathcal{C}[\![d \parallel c']\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![e \backslash c \parallel c']\!]^{\gamma;\delta}$, which is sufficient.
- Assume $\mathrm{D}, \delta \vdash_N c \parallel c' \xrightarrow{e} c \parallel d'$ and $\mathrm{D}, \delta \vdash_N c' \xrightarrow{e} d'$. Analogous to the above case.
- Assume $\mathrm{D}, \delta \vdash_N c; c' \xrightarrow{e} d; c'$ and also $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} d$. To show $\mathcal{C}[\![d; c']\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![e \backslash (c; c')]\!]^{\gamma;\delta}$. If $\mathrm{D}, \delta \models \text{Success} \subseteq c$ then $\mathcal{C}[\![e \backslash (c; c')]\!]^{\gamma;\delta} = \mathcal{C}[\![(e \backslash c; c') + e \backslash c']\!]^{\gamma;\delta} = \mathcal{C}[\![e \backslash c; c']\!]^{\gamma;\delta} \cup \mathcal{C}[\![e \backslash c']\!]^{\gamma;\delta}$. By the IH, $\mathcal{C}[\![d]\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![e \backslash c]\!]^{\gamma;\delta}$ so in particular $\mathcal{C}[\![d; c']\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![e \backslash c; c']\!]^{\gamma;\delta}$, which is sufficient.
- Assume $\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau} c'$ and $\mathrm{D}, \delta \vdash_N c' \xrightarrow{e} c''$. By the IH, we have $\mathcal{C}[\![c']\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![c]\!]^{\gamma;\delta}$ and $\mathcal{C}[\![c'']\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![e \backslash c']\!]^{\gamma;\delta}$. We need to show $\mathcal{C}[\![c'']\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![e \backslash c]\!]^{\gamma;\delta}$. But this follows from the IH and monotonicity of residuation: $\mathcal{C}[\![c'']\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![e \backslash c']\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![e \backslash c]\!]^{\gamma;\delta}$.

Proving 2:

- Assume $\mathrm{D}, \delta \vdash_N f(\vec{a}) \xrightarrow{\tau} c[\vec{V}/\vec{X}]$ where $(f(\vec{X}) = c) \in \mathrm{D}$ and $\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta}$. To show $\mathcal{C}[\![c[\vec{v}/\vec{X}]]\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![f(\vec{a})]\!]^{\gamma;\delta} = \gamma(f)(\vec{v})$, which holds by definition of γ and Lemma 3.
- Assume $\mathrm{D}, \delta \vdash_N c + c' \xrightarrow{\tau} c$. To show $\mathcal{C}[\![c]\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![c + c']\!]^{\gamma;\delta} = \mathcal{C}[\![c]\!]^{\gamma;\delta} \cup \mathcal{C}[\![c']\!]^{\gamma;\delta}$. Done.
- Assume $\mathrm{D}, \delta \vdash_N c + c' \xrightarrow{\tau} c'$. Analogous to the above case.
- Assume $\mathrm{D}, \delta \vdash_N c \parallel c' \xrightarrow{\tau} d \parallel c'$ and $\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau} d$. To show $\mathcal{C}[\![d \parallel c']\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![c \parallel c']\!]^{\gamma;\delta}$, which follows easily by the IH.
- Assume $\mathrm{D}, \delta \vdash_N c \parallel c' \xrightarrow{\tau} c \parallel d'$ and $\mathrm{D}, \delta \vdash_N c' \xrightarrow{\tau} d'$. To show $\mathcal{C}[\![c \parallel d']\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![c \parallel c']\!]^{\gamma;\delta}$, which follows easily by the IH.
- Assume $\mathrm{D}, \delta \vdash_N \text{Success} \parallel c \xrightarrow{\tau} c$. To show $\mathcal{C}[\![c]\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![\text{Success} \parallel c]\!]^{\gamma;\delta}$, holds trivially.
- Assume $\mathrm{D}, \delta \vdash_N c \parallel \text{Success} \xrightarrow{\tau} c$. To show $\mathcal{C}[\![c]\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![c \parallel \text{Success}]\!]^{\gamma;\delta}$, holds trivially.
- Assume $\mathrm{D}, \delta \vdash_N \text{Success}; c' \xrightarrow{\tau} c'$. To show $\mathcal{C}[\![c']\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![\text{Success}; c']\!]^{\gamma;\delta}$, holds trivially.
- Assume $\mathrm{D}, \delta \vdash_N c; c' \xrightarrow{\tau} d; c'$ and $\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau} d$. To show $\mathcal{C}[\![d; c']\!]^{\gamma;\delta} \subseteq \mathcal{C}[\![c; c']\!]^{\gamma;\delta}$, which follows easily by the IH.

□

Theorem 10 The proof is by induction on the derivation of $\mathrm{D}, \delta \vdash_D c \xrightarrow{e} c'$.

- $\mathrm{D}, \delta \vdash_D \text{Success} \xrightarrow{e} \text{Failure}$. Clearly no τ-transitions can be taken in the nondeterministic reduction system. However, there is just one contract c_1 such that $\mathrm{D}, \delta \vdash_N \text{Success} \xrightarrow{e} c_1$ which is Failure. We must then show: $\mathrm{D}, \delta \models \text{Failure} \subseteq \text{Failure}$. By definition $\mathrm{D}, \delta \models \text{Failure} = \emptyset$, so we must show $\emptyset \subseteq \emptyset$ which is trivially true.

- $\mathrm{D}, \delta \vdash_D \text{Failure} \xrightarrow{e} \text{Failure}$. Again no τ-transitions are possible. There is just one contract c_1 such that $\mathrm{D}, \delta \vdash_N \text{Failure} \xrightarrow{e} c_1$ namely Failure. We must show $\mathrm{D}, \delta \models \text{Failure} \subseteq \text{Failure}$, which is true since $\emptyset \subseteq \emptyset$.
- $\mathrm{D}, \delta \vdash_D \text{transmit}(\vec{X}|P).\, c \xrightarrow{\text{transmit}(\vec{v})} c[\vec{v}/\vec{X}]$ where $\delta \oplus \vec{X} \mapsto \vec{v} \models P$ and $\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta}$. In this case we can only do the reduction

$$\mathrm{D}, \delta \vdash_N \text{transmit}(\vec{X}|P).\, c \xrightarrow{\text{transmit}(\vec{v})} c[\vec{v}/\vec{X}].$$

  Now we must show $\mathrm{D}, \delta \models c[\vec{v}/\vec{X}] \subseteq c[\vec{v}/\vec{X}]$, which is obviously true.
- $\mathrm{D}, \delta \vdash_D \text{transmit}(\vec{X}|P).\, c \xrightarrow{\text{transmit}(\vec{v})} \text{Failure}$ and $\delta \oplus \vec{X} \mapsto \vec{v} \not\models P$ where $\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta}$. No τ-transitions are possible and only one contract c_1 exists such that

$$\mathrm{D}, \delta \vdash_D \text{transmit}(\vec{X}|P).\, c \xrightarrow{\text{transmit}(\vec{v})} c_1,$$

  so $c_1 = \text{Failure}$. This means we must show $\mathrm{D}, \delta \models c[\vec{v}/\vec{X}] \subseteq c[\vec{v}/\vec{X}]$ which clearly holds.
- $\mathrm{D}, \delta \vdash_D f(\vec{a}) \xrightarrow{e} c'$. This implies that $(f(\vec{X}) = c) \in \mathrm{D}$ and $\mathrm{D}, \delta \vdash_D c[\vec{v}/\vec{X}] \xrightarrow{e} c'$ with $\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta}$. By a derivation of $\mathrm{D}, \delta \vdash_D c[\vec{v}/\vec{X}] \xrightarrow{e} c'$ we use the IH to get contracts $c_1, \ldots, c_n$ such that $\mathrm{D}, \delta \vdash_N c[\vec{v}/\vec{X}] \xrightarrow{\tau*} c''_i \xrightarrow{e} c'$ and $\mathrm{D}, \delta \models c' \subseteq \sum_{i=1}^{n} c_i$. However we need to show $\mathrm{D}, \delta \vdash_N f(\vec{a}) \xrightarrow{\tau*} c''_i \xrightarrow{e} c_i$ and $c' \subseteq \sum_{i=1}^{n} c_i$, the latter of which follows directly from the IH. By the non-deterministic reduction rules, $f(\vec{a})$ has just one reduction $\mathrm{D}, \delta \vdash_N f(\vec{a}) \xrightarrow{\tau} c[\vec{v}/\vec{X}]$. Thus we can extend all reductions of $\mathrm{D}, \delta \vdash_N c[\vec{v}/\vec{X}] \xrightarrow{\tau*} c''_i \xrightarrow{e} c_i$ with one more τ-transition giving reductions $\mathrm{D}, \delta \vdash_N f(\vec{a}) \xrightarrow{\tau*} c''_i \xrightarrow{e} c_i$ for all $0 < i \leq n$.
- $\mathrm{D}, \delta \vdash_D c + c' \xrightarrow{e} d + d'$. This implies that $\mathrm{D}, \delta \vdash_D c \xrightarrow{e} d$ and $\mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d'$. From the non-deterministic reduction rules we see that $c + c'$ may be reduced by a τ-transition into either c or c'. By the IH we then have contracts $d_0, \ldots, d_n$ and $d'_0, \ldots, d'_m$ such that $\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau*} d''_i \xrightarrow{e} d_i$ for $0 < i \leq n$, $\mathrm{D}, \delta \models d \subseteq \sum_{i=1}^{n} d_i$ and $\mathrm{D}, \delta \vdash_N c' \xrightarrow{\tau*} d'''_j \xrightarrow{e} d'_j$ for $0 < j \leq m$, $\mathrm{D}, \delta \models d \subseteq \sum_{j=1}^{m} d_j$. Thus we can extend the non-deterministic reductions of c and c' to get reductions of $c + c$ into contracts $c_0, \ldots, c_{n+m}$. That is: there are contracts, c_i such that $\mathrm{D}, \delta \vdash_N c + c' \xrightarrow{\tau*} c''_i \xrightarrow{e} c_i$ with $0 < i \leq m + n$. As seen from the IH we know that $\mathrm{D}, \delta \models d \subseteq \sum_{i=1}^{n} d_i$ and $\mathrm{D}, \delta \models d \subseteq \sum_{j=1}^{m} d_j$. Taking the union of these we get $\mathrm{D}, \delta \models d \bigcup d' \subseteq \sum_{i=1}^{n} d_i + \sum_{j=1}^{m} d_j$. By definition this is $\mathrm{D}, \delta \models d + d' \subseteq \sum_{i=1}^{m+n} d_i$ (given proper enumeration of contracts in d_i and d_j which is the desired goal.
- $\mathrm{D}, \delta \vdash_D c \parallel c' \xrightarrow{e} d \parallel c' + c \parallel d'$. By a derivation $\mathrm{D}, \delta \vdash_D c \xrightarrow{e} d$ we use the IH

to get contracts d_i such that $c \xrightarrow{\tau *} c_i'' \xrightarrow{e} d_i$ and $\mathrm{D}, \delta \models d \subseteq \sum_{i=1}^{n} d_i$. Then use the left $\cdot \parallel \cdot$-introduction rule to get contracts $d_i \parallel c'$ such that $c \parallel c' \xrightarrow{\tau *} c_i'' \parallel c' \xrightarrow{e} d_i \parallel c'$ and $\mathrm{D}, \delta \models d \parallel c' \subseteq \sum_{i=1}^{n} d_i \parallel c'$. By a derivation $\mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d'$ we now again use the IH to get contracts d_i' such that $c' \xrightarrow{\tau *} c_i''' \xrightarrow{e} d_i'$ and $\mathrm{D}, \delta \models d' \subseteq \sum_{i=1}^{m} d_i'$. Then use the right $\cdot \parallel \cdot$-introduction rule to get contracts $c \parallel d_i'$ such that $c \parallel c' \xrightarrow{\tau *} c \parallel c_i''' \xrightarrow{e} c \parallel d_i'$ and $\mathrm{D}, \delta \models c \parallel d' \subseteq \sum_{i=1}^{m} c \parallel d_i'$. Taking all contracts $d_i \parallel c'$ and $c \parallel d_i'$ we need to show $\mathrm{D}, \delta \models d \parallel c' + c \parallel d' \subseteq \sum_{i=1}^{n} d_i \parallel c + \sum_{i=1}^{m} c \parallel d_i'$ which follows directly by the above.

- $\mathrm{D}, \delta \vdash_D c; c' \xrightarrow{e} d; c' + d'$ and $\mathrm{D} \vdash c$ nullable. There are two possible reductions of $c; c'$ under the non-deterministic reduction rules. Either $\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau *}$ Success and so $\mathrm{D}, \delta \vdash_N$ Success$; c' \xrightarrow{\tau} c'$ or $\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau *} c_p \xrightarrow{e} d_p$ where $c_p \neq$ Success and then
  $\mathrm{D}, \delta \vdash_N c; c' \xrightarrow{\tau *} c_f \xrightarrow{e} d_p$.

  In the former case, by a derivation of $\mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d$ we get by the IH that there exist contracts d_i' such that $c' \xrightarrow{\tau *} c'' \xrightarrow{e} d_i'$ and $\mathrm{D}, \delta \models d' \subseteq \sum_{i=1}^{n} d_i'$. Taking $c_p = c''$ and $d_p = d$.

  In the latter case there is no sequence of τ-transitions that makes $c =$ Success so all contracts d_q such that $c; c' \xrightarrow{\tau *} c_q \xrightarrow{e} d_q$ must have the form $d_i; c'$. By a derivation $\mathrm{D}, \delta \vdash_D c \xrightarrow{e} d$ the IH gives that there are contracts d_i such that $c \xrightarrow{\tau *} c''' \xrightarrow{e} d_i$ and $\mathrm{D}, \delta \models d \subseteq \sum_{i=1}^{m} d_i$. This implies

  $$\mathrm{D}, \delta \vdash_N c; c' \xrightarrow{\tau *} c'''' \xrightarrow{e} d_i; c'$$

  for $0 < i \leq m$ and furthermore that $\mathrm{D}, \delta \models d; c' \subseteq \sum_{i=1}^{m} d_i; c'$.

  We have thus shown that there are contracts c_i such that

  $$\mathrm{D}, \delta \vdash_N c; c' \xrightarrow{\tau *} c'' \xrightarrow{e} c_i$$

  and that c_i is either $d'i$ or $d_i; c'$. We still need to show that $\mathrm{D}, \delta \models d; c' + d \subseteq \sum_{i=1}^{k} c_i$; that is: $\mathrm{D}, \delta \models d; c' + d \subseteq \sum_{i=1}^{m} d_i; c' + \sum_{i=1}^{n} d_i'$. This follows directly by the already noted fact that by the IH $\mathrm{D}, \delta \models d; c' \subseteq \sum_{i=1}^{m} d_i; c'$ and $\mathrm{D}, \delta \models d' \subseteq \sum_{i=1}^{n} d_i'$
- $\mathrm{D}, \delta \vdash_D c; c' \xrightarrow{e} d; c'$ and $\mathrm{D} \nvdash c$ nullable. To show: $\mathrm{D}, \delta \vdash_N c; c' \xrightarrow{\tau *} c_i'' \xrightarrow{e} c_i$ and $\mathrm{D}, \delta \models d; c' \subseteq \sum_{i=1}^{n} c_i$. By a derivation $\mathrm{D}, \delta \vdash_D c \xrightarrow{e} d$ the IH yields contracts $d_0, \ldots, d_n$ for $0 < i \leq n$ such that $\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau *} c_i''' \xrightarrow{e} d_i$ and $\mathrm{D}, \delta \models d \subseteq \sum_{i=1}^{n} d_i$. Since $\mathrm{D} \nvdash c$ nullable $c \neq$ Success so no number of τ-reductions can make $c; c' = c'$. The form of all c_i must then be $d_i; c'$. The goal is then to show $\mathrm{D}, \delta \models d; c \subseteq d_i; c'$ which follows by $\mathrm{D}, \delta \models d \subseteq \sum_{i=1}^{n} d_i$.

□

Proposition 11 By induction on the derivation of $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} c'$.

$\boxed{\dfrac{(f(\vec{X}) = c) \in \mathrm{D}, \vec{v} = \mathcal{Q}\llbracket\vec{a}\rrbracket^{\delta}}{\mathrm{D}, \delta \vdash_C f(\vec{a}) \xrightarrow{\tau} c[\vec{v}/\vec{X}]}}$ Here, we have

$$\mathcal{C}\llbracket f(\vec{a})\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta} = \mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta}(f)(\mathcal{Q}\llbracket\vec{a}\rrbracket^{\delta}) = \mathcal{C}\llbracket c[\vec{a}/\vec{X}]\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta},$$

as desired.

$\boxed{\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} d}{\mathrm{D}, \delta \vdash_C c + c' \xrightarrow{\tau} d + c'}}$ In this case,

$$\mathcal{C}\llbracket d + c'\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta} = \mathcal{C}\llbracket d\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta} \cup \mathcal{C}\llbracket c'\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta} = \mathcal{C}\llbracket c\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta} \cup \mathcal{C}\llbracket c'\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta},$$

where the last equality follows from the IH. But $\mathcal{C}\llbracket c\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta} \cup \mathcal{C}\llbracket c'\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta} = \mathcal{C}\llbracket c + c'\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta}$, concluding the proof of the case.

$\boxed{\dfrac{\mathrm{D}, \delta \vdash_C c' \xrightarrow{\tau} d'}{\mathrm{D}, \delta \vdash_C c + c' \xrightarrow{\tau} c + d'}}$ As the previous case.

$\boxed{\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} d}{\mathrm{D}, \delta \vdash_C c \parallel c' \xrightarrow{\tau} d \parallel c'}}$ We have that $\mathcal{C}\llbracket d \parallel c'\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta}$ equals

$$\left\{s : s \in \mathit{Tr} \mid \exists s_1 \in \mathcal{C}\llbracket d\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta} s_2 \in \mathcal{C}\llbracket c'\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta}. (s_1, s_2) \rightsquigarrow s\right\}$$

By the IH, we gather that $\{t \in \mathcal{C}\llbracket d\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta}}\}$ equals $\{s' \in \mathcal{C}\llbracket c\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta}}\}$, whence

$$\{s : s \in \mathit{Tr} \mid \exists s_1 \in \mathcal{C}\llbracket d\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta} s_2 \in \mathcal{C}\llbracket c'\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta}. (s_1, s_2) \rightsquigarrow s\}$$

equals

$$\left\{s : s \in \mathit{Tr} \mid \exists s_1 \in \mathcal{C}\llbracket c\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta} s_2 \in \mathcal{C}\llbracket c'\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta}. (s_1, s_2) \rightsquigarrow s\right\}$$

as desired.

$\boxed{\dfrac{\mathrm{D}, \delta \vdash_C c' \xrightarrow{\tau} d'}{\mathrm{D}, \delta \vdash_C c \parallel c' \xrightarrow{\tau} c \parallel d'}}$ As the previous case.

$\boxed{\mathrm{D}, \delta \vdash_C \text{Success} \parallel c \xrightarrow{\tau} c}$ We have $\mathcal{C}\llbracket\text{Success}\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta} = \{\langle\rangle\}$, and thus obtain $\{s : s \in \mathit{Tr} \mid \exists s_1 \in \mathcal{C}\llbracket\text{Success}\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta} s_2 \in \mathcal{C}\llbracket c'\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta}. (s_1, s_2) \rightsquigarrow s\} = \{s' : s' \in \mathcal{C}\llbracket c'\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta}\} = \mathcal{C}\llbracket c'\rrbracket^{\mathcal{D}\llbracket\mathrm{D}\rrbracket^{\delta};\delta}$, as desired.

$\boxed{\mathrm{D}, \delta \vdash_C c \parallel \text{Success} \xrightarrow{\tau} c}$ As the previous case.

$\boxed{\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} d}{\mathrm{D}, \delta \vdash_C c; c' \xrightarrow{\tau} d; c'}}$ We have $\mathcal{C}[\![c; c']\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta} = \{ss' : s \in \mathit{Tr}, s' \in \mathit{Tr} \mid s \in \mathcal{C}[\![c]\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta} \wedge s' \in \mathcal{C}[\![c']\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta}\}$. But by the IH, we gather that $\mathcal{C}[\![c]\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta} = \mathcal{C}[\![d]\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta}$, whence $\{ss' : s \in \mathit{Tr}, s' \in \mathit{Tr} \mid s \in \mathcal{C}[\![c]\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta} \wedge s' \in \mathcal{C}[\![c']\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta}\} = \{ss' : s \in \mathit{Tr}, s' \in \mathit{Tr} \mid s \in \mathcal{C}[\![d]\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta} \wedge s' \in \mathcal{C}[\![c']\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta}\}$.

$\boxed{\mathrm{D}, \delta \vdash_C \mathrm{Success}; c' \xrightarrow{\tau} c'}$ As the previous case, noting that $\mathcal{C}[\![\mathrm{Success}]\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta} = \{\langle\rangle\}$.

$\boxed{\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} c'}{\delta \vdash_C \mathrm{letrec}\,\mathrm{D}\ \mathbf{in}\ c \xrightarrow{\tau} \mathrm{letrec}\,\mathrm{D}\ \mathbf{in}\ c'}}$ Here, $\mathcal{C}[\![\mathrm{letrec}\,\mathrm{D}'\ \mathrm{in}\ c]\!]^{\delta} = \mathcal{C}[\![c]\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta}$ for some D'. By the IH, we have $\mathcal{C}[\![c]\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta} = \mathcal{D}[\![c']\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta}$ and hence $\mathcal{C}[\![\mathrm{letrec}\,\mathrm{D}\ \mathrm{in}\ c]\!]^{\delta} = \mathcal{C}[\![\mathrm{letrec}\,\mathrm{D}\ \mathrm{in}\ c']\!]^{\delta}$, as desired.

□

Proposition 12 "If". To show: For all $\mathrm{D}, \delta, c, c'$: $\mathrm{D}, \delta \vDash c = \mathrm{Success}$ if $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau^*} \mathrm{Success}$.

A trivial induction on the length of the τ-reduction sequences using Proposition 11 furnishes $\mathcal{C}[\![c]\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta} = \mathcal{C}[\![\mathrm{Success}]\!]^{\mathcal{D}[\![\mathrm{D}]\!]^{\delta};\delta}$, and the result follows.

"Only if": To show: For all $\mathrm{D}, \delta, c, c'$: $\mathrm{D}, \delta \vDash c = \mathrm{Success}$ only if $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau^*} \mathrm{Success}$. Note that $\mathrm{D}, \delta \vDash c = \mathrm{Success}$ implies $\mathrm{D} \models c$ nullable and, by Proposition 5, $\mathrm{D} \vdash c$ nullable. Consequently, the result follows if we can prove $\mathrm{D} \vdash c$ nullable $\Longrightarrow$ $(\mathrm{D}, \delta \vDash c = \mathrm{Success} \Longrightarrow \mathrm{D}, \delta \vdash_C c \xrightarrow{\tau^*} \mathrm{Success})$.

Claim: The set of derivations of $\mathrm{D} \vdash c$ nullable is finite.

Proof of claim: Observe that all contracts c' that can occur in a derivation of $\mathrm{D} \vdash c$ nullable must occur in either D or c. Furthermore there no contract can occur twice on any path in a derivation tree. Thus the depth of any derivation tree of $\mathrm{D} \vdash c$ nullable is bounded by the sum of the sizes of D and c. Since, furthermore, the outdegree of derivation trees is bounded by 2, we can conclude that the set of derivation trees for $\mathrm{D} \vdash c$ nullable is finite.

Let us define the *maximal derivation depth* of a derivable judgement $\mathrm{D} \vdash c$ nullable to be the maximal depth of any of the derivations of $\mathrm{D} \vdash c$ nullable. By the claim above this is well-defined.

We shall now prove by Noetherian (well-founded) induction on the maximal derivation depth of $\mathrm{D} \vdash c$ nullable that $\mathrm{D}, \delta \vDash c = \text{Success}$ implies $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau^*} \text{Success}$. We do this by cases on the syntax of c.

- Success.
  In this case, we have $\mathrm{D}, \delta \vdash_C \text{Success} \xrightarrow{\tau^0} \text{Success}$ and we are done.
- $c_1 + c_2$.
  Let $\mathrm{D} \vdash c_1 + c_2$ nullable with maximal derivation depth n. Assume $\mathrm{D}, \delta \vDash c_1 + c_2 = \text{Success}$. It follows that both $\mathrm{D}, \delta \vDash c_1 = \text{Success}$ and $\mathrm{D}, \delta \vDash c_1 = \text{Success}$ and thus $\mathrm{D} \models c_1$ nullable and $\mathrm{D} \models c_2$ nullable. By Proposition 5 we thus have that $\mathrm{D} \vdash c_1$ nullable and $\mathrm{D} \vdash c_2$ nullable. Since both $\mathrm{D} \vdash c_1$ nullable and $\mathrm{D} \vdash c_2$ nullable yield a derivation of $\mathrm{D} \vdash c_1 + c_2$ nullable it follows that the maximal derivation depths of $\mathrm{D} \vdash c_1$ nullable and $\mathrm{D} \vdash c_2$ nullable are less than n. Consequently we can apply the induction hypotheses to them and obtain that $\mathrm{D}, \delta \vdash_C c_1 \xrightarrow{\tau^*} \text{Success}$ and $\mathrm{D}, \delta \vdash_C c_2 \xrightarrow{\tau^*} \text{Success}$. By induction on the combined length of the two reductions, it can now be shown that $\mathrm{D}, \delta \vdash_C c_1 + c_2 \xrightarrow{\tau^*} \text{Success} + \text{Success}$. Now, we can apply Rule

$$\mathrm{D}, \delta \vdash_C \text{Success} + \text{Success} \xrightarrow{\tau} \text{Success}$$

  and we are done.
- $c_1 \parallel c_2$.
  Let $\mathrm{D} \vdash c_1 \parallel c_2$ nullable with maximal derivation depth n. Assume $\mathrm{D}, \delta \vDash c_1 \parallel c_2 = \text{Success}$. It follows that both $\mathrm{D}, \delta \vDash c_1 = \text{Success}$ and $\mathrm{D}, \delta \vDash c_1 = \text{Success}$.

  $\mathrm{D} \vdash c_1 \parallel c_2$ nullable can only be derived from $\mathrm{D} \vdash c_1$ nullable and $\mathrm{D} \vdash c_2$ nullable, each of which consequently has maximal derivation depth less than n. We can thus apply the induction hypothesis to $\mathrm{D} \vdash c_1$ nullable and $\mathrm{D} \vdash c_2$ nullable, which yield that $\mathrm{D}, \delta \vdash_C c_1 \xrightarrow{\tau^*} \text{Success}$ and $\mathrm{D}, \delta \vdash_C c_2 \xrightarrow{\tau^*} \text{Success}$. By induction on the combined length of the two reductions it can now be shown that $\mathrm{D}, \delta \vdash_C c_1 \parallel c_2 \xrightarrow{\tau^*} \text{Success} \parallel \text{Success}$. Using one of the two rules for eliminating a parallel Success, we thus arrive at $\mathrm{D}, \delta \vdash_C c_1 \parallel c_2 \xrightarrow{\tau^*} \text{Success}$ and we are done.
- $c_1; c_2$.
  Similar to above.
- $f(\vec{a})$.
  Let $\mathrm{D} \vdash f(\vec{a})$ nullable with maximal derivation depth n where $(f(\vec{X}) = c) \in \mathrm{D}$. Assume $\mathrm{D}, \delta \vDash f(\vec{a}) = \text{Success}$. It follows that $\mathrm{D}, \delta \vDash c[\vec{v}/\vec{X}] = \text{Success}$ where $\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^\delta$. Since $\mathrm{D} \vdash f(\vec{a})$ nullable can only be derived from $\mathrm{D} \vdash c$ nullable it follows that the maximal derivation depth of $\mathrm{D} \vdash c$ nullable is less than n. Furthermore, it can be shown that for each derivation of $\mathrm{D} \vdash c$ nullable there

is a derivation of $\mathrm{D} \vdash c[\vec{v}/\vec{X}]$ nullable equal depth. Consequently the maximal derivation depth of $\mathrm{D} \vdash c[\vec{v}/\vec{X}]$ nullable is also less than n, and we can apply the induction hypothesis to obtain that $\mathrm{D}, \delta \vdash_C c[\vec{v}/\vec{X}] \xrightarrow{\tau^*}$ Success. Prefixing this reduction sequence with Rule $\mathrm{D}, \delta \vdash_C f(\vec{a}) \xrightarrow{\tau^*} c[\vec{v}/\vec{X}]$ we arrive at $\mathrm{D}, \delta \vdash_C f(\vec{a}) \xrightarrow{\tau^*}$ Success and we are done.
- Other cases. In all other cases $\mathrm{D} \vdash c$ nullable is not derivable.

Lemma 13 We show the lemma by proving the stronger result that τ-reduction is normalizing and confluent.

First we show that all guarded contracts are τ-normalizing, i.e., there exists a (τ-normal form) c' s.t. $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau^*} c'$ and for no c'' $\mathrm{D}, \delta \vdash_C c' \xrightarrow{\tau} c''$. Proof by induction on the (minimal) height of the derivation of guardedness of c. Use Figure 10.

- Assume $\mathrm{D} \vdash$ Success guarded. Clearly, there is no rule such that Success reduces via τ, so we must already have a τ-normal form.
- Assume $\mathrm{D} \vdash$ Failure guarded. Analogous to the case above.
- Assume $\mathrm{D} \vdash \mathrm{transmit}(\vec{X} \mid P).\, c$ guarded. Analogous to the cases above.
- Assume $\dfrac{\mathrm{D} \vdash c \text{ guarded} \quad (f(\vec{X}) = c) \in \mathrm{D}}{\mathrm{D} \vdash f(\vec{a}) \text{ guarded}}$. Since $(f(\vec{X}) = c) \in \mathrm{D}$ we can build a derivation of $\mathrm{D}, \delta \vdash_C f(\vec{a}) \xrightarrow{\tau} c[\vec{v}/\vec{X}]$ where $\vec{v} = \mathcal{Q}[\![\vec{a}]\!]^\delta$. It is left to show that $c[\vec{v}/\vec{X}]$ is τ-normalizing.

  Claim: For any derivation of $\mathrm{D} \vdash c$ guarded there is a derivation of $\mathrm{D} \vdash c[\vec{}/\vec{X}]$ guarded of equal height.

  Proof of claim: By induction on guardedness.

  By the above claim it follows that the height of derivation of $\mathrm{D} \vdash c[\vec{}/\vec{X}]$ guarded is the same as the height of $\mathrm{D} \vdash c$ guarded, which is less than the height of $\mathrm{D} \vdash f(\vec{a})$ guarded. Applying the induction hypothesis to $\mathrm{D} \vdash c[\vec{}/\vec{X}]$ guarded we get that $c[\vec{}/\vec{X}]$ is τ-normalizing and since $\mathrm{D}, \delta \vdash_C f(\vec{a}) \xrightarrow{\tau} c[\vec{}/\vec{X}]$ also that $f(\vec{a})$ is τ-normalizing.
- Assume $\dfrac{\mathrm{D} \vdash c \text{ guarded} \quad \mathrm{D} \vdash c' \text{ guarded}}{\mathrm{D} \vdash c + c' \text{ guarded}}$. There are three cases to consider.

(1) Suppose $\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} d}{\mathrm{D}, \delta \vdash_C c + c' \xrightarrow{\tau} d + c'}$. By the IH we are done.

(2) Suppose $\dfrac{\mathrm{D}, \delta \vdash_C c' \xrightarrow{\tau} d'}{\mathrm{D}, \delta \vdash_C c + c' \xrightarrow{\tau} c + d'}$. Again by the IH we are done.

(3) Suppose $\mathrm{D}, \delta \vdash_C$ Success + Success $\xrightarrow{\tau}$ Success. But Success is already a τ normal form so we are done.

- Assume $\dfrac{\mathrm{D} \vdash c \text{ guarded} \quad \mathrm{D} \vdash c' \text{ guarded}}{\mathrm{D} \vdash c \parallel c' \text{ guarded}}$. There are four cases to consider.

(1) Suppose $\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} d}{\mathrm{D}, \delta \vdash_C c \parallel c' \xrightarrow{\tau} d \parallel c'}$. By the IH on the two premisses of the derivation of guardedness of $c \parallel c'$, we obtain what was required.

(2) Suppose $\dfrac{\mathrm{D}, \delta \vdash_C c' \xrightarrow{\tau} d'}{\mathrm{D}, \delta \vdash_C c \parallel c' \xrightarrow{\tau} c \parallel d'}$. Analogous to the case just shown.

(3) Suppose $\mathrm{D}, \delta \vdash_C \text{Success} \parallel c \xrightarrow{\tau} c$. By assumption $\mathrm{D} \vdash c$ guarded and by the IH we are done.

(4) Suppose $\mathrm{D}, \delta \vdash_C c \parallel \text{Success} \xrightarrow{\tau} c$. By assumption, we have $\mathrm{D} \vdash c'$ guarded and by the IH we are done.

- Assume $\dfrac{\mathrm{D} \vdash c \text{ guarded} \quad \mathrm{D} \vdash c' \text{ guarded}}{\mathrm{D} \vdash c; c' \text{ guarded}}$. There are two cases to consider.

(1) Suppose $\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} d}{\mathrm{D}, \delta \vdash_C c; c' \xrightarrow{\tau} d; c'}$. Easy, by the IH.

(2) Suppose $\mathrm{D}, \delta \vdash_C \text{Success}; c' \xrightarrow{\tau} c'$. Immediate by the IH.

Second, we prove confluence by showing that the diamond property holds for τ-reduction, i.e. if $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} c'$ and $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} c''$, then there exists a d with $\mathrm{D}, \delta \vdash_C c' \xrightarrow{\tau^=} d$ and $\mathrm{D}, \delta \vdash_C c'' \xrightarrow{\tau^=} d$. The proof is by induction on the derivation of $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} c'$.

- $\dfrac{(f(\vec{X}) = c) \in \mathrm{D}, \vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta}}{\mathrm{D}, \delta \vdash_C f(\vec{a}) \xrightarrow{\tau} c[\vec{v}/\vec{X}]}$.

  No other rules match any subterm of $f(\vec{a})$, and we must hence have $c' = c''$, whence the result follows.

- $\dfrac{\mathrm{D}, \delta \vdash_C c_1 \xrightarrow{\tau} d_1}{\mathrm{D}, \delta \vdash_C c_1 + c_2 \xrightarrow{\tau} d_1 + c_2}$.

  If the τ-rewrite step $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} c''$ takes place inside c, we have $c'' = c_1'' + c_2$, and the IH furnishes a d_1' such that $\mathrm{D}, \delta \vdash_C c_1'' \xrightarrow{\tau^=} d_1'$ and $\mathrm{D}, \delta \vdash_C d_1 \xrightarrow{\tau^=} d_1'$. We hence have $\mathrm{D}, \delta \vdash_C c' \xrightarrow{\tau^=} d_1' + c_2$ and $\mathrm{D}, \delta \vdash_C c'' \xrightarrow{\tau^=} d_1' + c_2$, as desired.

- $\dfrac{\mathrm{D}, \delta \vdash_C c_2 \xrightarrow{\tau} d_2}{\mathrm{D}, \delta \vdash_C c_1 + c_2 \xrightarrow{\tau} c_1 + d_2}$.

  As the previous case.

- $\mathrm{D}, \delta \vdash_C \text{Success} + \text{Success} \xrightarrow{\tau} \text{Success}$.

  In this case, we must have $c' = c''$, and the result follows.

- $\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} d}{\mathrm{D}, \delta \vdash_C c \parallel c' \xrightarrow{\tau} d \parallel c'}$.

  Exactly as the case $\dfrac{\mathrm{D}, \delta \vdash_C c_1 \xrightarrow{\tau} d_1}{\mathrm{D}, \delta \vdash_C c_1 + c_2 \xrightarrow{\tau} d_1 + c_2}$.

- $\dfrac{\mathrm{D}, \delta \vdash_C c' \xrightarrow{\tau} d'}{\mathrm{D}, \delta \vdash_C c \parallel c' \xrightarrow{\tau} c \parallel d'}$. As the previous case.
- $\mathrm{D}, \delta \vdash_C \text{Success} \parallel d \xrightarrow{\tau} d$.

  In this case, $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} c''$ must be an application of either of the rules $\mathrm{D}, \delta \vdash_C c \parallel \text{Success} \xrightarrow{\tau} c$, or $\dfrac{\mathrm{D}, \delta \vdash_C d \xrightarrow{\tau} d'}{\mathrm{D}, \delta \vdash_C c_1 + d \xrightarrow{\tau} c_1 + d'}$. In the first case, we have $c' = \text{Success} = c''$, and we are done. in the second case, we have $c_1 = \text{Success}$, and thus $\mathrm{D}, \delta \vdash_C d \xrightarrow{\tau} d'$ and $\mathrm{D}, \delta \vdash_C c_1 + d \xrightarrow{\tau} d'$, as desired.
- $\mathrm{D}, \delta \vdash_C c \parallel \text{Success} \xrightarrow{\tau} c$.

  Symmetric to the previous case.
- $\dfrac{\mathrm{D}, \delta \vdash_C c_1 \xrightarrow{\tau} d_1}{\mathrm{D}, \delta \vdash_C c_1; c_2 \xrightarrow{\tau} d_1; c_2}$.

  If the reduction step $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} c''$ takes place inside c_1, then $c'' = d_1''; c_2$, and the IH furnishes existence of a d_1' such that $\mathrm{D}, \delta \vdash_C d_1 \xrightarrow{\tau^=} d_1'$ and $\mathrm{D}, \delta \vdash_C d_1 \xrightarrow{\tau^=} d_1'$. Then, $d_1'; c_2$ is a common τ-reduct of c' and c'', and the desired result follows. Otherwise, $\mathrm{D}, \delta \vdash_C c \xrightarrow{\tau} c''$ is an application of the rule $\mathrm{D}, \delta \vdash_C \text{Success}; c' \xrightarrow{\tau} c'$, which is impossible, since Success is a τ-normal form, i.e. it cannot be the case that $\mathrm{D}, \delta \vdash_C c_1 \xrightarrow{\tau} d_1$.
- $\mathrm{D}, \delta \vdash_C \text{Success}; c' \xrightarrow{\tau} c'$.

  Symmetric to the previous case.

□

Theorem 14 "If": By induction on the height of the derivation of $\mathrm{D}, \delta \vdash_C c \xrightarrow{\vec{de}} c'$. "Only if": By induction on the height of the derivation of $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} c'$.

Proving "Only if": (note that we only consider non-τ-derivations)

- Assume $\mathrm{D}, \delta \vdash_N \text{Success} \xrightarrow{e} \text{Failure}$. From the reduction semantics of we see that there is just one possible reduction $\mathrm{D}, \delta \vdash_C \text{Success} \xrightarrow{e} c'$ giving $c' = \text{Failure}$ so $c'' = \text{Failure}$.
- Assume $\mathrm{D}, \delta \vdash_N \text{Failure} \xrightarrow{e} \text{Failure}$. Analogous.
- Assume $\dfrac{\delta \oplus \vec{X} \mapsto \vec{v} \models P, \vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta}}{\mathrm{D}, \delta \vdash_N \text{transmit}(\vec{X} \mid P).\, c \xrightarrow{\text{transmit}(\vec{v})} c[\vec{v}/\vec{X}]}$. Again we see that there is a unique reduction of the $\text{transmit}(\vec{X}|P).\, c$-contract and we have,

$$\frac{\delta \oplus \vec{X} \mapsto \vec{v} \models P, \vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta}}{\mathrm{D}, \delta \vdash_C \text{transmit}(\vec{X} \mid P).\, c \xrightarrow{\text{transmit}(\vec{v})} c[\vec{v}/\vec{X}]}$$

by which we conclude $c'' = c[\vec{v}/\vec{X}]$.

- Assume $\dfrac{\delta \oplus \vec{X} \mapsto \vec{v} \models P, \vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta}}{\mathrm{D}, \delta \vdash_N \mathrm{transmit}(\vec{X}|P).\, c \xrightarrow{\mathrm{transmit}(\vec{v})} \mathrm{Failure}}$. Analogous.
- Assume $\dfrac{\mathrm{D}, \delta \vdash_N c \xrightarrow{e} d}{\mathrm{D}, \delta \vdash_N c \parallel c' \xrightarrow{e} d \parallel c'}$. By the IH we gather that $\mathrm{D}, \delta \vdash_C c \xrightarrow{\vec{d}e} d$.

  We can extend $\vec{d}$ with l and build a *unique* derivation $\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\vec{d}e} d}{\mathrm{D}, \delta \vdash_C c \parallel c' \xrightarrow{l\vec{d}e} d \parallel c'}$.
- Assume $\dfrac{\mathrm{D}, \delta \vdash_N c' \xrightarrow{e} d'}{\mathrm{D}, \delta \vdash_N c \parallel c' \xrightarrow{e} c \parallel d'}$. Analogously by extending $\vec{d}$ with r.
- Assume $\dfrac{\mathrm{D}, \delta \vdash_N c \xrightarrow{e} d}{\mathrm{D}, \delta \vdash_N c; c' \xrightarrow{e} d; c'}$. By the IH we have a derivation $\mathrm{D}, \delta \vdash_C c \xrightarrow{e} d$.

  Thus we can construct the unique derivation $\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{e} d}{\mathrm{D}, \delta \vdash_C c; c' \xrightarrow{e} d; c'}$.

Proving "If":

- Assume $\mathrm{D}, \delta \vdash_C \mathrm{Success} \xrightarrow{e} \mathrm{Failure}$. There is no $\vec{d}$ in this case, and we can immediately build $\mathrm{D}, \delta \vdash_N \mathrm{Success} \xrightarrow{e} \mathrm{Failure}$, also choosing no τ-transitions for the first part.
- Assume $\mathrm{D}, \delta \vdash_C \mathrm{Failure} \xrightarrow{e} \mathrm{Failure}$. Analogous.
- Assume $\dfrac{\delta \oplus \vec{X} \mapsto \vec{v} \models P, \vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta}}{\mathrm{D}, \delta \vdash_C \mathrm{transmit}(\vec{X} \mid P).\, c \xrightarrow{\mathrm{transmit}(\vec{v})} c[\vec{v}/\vec{X}]}$. Take no $\vec{d}$ and choose no τ-transitions. Then immediate.
- Assume $\dfrac{\delta \oplus \vec{X} \mapsto \vec{v} \not\models P, \vec{v} = \mathcal{Q}[\![\vec{a}]\!]^{\delta}}{\mathrm{D}, \delta \vdash_C \mathrm{transmit}(\vec{X}|P).\, c \xrightarrow{\mathrm{transmit}(\vec{a})} \mathrm{Failure}}$. Analogous.
- Assume $\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\vec{d}e} c'}{\mathrm{D}, \delta \vdash_C c + d \xrightarrow{\mathrm{f}\vec{d}e} c'}$. Must build a derivation of

$$\mathrm{D}, \delta \vdash_N c + d \xrightarrow{\tau^*} c'' \xrightarrow{e} c'.$$

  By IH: $\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau^*} c'' \xrightarrow{e} c'$. So we just need the first part. Clearly, we have $\mathrm{D}, \delta \vdash_N c + d \xrightarrow{\tau} c$. Thus, by choosing $c = c''$ and exactly one τ-transition, we are done.
- Assume $\dfrac{\mathrm{D}, \delta \vdash_C d \xrightarrow{\vec{d}e} d'}{\mathrm{D}, \delta \vdash_C c + d \xrightarrow{\mathrm{s}\vec{d}e} d'}$. Analogous.

- Assume $\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{\vec{d}e} d}{\mathrm{D}, \delta \vdash_C c \parallel c' \xrightarrow{\mathrm{l}\vec{d}e} d \parallel c'}$. By using the IH, taking $c = c''$, and making no τ-transitions in the first part, we are done.
- Assume $\dfrac{\mathrm{D}, \delta \vdash_C c' \xrightarrow{\vec{d}e} d'}{\mathrm{D}, \delta \vdash_C c \parallel c' \xrightarrow{\mathrm{r}\vec{d}e} c \parallel d'}$. Analogous.
- Assume $\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{e} d}{\mathrm{D}, \delta \vdash_C c; c' \xrightarrow{e} d; c'}$. Analogous.
- Assume $\dfrac{\mathrm{D}, \delta \vdash_C c \xrightarrow{e} c'}{\delta \vdash_C \mathrm{letrec\,D\ in}\ c \xrightarrow{e} \mathrm{letrec\,D\ in}\ c'}$. Analogous.

It is obvious that, if $\vec{d}$ exists in the above case, it is unique. Furthermore, for all c'' such that $\mathrm{D}, \delta \vdash_C c \xrightarrow{\vec{d}e} c''$ we have $c' = c''$. Again, it is obvious.

□

References

[AE03] Jesper Andersen and Ebbe Elsborg. Compositional specification of commercial contracts. M.S. term project, December 2003.

[Ark02] A. Arkin. Business process modeling language, 2002.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[BM02] J.C.M. Baeten and C.A. Middelburg. *Process Algebra with Timing.* Springer, 2002.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[Con71] J. H. Conway. *Regular Algebra and Finite Machines.* Chapman and Hall, 1971.

[Ebe02] Jean-Marc Eber. Personal communication, June 2002.

[GM00] Guido Geerts and William E. McCarthy. The ontological foundations of rea enterprise information systems. Unpublished, August 2000.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, 1985.

[JE03] Simon Peyton Jones and Jean-Marc Eber. How to write a financial contract. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming.* Palgrave Macmillan, 2003.

[JES00] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 280–292. ACM Press, 2000.

[KPA03] Kåre J. Kristoffersen, Christian Pedersen, and Henrik R. Andersen. Runtime verification of timed LTL using disjunctive normalized equation systems. Unpublished, September 2003.

[KPA04] K.J Kristoffersen, C. Pedersen, and H.R. Andersen. Checking temporal business rules. In *Proceedings of the First International REA Workshop*, 2004.

[lex] http://www.lexifi.com.

[McC82] William E. McCarthy. The REA accounting model: A generalized framework for accounting systems in a shared data environment. *The Accounting Review*, LVII(3):554–578, July 1982.

[Mil89] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice-Hall, 1989.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-calculus.* Cambridge University Press, 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, September 1992.

[nav] http://www.navision.dk.

[sap] http://www.sap.com.

[sim] http://www.simcorp.com.

[SMTA95] Munindar P. Singh, Greg Meredith, Christine Tomlinson, and Paul C. Attie. An event algebra for specifying and scheduling workflows. In *Database Systems for Advanced Applications*, pages 53–60, 1995.

[vdADtHW02] W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and P. Wohed. Pattern-based analysis of BPML (and WSCI). Technical Report FIT-TR-2002-05, Queensland University, 2002.

[vdAvH02] Wil van der Aalst and Kees van Hee. *Workflow Management—Models, Methods, and Systems.* MIT Press, 2002.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.

SMAWL: A SMAll Workflow Language Based on CCS

Christian Stefansen

Department of Computer Science, University of Copenhagen

Abstract

While most workflow products and researchers have used Petri nets as the underlying concurrency model, there has been a push towards algebraic methods (for purposes of reasoning) and models that provide first class primitives for mobility (i.e. channel passing, locality, and process passing). Today the *de facto* benchmark for workflow models is a large number of workflow patterns (control-flow patterns, dataflow patterns, interaction patterns, and resource patterns) compiled by van der Aalst. We demonstrate that all control-flow patterns can be expressed in a process calculus (Calculus of Communicating Systems, CCS) and subject the patterns to a closer analysis. We then consider a taxonomy of workflow languages and discuss some important languages with respect to this taxonomy. Last, we present a small workflow language, SMAWL, that aims to accommodate all control-flow patterns while maintaining a strict formal semantics through its direct connection to CCS.

(This is a merged version of material from a two technical reports and a conference paper of 2005 [33–35].)

1 Introduction

In this chapter we will look a workflows and how to express them in a way that is representable in a computer. First workflows are defined (Section 2), then we consider the problems of having a loose or no workflow handling (Section 3). This leads to the design criteria for a workflow model. We then take a small detour and compare workflows with conventional programs to establish an understanding of the differences (Section 5). Afterwards, a workflow model based on Calculus of Communicating Systems is presented and evaluated with respect to the design criteria (Section 6).

The text is interspersed with comments on two of the most important process languages available: YAWL (*Yet Another Workflow Language*) and WS-BPEL (*Web Services – Business Process Execution Language*; henceforth simply BPEL). These two languages alone cover a very broad spectrum of workflow issues. YAWL [37] is an academically developed workflow language, which is probably more expressive than any of the industry offerings, to which it has been compared. BPEL [36] is the industry flagship standard for business process execution. There are many other standards and products. For an overview see van der Aalst [41] and p. 172 [42].

1.1 Contributions

The main contributions of this chapter are:

(1) A CCS encoding of the 20 control flow patterns in Table 1
(2) An analysis of the 20 patterns, and a description and encoding of two new patterns: *(8a) N-out-of-M Merge* and *(16a) Deferred Multiple Choice*
(3) A presentation of a speculative *overlay* operator and its use for certain workflow scenarios
(4) A short discussion of using CCS vs. π-calculus for workflow description languages.
(5) A further analysis of control patterns and directions for future research.

2 Definitions

Two important textbooks on workflow modeling venture to give definitions, and these are quoted below to convey an intuition:

> "A business process is a collection of interrelated work tasks, initiated in response to an event, that achieves a specific result for the customer of the process." — Sharp and McDermott [32]
>
> "A workflow comprises cases, resources, and triggers that relate to a particular process" and "[t]he definition of a process indicates which tasks must be performed—and in what order—to successfully complete a case. In other words all possible routes are mapped out. A process consists of tasks, conditions, and subprocesses. By using AND-splits, AND-joins, OR-splits, and OR-joins, parallel and alternative flows can be defined. Subprocesses also consist of tasks, condi-

> tions, and possible further subprocesses. The use of subprocesses can enable the hierarchical structuring of complex processes." — van der Aalst and van Hee [42]

Wikipedia [47] provides this definition:

> "Workflow is the operational aspect of a work procedure: how tasks are structured, who performs them, what their relative order is, how they are synchronized, how information flows to support the tasks and how tasks are being tracked. As the dimension of time is considered in Workflow, Workflow considers 'throughput' as a distinct measure. Workflow problems can be modeled and analyzed using Petri nets."

Readers interested in a Petri net-based approach are referred to van der Aalst and van Hee's "Workflow Management" [40]. Sharp and McDermott give a treatise with a focus on the business aspects of workflow modeling in "Workflow Modeling" [32], which also ties in with managerial vogues such as *TQM* [1] and *Kaizen*.

3 Why Express Workflows Formally?

The simplest conceivable workflow system would simply consist of task list, where tasks would be checked off when completed. To complete the workflow the user would either draw on domain knowledge to figure out what tasks should precede others or proceed by trial and error, suspending tasks until the necessary preconditions were satisfied. This approach works in very simple scenarios such as personal task lists but fails to scale to larger workflows with more agents:

- **No aid in adhering to established practice.** If an established best practice exists, the system has no way of helping the user follow this. Designing and accounting for best practices is becoming increasingly important with the introduction of legislative requirements such as the Sarbanes–Oxley Act.
- **No simple/automatic way of deciding what to do next.** The interdependence between tasks is not represented and so the required task- and data-dependency analysis must be brought to bear solely on the user's domain knowledge in an *ad hoc* fashion. This makes it more difficult to delegate and divide labor—especially to new employees with limited training.
- **No way to express new tasks created as result of other tasks' outcomes, repeating tasks or choice structure.** Again, all this information must be learned, maintained, and remembered by the users.

[1] Total Quality Management

- **No simple/automatic way of deciding how to best delegate tasks at hand.** Delegation can be based on e.g. experience/skills, current workload, expected processing time, confidentiality, one handler pr. client, or even internal auctioning.
- **Training users is more costly.** Very little or no domain knowledge is explicitly represented so proper use relies heavily on correct user understanding of the entire workflow.
- **Changing business processes is costly and potentially resultless.** In the absence of a system providing guidance and support, users may revert to old practices. Changing processes is costly, because the current processes must first be mapped through interviews, and users must be taught the new processes through practice and repetition rather than by system guidance.

Since some knowledge of the structural constraints of the tasks is needed in any case to perform a number of routine analyses, it seems reasonable to suggest that these constraints are captured explicitly in the workflow system. Some additional benefits of an explicit representation of workflows are:

- **Formal analysis.** This foremost argument for domain-specific languages is this: Programs in a DSL double as data that can be analyzed and transformed. This means that a workflow written in a DSL for workflows immediately lends itself to analyses, pertaining to e.g. reachability, forward/backward slicing, deadlock, longest/shortest propagation delays, resources needed, scheduling/planning etc. This analysis can be carried out statically on a program, but also on a running program. Languages with reduction semantics—i.e. an execution semantics where every atomic step in the execution can be represented as a program within that language—are particularly convenient to analyze, because the state is represented explicitly in the program expression.
- **Redesign aid: longest propagation delay, dependency** Formal analysis is not only useful for timely business reporting, but also for redesign. Workflow design tools can help identify long paths, resource-demanding sections etc. in the design process. Thus workflow design tools become an aid for process redesign.
- **Activity-based costing.** In a workflow-driven system, costs are immediately available pr. employee, pr. case, pr. client, pr. tasks type etc. And this is without the extra work of assigning cost dimensions on job scheduling entries as seen in some ERP systems.
- **Design time or even runtime flexibility trade-off.** No one desires an overly rigid workflow system—and no one knows exactly at design time what will be too rigid and what will not. Workflow systems can be normative (allowing the users to skip and jump tasks and their leisure), prescriptive (forcing the user to follow the workflow) or anything in between. The important observation is that

this decision does not have to take place design time. Instead the system could take a flexible stance allowing all violations when first deployed and gradually imposing stronger rules when it has been established that it fits with practice. The trace of events will provide statistics about the tasks skipped, by whom, how often, etc. This will help diagnose if the workflow is ill-designed or the company has problem adhering to its own best practice.

- **(Long term) Support for outsourcing, commissioning, subcontracting.** Describing workflows in a language with an established semantics is the first step towards being able to outsource workflows or parts of workflows. Naturally, a workflow only documents the skeleton of the work that has to be done, and the work descriptions for every work item must be filled out as well.
- **First step towards mobile processes.** Finding a formal representation of workflows is the first step toward mobile business processes. This opens up an interesting domain of business models for semi-automated process auctioning over the Internet.

Having enumerated some of the potential gains, we should also note some of the costs and issues involved:

- Work items are coupled with other systems and legacy systems. Sometimes data stay within the systems, sometimes it follows the workflows. There is a significant challenge in providing interfaces for all systems so that a workflow management system gains access to the case data it needs. In these days of XML, a solution seems to be within reach, though.
- Workflows can easily become too rigid. It is imperative that one begins with a flexible systems and narrows it in only after significant experience.
- There is a potential gap between who designs the workflows and who uses the workflows. Reducing this gap deserves significant attention.

Weighing the costs and benefits it appears that significant gains are possible with formal representations of workflows.

Although as noted a completely unstructured task list is insufficient for the job, it is useful *ab initio* to think of all workflows being unstructured. We then add structure as needed afterwards. Going from simple workflows to structured workflows can be done by design, by discovery or by a hybrid method. Structuring workflows by design means having domain experts and users impose the constraints deemed necessary *a priori* or at runtime. Structuring workflows by discovery is done by *process mining*: Given an existing event log, one tries to discover invariants and recast the invariants as workflow structure. This is often done using a genetic algorithm that randomly generates a number of workflows, ranks them according to a fitness measure, and refines them iteratively [46].

For the remainder of this section we will concentrate on workflows structured by design.

4 Design Criteria, Requirements, and Patterns

In designing a new language a range of considerations must be taken into account. Ultimately, language design is about the convenience of program writing in the language. This is its domain fit, availability of common idioms, precise error messages, good tools, few possibilities to introduce errors, minimal number of artifacts etc. Many design criteria can be read directly out of the goals outlined in the previous section. Since we are designing a domain-specific language, formal analyzability becomes particularly important—it is a major justification for domain-specific languages. This usually means that the language should be as simple as possible. Compositionality is another desirable, albeit not strictly necessary, feature to facilitate automated analysis. A very intriguing new area of research is that of process mining, i.e. recovering processes from event traces. The language should ideally lend itself well to program rediscovery by genetic algorithms, but this property is very hard to quantify. It does seem, however, to go hand in hand with the idea of a language with few and simple constructs.

Apart from the general design principles, there are some concrete requirements to address. The idea of *patterns* was popularized in Gamma et al.'s seminal book "Design Patterns" [12]. This sparked a maelstrom of patterns in a variety of research areas until some saturation was achieved. The idea was also applied to the workflow domain: in their seminal 2002–2004 work van der Aalst *et al.* introduced the idea of control-flow patterns (*née* workflow patterns) [41,40,37]. The result was 20 control-flow patterns identified empirically in current workflows, workflow systems, and workflow standards. Since then this body of work has been extended with data patterns, resource patterns, and most recently service interaction patterns. Moreover, the patterns have become the *de facto* standard for workflow systems benchmarking. This means that as a bare minimum, one must take into account the patterns, and for each of them attempt to cover it or deliver a reasonable explanation for its omission. While seemingly straightforward, this process is impaired somewhat by the current state of the patterns. The patterns lack a formal foundation and are in many cases ambiguously described. Hence we are faced with the challenge of interpreting the patterns as well.

This section deals almost exclusively with control flow patterns. Incorporating data patterns, resource patterns and service interaction patterns in our language is outside the scope of this work.

5 Workflows as Programs

It would seem that workflows could be expressed in a structured programming language such as Java, and indeed many workflows are implicitly embedded in existing programs. However, it is useful to examine the differences more closely.

Goto revisited Workflows often use *arbitrary cycles*, a pattern akin to goto in structured programming. Goto is traditionally considered bad programming style for two reasons: (1) it makes programs more difficult to understand and debug: if the program may jump to the current location from any other location, very little is known about the state at that point; (2) goto can be eliminated through the combined use of while, if, and function calls.

These two objections are present, albeit less saliently so, for workflows as well. Workflows are most often described using graphical tools meaning that a goto is *explicitly* represented as an arc. Debugging is therefore made somewhat simpler because one immediately can examine the origins of any incoming arcs. Workflow systems should also support equivalents to whiles, ifs, and function calls making it possibly to rewrite the workflow without use of goto. However, for graphical tools it is not always desirable to factor out parts of the workflow in small functions because this may obscure the overall structure of the workflow. This can be alleviated, though, with a language that uses functions instead of goto and displays functions inline or abstracts away from them at the designer's discretion.

Concurrent execution The workflow designer strives to parallelize as much as possible because this affords flexibility: it minimizes the longest path, permitting a runtime trade-off between completion time and resource load at the discretion of workflow manager. It gives flexibility in scheduling because more tasks and sub-workflows can be started without waiting. For the same reason it allows better resource exploitation and less distribution constraints. On the downside, the subtleties of concurrent programming, including deadlocks, livelocks and race conditions, persist.

Although it seems natural to require explicit concurrency constructs in the workflow model, this is not strictly necessary. An implicitly concurrent model simply specifying sequential constraints such as

$$\begin{aligned} & a\,\texttt{before}\,b, \\ & all\,[a_1,\ldots,a_n]\,\texttt{before}\,b \\ \text{or}\ & one\,of\,[a_1,\ldots,a_n]\,\texttt{before}\,b \end{aligned}$$

could conceivably express the same. In conventional programming one often employs parallelism for performance reasons starting with a sequential program and splitting it up. In workflow design one approach is to begin with a fully parallel workflow and impose constraints only as they become necessary. These observations suggest a programming model where concurrency is the rule rather than the exception.

Delegation Delegation is the allocation of tasks to agents. When many workflows are running, a number of tasks will be available for allocation and a number of agents—human or not—will be ready to take on tasks according to their skill level, workload, availability, speed etc. In the programming language analogy, workflows are communicating multithreaded programs running inside a workflow engine that can call out on several processors to complete the atomic tasks being solicited. Thus, while not directly a part of workflow programming model, delegation affects the architecture of a workflow system. The interplay is best seen in the difference between *explicit* (or *data-dependent*) *choices* made by the workflow engine and *implicit* (or *deferred*) *choices* made implicitly by soliciting both branches for allocation and removing the non-chosen branch, once a task has been started in the other branch. The delegation layer sits between the agents and the running workflows, and essentially works as a scheduler. The solicitation part is the defining difference between delegation and operating system scheduling on multi-processor systems: the tasks may be delegated, solicited or even auctioned off to the processors (agent).

Cancellation Cancellation means aborting a selected number of threads and/or activities and returning to a well-defined state. It has proven particularly tricky to imitate in current workflow systems [40,37] large because they were based on Petri nets, flow charts, etc. that do not have a natural notion of exceptions. Exceptions in distributed systems in general entail serious considerations as multiple peers attempt to agree on a common perception of the distributed state.

There are some simplifying assumption to be made, however. First, it is relatively straightforward to return all threads inside the workflow engine to a well-defined state. What remains are the tasks in progress outside the workflow engine itself. In a transactional setting these can be assumed to be willing to accept a cancellation at any point with no further obligations. Unfortunately, this will rarely be the case. Cancellation of an almost completed outside task could easily produce an aftermath in the form of e.g. partial invoicing.

As we go from atomic tasks with a simple request/response interface toward tasks with a more extensive interface, ACID[2] issues become apparent. Cancellation is the pattern that shows this most clearly. Also, should we wish to depart from the idea of a centralized workflow management server, we again face all the problems of distributed transactions. Most likely, the solution is a hybrid. Internal workflows can be canceled easily and outsourced parts will have a more elaborate cancellation protocol.

Unconventional control flow patterns Some control flow patterns spawn a number of threads, but only wait for one of a few of them to finish before continuing. The remaining threads may be allowed to finish or they may be canceled.

Skip, redo, and undo tasks In workflows we can randomly decide to skip, redo, or undo tasks. In conventional programs this is inconceivable, because the the data output of that task would be missing. In a workflow system, we can adopt a more lax attitude and simply prompt the user for it when the need arises. This distinction vanishes, of course, when parts of the workflow are automated. More generally, workflow users often desire to change an ongoing workflow, e.g. add or remove a sequential constraints, add or remove extra tasks, etc. This level flexibility further removes workflows from conventional programming. To allow everything would mean anarchy; it is therefore a challenge to find ways of specifying the degree of flexibility that the user is afforded in terms of runtime changes to a workflow.

Dataflow for only selected tasks Some dataflow patterns break with a venerable tradition, and propose that data flow be completely orthogonal to the control flow, e.g. a task can decide to forward its output to another task—and *only* this

[2] Atomicity (either all or no tasks are completed), consistency (data integrity constraints hold before and after transaction), isolation (intermediate states are invisible to the outside world), and durability (a reported success is not reverted).

other task—several steps down the line. So far no workflow languages have adopted such a liberal data flow model, and the advantages relative to the increase in complexity are not obvious. Certain security scenarios can be neatly expressed with directed data flow, but these can also be solved with access control lists or a role scheme.

6 Control Patterns

We now narrow the focus to consider only control flow patterns.

6.1 Why Consider a Process Calculus

In 2003-2004 there was an intense debate [50,43][3] in the workflow community about the relative merits of Petri nets [23] and π-Calculus [20] for workflow modeling. Nonetheless, very little of this debate has been published or put in writing elsewhere. During the discussions, van der Aalst presented seven challenges [43] to model workflow in π-Calculus. This section (based on a previous paper [34,35]) responds to those challenges by showing how to code the 20 most commonplace workflow patterns in CCS (a subset of π-Calculus), and describes two new workflow patterns that were identified in the process. The applicability of π-Calculus to the workflow modeling domain is briefly discussed and a new *overlay* operator is discussed with applications to workflow descriptions. The central challenge that was posed is this:

> [...] show how existing patterns can be modeled in terms of Pi calculus.

Whatever one's beliefs, a concrete encoding of the 20 workflow patterns provides a basis for a more informed debate and makes it easier for implementers to make a choice of formal foundation. The choice of the π-Calculus here is more or less arbitrary. A vast number of other calculi (e.g. the Join calculus) would be equally interesting.

The π-calculus and its predecessor CCS (Calculus of Communicating Systems) [19] have strong mathematical foundations and have been widely studied for years. This

[3] Only two references may not support the claim of an "intense debate" very well. There were many more papers, most of which were so speculative and agenda-driven that we will not dignify them by reference.

has lead to a plethora of deep theoretical results (see [30] for an overview) as well as applications in programming languages (e.g. PICT [24]), protocol verification, software model checking (e.g. Zing [3,49]), hardware design languages [4], and several other areas.

Workflow description languages, it would seem, have a lot in common with these areas. First, workflows can be thought of as parallel processes. Second, workflows often defy the block structure found in conventional programming. Third, the prospect of doing formal verification on workflows is very relevant and using a process calculus makes algebraic reasoning and algebraic transformation immediately possible due to the large body of research already present. Lastly, although they are somewhat bare in their style, CCS and π-calculus enforce a very strong separation of process and application logic that seems appropriate for workflow modeling.

On the other hand process calculi are highly theoretical constructs that require a great deal of expert knowledge. Workflow description systems should be accessible to anyone possessing a knowledge of the workflow domain being modeled, and so a significant challenge lies in bridging this gap. In particular providing understandable graphical tools and user-friendly abstractions constitutes a pivotal challenge if π-calculus-based systems are to succeed.

In the same vein a fair concern to raise is whether the level of formalism in π-calculus is necessary considering how graphical systems today are already difficult to understand both for users and programmers. First, using π-calculus does not preclude a high-level graphical representation, and clearly more research into this area is warranted. Second, a rigorous foundation remains necessary, because we strive to understand systems *before* we implement them. A well-tested, well-understood foundation will make implementation and subsequent adaptation easier and more routine.

A major strength of the π-calculus is its ability to express passing of channels between nodes and the ability to pass processes (in the higher-order π-calculus, which can be translated down to regular π-calculus). This far, however, only few examples exist that put this capability to use in the workflow domain. It may turn out, though, that channel-passing (and possibly locality) will become highly relevant when addressing service infrastructure and actual implementation of workflow tasks. By choosing π-calculus we stand a better chance of finding a unified foundation for business systems programming.

[4] In a sense, a microprocessor is one huge workflow.

In terms of practical workflow management π-calculus promises seamless integration between workflow systems and existing verification tools (model checking [6,14], behavior types/conformance checking [26,18] etc.). For the end-user this means access to verification tools that will make writing and adapting workflows faster and less error-prone.

In our view it is important that future business systems are built not only to carry out a specific task, but in a way that makes them amenable to automatic source code manipulation and formal verification, and to attain this goal it is desirable to have a small and rigorously defined core model. What we are seeking is not a formalism to use as it is, but the formalism that will provide the most suitable underpinning for future work.

For the purpose of workflow control patterns we can afford ourselves a limited view of the concurrency models available. Many process calculi, including π, Join, CCS, and Ambients whether synchronous or asynchronous retain simplicity by assuming global consensus, i.e. the existence of a global mechanism ensuring that only one process consumes a message sent on a channel with multiple listeners. It is useful to start out with a centralized workflow management system and deal with distribution issues separately later.

The following patterns may look a bit daunting when first presented, but one should remember that they are not what the workflow user/programmer should see any more than object calculus terms are what a Java programmer or UML user sees.

6.2 Modeling Control Flow in CCS

In this section we will use the CCS syntax defined by the following BNF:

$$P ::= \mathbf{0} \;\Big|\; \tau.P \;\Big|\; a?x.P \;\Big|\; a!x.P \;\Big|\; P+P \;\Big|\; (P \mid P) \;\Big|\; \mathsf{new}\ a\ P \;\Big|\; a?{*}x.P$$

$\mathbf{0}$ is the empty process, τ is “some unobservable transition, $a!x$ and $a?$ try to send and receive on channel a, $+$ is choice, $|$ runs processes in parallel, $\mathsf{new}\ a\ P$ protects the channel a from communication outside P, and $a?{*}x.P$ spawns the process P each time a message is received on a. The syntax allows unguarded choice but guards all replicated processes with an input prefix ($a?{*}$). For simplicity we often write $a?x$ instead of $a?x.\mathbf{0}$, and we may omit the name x if it is irrelevant in the context. Capital letters, usually P, Q, R, denote processes, and small letters,

$a, b, c, \ldots$ denote activities. Internal messages are words in small letters, like *ok* and *go*.

The invocation of an activity a can be encoded as $a!.a?$ denoting a simple request/-response mechanism[5]. More sophisticated protocols for monitoring activity progress can be added, but here the statuses started/finished will suffice. Often if a is a workflow activity we will simply write a for $a!.a?$. This will make the workflow activities easier to separate out from the internal message passing.

The non-determinism inherent in CCS should not be considered a problem; rather it is a convenient and elegant method of abstracting from implementation details, application logic or user input. The expression $\tau.P + \tau.Q$ could be abstraction over a data-dependent choice to be made by the system or the decision of a human being.

It is important to notice that the workflow patterns coded here are abstract patterns that do not make any assumptions about the data flow. This separation of concerns, besides being an important design principle, allows a clear and focused comparison without too much clutter. It also means that one can plug in one's data-flow language of choice at any later point. The analysis made here holds *regardless* of the data-flow language chosen, not just for one specific data-flow language, and so in effect the analysis is stronger and more general because of this abstraction[6].

6.3 The Control Flow Patterns

In this section we consider each of the 20 workflow patterns in turn, discuss their encoding in CCS, and present new patterns. To facilitate comparison the discussion is structured according to the taxonomy given in [39] (see Table 1), and the pattern descriptions are taken verbatim from [38] with minor clarifications in square brackets.

In the following pattern encodings some internal message-passing is often used for synchronization purposes. Such internal messages, like *done*, *ok*, *start*, and *go*, should be concealed to outside expressions through use of the new operator. Except for the first example, we tacitly assume their existence to avoid cluttering

[5] The mechanism does not work if multiple instances of a are invoked simultaneously. We ignore this for the moment.

[6] Afterwards, a similar study could be made with regard to data-flow requirements, and of course any realistic system should handle both control flow and data flow.

Table 1 The twenty workflow patterns [38] and two new ones

Basic Control Patterns

1 Sequence

2 Parallel Split

3 Synchronization

4 Exclusive Choice

5 Simple Merge

Advanced Branching and Synchronization Patterns

6 Multiple Choice

7 Synchronizing Merge

8 Multiple Merge

8a N-out-of-M Merge ***(new)***

9 Discriminator

9a N-out-of-M Join

Patterns Involving Multiple Instances

12 MI without synchronization

13 MI with a priori known design time knowledge

14 MI with a priori known runtime knowledge

15 MI with no a priori runtime knowledge

Structural Patterns

10 Arbitrary Cycles

11 Implicit Termination

Cancellation Patterns

19 Cancel Activity

20 Cancel Case

State-Based Patterns

16 Deferred Choice

16a Deferred Multiple Choice ***(new)***

17 Interleaved Parallel Routing

18 Milestone

the expressions.

Pattern 1. Sequence – *execute activities in sequence.* The first encoding that comes to mind is $a.P$, but unfortunately this does not do the job completely. It is desirable to be able to put two arbitrary processes after each other like $P.Q$. A simple solution is provided by Milner [19]: We require all processes to send on an agreed-upon channel, say *done*!, when they are done and the encoding of the sequence $P.Q$ then becomes

$$\mathsf{new}\ go\ (\{^{go}/_{done}\}P \mid go?.Q)$$

where $\{^{go}/_{done}\}P$ is an alpha-conversion that is handled statically on source code level. Henceforth we will use the simpler notation P_{go} to signify that P signals on channel go on completion and that all pre-existing free *ok*s in P have been alpha converted. If the channel is not relevant, it will be omitted. □

Pattern 2. Parallel Split – *execute activities in parallel.*

$$P_1 \mid \cdots \mid P_n$$

□

Pattern 3. Synchronization – *synchronize two parallel threads of execution.*

$$P_{1,ok} \mid \cdots \mid P_{n,ok} \mid \underbrace{ok?.\cdots.ok?}_{n}.R$$

□

Pattern 4. Exclusive Choice – *choose one execution path from many alternatives.*

$$\tau.P_1 + \cdots + \tau.P_n$$

The τ transition prefix on each branch allows a data-dependent decision or active decision; that is, the system can decide upon a particular branch (and do away with all other branches) without immediately activating the activities of that branch (cf. *(16) Deferred Choice*). In future patterns involving choice the τ prefix will be omitted unless needed. □

Pattern 5. Simple Merge – *merge two alternative execution paths.*

$$(P_{1,ok} + \cdots + P_{n,ok}) \mid ok?{*}.R$$

The pattern assumes (unnecessarily for our purpose) that none of the processes P_i are ever run in parallel. This assumption is expressed here by the use of $+$. □

Pattern 6. Multiple Choice – *choose several execution paths from many alternatives.* A simple encoding would be

$$(\tau.P_1 + \tau.\mathbf{0}) \mid \cdots \mid (\tau.P_n + \tau.\mathbf{0})$$

but this (a) does not enforce a minimum number of activities to be executed and (b) does not explicitly tell if the system is still waiting for a choice to be made or already decided to take the **0** branch. Addressing (b) we get

$$(\tau.P_1 + lazy!) \mid \cdots \mid (\tau.P_n + lazy!)$$

where the system outside should then accept messages on channels *ok* and *lazy* appropriately. Addressing (a) amounts to requiring some number of P_is to be started before the system is able to perform rendez-vous on *lazy*. To avoid cluttering up the expression, we do not do this here. □

Pattern 7. Synchronizing Merge – *merge many execution paths. Synchronize if many paths are taken. Simple merge if only one execution path is taken.*

$$(P_{1,ok} + ok!) \mid \cdots \mid (P_{n,ok} + ok!) \mid \underbrace{ok?.\cdots.ok?}_{n}.Q$$

All P_i perform an *ok*! when they are done so the synchronization mechanism guarding Q should wait for exactly n such messages. □

Pattern 8. Multiple Merge – *merge many execution paths without synchronizing.*

$$(P_{1,ok} + lazy!) \mid \cdots \mid (P_{n,ok} + lazy!) \mid ok?{*}.Q \mid lazy?{*}$$

Signals *ok*! and *lazy*! signify if an activity was executed or not. Notice that other processes may connect to the merge by using the named entry-point *ok*. If this in undesired in the context, a new *ok* can be added around the expression. Additionally, any occurrences of *ok* in Q should be removed by alpha conversion. □

Pattern 9. Discriminator – *merge many execution paths without synchronizing. Execute the subsequent activity only once.*

$$(P_{1,ok} + lazy!) \mid \cdots \mid (P_{n,ok} + lazy!) \mid ok?.(Q \mid lazy?{*} \mid ok?{*})$$

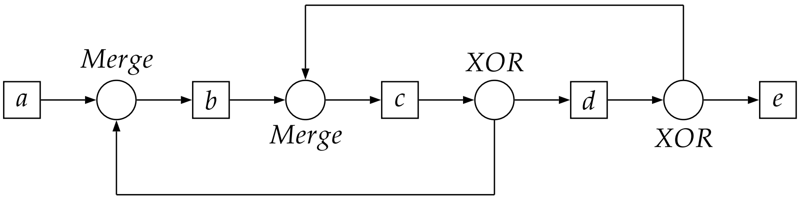
Here the *Multiple Choice* pattern is used but other split patterns could be used too. The first signal on *ok* initiates Q and all remaining signal are consumed. Contrary to Petri nets, ignoring all future signals is not a problem because any loop around this construct would simply instantiate a new process (with new channel names) in each iteration. □

Pattern 9a. N-out-of-M Join – *merge many execution paths. Perform partial synchronization and execute subsequent activity only once.*

$$(P_{1,ok} + lazy!) \mid \cdots \mid (P_{n,ok} + lazy!) \mid \underbrace{ok?.\ldots.ok?}_{n}.(Q \mid lazy?{*} \mid ok?{*})$$

For the purpose of this demonstration, the pattern is combined with *Multiple Choice* pattern. Once n messages are received on *ok*, the process Q is activated and all remaining messages, whether *ok* or *lazy*, are discarded. □

Pattern 10. Arbitrary Cycles – *execute workflow graph w/out any structural restriction on loops.* Consider the following example from [38]:



Since activities b and c have multiple merge points that do not originate from the same split block, it is necessary to promote them to named services (or "functions") by using the replication operator—a lot like what one would do in any structured programming language. Here we name them go_b and go_c and the example becomes

$$a.go_b! \mid go_b?{*}.b.go_c! \mid (go_c?{*}.c.(go_b! + d.(go_c! + e))).$$

□

Pattern 11. Implicit Termination – *terminate if there is nothing to be done.*

Implicit termination means detecting if all processes have come to a stable state and there are no pending activities or messages. In other words if the residual workflow process—disregarding subexpressions of the form $a?{*}.P$—is structurally congruent to $\mathbf{0}$. Subexpression of the form $a?{*}.P$ can be thought of as functions; they cannot and should not be reduced. Any active invocations of such a function will manifest themselves as some reduced form of the body P. (However, if a function listens on a channel that cannot be reached by the active process expression, it may be garbage collected by the runtime system, but this is separate from detecting implicit termination.)

Explicit termination was modeled in the *Sequence* pattern where we adopted the convention that all processes signal on a pre-determined channel on completion. When the top-level expression wishes to send its signal, the process explicitly terminates. □

Pattern 12. MI without synchronization – *generate many instances of one activity without synchronizing them afterwards.*

$$P_{loop} \mid loop?{*}.(create!.loop! + R) \mid create?{*}.Q$$

After process P finishes an arbitrary number of instances of Q are spawned and once no more Qs need to be started, R is executed. Execution of R does not wait for the completion of any of the instances of Q. The choice inside the loop can be an *Exclusive Choice* or a *Deferred Choice.* □

Pattern 13. MI with a priori known design time knowledge – *generate many instances of one activity when the number of instances is known at the design time (with synchronization).*

$$\underbrace{create!.\cdots.create!}_{n}.\underbrace{ok?.\cdots.ok?}_{n}.Q \mid create?{*}.P_{ok}$$

This pattern creates exactly n instances of P and waits for all of them to complete before passing control on to the process Q. □

Pattern 14. MI with a priori known runtime knowledge – *generate many instances of one activity when a number of instances can be determined at some point during the runtime (as in FOR loop but in parallel).* First a counter is needed to keep track of the number of instances that need to be synchronized. It can be coded (a) directly in CCS through cunning use of the `new` operator (cf. sect. 7.5 of [19]), (b) by adding a notion of simple datatypes such as integers to CCS or (c) by using channel-passing π-calculus style. For now we will just add a process *Counter* and not bother with its implementation details; the important part being that is it feasible in CCS. *Counter* has the ability to receive messages on channel *inc* (increase) and to send message on channels *dec* (decrease) or *zero* (check if zero) depending on its state:

$$\begin{aligned} P_{loop} &\mid loop?{*}.(create!.loop! + break!) \mid create?{*}.inc!.Q_{ok} \\ &\mid break?{*}.(ok?.dec?.break! + zero?.R) \mid Counter \end{aligned}$$

From P the process enters a loop where multiple instances of Q are created. Once creation is done, the process iterates the *break* loop until a message on *zero* is received, i.e. 0 instances remain. Notice that one can decide not to create any instances at all. If one desires a minimum number of processes to be spawned, the *break*! prefix should be guarded by unfolding the *create* loop an appropriate number of times. □

Pattern 15. MI with no a priori runtime knowledge – *generate many instances of one activity when a number of instances cannot be determined (as in*

WHILE loop but in parallel). This pattern is merely a simplified version of pattern 14. There is no longer a need to distinguish between the creation phase and the synchronization phase. Hence it collapses to:

$$\begin{aligned} P_{loop} \;&|\; loop?*.(create!.loop! + ok?.dec?.loop! + zero?.R) \mid create?*.inc!.Q_{ok} \\ &|\; Counter \end{aligned}$$

More advanced synchronization schemes can be plugged in at this place. Maybe only som n instances need to finish, maybe the completion condition is some predicate ρ over the data produced so far. In other words, the logic deciding this loop can be pushed up to a data-aware layer or handled through more complex join conditions in the process expression. □

Pattern 16. Deferred Choice – *execute one of the two alternatives threads. The choice which thread is to be executed should be implicit.*

$$P_1 + \cdots + P_n \quad \text{where each subprocess is guarded.}$$

This is very similar to the *Exclusive Choice*. Here the choice is made exactly when a (non-silent) transition of either of the P_is occurs, and hence we require all P_i to be guarded. In the *Exclusive Choice* we might have $\tau.P + \tau.Q$ to signify that an abstract upper-layer would decide which branch to follow (i.e. what τ transition to take). □

Pattern 17. Interleaved Parallel Routing – *execute two activities in random order, but not in parallel.*

$$\begin{aligned} &\mathsf{new}\ lock, unlock, locked, unlocked \\ &\ (lock!.P_1.unlock! \mid \cdots \mid lock!.P_n.unlock! \\ &\ \mid (unlocked?*.lock?.locked! \mid locked?*.unlock?.unlocked! \mid unlocked!)) \end{aligned}$$

This is generalized slightly from two to any finite number of interleavings. Each process P_i acquires the lock on activation and releases it upon termination. □

Pattern 18. Milestone – *enable an activity until a milestone is reached.* Assume that process Q can only be enabled after P has finished and only before R is started. This is done by the help of a small flag that can be changed using *set*! and *clear*!

and tested using $ison?$ and $isoff?$:

$$\begin{aligned}&Milestone(ison, isoff, set, clear) =\\&\quad on?{*}.(ison!.off! + set?.on! + clear?.off!) \mid\\&\quad off?{*}.(isoff!.off! + set?.on! + clear?.off!) \mid off!\end{aligned}$$

Now the milestone can be set up as

$$P.set!.{}_{clear}R \mid ison?{*}.Q \mid Milestone$$

where ${}_{clear}R$ denotes the modification of process R that signals on $clear$ when it starts its first activity or makes its first explicit choice. □

Pattern 19. Cancel Activity – *cancel (disable) an enabled activity.* Suppose there are activities a, b, and c being carried out in sequential order. Anytime during the execution of activity b the process can be cancelled. We break up activity b into $b!.b?$ so it will be clear exactly when a cancellation can be accepted. Assume cancellation is obtained by signaling on channel $cancel$:

$$a.b!.(b?.c + cancel?)$$

□

Pattern 20. Cancel Case – *cancel (disable) the process.* The example of this pattern given at [38] is an online travel reservation system. If the reservation for a plane ticket fails, then all pending reservations for flights, hotel, car rental, etc. in the itinerary must be cancelled immediately to avoid unnecessary work.

Intuitively, this corresponds to returning to some predefined state in the system after relinquishing all resources and information bound in the current context. In this respect the cancellation patterns look at lot like a program exceptions, and indeed the cancellation pattern can be coded in CCS by using a source code transformation.

Pattern 19 showed how to insert a cancellation point a one particular spot in a process. We now insert cancellation points for all states in the process $a.b.c$ to obtain

$$\begin{aligned}&a!.(a?.(b!.(b?.(c!.(c? + cancel?) + cancel?) + cancel?)\\&\qquad\qquad + cancel?) + cancel?) + cancel?\end{aligned}$$

□

Clearly this becomes very tedious and strongly suggests that a more abstract language would be useful.

6.3.1 More Split and Join Patterns

A central part of any workflow formalism is that of splitting the control flow into several possible paths and later joining control flow from different places into fewer paths. This is addressed by the *Basic Control Patterns*, the *Advanced Branching and Synchronization Patterns*, and to a certain extent the *Patterns Involving Multiple Instances*, the *Deferred Choice* pattern, and the *Interleaved Parallel Routing* pattern.

A structured view of these patterns proves beneficial. Consider Table 2 for an overview of Split, Synchronize, and Merge patterns. Split patterns split one path into some m paths. The difference between the split patterns is whether they require control flow to enter only *one* path (choice), *some* paths (multiple choice) or *all* paths (parallel split).

The same trichotomy is useful when considering synchronization and merging. Synchronization waits for a number of signals (one, some or all) and then launches one continuation, whereas merge spawns a continuation for each received signal (at most one, at most some number or for all).

Two new patterns emerge from this exercise: *(16a) Deferred Multiple Choice* and *(8a) N-out-of-M Merge.* These are described and coded below.

More generally, one could give every join construct a predicate that dynamically decided when to continue. If equipped with simple constructs for writing such predicates, the language becomes much more expressive than the patterns here, but for analysis purposes it is very important that the predicates remain within the boundaries of Presburger arithmetic or some other (preferably simpler) first-order theory in which all statements are decidable. This idea is examined in detail below.

New Pattern 8a. N-out-of-M Merge – *merge many execution paths without synchronizing, but only execute subsequent activity the first n times. Remaining*

Table 2 Split, Synchronize, and Merge Patterns. $l \to n/m$ means "control flows from l path(s) to n out of m possible paths". $n/m \to l$ means "control flows from n of m possible paths into l.

	Split	**Synchronize**	**Merge**
All	Parallel Split $1 \to m/m$	Synchronization $m/m \to 1$	Multiple Merge $m/m \to m$
One	Exclusive Choice $1 \to 1/m$	-	Simple Merge $1/m \to 1$
	Deferred Choice $1 \to 1/m$	Discriminator $1/m \to 1$	-
Some	Multiple Choice $1 \to n/m$	Synch. Merge $n/m \to 1$	Multiple Merge $n/m \to n$
	Deferred Multiple Choice $1 \to n/m$	N-out-of-M Join $n/m \to 1$	N-out-of-M Merge $n/m \to n$

incoming branches are ignored.

$$(P_{1,ok} + lazy!) \mid \cdots \mid (P_{n,ok} + lazy!) \mid \underbrace{ok?.go! \mid \cdots \mid ok?.go!}_{n} \mid go?{*}.Q \mid lazy?{*}$$

The pattern is similar to *Multiple Merge* except that it is only capable of receiving n messages on *ok* and then it is done. □

New Pattern 16a. Deferred Multiple Choice – *execute several of the many alternative threads. The choice of which threads are to be executed should be implicit.* The important question here is: how do we know when the users are done choosing the threads? Two approaches are possible: (a) we fix an n number of threads to be activated or (b) we set up a virtual activity that means "all desired threads have been started, remove remaining choices". Option (a) requires all subprocesses to emit a signal on activation and consume n such signals before activating a $lazy?{*}$ process to remove the remaining branches. Here we follow option (b):

$$(P_1 + lazy!) \mid \cdots \mid (P_n + lazy!) \mid donechoosing.lazy?{*}$$

□

Ideally, the control flow patterns should have a formal representation. Such a representation should ensure that the patterns have a clear semantics, are atomic, have a minimum encoding bias (i.e. with respect to any particular concurrency model), have minimum overlap, and are as simple as possible. The current description format was clearly devised with a different set of goals in mind. Several overlapping patterns were added for easy of benchmarking. E.g. the presence of four multiple instance patterns instead of just one allows a finer benchmarking of systems that do not support the most general pattern. Also, the patterns were developed in response to the previous very limited research asserting that split, join, loop, and sequence were sufficiently expressive for all workflows. Thus, an (industry) accessible description format was needed.

Formal descriptions are important, however, as they facilitate further research: In the current state of affairs no language can reasonably make the claim of supporting all patterns. Although we can make this statement plausible by meticulously encoding every pattern in isolation as we have done, there is no theorem showing that every conceivable combination of the patterns can also be expressed. Formal descriptions also open the doors to verification, simulation, rigorous comparison, compositional analysis, and algebraic reasoning. First and foremost, a formal description of the patterns is an important step towards designing a programming language for workflows. Currently, the only formal description is the one implicit in the implementation of the YAWL system [37] but this does not satisfy the minimum coding bias criterion—neither do Petri net-inspired graphical illustrations. Before considering a formal representation a brief analysis of the patterns is in place.

In their current state the patterns do not live up to the criteria sketched above. A closer look at the patterns, reveals that they straddle graphical and algebraic approaches, and that attaching a semantics is non-trivial. Some patterns do not seem atomic. The "Multiple instances without a priori runtime knowledge" pattern contains both iteration, splitting, and merging in one pattern. On the other hand the "Synchronizing Merge" pattern has no well-defined semantics in isolation simply because it is not known how many completion signals it should wait for. Non-trivial overlap also exists: "Multiple Merge" could be expressed as "Sequence" without explicit synchronization, and if instead "Sequence" is taken to imply tacit synchronization then it would seem that a pattern for explicit synchronization was no longer needed.

Let us return to the idea of workflows as initially completely unstructured. Tasks

$a_1, \ldots, a_n$need to be performed, and every workflow pattern is a different way of imposing constraints (w.r.t. ordering, data-dependency, time, etc.) on the task list. The question is how to express these constraints. Curbera et al. [7] identify two major directions: the graph-oriented approaches and the algebraic approaches. Examples of the former include Petri net-based models (most significantly YAWL [37]), Event-Driven Process Chains (EPCs) [31], and dependency graph models (e.g. GNU `make` [7]). Examples of the latter include—most prominently—models based on π-Calculus and Join Calculus. According to Curbera et al. BPEL [36]—arguably today's most widespread standard—is a hybrid approach, which combines ideas from WSFL (graph-based, loosely based on Petri nets) and XLang (algebraic, loosely based on π-Calculus). This distinction may reveal the starting point of the language designers and their preferred type of formal reasoning, but what is more interesting are the restrictions on the language relative to the patterns. As we have seen, all the control flow patterns can be encoded in a process calculus as they are. Curbera et al., however, seem to suggest that the algebraic approach enforces a strictly nested syntactical regime, e.g. where all splits and join match one-to-one by design of the constructs. The algebraic approach does enforce a syntax tree, naturally, but this does not imply any of the constraints suggested on the control flow graph unless chosen to by design. A more rigorous taxonomy of workflow models is needed.

Let us frame this discussion by considering some challenging examples. All these examples can be given a strict semantics if disambiguating hints are provided by the workflow designer, but not all models will be able to express them (i.e. without heavy reliance on some mechanism for shared state). The goal, however, is a minimal burden on the workflow designer (hopefully yielding the corollary "minimal opportunity to introduce bugs" [8]).

6.4.1 Non-Matched Splits and Joins (No Block Structure)

As mentioned previously a major difference between workflows and the control flows found in conventional programs is the widespread lack of block structure. Consider the Petri net in Figure 1. The Petri net exhibits a pattern that is cumbersome to express in workflow languages that only have block constructs. The core problem is the pattern shown in Figure 2. On the other hand, graphical models such as Petri nets or even GNU `make` have no problems expressing this. The example was provided as a challenge to the π-calculus community during a 2004 surge in interest

[7] `http://www.gnu.org`

[8] In an abuse of Alfred North Whitehead's famous quote we could say that the ultimate goal of computer science is to eliminate to need for intelligent thought.

Fig. 1 Van der Aalst's *Challenge 4* [43]

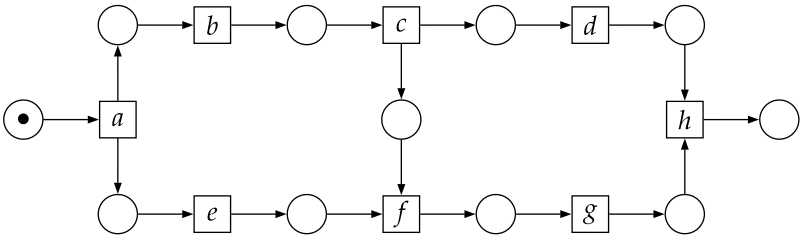
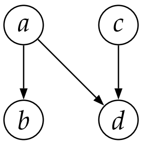


Fig. 2 Non-nested control flow



for process algebraic approaches to workflow modeling. XLang, for instance, claims to be based on the π-calculus and indeed has constructs for sequence, choice, parallel flow, and iteration (`sequence`, `switch`/`pick`, `all`, and `while,` respectively). It adopts an implicit synchronization scheme for the sequence operator similar to the idea developed in our encoding of the *Sequence* pattern. XLang lacks internal message primitives, however, and this makes the link between c and f impossible to express[9]. The conclusion is: a language with block structure only, will not be sufficient. There are several remedies available, of which internal message-passing immediately springs to mind. First, though, let us consider another idea. We previously [35] conjectured the usefulness of an *overlay operator* inspired by the parallel operator || found in Communicating Sequential Processes (CSP) [13]. The overlay operator & is defined by the rules in Figure 3. The major difference between & and || is the "no mention" rule (the first rule), which allows P to transition on α if Q does not mention α. Thus the || operator can be obtained by removing the first rule. The reduction semantics may seem convoluted, but it is worth noticing the neat trace semantics: $traces(P\&Q) = \{t : t_{\alpha P} \in traces(P) \wedge t_{\alpha Q} \in traces(Q)\}$, where $t_{\alpha P}$ denotes the projection of the trace t onto the ports of P[10]. The operator is perhaps best explained by coding the example in Figure 1:

$$C4 \equiv a.(b.c.d|e.f.g).h \ \& \ (c|e).f$$

[9] The link between c and f could of course be simulated through use of shared data. The statement of impossibility made here holds only in the absence of controlled access to shared data. A formal (Petri net-based) proof of such an expressiveness result can be found in Kiepuszewski's Ph.D. dissertation, theorem 5.3.2, pp. 139-141 [16].

[10] The trace semantics of || is given as $traces(P||Q) = traces(P) \cap traces(Q)$.

Whenever $C4$ wants to transition on a channel that is mentioned in both operands to the & operator, both operands have to agree to this. If a transition is only mentioned in one of the operands, the & operator does not impose constraints on that transition. Here if a, e was performed first, the left process would be ready to do f but the right would not, since it would need c to complete first.

The operator is interesting, not only as a remedy for block-structured languages, but also because it allows the overlaying of global business rules to all processes. E.g. one might have a rule that says "delivery shall occur before invoicing". This rule would now simply be enforced on all workflows by overlaying them with the rule workflow. Also, the workflow designers do not have to describe all business rules repeatedly because the can simply overlay the global rules to their workflows. Of course, this requires a strict nomenclature of tasks in order to work, and possibly something more flexible than simple alphabetic matching would be needed.

BPEL has a different solution for the problem. As mentioned BPEL combines the block constructs of XLang with the free graph ideas of WSFL. In addition to block constructs, BPEL allows dependencies in the form of named *links* between nodes, each of which should acknowledge participation in the link by including a `source` or `target` tag containing the link identifier. In BPEL, challenge 4 would most likely be solved by creating the link from c to f using its a `source` tag in c and a `target` tag in f. Links are limited, however, as they may not extrude or intrude while loops, they may not introduce cycles, and they are governed by several other syntactical restrictions. BPEL therefore does not offer first-class support for arbitrary cycles.

Compared to BPEL's notions of links through `source` and `target`, the & operator offers separation of concerns: the source and target nodes do not explicitly express the link because it is factored out. Whether non-local logical constraints in the style of the & operator is a good or a bad idea, remains an open question pending practical studies.

It should be stressed, that the operator remains speculative at this point and has not been developed formally in a CCS setting. The operator can be built into the reduction rules of a workflow system, or it can be translated down to regular CCS. It is not important whether the introduction of & adds expressiveness[11] to CCS itself, but an &-like construct can strictly improve expressiveness for more limited high-level workflow languages.

[11] In Felleisen's definition of expressiveness [9].

Fig. 3 Reduction Rules for the Overlaying Operator &

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \notin \text{ports}(Q)}{P\&Q \xrightarrow{\alpha} P'\&Q}(+\text{sym.}) \qquad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\alpha} Q'}{P\&Q \xrightarrow{\alpha} P'\&Q'}$$

6.4.2 Multiple Merge and Multiple Instances

Another dividing concern is that of multiple instances. There are two sources of multiple instances: (1) using the *Multiple Merge* pattern to join paths from a previous split pattern and (2) through the multiple instance patterns 12-15. Kiepuszewski argues that using *Multiple Merge* is a convenient and natural way of expressing multiple instances [16]. Whereas this may hold for graphical languages, it is not necessarily desirable for algebraic languages. For instance it is far from obvious that an idiom such as

$$merge(parsplit(A, B, C, D), E)$$

is really needed. Disallowing *Multiple Merge* and instead appending a function call to each branch would not seem like an unreasonable burden, since there is a fixed and typically low number of branches.

In case (2) where we explicitly spawn multiple instances, there is clear need. First, because expressions such as $split(P, P, P, P)$ are tedious even if P only consists of a function call; second, because there could easily be a high number of Ps; and third, because the final number of branches is not known design-time or even runtime (cf. pattern 15: *Multiple Instances With No A Priori Runtime Knowledge*). With the exception of the runtime generativity behavior (i.e. patterns 14 and 15), the regular split/join patterns would seem able to capture multiple instance patterns with some few alterations. We will explore this in more depth in section 6.4.4.

BPEL supports the multiple instance patterns only through scant workarounds [1], and BPEL also does not support *Multiple Merge.*

On a different note, there is an overlap between *Sequence*, *Multiple Merge* and synchronization. Consider the expressions

$$\begin{aligned} merge(parsplit(A, B, C, D), E) \\ sequence(parsplit(A, B, C, D), E) \\ synch(parsplit(A, B, C, D), E) \end{aligned}$$

There are two interpretations: (1) All four threads A, B, C, D should spawn their

Fig. 4 The Travel Agency: *Synchronizing Merge* in action

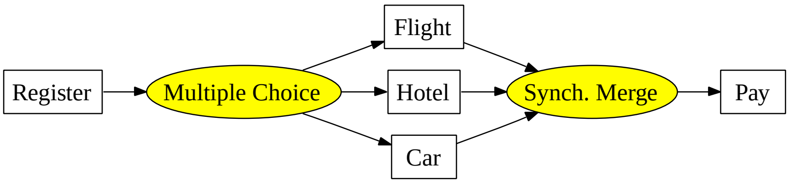
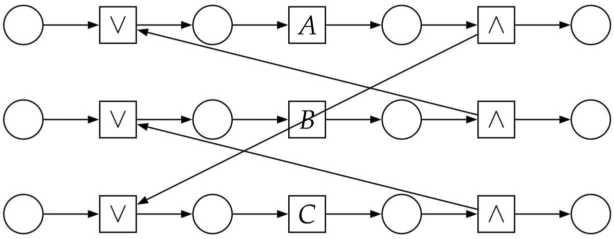


Fig. 5 OR-join nightmare scenario (example from Figure 3, p. 12 [17])



own copy of E or (2) when all threads A, B, C, D are done, one instance of E should run. However, these two interpretation are split across three control flow patterns, and hence one is unnecessary. Either we assume that *sequence* in expression 1 implicitly joins the four threads (interpretation 2). In this case the *synch* pattern is unneeded. Alternatively, we assume that *sequence* in expression 1 does not join the four threads (interpretation 1). In this case the *merge* pattern is unnecessary. The question is this: should synchronization be explicit or should merge be explicit? Merge is by far the most dangerous construct to forget, thus strongly suggesting that merge should not happen implicitly. The existence of such overlaps underlines the need for a formal representation of the patterns.

6.4.3 Synchronizing Merge

As it turns out, a lot of the problems with the patterns can be framed neatly by looking at the "Synchronizing Merge" pattern.

Synchronizing Merge is described as "merge many execution paths. Synchronize if many paths are taken. Simple merge if only one execution path is taken." Before continuing consider Figure 4 (The Travel Agency) for a motivating example[12]. Synchronizing Merge occurs far too often to be ignored, and indeed there is nothing contrived about the the Travel Agency example. At runtime a user (data-dependent,

[12] This example was initially suggested by Wil M.P. van der Aalst.

explicit) choice is made as to whether flight, hotel, and/or car reservations are needed. Once the reservations go through, the task *Pay* is activated, and, of course, the join should only wait for the branches that were actually activated.

The problem is that Synchronizing Merge has non-local semantics; it cannot be given a semantics without either

(1) integrating it with the preceding split patterns
(2) leaving the problem to the runtime system or
(3) establishing a method of propagating information about the paths actually taken in prior splits to the Synchronizing Merge.

This well-known problem appears in the context of high-level Petri nets [39], Event-Driven Process Chains [44,17,31,27], and BPEL [7]. The problem is more complicated in the presence of loops because the semantics of a Synchronizing Merge can be dependent on a split further down the line. The problem can be alleviated in a variety of ways. Runtime checks, syntactic constraints, message-passing etc. may work, but as we have seen, even a seemingly unobtrusive idea such as imposing a block structure has a serious impact on the expressiveness.

Event-Driven Process Chains [31] are probably they least constrained model: EPCs define the concepts of *functions* (called tasks is this report), *information objects* (data), and *events* (events or signals) in a graphical language where functions and events constitute the nodes. Functions and events can be connected using arcs; an arc from an event to a function stipulates that the event be triggered before the function is executed; an arc from a function to an event means the event is fired upon completion of the function. In addition to the function and event nodes there are logical connectives $\wedge$, $\vee$, and XOR. With only a few restrictions, arcs can connect events to connectives, connectives to events, functions to connectives, and connectives to functions.

EPCs form the basis of SAP R/3's workflow modules and the ARIS (Architektur integrierter Informationssysteme) framework. EPCs have great modeling power, thanks to the unconstrained graphical form (allowing arbitrary cycles) and the $\vee$-connective, which expresses the idea of the Synchronizing Merge pattern. EPCs were initially proposed in an informal fashion, and a long discussion about the precise semantics of the $\vee$-connective ensued.

YAWL relies on the runtime system for the semantics of Synchronizing Merge: The OR-join transition is allowed "if the number of consumed tokens cannot be increased by postponing the occurrence of the OR-join" [45]. The formal definition can be found in the same report p. 30. YAWL is designed to run on a centralized server that stores the entire workflow state and therefore can make judgments about the OR-join. In a distributed setting, the workflow resides on different servers, and deciding whether an OR-join is allowed or not may induce too much overhead – the non-local semantics again becomes a problem. Consider Figure 5. Assume that each horizontal thread of execution resides on a different server, and observe the cyclic dependency. One semantics could be to solicit all three OR-joins for execution. Once one join is taken, the others should no longer be possible, because there is a way of eventually feeding both incoming arcs on them. The creates a global consensus problem, that is unacceptable in distributed settings unless kept at a strictly minimal level of occurrence.

BPEL also faced the Synchronizing Merge problem, but adopted another approach. In BPEL information about non-chosen paths is propagated by *dead path elimination* (DPE). A join node initially has the value *unevaluated* assigned to each incoming link. If at runtime it becomes clear that the link cannot be activated, the value *false* is assigned to it. This is done by propagating *false* down through a branch once it is clear that another alternative was chosen[13]. If a link can be activated, eventually it will be or the link will become tagged as *false.* The join condition is satisfied once all incoming links are evaluated to either *true* or *false.* Essentially, this means that in BPEL Synchronizing Merge has been replaced by a runtime facility. A signal is always received, and its boolean value decides the semantics. Unfortunately, BPEL imposes heavy restrictions on the propagation of (internal) signals and therefore it cannot be said to be as general as YAWL.

The *Synchronizing Merge* pattern is difficult to avoid without serious setbacks in expressiveness. At the same time, any system allowing it must employ some variation of the three solutions suggested (add structure to workflow, propagate information or decide runtime). As we have already argued, requiring block structure—even locally—is unacceptably restrictive. Propagating information only works in limited settings, and breaks down when cycles are present. This is clearly seen by the number of constraints on links in BPEL. A runtime rule such as the one defined for YAWL seems to be the proper solution for single-server systems, but perhaps a less restrictive propagation semantics could be designed for future versions of BPEL. The last word about *Synchronizing Merge* is yet to be said.

[13] Provided the flag `suppressJoinFailure` is set – otherwise an exception is raised explaining that the process has become stuck.

6.4.4 Generalizing Split and Join Patterns

Synchronizing Merge is the *only* one of the split and join patterns that does not have a strictly local semantics. This observation inspires two ideas with regard to the remaining split and join patterns:

(1) Split and join patterns could be expressed more generally with a grammar for what we could call "split and join conditions".
(2) Multiple Instance patterns could be split up and merged into the general way of expressing split and join conditions.

It should be noted though, that some of the advanced Multiple Instance patterns (*with a priori runtime knowledge* and *without a priori runtime knowledge*) in some cases require a *Synchronizing Merge.*

The split patterns (*Exclusive Choice*, *Deferred Choice, Parallel Split*, and *Multiple Choice*) can be covered in a number of ways. E.g. it would be enough to simply have a generic

$$\mathit{split}(\mathit{min}, \mathit{max}, [P_1, \ldots, P_n])$$

where *min* and *max* denoted the minimum and maximum number of branches that needed to be started. Thus *Choice* would have $(\mathit{min}, \mathit{max}) = (1, 1)$, *Parallel Split* would have $(\mathit{min}, \mathit{max}) = (n, n)$, and *Multiple Choice* would have $(\mathit{min}, \mathit{max}) = (1, n)$. The distinction between explicit and implicit (deferred) choice would be in guarding the P_i with predicates. This pattern is far more general that any of the patterns it tries to encompass. First it allows mixing guarded and unguarded processes, i.e. mixing explicit and implicit choice. Second, $(\mathit{min}, \mathit{max})$ can take on different values to signify limits branching limits. So far we have tacitly assumed that every P_i would be run at most once, but this does not have to be the case. An even more generic form could take this into account with a minimum and maximum for each branch:

$$\mathit{split}(\mathit{min}, \mathit{max}, [(P_1, \mathit{min}_1, \mathit{max}_1), \ldots, (P_n, \mathit{min}_n, \mathit{max}_n)])$$

Thus Multiple Instances can be expressed as $\mathit{split}(1, \infty, [(P, 1, \infty)])$, and a host of other variations are possible (along with a wide range of new sources of errors). Once again this covers significantly more territory than the patterns it was intended to model.

When is the split completed? If $\mathit{min} \neq \mathit{max}$, the semantics need further specification. Even more so if some P_i are predicate guarded and some are not. Every P_i can invoke another P_j during its execution, but at some point the creation of threads should be considered completed, even if the maximum has not been reached. In other words, if a split has spawned between *min* and *max* branches, it should solicit tasks for execution, until there is no longer any need. This need is decided by the synchronization side. Let us therefore consider the synchronization patterns. Here we can equivalently conjure up a syntax such as

$$synch(waitfor, P)$$

to denote that all active branches of P should be synchronized, but control should be passed on, once $waitfor$ branches have completed. Thus assuming P has n branches, we can write

$$synch(n, P)\ (\textit{Synchronization}),\ synch(1, P)\ (\textit{Discriminator}),\ \text{or}\ synch(m, P),$$

where $1 < m < n$ (*M-out-of-N Join*). Following the observation in section 6.4.2, it is probably safer to require explicit merge rather than explicit synchronization. Hence *synch* should become an implicit part of *split* and in addition there should be a way of doing a merge.

What was just presented is a highly speculative design, and only one of many possible designs. The *min* and *max* designations are merely a simplified predicate language describing the allowable splits from a universe of options. An equally workable solution, therefore, is to simply introduce a minimal predicate language, very much like connective nodes of EPCs or the join conditions of BPEL. This especially makes sense for joins, where we need to answer two questions: (1) when can control be passed on, and (2) when should all remaining processes be canceled and/or garbage collected. For splits we usually need to answer the question: when it the split done spawning branches. As long as branches can be spawned, the join side cannot do a *Synchronizing Merge.*

Where do we stop? This should not degenerate into a race for more patterns, in turn demanding more expressiveness of an already overburdened set of constructs. The language must be kept simple, and already at this point, the convenience of a simple reduction semantics seems far away. Conversely, the language needs to present a few comprehensible constructs rather than an array of partially overlapping patterns. Thus, the hunt for a simple language is certainly justified. Pinpointing the exact design criteria for a workflow language is future work. The existing patterns are justified by observation in real-life scenarios, and, as we have seen, there are many ways of designing languages to express them all. What is not acceptable is the current state of the patterns, where part of the generativity is camouflaged in the *Multiple Merge* pattern and some is packaged in the protective padding of the *Multiple Instance* patterns. YAWL for instance claims to support all patterns, but YAWL would not be able to handle a *Multiple Instance* pattern with a *Discrimi-*

nator join, since all the *Multiple Instance* patterns described currently, use either Multiple Merge or Synchronization to join. Therefore the line of research to clean up the patterns and furnish them with a formal semantics should continue. This effort should be two-pronged, simplifying high-level descriptions on one side while developing a suitable low-level calculus on the other side.

6.4.5 Other Patterns

The *Arbitrary Loop* pattern requires a non-block structure as well as repetition. We can solve this using BPEL-style links. Alternatively, it would be enough to allow functions (essentially merge points), synchronization, and forks without synchronization. The idea of functions or sub-workflows seems oddly missing from the patterns. The merge patterns buy us a lot in terms of code reuse, but it does make the patterns seem heavily slanted towards graphical languages (read: Petri nets).

Explicit state in workflow systems is essential to differentiate between deferred (implicit) and active (explicit) choice. Petri nets represent state as places, and the π-calculus represents state as process expressions.

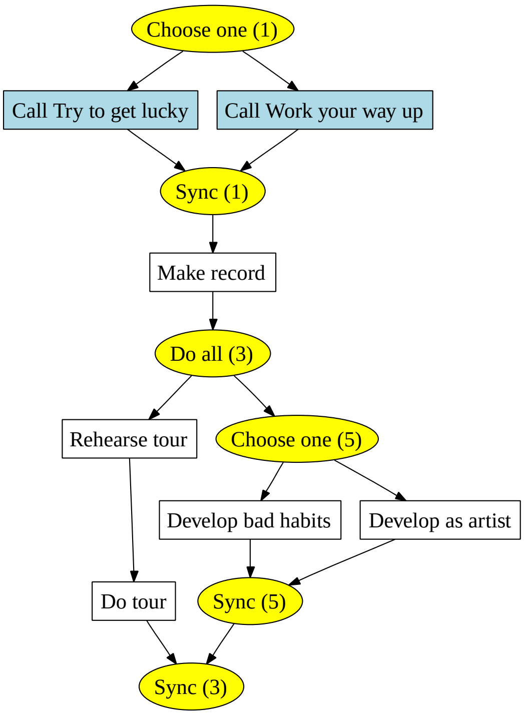
The cancellation patterns are the biggest joker. YAWL has an operator that clears all tokens from a subnet, BPEL has exceptions, LOTOS [4] has the disrupt operator, and none of these have made a convincing argument, that they will work in a distributed setting. Neither Petri nets nor π-calculus handle cancellation patterns very nicely. Cancellation patterns are ubiquitous. Not just because of failure in distributed workflow, but because the *Discriminator* and the *n/m-Merge/Synchronization* patterns conceivably invoke cancellation methods on the remaining threads to ensure graceful termination.

6.4.6 Towards a Workflow Description Language Based on CCS

The workflow patterns coded in the previous sections show that CCS is too low-level to be used directly as a workflow language, and that certain patterns are very cumbersome to write. The language needs high-level constructs and a more pleasing syntax. The goals should be (a) to reduce the amount of user-specified internal synchronization mechanisms and (b) to provide elegant constructs for the 20 (+2) workflow patterns. The language SMAWL, presented here and in previous work [34,35], seeks to do exactly this while maintaining a strong link to CCS. The example in Figure 6 based on [38] shows what a workflow specification might look

Fig. 6 How to become a recording star (adapted from the *Recording Star* example *[38])*

```
workflow Become a recording star =
    chooseone {
        ⇒ call (Work your way up)
        ⇒ call (Try to get lucky)
    };
    Make record;
    doall {
        ⇒ chooseone {
            ⇒ Develop as an artist
            ⇒ Develop bad habits
            }
        ⇒ Rehearse tour;
            Do tour
    }
end
```

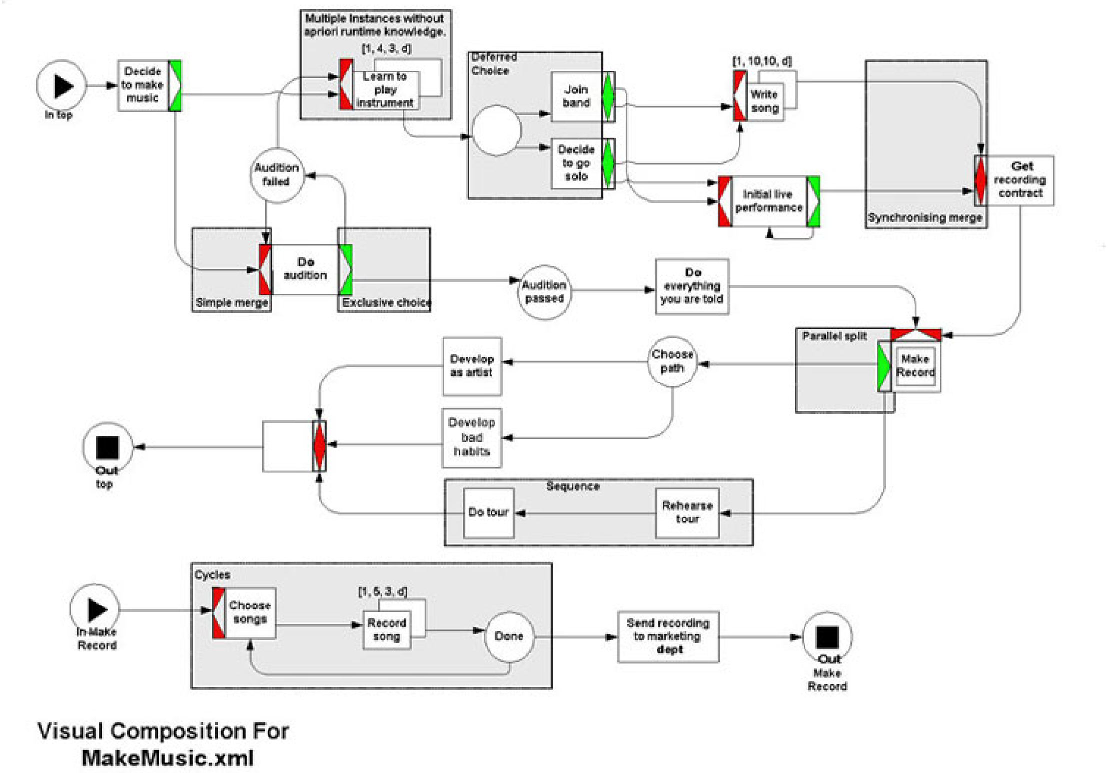


like in SMAWL along with an automatically generated[14] graphical representation.

Having completed the analysis above, a natural idea is to design a language out of small building blocks of based on some of the patterns. This way there would be, e.g., a number of split combinators and a number of join combinators. However, as mentioned in section 6.4.2 explicit synchronization every time a split block is left is tedious and possibly dangerous; a more pleasing style is therefore to implicitly synchronize after every split construct and have the programmer explicitly write if a continuation should be spawned for each active thread (a merge). These and numerous other considerations lead to the following syntax:

[14] Using a Standard ML program to map the abstract syntax tree to a `dot` file, and running `dot`.

Fig. 7 Recording Star example in YAWL (taken from [38])



$$
\begin{array}{rcl}
W & ::= & \textbf{workflow}\ w = P\ \textbf{end} \\
D & ::= & \textbf{fun}\ f = P\ \textbf{end}\ D \;\big|\; \textbf{newlock}\ (l, u)\ D \\
 & \big| & \textbf{milestone}(ison, isoff, set, clear)\ D \;\big|\; \epsilon \\
P & ::= & activity \;\big|\; \textbf{send}(f) \;\big|\; \textbf{receive}(f) \;\big|\; \textbf{call}(f) \;\big|\; P;\, P \;\big|\; \textbf{lock}(l, u)\{P\} \\
 & \big| & \textbf{choose any (wait for k)}\{PP\ \textbf{merge}(n)\ P\} \;\big|\; \textbf{choose one}\{PP\} \\
 & \big| & \textbf{cancel}\ \{P\} \;\big|\; \textbf{do all (wait for}\ \mathrm{k})\{PP\ \textbf{merge}(n)\ P\} \;\big|\; \textbf{multi}(n)\{P\} \\
PP & ::= & \Rightarrow \rho\ P\ PP \;\big|\; P\ PP \;\big|\; \Rightarrow P
\end{array}
$$

In the syntax ϵ denotes the empty string, f is a function name, w is the workflow name, ρ is a data-dependent predicate[15], k is a natural number and n is a natural

[15] The format of predicates ρ is not of the essence here; such predicates will simply be compiled to a τ prefix and the responsibility of deciding them will be delegated to a

number or ∞. ρ is what allows data-dependent choices, all other choices default to deferred choices.

A program consists of a pair (W, D), i.e. a named workflow and a list of function/milestone/newlock declarations. Arbitrary loops are encoded through named functions. Milestones are defined separately and can be read/set by any process knowing the correct channels. Parallel interleaving is handled through global locks. This means that two processes that are not allowed to run at the same time, do not have to be within the same logical block.

The construct **choose any (wait for k)**$\{PP$ **merge**(n) $P\}$ provides multiple choice over the PPs, then merges each of the threads to P, and finally synchronizes all threads. If the clause **wait for** k is given, the synchronization will be an *N-out-of-M Join*. If the clause is not provided, synchronization will wait for all threads to signal done. In the **merge** part of the clause a value of $n = \infty$ signifies all threads PP should merge to P upon completion. A value of $n \neq \infty$ signifies that only the first n threads to complete should give rise to an instantiation of P. If **merge**$(n)P$ is missing, n is taken to be 0 and P can be anything. The construct **do all** implements parallel split with the same options of combining with merge and synchronize patterns. The remaining constructs should be self-explanatory.

It may be helpful to expand our example. The example shown in Figure 8 is based loosely on the Petri net example at [38] to make it easier to compare the Petri net approach to a CCS approach.

Compiling the workflow description language is a relatively easy task since the language is based directly on patterns that have already been described in CCS. The main transformation $\mathcal{T}[\![\cdot]\!]$ (see figure 9) is a map $W \cup D \to Channel \to CCS$, where $Channel$ denotes the set of valid channel names. The following auxiliary function are needed: mergeprefix is the map $\mathbb{N} \cup \{\infty\} \times Channel \times Channel \to CCS$ defined by:

$$\begin{aligned} \text{mergeprefix}(\infty, ok, go) &= ok?*.go! \\ \text{mergeprefix}(n, ok, go) &= \underbrace{ok?.go!.\cdots.ok?.go!}_{n}.ok?* \end{aligned}$$

The function $\nu : \{()\} \to Channel$ returns a fresh channel name that has not previously been used.

data-aware layer. The idea of parameterizing over the predicate/expression language is also found in BPEL, which can use XPath or another plugged-in language.

Fig. 8 A Larger Example: How To Become a Recording Star

```
workflow Becomearecordingstar =
    chooseone {
        ⇒ call (Workyourwayup)
        ⇒ call (Trytogetlucky)
    };
    Makerecord;
    doall {
        ⇒ chooseone {
            ⇒ Developasanartist
            ⇒ Developbadhabits
            }
        ⇒ Rehearsetour;
            Dotour
    }
end
```

```
fun Workyourwayup =
    multi {Learntoplay};
    chooseone {
        ⇒ Joinaband
        ⇒ Decidetogosolo
    };
    chooseany {
        ⇒ multi {Writesong}
        ⇒ multi {Performlive}
    }
end

fun Trytogetlucky =
    Doaudition;
    chooseone {
        ⇒ Audition.failed
            chooseone {
                ⇒ call (Trytogetlucky)
                ⇒ call (Workyourwayup)
            }
        ⇒ Audition.passed
            Doeverythingyouaretold
    }
end
```

Fig. 9 The Transformation $\mathcal{T}\llbracket\cdot\rrbracket$ from SMAWL to CCS

$$\mathcal{T}\llbracket\textbf{workflow}\ w = P\ \textbf{end}\rrbracket = \mathcal{T}\llbracket P\rrbracket$$

$$\mathcal{T}\llbracket\textbf{fun}\ f = P\ \textbf{end}\ D\rrbracket = \lambda ok.f?{*}.\mathcal{T}\llbracket P\rrbracket f \mid \mathcal{T}\llbracket D\rrbracket ok$$

$$\begin{aligned}&\mathcal{T}\llbracket\textbf{milestone}(ison, isoff, set, clear)\ D\rrbracket = \\ &\quad \lambda ok.\mathrm{Milestone}(ison, isoff, set, clear) \mid \mathcal{T}\llbracket D\rrbracket ok\end{aligned}$$

$$\begin{aligned}&\mathcal{T}\llbracket\textbf{newlock}(l, u)\rrbracket = \lambda ok.\mathrm{let}\ unlocked \Leftarrow \nu()\ \mathrm{in}\ \mathrm{let}\ locked \Leftarrow \nu()\ \mathrm{in} \\ &\quad unlocked? * .l?.locked! \mid locked? * .u?.unlocked! \mid unlocked!\end{aligned}$$

$$\mathcal{T}\llbracket\textbf{lock}(l, u)\{P\}\rrbracket = \lambda ok.\mathrm{let}\ ok' \Leftarrow \nu()\ \mathrm{in}\ l!.\mathcal{T}\llbracket P\rrbracket ok' \mid ok'?.u!.ok!$$

$$\mathcal{T}\llbracket activity\rrbracket = \lambda ok.activity!.activity?.ok!$$

$$\mathcal{T}\llbracket\textbf{send}\ f\rrbracket = \lambda ok.f!.ok!$$

$$\mathcal{T}\llbracket\textbf{receive}\ f\rrbracket = \lambda ok.f?.ok!$$

$$\mathcal{T}\llbracket P; Q\rrbracket = \lambda ok.\mathrm{let}\ ok' \Leftarrow \nu()\ \mathrm{in}\ \mathcal{T}\llbracket P\rrbracket ok' \mid ok'?.\mathcal{T}\llbracket Q\rrbracket ok$$

$$\mathcal{T}\llbracket\rho\ P\rrbracket = \lambda ok.\tau.\mathcal{T}\llbracket P\rrbracket ok$$

$$\mathcal{T}\llbracket\textbf{choose one}\ \{\Rightarrow P_1 \Rightarrow \cdots \Rightarrow P_n\}\rrbracket = \lambda ok.\mathcal{T}\llbracket P_1\rrbracket ok + \cdots + \mathcal{T}\llbracket P_n\rrbracket ok$$

$$\begin{aligned}&\mathcal{T}\llbracket\textbf{choose any (wait for k)}\ \{\Rightarrow P_1 \Rightarrow \cdots \Rightarrow P_n\ (\textbf{merge}\ l\ Q)\}\rrbracket = \\ &\quad \lambda ok.\mathrm{let}\ ok' \Leftarrow \nu()\ \mathrm{in}\ \mathrm{let}\ ok'' \Leftarrow \nu()\ \mathrm{in}\ \mathrm{let}\ ok''' \Leftarrow \nu()\ \mathrm{in} \\ &\qquad (\mathcal{T}\llbracket P_1\rrbracket ok' + lazy!) \mid \cdots \mid (\mathcal{T}\llbracket P_n\rrbracket ok' + lazy!) \mid lazy?{*} \\ &\qquad \mid \mathrm{mergeprefix}(l, ok', ok'') \mid ok''?{*}.\mathcal{T}\llbracket Q\rrbracket ok''' \\ &\qquad \mid \underbrace{ok'''?.\cdots.ok'''?}_{k}.(ok! \mid ok'''?{*})\end{aligned}$$

$$\begin{aligned}&\mathcal{T}\llbracket\textbf{do all (wait for}\ \mathrm{k})\ \{\Rightarrow P_1 \Rightarrow \cdots \Rightarrow P_n\ (\textbf{merge}\ l\ Q)\}\rrbracket = \\ &\quad \lambda ok.\mathrm{let}\ ok' \Leftarrow \nu()\ \mathrm{in}\ \mathrm{let}\ ok'' \Leftarrow \nu()\ \mathrm{in}\ \mathrm{let}\ ok''' \Leftarrow \nu()\ \mathrm{in} \\ &\qquad \mathcal{T}\llbracket P_1\rrbracket ok' \mid \cdots \mid \mathcal{T}\llbracket P_n\rrbracket ok' \\ &\qquad \mid \mathrm{mergeprefix}(l, ok', ok'') \mid ok''?{*}.\mathcal{T}\llbracket Q\rrbracket ok''' \\ &\qquad \mid \underbrace{ok'''?.\cdots.ok'''?}_{k}.(ok! \mid ok'''?{*})\end{aligned}$$

$$\mathcal{T}\llbracket\textbf{call}\ f\rrbracket = \lambda ok.f!.f?.ok!$$

$$\mathcal{T}\llbracket\textbf{cancel}\ \{P\}\rrbracket = \lambda ok.\mathcal{C}\llbracket\mathcal{T}\llbracket P\rrbracket ok\rrbracket$$

$$\begin{aligned}&\mathcal{T}\llbracket\textbf{multi}(n)\ \{P\}\rrbracket = \lambda ok.\mathrm{let}\ create \Leftarrow \nu()\ \mathrm{in}\ \mathrm{let}\ ok' \Leftarrow \nu()\ \mathrm{in} \\ &\quad \underbrace{create!.\cdots.create!}_{n}.\underbrace{ok'?.\cdots.ok'?}_{n}.ok! \mid create?{*}.\mathcal{T}\llbracket P\rrbracket ok'\end{aligned}$$

$$\begin{aligned}&\mathcal{T}\llbracket\textbf{multi}(\infty)\ \{P\}\rrbracket = \lambda ok.\mathrm{let}\ inc \Leftarrow \nu()\ \mathrm{in}\ \mathrm{let}\ dec \Leftarrow \nu()\ \mathrm{in}\ \mathrm{let}\ zero \Leftarrow \nu()\ \mathrm{in} \\ &\quad \mathrm{let}\ loop \Leftarrow \nu()\ \mathrm{in}\ \mathrm{let}\ ok' \Leftarrow \nu()\ \mathrm{in}\ \mathrm{let}\ create \Leftarrow \nu()\ \mathrm{in} \\ &\quad loop! \mid loop?{*}.(create!.loop! + zero?.ok! + ok'?.dec?.loop!) \\ &\quad \mid create?{*}.inc!.\mathcal{T}\llbracket P\rrbracket ok' \mid Counter(inc, dec, zero)\end{aligned}$$

The transformation of the **cancel** construct makes use of the function $\mathcal{C}[\![\cdot]\!]$, which inserts cancellation point at all possible states as discussed in the **Cancel Case** pattern. Formally, the cancellation transformation $\mathcal{C}[\![\cdot]\!]$ on CCS expressions is expressed as:

$$\begin{aligned}
\mathcal{C}[\![\mathbf{0}]\!] &= \mathbf{0}\\
\mathcal{C}[\![\alpha.P]\!] &= \alpha.\mathcal{C}[\![P]\!] + cancel?\\
\mathcal{C}[\![P + Q]\!] &= \mathcal{C}[\![P]\!] + \mathcal{C}[\![Q]\!] + cancel?\\
\mathcal{C}[\![P \mid Q]\!] &= \mathcal{C}[\![P]\!] \mid \mathcal{C}[\![Q]\!] + cancel?\\
\mathcal{C}[\![\mathsf{new}\ \ a\ P]\!] &= \mathsf{new}\ \ a\ \mathcal{C}[\![P]\!] + cancel?\\
\mathcal{C}[\![a?{*}x.P]\!] &= (a?{*}x.\mathcal{C}[\![P]\!]) + cancel?
\end{aligned}$$

where α denotes any prefix τ, $a?x$, or $a!x$.

Given a program (W, D), the CCS expression representing the program is obtained as $\mathcal{T}[\![W]\!]abort \mid \mathcal{T}[\![D]\!]dontcare$. Incidentally, the transformation also shows, that new operators can statically be removed bar the cases where new is used inside replicated processes such as the *Counter* processes.

6.4.7 Classification of Workflow Models

Given the control flow patterns on one side and a wide swath of potential models on the other side, it behooves us to bring these together on a more formal footing. At least one such attempt has been made:

> "Specifically only two products Verve and Forte Conductor support Pattern 8 (Multimerge) and they have very similar support for the rest of the patterns. We will classify them into the class of Standard Workflow Models with the main characteristic being the ability to create multiple concurrent instances of one activity. In contrast Staffware Changengine and iFlow do not have support for Pattern 8 (MultiMerge) otherwise they are very similar to Standard Workflows. We will classify them as Safe Workflow Models. Both SAP R/3 Workflow and FileNet Visual WorkFlow do not support Pattern 10 (Arbitrary Cycles) and through the Test Harness we have learned that due to syntactical restrictions certain processes cannot be modelled. We will consider these products to be members of a class called Structured Workflow Models. Finally only one product MQ Series Workflow supports Pattern 7 (Synchronizing Merge) and we will consider it a member of a class which we will refer to as Synchronizing Workflow Models." p. 96 [16]

YAWL belongs to the Standard Workflow Models. It does allow *Synchronizing Merge*, but the important observation is that the semantics of the merge is based on global runtime reachability analysis, not on propagation of boolean synchronization signals. The rationale for the name "Synchronizing Workflow Models" for models supporting *Synchronizing Merge* is that they *always* synchronize on all inbound links. E.g. BPEL waits until either a true or false signal has been received on all links before continuing. In essence, the *Synchronizing Merge* problem has been avoided by recasting it as pure *Synchronization*, hence the name Synchronizing Workflow Models. Kiepuszewski proves the interesting result that Synchronizing Workflow Models cannot deadlock! (This result also holds true for WS-BPEL [22]). YAWL, by contrast, does not wait for a signal from all links; rather its runtime reachability algorithm [21] determines whether a signal will ever arrive or not.

Kiepuszewski [16] argues that Standard Workflow Models should be preferred, because the convenience of generativity through *Multiple Merge* and arbitrary loops supersedes the benefits of ease of synchronization. It is true within Kiepuszewski's framework that arbitrary loops are a serious problem for Synchronizing Workflow Models. However, SMAWL, by virtue of its ties to CCS, offers the options to spawn both synchronized (**call**) and non-synchronized (**send**) threads at any point as well as ad hoc synchronization (**receive**)—while keeping a block structure convention, where all sub-expressions deliver explicit termination signals. SMAWL has at least the expressiveness of Standard Workflow Models, but at times it will be necessary to split complex non-nested cycles into smaller functions/sub-workflows

6.5 Related Work

Dong and Shensheng presented their encoding of some of the patterns in an unpublished paper in 2004 [8]. Their encodings differ a great deal from the ones presented here, but more fundamentally they use the channel-passing facility of π-calculus, whereas the encodings here do not and thus can be expressed in CCS. In our view the language should be as simple as possible, and as we have demonstrated, the full power of π-calculus is not needed. Also, staying within CCS makes the patterns more amenable to use in the tools current available.

In a related paper Puhlmann and Weske demonstrate control flow encodings in π-Calculus using an ECA (Event/Condition/Action) approach [25]. They, too, routinely invoke channel-passing without supplying a particular reason. The *Cancel Case* pattern is avoided altogether, and the *Synchronizing Merge* semantics is left to an unspecified runtime component that checks for reachability.

6.6 Conclusions and Future Work

In this section we:

(1) Discussed the differences between workflows and conventional programs.
(2) Showed a CCS encoding of the 20 control patterns.
(3) Presented two new patterns *(16a) Deferred Multiple Choice* and *(8a) N-out-of-M Merge.*
(4) Hypothesized the value of an overlaying operator.
(5) Analyzed the 20 control flow patterns.
(6) Sketched SMAWL, a set of high-level, CCS-based constructs covering all 20 patterns.

All these items aim to widen our understanding of workflow language design. A workflow language should be have minimal encoding bias, be as simple as possible, cover all 20 patterns (and more), lend itself to program analysis and transformation, and have a clear semantics. If we wish to cover all 20 patterns and have a clearly defined semantics, several issues need to be addressed:

- Split/join separation is necessary unless we enumerate all combinations in block structures and introduce BPEL-style links or a similar free-graph technique.
- Separation should also go for *Multiple Instance* patterns.
- *Synchronizing Merge* remains a problematic but necessary pattern. It must be given a formal semantics; and should preference should be given to a semantics that will scale to distributed settings.
- All patterns should have a *formal* description. Overlaps should be reduced as much as possible.
- Some patterns—e.g. *Interleaving* and *Milestone*—could be understood better if perceived as data patterns.
- Simple predicates on splits and joins would could do a lot of the work needed (cf. EPCs or BPEL XPath Join Conditions).

If in addition we desire a workflow language that is simple, unbiased, and amenable to analysis, we must identify the shortcomings of current standards and models and seek out ways alleviate those. To that end, we have established that:

- Only allowing block structure—e.g. XLang—in insufficient. It is provably less expressive [16].
- BPEL circumvents this through links, but as a model lacks proper support for multiple instances. BPEL also lacks a concisely described semantics and imposes serious constraints on free-graph idioms (links cannot cross certain blocks etc.).

- The π-Calculus and CCS can easily serve as foundation. Like Petri nets they have serious shortcomings for cancellation patterns. In distributed workflow systems, where a solicited task can be accepted a several different peers, the π-Calculus and CCS suffer from the consensus problem—an arbiter is needed to decide who actually gets the task. As long as there is only one centralized workflow server, this is not a problem. The system may experience scalability issues because the acceptance of one task causes the removal of other tasks, and this must happen atomically. Also see section 7 for more on solicitation and allocation patterns, also known as resource patterns.
- Petri nets are good de facto workflow modeling, but they lack flexible constructs for generativity (cannot copy nets). Whereas one has to concede that Petri nets work surprisingly well for free-form graphical structures, the introduction of advanced cancellation, exceptions, and distributed failure strongly suggests a less graphical approach. Petri nets do not handle cancellation and exceptions well—if at all—at this point [39], and complex cancellation/failure scenarios often do not have a pleasant graphical representation. More importantly, the workflow designer should not be limited because of graphical restrictions.
- A formal framework for expressiveness should be put in place. E.g. combining bisimulation notions with Felleisen's characterization of expressiveness[9] [16] . This should lead to a partial order on the expressiveness of workflow models.

The CCS encoding of the control flow patterns and the subsequent pre-design of SMAWL provided a response to van der Aalst's challenges 4–7 to (4) model the Petri net shown in figure 1, (6) model existing patterns, (5) provide new challenges, and (7) suggest new patterns (especially using mobility).

We expressed all patterns and several variants, e.g. we can combine deferred and non-deferred choices (it is not clear if this is possible using the current patterns). It would be interesting to see if Petri nets can handle the overlaying operator more elegantly than CCS. Also, a practical investigation into applicability of the overlaying operator might be very fruitful. In remains open if the newly discovered patterns occur often enough in practical settings to warrant their incorporation with the rest. Real world examples to support them would be very valuable.

The challenges proposed π-calculus, but mobility was not needed. It is our opinion that the foundation should be kept as simple as possible, and this research has in fact shown that CCS is sufficient for the purposes of the current workflow

[16] Quoting Felleisen [9]: ..."more expressive" means that the translation of a program with occurrences of one of the constructs c_i to the smaller language requires a global reorganization of the entire program.

patterns [17] . The response to the challenge is necessarily this: the coding of the 20 patterns has provided no compelling reasons to use the channel-passing mechanism of π-calculus. We see no need for the notion of mobility found in π-calculus for workflow systems. Workflow systems abstract away from the actual channels of delivery. Essentially, workflow systems deal with the flow of command/control and not as much the channels for exchanging messages, if one desires to shift focus to the flow of data—and in particular which channels are used for this—then maybe π-calculus is appropriate. It would seem that the π-calculus notion of mobility would be convenient for modeling service infrastructure and implementation of actual workflow tasks. For now this means that control flows are simpler and therefore easier to analyze.

When debating expressiveness an easy, but vacuous point is Turing completeness. Indeed, the π-Calculus is Turing complete, but in its raw form it is unsuited for any non-trivial programming/modeling task. For domain-specific languages like the ones debated here, it would seem more reasonable to consider criteria such as (a) ontological fit (i.e. domain closeness, convenience of expressing common idioms), (b) expressiveness, (c) amenability to formal analysis, (c) simplicity, and, for workflow languages, (d) graphical representation.

It remains extremely important to provide an intuitive graphical user interface—when possible—and easily understandable high-level abstractions for the user, and this should be addressed in future work. SMAWL took the first speculative step in the direction of a high-level control flow language, but SMAWL still lacks a graphical user interface (auto-generation and editing) and a base language for data manipulation. The most significant shortcoming of SMAWL it semantics. SMAWL semantics are currently defined in terms of the map $\mathcal{T}[\![\cdot]\!]$, but we have no guarantee that this map is correct (correct with respect to what?) or that SMAWL actually covers the patterns amply. The formal pattern description and expressiveness framework sought above will help accommodate this in the future.

SMAWL was designed with two goals in mind: covering all patterns and providing high-level constructs. These goals were met, although as pointed out, eliminating signal passing however nice was not possible. SMAWL being on syntactic sugar on CCS retains the properties of CCS in terms of analysis. Some more information is available because an analysis can draw on both the SMAWL program and the derived CCS expressions.

[17] Brief studies suggest that the Join calculus [11] is syntactically more convenient because it emphasizes atomic joins. The problems with distribution remain, though.

SMAWL misses a graphical representation, which could be an adapted version of UML 2.0 Activity Diagrams. As demonstrated by Wohed et al. [48] UML 2.0 Activity Diagrams are able to express almost all workflow control patterns [18] . SMAWL still needs a companion data language, it needs distribution, and possibly code mobility and locality. It is highly likely, that SMAWL would fall short for purposes of process mining, simply because the constructs are so chunky. This is an open question.

It is noteworthy that we have been able to make do so far without the notions of mobility or locality found in π-Calculus and Mobile Ambients [5] or Bigraphs[15], respectively. Furthermore, as noted, we have been able to accept the artificial global consensus inherent in the π-Calculus and CCS, simple because we worked with the assumption of a centralized workflow engine. If we loosen this asumption, models like Actors [2] or the recently proposed Transactors [10] may prove more fit.

7 Data, Resource, and Service Interaction Patterns

In the wake of the control flow patterns, data patterns [28] and resource patterns [29] were proposed. Thus far 39 data patterns are documented, spanning the areas of scope (*visibility*), data sharing (*interaction*), data passing (*data transfer*), and interfacing with control flow (*data-base routing*). The data patterns contain few controversial ideas, but they do seem slanted towards an imperative paradigm where shared data can be updated. The paper [28] analyzes BPEL4WS and XPDL with respect to the data patterns.

Resource patterns describe the mechanism of delegating enabled tasks in workflow to willing and able agents for execution. They include push and pull patterns, round robin allocation, history-based allocation, clustered allocation, role-based allocation, suspension/resumption, and several others. Resource patterns are implemented as delegation layer between the workflow engine soliciting tasks and the agents executing tasks. The most likely model is a state machine representing the state of each task, but this imposes new requirements on the interface to tasks (e.g. they must be able to receive suspend/resume/cancel messages). The setup

[18] Workarounds using signal-passing are needed for Interleaved Parallel Routing and Milestone. MI without a priori Runtime Knowledge is not supported and there are serious deficiencies when it comes to the ever-illusive Synchronizing Merge. Activity Diagrams are significantly weaker when it comes to data patterns where about half of the 40 patterns are covered.

has a certain similarity to auction mechanisms, and thus it would be interesting to consider resource patterns from a game-theoretic perspective (offering tasks to the highest bidder etc.).

References

[1] W. Aalst, M. Dumas, A. Hofstede, and P. Wohed. Analysis of web services composition languages: The case of BPEL4WS. In *Proc. of ER'03*, volume 2813 of *Lecture Notes in Computer Science (LNCS)*, pages 200–215. Spinger, 2003.

[2] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–69, January 1997.

[3] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, and Yichen Xie. Zing: Exploiting program structure for model checking concurrent software. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.

[4] Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.

[5] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126. ACM Press, 1983.

[7] Francisco Curbera, Rania Khalaf, Frank Leymann, and Sanjiva Weerawarana. Exception handling in the bpel4ws language. In Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske, editors, *Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pages 276–290. Springer, 2003.

[8] Yang Dong and Zhang Shensheng. Modeling workflow patterns with pi-calculus. Unpublished. http://www.workflow-research.de. Available from http://www.workflow-research.de.

[9] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 1990.

[10] John Field and Carlos A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 195–208. ACM, 2005.

[11] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January21-24 1996. ACM.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley Professional, 1st edition edition, January 1995.

[13] C.A.R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, 1985.

[14] Michael R. A. Huth and Mark Ryan. *Logic in computer science: modelling and reasoning about systems.* Cambridge University Press, 2000.

[15] O. Jensen and R. Milner. Bigraphs and mobile processes. Technical Report 570, Computer Laboratory, University of Cambridge, 2003.

[16] Bartosz Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows.* PhD thesis, Queensland University of Technology, 2002.

[17] E. Kindler. On the semantics of EPCs: A framework for resolving the vicious circle. Technical report, Reihe Informatik, University of Paderborn, Paderborn, Germany, August 2003.

[18] Naoki Kobayashi. Type systems for concurrent programs. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453. Springer, 2002.

[19] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice-Hall, 1989.

[20] Robin Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, 1999.

[21] W.M.P. van der Aalst Moe Thandar Wynn, David Edmond and A.H.M. ter Hofstede. Achieving a general, formal and decidable approach to the OR-join in workflow using reset nets. Technical Report FIT-TR-2004-02, Queensland University of Technology, Brisbane, 2004. QUT Technical Report.

[22] C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, and H.M.W. Verbeek. Formal semantics and analysis of control flow in ws-bpel. Technical Report BPM-05-13, BPM Center Report, 2005.

[23] *High-level Petri Nets – Concepts, Definitions and Graphical Notation*, final draft international standard iso/iec 15909, version 4.7.3 edition, May 2002.

[24] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, 2000.

[25] Frank Puhlmann and Mathias Weske. Using the pi-calculus for formalizing workflow patterns. In *Business Process Management, Third International Conference, BPM 2005, Nancy, France, September, 2004. Proceedings.*, LNCS. Springer Verlag, 2005.

[26] Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 166–179. Springer, 2002.

[27] Peter Rittgen. Quo vadis EPK in ARIS ? - Ansätze zu syntaktischen Erweiterungen und einer formalen Semantik. *Wirtschaftsinformatik*, Heft 1:27–35, February 2000.

[28] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns. Technical Report FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

[29] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow resource patterns. BETA Working Paper Series WP 127, Eindhoven University of Technology, Eindhoven, 2004.

[30] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes.* Cambridge University Press, 2001.

[31] August-Wilhelm Scheer. *Architektur integrierter Informationssysteme: Grundlagen der Unternehmensmodellierung.* Springer-Verlag, 1992.

[32] Alec Sharp and Patrick McDermott. *Workflow Modeling - Tools for Process Improvement and Application Development.* Artech House Publishers, 2000.

[33] Christian Stefansen. A declarative framework for enterprise information systems. Technical report, Department of Computer Science, University of Copenhagen, Sep. 2005. Qualification report.

[34] Christian Stefansen. SMAWL: A SMAll Workflow Language based on CCS. In *Proceedings of the CAiSE Forum of the 17h Conference on Advanced Information Systems Engineering*, June 2005.

[35] Christian Stefansen. SMAWL: A SMAll Workflow Language based on CCS. Technical Report TR-06-05, Harvard University, Division of Engineering and Applied Sciences, Cambridge, MA 02138, March 2005.

[36] Satish Thatte, Assaf Arkin, Sid Askary, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Canyang Kevin Liu, Prasad Yendluri, and Alex Yiu. Web services business process execution language version 2.0 working draft, 13 july 2005. Technical report, OASIS Open, Inc., 2005.

[37] Wil M. P. van der Aalst, Lachlan Aldred, Marlon Dumas, and Arthur H. M. ter Hofstede. Design and implementation of the yawl system. In Anne Persson and Janis

Stirna, editors, *CAiSE*, volume 3084 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2004.

[38] Wil M.P. van der Aalst. Workflow patterns.

[39] Wil M.P. van der Aalst and Arthur 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, Aarhus, Denmark, August 2002. DAIMI.

[40] Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, 2003.

[41] Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, B. Kiespuszewski, and A.P. Barros. Workflow patterns. Technical report, Eindhoven University of Technology, GPO Box 513, NL-5600 MB Eindhoven, The Netherlands, 2002.

[42] Wil M.P. van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[43] W.M.P. van der Aalst. Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype". http://tmitwww.tm.tue.nl/research/patterns/download/pi-hype.pdf, 2004.

[44] W.M.P. van der Aalst, J. Desel, and E. Kindler. On the semantics of EPCs: A vicious circle. In M. Nüttgens and F.J. Rump, editors, *Proceedings of the EPK 2002: Business Process Management using EPCs*, pages 71–80. Gesellschaft für Informatik, November 2002.

[45] W.M.P. van der Aalst and A.H.M. ter Hofstede. Yawl: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[46] W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G.Schimm, and A.J.M.M. Weijters. Workflow mining: a survey of issues and approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.

[47] Wikipedia, the free encyclopedia. The Wikimedia Foundation.

[48] Petia Wohed, Wil M.P. van der Aalst, Marlon Dumas, Arthur H.M. ter Hofstede, and Nick Russell. Pattern-based analysis of uml activity diagrams. http://is.tm.tue.nl/research/patterns/download/uml2patterns/20BETA%20TR.pdf, 2004.

[49] The Zing model checker. http://research.microsoft.com/zing/.

[50] Michael zur Muehlen. Workflow research. http://www.workflow-research.de.

Using Soft Constraints to Guide Users in Flexible Business Process Management Systems

Christian Stefansen[a] Signe Ellegård Borch[b]

[a] *University of Copenhagen, Copenhagen, Denmark*

[b] *IT University of Copenhagen, Copenhagen, Denmark*

Abstract

Current Business Process Management Systems (BPMS) allow designers to specify processes in highly expressive languages supporting numerous control flow constructs, exceptions, complex predicates, etc., but process specifications are expressed in terms of hard constraints, and this leads to an unfortunate trade-off: information about preferred practices must either be abandoned or promoted to hard constraints. If abandoned, the BPMS cannot guide its users; if promoted to hard constraints, it becomes a hindrance when unanticipated deviations occur.

Soft constraints can make this trade-off less painful. Soft constraints specify what rules can be violated and by how much. With soft constraints the BPMS knows what deviations it can permit, and it can guide the user through the process. The BPMS should allow designers to easily specify soft goals and allow its users to immediately see the seriousness of their violations at runtime. We believe, this provides a combination of guidance and flexibility not otherwise attainable.

Key words: business processes, workflow, constraint specification, soft goals, business process flexibility, Pareto optimality, soft constraints, workflow flexibility

1 Introduction

The past five–ten years have witnessed significant changes in the way business processes management systems (BPMS) are perceived, as well as the context in which they take part. Whereas BPMS were previously thought of as support systems, the interest in business processes is now focused on systems that *actively execute* processes in collaboration with other processes, services, and humans.

The development that lead to BPMS taking the center stage was brought on by management evangelists, who solicited the successful business process perspective (Hammer & Champy, Davenport, Michael Porter). Later important legal changes took place. Most notoriously, the *Sarbanes-Oxley Act* was signed into law and now requires managers to oversee that companies directly document the processes that produce any data used in external reporting under penalty of law.

At the same time, as the Internet grew in significance, so did the pressure to make systems Internet-accessible and interoperable. The *service-oriented architecture* (SOA) became the enabling paradigm, and it was followed by numerous standards designed to make a shift to the new paradigm possible. Previously, integrating business processes across several systems and business entities had been cumbersome or just plain uneconomical, but as businesses gradually started the move toward SOA, this became much easier to achieve. This development can also be seen as an instance of a general move towards higher levels of abstraction. Successive generations of programming models tend to enable increasingly abstract entities to be represented directly; we have now reached the level where distributed business processes can be represented explicitly.

In short, the business climate, the regulatory climate, and the prevailing IT architecture have all become increasingly favorable to the process paradigm. This development has allowed the BPMS to take on a new role: in a setting where the individual steps of a business process are exposed as Internet services, the BPMS can be an *active orchestrator* that *executes* the business processes by delegation, rather than just serving as a *passive support system* deep inside an isolated part of the organization.

The change from passive support system to active orchestrator places new demands on the BPMS. In passive support systems the business process could be written quite stringently as a program, and if the real-world process happened to deviate from the system-prescribed one, this was not necessarily a problem. As long as the users knew what they were doing, they could ignore the system and report dummy task completions back to the system until the real-world process and the system-prescribed process were in agreement again. Today, because the system orchestrates the process, users will find it much more difficult to ignore or trick the system if the real-world process deviates from the prescribed one. Because the BPMS is what makes the process progress, the system process and the real-world process have to agree to a much larger extent than before.

If the processes are described and executed stringently, users will find them exasperating to work with when the real-world process sees variations that were not anticipated in the process design. A very typical pitfall is to only express the best practice in the process design meaning that the users have to come up with workarounds when best practice cannot be followed. On the other hand if the process designer tries to forestall such rigidity by making the process very flexible, there is usually no way of indicating to the users what the preferred practice is. The result is then a system that provides no *support* to its users. Crudely put, a good system should continuously give users an overview of what they *may* do, what they *should* do, and what they *must* do – and let the process designers easily describe these modalities in the process template.

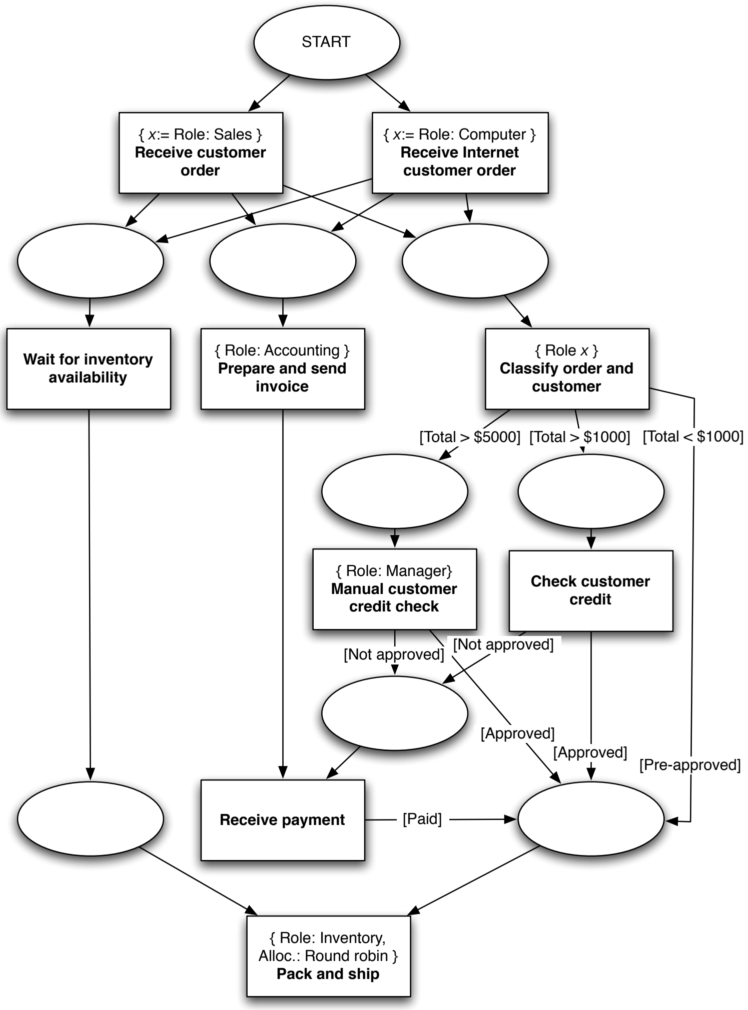
1.1 An Example Process

Let us illustrate these points with an example. Figure 1 shows a simplified order handling process in simplified *Colored Petri Net* notation. Several elements are omitted such as data sources, data formats, timeouts, service connectors, and access control. The process takes a received order from the sales department or from the company's website and facilitates shipping, invoicing, payment, and, if necessary, credit approval. The process specifies a base policy that orders less than \$1000 are pre-approved, orders of more than \$1000 are subject to credit checks, and orders over \$5000 require a manual credit check by a manager. If the customer is not credit approved, advance payment is required.

Having \$1000 and \$5000 as delimiters for a base policy seems very reasonable, but yet it raises the concern: will we *always* adhere strictly to this policy? Presumably not. A customer order of \$1001 should makes us think about bending the rule slightly, and conversely, we may wish to do an additional manual credit check on a \$2000 order, if the customer is known or suspected to be a bad payer.

Hence this policy is clearly bendable. It is also clear that deviating from the policy is less problematic in the case of a \$1001 order than in the case of, say, a \$9500 order. Therefore the user should be allowed to violate this policy and make an informed decision, and in doing so, the system should somehow indicate how serious a violation is being made. Introducing an extra path in the process – call it *User override* – does not adequately address the issue, because it does not measure the gravity of the violation; it merely records the fact that a violation has taken place. Also the existence of such an extra path does explain to the user what is the preferred path or what would be needed to justify an override.

Fig. 1 Simplified order handling process



To sum up, removing the credit policy will deprive the process specification of useful guideline information, but making it a hard constraint will render the users unable to make exceptions even in very reasonable cases. Something in between is needed: we need a measure of the *soft goal*: what is our *credit risk*? Violating the rule once will not seriously damage the company's goal to minimize credit risk, but if we constantly need to violate the credit policy, company performance on the credit risk goal will reflect this, and management will be able to take appropriate steps to rectify the situation.

Now consider the sequential constraint that *Pack and ship* can occur only when *Wait for inventory availability* is complete. Should this constraint be flexible? Definitely not. The constraint is a hard constraint, because shipping a good that one does not have in one's possession amounts to violating the laws of physics. Here the BPMS must stand firm on its constraint. (For the sake of argument, ignore the possibility of partial shipments. Our discussion will argue that generally there are many unanticipated exceptions even to hard constraints and thus hard constraints should be avoided to the largest extent possible.)

Last, consider the constraint that *Pack and ship* should be allocated *round robin* (i.e. by rotation) to employees with the *Inventory* role. Here we are dealing with a constraint that is much less significant than any of the others. It reflects a goal that employees should get a feeling of fairness by carrying out their task in a turn-based fashion, but it is also clear that this constraint must yield to more important operational goals such as delivery time – at least temporarily. Again we have a constraint that should not be elevated to law, but adds guiding information to the process to help users reach soft goals. If we find ourselves breaking the rule systematically, then there is an incongruence between the prescribed process and the real-world process worth looking into. We may find that the inventory department is overburdened, but we are much more likely to find this if the rule is in the process description so that we can systematically record and diagnose violations. If the system does not let users break that rule, they will likely break it anyway, but in roundabout ways that are more difficult to monitor quantitatively or even detect.

1.2 Contributions and outline

Our main contribution is to show how using soft constraints *directly in the business process descriptions* can (a) capture intentional information about best practices, (b) provide support to the user in extremely flexible BPMS, (c) help identify gaps between actual and prescribed practice, and (d) make the trade-off between flexible and rigidity significantly less salient. The paper presents:

(1) An analysis of the trade-off between flexibility and control from the user's perspective (Section 2). Specifically we argue that the user perspective leads to fundamentally different requirements, and that users *will* improvise and build ad hoc systems if sufficient flexibility is not provided.
(2) Issues related to getting flexibility in current systems (Section 3). Current systems provide a plethora of ways to specify flexible workflows, but an important advantage of strict control seems to have been forgotten: that of *guiding* the user. We argue that current systems cannot both be extremely flexible and still provide sufficient guidance, because intentional information is too easily

lost when systems become too flexible.

(3) The case for using soft constraints/goals as a remedy (Section 4). This section explains how soft constraints/goals can alleviate the problem identified in Section 3, by allowing system to be flexible, while capturing enough information to guide the user through the process. While completely flexible systems will let the user do anything, flexible systems *with soft constraints* will allow full flexibility *while supporting the user* in making choices that contribute towards organizational goals.

(4) How soft goals look from the user's point of view (Section 5). This section shows how soft goals can be used in a direct way as part of what the user sees when interacting with the BPMS.

(5) How process designers specify soft goals (Section 6). Here it is explained how to incorporate soft constraints as a direct part of the process specification.

(6) How soft goals can be incorporated into the BPMS by using multi-criteria optimization (Section 7).

(7) How this affects the process design methodology (Section 8). This sections contributes a tentative methodology for using soft constraints when designing, deploying, and modifying business processes.

The paper closes with a brief overview of related work (Section 9) and directions for future research (Section 10).

2 Background: the user perspective

Typically, business process modeling is initiated by management as an effort to optimize business processes, as in the vision of business process reengineering (Hammer, 1990). The focus lies on documenting existing processes, describing to-be processes, and finally, when the BPMS is running, to monitor and analyze actual processes for the purpose of making them more effective. In this way, the feedback provided by the BPM tool is used to manage and control the work practice of the organisation.

From a technological point of view, basic concerns with regard to BPMS are maintenance and migration, architectures and infrastructures, standards, exception handling, programming models and notations. Typical problems are how to centralize control of execution, how to make the BPMS interact with a number of different systems (e.g. ERP systems, legacy systems, Internet services) and users, and how to make runtime changes possible.

This article takes on a different perspective – that of the use context of the BPMS.

The user perspective gives rise to different questions than if flexibility of BPMS is seen from a managerial or technological point of view. Thus our point of departure is the simple question: what is the purpose of a BPMS from the user's perspective? One partial answer is that business process modelling is about reducing the space of choices for the user by identifying a valid and limited set of options for action. The model provides a way to navigate the space of possible actions, indicating what might be the next activity for the user to engage in.

When taking the perspective of the user a central challenge for the BPMS is how to fulfil the conflicting requirements for *flexibility* in work practice and *control* of the work practice (for a discussion of this problem, see e.g. Bernstein 2000).

One might be tempted to think that from the user's point of view, unlimited flexibility in the BPMS would be preferable. However, there are several reasons for introducing restrictions into the system:

(1) Conventions of practice are captured explicitly in an artifact (they are documented)
(2) The right way of doing things does not have to be constructed, or re-negotiated, every time
(3) Process models can serve as *coordination mechanisms* to share knowledge about the state of the work (Schmidt & Simone, 1996)
(4) The system can support the user in not violating business rules (e.g. regarding legal issues, regulations, policies)

On the other hand, if the system prevents its users from making local adjustments of the normal flow when appropriate, it is too rigid.

A widely documented observation within the CSCW field is the gap between the specified processes and the actual work practice:

(1) Users make (well justified) workarounds (see e.g. Kobayashi, 2005)
(2) They use the plans as maps for *situated action*, not as scripts for executing their work (Suchman, 1987)

One explanation of the gaps between the anticipated and actual use of the system is the general observation that IT systems are not always used the way the designer intended (see e.g. Robinson, 1993).

However, the kinds of (mis)uses mentioned above might also indicate design problems inherent to the BPMS. The question is whether the way BPMS are designed today make them a support or a barrier to the actual work practice: are they flexible enough, or too rigid?

One source of too much rigidity is the conflict between a technological perspective and a use perspective on the business processes. It is tempting for an IT expert to see the user as just another resource that can be invoked to execute a task: human and system activities are captured within the same model, and *orchestrated* by the same centralized technology, the workflow engine. In a notation like e.g. *WS-BPEL (Thatte, 2003)* no distinction is made between human and system activities in the process. A different approach is taken in *Windows Workflow Foundation* (Chapell, 2005), where human and system workflows are represented in two different notations (sequential and state-based). The choice to have two separate notations is based on the assumption that human and system workflows are fundamentally different in nature, especially when it comes to their requirements for flexibility: human work practices are much more likely to change, also in an *ad hoc* manner, than system flows.

The question is then: if the workflow system is not assuming that the users act like programs, executing their tasks in a procedural manner that corresponds to the way the workflow was modeled, what would be the right way to design a flexible system where the users would still benefit from the *navigational support* a BPMS can provide?

3 The current state of BPMS

Before proceeding further it is appropriate to examine the current state of affairs.

A fair number of current systems provide flexibility facilities for their users: Cancellation allows processes or parts thereof to be cancelled, tasks and sub-processes may be specified as skipable, users can choose to redo certain parts and of course the user always can make a choice between the alternative paths specified by the workflow designer.

In addition some systems provide policies (i.e. local or global rules) that govern in detail what is permissible in terms of data dimensions such as employee, role, location, project, customer type, etc. Such policies take into account the current data in the process instance and behave accordingly.

However, the current systems are built on the basis of the same modelling paradigm, namely that of *binary classification of sequences of activities.*

The activity of modelling under this paradigm can be understood as classifying all possible sequences of process-related events into allowable sequences and disallowable sequences.

Consider the following business process type [1]:

$a \rightarrow (b\ XOR\ c) \rightarrow d$

In essence, this business process type makes a classification. It classifies all possible sequences of the activities *a*, *b*, *c*, and *d* into those that comply with the description and those that do not comply. The sequences { *<a,b,d>, <a,c,d>* } comply; all others do not.

The example demonstrates a key issue in current systems: there is only one classification. One cannot state e.g. that *<a,b,d>* preferable to *<a,c,d>* or that *<c,d>* is acceptable, although not encouraged. The process designer is forced to make the unpalatable decision of making all possible runs either acceptable or unacceptable. Such a decision is often impossible to make correctly at design-time, and the result is then processes that are too restrictive or too lenient. Hence a more refined type of *specification* is needed, but merely inventing a clever specification in insufficient. The change from binary classification to finer classification must permeate the entire design/use methodology for business processes. See Section 8 for more on the methodology.

It is useful to consider the distinction between binary and finer classification from yet a perspective. When charting business processes, a business analyst is likely to arrive at a semi-formal description that contains many different *types* of constraints and goals. Goals and constraints come from a variety of sources (Regev and Wegmann, 2005) and therefore vary in significance. E.g. the analyst could have established that for three activities *a*,*b*, and *c* the sequence *<a,b,c>* is the best practice, *<b,a,c>* and *<a,c>* are acceptable, *<b,c>* can be done in exceptional cases, and other sequences are logically inconsistent. In other words, the analyst has captured *intentional information* about best practices along with physical constraints.

Yet today's BPMS demands that the process designer discard all the intentional information gathered by the analyst and specify the process as a control flow. If the designer writes

(a OR b) → c

[1] This description omits all other perspectives than the purely process-structural perspective, but it is sufficient for the following discussion.

then the description has lost all intentional information and every part of the specification appears to be a physical constraint. Information about the best practice is lost as well.

This situation is clearly undesirable. The knowledge gathered by the analyst is lost because the system has no way of capturing it. The designer then specifies a process that reveals nothing about the nature of its dependencies and constraints, and finally the user (i.e. the knowledge worker) works with a system that forces one particular view of the process and considers all violations equally unacceptable.

Could one imagine a completely free system that would only *inform* the user about the process, but not force it? Let's consider two extremes of flexibility. The *dictatorial* process specification is one which allows only the strictly complying sequences and blocks the user from doing anything else. The *anarchistic* process specification is one which allows all possible sequences of the three activities *a*, *b*, and *c*. Such a specification provides maximum flexibility, but it must be based on the assumption that users know what they are doing (see e.g. Bider, 2005). Since it is not specified what sequences are preferable to others, the users will not get any support from the anarchistic specification.

To summarize, a *dictatorial specification* (rigid):

(1) Allows only what is narrowly prescribed by the process designer.
(2) Represents only anticipated sequences.
(3) Cannot accommodate deviations and exceptional cases unless they have been meticulously accounted for in the process description. If they have not, users are forced to invent workarounds, and valuable historic information is then not captured in the system. If deviations happen often, users will tend to systematize these in auxiliary, ad hoc systems. This leads to data fragmentation and lack of belief in the system.
(4) Does not represent how business is actually done, only how it should be done according to a process designer's incomplete view of the world.

An *anarchistic specification* (flexible):

(1) Allows every possible combination and repetition of known tasks.
(2) Does not represent or endorse any particular sequences.
(3) Provides no support to the user (cannot suggest what tasks to do in what order to reach the goal).

Neither of these extreme approaches is ideal seen from the user's perspective. In the dictatorial specification the user is supported in anticipated sequences, but denied any deviation. In the anarchistic specification the user is allowed any deviation, but supported in none. In *both* of the specifications and in *any trade-off in between* there is no ranking of different sequences. There is no way of saying what sequences are preferred, discouraged, etc. We can design rigid and flexible processes, but they remain simple binary classifications. Some sequences are acceptable, some are not. The dictatorial specification allows fewer, the flexible specification allows more, but they cannot be ranked according to preference. This is the limitation of the binary classification that most of today's BPMS employ.

What would be nice would be a combination: to be able to specify what sequences were preferred anticipated, but not have to make that specification a hard constraint. Some parts of the specification would be violable, others not. What is needed is a way of specifying this, without just allowing everything to happen indiscriminately.

Our solution is therefore to capture finer classifications of processes explicitly, or put differently, try to make the intentional information gathered by analysts *explicit* in the process specification. This way the user can be supported while we retain the distinction between strict and soft rules. The following section discusses one way of achieving this.

4 Modeling soft goals

As mentioned in the beginning of Section 3 BPMS provide powerful capabilities for specifying permissible and non-permissible courses of action, but this remains a *binary classification* – either a policy is satisfied or not. In the current systems there has been a strong focus on extending expressiveness so that we can accurately specify *hard constraints*, but overlooking the equally important question of *soft goals*. Soft constraints are constraints that should be used as guidelines, but they can be violated in exceptional situations or if the user has a good reason. Soft goals (Yu and Mylopoulos, 1994; Soffer, 2005) are business objectives that are not *directly* linked to business output or performance. Soft constraints are operational guidelines promoting good performance on soft goals. Whereas hard constraints are usually very stable over the evolution of a process, soft constraints change more often.

As explained in Section 3, when users and designers are forced to work with systems that focus unilaterally on hard constraints, they face two equally displeasing alternatives: (1) Important soft constraints are inadvertently promoted to hard constraints because designers add them as policies. The result is a system that is too rigid and inspires hacks that circumvent the constraints by going outside the system. (2) Soft constraints are left out of the system altogether, meaning that important organizational goals are disconnected from, arguably, the organization's most important process tool.

If we accept the premise that BPMS need to account for soft constraints and/or soft goals in addition to hard constraints, the pressing question is this: how is this presented to the direct users of BPMS? We believe that making soft goals and soft constraints visible for the end users is a key issue.

The challenge when modelling both hard and soft constraints is to get the right balance between the two. BPMS must offer enough rigidity to satisfy hard constraints, whether they are business requirements, nonnegotiable logical constraints, data constraints, or externally imposed constraints, such as legislation (Soffer, 2005). At the same time, users must be afforded the maximal flexibility possible, but continually be informed about the impact of their choices on various organizational goals.

As mentioned above, it is not enough to simply add soft goal support to the BPMS. A change of methodology is required to ensure that soft goals are used through the entire process lifecycle – by analysts, designers, reengineers, users, and process miners. Our main focus here is the user perspective, and secondarily the designer's perspective; a more comprehensive methodology will be addressed in future work. For previous work on methodologies incorporating soft goals cf. (Yu and Mylopoulos, 1994; Regev and Wegmann, 2005). In the following we examine soft goals from the perspective of the user and the designer.

5 How users interact with soft goals

In our example process in Figure 1 users will be presented to the activities as they become enabled, e.g. when a sales person is done entering an order received by telephone, the *Prepare and send invoice* activity becomes available to employees with the *Accounting* role, and the *Classify order and customer* activity becomes available to the sales person who entered the order.

In the following let us focus on two soft goals: minimizing *order processing time* and minimizing *credit risk*. Also assume that an automatic credit check takes up to five hours and a manual credit check takes two days. We can measure credit risk as zero when no policy violations take place and as the number of dollars in excess of the limit, when violations do take place. If a sales person, pre-approves a $1400 order, he has traded a *processing time* improvement of five hours with an added *credit risk* of $400. (The exact definition of such soft goals can be debated, and certainly more research on best practices in this field is required. We postpone this discussion to future work and accept the sketched goals for the purpose of illustrating a point.) The user interface must make this trade-off clear before the user finalizes his choice. The user, in other words, should be supported in navigating the decision space.

In general when the user is presented with a choice, the impact of each possibility should be explained in terms of soft goals. Some options are much less reasonable than others and naturally the user interface prioritize. For instance, performing a manual credit check on a $500 order adds nothing to the *credit risk* goal (perhaps a bit unreasonably) and delays the order by two days. This can be determined by a BPMS and supported in the user interface. This idea can be brought to bear on the notion of *dominant choices* introduced in Section 6 and 7.

As an alternative to optimizing and presenting solution pertaining to several goals, we can combine all soft goals into a weighted sum. While this simplifies the presentation – choice can simply be sorted based on one number – it has the drawback of maintaining a static priority between the goals. The weighting can be adjusted over time based on decision analysis, but despite its simplicity, we consider a weighed sum a supplement rather than a substitute for presenting the impact for each soft goal separately.

6 How designers specify soft goals

Soft goals have a large variation in scope: some goals pertain only to one particular activity (the round robin allocation guideline in our example being an case in point), others pertain to part of a process (*credit risk*), and some may be influenced by many running processes of different types (e.g. *processing time*). This indicates that soft goals may be independent of any particular process, but could also part of one. Hence a BPMS *must* allow goals both a system-level and at process-level.

Conceptually we can think of the change of approach in the following way: earlier, the designer would specify a process where every part of the specification was strictly non-violable; now, the designer will specify a process consisting of both a violable part and a non-violable part. The non-violable part will be much smaller than before, and the violable parts of the specification will be annotated with information about their impact on the soft goals identified by the process analyst. (We refer the reader to (Yu and Mylopoulos, 1994) for a method for representing the impact of activities on soft goals.)

Abstractly, we can think of a business process specification as simply a set of constraints and rules. Where previously every constraint or rule was strict, every constraint or rule is now either strict or soft. In the latter case it will annotated with enough information to automatically calculate its impact on soft goals at runtime.

It should be made clear that when we speak of a business process comprising constraints and rules, this includes everything in a process specification: control flow, dataflow, user allocation rules, exceptions, compensation, timeouts, etc. We should note that in most processes there will still be strict rules. E.g. data flows that feed input to computer activities would typically be non-violable, because the computer activity would otherwise fail and leave the process in an undefined state. We advocate to only use strict rules when they are categorically mandated as is the case with the data flow here. Other constraints can be expressed in terms of soft goals and result in a much more pleasing system to work with for all parties involved.

Let us examine several ways of specifying the violable parts (soft constraints) of the process.

(1) Goals can be specified directly as a function on the trace (i.e. history) of the process. E.g. *credit risk* would be specified globally as the sum of all violations:

$$z_{\text{Risk}} =$$

$$\textstyle\sum_{\text{traces}} \max\left([\mathit{Credit\_check} = \text{no}] \cdot (\mathit{total} - \mathit{autoLimit}), 0\right) +$$

$$\textstyle\sum_{\text{traces}} \max\left([\mathit{Credit\_check} = \text{auto}] \cdot (\mathit{total} - \mathit{manLimit}), 0\right)$$

While such a direct expression of the goal will be appealing to optimization adherents, its merits as a tool for process designers are debatable.

(2) A more natural, albeit indirect, approach is to annotate the activities in question locally with a measure of the gravity of a violation, e.g.:

$$\mathit{Total} < \$1000 \quad \text{OTHERWISE:} \quad z_{\text{Risk}} \mathrel{+}= \mathit{Total} - \$1000$$

which states that for the particular path *Total* should be less than \$1000, otherwise the soft goal z_{Risk} is compromised by the *Total* - \$1000. In essence, rules that can be violated carry a price tag for a violation on them. The BPMS still needs to consider a decision horizon beyond just the current task, though, because more serious ramifications of the decision could be waiting further down the line.

Our example process has two soft goals, z_{Risk} and z_{Time}. With the augmented process specification as outlined above, the BPMS is able to calculate for each proposed step the immediate impact on the soft goals.

The specification methods outlined above are just two ways of specifying soft goals. Given the both local and global nature of soft goals, it is prudent to allow specification of goals on both levels. Some subgoals could be computed locally and then be composed to form the main goal globally. This approach retains some modularity and allows the code to be associated to its most closely related level of abstraction.

7 Soft goals as an optimization model

The field of operations management/optimization has been used widely in particularly complex resource allocation scenarios. Given the setup described here, optimization can be used automatically to aid users in their decision-making in a business process based on the process data and the soft goals and hard constraints captured in the process specification. Whereas formulating and solving optimization problems are typically fairly complex endeavours that do not apply directly to the soft goals, it is useful to understand the workings of the BPMS in terms of an optimization problem.

Our example has soft goals z_{Risk} and z_{Time}. In the simplest setup the BPMS would limit its horizon to only one step in the process. Hence for each enabled task, it would compute its impact on the soft goals, if it were to be executed. This is a straight-forward computation, whose result can be shown to the user to support decision-making.

This approach will often be insufficient because a locally attractive decision may lead down a path that adversely affects the soft goals. Hence a longer look-ahead is necessary, and this can be accomplished with an optimization problem formulation. It should be pointed out that the formulation will be seen neither by designers nor users; it merely defines how the BPMS finds optimal ways to progress so it can support the user by suggesting these. The BPMS builds the formulation based on the soft goal information provided by the designer.

Given a set of soft goals $z_1, z_2, \ldots, z_n$, the optimization problem can be rendered schematically as:

minimize $z_1, z_2, \ldots, z_n$

subject to $\langle$ *hard constraints in process* $\rangle$

where the decision variables describe the space of choices theoretically available to the user, and the hard constraints are derived directly from the process specification.

A concrete example will convey the intuition more clearly: consider the subset of our general order handling process shown in Figure 2. The decision variable has been made explicit in Figure 2 as *Check*, and it can take on one of the values {*no*, *auto*, *manual*}. As before, there are two soft goals to minimize, namely *credit risk* and *processing time* defined as:

credit risk =

$[\mathit{Credit\_check} = \text{no}] \cdot \max(\mathit{total} - \mathit{autoLimit}, 0) +$

$[\mathit{Credit\_check} = \text{auto}] \cdot \max(\mathit{total} - \mathit{manLimit}, 0)$

proc.time =

$[\mathit{Credit\_check} = \text{auto}] \cdot 5 \text{ hours} +$

$[\mathit{Credit\_check} = \text{manual}] \cdot 2 \text{ days}$

where [*condition*] evaluates to 1 when true and 0 when false. The problem thus becomes

minimize *credit risk*, *proc.time*

subject to *Classify* before *Log*

Classify before *Check*

Classify before *Manual check*

$\mathit{Appr.} = 1 \vee \mathit{Pay}$ before *Pack*

(etc.)

where *Act1* before *Act2* is a shorthand for introducing time variables *start* and *done* for the start and completion times of the activities, and demanding that *Act1.done* < *Act2.start*.

Again it should be pointed out that neither the user nor the designer will see this problem formulation. This is merely a conceptual way of describing the internal working of the BPMS necessary to support soft goals.

The optimization problem has several solutions, and in general there will be many more solutions than what can be easily presented to the user. To overcome this the system should present only *Pareto optimal* solutions (*dominant* solutions), i.e. solutions where no soft goal can be improved without adversely affecting another soft goal. Assuming we receive an order of \$1400 in our example, at least three solutions are possible:

(1) *Check* = no => *credit risk* = \$400, *proc.time* = 0
(2) *Check* = auto => *credit risk* = \$0, *proc.time* = 5 h
(3) *Check* = manual => *credit risk* = \$0, *proc.time* = 2 d

Choice 3 is dominated by choice 2 (but not by choice 1!), and choice 1 and choice 2 are Pareto optimal and do not dominate each other. Hence choices 1 and 2 will be the first ones to be suggested to the user.

The system would also find a number of solution variations pertaining to the choice of what employee carries out the *Pack and ship* activity, but that decision variable is neutral to the decision at hand.

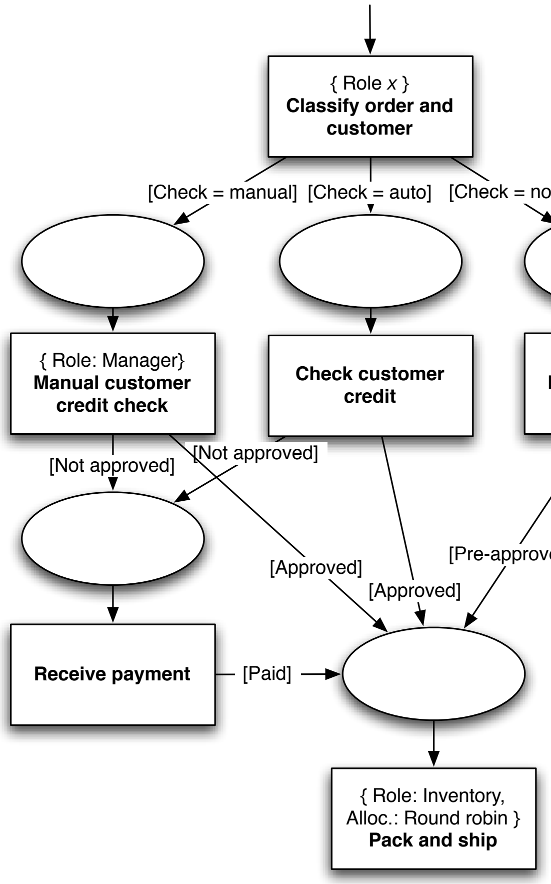
Note that by solving the optimization problem, we get a complete plan for action – not just for the current decision, but for the entire process. While this may prove unnecessary or too computationally intensive in practice, this is a convenient feature to be able to invoke occasionally.

Every time a step is taken, the optimization problem is rewritten to reflect the new state of the process. The optimization problem formulation should be derived automatically from the workflow specification and the soft goals, so the user or the designer will never see anything like the minimization problem above.

7.1 Discussion

The ability to specify soft goals enables the BPMS to support the user in navigating the process. However, the optimization formulation relies on more than just

Fig. 2 The credit approval part of the process



soft goal specifications; it relies on accurate probability estimates of future event sequences. Hence, the conceptual model where an optimization problem formulation is extracted from the process specification is appropriate, but it is important to note that the size of its look-ahead depends sharply on the quality of event sequence probability distributions available. In routine cases these can be extracted from a process log, but most often compromise is necessary in the form of a reduced look-ahead in the optimization step.

8 Towards a new process design methodology

The intent of this paper is to introduce soft goals as a feature n BPMS, but the intent is also to engender a revision of the method by which business processes are designed. This paper cannot fit a comprehensive revised process design methodology, but an outline is in order – what remains is deferred to future work.

Rigid constraints are the most common source of workarounds in BPMS, and lack of support is the most common source of confusion. Soft goals promise to alleviate these shortcomings, but they must be used gratuitously – also in situations where one would be inclined to use a hard constraint.

As mentioned soft goals will only be successful when they are pervasive in the process life cycle – not just in the BPMS. The normal process life cycle is to (1) analyze the business (2) design/adapt the process, (3) use it, (4) look for improvements, (5) repeat.

The analyst should aim to describe and classify the dependencies between soft goals and activity, e.g. in the style of (Yu and Mylopoulos, 1994). The designer can then incorporate these into the process specification, and a BPMS that understands such a process specification can now provide better user guidance. In other words: the intentional business information must be retained throughout the lifecycle of the process specification.

In the evolution of the process, we strongly recommend to start out with the most liberal process specification possible: this means (a) prefer soft goals to hard constraints and (b) prefer parallelism to sequence. A similar idea to (b) has been mentioned by Bider (Bider, 2005). We have not left out the hard constraints altogether, instead we have recast them as soft goals; that is, we say "our goal is to violate the constraints as rarely and as insignificantly as possible" instead of "this constraint can never be violated". Whereas hard constraints are likely to lead to frustration, soft constraints allow us to capture intentional information in a non-obstructive way. A process specification stands a better chance of success if the designer errs on the side of flexibility rather than control. To add constraints to a flexible process one simply needs a documented reason. To remove constraints from a rigid process one must identify workarounds and diagnose them, and one can never be sure to have found them all. Situations that require new workarounds may occur at inconvenient times in the future. Removing an unnecessary constraint can take time causing delays to the process instances in question

This is not to say that identifying soft goals is trivial. There may be widely disparate opinions among the stakeholders of a process, they may express goals imprecisely, or there may simply be too many goals to realistically capture. While this is very likely to be the case, it may in fact increase the usefulness of a system that is able to capture soft goals: it forces users to reflect upon and discuss soft goals, and – if applied optimally – enables the discussion of soft goals to happen continuously.

Upon explicitly representing soft goals in the system, users may find that the goals have become more narrow in their scope that the original formulation (to give an extreme example the expressed soft goal “ensure worker comfort” becomes the very narrow soft goal “schedule job A by round robin when possible” in the system). This demonstrates that soft goals – exactly like the processes they are part of – should evolve continuously and with help of their users. More research is needed to identify best methodological practices in this area.

9 Related research

The topic of offering more flexibility in BPMS has recently been treated extensively (Schmidt, 2005; Bider, 2005; Soffer, 2005; Borch and Stefansen, 2006). Soffer introduced a taxonomy between hard and soft constraints, and applied these labels to general categories of constraints such as environmental constraints, sharing dependency constraints, goal reachability constraints etc. (Soffer, 2005)

In a recent paper Regev, Bider, and Wegmann presented an approach to flexibility using invariants (Regev, Bider, Wegmann, 2007). Conceptually a process run is seen as a trajectory through a state space and time, and process specification then becomes a matter of limiting the trajectory while finding allowable movement in space-time that brings the process closer to the preset goals. In terms of that view of the world, out work here can be seen as systematically ranking sets of states and trajectories as *preferred*, *neutral*, *discouraged*, etc. for each soft goal.

In other words, the work presented in this paper should be seen as an augmentation of existing systems, that is independent of – but compatible with – the view of the world presented in (Regev, Bider, Wegmann, 2007). The languages used in the BPMS can largely be left in place, and flexibility is gained through extra soft goal annotations.

A different approach to flexibility is that of Sadiq, Orlowska and Sadiq who allow *pockets of flexibility* inside typical strict workflows (Sadiq, Orlowska, Sadiq, 2004; Sadiq, Sadiq, Orlowska, 2001). A pocket of flexibility allows the users to build part of the workflow ad hoc inside the pocket, which is connected to the rest of the workflow as if it were an activity or a subworkflow. What goes inside such a “build” area is governed by a simple constraint language. While the motivations driving this work are virtually identical to the ones listed here, the outcomes are remarkably different. Sadiq, Orlowska and Sadiq propose to make the workflow *regionally* amenable to *construction at runtime*. Again, we would argue that while this certainly does provide more flexibility, it does not address the fundamental problem that not all sequences are equally good; a ranking (e.g. *preferred*, *neutral*, *discouraged*) expressed as a soft constraint is needed for each soft goal.

It is common to distinguish between changes to a process template (type-level) and changes to a running process (instance-level) (Regev et al., 2005). This paper is chiefly concerned with the type-level. We have not proposed a faster way to change process templates, nor have we proposed a method for changing running processes; we proposed adding soft goals and soft constraints to the process specification language, so that we will need to change both templates and instances less frequently and converge faster to a useful, but flexible, process description.

Very recently a similar approach was proposed leveraging an optimization engine for solving allocation issues on-the-fly (Hamadi and Quimper, 2006). The approach presented here however is more comprehensive in that it is ubiquitous in the process and its description and not constrained to allocation.

Last, it is important to point out the large body of work on *soft systems*. The application of soft constraints and soft goals in BPMS can be seen as an instance of the general *Soft Systems Methodology* (SSM), a comprehensive methodology dealing with situations where people have diverging – or even contradicting – views on problem definitions and goals. For more on SSM, see e.g. (Checkland and Scholes, 1990).

10 Conclusion and future work

Current systems and theories allow designers to specify binary classifications of processes. We have argued that this is insufficient, and that it inevitably leads to rigid (dictatorial) processes or support-less (anarchistic) processes. We have also argued that the flexibility given to the user by other approaches, while useful, lacks the explicit representation of how any future course of action affects the set of soft goals. Clever specifications can only partially remedy this, because the classification remains binary given the current technology.

Soft goals seek to alleviate these problems. They allow designers to specify goals instead of rigid constraints, and allow users to immediately see the gravity of their violations. We believe that this allows systems to be very flexible while retaining the support found in systems with an emphasis on control.

To deliver on these promises, soft goals must be made explicit and be understood by the BPMS. We have outlined how this is done, and proposed the necessary changes to the design methodology employed by process designers.

Processes evolve in response to ideas and to changes in the world. It presupposed here that process designers will write goals themselves based on managerial targets. In the future this is more likely to be a symbiosis between managerial targets and inspiration fed back to the designers via process mining. To keep the goals up to date the process logs should be constantly mined for violations, new constraints etc. so that they can be incorporated into the process repository.

We have postponed a treatment of data-dependencies. Data-dependencies to a large extent (but not entirely) are hard constraints, but they can be used to derive sequential constraints. If a system derives such dependencies itself, the designer has one design burden less, and any changes in the data will immediately be reflected in the appropriate constraints and nowhere else. On the other hand data-dependencies also enforce sequential constraint and any flexibility afforded to the user, must ensure that data-dependencies cannot be violated. Formalizing this remains a challenge for future work.

On a more general note it remains to be investigated how dependency-driven and state-based processes interact with the ideas presented in the paper.

References

Bernstein, A. (2000), 'How can Cooperate Work Tools Support Dynamic Group Processes? Bridging the Specificity Frontier', *Proceedings of CSCW'00*, *Philadelphia.*

Bider, I. (2005) 'Masking flexibility behind rigidity: Notes on how much flexibility people are willing to cope with', *Extended abstract of keynote talk, BPMDS'05. Proceedings of the CAiSE'05 workshops, Vol. 1, FEUP, Porto, Portugal.*

Borch, S. E. and Stefansen, C., 'On Controlled Flexibility', *BPMDS'05. Proceedings of the CAiSE'05 workshops, Vol. 1, FEUP, Porto, Portugal.*

Chapell, D. (2005), 'Introducing Microsoft Windows Workflow Foundation: An Early Look', at http://msdn2.microsoft.com/en-us/library/aa480215.aspx71-92768-6, accessed May 1, 2007.

Checkland, P.B. and Scholes, J., (1990) 'Soft Systems Methodology in Action', ISBN 0-471-92768-6, *John Wiley & Sons Ltd.*

Hamadi, Y. and Quimper, C.-G. (2006), 'The Smart Workflow Foundation', *Microsoft Research, MSR-TR-2006-114*, November.

Hammer, M. (1990), 'Reengineering Work: Don't Automate, Obliterate', *Harvard Business Review, July/August.*

Kobayashi, M. (2005), 'Work coordination, workflow, and workarounds in a medical context'*, CHI'05 extended abstracts on Human Factors in Computing Systems, Portland.*

Regev, G., Bider, I., and Wegmann, A. (2007), 'Defining Business Process Flexibility with the Help of Invariants', *Softw. Process Improve. Pract., Vol. 12, 65-79, Wiley Interscience*

Regev G., Soffer P., and Schmidt R., (2006), Taxonomy of Flexibility in Business Processes, Workshop on Business Process Modeling, Design and Support (BPMDS'06), Proceedings of CAiSE'06 Workshops, p. 90-93.

Robinson, M. (1993), 'Design for unanticipated use', *Proceedings of the European Conference on Computer-supported Cooperative Work, Milan.*

Sadiq, S. and Orlowska, A. and Sadiq, W. (2005), 'Specification and validation of process constraints for flexible workflows', *Information Systems*, 30 pp 349–378

Sadiq, S. and Sadiq, W. and Orlowska, A. (2001), 'Pockets of Flexibility in Workflow Specification', *ER '01: Proceedings of the 20th International Conference on Conceptual Modeling*, pp 513–526, Springer

Schmidt, R. (2005), 'Flexible Support of Inter-Organizational Business Processes Using Web Services', *BPMDS'05. Proceedings of the CAiSE'05 workshops, Vol. 1, FEUP, Porto, Portugal.*

Schmidt, K. and Simone, C. (1996): 'Coordination Mechanisms: Towards a Conceptual Foundation of CSCW Systems Design', *Computer Supported Cooperative Work, Vol. 5, no 2/3.*

Soffer, P. (2005) 'On the Notion of Flexibility in Business Processes', *Proceedings of the CAiSE'05 workshops, Vol. 1, FEUP, Porto, Portugal.*

Suchman, L.A. (1987), 'Plans and Situated Actions: The Problems of Human-

Machine Communication', *Cambridge University Press*, December.

Thatte, S. and others (2003), 'Business Process Execution Language for Web Services, Version 1.1', at http://www.ibm.com/developerworks/library/specification/ws-bpel/, accessed May 1, 2007.

Yu, E. and Mylopoulos, J. (1994) 'Using Goals, Rules, and Methods to Support Reasoning in Business Process Reengineering', *Proc. 27th Hawai'i Int'l Conf. System Sciences, Maui, Hawai'i, Vol. 4, pp 234-235*, January

A Work Allocation Language with Soft Constraints

Christian Stefansen [a] Sriram Rajamani [b] Parameswaran Seshan [c]

[a] *University of Copenhagen, Copenhagen, Denmark*

[b] *Microsoft Research India, Bangalore, India*

[c] *SETLabs, Infosys Technologies Ltd., Bangalore, India*

Abstract

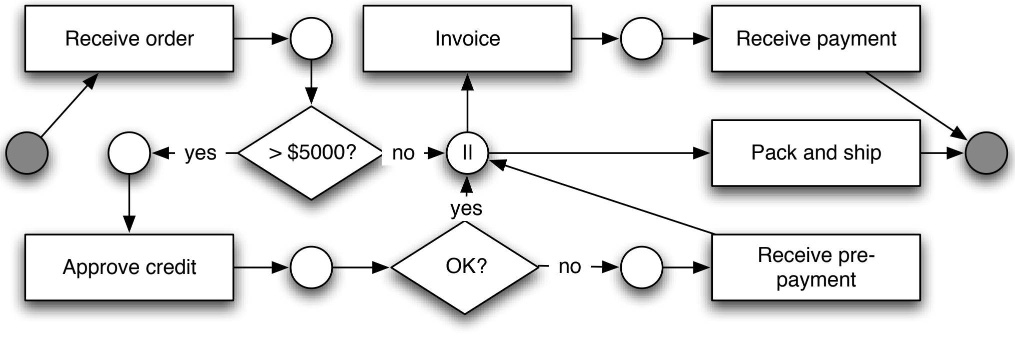
Today's business process orchestration languages such as *WS-BPEL* and *BPML* have high-level constructs for specifying flow of control and data, but facilities for allocating tasks to humans are largely missing. This paper presents SoftAlloc, a work allocation language with *soft constraints*, and explains the requirements and trade-offs that led to its design, in particular, what soft constraints are, and how they enable business process definitions to capture allocation rules, best practices, and organizational goals without rendering the business processes too strict.

SoftAlloc combines with virtually any business process language and any conceivable legacy system, while guaranteeing polynomial performance. We present the design, the formal definition, and an evaluation of SoftAlloc.

1 Introduction

Computer-orchestrated business processes are increasingly playing a direct role in how companies organize work and now commonly involve both human resources and computer resources. A widely accepted architecture is to have a business process orchestration engine that orchestrates the process by calling upon human resources and computing resources to perform the actual tasks. Tasks can often be handled by many different resources. This is particularly often the case for human tasks. Therefore the process orchestration engine must decide in negotiation with the human resources who of the eligible resources ultimately carries out the task. Assigning a task to a resource is referred to as *allocation*.

Fig. 1 Order process (Petri net-style notation; || is a shorthand for parallel split)



Allocating tasks to humans is inherently more complex than allocating to computer resources: in addition to having multiple, changing attributes that decide what they can, may or should do, humans have personal preferences and they may choose to override the allocation rules at runtime, e.g. because they possess domain knowledge that is not captured in the system or because they make conscious, reflected violations to speed up processing in cases where the process description focuses too narrowly on compliance.

Consider the example workflow given in Figure 1. In this workflow we might wish to say that the task *Receive payment* should be carried out by someone with the role *Finance*. We can imagine specifying this by attaching a rule to the task saying:

```
     role = "Finance"
```

We can then compose small building blocks of rules into larger rules. Now imagine that we want *Receive payment* to be carried out by the user who did *Invoice* to retain familiarity. We would then add

```
     role = "Finance"
 and user = whoDid ("Invoice")
```

as a rule to *Receive payment.*

But something is awry here. While we would certainly *prefer* the task *Receive payment* to be done by the same person who did the invoicing, this is only a *preference*—certainly not a strict rule that should be allowed to stand in the way of timely workflow completion if the designated person happens to be busy or temporarily absent. We have just committed one of the most common mistakes in workflow specification: we have promoted a *soft goal* to a strict rule and thereby created an inflexible system!

Alternatively, we might have removed the rule and only have said `role = "Finance"`, but that would have left out useful intentional information about our best practice. So just specifying the minimal number of constraints is not attractive either.

This example illustrates that allocation constraints can represent a wide spectrum of specifications: some rules are best kept strict (e.g. *Expense approval* must be done by a *Manager*) while other rules are simply guidelines (e.g. *Replenish printer cartridges* should be allocated on a rotation basis (*round robin*)). The latter allocation strategy represents an organizational *soft goal*, which might have been "rotate tedious tasks between qualified workers to achieve a sense of fairness and variation and keep workers happy". This is undeniably a laudable goal, but if the company is experiencing peak load, this goal must temporarily yield to more mission-critical business goals (e.g. response time *vis-à-vis* our customers). Therefore, it cannot be written as a hard constraint, but leaving it out entirely renders the system unable to suggest the preferred person.

Going back to our example what we probably mean could be written as

```
  role = "Finance"
prefer [10] user = whoDid ("Invoice")
```

which states that we require a finance person to handle the activity under all circumstances, but we prefer the person who did the invoicing in that process. The number 10 represents a *score* to indicate how strong a preference this is. This becomes more interesting, when more preferences are in play. Consider the following rule for allocating the *Credit approval* step in the workflow:

```
  role = "Manager" or role = "Finance"
prefer [10] role = "Manager"
       [-queueSize()]
```

The rule states that either *Manager* or *Finance* should handle the *Credit approval* task. A manager is preferred, but the number of items in the manager's queue is deducted from the preference level; i.e. someone with a short queue is preferred. Indeed, if all managers have more than 10 items in their work queues, someone from *Finance* will be preferred in the interest of time. This shows how soft constraints in conjunction with hard constraints can be used to express soft goals and performance heuristics in allocation.

Such soft constraints can be expressed and combined on many levels. We can imagine that the company has a general policy to prefer the shortest queue and seek to distribute activities by rotation (*round robin*):

```
prefer [-queueSize()]
       [rndRobin(1,10)]
```

This combines with a process-level policy to strongly prefer a user at the same location as the process:

```
prefer [15] user.location = proc.location
```

If we consider *Receive payment* with the same rule as before, there are now three combined rules in effect:

```
  role = "Finance"
prefer [10] user = whoDid ("Invoice")
-----------------------------------------
prefer [-queueSize()]
       [rndRobin(1,10)]
-----------------------------------------
prefer [15] user.location = proc.location
```

We can now try to run the allocation on *Receive payment* with the combination of the three levels of rules active. Suppose we have already compiled the necessary information about each user (where `rndRobin` is a number [1..10] indicating how long time it has been since that user did *Receive payment*):

User	`role`	`location`	`queue`	`rndRobin`
Ashok	Finance	India	3	9
Diego	Support	Germany	7	5
John	Finance	US	29	2
Julia	Finance	Germany	12	3
Uno	Finance	Germany	9	8

Assuming that `proc.location = "Germany"` and `whoDid("Invoice") = "Julia"` the work allocation language will produce the following suggestion sorted by score:

User	Score	Reasons
Julia	16	[15] proc.location [10] whoDid (”Invoice”) [3] rndRobin [-12] queueSize
Uno	14	[15] proc.location [8] rndRobin [-9] queueSize()
Ashok	6	[9] rndRobin [-3] queueSize()
John	-27	[2] rndRobin [-29] queueSize()

Notice, that Diego does not appear in the table as he does not satisfy the hard constraint on `role`. Julia will be able to see the contents of the score table when she receives the activity in her queue. This enables her to make an informed choice if for some reason she decides to re-allocate the task. In other words, soft constraints allow users to see the intensional information that has a bearing on the allocation, while they can retain their possibility to re-allocate based on any extra runtime knowledge they may have. By showing its reasoning the language supports the users in making an informed decision.

1.1 Contributions

This paper defines a declarative work allocation language, SoftAlloc, that (a) supports soft constraints, (b) plugs into any business process language in an aspect-like manner, (c) can be used with a wide range of platforms/legacy systems, (d) allows rules on many organizational levels to be combined, and (e) runs fast enough to allocate and re-allocate at runtime. Let us examine the reasons and implications:

Supports hard constraints *and* soft constraints The introduction and use of soft constraints in a work allocation language alleviates serious issues with workflows that are either too rigid or too lenient (see Stefansen and Borch [1] for a detailed discussion).

Plugs into any business process language A work allocation language definition that is orthogonal to the business process language, meaning that it can combine with any of the popular languages currently being used. The language takes on an aspect-like flavor in the sense that it is possible to disable the allocation aspect and still run the business process (which then would simply allow anybody to carry out any task). Similarly, an approximate allocation can be done without being hosted in a process language (e.g. for development and testing).

Plugs into any platform/legacy system Carries the same advantages.

Permits composition of rules in several scopes Rules can be attached to activities, to scopes, to entire workflows, to people and to the entire system as policies. The compositional design of the language ensures that it is meaningful and well-defined to compose rules on several different levels.

Runs in P-time (provided that the user-defined functions do so.) The allocation can always be solved in polynomial time, making it feasible to rerun the allocation as often as needed during execution, but it is yet able to express a wide range of heuristics to improve workflow performance and human resource utilization.

The language has been implemented, tested, and evaluated in Infosys' PEAS platform, and it is slated for inclusion in the PEAS platform with a GUI that is being developed. We would like to stress early that the paper does not contribute an interface to be used directly by business process analysts, designers or users. For that purpose the syntax is too low-level, and it therefore remains future work to design a user interface for the language. Also, in our allocation example from the introduction, the user, Julia, should see the reasons for the allocation in human language and not as snippets of code. This part is also future work.

2 Background and requirements

This project was done in collaboration with Infosys Technologies Ltd., India. Infosys uses *WS-BPEL* [2] and *BPML*, and typical applications include sales support, banking and *business process outsourcing* (BPO) projects.

While dedicated systems and professional workflow systems have support for allocation [3], popular process languages, including *WS-BPEL* [2] and *BPML*, do not. Since such languages are now taking over the role of dedicated workflow products to achieve a broad SOA integration, there is an increasing need for allocation facilities. Some products and initiatives address this (e.g. *BPEL 4 People* [4]), but they remain in an embryonic stage.

When humans are involved in workflows, deviations occur. Today, where a process engine orchestrates the process rather than merely guides it, deviations cannot be ignored. Instead the process engine must be built with flexibility in mind. This is particularly pertinent when we consider allocation. As mentioned in the introduction current process engines cannot support this flexibility in a satisfactory way, because even the most well-designed workflow will be too lenient in some cases and still too strict in other cases. It has previously been argued [1] that workflow systems (BPM engines) should be able to capture intentional information (e.g., about soft goals) so that the engine can *guide* its users and provide advice while still allowing (and thus enabling the monitoring of) some violations of best practice. The work presented here is based on those observations, but we are by no means at the road's end: allocation is only one small part of a workflow description where the idea of soft constraints could be leveraged.

In short, there is a need to augment existing business process languages with allocation support. There is also a need that those allocation rules support soft constraints so as to be able to guide the users whilst avoiding over-specification that leads to rigid systems.

2.1 Expressiveness requirements

To map out the allocation constraints that the language must be able to express we (1) compiled a large selection of actual business processes, and (2) cross-checked the requirements with the existing research on resource allocation patterns [3,5,6]. The most important scenarios were:

Allocate to creator The entire process is allocated to the resource who instantiated it.

Multiple roles An activity is allocated to several roles. A resource having at least one of the roles mentioned is required.

Soft constraints The ability to specify that some rules are violable, while others are not. The former are referred to as soft constraints, and they carry an integer score. This score is used to give a preference between the resources that satisfy the hard constraints.

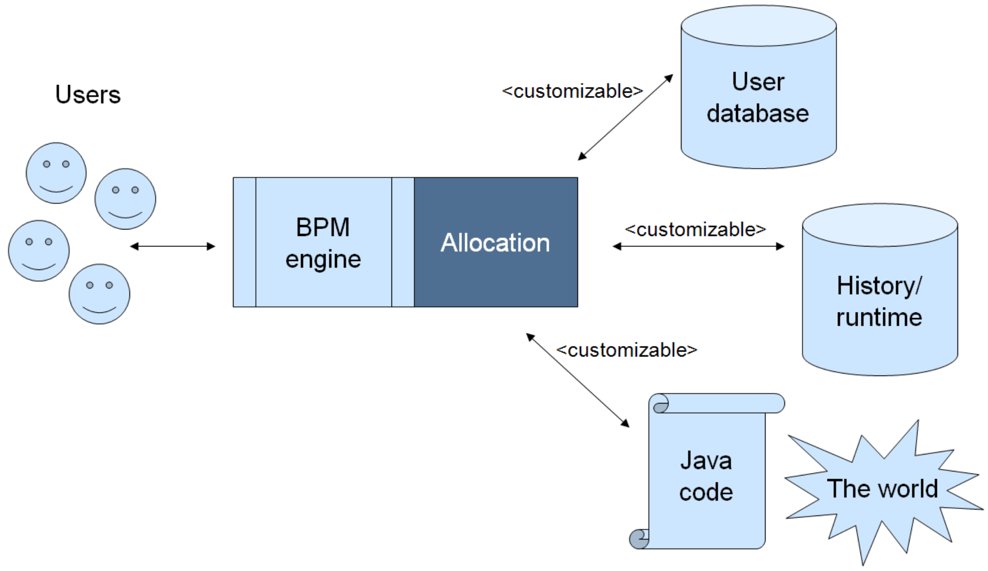
Multiple resources An activity is allocated to several resources, all of whom need to collaborate on the activity.

Scalar properties For some attributes there are several levels, e.g., 1 (Novice), 2 (Medium), 3 (Expert). In such cases it must be possible to specify (a) a minimum skill level for some needed skill (e.g. Java level 2) or (b) a soft preference for assigning the task to the resource with the highest skill level.

Overflowing When the queues of some people become too long, an overflow group can be used for allocation. This amounts to soft constraints based on queue length/expected average waiting time.

Minimize makespan This is just one of many goals one can try to optimize. The soft constraints we have focused on here intentionally *cannot* perform full optimization because performance has been prioritized—instead we can express

Fig. 2 The architectural context of work allocation



local greedy heuristics that in practice perform quite well.

One may justifiedly wonder how runtime negotiation is handled, i.e. the situation where activities are not given directly to a user, but allocated through some protocol of offer/accept to a group of users. Or activities are escalated and re-allocated. Or activities are allocated to a queue associated with many users rather than to one particular user. These issues are certainly as important as the work allocation rules we have discussed, but since they are established protocols that rarely, if ever, change, they are better handled as an integral part of the BPM engine itself. The architecture is such that the BPM engine handles all runtime requests and then queries the allocation engine for a suggestion when allocation/re-allocation is needed due to some change in the state of the system (see Figure 2).

Also notice that the process engine, not the language, decides how queues are handled, e.g. if resources can autonomously override task priorities, skip jobs in the queue, etc. If we wish to assign activities to a queue shared by many users, this is handled by inserting that queue's name in the list of potential users that is given to the allocation engine. The allocation engine will then obliviously consider the queue as if it were a user.

To do its job the allocation language needs to draw in information about user properties, queue sizes, workload, history, etc. For this, general external function calls are the best solution. While the names may be uniform, their implementation will depend on the concrete deployment. History is a good example of this. Building in an entire history reporting language would make the allocation language significantly more complex. Also, it would not add much value because a preferable reporting language (e.g., SQL) is usually already available.

Patterns In addition to the allocation scenarios above, a large number of "resource patterns" have been proposed [3]. Since we handle runtime negotiation in the process engine, only a few of these patterns apply: user data patterns (*Direct allocation*, *Role-based allocation*, *Capability-based allocation*, *Organizational allocation*), history-based patterns (*Separation of duties*, *Retain familiar*, *History-based allocation*), scope patterns (*Case handling*), and runtime information patterns (*Deferred allocation*).

Time of allocation When an activity can be started we say that it is enabled. E.g., in the process expression `a; (b||c); d` (`a`, then `b` and `c` in any order, then `d`) the activity `a` is the only enabled activity. After `a` has been completed, both `b` and `c` are enabled. Allocation can happen at three conceptually different times:

(1) Early allocation happens before the task is enabled and helps the system predict/guess bottlenecks early and avoid them through different allocation. The system may even speculatively allocate beyond branches (even if outcome of the branch is not yet known), which of course works best in conjunction with the ability to re-allocate.

(2) Allocation on enablement allocates an activity exactly when it becomes enabled. This is simple to model, to implement, and to understand for humans interacting with the system.

(3) Late allocation allocates as late as possible, i.e. if all suitable resources still have items in the queue, the process execution engine might as well postpone allocation until someone's queue is (almost) empty, and only then allocate the activity. Late allocation can be problematic for human workers in terms of planning because they only see a partial list of the tasks that potentially need their attention.

The language designed here can support all three modes or a combination thereof, because the allocation rules can simply be re-evaluated to obtain a new allocation when desired by the allocation engine. In other words this is orthogonal to the language design.

2.2 Contextual requirements

Some important requirements can be derived immediately from the business context and the architectural context:

- The language must work in tandem with any process model (e.g. *WS-BPEL*, *BPMN*, *EPCs*, Petri net-based models) that the BPMS uses.
- Because the environment (user database, log formats, legacy interfaces) is different in every organization, the functions that read these must be externalizable.
- Oftentimes organizations have soft goals on many different levels. To allocate an activity, we may have to examine general policies, department soft goals, process best practices, scope rules, and the allocation rules of the activity. This means that the work allocation language should allow several levels of rules to combine easily and seamlessly, so that, e.g., general policies can be changed without mandating a change to all subordinate processes.
- It is essential that the allocation engine can recompute a new suggestion immediately when changes occur. This precludes *NP*-hard computation so the language must be able to express only polynomial-time performance heuristics. For this reason we do not allow freely interdependent allocation rules; an allocation rule for a task can only depend on the allocation of other tasks that are guaranteed to have completed (no cyclic allocation dependencies). For the same reason time scheduling (i.e. specifying that an activity must be done at a particular time) is not possible, though scheduling constraints can be introduced in a limited way as external functions.

3 Language definition

This section describes the syntax, type system, and semantics of SoftAlloc. We assume that the workflow is specified using *WS-BPEL*, *BPML* or some comparable business process notation. The only assumption we make is that a workflow contains a finite set of tasks, each of which is uniquely named.

Syntax The syntax of the allocation language SoftAlloc is given in Figure 3. A ***rule*** has a `pick` prefix that specifies the number of users needed to perform the task, and two ***clauses***. The `where` clause specifies a *hard constraint*, which is a set of users from which allocations can be made to this task, and the `prefer` clause specifies a list, where each element is a pair of a *score* and a *soft constraint.*

Fig. 3 Work allocation language core grammar (top), implementation-specific operators and functions (middle), and implementation-specific rewrites (bottom)

```
rule        ::=  (pick const)? clauses

clauses     ::=  (where exp)?
                 (prefer pair*)?

exp         ::=  exp op exp | unary-op exp | value |
                 function (exp, ..., exp) |
                 user | task | process

pair        ::=  [ exp ] exp
```

```
op          ::=  + | - | * | / | < | > | = | <> |
                 and | or

unary-op    ::=  + | - | not

function    ::=  procStr | userStr |
                 queueSize | rndRobin |
                 whoDid | ...
```

```
        process.id ⟶ procStr(process,"id")
           user.id ⟶ userStr(user,"id")
              role ⟶ userStr(user,"role")
        exp <> exp ⟶ not exp = exp
        whoDid(id) ⟶ whoDid(process,id)
       queueSize() ⟶ queueSize(user)
  rndRobin(id,id') ⟶ rndRobin(user,id,id')
```

Implicitly, they define a scoring function to order the set of users given by the `where` clause. The only allowed variables are in the language are `user` and `task` and `process`. The language is parameterized over the set of operators, functions, and types; in Section 3.1 we explain this in more detail. This means that a large number of extensions to the language can be made without re-working the core semantics. Thus, the properties described here are valid for any valid set of operators, functions, and types that one might wish to use, as we describe later.

The `pick` prefix can be omitted. The clauses `where`, and `prefer` can also be omitted. In such case the default values are respectively: `pick 1`, `where true`, `prefer` (empty list). Additionally, if none of the keywords `pick`, `where` or `prefer` are present, the given expression is assumed to be the `where` clause, i.e. ***exp*** without any top-level keywords means `pick 1 where` ***exp*** with no `prefer` clause.

For completeness Figure 3 also shows an example of implementation-specific operators and functions and syntactic rewrites (for syntactic sugar). This example extension corresponds to the functions we have used in the examples throughout the paper. In Section 3.2 we explain how this example extension is defined. Interested readers are encouraged to consult the technical report [7] for more details on how the example extension interacts with the core language.

Type system An allocation rule is well-typed if and only if after syntactic rewrites it has a derivation in the type system given in Figure 4. The language has only static types, and types are inferred (no explicit type declarations).

Types are given by $\tau \ ::= \textbf{int} \mid \textbf{string} \mid \textbf{bool} \mid \beta$, where β represents any additional types that are given given as parameters as part of an implementation-specific extension of the language.

Extending and instantiating the language In addition to the types β, any parameters to instantiate the language come with (1) a domain $\mathcal{V}$ of values, which contains the set $\mathcal{U}$ of users, the set $\mathcal{T}$ of tasks, and the set $\mathcal{P}$ of process instance identifiers, (2) a domain $\mathcal{F}$ of function and operator symbols, (3) a function $\Theta : \mathcal{V} \rightarrow \tau$ that maps values to types, and (4) a function $\Omega : \mathcal{F} \rightharpoonup \tau^* \rightharpoonup \tau$ that maps functions and input types to output types. Intuitively, the type system requires hard constraints and soft constraints to be of type **bool**, and scores to be of type **int**.

Fig. 4 Type system

$$\frac{\Theta(\mathit{value}) = \tau}{\mathit{value} : \tau} \qquad \frac{}{\texttt{user} : \textbf{string}}$$

$$\frac{}{\texttt{task} : \textbf{string}} \qquad \frac{}{\texttt{process} : \textbf{string}}$$

$$\frac{\mathit{exp}_1 : \tau_1 \qquad \mathit{exp}_2 : \tau_2 \qquad \Omega(\mathit{op}, [\tau_1, \tau_2]) = \tau}{\mathit{exp}_1 \;\; \mathit{op} \;\; \mathit{exp}_2 : \tau}$$

$$\frac{\mathit{exp}_1 : \tau_1 \qquad \Omega(\mathit{op}, [\tau_1]) = \tau}{\mathit{op} \;\; \mathit{exp}_1 : \tau}$$

$$\frac{\mathit{exp}_1 : \tau_1, \ldots, \mathit{exp}_n : \tau_n \qquad \tau = \Omega(\mathit{function}, [\tau_1, \ldots, \tau_n])}{\mathit{function}(\mathit{exp}_1, \ldots, \mathit{exp}_n) : \tau}$$

$$\frac{\begin{array}{c} \mathit{exp} : \textbf{bool} \qquad \mathit{pexp}_1 : \textbf{int}, \ldots, \mathit{pexp}_m : \textbf{int} \\ n : \textbf{int} \qquad \mathit{cexp}_1 : \textbf{bool}, \ldots, \mathit{cexp}_m : \textbf{bool} \end{array}}{\texttt{pick } n \texttt{ where } \mathit{exp} \texttt{ prefer [}\mathit{pexp}_1\texttt{] } \mathit{cexp}_1 \cdots \texttt{[}\mathit{pexp}_m\texttt{] } \mathit{cexp}_m}$$

3.1 Denotational semantics

When the process execution engine calls upon the work allocation engine to allocate a user to a task, it sends the allocation rules that apply to that task along with a set of users, the name of the task to be allocated, and the current process instance id. In response the work allocation engine sends back a subset of those users as well as their scores (and leaves it up to the BPM engine to handle runtime negotiations). In other words the allocation rules implicitly map a set of users, a task, and a process instance id to a set of potential users to whom the task can be allocated. This set of users is specified by the *hard constraints*. Within the set, the users are ordered by a scoring, which is decided by the *soft constraints* (the `prefer` clause).

This is reflected by the denotational semantics, which maps a clause and a triple of a set of users, a task, and a process instance id, to a function from users to integer values (denoted by $\mathcal{V}_{|int}$). The semantic functions have the following signatures:

$$\begin{array}{l} \Gamma[\![\cdot]\!] : \boldsymbol{clauses} \rightarrow (2^{\mathcal{U}} \times \mathcal{T} \times \mathcal{P}) \rightarrow \mathcal{U} \rightharpoonup \mathcal{V}_{|int} \\ \Delta[\![\cdot]\!] : \boldsymbol{exp} \rightarrow (\mathcal{U} \times \mathcal{T} \times \mathcal{P}) \rightharpoonup \mathcal{V} \\ \Pi[\![\cdot]\!] : \boldsymbol{pair*} \rightarrow (\mathcal{U} \times \mathcal{T} \times \mathcal{P}) \rightharpoonup \mathcal{V}_{|int} \end{array}$$

Fig. 5 Denotational semantics

$$\Gamma[\![\texttt{where}\ exp\ \texttt{prefer [}pexp_1\texttt{]}\ cexp_1 \cdots \texttt{[}pexp_m\texttt{]}\ cexp_m]\!](U,t,p) =$$

$$\{(u,s) \mid u \in U \wedge \Delta[\![exp]\!](u,t,p) = \texttt{true}$$

$$\wedge\ \Pi[\![\texttt{[}pexp_1\texttt{]}\ cexp_1 \cdots \texttt{[}pexp_m\texttt{]}\ cexp_m]\!](u,t,p) = s\}$$

$$\Pi[\![\texttt{[}pexp_1\texttt{]}\ cexp_1 \cdots \texttt{[}pexp_m\texttt{]}\ cexp_m]\!] =$$

$$\lambda(u,t,p)\ .\ \mathbf{cond}(\Delta[\![cexp_1]\!](u,t,p), \Delta[\![pexp_1]\!](u,t,p), \texttt{0}) + \cdots$$

$$+\,\mathbf{cond}(\Delta[\![cexp_m]\!](u,t,p), \Delta[\![pexp_m]\!](u,t,p), \texttt{0})$$

$$\begin{aligned}
\Delta[\![value]\!] &= \lambda(u,t,p)\ .\ value \\
\Delta[\![\texttt{user}]\!] &= \lambda(u,t,p)\ .\ u \\
\Delta[\![\texttt{task}]\!] &= \lambda(u,t,p)\ .\ t \\
\Delta[\![\texttt{process}]\!] &= \lambda(u,t,p)\ .\ p
\end{aligned}$$

$$\begin{aligned}
\Delta[\![exp_1\ \ op\ \ exp_2]\!] &= \lambda(u,t,p)\ .\ \mathbf{eval}(op, \\
&\qquad \Delta[\![exp_1]\!](u,t,p), \Delta[\![exp_2]\!](u,t,p)) \\
\Delta[\![op\ \ exp]\!] &= \lambda(u,t,p)\ .\ \mathbf{eval}(op, \Delta[\![exp]\!](u,t,p)) \\
\Delta[\![func(exp_1, \ldots, exp_n)]\!] &= \lambda(u,t,p)\ .\ \mathbf{eval}(func, \\
&\qquad \Delta[\![exp_1]\!](u,t,p), \ldots, \Delta[\![exp_n]\!](u,t,p))
\end{aligned}$$

The definitions of these denotations are shown in Figure 5. The definitions use two auxiliary functions, **eval** and **cond**. The function **eval** is used to evaluate external functions that are defined in the parameters given to instantiate the language. Formally, **eval** : $\mathcal{F} \rightharpoonup \mathcal{V}^* \to \mathcal{V}$ maps a function or operator symbol, and a list of values to a return value. In an instantiation of the language, every defined function is required to be a total map from a list of values of appropriate argument types to a value of the appropriate return type. The function **cond** : $(\mathcal{V} \times \mathcal{V} \times \mathcal{V}) \to \mathcal{V}$ returns its second argument if the first argument is `true`, and its third argument otherwise.

Given a well-typed rule $r = \texttt{pick}\ n\ c$, with a user database U, a current task t, and a process instance id p, the denotation $\Gamma[\![c]\!](U,t,p)$ yields a partial map from users to their scores. Using this partial map, the allocation engine chooses n users for the task t in process instance p heuristically based on their scores (and other criteria, such as availability of the users).

Definition 1 (Well-typed evaluation) *We say that the functions* Ω *and* **eval** *are* consistent *if and only if for all function symbols* f *and values* $[v_1 : \tau_1, \ldots, v_n : \tau_n]$

$$\Omega(\mathit{f}, [\tau_1, \ldots, \tau_n]) = \tau \Rightarrow \mathbf{eval}(\mathit{f}, [v_1, \ldots, v_n]) = (v : \tau)$$

We also say that **eval** *is* well-typed *with respect to* Ω*.*

Theorem 2 (Progress and preservation) *The denotational function is defined on all well-typed rules. More formally, let* $\mathcal{U}$*,* $\mathcal{T}$*,* $\mathcal{P}$*,* $\mathcal{V}$*,* $\mathcal{F}$*,* Ω*,* Θ*, and* **eval** *be given such that* Ω *and* **eval** *are consistent. Let* $r = \texttt{pick}\ n\ c$ *be a well-typed rule. Then* $\Gamma[\![c]\!]$ *is well-defined, and furthermore* $\forall\ U \subseteq \mathcal{U}, u \in U, t \in \mathcal{T}, p \in \mathcal{P} : \Gamma[\![c]\!](U,t,p)(u) = s \Rightarrow s : \mathit{int}$*.*

Corollary 3 *The denotation* $\Gamma[\![c]\!](U,t,p)$ *yields a (partial) function* $\mathcal{U} \rightharpoonup \mathcal{V}_{|\mathit{int}}$*. Thus, every user has a unique score or no score.*

Proof. Follows from the definition of $\Gamma[\![\cdot]\!]$ and the fact that $\Delta[\![\cdot]\!]$ and $\Pi[\![\cdot]\!]$ are functions. ■

Last, we consider how composition of rules is defined:

Definition 4 (Composition of rules) *Given two rules* `where` *exp* `prefer` *pairs and* `where` *exp′* `prefer` *pairs′ their composition is defined as follows:*

$$\begin{aligned}&\Gamma[\![\texttt{where}\ \mathit{exp}\ \texttt{prefer}\ \mathit{pairs}]\!](U,t,p)\\ \oplus\ &\Gamma[\![\texttt{where}\ \mathit{exp'}\ \texttt{prefer}\ \mathit{pairs'}]\!](U,t,p) =\\ &\qquad\{(u,s) \mid u \in U \wedge \Delta[\![\mathit{exp}]\!](u,t,p) = \texttt{true}\\ &\qquad\qquad\wedge \Delta[\![\mathit{exp'}]\!](u,t,p) = \texttt{true}\\ &\qquad\qquad\wedge \Pi[\![\mathit{pairs}\ \mathit{pairs'}]\!](u,t,p) = s\}\end{aligned}$$

For almost any concrete instantiation of the language with a sane `and` operator, this definition will be equivalent to

$$\Gamma[\![\texttt{where}\ \mathit{exp}\ \texttt{and}\ \mathit{exp'}\ \texttt{prefer}\ \mathit{pairs}\ \mathit{pairs'}]\!](U,t,p).$$

3.2 Concrete prototype language

As promised we now return to the instantiations and extensions to the core language that are necessary to express the examples throughout the paper. Specifically, these are the ones shown in the middle and bottom parts of Figure 3. We now consider these extensions in more detail.

Recall, that the necessary parts for extension are (1) additional types β (if any), (2) function/operator names/symbols $\mathcal{F}$, types Ω, and semantics **eval**, (3) syntactic rewrites for the parser (if any), and (4) operator fixity (infix, prefix, postfix) and precedence for the parser. The necessary parts for instantiation are then (5) the domain of values $\mathcal{V}$ and (6) the domain of users $\mathcal{U}$, tasks $\mathcal{T}$, and process instance identifiers $\mathcal{P}$. This may seem somewhat daunting at first, but most of the domains needed for instantiation are likely to always be defined as standard types such as string or int. Furthermore, the functions typically turn out to be simple call-out wrappers equipped with a type check function.

Additional types Oftentimes we wish to write a query that checks for membership of a set. E.g., if a user has many different roles, we may wish to stipulate that the role `developer` is a member of that set of roles. To this end we introduce sets of the three primitive types. The complete type grammar becomes:

$$\tau ::= \mathit{int} \mid \mathit{string} \mid \mathit{bool} \mid \beta$$
$$\beta ::= \mathit{set\ of\ int} \mid \mathit{set\ of\ string} \mid \mathit{set\ of\ bool}$$

We identify the set of users $\mathcal{U}$ with strings, $\mathcal{T}$ with strings, and $\mathcal{P}$ with ints. The set of values $\mathcal{V}$ thus becomes the union over these and their powersets; Θ is the obvious map from values to types. (Notice, though, that it is not possible to write set literals in the language, although sets exist as (intermediate) values.)

Syntactic rewrites Figure 3 shows the rewrites that were introduced to sugar the language. Consider the example rule `role = "Finance"` from Section 1. It is syntactically rewritten as `userStr(user,"role") = "Finance"`, where `userStr` is a binary function that maps pairs of strings to sets of strings, and is well defined for all users. As another example, the rule `role = "Finance" and user = whoDid("Invoice")` is interpreted as

```
userStr(user,"role") = "Finance" and user = whoDid(process,"Invoice")
```

where `userStr` is a binary function as before, and `whoDid` is a binary function that takes an int and a string, and returns a string[1].

The functions are all outside the core grammar; we consider these next.

Function/operator symbols Referring again to Figure 3 the concrete language contains the infix operators `or`, `and`, `=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/` (integer division), `%` (modulo), and the unary prefix operators `+`, `-`, `not`. These all have the usual meanings and precedence. (Modulo is defined as in Java.) Given the functions listed, we have

$$\begin{aligned}\mathcal{F} = \{ &\ \texttt{or}, \texttt{and}, \texttt{not}, \texttt{=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%},\\ &\ \texttt{procStr}, \texttt{userStr}, \texttt{queueSize}, \texttt{rndRobin},\\ &\ \texttt{whoDid}\}.\end{aligned}$$

The type map Ω is shown in Figure 6.

Notice that `=` is overloaded for comparison between all three primitive types as well as for set membership test (but not for set to set comparison!)

Semantics of operators/functions: eval In the prototype the functions are handled by external dispatch to methods written outside the language (concretely, in Java)[2]. The semantics of all operators is completely routine as they map to the same operators in Java. One exception is `=`, which will mean either $\in$, $=$ or $\ni$ depending on the type of its arguments. Notice how the type system permits this kind of overloading.

The function `userStr` can be given either `"location"` or `"role"` as its second argument and then returns the set of current locations/roles for the user given as the first argument (or the empty set in case the user does not exist or some other

[1] Many of the rewrites simply provide shorthands to allow the omission of typical arguments like `user`, `task`, and `process`. In the Infosys prototype these were provided by convention to all external function calls to avoid having a large number of trivial syntactic rewrites.

[2] In the Infosys implementation the binary and unary operators have all been added to the language core because (1) they will generally be needed in any specialization and (2) the language performs better.

Fig. 6 The function type map Ω for the concrete example language

$$
\begin{aligned}
\texttt{or} &\mapsto \{[\mathit{bool}, \mathit{bool}] \mapsto \mathit{bool}\}\\
\texttt{and} &\mapsto \{[\mathit{bool}, \mathit{bool}] \mapsto \mathit{bool}\}\\
\texttt{not} &\mapsto \{[\mathit{bool}] \mapsto \mathit{bool}\}\\
\texttt{=} &\mapsto \{[\mathit{int}, \mathit{int}] \mapsto \mathit{bool},\\
&\quad\ [\mathit{set\ of\ int}, \mathit{int}] \mapsto \mathit{bool},\\
&\quad\ [\mathit{int}, \mathit{set\ of\ int}] \mapsto \mathit{bool},\\
&\quad\ [\mathit{string}, \mathit{string}] \mapsto \mathit{bool},\\
&\quad\ [\mathit{set\ of\ string}, \mathit{string}] \mapsto \mathit{bool},\\
&\quad\ [\mathit{string}, \mathit{set\ of\ string}] \mapsto \mathit{bool},\\
&\quad\ [\mathit{bool}, \mathit{bool}] \mapsto \mathit{bool},\\
&\quad\ [\mathit{set\ of\ bool}, \mathit{bool}] \mapsto \mathit{bool},\\
&\quad\ [\mathit{bool}, \mathit{set\ of\ bool}] \mapsto \mathit{bool}\}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{procStr} &\mapsto \{[\mathit{int}, \mathit{string}] \mapsto \mathit{set\ of\ string}\}\\
\texttt{userStr} &\mapsto \{[\mathit{string}, \mathit{string}]\\
&\qquad\qquad \mapsto \mathit{set\ of\ string}\}\\
\texttt{queueSize} &\mapsto \{[\mathit{string}] \mapsto \mathit{int}\}\\
\texttt{rndRobin} &\mapsto \{[\mathit{string}, \mathit{int}, \mathit{int}] \mapsto \mathit{int}\}\\
\texttt{whoDid} &\mapsto \{[\mathit{int}, \mathit{string}] \mapsto \mathit{string}\}\\
\texttt{<} &\mapsto \{[\mathit{int}, \mathit{int}] \mapsto \mathit{bool}\}\\
\texttt{>} &\mapsto \{[\mathit{int}, \mathit{int}] \mapsto \mathit{bool}\}\\
\texttt{<=} &\mapsto \{[\mathit{int}, \mathit{int}] \mapsto \mathit{bool}\}\\
\texttt{>=} &\mapsto \{[\mathit{int}, \mathit{int}] \mapsto \mathit{bool}\}\\
\texttt{+} &\mapsto \{[\mathit{int}, \mathit{int}] \mapsto \mathit{int}\}\\
\texttt{-} &\mapsto \{[\mathit{int}, \mathit{int}] \mapsto \mathit{int}\}\\
\texttt{*} &\mapsto \{[\mathit{int}, \mathit{int}] \mapsto \mathit{int}\}\\
\texttt{/} &\mapsto \{[\mathit{int}, \mathit{int}] \mapsto \mathit{int}\}\\
\texttt{\%} &\mapsto \{[\mathit{int}, \mathit{int}] \mapsto \mathit{int}\}
\end{aligned}
$$

string was given as the second argument. Analogously for `procStr`. The remaining functions perform as described in the examples in Section 1.

3.3 Miscellaneous issues

Process engine context When the process engine calls upon the allocation engine, it passes on a context consisting of a set of users, a task, and a process instance id (and, of course, the rule expressions). Thus far we have written out the context as (U, t, p). Since many of our external functions need part of this context they are subject to syntactic rewrites that add parts of the context as arguments. A solution that would be simpler and more robust to future changes would be for the process engine to pass the entire relevant context along as one parameter (e.g. a map) and have the allocation engine pass the entire context on to all external functions. Incidentally, the number of external function calls could be reduced by giving a function the set of users U rather than each user u one at a time.

Allocation as an optimization problem It is enlightening to consider work allocation as an optimization problem. In fact, work allocation (also knows as rostering or scheduling) is one of the most common problems addressed by integer programming techniques in practice.

Our allocation problem—through deliberate language design to ensure this—can be solved in polynomial time (i.e. the function $\Gamma[\![\cdot]\!]$ evaluates in polynomial time), provided that evaluating *exp* can be done in polynomial time. The naive, brute force algorithm simply evaluates each of the users: first it checks if they satisfy the hard constraint, and if they do, how many points they get from the soft constraints. It then leaves it to the process execution engine to pick—presumably—the top n satisfying users. The correctness of this algorithm is due to the monotonicity: the computation of each user's score is independent of who else is considered for allocation.

Assume we wish to evaluate the following rule in the context of task t in process p:

$$\texttt{pick } n \texttt{ where } exp$$

$$\texttt{prefer [}pexp_1\texttt{] } cexp_1 \cdots \texttt{[}pexp_m\texttt{] } cexp_m$$

Given a user database U with users $\{u_1, ..., u_{|U|}\}$ and the appropriate definitions of the functions called in the expressions, an allocation is a vector $a_1, ..., a_{|U|} \in \{0,1\}^{|U|}$ where $a_j = 1$ iff the task is allocated to user u_j. We then seek to

$$\begin{aligned} &\textit{maximize} && \textstyle\sum_{j\in[1..|U|]} \Big(a_j \cdot \sum_{i\in[1..m]} [\mathit{cexp}_i(u_j,t,p)] \cdot \mathit{pexp}_i(u_j,t,p)\Big) \\ &\textit{subject to} && \forall j \in [1..|U|] : a_j = 1 \implies \mathit{exp}(u_j,t,p) \\ &\textit{and} && \textstyle\sum_{j\in[1..|U|]} a_j = n. \end{aligned}$$

When we write $[\mathit{cexp}_i(u_j,t,p)]$ in the objective function $[\cdot]$ is the *indicator function* which maps to 1 when its argument evaluates to true and to 0 otherwise. The objective function can therefore be read as "for all users, if the task is allocated to them, add to the sum of total points, the points they receive for each preference that is true for them". Strictly speaking, we should have enclosed *exp*, *cexp*, and *pexp* in the evaluation function $\Delta[\![\cdot]\!]$, but this has been omitted to keep the formulation free of clutter.

4 Implementation in PEAS

The work allocation language has been implemented and tested with the PEAS platform, Infosys' proprietary platform for BPM that includes business process modeling, deployment, execution, and monitoring. The design of the system is shown in Figure 7. Notice that this architecture is slightly simplified relative to the language outlined in Section 3.1; specifically, it supports only allocation rules on task-level, but not on a scope, process or as a policy. Therefore the keyword `task` loses its use and as a result the constructor on the class `LateAllocation` takes only the allocation rule expression (as an `InputStream`, userdata, and external dispatch functions (but not the name of the task to which a resource must be allocated)). For historic reasons the process instance id is not passed through the interface, but available in the context. Notice the class `Type` which corresponds to the type production β in the semantics. The classes `UserScoreMap` and `UserValueMap` are used to return the map of user names to (score, reasons) pairs that the process engine use to decide ultimately, which user is suggested for the task.

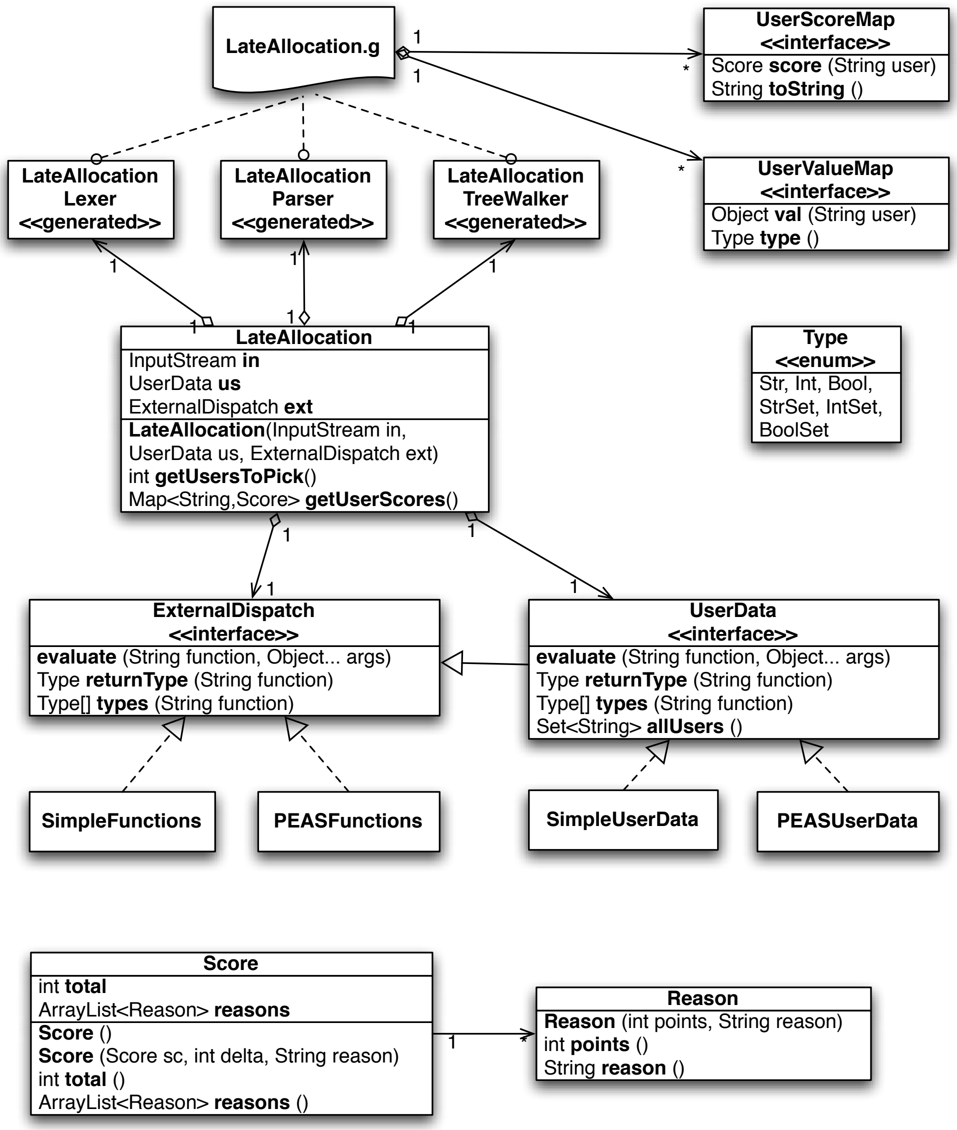
5 Evaluation

The prototype has been tested and evaluated, and it is slated for inclusion in Infosys' BPM platform, PEAS, with a GUI that is being developed. Several prototype workflows have been tested with the allocation language, including a CRM (Customer Relations Management) workflow, a sales process, an approval/review process, and a bank transaction process. To prove tangible business benefits based on more than just anecdotal evidence, an empirical study is necessary; this remains future work.

To evaluate the domain fit we applied two methods: (1) a patterns-based analysis and (2) qualitative discussions sessions with industry experts, who confirmed that the language fits well with the needs for human-intentisive workflows (e.g., insurance claims, call centers).

A patterns-based analysis in the style of Russell *et al.* [3] yields a first approximation of the domain fit of an allocation language. The analysis proceeds by evaluating if the language supports, partially supports or does not support each of the patterns in a suite of 43 patterns collected from industry and research. Table 1 shows a patterns-based evaluation of our work allocation language juxtaposed with the outcome of related patterns-based analyses. Note that only the 11 patterns that

Fig. 7 Class diagram including lexer and parser generator files



specify who is *ultimately* allowed to perform a task are included (cf. our discussion in Section 2.1) plus 3 new patterns. As mentioned earlier the language did not set out to specify runtime negotiation rules because these are more elegantly handled by the process engine. To be deemed to have a good domain fit and satisfy the requirements, SoftAlloc must fully support (+) or partially support (+/-) most of the desired patterns, and it must support soft constraints.

The patterns-based analysis suggests that SoftAlloc does indeed have a good

Table 1 A patterns-based comparison. Our evaluation is in the columns and rows in *italics*; the rest of the table is the result of related research by others [3,8,9,10]. + indicates full support, x/- indicates partial support (e.g. through coding a bit), and - indicates no support. B4P/WS-HT is short for *BPEL 4 People/WS-HumanTask*.

	Research			Open src.			Commercial				
Pattern	***SoftAlloc***	**B4P/WS-HT**	**newYAWL**	**JBoss jBPM**	**OpenWFE**	**Enhydra Shark**	**Staffware**	**Websphere**	**FLOWer**	**COSA**	**iPlanet**
Direct allocation	+	+	+	+	-	+	+	+	+	+	+
Role-based	+	+	+	-	+	+	+	+	+	+	+
Separat. of duties	+/-	+	+	-	-	-	-	+	+	+/-	+
Case handling	+/-	-	-	-	-	-	-	-	+	-	-
Retain familar	+/-	+	+	+	-	-	-	+	+	+	+
Capability-based	+	-	+	-	-	-	-	-	+	+	+
History-based	+/-	+/-	+	-	-	-	-	-	-	+/-	+
Organizational	+	+/-	+	-	-	-	+/-	-	+	+	+
Round robin	+/-	-	+	-	-	-	-	-	-	+/-	+/-
Random	+	-	+	-	-	-	-	-	-	+	+/-
Shortest queue	+/-	+/-	+	-	-	-	-	-	-	+	+/-
Mult. resources	+	?	?	?	?	?	?	?	?	?	?
Soft constraints	+	?	?	?	?	?	?	?	?	?	?
Combine patterns	+	?	?	?	?	?	?	?	?	?	?

domain fit. In fact, SoftAlloc supports or partially supports the entire set of patterns inside the scope of the work. In many of the other products +/- means that the pattern requires coding/workarounds *every time* it is used; in SoftAlloc +/- means that an interface to the existing systems of the company has to be coded *once* to get full support.

Some limitations apply to this methodology: whereas our language fares quite well in the comparison, the patterns-based analysis itself inadequately captures the expressive power of our language. The last three rows (*Multiple resources*, *Soft constraints*, and *Combine patterns*) illustrate this. E.g., soft constraints are not a single pattern, but an idea that could easily be expanded to comprise an entire suite of patterns in its own right. The patterns-based analysis falls short here because the inventors of the analysis did not anticipate soft constraints. This clearly shows the limitation of approaches with some finite set of patterns chosen without any particular guiding principle other that what is available in current products. Similarly, the patterns-based analysis does not mention if patterns can be combined and if so with what constraints. If two systems both allow, say, role-based allocation and organizational allocation, but only one of the systems permits the designer to combine these two patterns in one allocation rule, a patterns-based analysis will not reveal this critical difference of expressiveness. This clearly demonstrates that while patterns may be useful as a checklist to avoid forgetting important functionality, they should never be applied blindly or used as the only measure of expressiveness. In conclusion a patterns-based evaluation is somewhat unsuitable.

6 Related work

In the introduction we stated that we aim for a work allocation language that "(a) supports soft constraints, (b) plugs into any business process language in an aspect-like manner, (c) can be used with a wide range of platforms/legacy systems, (d) allows rules on many organizational levels to be combined, and (e) runs fast enough to allocate and re-allocate at runtime". In examining the related work it is useful to keep these key features in mind:

Allocation does have some similarities with the scheduling done in a multiprocessor system, but in the case of work allocation the processors (i.e. the human resources) are rarely homogeneous. In this way the problem area seems closer related to Grid scheduling, and indeed some interesting ideas have cropped up in the Grid scheduling field (makespan reduction [11], algorithms/heuristics [12]). However, these either do not support soft constraints (a) or require full opmization (e).

We have already discussed the work on resource patterns by Russell *et al.* [3]. Senkul and Toruslu [6] have suggested a simple allocation language, which in their proposal is translated into the constraint language *Oz* and solved by a constraint solver. The approach solves the entire workflow (or the entire set of running workflows) in one go, and it is therefore not clear how the approach would accommodate runtime flexibility whilst being scalable—in other words it does not have feature (e). Another approach uses *defeasible logic* [13], and *BPEL 4 People* has attempted to make *WS-BPEL* better suited for human-intensive workflows [4], but these do not address soft constraints (a) and composition (d).

There exists a variety of business rule languages (e.g. OMG's SVBR[14]) and thus it would seem obvious to take one of these as a starting point as it naturally plugs in many contexts (b,c) and composes well (d). As we examined these languages we found that there was no direct facility for soft constraints (a). It could be mimicked in some cases, but at the cost of significant extra complexity.

Ways of solving *NP*-hard scheduling problems have been studied for decades in operations research, where the survey by Ernst is a good place to start [15]. Although such approaches have superior expressiveness in modeling soft constraints (a), they are markedly more complicated to use, even for programmers, and they cannot generally guarantee optimal solutions in *P*-time (e).

Interdisciplinary papers have proposed auction/game theory-based [16] and *AI*-based [17] approaches to work allocation. These algorithms behave as dynamic systems and thus usually adapt very well to shifts in the supply and demand of tasks. However, they do not consider soft constraints directly (a).

7 Conclusion and future work

The patterns-based analysis shows that SoftAlloc can express all patterns for which it was designed and all examples that were deemed necessary. Industry expert interviews further supported the conclusion that SoftAlloc has a good fit to human-intensive workflows. The use of soft constraints has proven very beneficial, and as intended the language integrates with any system we have seen so far. The language was not built for direct use by business process designers/analysts. Instead a GUI (both for users and allocation rule designers) is being developed in the production setting where the language is to be used. The GUI coupled with the textual form presented in this paper will allow both programmers and domain specialists to use the language in their preferred way and support conflict detection.

Another improvement would be to devise ways of avoiding the fixed numerical preference levels. This is particularly relevant when several levels of rules are combined.

Based on the discussion of resource patterns it would be interesting to develop better measures of expressiveness to benchmark languages.

More benefits are yet to be reaped: by capturing work allocation rules directly, performance simulation can be used to identify bottlenecks, estimate capacity requirements, and suggest what resources to add. This is an important improvement over previous systems, where the lack of integration made performance analysis a non-routine job requiring specialized skills. In systems where work allocation rules are captured in a general-purpose language, workflow performance simulation is often infeasible.

Another promising idea is to leverage runtime statistics to improve the allocation optimization. Runtime statistics could include average completion time probability of task type per agent, inferred probability of delay, etc. All these statistics will result in more soft constraints that the scheduler can use in conjunction with the user-specified ones.

Acknowledgements

The authors would like to thank Shriram Krishnamurthi and the reviewers for some very useful comments.

References

[1] C. Stefansen, S. E. Borch, Using soft constraints to guide users in flexible business process management systems, International Journal of Business Process Integration and Management.

[2] OASIS Open, Inc., WS-BPEL 2.0 Committee Specification (May 2006).
URL `http://www.oasis-open.org/committees/download.php/18714/wsbpel-specification-draft-May17.htm`

[3] N. Russell, A. H. M. ter Hofstede, D. Edmond, W. M. P. van der Aalst, Workflow resource patterns, Tech. rep., Eindhoven University of Technology (2005).
URL `http://is.tm.tue.nl/research/patterns/download/Resource%20Patterns%20BETA%20TR.pdf`

[4] M. Kloppmann, D. Koenig, F. Leymann, BPEL 4 People, Tech. rep., IBM and SAP (July 2005).
URL `ftp://www6.software.ibm.com/software/developer/library/ws-bpel4people.pdf`

[5] Y. Hamadi, C.-G. Quimper, The smart workflow foundation, Tech. Rep. MSR-TR-2006-114, Microsoft Research (November 2006).

URL http://research.microsoft.com/research/pubs/view.aspx?type=Technical%20Report\&id=1160

[6] P. Senkul, I. H. Toroslu, An architecture for workflow scheduling under resource allocation constraints, Information Systems 30 (5) (2005) 399–422.
URL http://dx.doi.org/10.1016/j.is.2004.03.003

[7] C. Stefansen, S. Rajamani, P. Seshan, A work allocation language with soft constraints, Tech. rep., SETLabs, Infosys Technologies Ltd., Bangalore, India (March 2008).

[8] N. Russell, A. H. ter Hofstede, W. M. van der Aalst, D. Edmond, newYAWL: Achieving comprehensive patterns support in workflow for the control-flow, data and resource perspectives, Tech. Rep. BPM-07-05, Eindhoven University of Technology (2007).
URL http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2007/BPM-07-05.pdf

[9] N. Russell, W. M. P. van der Aalst, Work distribution and resource management in bpel4people: Capabilities and opportunities, in: Z. Bellahsene, M. Léonard (Eds.), CAiSE, Vol. 5074 of Lecture Notes in Computer Science, Springer, 2008, pp. 94–108.

[10] P. Wohed, B. Andersson, A. H. ter Hofstede, N. R. W. M. van der Aalst, Patterns-based evaluation of open source BPM systems: The cases of jBPM, OpenWFE, and Enhydra Shark, Tech. Rep. BPM-07-12, Eindhoven University of Technology (2007).
URL http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2007/BPM-07-12.pdf

[11] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, K. Kennedy, Task scheduling strategies for workflow-based applications in grids, in: CCGRID '05, IEEE Computer Society, 2005, pp. 759–767.
URL http://ieeexplore.ieee.org/iel5/10428/33122/01558639.pdf?tp=\&isnumber=\&arnumber=1558639

[12] S. Zhang, Y. Wu, N. Gu, Adaptive grid workflow scheduling algorithm, in: Grid and Cooperative Computing Workshops, Vol. LNCS 3252, 2004, pp. 140–147.
URL http://www.springerlink.com/content/fkaqt7lhmu9amnhx/

[13] G. Governatori, A. Rotolo, S. Sadiq, A model of dynamic resource allocation in workflow systems, in: ADC '04, Australian Computer Society, Inc., 2004, pp. 197–206.
URL http://portal.acm.org/citation.cfm?id=1012316

[14] OMG, Semantics of business vocabulary and business rules (SBVR), Tech. rep., Object Management Group, second SBVR Interim Specification, http://www.omg.org/docs/dtc/06-08-05.pdf (Sep. 2006).
URL http://www.omg.org/docs/dtc/06-08-05.pdf

[15] A. T. Ernst, H. Jiang, M. Krishnamoorthy, D. Sier, Staff scheduling and rostering: A review of applications, methods and models, Eur. J. Op. Res. 153.
URL http://www.sciencedirect.com/science/article/B6VCT-48BKYWR-4/2/4b065270117e5421bb1299161a78803d

[16] E. Koutsoupias, Selfish task allocation, Bulletin of EATCS 81 (2003) 79–88.
URL http://cgi.di.uoa.gr/~elias/publications/paper-kou03.pdf

[17] S. Abdallah, V. Lesser, Learning the task allocation game, in: AAMAS '06, ACM Press, New York, 2006, pp. 850–857.
URL http://dis.cs.umass.edu/~shario/papers/aamas06.pdf

Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications

John Field [a,*], Maria-Cristina Marinescu [a], Christian Stefansen [b]

[a] *IBM Research, T.J.Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532*

[b] *DIKU, University of Copenhagen, Universitetsparken 1, København Ø, Denmark*

Abstract

Our aim is to define the kernel of a simple and uniform programming model—the *reactor* model—which can serve as a foundation for building and evolving internet-scale programs. Such programs are characterized by collections of loosely-coupled distributed components that are assembled on the fly to produce a composite application. A reactor consists of two principal components: mutable state, in the form of a fixed collection of *relations*, and code, in the form of a fixed collection of *rules* in the style of Datalog. A reactor's code is executed in response to an external *stimulus*, which takes the form of an attempted update to the reactor's state. As in classical process calculi, the reactor model accommodates collections of distributed, concurrently executing processes. However, unlike classical process calculi, our observable behaviors are sequences of states, rather than sequences of messages. Similarly, the interface to a reactor is simply its state, rather than a collection of message channels, ports, or methods. One novel feature of our model is the ability to compose behaviors both synchronously and asynchronously. Also, our use of Datalog-style rules allows aspect-like composition of separately-specified functional concerns in a natural way.

Key words: reactors, datalog, actors, synchronous programming, asynchronous programming, synchronization, distributed programming
PACS:

[*] Corresponding author

Preprint submitted to Elsevier *20 September 2008*

1 Introduction

In modern web applications, the traditional boundaries between browser-side presentation logic, server-side "business" logic, and logic for persistent data access and query are rapidly blurring. This is particularly true for so-called web mash-ups, which bring a variety of data sources and presentation components together in a browser, often using asynchronous ("AJAX") logic. Such applications must currently be programmed using an agglomeration of data access languages, server-side programming models, and client-side scripting models; as a consequence, programs have to be entirely rewritten or significantly updated to be shifted between tiers. The large variety of languages involved also means that components do not compose well without painful amounts of scaffolding. Our ultimate goal is thus to design a uniform programming language for web applications, other human-driven distributed applications, and distributed business processes or web services which can express application logic and user interaction using the same basic programming constructs. Ideally, such a language should also simplify composition, evolution, and maintenance of distributed applications. In this paper, we define a kernel programming model which is intended to address these issues and serve as a foundation for future work on programming languages for Internet applications.

The reactor model is a synthesis and extension of key ideas from three linguistic foundations: synchronous languages [1–3], Datalog [4], and the actor model [5]. From Datalog, we get an expressive, declarative, and readily composable language for data query. From synchronous languages, we get a well-defined notion of "event" and atomic event handling. From actors, we get a simple model for dynamic creation of processes and asynchronous process interaction.

A reactor consists of two principal components: mutable state, in the form of a fixed collection of *relations*, and code, in the form of a fixed collection of *rules* in the style of Datalog [4]. A reactor's code is executed in response to an external *stimulus*, which takes the form of an attempted update to the reactor's pre-reaction state (*pre-state*). When a stimulus occurs, the reactor's rules are applied concurrently and atomically in a *reaction* to yield a *response* state, which becomes the initial state for the next reaction. In addition to determining the response state, evaluation of rules in a reaction may spawn new reactors, or generate new stimuli for the currently executing reactor or for other reactors. Importantly, newly-generated stimuli are processed *asynchronously*, in later reactions. However, we provide a simple mechanism to allow collections of reactors to react together as a unit when appropriate, thus providing a form of distributed atomic transaction.

As in classical process calculi, e.g., pi [6], the reactor model accommodates collections of distributed, concurrently executing processes. However, unlike classical process calculi, our observable behaviors are sequences of *states*, rather than sequences of *messages*. Similarly, the interface to a reactor is simply its state ("REST" style [7]), rather than a collection of message channels, ports, or methods. We accommodate information hiding by preventing certain relations in a reactor's state from being externally accessible, and by allowing the public relations to serve as abstractions of more detailed private state (as in database views). A significant advantage of using data as the interface to a component, and Datalog as a basis for defining program logic, is that the combination is highly "declarative": it allows separately-specified state updates (written as rules) to be composed with minimal concern for control- and data-dependence or evaluation order.

Contributions. We believe that the reactor model is unique in combining the following attributes harmoniously in a single language: (1) data, rather than ports or channels as the interface to a component; (2) synchronous and asynchronous interaction in the same model, with the ability to generate processes dynamically; (3) expressive data query and transformation constructs; (4) the ability to specify constraints/assertions as a natural part of the core language; (5) distributed atomic transactions; and (6) declarative, compositional specification of functionality in an "aspect-like" manner. We believe that Internet components can be developed more productively and composed more readily when these attributes are provided in a single programming model.

This paper is an updated and extended version of [8]. This version fills in many semantic details absent from the earlier version, corrects errors, and adds additional examples.

2 Reactor basics

A reactor consists of a collection of *relations* and *rules*, which together constitute a reactive, atomic, stateful unit of distribution. The full reactor syntax is given in Fig. 1.

Consider the declaration for `OrderEntryA` in Fig. 2. `OrderEntryA` defines a class of reactors that are intended to log orders—say, for an on-line catalog application. Reactor *instances* are created dynamically, using a mechanism we will describe in Section 3.2.

```
        REACTOR ::= def reactor-type-name = { {DECL .}* }
           DECL ::= ( RELATION-DECL | RULE-DECL ) .
  RELATION-DECL ::= [public | public write | public read] [ephemeral] rel-name : ( {TYPE ,}* )
      RULE-DECL ::= HEAD-CLAUSE <- BODY
           BODY ::= {BODY-CLAUSE ,}+
    HEAD-CLAUSE ::= [var-name.]rel-name[^] ( {(_ | var-name | new) ,}* )
                  | not [var-name.]rel-name[^] ( {(_ | var-name) ,}* )
    BODY-CLAUSE ::= [not] [var-name.][^|-]rel-name ( {(_ | var-name) ,}* ) | BASIC-PREDICATE
BASIC-PREDICATE ::= EXP (< | > | <> | =) EXP
           TYPE ::= int | string | ref reactor-type-name
            EXP ::= var-name | atom-name | NUMERIC-LITERAL | STRING-LITERAL
                  | EXP (+ | - | * | / | %) EXP | self
```

Fig. 1. Reactor syntax.

```
def OrderEntryA = {
  // (id, itemid, qty)
  public orders: (int, int, int).
  // (id, itemid, qty)
  log: (int, int, int).

  log(id, itemid, qty) <-
    orders(id, itemid, qty).
}

def OrderEntryA' = {
  // ... same as OrderEntryA ...

  not log(id, itemid, qty) <-
    not orders(id, itemid, qty).
}
```

```
def OrderEntryB = {
  // ... same as OrderEntryA ...

  // orderIsNew is true if order has
  // not previously been logged
  ephemeral orderIsNew: ().
  orderIsNew() <-
    orders(id, itemid, qty),
    not -log(id, itemid, qty)
}

def OrderEntryC = {
  // .. same as OrderEntryB ...

  // delete any new order whose id is the
  // same as a previously logged id,
  // but which is not a duplicate
  not orders(id, itemid, qty) <-
    ^orders(id, itemid, qty),
    not -log(id, itemid, qty),
    -log(id, _, _).
}
```

Fig. 2. Order entry: variations.

The state of a reactor is embodied in a fixed collection of *persistent relations*. Relations are sets of $(\tau_1, ..., \tau_n)$ tuples, where each τ_i is one of the types `int`, `string`, or `ref` *reactor-type-name*. The primitive types have the usual meanings. *Reactor references*, of the form `ref` *reactor-type-name*, are described in Section 3.2. Relations are empty when a reactor is instantiated. In addition to persistent relations, whose values persist between reactions, a reactor can declare *ephemeral* relations. These relations are written and read in the same manner as persistent relations, but they are re-initialized as empty with every reaction.

The state of `OrderEntryA` consists of two persistent relations, `orders` and `log`, each of which is a collection of 3-tuples of integer values. Relation `orders` has *access annotation* `public`, which means that the contents of `orders` may be read or updated by any client. By "update", we simply mean that tuples may be added to or deleted from `orders`; no other form of update is possible. Relation `log`, lacking any access annotation, is *private*, the default, and may thus only be read or updated by the reactor that contains `log`.

A reaction begins when a reactor receives an *update bundle* from an external source. An update bundle is a partial map from the set of relations of the recipient to pairs of sets (Δ^+, Δ^-), where Δ^+ and Δ^- are sets of tuples to be added and deleted, respectively, from the target relation. For any update bundle (Δ^+, Δ^-), we require that $\Delta^+ \cap \Delta^- = \emptyset$ and $\Delta^+ \cup \Delta^- \neq \emptyset$. In the examples that follow, an update bundle will typically contain an update to a single relation, usually adding or deleting only a single tuple. However, an update bundle can in general update any of the public relations of a reactor, and add and delete arbitrary number of tuples at a time.

The state of a reactor before an update bundle is received is called its *pre-state*. A reaction begins when the contents of an update bundle is applied atomically to the pre-state of a reactor, yielding its *stimulus state*. The stimulus state of a reaction is (conceptually) a copy of each relation of the reactor with the corresponding updates from the update bundle applied. So, for example, in the case of `OrderEntryA`, if relation `orders` contained the single tuple $(0, 1234, 3)$ prior to a reaction, and a reaction is initiated by applying an update bundle with $\Delta^+ = \{(1, 5667, 2)\}$ and $\Delta^- = \emptyset$, then the stimulus value of `orders` at the beginning of the reaction will be the relation $\{(0, 1234, 3), (1, 5667, 2)\}$. We will refer to the "value of relation r in the stimulus state" and "the stimulus value of r" interchangeably.

If a reactor contains no rules, the state of its relations at the end of a reaction—its *response state*—is the same as the stimulus state, and the reaction stores the stimulus values back to the corresponding persistent relations. Hence in its simplest form, a reaction is simply a state update. However, most reactors will have one or more rules which compute a response state distinct from the reactor's stimulus state (Section 2.1).

Rule evaluation can also define sets of additions and deletions to/from the *future state* of either local relations or—via reactor references—relations of other reactors. These sets form the update bundles—one bundle per reactor instance referenced in a reaction—that initiate subsequent reactions in the same or other reactors. Update bundles thus play a role similar to messages in message-passing models of asynchronous computation.

From the point of view of an external observer, a reaction occurs *atomically*, that is, no intermediate states of the evaluation process are externally observable, and no additional update bundles may be applied to a reactor until the current reaction is complete.

Fig. 3 illustrates the life cycle of a typical collection of reactors, both from the point of view of an external observer (the top half of the figure) and internally (the bottom half of the figure schematically depicts reactor M during reaction i). The pre-state of reactor M during reaction i is labeled $\texttt{-}S_i$, its stimulus state is labeled $\texttt{\^{}}S_i$, its future state is labeled $S\texttt{\^{}}_i$, and its response state is labeled S_i. The terms pre-state, stimulus state, response state, and future state are meaningful only *relative* to a particular reaction, because one reaction's response state becomes the next reaction's pre-state, and references to a reactor's future state are used (along with the pre-state) to define the stimulus state for a subsequent reaction. The only "true" state, which persists between reactions, is the response state. An external observer therefore sees only a sequence of response states (more specifically, the response values of public relations). Each rule of a reactor can refer programmatically to relation values in all four states: it can read the pre-state of a relation (schematically depicted as `-r` in Fig. 3), the stimulus state (`^r`), and the response state (`r`); it can write the response state and the future state (`r^`).

2.1 Rule valuation

Reactor rules are written in the style of Datalog [4,9]; their syntax is given in Fig. 1. The single rule of `OrderEntryA` can be read as "ensure that `log` contains whatever tuples are in `orders`". The right-hand side, or *body* of a reactor rule consists of one or more *body clauses*. In `OrderEntryA`, there is only one body clause, a *match predicate* of the form `orders(id, itemid, qty)`. A match predicate is a pattern which binds instances of elements of tuples in the relation named by the pattern (here, `orders`) to variables (here, `id`, `itemid`, and `qty`). As usual, we use '`_`' to represent a unique, anonymous variable. Evaluation of the rule causes the body clause to be matched to *each* tuple of `orders` and binds variables to corresponding tuple elements. Since the *head clause* on the left side of the rule contains the same variables as the body clause, it ensures that `log` will contain every tuple in `orders`. Each clause in a rule may be regarded as a predicate in the logical sense, hence a logical reading of `OrderEntryA`'s rule would be "for all `id`, `itemid`, `qty`, if `orders(id, itemid, qty)` then `log(id, itemid, qty)`."

In general, a RULE-DECL can be read "for every combination of tuples that satisfy

BODY, ensure that the HEAD-CLAUSE is satisfied" (by adding or deleting tuples to the relation in HEAD-CLAUSE). The semantics of Datalog rule evaluation ensures that no change is made to any relation unless necessary to satisfy a rule, and—for our chosen semantics—that rule evaluation yields a unique fixpoint result in which all rules are satisfied. Although our rule evaluation semantics is consistent with standard Datalog semantics, we have made several significant extensions, including head negation, reference creation, and the ability to refer to remote reactor relations via reactor references.

Returning to reactor `OrderEntryA`, let us consider the case where the pre-state values of `orders` and `log` are, respectively, $\{(0, 1234, 3)\}$ and$\{(0, 1234, 3)\}$, and an update bundle has $\Delta^+ = \{(1, 5667, 2)\}$ and $\Delta^- = \emptyset$. Then the stimulus value of `orders` will be equal to $\{(0, 1234, 3), (1, 5667, 2)\}$. No rule affects the value of `orders`, so the response value of `orders` will be the same as the stimulus value. In the case of `log`, rule evaluation yields the response state $\{(0, 1234, 3), (1, 5667, 2)\}$, i.e., the least change to `log` consistent with the rule.

Now, starting with the result of the previous `OrderEntryA` reaction described above, consider the effect of applying another update bundle such that $\Delta^+ = \emptyset$ and $\Delta^- = \{(0, 1234, 3)\}$. This reaction will begin by deleting $(0, 1234, 3)$ from `orders`, yielding the stimulus state `orders` $= \{(1, 5667, 2)\}$, `log` $= \{(0, 1234, 3), (1, 5667, 2)\}$. Evaluating the rule after the deletion has no net effect on `log` (since the only remaining tuple in `orders` is already in `log`), hence we get the response state `orders` $= \{(0, 1234, 3)\}$ and `log` $= \{(0, 1234, 3), (1, 5667, 2)\}$. We thus see that the effect of this rule is to ensure that `log` contains every orderid ever seen in `orders`. If we wanted to ensure that `log` is maintained as an *exact* copy of the current value of `orders` (which would mean that it is no longer a log at all), we could add the additional *negative* rule depicted in definition `OrderEntryA'` in Fig. 2. The negative rule of `OrderEntryA'` has the effect of ensuring that if an orderid is not present in `orders`, it will also be absent from `log`; i.e., it encodes tuple *deletion*. While negation is commonly allowed in body clauses for most Datalog dialects, negation on the head of a rule is much less common (though not unheard of, see, e.g., [10]).

Reactor definitions `OrderEntryB` and `OrderEntryC` of Fig. 2 add additional rules that refine the behavior of `OrderEntryA`, using references to both pre-state and stimulus values of relations. `OrderEntryB` defines an *ephemeral* nullary relation `orderIsNew`, which functions as a boolean variable, initially false. The new rule in `OrderEntryB` sets `orderIsNew` to true (i.e., adds a nullary tuple) if `orders` contains a value not found in `log` prior to the reaction (i.e., in `log`'s pre-state value). The definition of `OrderEntryC` further refines `OrderEntryB` by causing any new order whose orderid is a duplicate of a previously logged orderid to be deleted. The new

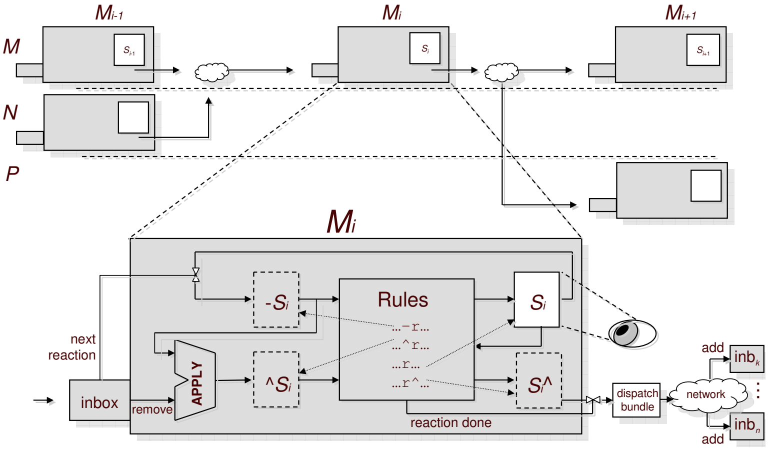


Fig. 3. Reaction schematic.

```
def OrderEntryD = {
  // (itemid, qty)
  public ephemeral write req: (int, int).
  // (key, itemid, qty)
  public log: (ref Nonce, int, int)
  // pending orders keyed to log entries
  public pending: (ref Nonce).
  // new key for this reaction
  ephemeral newKey: (ref Nonce).
```

```
  newKey(new) <- .            // 1
  log(k, item, qty) <-              // 2
      req(itemid, qty), newKey(k).
  pending(k) <- newKey(k).          // 3
}

def Nonce = {}
```

Fig. 4. Service-style order entry.

rule in `OrderEntryC` must distinguish the stimulus value of `orders`, i.e., `^orders` from its response value, i.e., `orders`, since the rule defines the response value to be something different from the stimulus value in the case where a duplicate order `id` is present.

It is important to note that the result of rule evaluation is oblivious to the order in which rules are declared. We believe this feature makes it easier to update the functionality of a reactor in an "aspect-like" fashion [11] by changing the rule set without concern for control- or data-dependencies. We see this feature demonstrated in the progression of examples depicted in Fig. 2, where rules can be mixed and matched liberally to yield updated functionality.

Consider now the very different formulation of an order entry reactor depicted in Fig. 4, `OrderEntryD`. In reactor type `OrderEntryA`, the set of all orders active in the system is publicly readable and writable by external clients. In the alternative `OrderEntryD` formulation above, the ephemeral relation `req` functions as a request “channel” or “port”: incoming collections of order entries in an update bundle are processed and cleared (deleted) immediately after the reaction, since `req` is ephemeral. Relation `req` has a `public write` access annotation, hence it is not externally readable.

Proponents of the “representational state transfer” (“REST”) style of component interaction [7] embodied by `OrderEntry` argue that it makes evolution of web applications easier by exposing state directly, rather than encapsulating it through access channels/ports/methods in a “service-oriented” style [12]. A notable feature of the reactor model is that both styles are easily accommodated.

The rules of `OrderEntryD` are straightforward, except for rule (1), which creates a new key. The expression `new` generates a new and unique instance of a *nonce*, a trivial reactor whose only function is to serve as a generator of globally-unique values. It is convenient to use such values as keys. While `Nonce`s contain no rules, in general, when a reactor is instantiated in a reaction, its rules are evaluated along with the rules of the parent reactor, as we shall see in Section 3.3. Rule (1) is an *unconditional rule*, and is an instance of the shorthand notation depicted in Fig. 6(b). It is important to note that rule (1) only generates one instance of a `Nonce` per reaction; a rule head clause containing `new` is satisfied once unique values have been generated for each instance of `new` per vector of rule variables that are instantiated by the rule.

2.2 Initialization, constants, and reaction failure

Consider the following rules:

```
r(x) <- s(x).

not r(x) <- s(x).
```

These rules are inherently contradictory, since they require that `x` be both present and absent from relation `r`. In such cases, a *conflict* results. Because rules are conditionally evaluated, conflicts cannot in general be detected statically and must be detected during rule evaluation. If such a conflict occurs, the reaction *fails*: the reactor rolls back to its pre-state and no update bundles are dispatched.

Consider the reactor definition `Cell` depicted in Fig. 5. Each instance of a `Cell` is intended to hold exactly one value. Instances of `Cell` contain two relations: a public unary relation `val` containing the publicly-accessible value of the cell, and a private

```
def Cell = {
  public val:(int).
  live:().

  // initializations
  live() <- .            // 1
  val(0) <- not -live(). // 2

  // singleton constraint: if not
  // satisfied, reaction fails; reactor
  // rolls back
  not live() <-
    val(x), val(y), x <> y. // 3
}
```

Fig. 5. Cell.

nullary (i.e., boolean) relation `live`. Recall that a reactor's relations are initially empty when the reactor is instantiated. Rules (1) and (2) together define an idiom which will allow us to initialize relations to non-empty values. First, consider rule (1). Rule (1) defines `live` to be a *constant*, since its response value evaluates to non-empty (i.e., "true") at the end of every reaction. Because of rule (1), `-live` in rule (2) is nonempty only during the first reaction in which the `Cell` is instantiated. Hence `val` will be initialized to 0 only once, in the reaction in which `Cell` is created. Thereafter, `-live` will be non-null, and the initialization will not recur, allowing `val` to be freely updated to arbitrary values.

Finally, consider rule (3) of `Cell`. The three clauses in its body collectively check to see whether `val` contains more than one value, i.e., whether it is a singleton. If not, the rule requires that its head clause (left-hand side) be satisfied, i.e., that `live` be set to empty (false). However, any such attempt is inconsistent with the assertion in rule (1) that `live` is non-empty (true), hence any attempt to update `Cell` without maintaining the singleton invariant will result in a conflict and reaction failure. We thus see that the reactor model allows "assertions" and "integrity constraints" in the style of databases to be expressed in precisely the same form as rules that express state updates. When some assertion fails, the reaction rolls back. Fig. 6(d) depicts a notational convention that will allow us to use `live` to define rules that represent assertions.

Clients of instances of reactor type `Cell` are required to ensure that they maintain its singleton constraint, e.g., by deleting the current value of the cell before adding a different value. However, if we wished, we could augment the declaration of `Cell` to make it easier for clients that wish to update its value to avoid having to delete the previous value by adding the following rule:

```
not val(x) <- -val(x), ^val(x'), x <> x'.
```

This rule is interesting because it refers to all three reactor states we have discussed thus far: pre-state, stimulus state, and response state. The body of the rule checks to see whether the stimulus value of `val` contains an item different from the pre-state. If so, the offending pre-state item is deleted from the response value of `val`. Note,

	Notation	Translation	Comments

a	r1(*exp*$_0$) <-...ri(*exp*$_i$)....	r1(x0) <- ...ri(xi)...,
		x0 = *exp*$_0$,...,
		xi = *exp*$_i$.
			Expressions exp$_i$ are instances of nonterminal EXP in Fig. 1; the xi are fresh variables.

b	*head* <- .	*head* <- 0 = 0.	
c	r(x1,...xn) := *body*.	r(x1,...xn) <- *body*. not r(x1',...xn') <- *body*, -r(x1',...xn'), x1' <> x1. ... not r(x1',...xn') <- *body*, -r(x1',...xn'), xn' <> xn.	
d	FAIL <-....	not live() <-....	Assumes the following definitions exist: live: (). live() <- .

e	*body*$_0$:{
	head$_1$ <- *body*$_1$.
	...
	head$_n$ <- *body*$_n$.
	}
		head$_1$ <- *body*$_1$, *body*$_0$.
		...
		head$_n$ <- *body*$_n$, *body*$_0$.	

f	INIT	not -live()	Assumes same definitions as (d).
g	*head*$_1$, ···, *head*$_n$ <- *body*.	*head*$_1$ <- *body*. ... *head*$_n$ <- *body*.	

Fig. 6. Notational conventions.

however, that it is still possible for the singleton constraint to fail if a client attempts to insert more than one distinct value in a single update bundle. Fig. 6(c) depicts a notational convention we will use in the sequel whenever we wish to "assign" values to singleton relations such as `val`. The example in Fig. 5 illustrates how the declarative nature of reactor rules makes it straightforward to "progressively refine" existing functionality by adding new features in a non-intrusive way.

3 Reactor composition

In the following sections we explain how reactors interact asynchronously and synchronously; we refer to this as *reactor composition.*

```
def Fibonacci = {
  // complete series thus far: each
  // elt. has form (index, value)
  public read series: (int, int).
  // must be true for reactor to run
  public write run: ().
  // holds indices in the sequence
  // less than the maximum
  ephemeral notLargest: (int).

  INIT: {
    series(1,0), series(2,1) <- .
    run() <- .
  }
```

```
  // indices in series less than max
  notLargest(n) <- series(n, _),       // 1
    series(n', _), n' > n.

  // compute next series value
  series^(n+1, x1+x2) <-
    not notLargest(n), series(n-1, x1),
    series(n, x2).                     // 2

  // halts if run set to false
  FAIL <- not -run(), not ^run().      // 3
}
```

Fig. 7. Self-Reacting Fibonacci

3.1 Asynchronous reactor composition

Up to this point, we have not explained how update bundles are generated, only how reactors react when an update bundle is applied. In this section, we show how updates are generated, and explain how this is intimately connected to asynchronous interaction.

Consider the reactor definition `Fibonacci` in Fig. 7, which computes successive values of a Fibonacci series. The relation `series` contains pairs whose first element is the ordinal position of the sequence value, and whose second element is the corresponding value of the sequence. The value of `series` is initialized using notational conventions (e) and (f) of Fig. 6. To compute the next element of the series, we need to first identify the last two elements of the series computed thus far. Universal quantification is required to determine the maximum element of a series; however, the body of a Datalog rule can only existential properties directly. To query universal properties, we typically require auxiliary relations. In `Fibonacci`, we use the ephemeral relation `notLargest` to contain all the indices of elements of `series` which are less than the maximum index. The body clause `not notLargest(n)` in rule (2) has the effect of binding `n` to the largest ordinal index currently contained in the series.

The relation in the head of rule (2) computing the next value of the Fibonacci sequence has the form `series^`. A relation name of this form refers to the *future state* of the relation. The future state effectively defines the contents of an update bundle which is processed in a subsequent reaction, *after* the current reaction ends. Hence the head of rule (2) defines an update bundle containing a single pair encoding the next value of the series; this pair will be inserted into `series` at the beginning of the next reaction. One can thus think of the future value of a relation as defining an asynchronous update or dispatching a "message". As a result, successive values of the series are separately visible to external observers as they are added to the list.

Rule (3), which uses the notation defined in Fig. 6(d), causes the reaction to fail and roll back if both the pre-state and the stimulus values of `run` are false. By using this encoding, `run` serves as a sort of switch which can be used to turn the self-reaction process on and off. Instances of `Fibonacci` can react to two distinct classes of update bundles: "internally" generated update bundles containing only new values of the series, and client-generated update bundles which only affect the value of `run`. A client cannot update `series` since `series` is not public. The `Fibonacci` reactor does not produce update bundles affecting the value of `run`, since it has no rules referring to the future value of `run`.

In general, distinct reactors operate *concurrently* and *independently.* Given this fact, it is possible for an update bundle to be generated by a client attempting to update the value of `run` while a previous reaction by the same instance is in progress. Since reactions take place atomically, we must enqueue pending client updates until the current reaction is complete. To this end, every reactor has an associated *inbox* containing a multiset of pending update bundles. When a reaction is complete, the reactor checks for a new update bundle in the inbox. If it exists, the reactor removes it and uses it to initiate a reaction. If no update bundle is present, the reactor performs no further computation until a new update arrives. Fig. 3 illustrates this process. We make no assumptions about the order in which inbox items are processed, except that they must be processed fairly.

While `Fibonacci` is designed such that update bundles can only update one relation at a time, update bundles can in general contain updates to more than one relation. Consider, e.g., a client that wishes to update ordered trees (e.g., XML trees) maintained on a server. Ordered trees can be maintained using two relations on nodes: a parent-child relation, and a next sibling relation. In this case, it is natural for an update to affect both relations.

3.2 Reactor references and multi-reactor asynchrony

Up to this point, our examples have only considered a single reactor type. Consider now the definitions for reactors `Sample`, `Sensor`, and `Nonce` depicted in Fig. 8. Reactor types `Sample` and `Sensor` encode a "classical" asynchronous request/response interaction. To enable two reactor instances to communicate, we use *reactor references* such as those stored in relation `rSensor`. Rule (1) of `Sam-`

```
def Sample = {
  // assumed  to be initialized by client
  public rSensor: (ref Sensor).
  // samples collected thus far; nonces
  // distinguish sample instances
  public log: (ref Nonce, int).
  // pulse: set to collect sample
  public write ephemeral pulse: ().
  // holds sensor response
  public write ephemeral response: (int).

  // request sample when pulse set
  s.req^(self) <- pulse(), rSensor(s). // 1

  // process response: add sample to log
  log(new, r) <- response(r).    // 2
}
```

```
def Sensor = {
  // set when sample is to be collected;
  // value is ref.to sample reactor
  public write ephemeral req: (ref Sample).
  // current sensor value
  public val:(int).

  // send resp. when client sets req
  r.response^(v) <- val(v), req(r).

  // sensor value is a singleton
  FAIL <- val(x), val(y), x <> y.
}

def Nonce = {}
```

Fig. 8. Asynchronous query/response.

`ple` has the effect of dispatching an asynchronous request for the sensor value (maintained by a `Sensor` reactor) whenever a client of `Sample` updates `pulse`. The expression `s.req^(self)` in Rule (1) contains an *indirect reference* to (the future value of) relation `rSensor`: after the reactor reference stored in relation `rSensor` is bound to variable `s`, we refer to relation `req` of the sensor instance indirectly using the expression `s.req^`. Since we refer to the future value of `s.req`, an asynchronous update bundle is dispatched to the `Sensor` instance. The update bundle contains a self-reference to the requesting `Sample` instance, which is generated by the `self` construct.

A `Sensor` instance responds to a request (in the form of an update to relation `req`) by dispatching the current value of the sensor back to the corresponding `Sample` instance. It does so by setting the `Sample`'s `response` relation via the reactor reference sent by the requester. The response is asynchronous, since `r.response^` refers to a future value. The requester processes the response from the `Sensor` instance by updating its `log` relation with the value of the response.

There are two ways of introducing references to reactors. The keyword `self` evaluates to a reference to the enclosing reactor. An expression of the form `new` in a head clause creates a new instance of the appropriate reactor type. The appropriate reactor type is inferred; e.g., if a relation *a* is declared as `a: (int, ref Acct, ref Acct)` then new instances created by the head clause `a(5, new, fromAcct)` will have the type `Acct`. Instantiation expressions may only appear in non-negated head clauses. Rule (2) of `Sample` creates instances of the trivial reactor `Nonce`. `Sample` uses nonces to distinguish multiple instances of the same sensor value.

In order to instantiate and connect `Sample` and `Sensor` instances together, another

```
def Acct = {
  public balance: (int).

  // balance is a singleton
  FAIL <- balance(x), balance(y), x <> y.

  // negative balances not allowed
  FAIL <- balance(x), x < 0.
}
```

```
def Minibank = {
  // (transfer amount, to account,
  //  from account)
  public write ephemeral transferReq:
    (int, ref Acct, ref Acct).

  to.balance(x+amt) :=
    transferReq(amt, to, _),
    to.-balance(x).

  from.balance(y-amt) :=
     transferReq(amt, _, from),
     from.-balance(y).
}
```

Fig. 9. Classic Transaction.

reactor must contain rules of the form

```
theSampler(new Sample) <- .

s.rSensor(new Sensor) <- theSampler(s).
```

A request-response cycle between `Sample` and `Sensor` instances requires three distinct reactions: the reaction in which a `Sample` client sets `pulse` (which dispatches the request to the sensor), the reaction in which the sensor responds to the request, and the reaction in which the requester updates the value of `log`.

3.3 Synchronous reactor composition

While the asynchronous form of process composition depicted in Fig. 8 is similar to that used in many process calculi and message-based distributed programming models, Fig. 9 depicts an example of *synchronous* reactor composition that is somewhat more unusual. In this example, a `MiniBank` reactor receives requests to transfer money between two `Acct` reactors. Such requests are encoded by updates to the ephemeral `transferReq` relation. As with the example in Fig. 8, we use references to wire the reactors together. However, unlike the previous example, the remote references in Fig. 9 refer to *response* values of relations, not future values. This means that a reaction initiated at a `MiniBank` reactor will *extrude* to include both of the `Acct` reactors (bound to variables `to` and `from`, respectively). This results in a composite, synchronous, atomic reaction involving *three* reactor instances. Scope extrusion is an inherently dynamic process, similar to a distributed transaction—see Section 9 for details. `MiniBank` uses the notation of Fig. 6(c) to define “assignments” to the singleton `balance` relations.

Note that the rules in Acct encode constraints on the allowable values of balance. In a composite reaction, a conflict (which is here manifested by a constraint failure) in any of the involved reactors causes the composite reaction to fail. When a composite reaction fails, all of the reactors revert to their pre-reaction states. A composite reaction is always initiated at a single reactor instance at which some asynchronously-generated update bundle is processed—in the case of the example in Fig. 9, reactor instance MiniBank.

Reactors involved in a composite reaction may separately define future values for relations of the same reactor instance. In such cases, a *single* update bundle, combining the composite future value updates for all of the involved reactors, is dispatched at the end of the composite reaction to each target reactor. In this sense, from the point of view of an external observer, a composite reaction has the same atomicity properties as a reaction involving a single reactor.

The example in Fig. 10 shows how multiple user interface components can be instantiated *dynamically* based on the current contents of an associated database. This mimics the process of building dynamic, data-driven user interface components. The basic idea of DataDisplay is that a button and an output field are generated for each item in a database. ButtonWidget and OutputWidget are user interface components representing the button and output field generated for each item in relation db. ButtonWidget's buttonListener relation contains a reference (in general, there could be more than one) to the "parent" DataDisplay reactor; rule (1) has the effect of notifying the parent whenever the button is pushed.

Rule (9) synchronizes the value of the output field to the value of the quantity currently maintained in the database. Rule (10) "wires" together corresponding button and database items such that when a button is pressed, the corresponding data item is decremented.

Ephemeral relations currDbItems and oldDbItems along with rules (4) and (5) compute the set of itemids present before and after the reaction. Using this information, rules (6) and (7) together create new widgets whenever a new item is added to db, while rule (8) deletes a widget whose corresponding item has been deleted from the database.

Note that the auxiliary ephemeral relations currDbItems and oldDbItems are not just included for clarity, they are necessary to correctly compute the set of old and new items. Consider, for example, a rule of the form

```
...<- db(i, _, _), not -db(i, _, _)
```

One might initially expect i to be bound only to newly-added items in the database.

```
def ButtonWidget = {
  // button label
  public label: (string).
  // set to true when button is pressed
  public write ephemeral pressed: ().
  // reference to DataDisplay, which will
  // be notified when button is pressed
  public write buttonListener:
    (ref DataDisplay).

  // notifies DataDisplay when button
  // is pressed, passing a self-reference
  // to indicate which button was pressed
  rD.buttonPressed(self) <-
    buttonListener(rD), pressed().      // 1
}

def OutputWidget = {
  // label (constant)
  public label: (string).
  // current value to be displayed
  public val: (string).
}

def DataDisplay = {
  // (itemid, quantity)
  public db: (int, int).
  // set when button(s) pressed
  public ephemeral write buttonPressed:
    (ref ButtonWidget).
  // (itemid, button widget)
  bWidgets: (int, ref ButtonWidget).
  // (itemid, output widget)
  oWidgets: (int, ref OutputWidget).

  // initialize widget labels
  rO.label("Inventory: ") <-
    oWidgets(_, rO).                  // 2
  rB.label("Click to decr") <-
    bWidgets(_, rB).                  // 3
```

```
  // itemids of db entries in prestate
  ephemeral oldDbItems: (int)
  oldDbItems(i) <- -db(i, _).         // 4

  // itemids of db entries in curr. state
  ephemeral currDbItems: (int).
  currDbItems(i) <- db(i, _).         // 5

  // create widgets when new item
  // added to db
  currDbItems(i), not oldDbItems(i): {
    bWidgets(i, new) <- .             // 6
    oWidgets(i, new) <- .             // 7

    // add self to notification list
    // for button widget
    rB.buttonListener(self) <-
      bWidgets(i, rB).                // 8
  }

  // delete widgets when corresp. item
  // deleted from db
  not currDbItems(i), oldDbItems(i): {
    not oWidgets(i, _) <- .           // 9
    not bWidgets(i, _) <- .           // 10
  }

  // output widget values track qty.
  // of corresponding db item
  rO.val(toString(q)) :=
    oWidgets(i, rO), db(i, q).        // 11

  // button decrements qty. of
  // corresponding db item
  db(i, q-1), not db(i, q) <-
    -db(i, q), bWidgets(i, rB),
    buttonPressed(rB).                // 12
}
```

Fig. 10. Data-Driven UI.

However, `i` will actually be bound to *every* itemid in the database, since for any itemid i in the database, there always exists *some* widgets b and o such that $\langle i, b, o\rangle$ is not in the pre-state of `db`. In other words, as written, the second body clause will always succeed, regardless of the value of i. This phenomenon can occur whenever unbound variables (such as the wildcards used above) are used in a negative body clause.

The rules which wire the database, the output field, and the button together constitute a dependency loop which would be tricky to manage if written in a traditional language. However, the normal Datalog evaluation process evaluates such recursive loops without difficulty.

4 Reactor semantics overview

In the next several sections, we formalize the operational semantics of the reactor model. The first two sections contain preliminaries: in Section 5, we review standard concepts from the semantics of Datalog with negation. In Section 6, we define basic notation for reactor concepts we will use in the rest of the paper.

The reactor model takes the point of view that each reactor instance maintains a set of relations which the instance "owns and manages" autonomously. However, as we have observed, a reactor can synchronously read or write the state of a different, *remote* reactor via reactor references. To facilitate formalizing the semantics of synchronous reactor interaction, Section 7 contains a sequence of source transformations which will serve to define a single "virtual address space" of relations. The transformed relations have the property that they may in principle contain tuples from *different* reactor instances of the same type. However, as we will see in Section 9, our operational semantics properly models the fact that each reactor independently manages only the persistent state defined by its own relation instances. To model synchronous multi-reactor reactions, the operational semantics temporarily *copies* the state of remote reactors into the state of the reactor instance which initiated the reaction. The *augmentation* transformation defined in Section 7.3 allows the initiating reactor to manage both its own state and temporarily-copied state of remote reactors in a uniform way.

Section 8 defines additional source transformations on the program resulting from the transformation of Section 7. These additional transformations eliminate head negation by defining auxiliary relations which encode addition and deletion information separately.

The transformations of Sections 7 and 8 rewrite a reactor program $\mathcal{T}$ consisting of multiple reactor definitions into a single "normal" (i.e., with body negation) Datalog program that captures the synchronous execution semantics of all of the reactor definitions in $\mathcal{T}$. This approach might seem to imply that every reactor in a distributed system needs to contain the code of all other reactors in the system. In practice, however, a reactor need only be aware of the rules of reactors which are accessible via reference types that are elements of relations declared in T, i.e., peer reactors with which it has already agreed to interact [1]

In Section 9, we complete the definition of the semantic model for reactors using a labeled transition system. This transition system carries out the following functions:

[1] It is possible to further weaken this "mutual knowledge" requirement, see Section 13.

- It orchestrates the creation and processing of *update bundles*, messages that take the form of state updates. Once paired with its recipient reactor, an update bundle can initiate a reaction; dually, reactors may generate update bundles (by defining *future values* of remote relations); the transition semantics takes care of delivering these bundles to their destination
- It models multiple concurrent, autonomous reactions and asynchronous interactions among such reactions.
- It orchestrates copying of relation values from remote reactors to the initiating reactor. The initiating reactor evaluates all the local and remote rules needed to define the state resulting from the synchronous reaction, and the state transition system ensures that the results of the synchronous reaction are copied back to remote reactors as necessary.
- It defines an *optimistic locking protocol* which ensures that the results of a synchronous composite reaction are computed atomically from the point of view of any external observer, and which allows multiple concurrent reactions to access a common reactor instance, provided that the results of the reactions do not conflict.
- It handles state *rollback* in the event that a reaction yields a *conflict*: an inconsistent state in a predicate is asserted to be both true and false (alternatively, in which a tuple is both inserted and deleted into the same relation).

In Section 10, we show that the notions of stratification and safety defined for normal Datalog programs extend in a straightforward way to reactor programs.

5 Datalog background

In this section, we review basic concepts from logic programming and Datalog [13]. We will occasionally deviate from standard definitions when appropriate for our context.

5.1 Basic definitions

A *term* is either a constant, a variable, or a built-in function applied to constants or variables. An *atom* consists of an n-ary *predicate symbol* and a list of arguments such that each argument is a term. We will consider both *user-defined* predicates (which we will also refer to as *relations*) and *built-in* predicates, such as equality, inequality, and arithmetic comparison. We allow function symbols only in arguments to built-in predicates. A *literal* is an atom or a negated atom. A *clause* is a finite list of literals. A *ground* clause is a clause which contains neither variables nor function symbols. A *unit* clause is a clause containing only one literal. A *fact* is a positive ground unit clause.

A *Herbrand interpretation* (or just "interpretation") is a set of facts. A *Herbrand model* is a Herbrand interpretation that satisfies every formula belonging to a given set of closed formulas (a given set of facts and rules). A *positive* Datalog program contains rules with only positive atoms. For such programs, there exists a least Herbrand model such that any other Herbrand model is a superset of this model. A *normal* Datalog program is a Datalog program in which negation may appear in body clauses, but not in the head. For normal programs, the existence of a least Herbrand model is no longer guaranteed.

5.2 Stratification semantics for normal programs

To ensure that a least Herbrand model exists without unduly affecting expressiveness, we will restrict the set of normal Datalog programs we consider to those that are *stratified* [14–16]. The main idea behind stratification is to partition the program such that for any relation, we fully compute its contents before applying the negation operator. For example, a program consisting of the following rules is not stratified because it contains recursion through negation:

```
q(x) <- p(x,y), not q(y).
p(1,2) <- .
```

Given a Datalog program P, its *dependency graph* D is a directed graph $\langle N, A\rangle$ with N the set of all user-defined predicate symbols in P and $a \in A$ an edge from p to q if p and q are user-defined predicate symbols in the body and head clauses of a rule r, respectively. An arc between user-defined predicate symbols p and q is marked if the body clause that has p as predicate symbol is negative. P is stratified if there exists no cycle in D containing a marked arc.

The *stratum* of a node in the dependency graph is the maximum number of negative arcs on a path leading to that node. Intuitively, the stratum of a computed relation r is the maximum number of negations that can be applied in evaluating r.

When a Datalog program is stratified, we can designate a single Herbrand model as its semantics by evaluating all the rules of a stratum in the minimal model of the preceding stratum to obtain another minimal model. This unique minimal model is called the *perfect* model.

5.3 Computing a solution

Given a set of rules R, the *immediate consequence operator* Γ_R for normal Datalog programs is a mapping on sets of ground atoms and is defined for a Herbrand interpretation I as follows: $\Gamma_R(I) = \{A|A \in I$ or there exists a rule $A \leftarrow L_1 \wedge ... \wedge L_n \in [\![R]\!]$ such that $L_i \in I$ for all positive literals L_i and $L_j \notin I$ for all negative literals $L_j \equiv \neg L\}$, where $[\![R]\!]$ denotes the set of all ground instances of a set of rules R. $\Gamma^*_R(I)$ is the limit of the sequence of sets $J_0, J_1, \ldots$ such that $J_0 = I$ and $J_k = \Gamma_R(J_{k-1})$ for $k > 0$. For the purpose of this paper we will apply Γ_R on one rule at a time.

We can partition a stratifiable program P into a collection of sets of rules $\mathcal{P} = P_o, \ldots, P_n$ such that each P_j consists of the rules of P whose head relation is in stratum j. Let I_o be an initial Herbrand interpretation. For $0 < j \leq n$, the sequence of instances I_j is defined as follows:

$$I_j = I_{j-1} \cup \Gamma^*_{P_{j-1}}(I_{j-1})$$

The final instance I_n provides the semantics of P under a stratification $P_o, \ldots, P_n$.

The *standard rule evaluation strategy* for a stratified program P

(1) computes a stratification of P

(2) partitions P into P_0, ..., P_n such that each P_j consists of the rules of P whose head belongs to stratum j

(3) given an initial instance I_0, computes the instances I_j for $0 < j \leq n$ as above, yielding solution I_n

Since we require that the full state of a reactor be computed during each reaction, we compute the interpretation of its Datalog program *exhaustively* (bottom-up), in contrast to some evaluation strategies that compute queries *on-demand* (top-down) as required by a particular query.

5.4 Additional semantic restrictions

A Datalog program is *domain independent* if the solution depends only on the initial set of facts and not on the universal set of all facts. A program is *weakly finite* [9] if applying the immediate consequence operator a finite number of times on a finite set of facts yields a finite set of results; i.e., infinite results can only be obtained via infinite recursion. Both domain independence and weak finiteness are desirable properties for Datalog programs intended to represent computations. Since these properties are in general undecidable, we will adopt a conservative syntactic characterization called *safety* [9] which guarantees domain independence and weak finiteness.

A Datalog rule is safe if all of its variables are *limited*. A variable is limited if:

(1) it occurs as an argument to a non-negated user-defined predicate in the body
(2) it occurs as one of the arguments to the built-in equality predicate and all of the other variables that occur in the same clause are limited

A program is safe if all of its rules are safe. Consider a few examples:

```
answer(x) <- mynumber(x), not zero(x). // 1
P(x) <- Q(y), R(z), x = y * z.         // 2
P(x) <- Q(y), R(z), y = x * z.         // 3
```

Rule (1) is safe, but removing `mynumber(x)` renders it unsafe because `x` then is not limited. Rule (2) is safe because `x` is the only non-limited argument to the equality predicate, hence its value is uniquely determined by `y` and `z`. By contrast, rule (3) is not safe, since both arguments contain non-limited variables and the value of `x` is not uniquely determined by `y` and `z`.

5.5 Datalog evaluation

The evaluator is defined as a relation $\hookrightarrow$ on pairs of *evaluation states*. An evaluation state is a pair $\langle I, i\rangle$ where

- I is a set of literals defining an *interpretation*.
- i is the index of the current stratum under evaluation

$$\frac{P_i \in \mathcal{P} \qquad R \in P_i \qquad I' = \Gamma_R(I) \qquad I' \neq I}{\mathcal{P} \vdash \left\langle I, i \right\rangle \hookrightarrow \left\langle I', i \right\rangle}$$

$$\frac{i < \mathit{numstrata}(\mathcal{P}) \qquad \forall R,\ (R \in P_i \text{ s.t. } P_i \in \mathcal{P}) \ \Rightarrow\ I = \Gamma_R(I)}{\mathcal{P} \vdash \left\langle I, i \right\rangle \hookrightarrow \left\langle I, i+1 \right\rangle}$$

Fig. 11. Small-step operational semantics for *eval*

Figure 11 shows two small-step semantic rules which capture the behavior of the evaluator. The first rule specifies a single step in which the immediate consequence operator Γ_R is applied to the current Herbrand interpretation I to obtain the next interpretation I' for a given rule R. R is a non-deterministically chosen rule from the stratum under current evaluation $P_i \in \mathcal{P}$. If applying Γ_R modifies the state by inferring new facts, then the $\hookrightarrow$ relation transitions to the next state corresponding to I'. If, on the other hand, there exists no rule that can be evaluated which modifies the current state, then the evaluation of the current stratum ends and is marked appropriately by incrementing the stratum index. The evaluation stops when the last stratum has been fully evaluated.

6 Basic reactor formalities

In this section, we will define some basic reactor notation that will be used in the rest of the paper.

A reactor *program* $\mathcal{T}$ is a collection $T_1, T_2, \ldots, T_n$ of reactor *type definitions*, each defined using the REACTOR grammar of Fig. 1, with the proviso that a RULE-DECL may contain at most one instance of the `new` keyword in its HEAD-CLAUSE.

Each type definition T has the form

`def` *reactor-type-name* `=` $\cdots$

hence we will frequently use *reactor-type-name* to denote the corresponding type definition.

Each reactor type T defines a collection of *ephemeral* relations *ephemeral*(T), declared using the syntactic form RELATION-DECL of Fig. 1 and the `ephemeral` keyword; similarly, it defines a collection of *persistent* relations *persistent*(T), declared using the same syntactic form without the `ephemeral` keyword. The *access attributes* `public`, `public read`, and `public write` may be used in the declaration to allow (possibly restricted) access to a relation by other reactors; otherwise, the relation is *private*, and not accessible to remote reactors. To simplify exposition, we will assume in the sequel that all reactor names for a program $\mathcal{T}$ are globally unique. The set of persistent relations declared in $\mathcal{T}$ are given by *persistent*(T); the set of ephemeral relations declared in $\mathcal{T}$ are given by *ephemeral*(T).

A *local* literal is a literal of the form

$$[\texttt{not}]\ \mathit{ref}\texttt{(}\ldots\texttt{)}$$

where *ref* has the form `t`, `-t`, `^t`, or `t^`, for some relation t.

A *remote* r-literal is a literal of the form

$$[\texttt{not}]\ r\,\texttt{.}\,\mathit{ref}\texttt{(}\ldots\texttt{)}$$

where *ref* is as above and r is a variable bound to a reactor reference. If r is a reference to a reactor of type S, then the relation referred to in *ref* (say p) must be a public relation, i.e., declared using the `public` access attribute in the definition of S. Furthermore, if the remote r-reference occurs in a HEAD-CLAUSE, then p must be declared using either the `public` or `public write` attributes; if the reference occurs in a BODY-CLAUSE, then p must be declared using either the `public` or `public read` attributes.

A reactor reference of the form `r` denotes the *response value* of relation `r`; `-r` denotes the *prestate value* of `r`; `r^` denotes the *future value* of `r`, and `^r` denotes the *stimulus* value of `r`.

7 Semantics of reactor references

In this section, we show how we can “compile away” reactor constructs which manipulate reactor references by translating such constructs into ordinary Datalog. This translation process serves to define the semantics of reference-manipulating constructs.

The first phase of the reference translation process (Sections 7.1 and 7.2) formalizes when remote reactors are *read* and *written* by another reactor via a reactor reference. The formalization works by introducing two auxiliary relations for every reactor of type T: *active_ T* and *touched_ T*, and transforming the rules of the original program to use these new relations appropriately. Intuitively, *touched_ T* will become true for reactor instance t of type T whenever a remote reactor accesses relations of t. Similarly, *active_ T* will become true for reactor t whenever a remote reactor writes a relation of t. The relations *touched_ T* and *active_ T* will be examined by the concurrent transition semantics of Section 9 to determine when and how to access the state of remote reactors. Importantly, the relation *active_ T* is also used to ensure that only the rules of reactors which have been *written to* are evaluated in the current reaction.

The second phase of the reference translation process (Section 7.3) encodes the semantics of synchronous reactions, in which multiple reactor instances which refer to one another via references are evaluated atomically. This *augmentation* translation adds an extra column to every relation r of reactor type T. This column contains reactor references (of type T) which encode the reactor instance which "owns" the tuples of r.

The final phase of the reference translation process (Section 7.4) formalizes the semantics of reactor instantiation and initialization, i.e., the `new` construct.

7.1 Formalizing remote reactor writes and reaction scope: the active *transformation*

The first step of the reference translation process makes reaction scope explicit by adding to all reactor definitions a relation called *active_ T*, where T is the reactor's type. The *active_ T* relation defines the semantics of "scope extrusion", the process by which multiple reactors react together synchronously. Only those reactor instances for which *active_ T* is true are included in the scope of the current reaction. The basic idea is to set *active_ T* to true for reactor instance t whenever a relation of t is written, and to guard all rules with a test of *active_ T* in such a way as to ensure that a rule is only executed for reactor instances that are part of the current reaction scope.

In every reactor in the program the transformation adds the declaration

$$\texttt{public ephemeral } \mathit{active\_T} : ().$$

We assume without loss of generality that there is no pre-existing relation called *active_ T*.

Next, we non-recursively transform each rule such that

$$\begin{array}{rcrl} [\texttt{not}]\ r.a(\mathbf{e})\,[\texttt{\^{}}]\ \texttt{<-}\mathit{body}. & \text{becomes} & [\texttt{not}]\ r.a(\mathbf{e})\,[\texttt{\^{}}] & \texttt{<-}\ \mathit{body}, \mathit{active\_T}(). \\ & & r.\mathit{active\_U}()\,[\texttt{\^{}}] & \texttt{<-}\ \mathit{body}, \mathit{active\_T}(). \end{array}$$

$$\begin{array}{rcrl} [\texttt{not}]\ a(\mathbf{e})\,[\texttt{\^{}}]\ \texttt{<-}\mathit{body}. & \text{becomes} & [\texttt{not}]\ a(\mathbf{e})\,[\texttt{\^{}}] & \texttt{<-}\ \mathit{body}, \mathit{active\_T}(). \end{array}$$

where T is the type name of the reactor containing the rules that are being transformed, and U is the type name of the reactor r.

Since all rules are now guarded by an *active_ T* relation, the *active_ T* relation of the reactor instance that initiated the reaction (i.e. received an update bundle) must be set to true. We therefore require that an update bundle write the *active_ T* relation of the receiving reactor, but conveniently the transformation already handles this.

Example 1 Let us consider how an example program is transformed. The following example models a simple publisher/subscriber mechanism, but for simplicity the mechanism handling subscription and unsubscription have been left out. Here is what the program looks like before (left) and after (right) the transformation:

```
def Pub = {
  public data : (int).
  subscriber  : (ref Sub).

  s.copy(x)   <- data(x), subscriber(s).




}

def Sub = {
  copy        : (int).
  intStore    : (int).

  intStore(x) <- copy(x).
}
```

```
def Pub = {
  public data     : (int).
  subscriber      : (ref Sub).
  public ephemeral active_Pub : ().
  s.copy(x)       <- data(x), subscriber(s),
                     active_Pub().
  s.active_Sub() <- data(x), subscriber(s),
                     active_Pub().
}

def Sub = {
  public copy     : (int)
  intStore        : (int)
  public ephemeral active_Sub : ().
  intStore(x)     <- copy(x), active_Sub().
}
```

Example 2 Intuitively, the semantics of the *active_ T* relations is intended to expose the control flow implicit in the scope extrusion process via an explicit data dependency. The example in the left column at first glance seems benevolent enough. Intuitively, however, we should not accept it, because `q` is not fully computed before it is tested for non-membership, but this test is necessary to extrude the scope to the reactor of type `R2` whose rule adds tuples to `q`. The right column shows the transformed example after adding the two *active_ T* relations. The example is now non-stratifiable due to the relation `active_R2` for the reactor of type `R2`. This relation effectively acts as a “summary node”—a relation which guards all rules and is written when the scope extrudes. A negative cycle is created between `q`, `s`, and `active_R2`. This negative cycle makes concrete our intuition that the example in the left column should not be an acceptable program.

```
// The rules are stratifiable, but the
// program should intuitively be rejected

def R1 = {
  r        : (int, int, ref R2).
  public q  : (int).
  s        : (int).


  s(x)      <- not q(x).
  z.t(y, x) <- s(x), r(x, y, z).
}




def R2 = {
  p        : (int, int, ref R1).
  public t  : (int, int).


  y.q(x)    <- p(z, x, y).
}
```

```
// After introducing the active relation
// the hidden negative cycle is exposed

def R1 = {
  r               : (int, int, ref R2).
  public q        : (int).
  s               : (int).
  public active_R1 : ().

  s(x)           <- not q(x), active_R1().
  z.t(y, x)      <- s(x), r(x, y, z),
                    active_R1().
  z.active_R2() <- s(x), r(x, y, z),
                    active_R1().
}

def R2 = {
  p               : (int, int, ref R1).
  public t        : (int, int).
  public active_R2 : ().

  y.q(x)         <- p(z, x, y), active_R2().
  y.active_R1() <- p(z, x, y), active_R2().
}
```

7.2 Formalizing remote reactor reads: the touched *transformation.*

In this section, we define the *touched_ T* relation, which records those remote reactor instances of type *T* which could be accessed in the current reaction (this set can be an overapproximation of the set of reactor instances which are going to be read or written in the current reaction). For each reactor of type *T*, we add a declaration of the form

$$\texttt{public ephemeral}\ \textit{touched\_ T}\ \texttt{: ().}$$

Let rule *R* be a rule of the form:

$$\textit{head}\ \texttt{<-}\ \textit{clause}_1\texttt{,}\ldots\texttt{,}\ \textit{clause}_n\texttt{.}$$

containing one or more remote literals *r*.

For each clause_i which binds r we add the following additional rules to T's declaration:

$$r\,.\,\mathit{touched\_}\,T_j\texttt{()}\quad\texttt{<-}\quad \mathit{clause}_i\,.$$

where T_j is the reactor type name of r respectively.

This construction ensures that the *touched_* T_j relation of reactor instance r becomes true whenever a relation of r is read or written by another reactor.

7.3 Flattening inter-reactor references: the augmentation *transformation*

The *augmentation* transformation in this section readies reactors for synchronous reaction by allowing them to be treated together as single "virtual" reactor. The augmentation transformation is applied *after* the *active_ T*, *touched_ T*, and instantiation (new) transformations, which are each applied independently to source rules.

The program is given as a set of reactor declarations, named $R_1, ..., R_k$. We assume without loss of generality that relation names are globally unique, i.e. any relation name is declared at most once.

In each reactor declaration R rewrite each rule

$$\begin{aligned}[\texttt{not}]\ \ r.h(\mathbf{e})\,[\texttt{\^{}}]\ \ \texttt{<-}\ & r_1\texttt{.}\,[\texttt{\^{}|-}]\,b_1(\mathbf{x_1}), ..., r_n\texttt{.}\,[\texttt{\^{}|-}]\,b_n(\mathbf{x_n}),\\ & [\texttt{\^{}|-}]\,c_1(\mathbf{y_1}), ..., [\texttt{\^{}|-}]\,c_m(\mathbf{y_m}),\\ & \mathit{predicates}.\end{aligned}$$

as

$$\begin{aligned}[\texttt{not}]\ \ h(r,\mathbf{e})\,[\hat{}]\ \ \texttt{<-}\ & [\texttt{\^{}|-}]\,b_1(r_1,\mathbf{x_1}), ..., [\texttt{\^{}|-}]\,b_n(r_n,\mathbf{x_n}),\\ & [\texttt{\^{}|-}]\,c_1(s,\mathbf{y_1}), ..., [\texttt{\^{}|-}]\,c_m(s,\mathbf{y_m}),\\ & \mathit{predicates}[s\backslash\texttt{self}].\end{aligned}$$

and rewrite

$$\begin{aligned}[\texttt{not}]\ \ h(\mathbf{e})\,[\texttt{\^{}}]\ \ \texttt{<-}\ & r_1\texttt{.}\,[\texttt{\^{}|-}]\,b_1(\mathbf{x_1}), ..., r_n\texttt{.}\,[\texttt{\^{}|-}]\,b_n(\mathbf{x_n}),\\ & [\texttt{\^{}|-}]\,c_1(\mathbf{y_1}), ..., [\texttt{\^{}|-}]\,c_m(\mathbf{y_m}),\\ & \mathit{predicates}.\end{aligned}$$

as

$$\begin{aligned}[\texttt{not}]\;\; h(s,\mathbf{e})\,[\texttt{\^{}}]\;\; \texttt{<-}\;\; & [\texttt{\^{}}|\texttt{-}]\, b_1(r_1,\mathbf{x_1}),...,[\texttt{\^{}}|\texttt{-}]\, b_n(r_n,\mathbf{x_n}),\\ & [\texttt{\^{}}|\texttt{-}]\, c_1(s,\mathbf{y_1}),...,[\texttt{\^{}}|\texttt{-}]\, c_m(s,\mathbf{y_m}),\\ & \mathit{predicates}[s\backslash\texttt{self}].\end{aligned}$$

where s is a fresh identifier, and $[x\backslash y]$ means "put x instead of y in the preceding expression".

The transformation does two things: it replaces all references to `self` with a fresh identifier, s, and it augments the relations by adding reactor references as the first column of every relation. Notice that the transformation also alters the *active_ T* relations added by the previous transformation. The occurrence of `active_T()` in the body of each rule gets replaced by `active_T(`s`)`; and r`.active_T()` in the head gets replaced by `active_T(`r`)`.

Example 3 Consider the example from last section. We now take the program with the *active* relation already added, and pass it through the augmentation transformation. Here is the program before (left) and after (right):

```
def Pub = {
  public data    : (int).
  subscriber     : (ref Sub).
  public ephemeral active_Pub : ().
  s.copy(x)      <- data(x), subscriber(s),
                    active_Pub().
  s.active_Sub() <- data(x), subscriber(s),
                    active_Pub().
}

def Sub = {
  public copy    : (int)
  intStore       : (int)
  public ephemeral active_Sub : ().
  intStore(x)    <- copy(x), active_Sub().
}
```

```
// Rules from Publisher



copy(s,x) <- data(id1,x),subscriber(id1,s),
             active_Pub(id1).
active_Sub(s) <-
             data(id2,x),subscriber(id2,s),
             active_Sub(id2).

// Rules from Subscriber



intStore(id3,x) <-
             copy(id3,x),active_Sub(id3).
```

7.4 Formalizing reactor creation: the instantiation *transformation*

The `new` construct is used to instantiate new reactors. To model this behavior, we must generate fresh reactor references appropriately for each rule containing `new` (recall that `new` can occur at most once in any rule head). But what behavior is "appropriate"?

Consider the following examples. We would like the rule

```
r(new) <- .
```

to produce exactly one new reactor per reaction. The rule

```
r(x, new) <- t(x)
```

should generate a distinct reactor for every value of `x`, in every reaction. A program containing the rules

```
s(x, new) <- t(x). // 1
s(x, new) <- t(x). // 2
```

should produce the same result as a program containing a single instance of the rule; moreover, replacing rule (2) above with the rule `s(x, new) <- q(x)` should also yield the same result when the contents of `t` and `q` are identical.

In general, we want a rule of the form

$$r\texttt{(}\mathbf{x}_{1,i}\texttt{, new, }\mathbf{x}_{i+1,n}\texttt{) <-}\textit{body}.$$

to generate a fresh reference value for every tuple of values $\langle \mathbf{x}_{1,i}, \mathbf{x}_{i+1,n} \rangle$ which satisfy the rule, in every reaction.

We make the intuition above precise by transforming rules containing `new` to standard Datalog, augmented with functors (term constructors).

In every reactor of type T in the program, the transformation adds the declaration

$$\texttt{public ephemeral } \textit{reactions\_T} \texttt{ : (int).}$$

The idea is that *reactions_ T* relations maintain a count of the number of reactions that the containing reactor has undergone since being instantiated. This count will be used to ensure that distinct reference values are appropriately generated for each reaction.

After applying the augmentation transformation of Section 7.3, each rule in a reactor of type T containing an instance of `new` will have the form

$$r\texttt{(}s\texttt{, }\mathbf{x}_{1,i}\texttt{, new, }\mathbf{x}_{i+1,n}\texttt{) <- }\textit{body}\texttt{, }\textit{active\_T}\texttt{(}s\texttt{).}$$

Let us assume that relation `r` has a declaration of the form

$$\ldots\ r\texttt{: (}\mathbf{T}_{1,i}\texttt{, ref }S\texttt{, }\mathbf{T}_{i+1,n}\texttt{).}$$

where $\mathbf{T}_{1,i}$ and $\mathbf{T}_{i+1,n}$ are vectors of types.

We transform the rule above into the following three rules:

$$r(s,\ \mathtt{x}_{1,i},\ \mathtt{s'},\ \mathtt{x}_{i+1,n})\ \texttt{<-}\ \mathit{body},\ \mathit{reactions\_}T(s,\ \mathtt{j}),\ \mathit{active\_}T(s)$$
$$\mathtt{s'}\ =\ \mathit{new\_r\_i}\texttt{<}s,\ \mathtt{j},\ \mathtt{x}_{i,1},\ \mathtt{x}_{i+1,n}\texttt{>}.\qquad \texttt{// 1}$$

$$\mathit{active\_}S(\mathtt{s'})\qquad \texttt{<-}\ \mathit{body},\ \mathit{reactions\_}T(s,\ \mathtt{j}),\ \mathit{active\_}T(s)$$
$$\mathtt{s'}\ =\ \mathit{new\_r\_i}\texttt{<}s,\ \mathtt{j},\ \mathtt{x}_{i,1},\ \mathtt{x}_{i+1,n}\texttt{>}.\qquad \texttt{// 2}$$

$$\mathit{reactions\_}S(\mathtt{s'},\ \mathtt{k})\qquad \texttt{<-}\ \mathit{body},\ \mathit{reactions\_}T(s,\ \mathtt{j}),\ \mathit{active\_}T(s)$$
$$\mathtt{s'}\ =\ \mathit{new\_r\_i}\texttt{<}s,\ \mathtt{j},\ \mathtt{x}_{i,1},\ \mathtt{x}_{i+1,n}\texttt{>},\ \mathtt{k}\ =\ \mathtt{0}.\ \texttt{// 3}$$

Here, $\mathit{new\_r\_i}\texttt{<...>}$ is a functor (data constructor) defined for each relation `r` in which `new` appears in position i in the head of some rule. The first argument (s) of the functor encodes a reference to the reactor's "parent" (i.e., the reactor generating the new reference); the second argument (`j`) is the current reaction count, and the remaining arguments contain the values to which the other variables in the original rule are bound. Embedding the parent reference in the encoding of a newly-generated (child) reference value ensures that the each reference value is globally unique. In addition to "installing" the newly-generated reference in its containing relation (rule 1 above), we must also "activate" the reactor rules for the type corresponding to the reference (rule 2), and initialize the reaction count for the newly-generated reactor to 0 (rule 3)

Finally, for every reactor of type T, we add the following two rules

$$\mathit{reactions\_}T(s,\ \mathtt{j'})\ \texttt{<-}\ \texttt{-}\mathit{reactions\_}T(s,\ \mathtt{j}),\ \mathit{active\_}T(s),\ \mathtt{j'}\ =\ \mathtt{j+1}.\ \texttt{// 4}$$
$$\mathit{reactions\_}T(s,\ \mathtt{j})\ \ \texttt{<-}\ \texttt{-}\mathit{reactions\_}T(s,\ \mathtt{j}),\ \mathit{active\_}T(s).\qquad \texttt{// 5}$$

Rules 4 and 5 together increment the reaction count for a reactor of type T on every reaction.

It is straightforward to define a total order on generated reactor references (e.g., based on a total ordering for relation names). This property will be used in the operational semantics defined in Section 9.

8 Eliminating head negation

We handle negation in head clauses by transforming reactor rules to normal Datalog rules. First, we treat each of the four states of a given relation as distinct relations from the point of standard Datalog semantics. The goal of reactor rule evaluation is to determine a unique, minimal solution for the response and future values of local and remote relations. Let r_i represent the response or future value of a local relation we wish to compute. If the reactor has just been created all the relations are empty by default; ephemeral relations are always initially empty. Let $\texttt{\^{}r}_i^{\Delta^+}$ and $\texttt{\^{}r}_i^{\Delta^-}$ be the addition and the deletion sets of the update bundle applied to the reactor. We will use $\texttt{r}\ (\mathbf{x})$ to denote the relation $\texttt{r}$ applied to a tuple of arguments $(\mathbf{x})$. The basic idea for determining the solution to r_i is to:

(1) introduce a pair of auxiliary relations ($\underline{\texttt{r}}_i^{\Delta^+}$,$\underline{\texttt{r}}_i^{\Delta^-}$) which contains the sets of tuples that will be added to and deleted from r_i;
(2) eliminate negation in head clauses by transforming the program to a normal Datalog program containing references to $\underline{\texttt{r}}_i^{\Delta^+}$ and $\underline{\texttt{r}}_i^{\Delta^-}$;

The resulting program is the input to the evaluator presented in Section 5.5. Fig. 12 shows how our rewriting technique transforms a program with negation in the head clauses to a program without them. The rules apply recursively. Let us define $\underline{\texttt{r}}_i^{\Delta^+}$, $\underline{\texttt{r}}_i^{\Delta^-}$, $\underline{\texttt{r\^{}}}_i^{\Delta^+}$ and $\underline{\texttt{r\^{}}}_i^{\Delta^-}$ the sets of additions and deletions to the response and future states of the persistent and ephemeral relations.

Rewrite:	$\texttt{r}_i\,(\mathbf{x})$ `<-` *body.*	as:	$\underline{\texttt{r}}_i^{\Delta^+}(\mathbf{x})$ `<-` *body.*	(I)
Rewrite:	`not` $\texttt{r}_i\,(\mathbf{x})$ `<-` *body.*	as:	$\underline{\texttt{r}}_i^{\Delta^-}(\mathbf{x})$ `<-` $\texttt{\^{}r}_i\,(\mathbf{x})$, *body.*	(II)
Rewrite:	*head* `<-` $\texttt{r}_i\,(\mathbf{x})$, *body.*	as:	*head* `<-` $\texttt{\^{}r}_i\,(\mathbf{x})$, `not` $\underline{\texttt{r}}_i^{\Delta^-}(\mathbf{x})$, *body.* *head* `<-` $\underline{\texttt{r}}_i^{\Delta^+}(\mathbf{x})$, *body.*	(III)
Rewrite:*head* `<-` `not` $\texttt{r}_i\,(\mathbf{x})$, *body.*		as:	*head* `<-` `not` $\texttt{\^{}r}_i\,(\mathbf{x})$, `not` $\underline{\texttt{r}}_i^{\Delta^+}(\mathbf{x})$, *body.* *head* `<-` $\underline{\texttt{r}}_i^{\Delta^-}(\mathbf{x})$, *body.*	(IV)
Rewrite:	$\texttt{r\^{}}_i\,(\mathbf{x})$ `<-` *body.*	as:	$\underline{\texttt{r\^{}}}_i^{\Delta^+}(\mathbf{x})$ `<-` *body.*	(V)
Rewrite:	`not` $\texttt{r\^{}}_i\,(\mathbf{x})$ `<-` *body.*	as:	$\underline{\texttt{r\^{}}}_i^{\Delta^-}(\mathbf{x})$ `<-` *body.*	(VI)

Fig. 12. Rewrite rules defining the semantics of reactor rule evaluation in terms of normal Datalog.

Rewrite rule (I) computes the set of tuples to be added to $\texttt{r}_i$ as the set of tuples that the body clauses resolve to. Rule (II) computes the deletion set very similarly;

the only difference is adding a body clause which makes sure that a tuple gets deleted from a relation only if it was already there in the stimulus state. The extra clause ensures that this rewriting rule does not introduce domain dependence—see Section 10.1 for details. Our rewriting approach does not update relations in place during a reaction; accounting for the addition and the deletion sets in the rules must be therefore done explicitly by modifying the rules. As a result, rewrite rule (III) restricts the matching for the tuples in r_i to the ones that are not in the deletion set; it also adds another rule which matches tuples in the addition set. Conversely, rule (IV) allows matching on tuples in the deletion set, as well as restricting matching to tuples not in the addition set. Rules (V) and (VI) do not apply for ephemeral relations because they do not have a future state.

Let us consider the order entry example in Fig. 2. After applying the head negation transformation the rule in the reactor of type `OrderEntryA'` will result in the following set of rules:

$$\begin{aligned}
&\underline{\texttt{log}}^{\Delta^+}\texttt{(id, itemid, qty) <- } && \texttt{orders(id, itemid, qty).}\\
&\underline{\texttt{log}}^{\Delta^-}\texttt{(id, itemid, qty) <- } && \texttt{\textasciicircum log(id, itemid, qty),}\\
& && \texttt{not \textasciicircum orders(id, itemid, qty),}\\
& && \texttt{not }\underline{\texttt{orders}}^{\Delta^+}\texttt{(id, itemid, qty).}\\
&\underline{\texttt{log}}^{\Delta^-}\texttt{(id, itemid, qty) <- } && \texttt{\textasciicircum log(id, itemid, qty),}\\
& && \underline{\texttt{orders}}^{\Delta^-}\texttt{(id, itemid, qty).}
\end{aligned}$$

Note that the tuple `(id, itemid, qty)` will be deleted from `log` if it is in the prestate and either is in the deletion set of `orders` or it is neither in the prestate, nor in the addition set of `orders`.

9 Semantics of reactor interaction

In this section, we define the operational semantics of distributed reactor interaction, using the rules depicted in Fig. 13. These rules effectively define a "scheduler" that initiates concurrent reactions and orchestrates the interchange of data among reactor instances. The transition relation of the operational semantics defines a computation on *worlds*. A world has the form

$$\mathcal{P}, N \vdash q_1, \ldots, q_k, s_{k+1}, \ldots, s_n$$

The components of a world are defined as follows:

- The *program* $\mathcal{P}$ is an ordered collection of subprograms $P_1, \ldots, P_m$ derived from a given reactor program $\mathcal{T}$ by first transforming $\mathcal{T}$ using the transformations described in Sections 7 and 8, then stratifying the resulting collection of Datalog rules into subprograms $P_1, \ldots, P_m$, as described in Section 5.2.
- The *network* N is a multiset of *update bundles*, each of which has the form (β_i, I_{β_i}), where β is a reactor reference, and I is a Herbrand interpretation containing only facts of the form $\underline{\mathtt{r}}\hat{}^{\Delta^+}(\beta, \mathbf{x})$ or $\underline{\mathtt{r}}\hat{}^{\Delta^-}(\beta, \mathbf{x})$, representing future additions or deletions, respectively, to reactor β
- A set of *initiating reactors* $s_{k+1}, \ldots, s_n$. An initiating reactor is a reactor at which an uncompleted reaction has been initiated. Each s_j takes the form $\langle b_{\alpha_j}, I_{\alpha_j}, l_j \rangle$, where b_{α_j} is an update bundle, I_{α_j} is an interpretation (over all of the relations defined by the transformation of $\mathcal{T}$), and l_j is an integer defining the current stratum of $\mathcal{P}$ being executed by the reactor.
- A set of *quiescent reactors* $q_1, \ldots, q_k$. A quiescent reactor is a reactor that is not an initiation site for an active reaction. Each q_i takes the form (α_i, I_{α_i}), where α is a reactor reference and I_{α_i} is an interpretation containing only facts of the form $\mathtt{-r}(\alpha, \mathbf{x})$. We thus see the *persistent state* of a reactor is encoded by terms over pre-state values of relations.

Each reactor in a world occurs exactly once in the world state, either as a quiescent reactor or as an initiating reactor.

In the world representation described above, we view a relation as a set of *facts*, i.e., a ground Herbrand interpretation, rather than as a named collection of tuples. While the two views of state are semantically equivalent, the "soup of atoms” Herbrand representation accommodates state manipulation involving multiple relations more readily than a state representation using sets of maps from names to tuples.

The rules of Fig. 13 require a number of auxiliary functions, primarily to manage operations on the Herbrand interpretations representing reactor states. The formal definitions of these functions are given in Fig. 14, here we provide more intuitive definitions.

$\uplus$: Multiset union

$\hookrightarrow$: The Datalog evaluation relation as defined in Section 5.5. Note that the translations of Sections 7 and 8 have the property that when a remote reactor is *written*, the reactor’s reference is added to a relation of the form $\underline{\mathtt{active\_}T}^{\Delta^+}$; when it is *read*, the reference is added to a relation of the form $\underline{\mathtt{touched\_}T}^{\Delta^+}$.

$$\text{START}\,\frac{b = (\beta, I^{\Delta}) \qquad i = 0 \qquad I' = \text{init}(I_0, I^{\Delta})}{\mathcal{P}, N \uplus \{b\} \vdash (\beta, I_0) \xrightarrow{\mathit{start}} \mathcal{P}, N \vdash \big\langle b, I', i \big\rangle}$$

$$\text{EVAL}\,\frac{\mathcal{P} \vdash \big\langle I, i \big\rangle \hookrightarrow \big\langle I', i' \big\rangle \qquad \underline{\texttt{touched\_}T}^{\Delta^+}(\alpha) \in I' \implies \underline{\texttt{touched\_}T}^{\Delta^+}(\alpha) \in I}{\mathcal{P}, N \vdash \big\langle b, I, i \big\rangle, \xrightarrow{\mathit{eval}} \mathcal{P}, N \vdash \big\langle b, I', i' \big\rangle}$$

$$\text{READ}\,\frac{\mathcal{P} \vdash \big\langle I, i \big\rangle \hookrightarrow \big\langle I', i' \big\rangle \qquad \underline{\texttt{touched\_}T}^{\Delta^+}(\alpha'') \in I' \setminus I \qquad I''' = I' \cup \text{init}(I_0'', \emptyset)}{\mathcal{P}, N \vdash \big\langle b, I, i \big\rangle, (\alpha'', I_0'') \xrightarrow{\mathit{read}} \mathcal{P}, N \vdash \big\langle b, I''', i' \big\rangle, (\alpha'', I'')}$$

$$\text{CONFLICT}\,\frac{\text{conflict}(I) \qquad \{(\alpha'', I'')\} = \text{revert}(\{\alpha''\}, I') \qquad b = (\alpha, I^{\Delta})}{\mathcal{P}, N \vdash \big\langle b, I, i \big\rangle \xrightarrow{\mathit{conflict}} \mathcal{P}, N \vdash (\alpha'', I'')}$$

$$\text{PREPARE}\,\frac{\begin{array}{c} i = \mathit{numstrata}(\mathcal{P}) \\ \neg\text{conflict}(I) \qquad L = \{\alpha_i \mid \underline{\texttt{active\_}T}^{\Delta^+}(\alpha_i) \in I\} \setminus \beta \qquad b = (\beta, I^{\Delta}) \end{array}}{\mathcal{P}, N \vdash \big\langle b, I, i \big\rangle \xrightarrow{\mathit{prepare\ to\ lock}} \mathcal{P}, N \vdash \big\langle b, I, i \big\rangle_{\{\},L}}$$

$$\text{ACQUIRE}\,\frac{\text{uptodate}((\alpha_i, I_i), I) \qquad \forall \alpha_j,\ i < j \leq n \implies \alpha_i < \alpha_j}{\begin{array}{l} \mathcal{P}, N \vdash \big\langle b, I, i \big\rangle_{\{\alpha_1, \ldots, \alpha_{i-1}\}, \{\alpha_i, \alpha_{i+1}, \ldots, \alpha_n\}}, (\alpha_i, I_i) \xrightarrow{\mathit{acquire\ lock}} \\ \mathcal{P}, N \vdash \big\langle b, I, i \big\rangle_{\{\alpha_1, \ldots, \alpha_i\}, \{\alpha_{i+1}, \ldots, \alpha_n\}} \end{array}}$$

$$\text{RETRY}\,\frac{\begin{array}{c} \neg\text{uptodate}((\alpha_i, I_i), I) \\ \{(\alpha', I'), (\alpha_1, I_1), \ldots, (\alpha_{i-1}, I_{i-1})\} = \text{revert}(\{\alpha', \alpha_1, \ldots, \alpha_{i-1}\}, I) \qquad b = (\alpha', I^{\Delta}) \end{array}}{\begin{array}{l} \mathcal{P}, N \vdash \big\langle b, I, i \big\rangle_{\{\alpha_1, \ldots, \alpha_{i-1}\}, \{\alpha_i, \alpha_{i+1}, \ldots, \alpha_n\}}, (\alpha_i, I_i) \xrightarrow{\mathit{snapshot\ stale/retry}} \\ \mathcal{P}, N \uplus \{b\} \vdash (\alpha', I'), (\alpha_1, I_1), \ldots, (\alpha_{i-1}, I_{i-1}), (\alpha_i, I_i) \end{array}}$$

$$\text{COMMIT}\,\frac{\{(\alpha_1, I_1) \ldots, (\alpha_n, I_n)\} = \text{persist}(I) \qquad \{b_1, \ldots, b_m\} = \text{bundles}(I)}{\mathcal{P}, N \vdash \big\langle b, I, i \big\rangle_{\{\alpha_1, \ldots, \alpha_n\}, \{\}} \xrightarrow{\mathit{commit}} \mathcal{P}, N \uplus \{b_1, \ldots, b_m\} \vdash (\alpha_1, I_1), \ldots, (\alpha_n, I_n)}$$

$$\text{CONTEXT}\,\frac{\mathcal{P}, N \vdash q_{l+1}, \ldots, q_k, \langle\cdot\rangle_{m+1}, \ldots, \langle\cdot\rangle_n \xrightarrow{<\mathit{any}>} N' \vdash q'_{l+1}, \ldots, q'_{k'}, \langle\cdot\rangle'_{m+1}, \ldots, \langle\cdot\rangle'_{n'}}{\begin{array}{l} \mathcal{P}, N \vdash q_1, \ldots, q_l, q_{l+1}, \ldots, q_k, \langle\cdot\rangle_1, \ldots, \langle\cdot\rangle_m, \langle\cdot\rangle_{m+1}, \ldots, \langle\cdot\rangle_n \xrightarrow{<\mathit{any}>} \\ \qquad N' \vdash q_1, \ldots q_l, q'_{l+1}, \ldots, q'_{k'}, \langle\cdot\rangle_1, \ldots, \langle\cdot\rangle_m, \langle\cdot\rangle'_{m+1}, \ldots, \langle\cdot\rangle'_{n'} \end{array}}$$

Fig. 13. High-level operational semantics

init(I_0, I^{Δ})**:** Given an an interpretation I_0 representing the persistent state of a quiescent relation and an interpretation I^{Δ} representing the contents of an update bundle, yields an initial interpretation I' suitable for evaluation. I' contains I_0, as well as stimulus values for relations generated by applying I^{Δ} to I_0.

conflict(I)**:** Given an interpretation I, determines whether I contains a *conflict*, i.e., whether the encoding of response or future state in I contains both a fact and its negation.

revert(S, I)**:** Given a set of reactor references S and an interpretation I representing the current state of the reaction, yields a collection of quiescent reactors of the form (α, I_α) such that $\alpha \in S$, and I_α contains only the prestate values of α's relations. In other words, this operation "rolls back" the state of the reactors in S to their pre-reaction values.

persist(I)**:** Given an interpretation I representing the current state of the reaction, yields a collection of quiescent reactors of the form (α_i, I_{α_i}), whose prestate values are computed from response values of the corresponding relations in I. The response values are in turn computed from relations of the form $\underline{\mathbf{r}}^{\Delta^+}$ and $\underline{\mathbf{r}}^{\Delta^-}$ and the stimulus values of the corresponding relations. The references α_i correspond to the active (written) reactors in the current reaction. This operation has the effect of updating the persistent state of reactors involved in the reaction with the new state resulting from the reaction.

bundles(I)**:** Given an interpretation I representing the current state of the reaction, yields a collection of update bundles corresponding to the future state values of relations computed in I.

uptodate(q, I)**:** Given the contents of a quiescent reactor q and an interpretation I representing the current state of the reaction, determines whether the response values of relations of q in I are identical to the prestate values of the corresponding relations in q, i.e., that the state "snapshot" created when q's state was incorporated into the current reaction has not been invalidated by updates to q made by other reactions.

Let us now look at the intuition behind each of the rules in the semantics. It may be useful to read the following descriptions in parallel with the inference rules given in Fig. 13.

START A reactor takes an update bundle addressed to it off the network and applies the update bundle to obtain the stimulus state used for the reaction. It also sets the index of the current stratum that is to be evaluated to zero, the bottom stratum.

EVAL This corresponds to a single step of the evaluator, where no further intervention by the scheduler is required.

READ If, on inspection of the evaluator state, the scheduler observes that at least one new reactor reference was added to a relation of the form $\underline{\texttt{touched\_}T}^{\Delta^+}$, then for each new reactor reference in $\underline{\texttt{touched\_}T}^{\Delta^+}$, the scheduler creates a "snapshot" of its state and copies it into the state of the reaction.

CONFLICT If the reaction state contains a conflict, the reaction rolls back. The update bundle that initiated the reaction is not re-queued on the network.

PREPARE If all strata have been fully evaluated (the evaluation has reached a fixpoint) and no conflict was detected during evaluation, then the scheduler prepares to lock all reactor instances that are active in the current reaction (except for the initiating reactor, which is effectively already locked). The set of locks to be acquired is represented by the set L.

ACQUIRE If the state of a reactor instance that is to be locked has not changed since its snapshot was taken, we lock the reactor instance by removing it from the externally-visible world (its state is preserved in the reaction state I). Since reactor references are totally ordered (see Section 7.4), we acquire locks in order of reference value to avoid deadlocks.

RETRY Conversely, if the state of a reactor instance that is to be locked has changed since its snapshot was taken, then the reaction rolls back and the update bundle that initiated the reaction is re-queued on the network; the reaction can thus be re-initiated in the future. This rule effectively encodes a form of optimistic transaction.

COMMIT If all reactor instances involved in a reaction were successfully locked, we generate the persistent values of the written relations, release them for future reactions, and add update bundles to the network. This rule also has the effect of "materializing" newly-instantiated reactors created during the reaction.

ANY This rule states that any combination of reactors that can react in isolation can react in any context containing them.

10 Translation properties

This section presents results which show that the program transformation which eliminates negation in head clauses preserves two important properties we care about: safety and stratification. Recall that the rewriting transformation assumes no remote references to other reactor instances are present. We then define what it means for a program before applying the augmentation transformation to be *D-stratified.* Lastly we informally show that the augmentation transformation yields stratifiable augmented programs.

$$\mathrm{active}(I) \triangleq \{s \mid \underline{\texttt{active\_}T}^{\Delta^+}(\alpha) \in I \text{ for some type } T\}$$

$$I \downarrow \alpha \triangleq \{r(\alpha,\mathbf{x}) \mid r(\alpha,\mathbf{x}) \in I,\ \text{where } r \text{ is an arbitrary relation name}\}$$

$$\begin{aligned}\mathrm{init}(I_0, I^{\Delta}) \triangleq I_0 \quad \cup \quad \{\ \texttt{\^{}r}(\mathbf{x}) \mid\ & (\texttt{-r}(\mathbf{x}) \in I_0 \text{ and } \underline{\texttt{r}}\texttt{\^{}}^{\Delta^-}(\mathbf{x}) \notin I^{\Delta})\\ & \text{or} \quad (\underline{\texttt{r}}\texttt{\^{}}^{\Delta^+}(\mathbf{x}) \in I^{\Delta})\ \}\end{aligned}$$

$$\begin{aligned}\mathrm{conflict}(I) \triangleq\ & \text{there exists } \mathbf{x} \text{ such that}\\ & (\underline{\texttt{r}}^{\Delta^+}(\mathbf{x}),\ \underline{\texttt{r}}^{\Delta^-}(\mathbf{x}) \in I)\\ & \text{or} \quad (\underline{\texttt{r}}\texttt{\^{}}^{\Delta^+}(\mathbf{x}),\ \underline{\texttt{r}}\texttt{\^{}}^{\Delta^-}(\mathbf{x}) \in I)\end{aligned}$$

$$\begin{aligned}\mathrm{revert}(S, I) \triangleq \{\ (\alpha, I_\alpha) \mid\ & \alpha \in S,\ I' = I \downarrow \alpha,\\ & I_\alpha = \{\texttt{-r}(\mathbf{x}) \mid \texttt{-r}(\mathbf{x}) \in I'\}\\ \}\ &\end{aligned}$$

$$\begin{aligned}\mathrm{persist}(I) \triangleq \{\ (\alpha, I'') \mid\ & \alpha \in \mathrm{active}(I),\\ & I' = I \downarrow \alpha,\\ & I'' = \{\ \texttt{-r}(\mathbf{x}) \mid \texttt{r} \in \mathit{persistent}(\mathcal{T}),\\ & \qquad\qquad (\ (\texttt{\^{}r}(\mathbf{x}) \in I' \text{ and } \underline{\texttt{r}}^{\Delta^-}(\mathbf{x}) \notin I')\\ & \qquad\qquad\quad \text{or} \quad (\underline{\texttt{r}}^{\Delta^+}(\mathbf{x}) \in I')\ )\ \}\\ \}\ &\end{aligned}$$

$$\begin{aligned}\mathrm{bundles}(I) \triangleq \{\ (\alpha, I^{\Delta}) \mid\ & I' = I \downarrow \alpha,\\ & I^{\Delta} = \{\underline{\texttt{r}}\texttt{\^{}}^{\Delta^+}(\mathbf{x}) \mid \underline{\texttt{r}}\texttt{\^{}}^{\Delta^+}(\mathbf{x}) \in I'\}\\ & \qquad \cup \quad \{\underline{\texttt{r}}\texttt{\^{}}^{\Delta^-}(\mathbf{x}) \mid \underline{\texttt{r}}\texttt{\^{}}^{\Delta^-}(\mathbf{x}) \in I'\}\\ \}\ &\end{aligned}$$

$$\mathrm{uptodate}((\alpha, I_\alpha), I) \triangleq (\texttt{-r}(\mathbf{x}) \in I_\alpha) \iff (\texttt{-r}(\mathbf{x}) \in (I \downarrow \alpha))$$

Fig. 14. Auxiliary functions used in the operational semantics.

Given a Datalog program P with possible negative head clauses, the *rule/relation dependency graph* G is a directed graph $\langle N, R, A\rangle$ with N the set of all predicate symbols in P, R the set of all rules in P, and $a \in A$ either one of the following:

- An edge from $n \in N$ to $r \in R$ if n is a predicate symbol in the body of rule r. An edge between $n \in N$ and $r \in R$ is marked if the body clause that has n as predicate symbol is negative.
- An edge from $r \in R$ to $n \in N$ if n is a predicate symbol in the head of rule r. An edge between $r \in R$ and $n \in N$ is marked if the head clause that has n as predicate symbol is negative.

Definition 4 *A program is* head-negation-stratified *if there exists no cycle in the corresponding rule/relation dependency graph G containing a marked edge.* □

Theorem 5 *If a program with negation in head clauses is head-negation-stratified then the program obtained via the transformation in Fig. 12 is stratified.*

Proof Starting from a dependency graph with no negative cycles, the only way to get a negative cycle is by adding edges—from either extra body clauses or new rules altogether. Adding a negatively marked edge between two nodes already reachable from each other can create a negative loop, but adding a positive edge can also have the same effect if the existing path contains a negatively marked edge. Given that the original dependency graph has no negative cycles, a negative cycle can only be created through the newly created relations, $\underline{\mathtt{r}}_i^{\Delta^+}$ and $\underline{\mathtt{r}}_i^{\Delta^-}$. For a cycle to be created it is necessary that it exist a rule that reads, and another one that writes, a newly created relation.

- Rewrite rules (V) and (VI) write the future states $\underline{\mathtt{r\hat{}}}_i^{\Delta^+}$ and $\underline{\mathtt{r\hat{}}}_i^{\Delta^-}$, but the future state cannot be read, and therefore they cannot create a loop.
- Rewrite rule (I) writes $\underline{\mathtt{r}}_i^{\Delta^+}$ and rewrite rules (III) and (IV) read it.
  Rules (I) and (IV) can create a negative loop only if there also exists another rule *body* ← *head*; the loop would then contain $\underline{\mathtt{r}}_i^{\Delta^+}$, *body*, and *head*. But in this case the original program would not be head-negation-stratifiable as it would have a negative loop between *head*, *body*, and $\mathtt{r}_i$. Therefore rules (I) and (IV) cannot create a negative loop.
  Rules (I) and (III) can create a negative loop only if there also exists a rule of the form `not` $\mathtt{r}_i$ ← *head*; the loop would then contain $\underline{\mathtt{r}}_i^{\Delta^-}$ and *head*. But in this case the original program would not be head-negation-stratifiable as it would have a

negative loop between $\mathtt{r}_i$ and *head*. Therefore rules (I) and (III) cannot create a negative loop.

- Rewrite rule (II) writes $\underline{\mathtt{r}}_i^{\Delta^-}$, and rewrite rules (III) and (IV) read it.

  Rules (II) and (IV) can create a negative loop only if there also exists another rule $\mathtt{r}_i \leftarrow$ *head*; the loop would then contain $\underline{\mathtt{r}}_i^{\Delta^+}$ and *head*. But in this case the original program would not be head-negation-stratifiable as it would have a negative loop between $\mathtt{r}_i$ and *head*. Therefore rules (II) and (IV) cannot create a negative loop.

  Rules (II) and (III) can create a negative loop only if there also exists a rule of the form $\mathtt{not}\ \mathtt{r}_i \leftarrow$ *head*; the loop would then contain $\underline{\mathtt{r}}_i^{\Delta^-}$ and *head*. But in this case the original program would not be head-negation-stratifiable as it would have a negative loop between $\mathtt{r}_i$ and *head*. Therefore rules (II) and (III) cannot create a negative loop.

□

Definition 6 *A rule is* head-negation-safe *if all of its variables are* head-negation-limited. *A variable is head-negation-limited if it occurs as:*

- *an argument to a non-negated user-defined predicate in the body*
- *an argument to a negated user-defined predicate in the body and it is used only in negated head user-defined predicates involving response states*
- *one of the arguments to the built-in equality predicate and all of the other variables that occur in the same clause are limited*

A program is head-negation-safe *if all of its rules are head-negation-safe.* □

Intuitively, the reason for allowing variables occurring in some negated body clause B to appear in the negated head clause H is that the rule has to remove every value occurring in H which is not in B. It does not need to compute the set of all values not occurring in B—which may not be finite.

Theorem 7 *If a program with negation in head clauses is head-negation-safe then the program obtained via the transformation in Fig. 12 is safe.*

Proof We will show that this property holds for each program rule $\mathtt{r}$ that is rewritten.

Rewrite rule (I):

- If $\mathtt{r}$'s *body* has no negative clause then the transformation trivially preserves safety.

- Otherwise the rule does not apply.

Rewrite rule (II):

- If `r`'s *body* has no negative clause: the transformation trivially preserves safety.
- If `r`'s *body* contains at least one negated clause and the variables occurring in this clause are used only in the negated head clause: the rewrite rule transforms the negated head clause into a positive head clause which can use variables occurring only in the original negated body clause. Nevertheless, the rewrite rule also adds a corresponding positive body clause which now bounds the variables used by the head clause. This renders the transformed `r` rule safe.

Rewrite rule (III):

- If `r`'s *body* has no negative clause, the concern is that the variable occurring in the newly added negative clause may not be bounded in a positive body clause but used in the head clause; this is obviously false by construction.
- If `r`'s *body* contains at least one negated clause and the variables occurring in this clause are used only in the negated head clause, the rewrite rule leaves *body* and *head* unmodified and therefore cannot affect the result of the transformation.

Rewrite rule (IV):

- If `r`'s *body* has no negative clause, the concern is that *head* is using variables occurring solely in the negated body clause. This property would then hold for the transformed `r`, which thus would not be safe. Rewrite rule (II) applies now, and will result in a safe rule.
- If `r`'s *body* contains at least one negated clause and the variables occurring in this clause are used only in the negated head clause, the rewrite rule preserves safety up to rule (II).

Rewrite rule (V): The argument is the same as for rewrite rule (I).

Rewrite rule (VI):

- If `r`'s *body* has no negative clause: the transformation trivially preserves safety.
- Otherwise the rule does not apply.

□

Given an initial reactor type C, and given the types of all reactors, we define the *extended rule/relation stratification graph* G on the program before augmentation as a directed graph $\langle N, R, A\rangle$ obtained by applying the following steps repeatedly until the reaction scope has been fully extruded—i.e. no new reactor types can possibly be added to the reaction scope. Let $S = \{C\}$.

- Let C_{crt} iterate over the set of newly added reactor types in S.
- For every rule r in C_{crt} which writes a remote relation p in C_i of a type that already exists in S, treat p as a local relation.
- For every rule in C_{crt} which reads and writes only local relations, build the rule/relation graph as explained in Section 10.1.
- For every rule r in C_{crt} which writes a remote relation p in C_i of a type that does not already exist in S:
  - · Add a node to N for each predicate symbol in C_i, and a node to R for each rule in C_i.
  - · For each rule r' in the current extended stratification graph which read relations in C_i, update A to contain an edge from $n \in N$ corresponding to the relation being read to $r' \in R$ if n is a predicate symbol in the body of rule r'. The edge between $n \in N$ and $r' \in R$ is marked if the body clause that has n as predicate symbol is negative.
  - · For rule r build the rule/relation graph as explained in Section 10.1.
  - · $S = S \cup C_i$.
- For every rule r in C_{crt} which instantiates a new reactor of type C_j that does not already exist in S, $S = S \cup C_j$.

Definition 8 *A set of reactor types is* D-stratified *if there exists no cycle in the corresponding extended stratification graph G containing a marked edge.* □

Note that we are interested in a program to be stratified after the *active_ T* relation has been added, for reasons explained in Section 7.1.

Theorem 9 *If a set of reactor types is D-stratified then the program obtained via augmentaion is head-negation-stratified.*

The augmentation transformation creates a global relation for all corresponding relations p in reactors of the same type. But before augmentation all relations p in reactors of identical type that the rules may refer to were represented by a single node via the graph construction above. Therefore no edges are either created, nor deleted, by the transformation, and the marks on the edges remain unchanged. Practically the augmentation transformation creates a monolithic dependence graph which is isomorphic to the extended stratification graph constructed above. It follows that stratification of the graph is unaffected by the augmentation transformation.

11 Extended Examples

In this section, we complete the exposition of the reactor model with a collection of extended examples. These example show how a number of common distributed programming patterns can be concisely encoded in the reactor model. All of the examples in this section make extensive use of the notational abbreviations defined in Fig. 6.

Example 10 (AJAX-style web form) Figures 15, 16, and 17 depict code for a simple AJAX (asynchronous Javascript and XML) web application, due to Bercik [17]. In this style of application, individual elements on a web page can be updated via asynchronous server requests. In contrast to traditional "page at a time" web applications, this approach minimizes the amount of data that needs to be exchanged between browser and server, which can in turn yield more responsive user interfaces and a reduced load on the server.

Figures 15, 16, and 17 represent browser code, server code, and database code, respectively, for a web form containing fields in which a user can enter a US zip (postal) code, a city name, and a US state name. When the user enters a zip code, a background server request (`XMLHttpRequest`) is dispatched, which ultimately has the effect of looking up the zip code in a database and returning the city and state to which the zip code corresponds back to the browser. The returned values are then used to complete the city and state fields on the web form (the user can overwrite the server-generated field values if desired).

Despite the conceptual simplicity of the web form example, a large amount of code in multiple programming languages is required to build a robust AJAX application. In this case, the browser code (Fig. 15 is written in a combination of HTML and Javascript; the server code (Fig. 16) is written in PHP; SQL queries and data definitions are used to access the database (Figures 16 and 17); and XML is used to transmit data from the server back to the browser (Figures 16 and 15). Specialized Javascript libraries are also needed to dispatch the server request and process the XML response. The result is code that is complex and fragile, even for a very simple application.

By contrast, consider the reactor definitions in Figures 18 and 19. While the reactor model is a foundational model, not a full-blown programming language, it is expressive enough to accurately model all of the essential operations present in the example. Here, reactors are used to model "widgets" representing single form fields (`InputWidgets`), the contents of of the page containing the main web form (`AjaxPage`), server side code (`Server`) and the zip code database (`ZipDB`). In addition, we also model the behavior of a rudimentary web browser (`Browser`). From this example, we see that reactors can compactly and uniformly model the synchronous interaction between the web page code and the browser, the asynchronous interaction between the browser and the server, the synchronous interaction between the server and the database, and the requisite logic used to connect these components together. Contrast the "three-tier" style of this example with the example in Fig. 10, which encapsulates its "business logic" in a single reactor.

Example 11 (Three-tier web application) Fig. 20 depicts another web application which mimics the structure of a conventional three-tier application for catalog ordering. In a manner similar to the previous example, page content in browser is modeled by the `OrderPage` reactor type, which is instantiated with various primitive widget reactors—`OutputWidget` and `ButtonWidget` from Fig. 10 (in the case of `ButtonWidget`, the type of its `buttonListener` relation must be changed from `(ref DataDisplay)` to `(ref OrderPage)` to account for its new "parent" type) and a new `FormWidget`—depending on the type of page being displayed. Instances of `OrderPage` perform local (i.e., "browser-side") computation which performs basic form validation.

One limitation of our current model is that all reactor references must be strongly typed, hence the need to update the declaration of `ButtonWidget` to refer to a different "parent" reactor type, which makes it difficult to model a browser reactor that can render arbitrary pages represented as reactors. In the future, we will consider more flexible type system which would allow a completely generic browser, oblivious to the type of the page, to be defined.

Example 12 (Small workflow system) Consider a workflow system (shown in Fig. 21) where employees handle incoming cases (say, problem reports) and attempt to resolve them. Regardless if a case is resolved or not, it is eventually archived to signify that no more work should be carried out on that case. Of course, it can be important to know the entire history of unresolved cases whether they are archived or not. To this end, the system outlined in Fig. 21 declares the relations `inbox`, `unresolved`, and `archive`. Cases received from the outside world through the `inbox` are initially duplicated to `unresolved` and eventually put in `archive` although they may continue to exist in `unresolved`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
<title>ZIP Code to City and State using XmlHttpRequest</title>
<script language="javascript"  type="text/javascript">
var url = "getCityState.php?param="; // The server-side script
function handleHttpResponse() {
  if (http.readyState == 4) {
    if (http.responseText.indexOf(’invalid’) == -1) {
      // Use the XML DOM to unpack the city and state data
      var xmlDocument = http.responseXML;
      var city = xmlDocument.getElementsByTagName(’city’).item(0).firstChild.data;
      var state = xmlDocument.getElementsByTagName(’state’).item(0).firstChild.data;
      document.getElementById(’city’).value = city;
      document.getElementById(’state’).value = state;
      isWorking = false;
    }
  }
}
var isWorking = false;
function updateCityState() {
  if (!isWorking && http) {
    var zipValue = document.getElementById("zip").value;
    http.open("GET", url + escape(zipValue), true);
    http.onreadystatechange = handleHttpResponse;
    isWorking = true;
    http.send(null);
  }
}
function getHTTPObject() {
  var xmlhttp;

  if (!xmlhttp && typeof XMLHttpRequest != ’undefined’) {
    try {
      xmlhttp = new XMLHttpRequest();
          xmlhttp.overrideMimeType("text/xml");
    } catch (e) {
      xmlhttp = false;
    }
  }
  return xmlhttp;
}
var http = getHTTPObject();
</script>
</head>
<body>
<form action="post">
  <p>
  ZIP code:
  <input type="text" size="5" name="zip" id="zip" onblur="updateCityState();" />
  </p>
  City:
  <input type="text" name="city" id="city" />
  State:
  <input type="text" size="2" name="state" id="state" />
</form>
</body>
</html>
```

Fig. 15. AJAX address lookup form: HTML and JavaScript client code.

```
<?php
/**
 * Connects to the database.
 */
function db_connect() {
  $database_name = ’mysql’;
  $database_username = ’root’;
  $database_password = ’’;
  $result = mysql_pconnect(’localhost’,$database_username, $database_password);
  if (!$result) return false;
  if (!mysql_select_db($database_name)) return false;
  return $result;
}
$conn = db_connect(); // Connect to database
if ($conn) {
  $zipcode = $_GET[’param’]; // The parameter passed to us
  $query = "select * from zipcodes where zipcode = ’$zipcode’";
  $result = mysql_query($query,$conn);
  $count = mysql_num_rows($result);
  if ($count > 0) {
    $city = mysql_result($result,0,’city’);
        $state = mysql_result($result,0,’state’);
  }
}
if (isset($city) && isset($state)) {
  // $return_value = $city . "," . $state;
  $return_value =
    ’<?xml version="1.0" standalone="yes"?><zip><city>’.
      $city.’</city><state>’.$state.’</state></zip>’;
}
else {
  $return_value = "invalid".",".$_GET[’param’]; // Include Zip for debugging purposes
}
header(’Content-Type: text/xml’);
echo $return_value; // This will become the response value for the XMLHttpRequest object
?>
```

Fig. 16. AJAX address lookup form: PHP server code.

```
CREATE TABLE ‘zipcodes‘ (
  ‘zipcode‘ mediumint(9) NOT NULL default ’0’,
  ‘city‘ tinytext NOT NULL,
  ‘state‘ char(2) NOT NULL default ’’,
  ‘areacode‘ smallint(6) NOT NULL default ’0’,
  PRIMARY KEY  (‘zipcode‘),
  UNIQUE KEY ‘zipcode_2‘ (‘zipcode‘),
  KEY ‘zipcode‘ (‘zipcode‘)
) TYPE=MyISAM;
```

Fig. 17. AJAX address loookup form: database code.

Incoming cases are allocated to employees who each has a work queue. Employees interact with the system through a GUI where they can mark cases resolved, archived or deallocated. These GUI events are handled through the public relation `GUIevent`.

```
def InputWidget = { // Reactor representing single input field in web form

  public size: (int).              // fixed field width, if specified
  public label: (string).          // field label
  public val: (string).            // value entered in field
  public blurListener:
    (ref Client).                  // listener(s) for blur event (change of focus)
}

def AjaxPage = { // Reactor representing composite web page for address lookup

  public rServer: (ref Server).    // ref. to server; initialized when page is created
  public rBrowser: (ref Browser).  // ref. to browser; initialized when page
                                   //   received by browser

  title: (string).                 // page title
  rZip: (ref InputWidget).         // widgets for zip, city, state
  rCity: (ref InputWidget).
  rState: (ref InputWidget).

  public ephemeral write           // positive response from server to lookup request
    zipFound: (string, string).
  public ephemeral write           // negative response from server to lookup request
    zipNotFound: (string).
  waiting: ().                     // flag set when zip lookup request in progress

  onBlur(_), not -waiting(): {     // when notified of blur event for zip widget and not
    s.getZip^(self,zs) <-          //   awaiting a prev. response, request city/state
      z.val(zs), rZip(z),          //   corresponding to zip
      rServer(s).
    waiting() <- .                 // set flag while awaiting response
  }

  zipFound(cs,ss): {               // success case: copy city/state values returned
    c.val(cs) <- rCity(c).         //   to corresponding widgets and reset waiting
    s.val(ss) <- rState(s).        //   flag
    not waitingForZip() <- .
  }

  not waitingForZip() <-           // failure case (no city/state pair found): just
    zipNotFound(_).                //   reset waiting flag

  INIT: {                          // initialize new child widgets when parent page
    rZip(new) <- .                 //   is created
    rZip(z) : {
      z.label("ZIP Code:") <- .
      z.size(5) <- .
      z.blurListener(self) <- .   // this reactor listens for blur events on zip widget
    }
    rCity(new) <- .
    c.label("City:") <- rCity(c).
    rState(new) <- .
    rState(s): { s.label("State:") <- . s.size(2) <- . }
    title("ZIP Code to City and State using the Reactor Model")
  }
}

def Browser = { // Reactor representing web browser

  currPage: (ref AjaxPage).        // current page in browser
  public ephemeral write           // response from getPage request
    showPage: (ref AjaxPage).

  currPage(p) := showPage(p).      // response from server to page request updates
  p.rBrowser(self) <-              //   current browser page and sets browser ref.
    showPage(p).                   //   in page to the browser
}
```

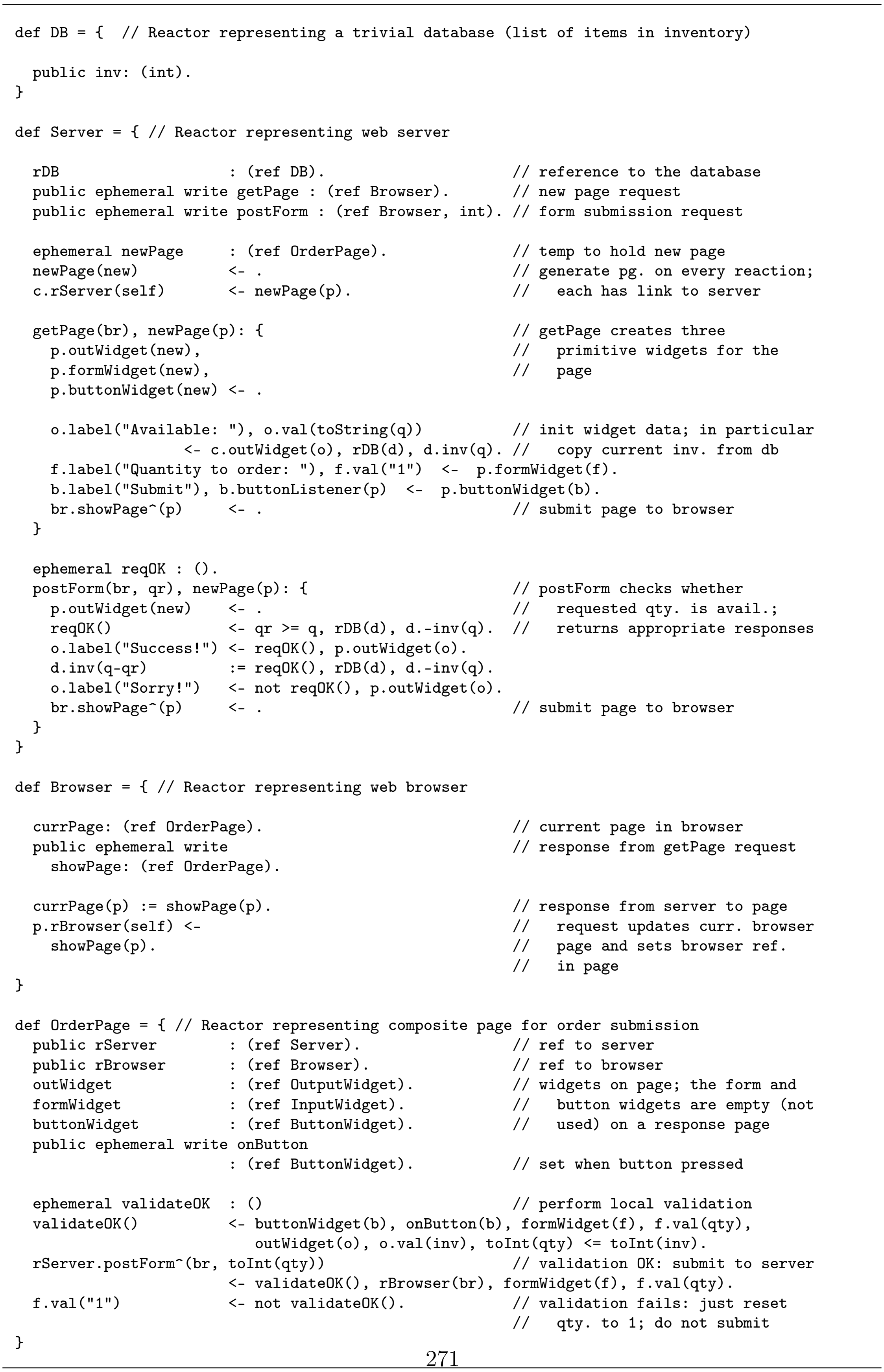
Fig. 18. Reactor-based address lookup form: “client”-related definitions.

```
def Server = { // Reactor representing web server

  public rDB: (ref ZipDB).         // ref. to zip database (assumed initialized elsewhere)
  public ephemeral write           // corresponds to HTTP GET of main page
    getPage: (ref Browser).
  public ephemeral write           // correponds to HTTP GET for zip lookup
    getZip: (ref Browser).

  ephemeral newPage:               // temporary to hold newly-created main page
    (ref AjaxPage).
  getPage(b): {                    // respond to getPage request
    newPage(new) <- .              // instantiate new page
    newPage(p): {                  // initialize server ref. in new page; return
      p.rServer(self) <- .         //   page to browser
      b.showPage^(p) <- .
    }
  }

  ephemeral lookupRes:             // temporary to hold result of zip lookup in DB
    (string, string).
  getZip(b,zs): {                  // respond to getZip request
    lookupRes(cs,ss) <- d.addr(zs,cs,ss), rDB(d).
    b.zipFound^(cs,ss) <- lookupRes(cs,ss).
    b.zipNotFound^(zs) <- not lookupRes(_,_).
  }
 }

def ZipDB = { // Reactor representing zip code / address database

  public addr: (string, string, string, string).     // zip, city, state, area code

  FAIL <- addr(z,c1,_,_), addr(z,c2,_,_), c1 <> c2.  // rules which together assert that
  FAIL <- addr(z,_,s1,_), addr(z,_,s2,_), s1 <> s2.  //   zip is primary key
  FAIL <- addr(z,_,_,a1), addr(z,_,_,a2), a1 <> a2.
}
```

Fig. 19. Reactor-based address lookup form: “server”-related definitions.

In the normal workflow employees can navigate freely between the various states, as long as two rules are satisfied: (1) every case that has not been archived must be allocated, and (2) any resolved item should be archived (although it will stay in the work queue for post-review until the employee explicitly deallocates it). These two rules are encoded by the lines marked `(1)` and `(2)` in the example. The two rules demonstrate a desirable coding style, where the constraints are separate from the GUI handling and any other rules that may modify the relations; this improves compositionality with future rules.

Example 13 (Aspect-like modeling of logging and access control) The reactor code in Figures 22 and 23 is based on an example due to Hankin et al. [18]. Fig. 22 depicts a simple application in which a `RequestMgr` reactor responds to requests by telephone company customers for billing information (via the ephemeral relation `getBill`). The billing information is derived by combining information



```
def DB = {  // Reactor representing a trivial database (list of items in inventory)

  public inv: (int).
}

def Server = { // Reactor representing web server

  rDB                   : (ref DB).                     // reference to the database
  public ephemeral write getPage : (ref Browser).       // new page request
  public ephemeral write postForm : (ref Browser, int). // form submission request

  ephemeral newPage     : (ref OrderPage).              // temp to hold new page
  newPage(new)          <- .                            // generate pg. on every reaction;
  c.rServer(self)       <- newPage(p).                  //   each has link to server

  getPage(br), newPage(p): {                            // getPage creates three
    p.outWidget(new),                                   //   primitive widgets for the
    p.formWidget(new),                                  //   page
    p.buttonWidget(new) <- .

    o.label("Available: "), o.val(toString(q))          // init widget data; in particular
                   <- c.outWidget(o), rDB(d), d.inv(q). //   copy current inv. from db
    f.label("Quantity to order: "), f.val("1")  <-  p.formWidget(f).
    b.label("Submit"), b.buttonListener(p)  <-  p.buttonWidget(b).
    br.showPage^(p)     <- .                            // submit page to browser
  }

  ephemeral reqOK : ().
  postForm(br, qr), newPage(p): {                       // postForm checks whether
    p.outWidget(new)    <- .                            //   requested qty. is avail.;
    reqOK()             <- qr >= q, rDB(d), d.-inv(q).  //   returns appropriate responses
    o.label("Success!") <- reqOK(), p.outWidget(o).
    d.inv(q-qr)         := reqOK(), rDB(d), d.-inv(q).
    o.label("Sorry!")   <- not reqOK(), p.outWidget(o).
    br.showPage^(p)     <- .                            // submit page to browser
  }
}

def Browser = { // Reactor representing web browser

  currPage: (ref OrderPage).                            // current page in browser
  public ephemeral write                                // response from getPage request
    showPage: (ref OrderPage).

  currPage(p) := showPage(p).                           // response from server to page
  p.rBrowser(self) <-                                   //   request updates curr. browser
    showPage(p).                                        //   page and sets browser ref.
                                                        //   in page
}

def OrderPage = { // Reactor representing composite page for order submission
  public rServer        : (ref Server).                 // ref to server
  public rBrowser       : (ref Browser).                // ref to browser
  outWidget             : (ref OutputWidget).           // widgets on page; the form and
  formWidget            : (ref InputWidget).            //   button widgets are empty (not
  buttonWidget          : (ref ButtonWidget).           //   used) on a response page
  public ephemeral write onButton
                        : (ref ButtonWidget).           // set when button pressed

  ephemeral validateOK  : ()                            // perform local validation
  validateOK()          <- buttonWidget(b), onButton(b), formWidget(f), f.val(qty),
                           outWidget(o), o.val(inv), toInt(qty) <= toInt(inv).
  rServer.postForm^(br, toInt(qty))                     // validation OK: submit to server
                        <- validateOK(), rBrowser(br), formWidget(f), f.val(qty).
  f.val("1")            <- not validateOK().            // validation fails: just reset
                                                        //   qty. to 1; do not submit
}
```

Fig. 20. Mini three-tier web application.

```
def WorkflowSystem = {
// Workflow system for case handling

public ephemeral inbox    : (string)            // (case)
public ephemeral GUIevent : (string, string,    // (case, employee,
                             string)            //  {"Resolve","Archive","Dealloc"})

employee                  : (string)            // (employee)
workQueue                 : (string, string)    // (case, employee)
archive                   : (string)            // (case)
unresolved                : (string)            // (case)


// Normal workflow

unresolved^(case)   <-  inbox (case)                              // Put in unresolved
workQueue^ (case,p) <-  not archive (case), unresolved (case),    // (1) Allocate
                        employee (p)
archive^ (case)     <-  workQueue (case,p), not unresolved (case) // (2) Archive resolved


// GUI events

not unresolved (case)  <-  GUIevent (case, _, "Resolve")          // Case resolved
archive (case)         <-  GUIevent (case, _, "Archive")          // Archive case
not workQueue (case,p) <-  GUIevent (case, p, "Dealloc")          // Remove from queue
}
```

Fig. 21. Example from a workflow system for case handling

in two databases: WP, a database of “white pages” listings, and BillDB, a billing database.

In Fig. 23, we augment the definition of RequestMgr in Fig. 22 with additional rules which add logging and access control functionality. The last two rules of the example encode assertions representing alternative access control policies. These rules have the effect of rolling back reactions which violate the policies. Hankin et al. [18] implement the basic billing functionality of Fig. 22 in the KLAIM process calculus [19], then add access control facilities using aspect-like [11] extensions to the core calculus. The aspect extensions can modify the behavior of the underlying application by inserting or removing expressions of the original program (in this case, by inserting code around database accesses which validates access control policies).

As with aspect-like approaches, the reactor model allows many “additive” behaviors, such as the new functionality of Fig. 23, to be incorporated into an existing program as additional rules, without requiring that existing code be modified. On the other hand, aspect languages typically provide meta-constructs to define “pointcuts” at which new behavior is introduced via syntactic pattern-matching; the reactor model provides no analogous functionality. Existing functionality in a

```
def Cust = { // Reactor representing a customer

  ...                              // customer name, address, etc.
  public ephemeral write           // response to request for billling info.
    bill: (int).
}

def WP = { // Reactor representing "white pages" database

  public dir: (ref User, string).  // customer, phone number
}

def BillDB = { // Reactor representing billing information

  public billing:                  // phone number, cost of call, customer
    (string, int, ref User).
}

def RequestMgr = { // Reactor managing requests to phone-related databases

  rWP: (ref WP).                   // refs. to white pages and billing
  rDB: (ref BillDB).               //   databases; assumed initialized elsewhere

  auth: (ref Cust, ref BillDB).    // authorization matrix: auth(c, d) means
                                   //   that customer c is authorized to
                                   //   access database d

  public ephemeral write getBill:  // getBill(r, c): requester r requests
    (ref Cust, ref Cust).          //   billing data for customer c
  r.bill^(a) <-
    getBill(r, c), y.dir(c, n), d.billing(n, a, c), rWP(y), rDB(d).
}
```

Fig. 22. Telephone records management system.

reactor program can generally be altered or removed only by editing rules in place. By contrast, [18] allow "advice" to be defined which has the effect of conditionally eliminating existing behaviors. Nevertheless, we believe that the rule-based style of the reactor model usually allows conceptually independent behaviors to be specified as distinct collections of rules, which in turn means that the scope of required edits is minimized when application requirements change.

Example 14 (Refactoring from synchronous to asynchronous interaction) The examples in Figures 24 and 25 both implement the functionality of the example in Fig. 8, which we considered earlier. Reactor `SyncSample` below is an entirely synchronous variant of the sensor sampling reactor `Sample` of Fig. 8. Reactor `AsyncSample` of Fig. 25 is rather more interesting. Like `Sample`, it is asynchronous. However, `AsyncSample` does not require that either the sensor or the sampling reactor handle asynchronous requests and responses. Instead, we use an auxiliary "agent" reactor, `SampleAgent`, which first reacts synchronously with `SyncSensor`, then (in a separate reaction) reacts synchronously with `AsyncSample`. Using such auxiliary agents, we can refactor synchronous reactor interaction to asynchronous

```
def RequestMgr = { // Reactor managing requests to phone-related databases

  rWP: (ref WP).
  rDB: (ref BillDB).
  auth: (ref Cust, ref BillDB).

  public ephemeral write getBill: (ref Cust, ref Cust).
  r.bill^(a) <-
    getBill(r, c), y.dir(c, n), d.billing(n, a, c), rWP(y), rDB(d).

  log: (ref Cust, ref Cust)        // log accesses
  log(r, c) <- getBill(r, c).

  FAIL <- getBill(r, c),           // ...and/or disallow unauthorized access
    not auth(r, d), rDB(d).

  FAIL <- getBill(r, c), r <> c,   // ...or use a more sophisticated policy: requester
    not auth(r, d), rDB(d).        //   is allowed to request information about
                                   //   self; otherwise, must be explicitly  authorized
}
```

Fig. 23. Telephone records management system augmented with logging/authorization functionality.

```
def SyncSensor = {
  public val: (int)
}

def SampleData = {
  // data collected to date; nonce used to
  // distinguish multiple measurements of same
value
  public log: (ref Nonce, int).
}
```

```
def SyncSample = {
  public rSensor: (ref SyncSensor).
  public rSampleData: (ref SampleData).
  public write ephemeral pulse: ().

  // add current value of sensor to log
  c.log(new, v) <-
    pulse(), rSensor(s),
    rSampleData(c), s.val(v).
}
```

Fig. 24. Fully synchronous sensor sampling.

interaction with minimal change to the original reactor code.

Example 15 (Business process for order fulfillment) The example in Fig. 26 depicts a simple business process example modeling order fulfillment. In this example, a `Gateway` reactor processes requests to fulfill a certain quantity of items (in this example, only one type of item is considered, for simplicity). The `Gateway` maintains an ordered list of references to `Warehouse` reactors which are capable of fulfilling the order. When a request is received, a new `Processor` reactor is spawned specifically to fulfill the (single) request. The `Processor` reactor asynchronously queries each warehouse on the list to determine if it can fulfill the order itself; if not, the remainder of the order is passed on to the next warehouse on the list for fulfillment, and so on. If all of the requested items can be supplied, an asynchronous response is sent to the requester, indicating success. Otherwise, a failure response, containing the number of items unfulfilled, is dispatched. The

```
def AsyncSample = {
  public rSensor: (ref SyncSensor).
  public rSampleData: (ref SampleData).
  public write ephemeral pulse: ().
  ephemeral newAgt:(ref SampleAgent).

  // delegates functionality to reactor
  // of type SampleAgent
  pulse(): {
    newAgt(new) <- .
    a.rSensor(s), a.rSampleData(c) <-
      rSensor(s), rSampleData(c).
  }
}
```

```
def SampleAgent = {
  // implements state machine that first
  // gets sample, then writes to log
  public rSensor: (ref SyncSensor).
  public rSampleData: (ref SampleData).
  ephemeral readSample: ().
  ephemeral writeLog: (int).

  // sends an async update to itself
  // when created
  getSample^() <- not -live().

  // reads sensor, sends async update
  // to self with the value read
  writeLog^(v) <-
    getSample(), rSensor(s), s.val(v).

  // finally, writes the sensor value to log
  c.log(new, v) <-
    writeLog(v), rSampleData(c).
}
```

Fig. 25. Fully asynchronous sensor sampling.

`FIRST` and `EMPTY` expressions are syntactic for rules and auxiliary relations which compute the first element of a list, and test a list for emptiness, respectively.

This example in Fig. 26 illustrates how auxiliary reactors can be instantiated that carry out independent "task threads", in a manner similar to process replication in standard process calculi. The example also illustrates how the data manipulation components of the language (here, e.g., the machinery used to manage lists of warehouses) interact naturally with process creation and inter-process communication.

Example 16 (Views defined by rules: shopping cart management) The example in Fig. 27 models shopping cart management for a catalog order application. Reactor type `CartManager` is intended to manage a shopping cart for a single user interaction ("session"). Each `CartManager` instance contains a "public" view of private cart data maintained on behalf of *all* users by reactor `DB`. Relation `currCart` contains the public contents of a single user cart, and `allCarts` maintains the private contents of all shopping carts currently managed in the system, along with auxiliary information about users and shipping information for each cart. The client interacts with the public cart (`currCart`) by reading and writing its contents directly. The rules of `CartManager` synchronize the public contents of `currCart` with private internal data; hence `currCart` functions as a *view*, in the database sense, of the private `DB` data. Note that the shipping information is maintained by `DB`, rather than `CartManager` instances. The `CartManager` rules are concerned only with synchronizing the public and private cart data. As before, `EMPTY` is syntactic sugar for auxiliary rules/relations which test a relation for emptiness. `SUM` is sugar

```
def Gateway = {
  // requests are pairs of the form
  // (requester, quantity requested)
  public ephemeral request:
    (ref Requester, int).

  // ordered list of warehouse refs which may
  // be used to fulfill requests;
  // lists are sets of tuples of the form
  // { <l1, l2>,<l2, l3>,...<ln, ln> }
  public whs:
    (ref Warehouse, ref Warehouse).

  // auxiliary reactor spawned to
  // process each request
  ephemeral processor: (ref Processor).

  // spawn reactor for request
  processor(new) <- request(_,_).

  // initialize persistent relations
  // of request processor
  p.whs(w1, w2),
  p.requester(r),
  p.remaining(q) <-
    whs(w1, w2), request(r, q).
}
```

```
def Processor = {
  // requester
  public requester: (ref Requester).
  // quantity of items left to fulfill
  public remaining: (int).
  // warehouses ramining to be queried
  public whs: (ref Warehouse, ref Warehouse).
  // quantity of items last warehouse
  // queried is willing to supply
  public ephemeral write willSupply: (int).

  // the following rules apply both when the
  // processor is initialized and after a
  // warehouse query returns
  remaining(q): {
    // quantity left to fulfill is nonzero:
    // query first warehouse on list
    w.request^(q) <-
      q > 0, FIRST[whs(_,_)](w).
    // qty. left to fullfill zero:
    // respond positively
    r.complete^() <- q = 0.
    // no warehouses left, qty. remaining
    // nonzero, respond with unfullfilled amt.
    r.incomplete^(q) <-
      q > 0, EMPTY[whs(_,_)].
  }

  willSupply(qw): {
    // decrement quantity supplied from
    // amount remaining to fulfill
    // (qty. to be supplied is assumed less
    // than or equal to qty. requested)
    remaining(q - qw) := -remaining(q).
    // remove first warehouse from list
    // when response received
    not whs(w1, w2) <- -whs(w1, w2),
      FIRST[-whs(_,_)](w1).
  }
}
```

Fig. 26. Order fulfillment workflow.

for rules/relations which compute the sum of the quantity values for each item in a cart.

The cart synchronization rules must account for various edge conditions: e.g., when there is a pre-existing private cart that can be used to populate the public cart, and when there is no pre-existing cart at all. This is quite straightforward to do by composing rules that separately handle each edge case, but would likely require much more thought and care if written in a non-declarative style. Similarly, the rule-based computation allows the distinct "concerns" of cart synchronization and shipping logic to be specified and managed independently, in an aspect-like manner.

```
def CartManager = {
  // singleton relation containing current userid
  public userid: (string).
  // public "view" of cart for clients
  public currCart: (int, int)
  // reference to persistent data
  rDB: (ref DB).

  userid(u), rDB(db), db.userCarts(u, n): {
    // when cart manager first created,
    // initialize currCart with allCarts
    // entry for user, if any exists
    currCart(i, q) <-
      INIT, -db.allCarts(n, i, q).

    // if no cart exists, create one
    ephemeral preExistingCart: (string).
    preExistingCart(u) <- -db.userCarts(u, _).
    db.userCarts(u, new) <-
      EMPTY[preExistingCart(u)].

    // all currCart entries must
    // also be in allCarts
    db.allCarts(n, i, q) <-
      currCart(i, q), db.allCarts
    // if qty. changed, remove old qty.
    not db.allCarts(n, i, q) <-
      currCart(i, q'),
      -db.allCarts(n, i, q), q <> q'.
    // item not in currCart:
    // delete from allCarts
    not db.allCarts(n, i, _) <-
      not currCart(i, _).
  }
}
```

```
def DB = {
  // maps userids to cartids
  userCarts: (string, ref Nonce).
  // maps cartids to items, qty
  allCarts: (ref Nonce, int, int).
  // carts with standard shipping
  standardShipping: (ref Nonce).
  // carts with free shipping
  freeShipping: (ref Nonce).

  // std. shipping if qty. < 10, free o.w.
  standardShipping(n) <-
    SUM[n, q, allCarts(n, _, q)](n,s),
    s <= 10.
  freeShipping(n) <-
    SUM[n, q, allCarts(n, _, q)](n,s),
    s > 10.
}
```

Fig. 27. Shopping cart management.

12 Related work

Fundamentally, reactors are "reactive systems" [20], combining and extending features from several, largely unrelated areas of research: synchronous languages, Datalog [4], and the actor model [5].

Esterel [1], Lustre [2], Signal [3], and Argos [21] are prominent synchronous languages. In synchronous languages, the term *causality* refers to dependencies, and all have restrictions on cyclic dependencies. Esterel only admits a program if all signals can be inferred to be either present or absent (as opposed to unknown); this is referred to as *constructiveness*. Esterel adopts a strict interleaving semantics, i.e. it assumes that reactions cannot overlap temporally. In Esterel signals are broadcast instantaneously so that all receptors of the signal will see it in the same instant and the signal will only exist in that reaction. The reactor model, on the other hand, supports both synchronous and asynchronous broadcasts (readers can react when a relation is changed) as well as synchronous and asynchronous point-to-point communication (by writing directly into a public relation of the receiver).

Lustre and Signal also limit cyclic dependencies, but add *sampling* in the form of the construct `x = Exp when BExp` meaning that `Exp` should be evaluated only when `BExp` is true. This facility provides a sophisticated way of reading values from preceding reactions other than the immediately previous one. In the reactor model, such predicates can be expressed directly as `x(Exp) <- BExp` where `x` should be a singleton relation. Argos is based on State Charts and hierarchical automata and distinguishes itself from other synchronous languages by being graphical.

Generally speaking, the group of synchronous languages does not allow cycles in the data flow graph – only pre-state to response-state connections are permitted when referring to the same variable. In the reactor model, stratification provides a more refined classification that widely allows recursion while ruling out cases where the fixed point could be ambiguous (of course, programs may still loop infinitely). Reactors provide several features not found in synchronous languages, namely asynchrony, generativity, and distributed transactions. We are not familiar with any other language that combines these features.

Active databases [22] commonly express triggers of the form *Event–Condition–Action* (ECA), where the action is carried out if on receipt of a matching event the condition holds true. This can be expressed as `action <- event, condition` in the reactor model. The reactor model eliminates the distinction between conditions and events, and adds support for distribution, process generation, and synchronous composition.

Transaction Datalog [23] introduces transactions and database updates to Datalog. In Transaction Datalog, inserts and deletes are special atoms in rule bodies, and backward derivation rather than forward derivation is used. To achieve concurrency in transactions a concurrent conjunction operator, `|`, is added. In the reactor model, all rules execute concurrently within the same reaction (subject to stratification) by default, and thus *sequentiality*, rather than concurrency, must be programmed explicitly when needed. We feel this increases the opportunity for aggressive parallelization. Another piece of work on Datalog with updates is DatalogU [24], which follow a similar approach to Transaction Datalog, but without transactions. DatalogU proposes a semantics in which deletions are performed before insertions so that newly inserted facts cannot be deleted immediately. The reactor model augments the features found in these Datalog variants with a framework for distribution, communication, and generativity (the ability to create new reactors), and allows deletion to happen concurrently with the rest of the computation.

In [25] an approach to deletion is presented which focuses only on modifying the extensional database. The semantics of rules with updates in the head is given by re-writing them into equivalent update-free rules that are XY-stratified—a special subcase of locally stratified programs. These programs can express all programs under inflationary fixpoint semantics. Deletions (and additions) in this approach occur as additional commands, separate from the program, which request the removal (or addition) of extensional facts. In this context, a deletion request is handled by simply not copying a fact to the database.

U-Datalog [26] has an non-immediate update semantics, in which updates are collected and applied in parallel to the database when the query evaluation is completed. This approach lacks what we think of a nice property that when a reaction quiesces all the rules have been re-satisfied.

In [27], rules which update the database by retracting tuples are regarded as integrity constraints, and programs with rules containing retractions are translated into normal programs. Consistency is restored by making one of the positive literals in the rule body false. This can be updates to the database, database initial facts, or the facts inferred from the rules. The semantic is nondeterministic.

In [28] procedural and declarative update languages are proposed which all have *saturation* semantics—a form of inflationary semantics. This means that facts can be added to the database, but never deleted. [29] defines Datalog-A, an extension of Datalog with updates; this approach promotes explicitly procedural constructs and uses a Dynamic Logic to specify their semantics.

The constructive semantics of LDL [30] for Datalog programs with updates is top-down. The declarative semantics of LDL programs imposes an *update dependency* restriction on programs. If p is the head predicate symbol of a rule s.t. the definition of p contains an update predicate (addition or deletion) p has the *update dependency property*. A set of rules is legal if every head predicate symbol that has the update dependency property is defined by a single rule. In the case in which the order of applying the updates is relevant, the program is rejected or the user can specify procedurally the order of execution.

KLAIM [31] is a process calculus type of programming language based on Linda. It has asynchronous communication based on creating tuples which will be received by any number of processes. There is a shared data space, though multiple located tuple spaces can exist at a time. The language is procedural in the sense that it expreses implementation as opposed to specifications, and operations are ordered.

Orc [32] is an orchestration language for distributed computation. The composition operators for services are parallel composition, sequencing, and selective pruning. The model does not have any computational power; it relies on services to provide this capability. The basic semantics is asynchronous, but it also provides a particular flavor of synchronous semantics which ensures that all internal events are processed as soon as possible, before any external response can be processed.

TCC [33] presents a declarative approach that starts from an asynchronous computational model (CCP) and ensures a timed computationl model in which computation is deterministic and time is bounded. The language supports hierarchical and modular construction of applications via parallel composition and pre-emption. To express time-out TCC introduces the notion of negative information. The absence of some positive information is interpreted as negative information solely at a point where the computation has quiesced, and can only trigger more activity in the next computation interval.

Aspect-oriented programming [11] addresses the separation and composition of orthogonal, but interacting, concerns of a program. Reactors do not have the features usually identified with aspects (point-cuts, advice, etc.), but instead provide one general mechanism for composition of rules in a non-intrusive and well-defined manner.

Other approaches to Internet application programming include Flapjax [34] (an event-stream and message library built on top of JavaScript), and Links [35] (a code generation approach, where the code for several tiers is generated from one source).

What differentiates us from the existing approaches beyond the technical details is a uniform programming model which synthesizes concepts from several domains without the usual complexity associated with this process. The semantic and syntactic machinery for capturing all these features in a single programming model mostly consists in minimal extensions to the standard Datalog semantics.

13 Future work

While this paper has not focused on implementation, there are two broad areas that are amenable to optimization: query incrementalization, and efficient implementation of synchronous composite reactions through low-overhead concurrency control. The former has already been studied in the Datalog community (e.g., [36]), and we intend to adapt those results appropriately to our setting. In the case of synchronous reactions, recent results on efficient implementation of software transactions (e.g., [37]) are likely to be relevant.

Other issues we plan to investigate include: (1) contract/interface type systems; (2) various abstraction facilities, such as reactor and rule parametricity and high-order rules, that read, write, and deploy other rules; (3) more sophisticated access control mechanisms; (4) function symbols (functors); (5) reactor garbage collection; (6) a truly distributed implementation; (7) support for long-running (rather than atomic) transactions.

Our semantic account of the reactor model has treated a collection of reactor types as defining an augmented program, whose evaluation is orchestrated by the rules defined in Section 9. However, it is neither necessary nor desirable in a practical system to require that the set of all reactor types be translated into a single program. It it only necessary that each reactor be aware of the other reactors that it interacts with within a given reaction. A truly distributed implementation will build the stratification graph and evaluate the rules incrementally. The following issues would need to be addressed in a practical distributed implementation of the reactor model. Given an initiating reactor which selects an update bundle off the network, we would like it to be the case that:

- The initiating reactor is responsible for defining the scope of the reaction by including all, and only, the reactor types reachable from it via writing a remote relation, or instantiating a new reactor type. The augmentation transformation only applies to the set of reactor types in the reaction scope. Likewise, the transformation which removes head-clause negation applies to the set of augmented reactor types in the reaction scope instead of the augmented program reflecting the set of all reactor types. We must extend the definition of head-negation-stratified and stratified programs to account for sets of reactor types rather than just augmented programs.
- The initiating reactor is responsible for the evaluation of all, and only, the rules in the reaction scope. This distributed evaluator evaluates rules in the same order as the evaluator presented in Section 5.5. Also, no rule in the distributed evaluator is executed until all instances of tuples for the rule's relation instances have been "loaded" into the evaluator's relations.

Acknowledgments. The authors gratefully acknowledge the contributions of Rafah Hosn, Bruce Lucas, James Rumbaugh, Mark Wegman, and Charles Wiecha to the development of the ideas embodied in this work.

References

[1] G. Berry, G. Gonthier, The ESTEREL synchronous programming language: design, semantics, implementation, Science of Computer Programming 19 (2) (1992) 87–152.

[2] P. Caspi, D. Pilaud, N. Halbwachs, J. A. Plaice, LUSTRE: a declarative language for real-time programming, in: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press, New York, NY, USA, 1987, pp. 178–188.

[3] T. Gautier, P. L. Guernic, SIGNAL: A declarative language for synchronous programming of real-time systems, in: Proceedings of the Functional Programming Languages and Computer Architecture, Springer-Verlag, London, UK, 1987, pp. 257–277.

[4] J. D. Ullman, Principles of Database and Knowledge-Base Systems, Vol. 1, Computer Science Press, 1988, Ch. 3.

[5] G. A. Agha, I. A. Mason, S. F. Smith, C. L. Talcott, A foundation for actor computation, Journal of Functional Programming 7 (1) (1997) 1–69.

[6] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, parts I-II, Information and Computation 100 (1) (1992) 1–77.

[7] R. T. Fielding, Architectural styles and the design of network-based software architectures, Ph.D. thesis, University of California, Irvine (2000).

[8] J. Field, M.-C. V. Marinescu, C. Stefansen, Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications, in: Ninth Intl. Conf. on Coordination Models and Languages, Vol. 4467 of Lecture Notes in Computer Science, Springer-Verlag, Paphos, Cyprus, 2007, pp. 76–95.

[9] R. Topor, Safe database queries with arithmetic relations, in: Proceedings of the 14th Australian Computer Science Conference, 1991.

[10] X. Wang, Negation in logic and deductive databases, Ph.D. thesis, University of Leeds (1999).

[11] G. Kiczales, Aspect-oriented programming, ACM Computing Surveys 28 (4es) (1996) 154.

[12] T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall, 2005.

[13] S. Abiteboul, R. Hull, V.Vianu, Foundations of Databases, Addison Wesley, 1995.

[14] A. V. Gelder, Negation as failure using tight derivations for general logic programs, in: Proc. Symp. on Logic Programming, 1986, pp. 127–139.

[15] A. W. K.R. Apt, H.A. Blair, Towards a theory of declarative knowledge, in: Foundations of Dedective Databases and Logic Programming, Morgan Kaufmann, 1988, pp. 89–148.

[16] S. Naqvi, Negation as failure for first-order queries, in: Proc. Fifth ACM Symposium on Principles of Database Systems, 1986, pp. 114–122.

[17] B. Bercik, Guide to using ajax and xmlhttprequest (2005).
URL `http://www.webpasties.com/xmlHttpRequest/index.html`

[18] C. Hankin, F. Nielson, H. R. Nielson, F. Yang, Advice for coordination, in: Proc. Intl. Conf. on Coordination Models and Languages, Vol. 5052 of LNCS, Springer-Verlag, 2008, pp. 153–168.

[19] R. de Nicola, G. L. Ferrari, R. Pugliese, Klaim: A kernel language for agents interaction and mobility, IEEE Trans. Softw. Eng. 24 (5) (1998) 315–330.

[20] D. Harel, A. Pnueli, On the development of reactive systems, in: K. R. Apt (Ed.), Logics and models of concurrent systems, Vol. 13 of NATO ASI Series F: Computer And Systems Sciences, Springer-Verlag New York, Inc., New York, NY, USA, 1989, pp. 477–498.

[21] F. Maraninchi, Y. Rémond, Argos: an automaton-based synchronous language, Computer Languages (27) (2001) 61–92.

[22] N. W. Paton, O. Díaz, Active database systems, ACM Comput. Surv. 31 (1) (1999) 63–103.

[23] A. J. Bonner, Workflow, transactions and datalog, in: PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, ACM Press, New York, NY, USA, 1999, pp. 294–305.

[24] M. Liu, Extending datalog with declarative updates, J. Intelligent Information Systems 19 (2002) 1–23.
URL `citeseer.ist.psu.edu/liu02extending.html`

[25] C. Zaniolo, On the unification of active databases and deductive databases, in: British National Conference on Databases, 1993.

[26] M. M. D. Montesi, E. Bertino, Transactions and updates in deductive databases, in: Knowledge and Data Engineering, 1995.

[27] L. Raschid, J. Lobo, Semantics for update rule programs and implementation in a relational database management system, in: ACM Transactions on Database Systems, Vol. 22, 1996, pp. 526–571.

[28] S. Abiteboul, V. Vianu, Procedural and declarative database update languages, in: PODS '88: Proceedings of the seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database system, 1988.

[29] S. Naqvi, R. Krishnamurthy, Database updates in logic programming, in: PODS '88: Proceedings of the seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, ACM Press, New York, NY, USA, 1988, pp. 251–262.

[30] S. Naqvi, S. Tsur, A Logical Language for Data and Knowledge Bases, Computer Science Press, 1989.

[31] R. D. Nicola, M. Loreti, A modal logic for klaim, in: Algebraic Methodology and Software Technology, 1998.

[32] J. Misra, W. Cook, Computation orchestration, in: Software and Systems Modeling, 2006.

[33] V. Saraswat, R. Jagadeesan, V. Gupta, Foundations of timed concurrent constraint programming, in: 9th Annual IEEE Symposium on Logic in Computer Science, 1994.

[34] S. Krishnamurthi, Flapjax (November 2006).
URL `http://www.flapjax-lang.org/`

[35] E. Cooper, S. Lindley, P. Wadler, J. Yallop, Links: Web programming without tiers, in: submitted to ESOP 2007, 2007.

[36] G. Dong, R. W. Topor, Incremental evaluation of datalog queries, in: Database Theory - ICDT'92, 4th International Conference, Germany, 1992, Proceedings, Vol. 646 of LNCS, Springer, 1992, pp. 282–296.

[37] T. Harris, S. Marlow, S. P. Jones, M. Herlihy, Composable memory transactions, in: ACM Conf. on Principles and Practice of Parallel Programming, Chicago, 2005, pp. 48–60.

Appendix A

List of Publications

(**J** = Journal article, **C** = Conference paper, **S** = Short conference paper, **W** = Workshop paper, **TR** = Technical report, **P** = Presentation (paper presentations not included), **POS** = Position paper, **INV** = Invited paper, **T** = Teaching, **PAT** = Patent)

In progress/processing

PAT1 John Field, Rafah A. Hosn, Bruce David Lucas, Maria-Cristina V. Marinescu, Christian Stefansen, Mark N. Wegman, and Charles Francis Wiecha. Data-Oriented Programming Model for Loosely-Coupled Applications. IBM Research. Patent pending.

J4 John Field, Maria-Cristina Marinescu, and Christian Stefansen. Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications. To appear in *Theoretical Computer Science*, Elsevier.

J3 Fritz Henglein, Ken Friis Larsen, Jakob Grue Simonsen, and Christian Stefansen. POETS: Process-Oriented Event-driven Transaction Systems. To appear in the *Journal of Logic and Algebraic Programming*, Elsevier.

C3 Christian Stefansen, Sriram Rajamani, and Parameswaran Seshan. A Work Allocation Language with Soft Constraints. To appear at the *International Conference on Web Services (ICWS),* 2008.

Peer-reviewed

S2 Christian Stefansen, Sriram Rajamani, and Parameswaran Seshan. A Work Allocation Language with Soft Constraints. Short paper to appear at *Conference on Advanced Information Systems Engineering Forum (CAiSE Forum),* 2008.

J2 Christian Stefansen and Signe Ellegård Borch. Using Soft Constraints to Guide Users in Flexible Business Process Management Systems. To appear in the *International Journal of Business Process Integration and Management*, 2008.

C2 John Field, Maria-Cristina Marinescu, and Christian Stefansen. Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications. *COORDINATION 2007, Coordination Models and Languages*, 76–95, Springer Verlag, 2007.

J1 Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional Specification of Commercial Contracts. *International Journal on Software Tools for Technology Transfer (STTT)*, Springer Verlag, 2006.

C1 Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional Specification of Commercial Contracts. *Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA 2004)*, University of Cyprus Report TR-2004-6, 103–110. University of Cyprus 2005.

S1 Christian Stefansen. SMAWL: A SMAll Workflow Language Based on CCS. *Proceedings of the 17th Conference on Advanced Information Systems Engineering Forum (CAiSE Forum).* University of Porto 2005.

W2 Signe Ellegård Borch and Christian Stefansen. On Controlled Flexibility. *Seventh Workshop on Business Process Modeling, Development, and Support, BPMDS*, June 2006

W1 Signe Ellegård Borch and Christian Stefansen. Evaluating the REA Enterprise Ontology from an Operational Perspective. *Enterprise Modelling and Ontologies for Interoperability, EMOI – INTEROP 2004, CAiSE Workshops (3),* 144–152, 2004.

Not peer-reviewed

INV1 Fritz Henglein, Jesper Andersen, Ebbe Elsborg, Jakob Grue Simonsen and Christian Stefansen. Compositional contract specification for REA. Invited paperat the *First Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS 2007),* Oslo, October 9–10, 2007. University of Oslo, 2007.

POS2 Ken Friis Larsen, Christian Stefansen, Fritz Henglein, Jakob Grue Simonsen. An Event-based Architecture for ERP Systems. *3gERP workshop,* 2007.

POS1 Christian Stefansen. Transforming the Resources/Events/Agents Model into a Formal Process-Oriented Enterprise Framework. Position paper for the *First International REA Technology Workshop*, Copenhagen, 2004.

TR4 Christian Stefansen, Sriram Rajamani, Parameswaran Seshan. *A Work Allocation Language with Soft and Hard Constraints.* Technical Report. SETLabs, Infosys Technologies Limited, India 2007.

TR3 Christian Stefansen. *A Declarative Framework for ERP Systems.* Master's thesis/qualification report. Department of Computer Science, University of Copenhagen 2005.

TR2 Christian Stefansen. *SMAWL: A SMAll Workflow Language Based on CCS.* Technical Report. Harvard University Computer Science Technical Report TR-06-05. Harvard University 2005.

TR1 J. Andersen, E. Elsborg, F. Henglein, J.G. Simonsen, C. Stefansen. *Compositional Specification of Commercial Contracts.* DIKU Technical Report 05-04. University of Copenhagen 2005.

Appendix B

Dansk sammenfatning

Denne afhandling er en samling af seks bearbejdede videnskabelige artikler relateret til to forskningsområder:
(I) En deklarativ programmeringsramme for virksomhedssystemer (ERP-systemer):

- *POETS: Process-Oriented Event-driven Transaction Systems*. Denne artikel beskriver en ontologisk analyse af et lille segment af virksomhedsdomænet, nemlig bogholderiet og debitorsystemet. Resultatet er en begivenhedsorienteret tilgang til at designe virksomhedssystemer (ERP-systemer) samt en skitse af arkitektur på abstrakt niveau.

- *Compositional Specification of Commercial Contracts.* Denne artikel beskriver designet, de forskellige semantikker og brugen af et domænespecifikt sprog (DSL) til at modellere kommercielle kontrakter.

- *SMAWL: A SMAll Workflow Language Based on CCS.* Denne artikel viser hvordan mønstre for arbejdsgange (*workflows)* kan kodes i CCS og designer herefter et makrosprog, SMAWL, til arbejdsgange baseret på disse mønstre. SMAWLs semantik defineres via oversættelse til CCS.

- *Using Soft Constraints to Guide Users in Flexible Business Process Management Systems.* Denne artikel viser hvordan et processprogs manglende mulighed for at udtrykke *bløde constraints*—constraints som kan overtrædes lejlighedsvist, men som nøje overvåges—medfører et tab at intensionel information i procesbeskrivelserne. Dette gør det videre vanskeligt for et procesudførelsessystem at hjælpe sine brugere med at overholde etableret praksis. Artiklen beskriver herefter hvordan bløde constraints kan anvendes til indfange foretrukken praksis eksplicit i procesbeskrivelserne.

- *A Work Allocation Language with Soft Constraints.* Baseret på ideen om bløde constraints forklarer denne artikel designet, semantikken og brugen af et sprog til at allokere arbejde i forretningsprocesser. Sproget lader procesdesignerne udtrykke både hårde og bløde constraints.

(II) *Reactors*-programmingsmodellen:

- *Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications.* Artiklen motiverer, forklarer og definerer en distribueret, datadreven programmeringsmodel. I modellen er en *reaktor* en distributionsenhed med tilstand. En reaktor specificerer konstruktive, deklarative constraints på sine data og andre reaktorers data i samme stil som i sproget *Datalog*. Et forsøg på at opdatere en reaktors data starter en *reaktion* i løbet af hvilken andre reaktorer muligvis inddrages. Reaktionen slutter når alle constraints i de inddragne reaktorer er opfyldt, eller når det har vist sig umuligt at opnå dette (konflikt)

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[2] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–69, January 1997.

[3] Anthony A. Atkinson, Rajiv D. Banker, Robert S. Kaplan, and S. Mark Young. *Management Accounting.* Prentice Hall, third internation edition edition, 2001.

[4] Signe Ellegård Borch and Christian Stefansen. Evaluating the REA enterprise ontology from an operational perspective. In *Proceedings from the Open INTEROP Workshop on "Enterprise Modelling and Ontologies for Interoperability", EMOI - INTEROP*, June 2004.

[5] Daniel Brixen. Incremental methods for REA-based reporting. M.S. thesis, Department of Computer Science, University of Copenhagen, January 2005. In Danish.

[6] Ken Friis Larsen and Michael Nissen. FunSETL—functional reporting for ERP systems. In *Proc. 19th International Symposium on Implementation and Application of Functional Languages (IFL)*, Freiburg, Germany, October 2007.

[7] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[8] Christian Stefansen. A declarative framework for enterprise information systems. Technical report, Department of Computer Science, University of Copenhagen, Sep. 2005. Qualification report.

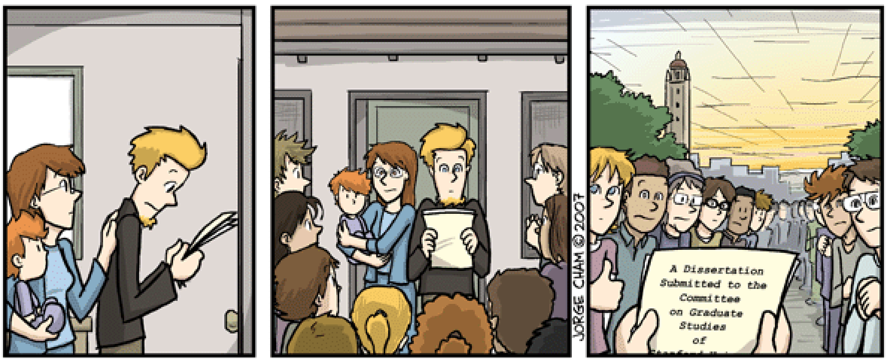
[9] Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, B. Kiespuszewski, and A.P. Barros. Workflow patterns. Technical report, Eindhoven University of Technology, GPO Box 513, NL-5600 MB Eindhoven, The Netherlands, 2002.

We die containing a richness of lovers and tribes, tastes we have swallowed, bodies we have plunged into and swum up as if rivers of wisdom, characters we have climbed into as trees, fears we have hidden in as if caves. I wish for all this to be marked on my body when I am dead. I believe in such cartography – to be marked by nature, not just label ourselves on a map like the names of rich men and women on buildings. We are communal histories, communal books. We are not owned or monogamous in our taste or experience. All I desired was to walk upon such an earth that had no maps.

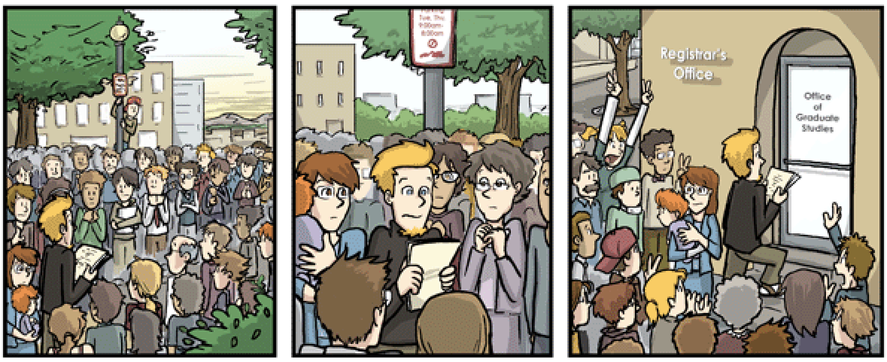
—Michael Ondaatje, *The English Patient*

In my end is my beginning.

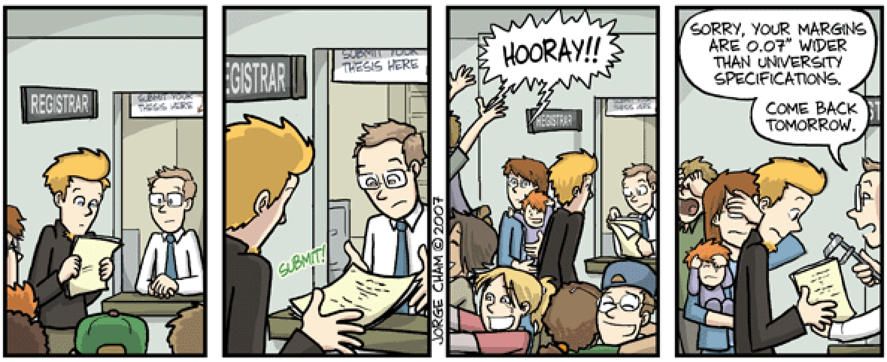
—T.S. Eliot, *East Coker*



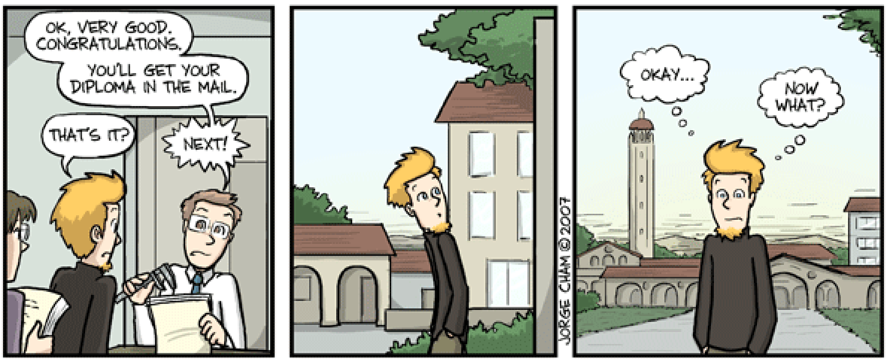
WWW.PHDCOMICS.COM



WWW.PHDCOMICS.COM



WWW.PHDCOMICS.COM


OK, VERY GOOD. CONGRATULATIONS.
YOU'LL GET YOUR DIPLOMA IN THE MAIL.
THAT'S IT?
NEXT!
JORGE CHAM © 2007
OKAY...
NOW WHAT?

THIS MARKS THE END OF THE THIRD CHAPTER IN THE PHD SAGA..!