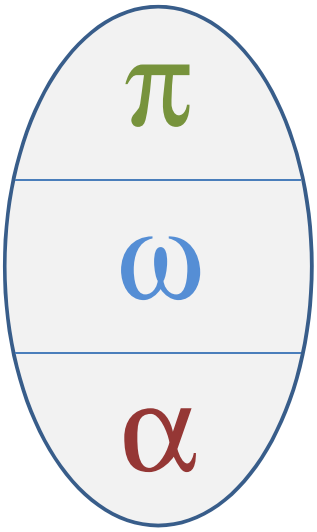
Protocol Modelling

Synchronous Composition of Data and Behaviour



Ashley McNeile

June 2016

A thesis submitted to the University of London
for the degree of Doctor of Philosophy

Birkbeck, University of London
Department of Computer Science and Information Systems

Declaration

This thesis is the result of my own work, except where explicitly acknowledged in the text.

Ashley McNeile. June 2016.

Abstract

This thesis develops and explores a technique called *Protocol Modelling*, a mathematics for the description of *orderings*. Protocol Modelling can be viewed as a hybrid of *object orientation*, as it supports ideas of data encapsulation and object instantiation; and *process algebra*, as it supports a formally defined idea of process and process composition.

The first half of the thesis focuses on describing and defining the Protocol Modelling technique. A formal denotational semantics for protocol machines is developed and used to establish various properties; in particular that composition is closed and preserves type safety. The formal semantics is extended to cover instantiation of objects. Comparison is made with other process algebras and an approach to unification of different formulations of the semantics of process composition is proposed.

The second half of the thesis explores three applications of Protocol Modelling:

Object Modelling. This explores the use of Protocol Modelling as a medium for object modelling, and the facility to execute protocol models is described. Protocol Modelling is compared with other object modelling techniques; in particular by contrasting its compositional style with traditional hierarchical inheritance.

Protocol Contracts. This proposes the use of protocol models as a medium for expressing formal behavioural contracts. This is compared with more traditional forms of software contract in the generalization of the notion of contractual obligation as a mechanism for software specification.

Choreographed Collaborations. In this application Protocol Modelling is used as a medium to describe choreographies for asynchronous multiparty collaborations. A

compositional approach to choreography engineering, enabled by the synchronous semantics of Protocol Modelling, is explored and results established concerning sufficient conditions for choreography realizability. The results are extended to address choreographies that employ behavioural rules based on data.

Acknowledgements

A number of people deserve my gratitude for their contribution to my work. Michael Jackson was brave enough to allow me to work in his company, Michael Jackson Systems Ltd., through the 1980s and it was during this time that I became fascinated by the challenge of modelling software behaviour. I would also like to thank the others at MJSL with whom I worked and engaged in enjoyable debates about all aspects of software engineering.

Nick Simons made a very special contribution by collaborating with me in the development of the ModelScope tool. He did magnificent work in developing a textual concrete syntax for Protocol Modelling and writing the software that compiles and executes the language. ModelScope has proved remarkably robust in use and this is a tribute both to Nick's programming skills and to his wisdom in bouncing my specifications back to me when he felt they were not of sufficient quality to implement.

The development of the formal side of Protocol Modelling owes much to the energy and enthusiasm of Ella Roubtsova. She has done a great deal to encourage me to develop and describe the ideas, as well as agreeing to co-author a number of papers about Protocol Modelling and its applications. The rigour required to meet the standards of peer-reviewed publication provided the motivation to develop the formal basis for Protocol Modelling that is the backbone of this thesis and without Ella this would not have happened.

I am grateful to my wife, Vicky, for throwing down the gauntlet represented by the completion of her own PhD in 2010. This challenge could not go unanswered. Her patience and support have been an enormous comfort when, unable to complete a proof, I have become frustrated and morose, sometimes for weeks on end. She has struggled magnificently to resist the temptation to have my computer crushed and/or me sectioned.

I would like to thank my supervisors at Birkbeck, Trevor Fenner and Keith Mannock, for their wisdom and guidance; and my examiners, Emilio Tuosto and Mark Josephs, for their work in conducting my examination and for the benefit of their expertise.

Contents

List of Figures

Part I

Protocol Modelling

Chapter 1

Introduction

This chapter introduces *Protocol Modelling*, hereafter abbreviated as **PM**, exploring its origins and positioning it with respect to prevalent paradigms of software modelling and description. The notations of PM are explained using a small example as illustration.

1.1 Origins

PM can be viewed as a merger of two pre-existing fields of work in software modelling: *object life-cycle modelling* and *process algebra*. PM borrows from process algebra in establishing a formal relationship between the states of a process and the events it can handle, and in supporting process composition. It departs from traditional process algebra in allowing processes to maintain data, allowing states to be computed from stored data, and supporting ideas of identity (objects) and instantiation.

1.1.1 Object Life-cycle Modelling

The 1970s and 1980s saw a revolution in ideas about software modelling and structure, as the procedural view of computing gave way to *object orientation*: the idea that software can be thought of as consisting of objects that own both data and behaviour. The lasting legacy of this revolution has been *OO programming languages* which now dominate the IT industry. But some of the early work on object orientation, with origins predating the OO programming revolution, focused on building behaviourally expressive

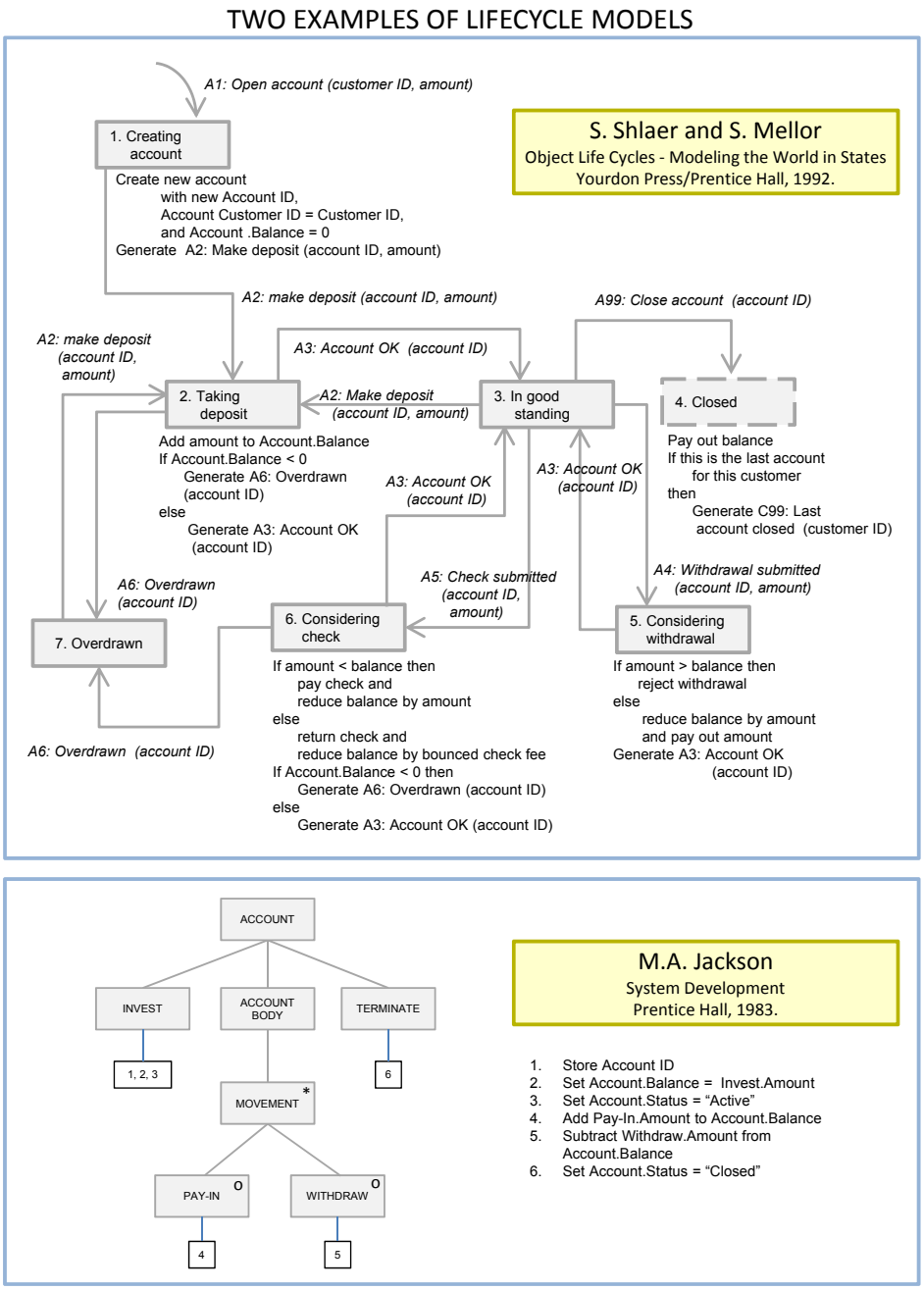


Figure 1.1: Two Examples of Object Life-Cycle Modelling

object based domain models, part of which involved pioneering modelling techniques for describing object lifecycles. Two such techniques, both of which emerged in the 1970s, were Jackson System Development (JSD) developed by Jackson and Cameron [41], and Recursive Design [74] developed by Shlaer and Mellor. Both espoused the

idea that software should be structured as objects that mirror objects of the domain and both, independently, developed and championed the idea of modelling object life-cycles. Life-cycles were defined in terms of the states that an object can adopt and how events in the system cause objects to progress through these states. In support of this, both techniques had graphical techniques for modelling event-based behaviour, as shown in Figure 1.1.

> *In 1983, also Jackson presented a new method, called Jackson System Development (JSD), which can be seen as the generalization of JSP (Jackson Structured Programming). One could rightfully say that JSD already contains important elements of object-oriented design, and therefore that it is a precursor of object-oriented design.*
> **– Maurice Verhelst [78]**

Both JSD and Recursive Design can be seen as foreshadowing object orientation, as recognized for instance in the quote above, and both won adopters and recognition completely independently of the appearance of OO programming. However, both were largely eclipsed when the OO programming revolution, beginning with Smalltalk, took off in the late 1980s. As OO programming ideas started to dominate thinking about software development, modelling paradigms quickly fell in behind, aligning themselves closely to the capabilities and features of the new OO programming languages: classes, inheritance, relationships, attributes and methods. Driven by the success of OO programming a raft of OO modelling techniques appeared during the 1980s and 1990s, to be cemented by the synthesis of *Unified Modeling Language* (UML) in the early 1990s. As none of the mainstream OO languages provided any direct support for implementing object life-cycles, object life-cycle modelling largely vanished from the mainstream.

The resultant orthodoxy that was established in UML, and which has persisted since, divides modelling into two domains that, to a large extent, are presented and treated as independent without significant crossover:

- **Object modelling**, as embodied in UML *Class Diagrams* and *Interaction Diagrams*. These models address object based software. Class Diagrams show classes, attributes, methods, relationships and inheritance. Interaction Diagrams (Sequence and Collaboration Diagrams) show how objects work together by message passing to implement functional behaviour.

- **Process modelling**, as embodied in UML *Activity Diagrams*. These models deal with process flows and embody an idea of event based movement through states. But the process flows of Activity Diagrams have no formal relationship to the objects of Class Diagrams, and do not support any significant object-based richness of structure such as relationships or inheritance.

The implementation realm mirrored this split, so that we have programming languages that support objects (C++, Java, C♯, etc.) and others that support process (BPEL and its derivatives).

However, this separation of modelling into two domains is an accident of history. It does not appear to be essential, as in PM it does not exist. And it is arguably damaging, as it creates barriers in the way we think about software engineering. The work described in this thesis shows that, if these barriers are removed, new ways of combining concepts are enabled leading to new ways of solving problems. However, this is only achieved by adopting notions that are rather different from those in UML.

1.1.2 Process Algebra

The second ingredient in the formation of PM is *process algebra*. Process algebras attempt to capture process behaviour in algebraic form, and provide formal semantics for these algebras to enable reasoning. Process algebra has its origins in the 1970s with early work on the denotational semantics of programming languages such as ALGOL and PL/I, in the context of giving a formal treatment of parallel composition constructs. The body of theory that has grown from this has proved to be a valuable tool for reasoning about concurrent behaviour, and has formed the basis for various *model checking* [20] techniques used to verify the correctness of concurrent software.

Most of the theoretical work on process algebra has been done in academia, through the work of Hoare [38], Milner [58], Baeten et al [4] and a number of others. PM draws directly on this work, but departs from the main traditions of the field in two significant ways. The first departure relates to the treatment of *data* and the second to the positioning and use of *parallel composition*. These differences are explored through the rest of the thesis. Before doing that, we introduce the basics of PM using a small illustration to explain, informally, the key ideas.

1.2 Illustration of a Protocol Model

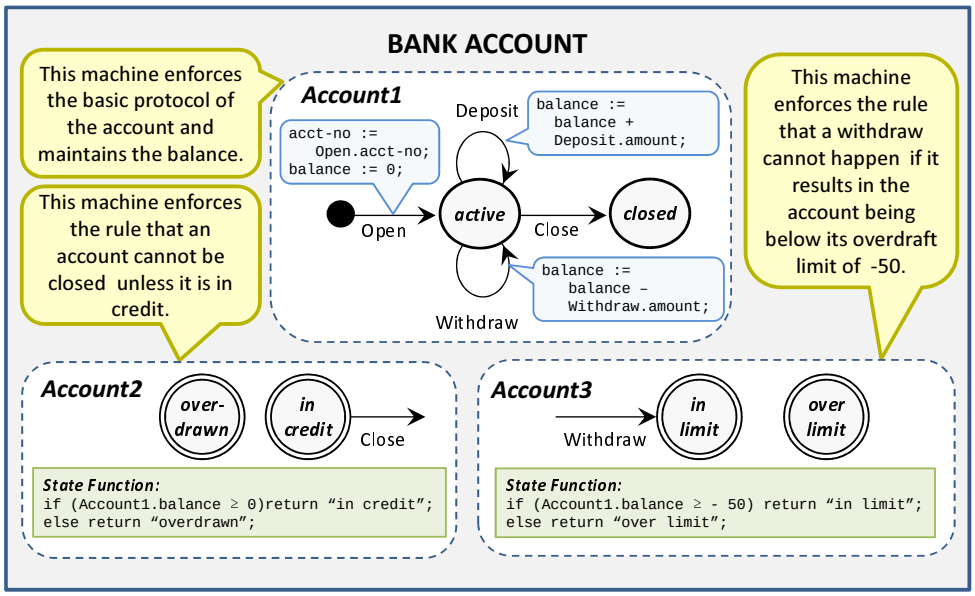


Figure 1.2: Protocol Model of a Bank Account

Figure 1.2 shows a protocol model of a simple bank account. The account is specified using three protocol machines: *Account1*, *Account2* and *Account3*. These three machines are in parallel composition using a generalized form of the parallel operator $\|$ of Hoare's CSP, so the full behaviour of the account is given by:

$$Account1 \parallel Account2 \parallel Account3$$

Throughout this thesis we use the graphical convention that cartouches with dotted outlines, such as those in Figure 1.2, work together to define the behaviour of a single object.

Informally, the semantics of the composition is that an action[1] is allowed by the model as a whole if and only if it is allowed in each machine where it appears. For instance:

- *Close* can only take place if *Account1* is in state *active* **and** *Account2* is in state *in credit*.

[1]There is a question about whether to use *event* or *action* to denote an atomic element of behaviour. We have chosen *action*.

- *Withdraw* can only take place if *Account1* is in state *active* **and** *Account3* ends up (after the withdrawal) in the state *in limit*.

For a *Deposit* to be allowed, however, only requires that *Account1* is in state *active*, as *Deposit* does not appear as an action on any transition in the other machines.

The upper machine, *Account1*, conforms to the familiar form of a conventional state transition diagram. *Account2* and *Account3*, however, depart from conventional state transition notation as their states are calculated by a function (the *State Function* shown in each of the two machines) rather than being driven by the transitions. The state icons of derived-state machines are shown with a double outline as a graphical signal of the fact that they are calculated. UML has a concept of *Derived Element*, see [64], but PM has extended this concept so that states of a state machine can also be derived.

The example shows the influence of *object orientation*, in that *Account1* encapsulates the *balance* of the account; and of *process algebra*, in the use of parallel composition of the three machines to yield the full behaviour of the account. This example also shows how protocols, the rules governing the allowed sequencing of actions in a model, are **explicit** in a PM model. In a conventional software development process, the protocols of a system are not normally explicitly modelled at all. This may be because the modelling formalisms that target this need, for instance the StateChart in UML, do not readily express the interaction between data and behaviour. Consequently protocol rules are often unspecified or poorly specified and appear as emergent properties of the code. If the protocols are not properly explicit in models or specifications, different programmers working on different parts of the system may make different assumptions about what is possible and thus create incoherent behaviour.

1.3 Introduction to the Basic Concepts

The basic structural elements of a Protocol Model are *actions*, which model atomic occurrences; and *protocol machines*, which are the basic unit of behavioural definition. This section describes these basic concepts. The discussion is informal, and is intended to provide an intuition of how PM models are defined and how they behave. Part II of this thesis will provide a formal treatment of the semantics.

1.3.1 Action

PM is used to model possible sequences of occurrences of some kind. There is no particular restriction on what can constitute an occurrence in the modelled domain; however for the purposes of modelling an occurrence must be:

- Treated as atomic (not sub-divisible) and instantaneous
- Finitely describable as data.

Any set of occurrences that conform to these can be chosen to constitute the set of *actions* of a model. For the purposes of this introduction, actions can be thought of as messages originating outside the protocol model and presented as input messages to it. This is actually a restrictive assumption that will be relaxed in Section 4.2.3, where the concept of action will be broadened to include outputs as well as inputs.

Each action presented to the model carries a set of data items called *fields*. For instance the *Deposit* and *Withdraw* actions in Figure 1.2 on page 20 would each carry three fields:

- The *type of the action*, either `Deposit` or `Withdraw`
- The *identifier of the account* to which the action applies (assuming the bank has more than one account)
- An *amount* by which the *balance* of the account is to be increased or reduced.

The reaction of a protocol model to an action is instantaneous. Taking actions to be instantaneous is not a restrictive assumption: if we want to model something that we do not regard as instantaneous but persistent over some period of time, we model it as having separate *start* and *end* actions.

1.3.2 Alphabet

The set of different types of action that a machine understands is called its *alphabet*. For this discussion we shall take the alphabet of a machine to be a set of action types that appear on any transition in the graphical depiction of the machine. Thus in Figure 1.2 the alphabet of *Account1* is $\{Open, Deposit, Withdraw, Close\}$; and the alphabet of *Account2* is $\{Close\}$.

As we shall see in Part II the concept of alphabet is important in developing a theory of how PM applies to object modelling.

1.3.3 Protocol Machine

A *protocol machine* is a machine whose behaviour is defined in terms of its ability to *allow* or *refuse* actions presented to it. If a machine allows an action A it will, *in general*, adopt a new state reflecting the occurrence of A. The words "in general" indicate that a change of state is not mandatory, and may only be reflected in the machine as a change of some data attribute (like *balance* in *Account1* in Figure 1.2). If it refuses A, the machine is unable to advance to a new state, as no possible such advance is defined for it, and so it **must** remain in the same state.

States. Every protocol machine owns a set of *states* that are used in the determination of its behaviour (whether it allows or refuses an action). A machine's state may be *stored*, as is the case with *Account1* in Figure 1.2; or it may be *derived*, as is the case with *Account2* and *Account3*. *Stored-state* machines have the familiar form of a state transition diagram, where the next state of the machine is driven by the topology of the machine. *Derived-state* machines employ a function (the *state function*) to calculate the state when needed, this function being part of the definition of the machine. Derived-state machines do not need to be topologically connected, as the transitions do not drive state. Thus the transitions in *Account2* end "in mid-air", and the transition in *Account3* begins "in mid-air".

Constraints. Whether a machine in a given state allows or refuses an action A is determined autonomously by the machine. To make its decision, the machine may use the state before the action (a rule specified this way is called a *pre-state constraint*) and the state that the machine would reach if it were to allow the action (a rule specified this way is called a *post-state constraint*). Examples of pre-state constraints in Figure 1.2 are given by the rule in *Account1* that *Deposit*, *Withdraw* and *Close* are only possible in the state *active*; and the rule in *Account2* that *Close* is only possible in the state *in credit*. The rule in *Account3* that *Withdraw* is only possible if it *results* in the state *in limit* is an example of a post-state constraint. Notice that a post-state constraint requires that the post-state is **derived**.

Quiescence. Between actions a machine always reaches a quiescent state, wherein it cannot undergo any further change of state unless and until presented with another action. This is axiomatic: a machine that **does not** reach quiescence **is not** a protocol machine. When a machine is quiescent, its state and data have well-defined values that are accessible to other, composed, machines. Access to a machine's state or data when it is not quiescent is meaningless.

Attributes. A protocol machine owns a set of *stored attributes*, which only it can alter and only when moving to a new state in response to an action. An example of a stored attribute is the *balance* maintained by *Account1* in Figure 1.2. This attribute is initialized to zero by the *Open* and incremented/decremented by *Deposit* and *Withdraw*. PM has the notion of data *encapsulation* familiar in object orientation, whereby the stored attributes owned by a protocol machine can only be altered by that machine, although it can be accessed by other, composed, machines. Note that a stored state, as exemplified by the state of *Account1*, is a stored attribute that has protocol significance because it determines protocol constraints.

In addition to its stored attributes, a protocol machine may own a set of *derived attributes*. The value of a derived attribute is defined by a *derivation function* which can be viewed as a property of the attribute. Derived attributes are determined by their derivation function and assumed to be computed instantaneously on demand. They only have defined values at quiescence. For instance, suppose that machine *Account1* has an attribute that expresses the balance in another currency, say USD($) instead of GBP(£). The value of this attribute would be derived rather than stored: *Account*1 would obtain the $\$/£$ exchange rate and use this along with the stored GBP balance to calculate the USD balance whenever it is required. Note that a derived state, as exemplified by the states of *Account2* and *Account3*, is a derived attribute that has protocol significance because it is used to determine protocol constraints.

When it consumes an action a protocol machine can alter the values of its stored attributes. In the graphical syntax, these updates are shown in bubbles attached to the transitions, as *Account1* has updates for *balance* on *Open*, *Deposit* and *Withdraw*. The update calculation may make use of attributes of the action being consumed and the values of attributes (stored and derived) of the machine itself and other machines with

which it is composed. As the values of machine attributes are only well-defined at quiescence, the values used in the context of an update are those that pertained at the last quiescent state of the model.

1.3.4 Observation Universe

We use the term *observation*[2] to define the observed association of a value to a symbolic name during execution of a machine. An example of an observation is $(x = 3)$ where the symbolic name x is observed to have the value 3.

Given a domain or area of interest that is to be modelled, an *observation universe* defines a universe of possible observations by giving an exhaustive list of:

- the symbolic names (data variable names) that can be used in action fields and stored attributes; and
- the complete set of values that each symbolic name can take.

Any machine used to model this domain will use some subset of the symbolic names of the universe to define its alphabet and attributes.

1.3.5 Protocol Model

A *protocol model* is a set of *composed* protocol machines that provides a complete definition of some domain or area of interest. A protocol model must give complete coverage of the data universe, so it defines behaviour for all possible actions (all combinations of values of the action fields defined in the universe) and gives a value to all the machine attributes defined in the universe under all behaviour scenarios.

A protocol model is also a protocol machine, so also reaches quiescence between successive actions. A model is quiescent when *all of its component machines are simultaneously quiescent*. When the model is quiescent, the *global state and data* of the model is well-defined. A consequence of this is that the presentation of actions to a model always has a well-defined ordering over time, and between any successive pair the model has well-defined global data.

[2]This terminology is borrowed from Hoare and He, [35].

Chapter 2

Behaviour Modelling Concepts

This chapter provides an informal introduction to some of the key concepts and terminology used in PM to model behaviour, and prepares for the denotational semantics in Part II.

2.1 Synchronous and Asynchronous Composition

Composition is central to PM, as it is the means by which models are built from constituent building blocks. It is also central to PM that inter-machine data access is possible between composed machines. However, in the presence of inter-machine data access, the behaviour of a composition is dependent on whether the composition has *synchronous semantics* or *asynchronous semantics*. In this section we illustrate this and motivate the choice of synchronous semantics in PM.

The distinction between synchronous and asynchronous composition is apparent in the example shown in Figure 2.1. This figure shows a model with a machine:

$$S = P \parallel Q \parallel R$$

specified as three composed machines. We suppose that P and Q start at their respective $\bullet$ (initial) states and therefore R starts in the state `not A and not B`. This means that P allows x, Q allows y, and R allows both x and y. Under the rules of $\parallel$ composition, S therefore allows x and y. Now consider how the execution of S advances

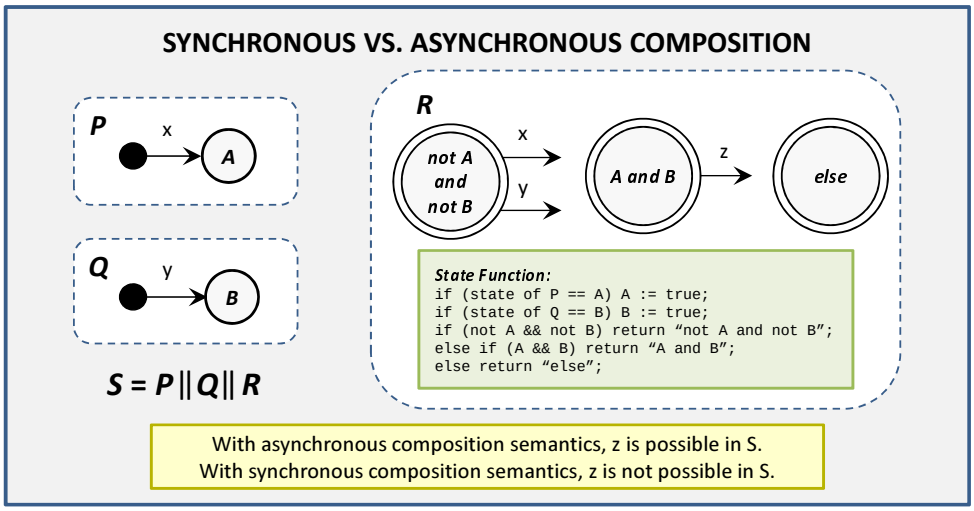


Figure 2.1: Synchronous and Asynchronous Composition

under two different assumptions about the behavioural semantics, *synchronous* and *asynchronous*, of the composition:

Synchronous Composition. In the synchronous case, S advances strictly one action at a time. Suppose x happens. P is then in state `A`, Q is still at its $\bullet$ state. R evaluates its state to `else` (as it is in neither of its other two states). As R refuses all actions of its alphabet $\{x,y,z\}$ when in state `else`, under the rules of $\|$ nothing further can happen in S. A similar argument applies if y happens first. These are the only two possibilities, and in neither is z possible in S.

Asynchronous Composition. In the asynchronous case, there is no discipline of advancing one action at a time. So x and y could take place simultaneously and the first coherent state that R obtains could be `A and B`.[1] R is now in a state where z is possible; and as z is not in the alphabets of P and Q, z is then possible in S. (Equally, though, x and y might not take place simultaneously and the system might not allow z.)

Consider the example in Figure 1.2 on page 20 and suppose that a *Withdraw* and a *Close* are presented together. With asynchronous semantics it would be possible for

[1] It could, perhaps, be argued that the presence of both x and y in R means that they cannot happen simultaneously. Suppose, though, that R is replaced by two machines, each essentially a clone of R with the same state function, where x has been removed from one and y from the other.

Account2 to determine that the balance is not overdrawn so that account closure is allowed at the same time that *Account1* allows the withdrawal, taking the account overdrawn, before closing the account. The failure to ensure that the account cannot end up closed and overdrawn mirrors the failure to prevent z from happening in S; and is generally known as a *race condition*. PM seeks to create models whose behaviour is not at the mercy of race conditions and so the asynchronous composition paradigm is not suitable.

As a note of caution, it is important to understand that the term *synchronous* can be used in two contexts:

- The semantics of *composition*, where it refers to whether or not composed processes can advance independently and concurrently.
- The semantics of *inter-process message passing*, where it refers to whether or not the send and receive events for a message sent from one process to another are simultaneous (or, equivalently, whether the sender is blocked until the receiver has received and processed the message).

This distinction is explored by Bergstra et al. [11] who use the terms synchronous/asynchronous *co-operation* for the former and synchronous/asynchronous *communication* for the latter; and classify well known process algebras and related formalisms along both dimensions.[2]

> - ***synchronous co-operation + synchronous communication**: SCCS, MEIJE, ASP, ASCCS.*
> - ***synchronous co-operation + asynchronous communication**: No example known to us.*
> - ***asynchronous co-operation + synchronous communication**: CCS, CSP, ACP, Ada, Petri Nets, uniform processes of de Bakker and Zucker.*
> - ***asynchronous co-operation + asynchronous communication**: CHILL, data flow networks, restoring circuit logic.*
>
> **– Jan Bergstra, Jan Klopp and John Tucker [11]**

[2]This paper was published in 1985 and some of the examples, such as CHILL, show its age. But the framework remains valid and useful.

Our use of the term synchronous in the context of Figure 2.1 corresponds to synchronous **co-operation**. As we shall see in Chapter 4, PM also uses synchronous communication, and so belongs to the first category in the taxonomy of Bergstra et al. cited above.

2.2 Independence and Dependence

We distinguish two types of machine:

- A *dependent* machine is one that accesses the values of attributes that it does not own (i.e., that belong to other machines composed with it). Both *Account2* and *Account3* in Figure 1.2 on page 20, considered as "stand-alone" machines, are dependent as they use *balance*, which they do not own, to compute their state.
- An *independent* machine is one that makes no such access. *Account1* is independent; as is the protocol machine formed as the composition $(Account1 \parallel Account2)$ as the reference to the *balance* attribute used by *Account2* to determine its state is resolved by *Account1*, which is within the composition.

While derived state calculations are the source of dependency in the machines *Account2* and *Account3*, the definition extends to **any** inter-machine reference whereby one machine uses the attribute of another in a computation that it undertakes. Note that two machines having elements of their alphabets in common **is not** a source of dependency between them. Thus *Account1* is deemed independent even though actions in its alphabet are also in the alphabets of *Account2* and *Account3*.

A *protocol model*, a composition of machines that gives a complete definition for some domain, **must** be independent. This means that the behaviour of a protocol model is completely driven by actions and it makes no access to any data that it does not itself own.

A formal definition of independence is given in Section 4.2.8.

2.3 Determinism

Protocol machines are deterministic, in the sense of supporting repeatability of behaviour. Informally, this means that if you execute a protocol machine twice, giving an identical data environment and presenting an identical sequence of actions in both cases, then its behaviour in terms of what actions it allows and refuses at each step will be identical.

Because of the interaction between data and behaviour, the concept of determinism has to be defined with care, and we give a formal definition in Section 4.2.2. Determinism is a somewhat slippery notion and there is a fuller discussion of it in Section 5.1.5.

2.4 Post State Constraints and Guards

A machine with a derived state can specify post-state constraints, as in *Account3*. Post-state constraints are unconventional, as violation of such a constraint is only apparent once consumption of the violating action has been attempted and the machine has thereby ended up in the "wrong" state.

Traditional state transition modelling would not use a separate machine for the constraint in *Account3*, but instead use a *guard* on the transition for *Withdraw* in *Account1*. A guard is a boolean condition attached to a transition which must evaluate to true if the action that fires the transition is to be allowed. In this case the guard would be:

$$(\texttt{balance} - \texttt{Withdraw.amount} \geqslant -50)$$

There is no difference of expressive power between guards and constraints expressed using derived states, and in theory PM could have been formulated using either. However the use of guards was not favoured for the following reasons:

- Formalization of PM without guards is simpler, as the semantic domain of "completions" described in Chapter 4 delivers a semantics for post-state constraints "for free". If guards were used, extra formal machinery would be needed to define their semantics.
- PM supports topology based reasoning techniques on derived-state machines, such as are used to specify post-state constraints and so allows topological reasoning to address the interaction between data and behaviour. The reasoning

techniques are described in Chapter 6 and is the basis for formal the analysis of collaborations using data-based rules to constrain behaviour discussed in Chapter 9.

- In the context of object modelling, using derived-state machines allows re-use in a way that guards do not. The discussion of object modelling and mixins in Chapter 7 describes this kind of re-use.

Chapter 3

Thesis Structure and Contribution

3.1 Structure

The remainder of this thesis is structured into two parts as follows.

Part II. The first chapter of this part of the thesis develops a denotational semantics for PM. The semantics is built up over four sections addressing:

Data. Defines a notation for describing data in PM and introduces the concept of an *observation universe*, being a structure that captures all possible associations of values to symbols that may be observed and specifies which are valid and which are not.

Machines. Defines the semantics of a single protocol machine using the notion of *completions*, which are similar to *traces* but include the evolution of data by giving the value to symbol observations that pertain at reach step in the trace.

Composition. Defines the semantics of machine composition in two forms: *homogeneous*, where the composed machines use the same observation universe; and *heterogeneous*, where the composed machines use different observation universes.

Models. Defines what is meant by a *model* in PM and the conditions under which a model is *well-behaved*, in the sense that it preserves data integrity rules under all execution scenarios.

This formal basis is used to describe and discuss:

- The definition of a machine's *alphabet*, being the set of actions that the machine understands.
- How PM can be used to model *objects* and object instantiation.

The second chapter of this part positions PM and its formal semantics against other related approaches to behaviour modelling, and discusses some of the questions it raises about the nature of such notions as *determinism* and *concurrency*.

The final chapter of this part gives a catalogue of transformations and reasoning techniques that can be used to work with protocol machines and models expressed as labelled transition systems. These techniques are used in the last part of the thesis, in particular in the application of PM to choreography and multiparty collaborations.

Part III. Three applications of PM are described:

Object Modelling. Expands on the ideas in Part II on the use of PM to represent objects and object models. The compositional style of PM, which lends itself to mixin based modeling of behavioural variation, is contrasted with the more mainstream inheritance based approach of mainstream OO languages.

Protocol Contracts. Proposes a general framework for thinking about contractual software specifications (specifications that have a formal notion of compliance), and shows how PM provides a suitable medium for a particular kind of contractual specification, concerned with its protocol for interaction.

Choreography. Addresses the application of PM to the design of stateful multiparty collaborations whose participants interact using asynchronous message exchange. This includes a definition of choreography realizability, and gives sufficient conditions for a choreography expressed in PM to be realizable.

3.2 Contribution

The mission behind the construction of this thesis was to formulate a better way of describing, in a formal way, the interaction of data and behaviour in software. This vision was fuelled by the author's conviction that:

- The current modelling mainstream, for reasons outlined in the opening chapters of this thesis, has become overly skewed to the concepts and conventions of object oriented programming. In particular, the split that has opened up between object and process modelling is an product of this skew and is artificial.

- State and data should be treated as dual, so that data can be refactored as state and vice-versa, with data providing a means of abstracting over state spaces. The machinery that traditional process algebras employ to model the behavioural effect of inter-process data access, such as guards or constraints, do not allow data and state to be viewed as interchangeable and have no direct way of seeing data as abstracting state spaces.

At a high level, the main contributions of the work in this thesis are:

- The creation of a formal synchronous compositional algebra for interactive computation that provides a theoretical treatment of data integrity under composition.

- Some new insights into the semantics of process algebraic composition, particularly with respect to the duality of data and state and the representation of inputs and outputs.

- Formal demonstration that, with the synchronous composition of Protocol Modelling, *composition preserves choreography realizability*. So if a choreography is defined as a composition and it is known that the components are individually realizable, then the whole choreography is realizable. This has the potential to make choreography engineering more scalable to complex problems.

- A formalism for choreography description with generally greater expressive power than those given in the literature, particularly in the context of:

  - choreographies that cannot be described as a single finite state machine;
  - choreographies involving race conditions, where the order of receipt of messages is not pre-defined; and/or use "fork and join" flow topologies;
  - choreographies defined using rules based on data in addition to flow topology.

3.3 Reading Guidance

This thesis contains a large number of definitions, some of which are crucial to understanding PM and the main results that derive from it. In order to help the reader focus on those definitions that are crucial, and to skip lightly over those that are not, the following colour coding has been used:

A definition that is central to PM. (3.1)

A definition that, although not central to PM, is used extensively through the thesis. (3.2)

The thesis contains many formal definitions. The notations and standard functions used in the these definitions are catalogued in Appendix . Each formal definition in the main body of the text is followed by a natural language rendition.

Part II

Semantics

Chapter 4

Denotational Semantics

This part of the thesis develops a denotational semantics for PM. The aim is to provide a medium in which it is possible to conduct formal reasoning about protocol models and their behaviour. We start with describing the semantic domain. A semantic function that maps an executable implementation of PM to the domain described in this chapter is given in Appendix C.

> *There is a race between the increasing complexity of the systems we build and our ability to develop intellectual tools for understanding that complexity. If the race is won by our tools, then systems will eventually become easier to use and more reliable. If not, they will continue to become harder to use and less reliable for all but a relatively small set of common tasks. Given how hard thinking is, if those intellectual tools are to succeed, they will have to substitute calculation for thought.*
> **– Leslie Lamport [62]**

4.1 Data

The basic notion we use for formalizing data is that of an *observation*. An observation associates a value with a symbol and will be written `(symbol=value)`, for example `(s1=6)` and `(s2="frog")` and the intuition is that this an observation made at a point in time that sees a particular value assigned to a symbol.

If w is an observation w^{symb} gives the symbol part and w^{val} gives the value part. So if w is the observation `(s1=6)` then $w^{symb} =$ `s1` and $w^{val} =$ `6`. The domains of *symbols* (for the left hand side of observations) and *values* (for the left hand side of observations) are disjoint and this means that a value cannot reference a symbol or vice versa. So `(s1=s2)` is not legal. Two observations are equal iff[1] both the symbol and value parts of the two are equal.

We will be using *sets of observations* to model the fields owned by an action and the attributes owned by a machine. In both cases they are used to capture both stored and derived values.

4.1.1 Finiteness

The following are assumed to be finite:

- The set of symbols available for use in observations,
- The set of values that any symbol may take.

This means that the number of possible sets of observations is also finite. We denote:

- The set of all possible symbols by $\mathcal{Y}$.
- The set of all possible values by $\mathcal{V}$.
- The set of all possible observations by $\mathcal{E}$.

Note that $\mathcal{E}$ represents every expressible observation, so $\mathcal{E} = \mathcal{Y} \times \mathcal{V}$, whether it has defined meaning or interpretation or not.

The reasons for this assumption of finiteness is that:

- We are only concerned with modelling systems that use finite storage.
- Parts of the semantics defined in this thesis would be very much harder (perhaps impossible) to formulate if the possibility of infinite sets of symbols or values is allowed. This applies particularly to the definition of behavioral equivalence set out shortly in Section 4.2.6 and the semantic function developed in Appendix C.

[1] iff = if and only if.

Although we assume it to be finite, we take the number of symbols and the number of values to be very large, so large that there is no use or application of the theory that can come anywhere near exhausting the supply.

4.1.2 Consistency

A set of observations is *consistent* if it contains at most one entry for a given symbol. Thus the set {`(s1=5)`, `(s2=3)`, `(s1=4)`} is **not** consistent because it contains two observations for the symbol `s1`.

Fn: *con*. We use the function *con* with signature:

$$con :: power(\mathcal{E}) \rightarrow boolean$$

to indicate that a set W of observations is consistent, as follows:

$$\begin{array}{l} con(W) \Leftrightarrow \\ \qquad \forall\, w1, w2 \in W : \quad {w1}^{symb} = {w2}^{symb} \;\Rightarrow\; w1 = w2 \end{array} \tag{4.1}$$

which requires that a consistent set of observations ascribes at most value to a given symbol.

4.1.3 Actions and Action Fields

An action is specified as a set of observations, where the symbols represent the *fields* of the action. So a deposit action for a bank account might be specified as:

{(`actionType="Deposit"`), (`acct-id=012345`), (`amount=£100`)}

The set of observations specifying an action must be consistent, so for any action A we have $con(A)$.

Note that an action is **immutable**, once formed it is fixed and cannot be changed. As described later in Section 4.2.3, actions can be input or output from a machine, or a combination of the two.

4.1.4 Machine Attributes

As outlined earlier, a protocol machine owns a *state* and a set of *attributes*, and both the state and the attributes may be either *stored* or *derived*. The attributes and values they take at a given time are represented as a set of observations. It is only necessary to represent the data that the machine makes public to other machines with which it is composed, and not what is private and not visible to other machines.

The set of observations specifying the state and attributes of a machine must be consistent at quiescence. Between a machine's quiescent states this set is undefined.

Unlike action fields, the values of the expressions defining a machine's state and attributes are **not immutable** and evolve from one quiescent state to the next as the result of the model's response to actions. Thus a given symbol can, in general, receive a new value in each new quiescent state of the machine.

4.1.5 Observation Universe

An *observation universe*, hereafter simply called a *universe*, is used to define a structure on symbols and values over which machines are constructed and captures:

- All possible observations that may be made, and
- The subset of observations that are considered "valid".

> *The objective of science is not to construct a list of actual observations of a particular system, but rather to describe all possible observations of all possible systems of a certain class.*
> ...
> *In applying this insight to computer programming, we shall confine attention to programs in a high level language, which operate on a fixed collection of distinct global variables.*
> **– C.A.R. Hoare [35]**

An observation universe, $\mathfrak{U}$, is defined as a tuple:

$$\mathfrak{U} = \langle \mathcal{U},\ \mathcal{A},\ \mathcal{D},\ \mathbb{V},\ \mathit{fixes} \rangle \tag{4.2}$$

where:

- $\mathcal{U} \subseteq \mathcal{Y}$ is a set of symbols used in the universe.
- $\mathcal{A} \subseteq \mathcal{U}$ is the subset of the symbols of the universe that are used to define *actions*. The motivation for distinguishing a reserved set of symbols for defining actions is to retain separation between the definition of *actions*, which correspond to the labels on the transitions of a labelled state transition system, and the definition of *machine storage*, which correspond to state in a labelled state transition system.
- $\mathcal{D} \subseteq \mathcal{U}$ is the subset of the symbols of the universe that are derived. The idea here is that some symbols, by virtue of their semantics, have values that are derived from others. For instance a boolean-valued symbol that specifies whether or not the current year is a leap-year can be derived from a symbol that specifies the current year.
- $\mathbb{V} \in \mathit{power}(\mathit{power}(\mathit{restr}(\mathcal{E},\mathcal{U})))$ is a set of sets of observations. The elements of $\mathbb{V}$ are sets of observations that are restricted to the symbols in $\mathcal{U}$ and which are regarded as **valid**, particularly in the sense that derived values are correctly calculated.[2] A set of observations W is *valid according to* $\mathbb{V}$ iff there is an element $V \in \mathbb{V}$ such that $W \subseteq V$ and we denote this by $W \Subset \mathbb{V}$.
- *fixes* is a mapping that defines the sets of observations that fix, or determine, the value of a derived symbol in $\mathcal{D}$. The *fixes* function has signature:

  $$\mathit{fixes} :: \mathit{restr}(\mathcal{E},\mathcal{D}) \rightarrow \mathit{power}(\mathit{power}(\mathit{restr}(\mathcal{E},\mathcal{U}\backslash\mathcal{D})))$$

  which defines the fixes for a derived observation (one that uses a symbol in $\mathcal{D}$) to be a set of sets of non-derived observations (that use symbols in $\mathcal{U}$ but not in $\mathcal{D}$).

An analogy for *fixes* is a *spreadsheet*, where some cells contain formulas and some contain values. The former are the derived cells. Consider a particular derived cell $A1$ with the formula $A1 = B1 * C1$, where $B1$ and $C1$ are termed *basis symbols* for $A1$.[3] Now consider a particular value for the derived cell $A1$, say 0. $A1$ will have this value if either $B1 = 0$ or $C1 = 0$, and we can therefore say that these are both "fixes" for the observation $(A1 = 0)$. We express these fixes as a set of sets of observations:

$$\mathit{fixes}((A1 = 0)) = \{\{(B1 = 0)\}, \{(C1 = 0)\}\}$$

[2] Formal definition of functions such as *restr*() can be found in Appendix A.

[3] This uses Excel style notation whereby $A1$ is the cell in column A and row 1. $A1$ can be viewed as its symbol.

Assuming that $B1$ and $C1$ must be positive integers, the set of fixes for the observation $(A1 = 1)$ has just a single member:

$$\mathit{fixes}((A1 = 1)) = \{\{(B1 = 1), (C1 = 1)\}\}$$

The full set of fixes for all possible observations of $A1$ captures the formula $A1 = B1 * C1$ by enumerating the results for all possible values of $B1$ and $C1$.

Having derived symbols in a universe means that we can define a notion of *computational closure*. A set of observations W is computationally closed, denoted $\mathit{closed}\,(W)$, as follows:

$$\mathit{closed}(W) \quad \Leftrightarrow \quad \forall\, v \in \mathit{restr}(W, \mathcal{D}) : \exists\, V \in \mathit{fixes}(v) \text{ with } V \subset W \tag{4.3a}$$

which requires that where W contains a derived value, it also contains a fix for that value. For convenience we also define *closed* on a set of sets of observations:

$$\mathit{closed}(\mathbb{W}) \quad \Leftrightarrow \quad \forall\, W \in \mathbb{W} : \quad \mathit{closed}(W) \tag{4.3b}$$

4.1.6 Universe Normal Form

The way fixes are expressed is not unique. For instance suppose that `(gender="male")` and `(gender="female")` are the only possibilities for `gender` present in the universe. Then the two sets of observations:

- {`(breed="dingo"),(gender="male")`}
- {`(breed="dingo"),(gender="female")`}

represent the same set of possibilities as just:

- {`(breed="dingo")`}

Suppose that a derived observation {`(dog="true")`} pertains iff {`(breed="dingo")`}; and suppose that we have two representations of the fixes for {`(dog="true")`} one including the gender and using two entries, and the other omitting the gender and using a single entry. Clearly these representations are semantically equivalent. To eliminate such differences in expression we will use a *normal form* for expressing fixes, as follows.

Consider an element $V \in \mathit{fixes}(v)$ where $v \in \mathit{restr}(\mathcal{E}, \mathcal{D})$. Given any symbol $y \notin \mathit{symb}(V)$ we can replace the subset $\{V\}$ of $\mathit{fixes}(v)$ by the expanded set:

$$\{V \cup \{w\} \mid w \in \mathit{restr}(\mathcal{E}, \{y\})\} \tag{4.4}$$

using the function *restr* to select from $\mathcal{E}$ all the observations possible for symbol y. This replacement does not change the meaning of $\mathit{fixes}(v)$, as all possible values of y are covered giving the same fix for v. This replacement is termed *fix expansion*. The opposite procedure to fix expansion is termed *fix reduction*. This entails, in the example above, replacing the expanded set defined by (4.4) with the original set $\{V\}$.

We say that a fix $\mathit{fixes}(v)$ where $v \in \mathit{restr}(\mathcal{E}, \mathcal{D})$ is in *normal form* iff:

$$\text{No reduction is possible in } \mathit{fixes}(v) \tag{4.5}$$

and normal form is unique by Theorem B.1.

The notation $\mathit{fixes}(v)^{norm}$ is used to signify conformance to normal form; so if $\mathit{fixes}(v) = \mathit{fixes}(v)^{norm}$ then (4.5) is true in $\mathit{fixes}(v)$. Normal form always exists, as if a fix is represented using the full set of symbols in $\mathcal{U}$ it is either in normal form or can be reduced; and if it can be reduced, the same argument applies to the result.

If all fixes in a universe are in normal form then the universe said to be in normal form. From now on, unless explicitly stated, we will assume that this is the case.

4.1.7 Definition of Total Universe

We can also construct the *total universe*, $\mathbb{U}$, for the set of symbols $\mathcal{U}$ as:

$$\mathbb{U} = \{W \mid W \in \mathit{power}(\mathcal{E}) \quad \wedge \quad \mathit{symb}(W) = \mathcal{U} \quad \wedge \quad \mathit{con}(W)\} \tag{4.6}$$

so that $\mathbb{U}$ contains every possible set of observations that gives a unique value to every symbol in $\mathcal{U}$.

The valid universe $\mathbb{V}$ and the total universe $\mathbb{U}$ may be contrasted as follows:

- Given a symbol in $\mathcal{U}$, $\mathbb{U}$ must represent every value that the symbol can take in $\mathcal{E}$. There is no requirement that $\mathbb{V}$ does so. Thus $\mathcal{E}$ and $\mathbb{U}$ may allow both positive

and negative values of *balance* of a bank account, but it is possible that $\mathbb{V}$ only represents non-negative values. More generally, if all symbols in $\mathcal{U}$ have a *type* (such as *boolean*, *integer*, *string*, etc.), then in $\mathbb{V}$ all observations conform to type.

- The values of derived symbols in $\mathbb{V}$ must obey well-formedness conditions described in the next section, which is not required in $\mathbb{U}$. In particular, in $\mathbb{U}$ there is not discipline that derived values are determined by their fixes as is required in $\mathbb{V}$ by (4.7d).

In Section 4.4, with the concept of *well-behaved machines*, we will see how a protocol model acts as the guarantor of the validity, as represented by $\mathbb{V}$, for the data that the model owns.

4.1.8 Universe Well-Formedness

An observation universe $\mathfrak{U} = \langle \mathcal{U},\ \mathcal{A},\ \mathcal{D},\ \mathbb{V},\ \mathit{fixes} \rangle$ must obey certain well-formedness conditions that are set out below.

1. The valid universe is contained by the total universe.

$$\mathbb{V} \subseteq \mathbb{U} \tag{4.7a}$$

This means that the every element of $\mathbb{V}$ gives a unique value to every symbol in $\mathcal{U}$.

2. Fixes are exhaustive.

$$\begin{aligned} &\forall\, U \in \mathbb{U} \text{ and } d \in \mathcal{D}\ : \\ &\quad \exists\, v \in \mathit{restr}(\mathcal{E}, \{d\}) \text{ and } V \in \mathit{fixes}(v) \text{ with } V \subseteq U \end{aligned} \tag{4.7b}$$

For a given derived symbol, every element of the total universe fixes a value for that symbol. The fix does not necessarily give the value that is present, so it could be that $v \notin U$.

3. Fixes are Unique.

$$\begin{aligned} &\forall\, v1, v2 \in \mathit{restr}(\mathcal{E}, \{d\}) \text{ and } V1, V2 \in \mathit{power}(\mathcal{E}) \text{ with} \\ &\quad d \in \mathcal{D} \text{ and } V1 \in \mathit{fixes}(v1) \text{ and } V2 \in \mathit{fixes}(v2)\ :\ \mathit{con}(V1 \cup V2) \Rightarrow v1 = v2 \end{aligned} \tag{4.7c}$$

This says that no $U \in \mathbb{U}$ contains conflicting fixes, being two fixes that give different values to the same derived symbol.

4. The valid universe derives correctly.

$$closed(\mathbb{V}) \tag{4.7d}$$

Every element of the valid universe correctly calculates (contains a fix for) every derived symbol.

5. Derivation is based on persistent stored data.

$$\forall\, v \in restr(\mathcal{E}, \mathcal{D}) : \quad symb(fixes(v)) \cap (\mathcal{A} \cup \mathcal{D}) = \emptyset \tag{4.7e}$$

This requires that:

- Action symbols $\mathcal{A}$ are not used as basis symbols for derivation. This is because it is assumed that actions are "ephemeral" and so not available for the computation of persistent attributes.
- Derived symbols $\mathcal{D}$ are not used as basis symbols for derivation, as fixes are expressed in terms of the recursive closure of the basis symbol relationship.[4]

6. The valid universe is complete.

$$\forall\, W1, W2 \Subset \mathbb{V} \text{ with } closed\,(W1) \text{ and } closed\,(W2) \text{ and } con(W1 \cup W2) : \\ (W1 \cup W2) \Subset \mathbb{V} \tag{4.7f}$$

Says that the universe is *complete*, in the sense that if two valid and closed sets of observations are consistent, then their union is also valid.

4.2 Protocol Machines

This chapter provides a formal denotational semantics for the behaviour of a protocol machine, and defines behavioural equivalence.

[4] In the same way as it is possible to trace from a derived cell in a spreadsheet via the derived cells among its basis cells, recursively, to reach a set of value cells that ultimately determine the derivation.

4.2.1 Formalization of Behaviour

The formalization of behaviour uses a concept of *completions*. A completion is similar to a *trace*: a sequence of actions corresponding to a possible execution instance of a machine. But, unlike a trace, a completion describes an instance of the execution of a machine **coupled with the evolution of its environment**. The need to consider the evolution of the machine's environment as well as the machine itself is a consequence of the fact that, in general, a protocol machine is not "closed". A machine can access the data of other, composed machines, and use the data so accessed to decide how to behave. Only by describing the evolution of the machine and its environment together can behaviour be fully captured.

4.2.1.1 Steps

A *completion* is described as a sequence of *steps*. A step describes the reaction of a machine to a single action presented to it. A step for a machine P is specified in terms of four parts:

- A set of observations that is the *action* presented to P in the step.
- A set of observations called *perceives*, which gives the values of the publicly available attributes of all the machines composed with P and used by P to determine its behaviour. The values in the *perceives* set of a step are as perceived by P in the quiescent state of the model at the **end** of the step. This set only includes observations that P needs in order to determine and define its state and behaviour.
- The *decision* that P takes on whether to *allow* or *refuse* the *action* of the step. The decision of a step can also be *crash* indicating that the machine has undergone an irrecoverable failure.
- A set of observations called *offers* giving the values of the public attributes of P in the quiescent state of the machine at the **end** of the step. These are values that other machines, composed with P, may access and use to determine their own behaviour.

As mentioned above, a completion describes not just the evolution of a machine, but the evolution of the machine's environment. This is the role of the *perceives* part of the step, as this describes the data that the machine sees in its data environment. The

evolution of this data is conceptually independent of the evolution of the machine itself.

In the formalization we need to be able to address the parts of a step, and we do this as follows. Suppose that s is a step, then we define:

s^{α}, a set observations defining the *action* of s. (4.8a)

s^{π}, a set of observations defining the *perceives* of s, being values perceived by P in its environment at the end of s. (4.8b)

s^{δ}, one of *allow*, *refuse* or *crash*, being the *decision* of s. (4.8c)

s^{ω}, a set of observations defining the *offers* of s, being values of public attributes of P made available to other machines. (4.8d)

As a mnemonic, the Greek letter used for each part of a step corresponds to its English name: α for *action* and so on.

Two steps, $s1$ and $s2$, are considered equal iff all their parts are equal: ${s1}^{\alpha} = {s2}^{\alpha}$ and ${s1}^{\pi} = {s2}^{\pi}$ and ${s1}^{\delta} = {s2}^{\delta}$ and ${s1}^{\omega} = {s2}^{\omega}$.

We also define the function *step* that constructs a step from parts:

$$s = step(s^{\alpha}, s^{\pi}, s^{\delta}, s^{\omega})$$

As notational conveniences, we also use:

- s^{τ} to denote $s^{\alpha} \cup s^{\pi} \cup s^{\omega}$, this being the *total data image* of a step.
- s^{ε} to denote $s^{\alpha} \cup s^{\pi}$, this being the *external data image* of the step, that part of the data image that is external to (not owned by) the machine.

In addition, we use the step part addressing mechanism to extract the parts from the **last step** of a sequence. Thus, if t is a finite sequence of one or more steps, $t^{\sharp}$ denotes $last(t)^{\sharp}$ where $\sharp \in \{\alpha, \pi, \delta, \omega, \tau, \varepsilon\}$.

Because the set of all possible observations is finite, so is the set of all possible steps. This set is denoted by $\mathcal{S}$.

4.2.1.2 Completions, Prefixes and Traces

A *completion* of P is a sequence of steps that provides a partial description of P's behaviour by describing a possible execution scenario. A completion has either:

- An infinite number of steps, all with a decision of *allow*; or
- A finite number, one or more, of steps each with a decision of *allow* followed by a single step with a decision of *refuse* or *crash*.

Note that a *stem* of a completion will never contain a refusal or crash step, as refusal and crash can only occur as the **last** step of a completion.

There is an important conceptual distinction between *refuse* and *crash*:

- Reaching a *refuse* does not mean that execution of a machine is finished.[5] If a machine refuses an action A it remains in the state that pertained prior to presentation of A and is free to receive **any action**, reacting exactly as though the A had never happened. Thus, after the bank account in Figure 1.2 on page 20 has refused a *Close* because the account is overdrawn (and so *Account2* refuses the action) it would be possible to make a *Deposit* to restore the balance to a credit value and then try the *Close* again, this time successfully.
- Reaching a *crash* marks the end of execution of the machine. Trying again after a *crash* is meaningless as the machine is in an incoherent state and has no specified behaviour.

As we shall see shortly, the behaviour of a protocol machine is defined as a *set of completions*. If $\mathcal{B} \subseteq S^{\infty}$ is a set of completions then we will use the following terminology:

- *prefix* for an element of $\mathit{prefixes}(\mathcal{B})$ (the prefixes of completions in $\mathcal{B}$).
- *stem* for an element of $\mathit{stems}(\mathcal{B})$ (the proper prefixes of completions in $\mathcal{B}$).
- *trace* for an element of $\mathit{actions}(\mathit{stems}(\mathcal{B}))$ (the set of sequences formed from the action parts of proper prefixes of the completions in $\mathcal{B}$).
- $\mathit{traces}(\mathcal{B})$ as a synonym for $\mathit{actions}(\mathit{stems}(\mathcal{B}))$.

The reason for defining a trace in terms of **stems** is so that a trace is a sequence of *allowed* actions, giving correspondence to the traditional definition of this term.

[5] In this respect, the term "completion" is perhaps misleading.

4.2.2 Formalization of Protocol Machine

This section formalizes the definition of a protocol machine. This is done by constructing a formal definition of protocol machine behaviour in terms of its completions.

4.2.2.1 Definition of Protocol Machine

A machine P is defined as a tuple:

$$P = \langle\, \mathfrak{U}, \Omega_P, \mathcal{B}_P \,\rangle \tag{4.9}$$

where:

- $\mathfrak{U}$ is a universe, as defined in Section 4.1.5.
- $\Omega_P \subseteq \mathcal{U} \backslash \mathcal{A}$ is a set of symbols used to represent the data offered by P.
- $\mathcal{B}_P \subseteq S^\infty$ is a set of completions that defines the behaviour of P.

We need to include a universe $\mathfrak{U}$ in the definition because P can access (read) the local storage of other composed machines, as modelled by the *perceives* part of each step in a completion. To create a complete definition of machine behaviour we have to ensure that we cover all possible sets of values that the machine might perceive and use the universe to do this.

We allow completions in $\mathcal{B}_P$ to be potentially infinite. This is to allow PM to model machines with "loops" giving no inherent limit to the length of their traces. This is true, for instance, of the machine *Account1* in Figure 1.2 on page 20 as there is no finite limit to the number of deposits and withdraws that could happen.

4.2.2.2 Protocol Machine Well-Formedness

For $P = \langle\, \mathfrak{U}, \Omega_P, \mathcal{B}_P \,\rangle$ where $\mathfrak{U} = \langle\, \mathcal{U},\ \mathcal{A},\ \mathcal{D},\ \mathbb{V},\ \mathit{fixes} \,\rangle$ to qualify as a *protocol machine* the following five properties must hold.

1. Steps are consistent with the universe.

$$\begin{array}{l} \forall\, s \in \mathit{asSet}(\mathcal{B}_P) : \\ \quad s^\tau \Subset \mathbb{U} \end{array} \tag{4.10a}$$

The data image of every step must be consistent with the total universe $\mathbb{U}$, as defined by (4.6). This means that $con(s^\tau)$, so the step gives at most one value to a given symbol.

2. Steps use symbol sets correctly.

$$\begin{array}{l} \forall\, s \in asSet(\mathcal{B}_P) : \\ \quad symb(s^\alpha) \subseteq \mathcal{A} \quad \wedge \quad symb(s^\omega) = \Omega_P \quad \wedge \quad symb(s^\pi) \subseteq \mathcal{U} \setminus (\mathcal{A} \cup \Omega_P) \end{array} \tag{4.10b}$$

This requires that the *action*, *perceive* and *offers* parts each use the appropriate set of symbols; and that the offered symbol set Ω_P is completely populated by s^ω. The *action* part is disjoint from the *offers* and *perceives* parts as it uses action symbols, $\mathcal{A}$. The *offers* and *perceives* parts are required to be disjoint because if a machine *offers* a value for a particular symbol, it already "knows" the value and does not need to *perceive* it from its environment.

3. Steps are exhaustive.

$$\begin{array}{l} \forall\, U \in \mathbb{U} \text{ and } t \in stems(\mathcal{B}_P) : \\ \quad \exists\, s \in next_P(t) \text{ with } s^\tau \subseteq U \end{array} \tag{4.10c}$$

where the $next_P()$ function gives the set of steps in P that can follow a given prefix. Because P has, in general, no control over the data environment in which it exists, allowance must be made for every possibility. Informally (4.10c) says that for every stem t it contains, $\mathcal{B}_P$ contains sequences that define a next step for P under all possible circumstances, where "all possible circumstances" is represented by the elements of the total universe, $\mathbb{U}$. This definition is made in terms of the total universe $\mathbb{U}$ (rather than the valid universe $\mathbb{V}$) to ensure that the machine has defined behaviour even in the presence of invalid data. Note that requiring that the steps of a machine be exhaustive does not require that any action is allowed in any state of a model, as the decision part of a step can be *refuse* or *crash*. Rather, it requires that the reaction of a machine to any combination of action and perceived data is always *defined*.

4. Steps are unique.

$$\begin{array}{l} \forall\, t \in stems(\mathcal{B}_P) \text{ and } s1, s2 \in next_P(t) \ : \\ \quad con(s1^\tau \cup s2^\tau) \quad \Rightarrow \quad s1 = s2 \end{array} \tag{4.10d}$$

Informally (4.10d) says that if the machine has reached a particular point represented by the stem t and takes a further step s, then for a given element of the universe U there is only one possible next step. Furthermore, to generate distinct steps the data images of two steps must be inconsistent. This means that a machine cannot select different behaviour for a given element of the universe, depending on how much data it cares to take into account.

5. Machines are deterministic.

$$\begin{array}{l} \forall\, t \in \mathit{stems}(\mathcal{B}_P) \text{ and } s1, s2 \in \mathit{next}_P(t) \ : \\ \quad \mathit{con}({s1}^{\varepsilon} \cup {s2}^{\varepsilon}) \quad \wedge \quad {s1}^{\delta} = {s2}^{\delta} = \mathit{allow} \quad \Rightarrow \quad s1 = s2 \end{array} \tag{4.10e}$$

Informally (4.10e) says that if a machine has reached a particular point represented by the stem t and takes a further allowed step, then the step is uniquely determined by its **external data image**, comprising the action and perceives. In other words the machine cannot autonomously choose different updates to its offered data, which represents its state, when presented with given external data. In traditional state machine terms this corresponds to requiring at most a single outgoing transition from a given state with a given label, as two transitions with the same label would represent an autonomous choice of next state. Note that there may be multiple *crash* steps for a given external image so the result of a crash, in terms of update to offered data, is in general non-deterministic. [6]

4.2.3 Input and Output

The introduction to the concept of action in Protocol Modelling (Section 1.3.1) states:

For the purposes of this introduction, actions can be thought of as messages originating outside the protocol model (from the domain) and presented to the model.

This was a simplification as we allow actions, in general, to model both input and

[6] PM semantics gives possibilities, but does not, in general, define how an execution choice is made between possibilities. Under normal circumstances a machine will only crash if no *allow* step is available to it, however it is possible to imagine a scenario where this is not the case.

output. Given a step s in a machine, the action part s^α can be divided into:

$$\boxed{\begin{array}{l}\text{Input: } restr(s^\alpha, \mathcal{A}\backslash\mathcal{D}) \\ \text{Output: } restr(s^\alpha, \mathcal{D})\end{array}} \tag{4.11}$$

where $\mathcal{D}$ is the set of derived symbols in the universe, as defined in Section 4.1.5. In other words, the *derived fields* of an action are *output* and the *non-derived fields* are *input*.

A consequence of this definition is that the distinction between input and output is made by reference to the universe, as this defines which symbols are derived and which are not, so the Ω and $\mathcal{B}$ parts of a machine definition do not distinguish input and output. This property is exploited in the application of PM to choreography in Chapter 9 where a choreography is projected to different universes, whereby a given action is sending (output) a particular message in one universe and receiving (input) the message in another.

4.2.4 Initiation

The life of a machine starts with an *initiation step*. In addition to the rules defined in (4.10), the set of initiation steps $next_P(<>)$ for a machine P must obey:

$$\boxed{\forall\, si \in next_P(<>) :\ si^\alpha = \emptyset} \tag{4.12}$$

which requires that initiation can be use any action.

In a graphical representation such as *Account1* in Figure 1.2 on page 20 the initiation step takes a machine to its start state, shown as a solid black •. If a machine has a derived state, such as *Account2* or *Account3* in Figure 1.2, then after initiation the machine is in the state returned by the state derivation function.

Unless otherwise stated we shall use the term *step* in a generic way, not distinguishing between initiation steps and non-initiation steps.

4.2.5 Discussion of Completions

This section provides some informal illustrations of how completions are used to represent behaviour.

4.2.5.1 Interpretation of *Perceives* and *Offers*

Consider the example protocol model shown in Figure 1.2 on page 20. *Account1* makes the value of *balance* available to other composed machines, so *balance* is a symbol of a observation in the *offers* set of every step *Account1* makes. *Account1* is independent, in the sense defined in Section 4.2.8, because it does not need to use the attributes of any other machine, so its *perceives* set is always empty. By contrast, *Account2* and *Account3* have empty *offers* sets and need *balance* from *Account1* so always have *balance* as a symbol of a observation in their *perceives* set.

The definition of the *perceives* part of a step is made in terms of the values perceived at the **end** of the step, as opposed to the start of the step as perhaps might be expected. The motivation for this can be seen by considering the machine *Account3* in Figure 1.2 on page 20. This machine uses a post-state constraint whereby the decision made on an action is based on the value of *balance* perceived as a **result** of the action: in particular whether the balance is within the allowed limits or not. The implication of supporting post-state constraints is that the *decision* of a step can only be determined with reference to perceived values at the **end** of the step.

There is an obvious symmetry between the *perceives* and *offers*, whereby the *offers* of one machine provides values for the *perceives* of another, composed, machine. This will be exploited in the formalization of composition, in Section 4.3.1.

4.2.5.2 Illustration of Completions

Figure 4.1 shows two derived-state machines P and Q whose state is based on the value of an attribute x that belongs to another, composed, machine not shown in the figure. P allows the action a provided that x is true using a pre-constraint whereas Q allows the action a provided that x is true using a post-constraint. The lower part of the figure depicts scenarios for P and Q, with each line in each box representing a single *step*.

All the scenarios shown in Figure 4.1 are completions, starting with an initiation and having a sequence of *allow* steps with a final *refuse*. The scenarios 1 to 3 on the left are for P and 4 to 6 on the right are for Q. In each scenario, the reason for the final refusal is circled in red. The following points provide commentary on the six scenarios:

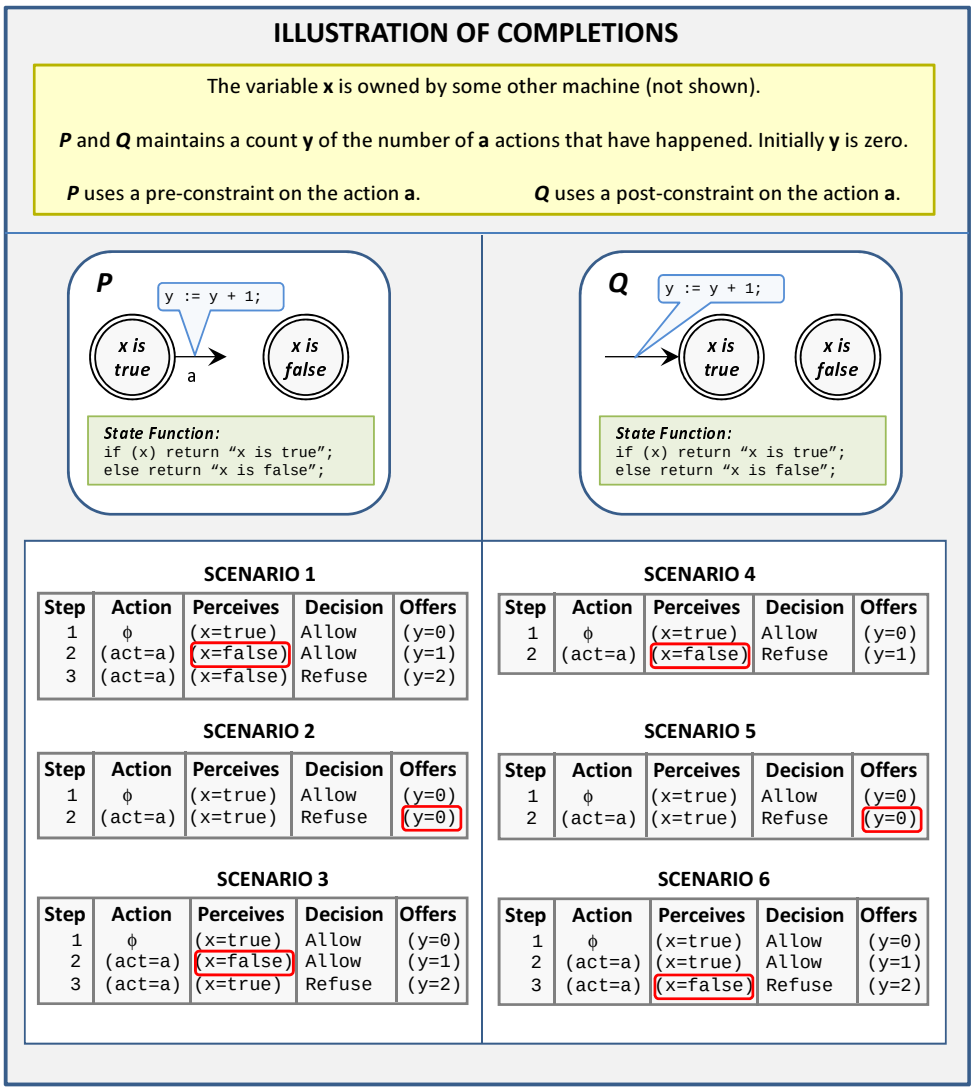


Figure 4.1: Illustration of Completions

- In all six scenarios, the model is initiated with $x = true$ and $y = 0$. The initiation is modelled as Step 1.
- In Scenario 1, after the first a has been allowed (at Step 2) the value of x is *false*. We do not know why this happens, as x is not owned by P. This means that the second a (at Step 3) is refused.
- In Scenario 2, Step 2 represents the wrong value of y, which should be 1 if the a were accepted. Therefore this step is refused. This illustrates the Exhaustiveness Rule (4.10c) as this rule requires that **all** possible values of the universe, includ-

ing ones that represent incorrect updates of the machine's own attributes, are considered.

- Scenario 3 is the same as Scenario 1 except that the final value of x is *true* rather than *false*. This makes no difference to the fact that a is refused in Step 3, as P uses a pre-constraint so its decision is based on the value of x **before** presentation of the action.
- In Scenario 4, the value of x is *false* after the first a (at Step 2) and because Q uses a post-constraint this means that the first a is refused.
- In Scenario 5, as in Scenario 2, the value of y in Step 2 is wrong so the step is refused.
- Scenario 6 is the similar Scenario 4 but x does not become false until after Step 3, so Step 2 is allowed.

4.2.6 Equality

Clearly if P and Q are defined over the same universe then $\mathcal{B}_P = \mathcal{B}_Q$ and $\Omega_P = \Omega_Q$ are sufficient conditions for two protocol machines to be equal. However it is not a necessary condition as P and Q, although equivalent, may be represented differently. Using the example from Section 4.1.6, if P has two steps whose actions are specified giving both possible values of `gender` but are otherwise identical; and Q had a corresponding single step that omits `gender`, the behaviour of P and Q would be the same. To eliminate such differences in expression we will use a *normal form* for expressing behaviour. Using a similar approach to that used for *fixes* in Section 4.1.6, we first define a procedure called *step expansion* that changes the way a machine is represented without changing its behaviour.

4.2.6.1 Step Expansion and Step Reduction

Consider a step s of a machine P. The implication of the uniqueness rule (4.10d) is that, for given values of all the symbols used in s, the behaviour of P does not depend on any symbol not in $symb(s)$. More formally, suppose that $t \in prefixes(\mathcal{B}_P)$ and that $y \in \mathcal{U} \backslash symb(s)$ where $s = last(t)$. We can replace the set $completions_P(t)$ in $\mathcal{B}_P$ by the

expanded set:

$$\{trunc(t)^\frown s'^\frown t'' \mid \{v\} \in restr(\mathbb{U}, \{y\}) \quad \wedge \quad t^\frown t'' \in \mathcal{B}_P\} \tag{4.13}$$

where:

$$s' = step(s^\alpha \cup v, s^\pi, s^\delta, s^\omega) \qquad \text{if } (y \in \mathcal{A})$$
$$s' = step(s^\alpha, s^\pi \cup v, s^\delta, s^\omega) \qquad \text{if } (y \in \mathcal{U} \backslash \mathcal{A})$$

without changing the behaviour of P. This is because all the steps that replace s are identical to s except for the addition of the new symbol, and all possible values of the symbol in $\mathcal{U}$ are covered. This replacement procedure is termed *step expansion*.

Using repeated application of step expansion it would be possible to render a machine in a format where every step is defined in terms of the *full symbol set* of universe, so that $\forall s \in asSet(\mathcal{B}_P) : symb(s) = \mathcal{U}$. If we required that a machine were always expressed in this way then we could use $\mathcal{B}_P = \mathcal{B}_Q$ and $\Omega_P = \Omega_Q$ as the test that $P = Q$. However there are occasions on which we will want to be able to extend the universe in which a protocol machine is defined, by adding symbols to it, without requiring that the machine be redefined. This happens when we compose two machines that are defined using different universes, as described in Section 4.3, and each machine now exists in a larger universe than that in which it was originally defined. For this reason we define a behavioural equivalence that does not rely on using every symbol of $\mathcal{U}$ for every step.

The opposite procedure to step expansion is termed *step reduction*. This entails, in the example above, replacing the expanded set defined by (4.13) with the original set $completions_P(t)$.

4.2.6.2 Behavioural Equivalence

Given two machines P and Q defined over the same universe $\mathfrak{U}$ we define an equivalence $\cong$ as follows:

$$\begin{aligned} &P \cong Q \Leftrightarrow \\ &\quad \Omega_P = \Omega_Q \quad \wedge \\ &\quad \forall\, U^* \in \mathbb{U}^* \;:\; decisions(matches_P(U^*)) = decisions(matches_Q(U^*)) \end{aligned} \tag{4.14}$$

The definition (4.14) says that two machines are equivalent if they offer the same set of symbols and, for a given sequence U^* of universe elements, the sequence of steps in each machine that matches U^* agree on the decision at each step. Note that $\cong$ is an equivalence relation. The equivalence classes of this relation are the behaviourally distinct machines in $\mathfrak{U}$.

4.2.6.3 Behaviour Normal Form

In order to be able to test equivalence of behaviour by equality of representation, we define a standard representation for a machine called *normal form* . This is defined as:

A representation of P in which no step reduction is possible	(4.15)

The notation ${\mathcal{B}_P}^{norm}$ is used to signify conformance to normal form; so if ${\mathcal{B}_P}^{norm} = \mathcal{B}_P$ then (4.15) is true in $\mathcal{B}_P$. Normal form always exists, as if a machine is represented using the full universe it is either in normal form or can be reduced; and if it can be reduced, the same argument applies to the result.

Normal form is unique by Theorem B.2 and we may therefore use this syntactic form to define semantic (behavioural) properties.

4.2.7 State

It will be useful to have a notion of the *state* of a protocol machine. In some ways this is problematical, as the behaviour of a dependent machine in reaction to an action can be determined by data that it does not own. Should this data be regarded as part of its "state" or not? It is easiest to do so, defining state as follows.

4.2.7.1 Definition of State

Given a machine P we define a relation Σ_P on the prefixes of P as follows:

t_1,$t_2 \in$ *prefixes*$(\mathcal{B}_P)$ then $t_1 \, \Sigma_P \, t_2 \Leftrightarrow$ for any sequence of steps t : $t_1{}^\frown t \in$ *prefixes*$(\mathcal{B}_P) \Leftrightarrow t_2{}^\frown t \in$ *prefixes*$(\mathcal{B}_P)$	(4.16)

which defines two prefixes as related if a continuation of one is also a continuation of the other. It is clear that Σ_P is an equivalence relation on *prefixes*$(\mathcal{B}_P)$. The *states* of P are the equivalence classes of Σ_P.

Fn: $state_P$. We use this to define the function $state_P :: prefixes(\mathcal{B}_P) \rightarrow \Sigma_P$ where the right hand side denotes the set of equivalence classes (the quotient set) of equivalence relation Σ_P over the set $prefixes(\mathcal{B}_P)$.[7]

4.2.7.2 Finite-State Protocol Machines

If the set Σ_P is finite then P is a *finite-state protocol machine*. In this case we can assume without loss of generality that there is a symbol $state_P \in \Omega_P$ called the *state attribute* with the property that:

$$\begin{array}{l} \forall\, t_1, t_2 \in prefixes(\mathcal{B}_P) : \\ \quad restr(t_1{}^{\omega}, \{state_P\}) = restr(t_2{}^{\omega}, \{state_P\}) \quad \Leftrightarrow \quad state_P(t_1) = state_P(t_2) \end{array} \tag{4.17}$$

so that the value of $state_P$ in a step uniquely determines the state of P. As the number of states is assumes finite, the values of $state_P$ could be constructed, for instance, by numbering the elements of Σ_P.

The state attribute of P can be seen as an enumerated attribute with values corresponding to the state icons of a graphical representation of the machine. Note that the a state attribute can be either stored or derived, being classed as derived iff it is possible to construct fixes for all values of $state_P$ that conform to (4.7).

4.2.8 Independence and Autonomy

Using normal form we can formalize the semantic concepts of independence (introduced in Section 2.2) and autonomy. A machine P is *independent* iff:

$$perceives(asSet(\mathcal{B}_P{}^{norm})) = \{\emptyset\} \tag{4.18}$$

which means that it is not reliant on any other machine to supply the values of perceived data. A machine that is not independent is *dependent*.

A machine P is *autonomous* iff:

$$actions(asSet(\mathcal{B}_P{}^{norm})) = \{\emptyset\} \tag{4.19}$$

An autonomous machine executes steps, without regard to the actions in the universe, until it gets to a refusal or a crash, after which it cannot proceed further.

[7] We have overloaded Σ_P to mean both the equivalence relation and its quotient set. From here on it refers to the latter.

Suppose that a protocol machine P is both independent and autonomous and, moreover, that there is a function asm_P:

$$asm_P :: \ restr(\mathbb{U}, \Omega_P) \rightarrow restr(\mathbb{U}, \Omega_P)$$

whereby the offers in step n is computed from the offers in step $n-1$, thus:

$$\forall\, t \in prefixes(\mathcal{B}_P)\ with\ t \neq <> :\ t^{\omega} = asm_P(trunc(t)^{\omega})$$

then P is an *Abstract State Machine*. This follows directly from the definition of Abstract State Machine given by Gurevich [32].

4.3 Composition

This section provides a formalization of machine composition. PM has two forms of composition:

- *Homogeneous composition*, where the machines being composed are defined over the same universe.
- *Heterogeneous composition*, where the machines being composed are defined over different universes.

We use the notation $\|$, borrowed from Hoare's CSP, for both homogeneous and heterogeneous composition and justify this use in the final section of this chapter by showing that the PM forms of composition have essentially similar semantics.

4.3.1 Formalization of Homogeneous Composition

We suppose that we have two machines, $P = \langle \mathfrak{U}, \Omega_P, \mathcal{B}_P \rangle$ and $Q = \langle \mathfrak{U}, \Omega_Q, \mathcal{B}_Q \rangle$ defined over the same universe $\mathfrak{U}$. We construct the *homogeneous* composition:

$$P \parallel Q = \langle \mathfrak{U}, \Omega_P \cup \Omega_Q, \mathcal{B}_{P\|Q} \rangle \tag{4.20}$$

where $\mathcal{B}_{P\|Q}$ is constructed as described in following sections.

4.3.1.1 Step and Completion Compatibility

Suppose that sp and sq are steps from completions in $\mathcal{B}_P$ and $\mathcal{B}_Q$ respectively. These two steps are deemed *compatible*, written $sp \sim sq$ iff the data images of the two steps are consistent:

$$sp \sim sq \Leftrightarrow con(sp^{\tau} \cup sq^{\tau}) \tag{4.21}$$

Compatibility of steps requires that, where the same symbol appears in the data image of both steps, both steps give it the same value. In particular, if one machine *offers* a value for a symbol and the other machine *perceives* a value for the same symbol, these values are the same.

Based on this we now define *compatibility of completions*. Suppose that $cp \in \mathcal{B}_P$ and $cq \in \mathcal{B}_Q$ then cp and cq are deemed *compatible*, written $cp \sim cq$, iff:

$$\begin{aligned} &\forall\, tp \in prefixes(cp), tq \in prefixes(cq) \text{ and } i \text{ with} \\ &\qquad 1 \leqslant i \leqslant\ min(length(tp), length(tq)) :\ tp_i \sim tq_i \end{aligned} \tag{4.22}$$

Informally, cp and cq being compatible means that they can co-exist as evolutions of the two machines without causing data inconsistency.

4.3.1.2 Step Composition

We define a *step composition* operator, $s1 \parallel s2$, on two steps $s1$ and $s2$ with $s1 \sim s2$ as follows:

$$(s1 \parallel s2)^{\alpha} = s1^{\alpha} \cup s2^{\alpha} \tag{4.23a}$$

$$(s1 \parallel s2)^{\pi} = (s1^{\pi} \cup s2^{\pi}) \backslash (s1^{\omega} \cup s2^{\omega}) \tag{4.23b}$$

$$\begin{aligned} (s1 \parallel s2)^{\delta} = \ & allow \quad \text{if } (s1^{\delta} = allow \ \wedge\ s2^{\delta} = allow) \\ & crash \quad \text{if } (s1^{\delta} = crash \ \vee\ s2^{\delta} = crash) \\ & refuse \quad \text{otherwise} \end{aligned} \tag{4.23c}$$

$$(s1 \parallel s2)^{\omega} = s1^{\omega} \cup s2^{\omega} \tag{4.23d}$$

This combines two steps to produce a "composite step":

$$s1 \parallel s2 = step\,((s1 \parallel s2)^{\alpha}, (s1 \parallel s2)^{\pi}, (s1 \parallel s2)^{\delta}, (s1 \parallel s2)^{\omega})$$

Note the following:

- (4.23b) removes (by set subtraction) the combined *offers* of the two composed steps. This models the fact that the values offered by one machine may resolve the values needed (perceived) by the other machine. As $s1 \sim s2$ (4.21) requires that it is not possible for one machine to perceive a different value for a given symbol from that offered by the other.
- In the absence of a *crash*, (4.23c) follows the CSP $\parallel$ rule, namely that if either step refuses the composite refuses.

It is clear from these definitions that $(s1 \parallel s2) = (s2 \parallel s1)$.

4.3.1.3 Machine Composition

Based on step composition we define *composition of completions*. Suppose that $cp \in \mathcal{B}_P$ and $cq \in \mathcal{B}_Q$ with $cp \sim cq$.

$$\begin{array}{lll} cp \parallel cq = \text{the sequence } c \text{ such that} & \forall\, t \in prefixes(c) : & \\ \qquad t_i = cp_i \parallel cq_i & \text{if } (cp_i,\ cq_i \text{ both defined}) & \\ \qquad t_i \text{ is undefined.} & \text{otherwise} & \end{array} \tag{4.24}$$

We need to restrict the range of i to values for which cp_i and cq_i are defined as either or both of cp and cq may be finite and so they may be of different lengths. The result of this construction will meet the definitions given in Section 4.2.1.2 by yielding either an infinite sequence of steps all with a decision of *allow*, or a finite sequence of *allows* followed by a final *refuse* or *crash*.

Finally we define the *composition construction on behaviours*:

$$\mathcal{B}_{P \parallel Q} = \{\ cp \parallel cq \ \mid\ cp \in \mathcal{B}_P \ \wedge\ cq \in \mathcal{B}_Q \ \wedge\ cp \sim cq\ \} \tag{4.25}$$

which says that the behaviour of $P \parallel Q$ is the set of composed completions for each compatible pair cq, cp. Each completion is constructed per (4.24).

Note that composition is defined even between machines that make incompatible updates to shared attributes. Suppose, for instance, that P is the machine *Account1* in Figure 1.2 on page 20 and that Q is a similar machine except that its update for a *Deposit* is:

```
balance := balance + Deposit.amount + 1;
```

In $P \parallel Q$ a *Deposit* that would be allowed in P would be refused in Q and vice-versa, so all deposits will be refused in $P \parallel Q$.

4.3.2 Machine Dependency

Reasoning about machine composition will be concerned with the nature of the dependencies between machines, as introduced in Section 2.2. This section gives a formal definition of dependency and explores the treatment of dependency in composition.

A machine Q *depends* on another machine P, written $P \rightarrow Q$, iff:

$$\exists\, sp \parallel sq \in asSet(\mathcal{B}_{P\parallel Q}) \text{ with } sq^{\pi} \cap sp^{\omega} \neq \emptyset \tag{4.26}$$

If $P \rightarrow Q$ then determination of at least one step of Q in $P \parallel Q$ cannot be made independently of the determination of the corresponding step in P, as it needs to use the offered data of P to determine how to behave. Note that if Q were an *independent machine*, per (4.18), then $sq^{\pi} = \emptyset$ in all steps so dependency is impossible.

The result of drawing all dependencies between the machines in a set is a directed graph with zero, one or two directed edges between each pair of nodes, the direction indicating the dependency. This is called the *dependency graph* of the set. It is possible for the dependencies to form a cycle. Suppose we have a set of machines defined over a universe $\mathfrak{U}$. The set has *circular dependencies* if the dependency graph is cyclic. In general, such a closed path represents a situation where composition does not yield a deterministic machine (one that obeys (4.10e)) as illustrated by the following case. Each of protocol machines P and Q has two different steps defined for an action `"go"` with mutual dependency between P and Q:

Steps for P:

$$\begin{aligned} &step(\{\texttt{(action="go")}\},\{\texttt{(Qstate="foo")}\},allow,\{\texttt{(Pstate="foo")}\}) \\ &step(\{\texttt{(action="go")}\},\{\texttt{(Qstate="bar")}\},allow,\{\texttt{(Pstate="bar")}\}) \end{aligned} \tag{4.27a}$$

Steps for Q:

$$\begin{aligned} &\mathit{step}(\{\texttt{(action="go")}\},\{\texttt{(Pstate="foo")}\},\mathit{allow},\{\texttt{(Qstate="foo")}\}) \\ &\mathit{step}(\{\texttt{(action="go")}\},\{\texttt{(Pstate="bar")}\},\mathit{allow},\{\texttt{(Qstate="bar")}\}) \end{aligned} \tag{4.27b}$$

Steps for $P \parallel Q$ (composition of (4.27a) and (4.27b)):

$$\begin{aligned} &\mathit{step}(\{\texttt{(action="go")}\},\emptyset,\mathit{allow},\{\texttt{(Pstate="foo"),(Qstate="foo")}\}) \\ &\mathit{step}(\{\texttt{(action="go")}\},\emptyset,\mathit{allow},\{\texttt{(Pstate="bar"),(Qstate="bar")}\}) \end{aligned} \tag{4.27c}$$

Here the composition (4.27c) gives two different possible allowed steps for the combination of *action* `"go"` and *perceives* $\emptyset$, in violation of (4.10e). So $P \parallel Q$ is not a protocol machine.

Now consider a set of machines whose dependencies are **acyclic**, so permitting representation as an acyclic graph. If two machines in the set are composed, the corresponding nodes are combined. As proved in Theorem B.3 it is always possible to choose a composition that preserves acyclicity, so that the graph after composition is also acyclic. An example is shown in Figure 4.2. This strategy can be repeated so that the whole set is reduced to a single machine and it is never necessary to compose machines with mutual dependency.

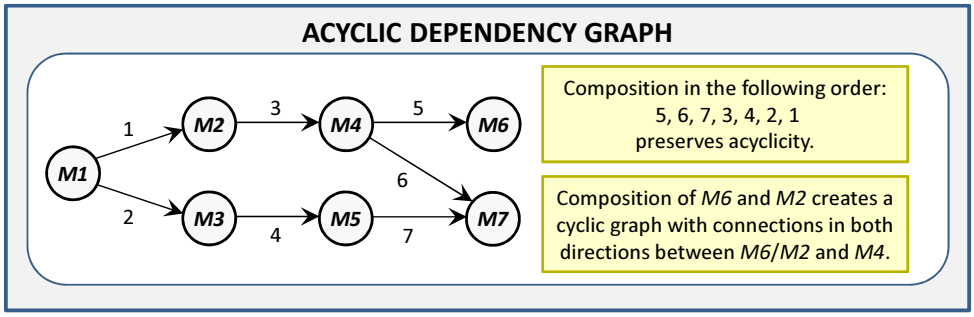


Figure 4.2: Acyclic Dependency Graph

We therefore require that:

A set of machines in composition has an acyclic dependency graph. (4.28)

Requiring that the ordering of composition ensures the preservation of acyclicity is **not a restriction** because, as we set out below, composition is commutative and associative and so the end result of composition is independent of the (pair-wise) order in which is carried out.

4.3.3 Properties of Homogeneous Composition

Homogeneous composition has the following properties:

Commutativity and Associativity. Theorem B.5 demonstrates that (4.25) is both commutative and associative. This means that the result of composing all members of a given set of machines is independent of the order used to do pairwise composition.

Closure of Composition. When two protocol machines are composed the result is also protocol machine, obeying the well-formedness conditions (4.10). This is proved as Theorem B.6. In particular, composition preserves deterministic behaviour, as it does in classical CSP.

> *Finally, the concurrent operator does not introduce non-determinism.*
> **– C.A.R. Hoare [39]**

Idempotence. From the definitions of step composition (4.23) it is clear that composition is *idempotent*, so if P is a protocol machine then $P \parallel P = P$.

Decomposition Uniqueness. Decomposition is unique, meaning that given a step in a composition of two machines $P \parallel Q$ there are unique steps of $\mathcal{B}_P$ and $\mathcal{B}_Q$ from which it is formed. In other words:

$$\begin{aligned} & t \in \mathit{stems}(\mathcal{B}_{P\parallel Q}) \quad \wedge \quad s1, s2 \in \mathit{next}_{P\parallel Q}(t) \quad \wedge \\ & \quad (s1 = s2) \wedge (s1 = sp1 \parallel sq1) \wedge (s2 = sp2 \parallel sq2) \quad \Rightarrow \\ & \quad (sp1 = sp2) \wedge (sq1 = sq2) \end{aligned}$$

and this is established in Theorem B.7.

Uniqueness of decomposition means that given $t \in \mathcal{B}_{P\parallel Q}$ the components $tp \in \mathcal{B}_P$ and $tq \in \mathcal{B}_Q$ such that $t = tp \parallel tq$ are uniquely defined by t, so we can use the following notation:

$$t\upharpoonright_P = tp$$

$$\mathcal{B}_{P\parallel Q}\upharpoonright_P = \{t\upharpoonright_P \mid t \in \mathcal{B}_{P\parallel Q}\}$$

From this it follows that:

$$\mathit{prefixes}(\mathcal{B}_{P\parallel Q}\upharpoonright_P) \subseteq \mathit{prefixes}(\mathcal{B}_P) \tag{4.29}$$

Normal Form in Composition. Composition does not, in general, preserve normal form (as defined in Section 4.2.6.3) as can be seen from the following example. Suppose that P and Q are independent machines (in the sense defined in Section 4.2.8) defined over the same universe and with $\Omega_P = \Omega_Q = \emptyset$. Suppose that $\mathcal{A} = \{a, b, c\}$ and that these symbols are booleans. If:

- P *allows* any step with $(a \wedge b) \vee (\neg\, a \wedge c)$ and *refuses* otherwise
- Q *allows* any step with $(a \wedge c) \vee (\neg\, a \wedge b)$ and *refuses* otherwise

then $P \parallel Q$ will *allow* iff $(b \wedge c)$, and the value of a is immaterial. Clearly a could be removed by step reduction in $P \parallel Q$ to recover normal form.

4.3.4 Formalization of Heterogeneous Composition

A consequence of the definition of input/output in Section 4.2.3 is that output action data produced by one machine cannot be consumed as input by another machine with which it is in homogeneous composition, as which symbols are input and which are output is fixed by the universe. This means that there is no analogue in homogeneous composition of the kind of composition seen in CSP and CCS using "reactions" on input/output, using notations like `P?a` (to read from process `P` into variable `a`), and `Q!a` (to send `a` to process `Q`). For this we need to consider composition *across universes*, called *heterogeneous composition*.

Suppose that we have two universes:

- $\mathfrak{U} = \langle \mathcal{U},\ \mathcal{A},\ \mathcal{D},\ \mathbb{V},\ \mathit{fixes} \rangle$
- $\mathfrak{U}' = \langle \mathcal{U}',\ \mathcal{A}',\ \mathcal{D}',\ \mathbb{V}',\ \mathit{fixes}' \rangle$

with compatibility conditions:

$$(\mathcal{A} \cap \mathcal{U}') \backslash \mathcal{A}' = (\mathcal{A}' \cap \mathcal{U}) \backslash \mathcal{A} = \emptyset \tag{4.30a}$$

$$\mathcal{D} \cap \mathcal{D}' = \emptyset \tag{4.30b}$$

$$\mathit{restr}(\Cup \mathbb{V}, \mathcal{U}') = \mathit{restr}(\Cup \mathbb{V}', \mathcal{U}) \tag{4.30c}$$

where:

- Condition (4.30a) requires that the two universes have the same separation of fields (used in actions) from attributes (used for machine storage);
- Condition (4.30b) requires that a shared symbol cannot be derived (potentially in different ways) in both universes.
- Condition (4.30c) requires that where a symbol is shared between two universes the two universes give the symbol the *same type*, so that a value valid in one universe will also be valid in the other.

We define the *combined universe* $\mathfrak{U}'' = \mathfrak{U} \boxplus \mathfrak{U}'$ as follows:

$$\mathfrak{U} \boxplus \mathfrak{U}' = \langle\, \mathcal{U} \cup \mathcal{U}',\ \mathcal{A} \cup \mathcal{A}',\ \mathcal{D} \cup \mathcal{D}',\ \mathbb{V}'',\ \mathit{fixes}'' \,\rangle \tag{4.31a}$$

The last two components of the tuple, $\mathbb{V}''$ and fixes'', are defined as follows.

$$\mathbb{V}'' = \{\, V \mid V \in \mathbb{U}'' \ \wedge\ \mathit{restr}(V,\mathcal{U}) \in \mathbb{V} \ \wedge\ \mathit{restr}(V,\mathcal{U}') \in \mathbb{V}' \,\} \tag{4.31b}$$

For fixes'' we use the function fixes° where:

if $\mathbb{W} \in \mathit{power}(\mathit{power}(\mathcal{E}))$ then:

$$\mathit{fixes}^\circ(\mathbb{W}) = \begin{cases} \emptyset & \text{if } (\mathbb{W} = \emptyset) \\ \bigcup\limits_{W \in \mathbb{W}} \mathit{fixes}^\circ(W) & \text{otherwise} \end{cases}$$

or if $W \in \mathit{power}(\mathcal{E})$ then:

$$\mathit{fixes}^\circ(W) = \begin{cases} \{W\} & \text{if } (\mathit{symb}(W) \cap \mathcal{D}'' = \emptyset) \\ \mathit{fixes}^\circ\Big(\Big\{ w = \bigcup\limits_{v \in W} \xi(v) \;\Big|\; \xi(v) \in \mathit{fixes}^\circ(v) \wedge \mathit{con}(w) \Big\}\Big) & \text{otherwise} \end{cases}$$

or if $v \in \mathcal{E}$ then:

$$\mathit{fixes}^\circ(v) = \begin{cases} \mathit{fixes}(v) & \text{if } (v^{\mathit{symb}} \in \mathcal{D}) \\ \mathit{fixes}'(v) & \text{if } (v^{\mathit{symb}} \in \mathcal{D}') \\ \{\{v\}\} & \text{otherwise} \end{cases}$$

in:

$$\textit{fixes}''(v) = (\textit{fixes}^{\circ}(\{v\}))^{norm} \tag{4.31c}$$

where we require in (4.31c) that $\textit{fixes}''(v)$ is defined for all v with $v^{symb} \in \mathcal{D} \cup \mathcal{D}'$.

This definition equates to the following algorithm:

a. Given a derived observation, create a "proto-fix" by replacing it with the set of fixes given by the universe in which it derived (condition (4.30b) requires that it is not derived in both universes).

b. in each member of the proto-fix, replace each derived observation by its fixes, given in the universe in which it derived, limiting to those that are consistent with the receiving member of the proto-fix. Where an observation has multiple fixes, the receiving proto-fix element is cloned to receive each one.

c. Repeat this recursively, until all derived observations have been eliminated.

Theorem B.8 shows that, provided there are no cycles in the symbol dependencies induced by derivation, then the combined universe $\mathfrak{U}''$ formed according to (4.31) observes the well-formedness conditions (4.7) if both $\mathfrak{U}$ and $\mathfrak{U}'$ do so. Note the following:

- Normalisation of the raw result from $\textit{fixes}^{\circ}$ is necessary, as can be seen by considering the example of $x = a \vee b$ in $\mathfrak{U}$ and $b = a \wedge c$ in $\mathfrak{U}'$ where x, a, b, c are booleans.

- It is possible that not all values allowed in the source universes are allowed in the combined universe, as can be seen by considering the example[8] of $x = a \veebar b$ in $\mathfrak{U}$ and $a = b$ in $\mathfrak{U}'$ where x, a, b are booleans. However, at least one value will always remain, so the symbol is not removed from the universe. This is because (4.7f), which requires that every derived symbol has at least one fix consistent with every member of the valid universe, is true in the combined universe.

Now suppose we have machines P and Q defined over $\mathfrak{U}$ and $\mathfrak{U}'$ respectively. The heterogeneous composition of P and Q is then defined, as in (4.20), as:

$$P \parallel Q = \langle \mathfrak{U} \boxplus \mathfrak{U}', \Omega_P \cup \Omega_Q, \mathcal{B}_{P \parallel Q} \rangle \tag{4.32}$$

[8] Using $\veebar$ for exclusive or.

The formation $\mathcal{B}_{P\|Q}$ in (4.32) requires that the steps of P and Q are defined for all elements of the combined universe. However this is always the case as follows. Consider an element $U'' \in \mathbb{U}''$. As $restr(U'', \mathcal{U}) \in \mathbb{U}$, U'' selects a single step in P. Similarly, as $restr(U'', \mathcal{U}') \in \mathbb{U}'$, U'' also selects a single step in Q.

In a heterogeneous composition a symbol $d \in \mathcal{D}$ of an action of compatible steps of P and Q that both use d is "sent" by P and "received" by Q; and similarly a symbol $d' \in \mathcal{D}'$ is "sent" by Q and "received" by P. Note that, in the terminology discussed in Section 2.1, this is **synchronous communication**. This form of model is appropriate to represent the way software components interact using method calls or procedure invocation. Later, in Chapter 9, we discuss **asynchronous communication** in the context of choreographed collaborations between independent participants. In common with (most) other process algebras, protocol modelling does not class asynchronous collaboration as composition; and different forms of argument are used to reason about it.

4.3.5 Properties of Heterogeneous Composition

We do not provide a detailed analysis but some points to note are:

- The results concerning associativity, commutativity, closure of composition and uniqueness of composition construction given in Section 4.3.3 for homogeneous composition are unchanged for heterogeneous composition.
- Heterogeneous composition is **not idempotent**, as the machines involved are defined using different universes.

4.4 Protocol Models

This section uses the composition mechanism to formalize the idea of a *protocol model*, as introduced in Section 1.3.5, and discusses the idea of *well-behaved* machines and models. A well-behaved machine is one that can ensure that its data image remains within the valid universe $\mathbb{V}$ and so preserves data integrity.

4.4.1 Formalization of Model

Suppose that we have a set of protocol machines defined over the universe $\mathfrak{U}$ and that the parallel composition of all the machines is a protocol machine M. The machine M

is a *protocol model* of $\mathfrak{U}$ iff:

$$\Omega_M = \mathcal{U} \backslash \mathcal{A} \tag{4.33}$$

In other words the offers of M gives a value to every non-action symbol in the universe. This means that M is independent, per (4.18), as there are no symbols left in $\mathcal{U}$ that can be used to define the perceives of M.

The building blocks of a model are machines defined without composition, and these are referred to as the *atomic machines* of the model.

4.4.2 Well-Behaved Machines

The formal definitions of a protocol machine in Section 4.2.2 are made in terms of the **total universe**, $\mathbb{U}$. There is no guarantee that a machine will preserve *data integrity* by ensuring that the data image remains in the **valid universe**, $\mathbb{V}$. So, for instance, there is no guarantee that a step in a machine might not give a value to a derived symbol that conflicts with values pertaining in the basis symbols. In building protocol models we will want to ensure that the models we create guarantee data integrity, and to this end introduce the notion of a *well-behaved machine*.

4.4.2.1 Definition of Well-Behaved Machine

A machine P is said to be *well-behaved* iff it can be expressed using steps that obey:

$$\forall\, s \in asSet(\mathcal{B}_P): \quad s^\delta = allow \;\Rightarrow\; closed(s^\tau) \;\wedge\; s^\tau \Subset \mathbb{V} \tag{4.34}$$

This is understood as follows. If a well-behaved machine allows a step, (4.34) requires that the new **total data image** of the machine will conform to $\mathbb{V}$. This means, in particular, that when the machine advances to a new state, all attributes in the machine's data image will conform to their type and, because (4.34) requires that steps are closed, all derived machine attributes and derived (output) action fields in the machine's data image will be correctly computed from their base attributes.

A well-behaved model of a bank account would guarantee for example, that non-numeric amounts in a deposit or withdraw action would not be allowed, and that the derived state *in credit* or *overdrawn* is correctly derived from the balance at all

times. A language that preserves abstractions, as *in credit* and *overdrawn* are abstractions of *balance*, is known as a *safe language* [69]. The main task of this chapter is to consider whether safety is preserved under composition, so that a model built from well-behaved machines is itself well-behaved.

4.4.2.2 Abstractness of Well-Behaved Property

Because the definition given above deems a machine well-behaved if *any* representation obeys (4.34), the definition is abstract, in the sense that it is true of a machine independent of whether a particular representation obeys (4.34).

If a machine is well-behaved then, as shown in Theorem B.9, the normal form representation will obey (4.34). As we assume that machines are expressed in normal form we can use (4.34) to discriminate between those machines that possess the property of being well-behaved and those that do not.

4.4.2.3 Homogeneous Composition of Well-Behaved Machines

We now show that a model constructed by homogeneous composition of well-behaved machines is itself well-behaved. We suppose that P and Q are both well-behaved and take steps sp and sq respectively giving a composed step $sp \parallel sq$. We show that $sp \parallel sq$ obeys (4.34). We assume that:

$$(sp \parallel sq)^{\delta} = allow \tag{4.35}$$

As by (4.35) and (4.23c) we must have $sp^{\delta} = sq^{\delta} = allow$, and as P and Q are well-behaved we have:

$$closed(sp^{\tau}) \wedge closed(sq^{\tau}) \tag{4.36}$$

which gives $closed((sp \parallel sq)^{\tau})$.

In addition:

$$sp^{\tau} \Subset \mathbb{V} \quad \wedge \quad sq^{\tau} \Subset \mathbb{V} \tag{4.37}$$

As $closed(sp^{\tau})$ and $closed(sq^{\tau})$, (4.7f) requires that $(sp \parallel sq)^{\tau} \Subset \mathbb{V}$. This establishes (4.34) for $P \parallel Q$.

4.4.2.4 Heterogeneous Composition of Well-Behaved Machines

The argument given above does not work for heterogeneous composition. The fundamental reason for this is that, while P and Q may be well-behaved in their own universes, this does not mean that they are well-behaved in each other's. We now show sufficient conditions to guarantee that a machine created by heterogeneous composition of two machines well-behaved in their own universes is well-behaved in the combined universe. Suppose that P and Q are defined over universes $\mathfrak{U}$ and $\mathfrak{U}'$ respectively. The conditions are:

$$\mathcal{D}\backslash\mathcal{A} \subseteq \Omega_P \quad \wedge \quad \mathcal{D}'\backslash\mathcal{A}' \subseteq \Omega_Q \tag{4.38a}$$

$$\begin{aligned} &(sp \parallel sq)^{\delta} = allow \Rightarrow \\ &\quad symb(sp^{\alpha}) \cap \mathcal{D}' \subseteq symb(sq^{\alpha}) \cap \mathcal{D}' \quad \wedge \\ &\quad symb(sq^{\alpha}) \cap \mathcal{D} \subseteq symb(sp^{\alpha}) \cap \mathcal{D} \end{aligned} \tag{4.38b}$$

The proof theat these conditions are sufficient to ensure that the compoistion is well-behaved is given as Theorem B.10. These conditions are interpreted as follows:

- The first condition (4.38a) requires that, if a machine P used as a component of M is defined over a universe $\mathfrak{U}$, then the offers of P must include all the non-action derived symbols $\mathcal{D}\backslash\mathcal{A}$ of $\mathfrak{U}$. This implies that the model M is *computationally complete* in the sense that every non-action derived symbol is given a value in every step.
- In the second condition (4.38b), $symb(sq^{\alpha}) \cap \mathcal{D}'$ represents the fields output (sent) by Q to P and $symb(sp^{\alpha}) \cap \mathcal{D}'$ represents the fields received by P from Q. This condition requires that *messaging is coherent* in the sense that the behaviour of the receiver cannot be based on more fields than the sender has actually sent.

4.4.2.5 Well-behaved Models

A *well-behaved model* is one built by composition from atomic machines that are well-behaved and expressed with steps that obey (4.34), and in which all heterogeneous compositions obey the conditions (4.38). As a consequence of the above results, all machines in the model, from the atomic machines up to the model itself, will obey (4.34).

4.4.3 Robust Machines

The semantics of steps in a PM machine (4.8) include the possibility that a step may *crash*. This happens when computations entailed in determining the new state of the machine fail a pre-condition (see Appendix C where the description of the Semantic Function includes discussion of the treatment of pre-condition failure). However it is possible that a machine is built so that it "checks its own pre-conditions" and, in the event that it determines that pre-conditions fail, **forces refusal of the step**. In effect, the function that determines the new state then has no pre-conditions, but instead sometimes returns a result that coerces refusal of the current step, leaving the state of the model unchanged.

We define a *robust machine* to be one in which a *crash* is impossible:

$$P \text{ is robust } \Leftrightarrow \mathit{crash} \notin \mathit{decisions}(\mathit{asSet}(\mathcal{B}_P)) \tag{4.39}$$

At least at the level of semantics it is easy to render any machine robust. Given any non-robust P, we define another machine $\mathit{robust}(P)$ in which any step s with $s^\delta = \mathit{crash}$ is replaced by $s' = \mathit{step}\,(s^\alpha, s^\pi, \mathit{refuse}, s^\omega)$. This converts P into a robust machine. Note the following:

- If $\mathit{robust}(P) = P$ then P is already robust.
- If P and Q are robust then so is $P \parallel Q$. This is because by (4.23c) composition cannot introduce a *crash* where it doesn't exist in the component machines.

Unless otherwise stated, the default assumption is that a protocol machine or model is **not** robust unless specifically rendered so as above.

Being robust and being well-behaved are independent properties, in that possession of one does not imply possession of the other. Being robust is a weaker property than being well-behaved as it confers no guarantees of data integrity, in particular that a derived symbol in an allowed step has a value correctly computed from its basis symbols.

4.4.4 Stored-State and Derived-State Machines

Using the notion of state defined in Section 4.2.7 it is then possible to distinguish two types of well-behaved finite-state protocol machine: *stored-state* and *derived-state*. A well-behaved finite-state machine P is stored-state if its state in Σ_P is independent of P's data environment, which is the case iff:

$$\forall\, s \in asSet(\mathcal{B}_P) \text{ with } s^\delta = allow : \quad closed(restr(s^\omega, (\mathcal{U}\backslash\mathcal{D}) \cup \{state_P\})) \tag{4.40}$$

which requires that, if the stare attribute is derived, it is fixed completely by the offered attributes of the machine. This definition requires that the machine is well behaved, so that if $state_P$ is derived the well-formedness condition (4.7c) guarantees that it has a single value.

The significance of a stored-state machine is that its behaviour can be depicted as a conventional labelled transition system, without the need to assign a state to every element of the machine's universe. This can be done for a stored-state machine P as follows:

- Draw a set of nodes $\mathcal{N}$ of the state transition system corresponding to the distinct values of $restr(\mathbb{V}, \Omega_P\backslash\mathcal{D})$. By (4.40) and (4.7c), each node in $\mathcal{N}$ maps to a single state in Σ_P.
- By the definition (4.16), a state in Σ_P defines the set of continuation traces and therefore the set of next steps. For each node in $\mathcal{N}$, draw an outgoing transition for every allowed step possible from the corresponding state in Σ_P. The label for the transition for a step s is s^α and the destination node is the node corresponding to $restr(s^\omega, \Omega_P\backslash\mathcal{D})$.

The nodes are an abstraction of the offered data, Ω_P, of the machine. Because this abstraction contains no derived attributes it is independent of the data environment (the perceived data) of each step which does not therefore need to be reflected in the set of nodes used to depict the behaviour of the machine.

A finite-state machine that is not stored-state is termed *derived-state*. A derived-state machine is dependent on data it perceives in its environment to determine how to behave, so its behaviour cannot be depicted as a conventional state transition diagram and instead requires a *state function* to determine the state.

Note that a finite-state machine that is independent is also stored-state, but not necessarily the other way round. This is because steps with different perceived data in s^π may give rise to different values of derived attributes in Ω_P even if they do not give rise to a different states in Σ_P.

4.4.5 Modes of Use

There are two *modes of use* of composition in modelling which are found to be particularly useful in practice and merit special mention. These modes of use are termed *constructive* and *regulatory*.

Constructive. The *constructive* mode of use is characterized by a set of machines where the dependency graph of the set is acyclic and the machines' owned data are disjoint sets, so for any P and Q with $P \neq Q$ we have $\Omega_P \cap \Omega_Q = \emptyset$. This pattern realizes the notion that *a protocol machine owns a set of stored attributes, which only it can alter*, introduced in Section 1.3.3. The discipline of requiring that composed machines use disjoint sets of symbols echoes the idea of encapsulation of data (*instance variables*) in object orientation. Application of this mode of use is explored in Part III, Chapter 7, discussing protocol machines as a medium for object modelling.

This mode uses heterogeneous composition, so that each machine has its own universe. Thus if P is a derived state machine and is dependent on a symbol y in its perceived data environment, but doesn't care how y is computed, then in P's universe y will not be derived. However, some other machine Q with $y \in \Omega_Q$ may be responsible for derivation of y from other data, so y will be a derived attribute in Q's universe.

Regulatory. The *regulatory* mode of use is characterized by the pattern $P \parallel Q = P$ where P and Q are both protocol machines. In this pattern P and Q share owned data, as we must have $\Omega_Q \subseteq \Omega_P$. This mode is concerned to specify behavioural conformance between machines, where Q can be thought of as specifying a *behavioural contract* that P obeys. This mode is used in Chapter 9, in the context of defining the behaviour required from participants in a multiparty collaboration.

This mode uses homogeneous composition, as it requires that the offered data of the composed machines overlap, and that idempotence is supported. In particular, a

symbol y derived in P's universe will also be derived in Q's, so that Q can specify how P should perform the derivation.

4.5 Alphabet

The definition of behaviour of a machine is based on the set of actions in the universe over which the machine is defined. The property (4.10c) demands that behaviour of a machine is exhaustive, so is defined in terms of all possible actions in the total universe, including *those that have no effect on the machine's state*. Making the definition of behaviour exhaustive in this way greatly simplifies the formalization of behaviour and composition, but for the purposes of making descriptions of machines, such as that in Figure 1.2 on page 20, we want to distil from the full universe of actions those that actually affect the state of the machine from those that do not. An action that has no effect on the state of a machine in any context is one which we say the machine *ignores*. In this chapter we show how to recognize such actions. This then allows us to recognize the set of actions that a machine **does not ignore** and thereby give meaning to the concept of a machine's *alphabet*.

4.5.1 Definition of Ignore

In a simpler algebra, where we are not concerned with data and machines can be described simply in terms of traces of actions represented as single symbols, the idea of a machine ignoring an action might be captured by saying that P *ignores* an action symbol a iff:

$$t \in traces_P \;\Rightarrow\; t^\frown a \in traces_P \;\wedge\; state_P(t^\frown a) = state_P(t) \tag{4.41}$$

where $state_P(t)$ is defined on traces as described in Section 4.2.7.1. This definition means that the action a is always allowed and never alters the state of the machine.

We now make an equivalent definition for PM. Suppose P is a finite-state protocol machine defined over a universe $\mathfrak{U}$ and Ω_P contains a state attribute $state_P$ as described in Section 4.2.7.2. For a set of observations $A \in restr(\mathbb{U}, \mathcal{A})$ we say that P *ignores* A iff:

$$\begin{array}{l} \forall\, t \in stems(\mathcal{B}_P) \text{ and } s \in next_P(t) \text{ with } con(A \cup s^\alpha) \;: \\ \quad s^\tau \Subset \mathbb{V} \;\wedge\; restr(s^\omega, \Omega_P \backslash \mathcal{D}) = restr(t^\omega, \Omega_P \backslash \mathcal{D}) \;\Rightarrow\; s^\delta = allow \end{array} \tag{4.42}$$

This definition requires that P has a state attribute $state_P$ to ensure that the offered data Ω_P encodes the machine's behavioural state. If this were not the case, the preservation of value of stored attributes required by the definition would not achieve a preservation of behaviour.

According to (4.42), given a valid external data image there is always an allowed step for the action A that preserves the values of the non-derived offered symbols of the machine. Moreover because of the deterministic behaviour condition (4.10e), this is the only step that is allowed for this external data image. This is analogous to (4.41). If the external data image is not valid, the decision of the machine for a given action is immaterial to the determination of whether the action is ignored or not.

4.5.2 Closure of Ignore

If if P and Q both meet conditions (4.42) for an action A then so does $P \parallel Q$. The proof of this is given as Theorem B.12.

4.5.3 Definition of Alphabet

In Section 1.3.3, we argued to develop a theory of behaviour without the concept of an alphabet: simply distinguishing *allow* and *refuse/crash* but without differentiating between allowed actions that cause a change in state and those that do not. We exploit this in the formalization of behaviour, which requires that all the machines being composed within a model have behaviour defined in terms of the full set of actions available to the model, $restr(\mathbb{U}, \mathcal{A})$. Having proceeded on this basis for the formalization of behaviour (Section 4.2.2) and composition (Section 4.3.1), the theory described in this section gives a formal meaning to the *alphabet* of a machine: being the set of actions that the machine **does not ignore**.

We define:

$$ignores_P = \{\ A \mid A \in restr(\mathbb{U}, \mathcal{A}) \ \wedge \ \text{(4.42) holds for } A \text{ in } P\ \} \tag{4.43}$$

and:

$$alphabet_P = \{\ A \mid A \in restr(\mathbb{U}, \mathcal{A}) \ \wedge \ A \notin ignores_P\ \} \tag{4.44}$$

It is arguable that the term *alphabet* is not very apposite, as the elements are sets of observations, rather than single symbols. However it is established by usage and we stick with it.

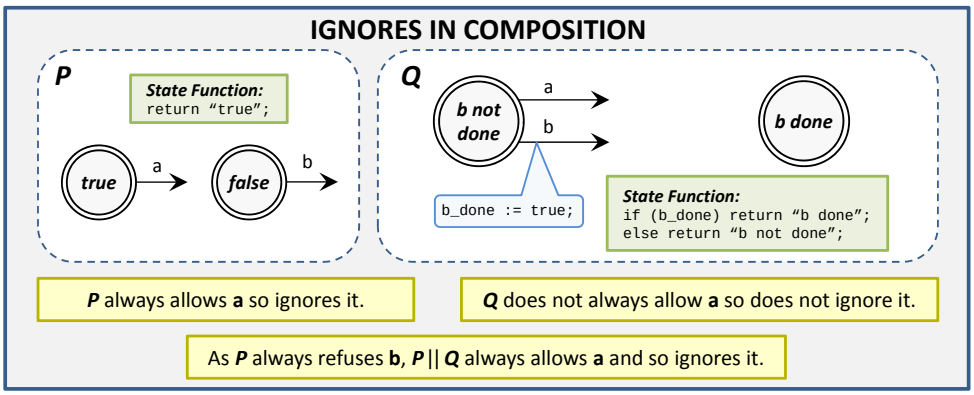


Figure 4.3: Ignores in Composition

The relationship between the alphabets of a composition and that of the components is as follows.

Homogeneous composition. In homogeneous composition:

$$alphabet_{P\|Q} \quad \subseteq \quad alphabet_P \quad \cup \quad alphabet_Q \tag{4.45}$$

The equivalent law in classical CSP[9] is different: $alphabet_{P\|Q} = alphabet_P \cup alphabet_Q$. This is because *alphabet* in classical CSP is not determined formally on behaviour. By this rule, the action a in Figure 4.3 would be deemed to be in the alphabet of the composite $P \parallel Q$ simply by virtue of being in the alphabets of P and Q, irrespective of how it figures in the behaviour of the composite.

Heterogenous composition. For heterogeneous composition, the equivalent relationship is:

$$alphabet_{P\|Q} \quad \subseteq \quad \{\, A \cup A' \mid \; A \in alphabet_P \;\vee\; A' \in alphabet_Q \,\} \tag{4.46}$$

reflecting the fact that the actions of the combined universe are defined in terms the combined action symbol set $\mathcal{A} \cup \mathcal{A}'$.

[9]By "classical CSP" we mean as described by Hoare in [38], as opposed to the "alphabetized version of CSP" described by Roscoe in [70].

4.5.4 Graphical Alphabet

Some care needs to be taken relating the formalisation of alphabet given above with the alphabet as shown in the graphical syntax of PM as used, for instance, in Figure 1.2 on page 20. In the context of the graphical syntax we will normally represent the alphabet of a machine as a set of *action types*, such as $\{ Open, Deposit, Withdraw, Close \}$. If the graphical representation of a machine P has an action type `x` listed as a member of its alphabet, then this relates to the formal definition of alphabet (4.44) as follows:

$$A \in restr(\mathbb{U}, \mathcal{A}) \wedge \texttt{(action\_type = x)} \in A \quad \Rightarrow \quad A \in alphabet_P$$

However, an action type field is not required by the formal semantics of PM and the definition of a protocol model does not necessarily require its use. This gives the ability to define action abstractions used, for example, in the bank account mixins shown in Figure 7.5 in Chapter 7. Here the machines *Suspension*, *Balance* and *Limit Control* use abstractions *Funds In* and *Funds Out* to model money movement in to or out of the account independently of whether enacted by action type *Deposit*, *Withdraw* or *Transfer*.

4.6 Objects

In the introduction to this thesis it was claimed that PM can be used to create object models: models that comprise a population into which new individuals can be introduced, or instantiated. This section examines how PM supports object models.

The ideas in this chapter are realized in the modelling tool, ModelScope, described in Chapter 7.

4.6.1 Object Identity

In order to support a concept of objects, we assume that every protocol machine is endowed with a set of *object identities*. A machine is given this set of identities in its initiation step, and thereafter it is immutable so cannot be changed by any subsequent action.

Given a machine P the function $\iota(P)$ returns the set of object identities in P. The object identities in a machine are public, and are encoded in the offers part of each

step. The function ι must obey the rule that if two machines, P and Q, are composed then:

$$\iota(P \parallel Q) = \iota(P) \cup \iota(Q) \tag{4.47}$$

This mirrors the composition rule (4.23d).

4.6.2 Object Machines and Object Models

We use the term *object machine* for a machine that has a single object identity, so P is an object machine iff $|\iota(P)| = 1$. For object modelling we work with a protocol model that is a composition of object machines. We will denote an object machine by giving it a superscript of o, thus: P^o.

Two or more object machines in a model may have the same object identity. An *object* is the composition of machines that share the same object identity. An *object model* is a model that is a composition of object machines. Suppose M is an object model that comprises the set S of object machines, so that:

$$M = \prod_{P^o \in S} P^o \tag{4.48}$$

where Π is used to denote $\parallel$ composition over a population of processes. Allowing that the composition in (4.48) may be heterogeneous, we use $\mathfrak{U}_M$ to denote the combined universe of all the components of M.

The object P in M with identity O is:

$$P = \prod_{P^o \in S \ \wedge \ \iota(P^o) = \{O\}} P^o \tag{4.49}$$

We also assume that every action is endowed with a set of object identities, encoded in the fields of the action, which may be viewed as specifying the objects in the model to which the action is addressed. Note that, in general, a given action may be addressed to many objects. If A is an action then $\iota(A)$ returns the set of object identities in A.

Based on this, we define the following well-formedness conditions on a model, M:

$$\bigcup_{A \in restr(\mathfrak{U},\mathcal{A})} \iota(A) \subseteq \iota(M) \tag{4.50}$$

which says that the objects addressed by A must be present in the model; and

$$\iota(A) \cap \iota(P) = \emptyset \Leftrightarrow A \in ignores_P \tag{4.51}$$

which says that an object ignores an action that is not addressed to it.

4.6.3 Instantiation

Instantiation in PM is a matter of "waking up" rather than "bringing into existence". The intuition is that all objects are present in a model from its initiation, but an object is silent and invisible until in receipt of an action that it does not ignore. The prime motivation and advantage of this unusual way of constructing the notion of instantiation is that it allows lines of formal reasoning developed in the context of a fixed model (one that has a fixed population of machines) to be carried over to models with a dynamic population, as there is no difference between the two kinds of model.

Suppose we have two machines P and Q with $P \neq Q$ in a finite-state well-behaved model, and moreover suppose that $\Omega_P \cap \Omega_Q = \emptyset$. We say that Q is *oblivious* of P iff:

$$\begin{aligned} t \in prefixes(\mathcal{B}_Q) \ \wedge \ & (restr(actions(asSet(t)),\mathcal{U}_P) \Subset ignores_P) \\ \Rightarrow \ & t \in prefixes(restr(\mathcal{B}_{P\|Q} \, , \, \mathcal{U}_Q \backslash \Omega_P)) \end{aligned} \tag{4.52}$$

where $\mathcal{U}_P$ is the universe symbol set of P and $\mathcal{U}_Q$ that of Q. This states that, in any behaviour of $P \parallel Q$ involving only actions ignored by P:

- Q does not require knowledge of any of the attributes offered by P, and
- the behaviour (offered attributes and decisions) of Q is as it would have been had P not been present.

In other words, the presence of P is undetectable in Q; so the only way that Q can be influenced by, and therefore aware of, the existence of P is because an action is presented that is not ignored by P. It might be thought that this is true of any P and Q; however, imagine that Ω_Q contains a derived attribute that counts the number of object identifiers in composition with it. Assuming that $|\iota(P)| > 0$ this count would be higher in $P \parallel Q$ than in Q alone, enabling the presence of P to be detected.

Now we define machines P and Q to be *mutually oblivious*, written $P \bowtie Q$, iff:

$$P \text{ is oblivious of } Q \text{ and } Q \text{ is oblivious of } P \tag{4.53}$$

If two machines are mutually oblivious then the presence of either can only be detected on first presentation of an action that is not ignored by both. This concept can be used to define an *object model*, a model of a population of instantiable objects, as a **model in which objects are mutually oblivious**. M is an *object model* iff for any two machines P and Q in M:

$$\iota(P) \cap \iota(Q) = \emptyset \Rightarrow P \bowtie Q \tag{4.54}$$

An object model is endowed with its full population of instances at initiation, but the presence of any given instance in such a model can only be detected when the object is presented with an action that it does not ignore. Up to that point it is "invisible". The first action that an object does not ignore, and thereby reveals its existence, can be thought of as the *creation action* of the object.

This approach to handling objects and their instantiation allows an object model to be treated as a fixed model, with a fixed population of machines. This means that any result that is true for any fixed model is also true for an object model. We will use this in Chapter 9 to establish results concerning the realizability of choreographies expressed as object models.

Chapter 5

Discussion

This chapter concludes the formal development of PM by locating it within the body of work on compositional modelling of processes, particularly where such modelling addresses the possibility of sharing data between composed components.

5.1 Positioning

We attempt to position PM with respect to other software description and modelling techniques. In particular we explore the way commonly used concepts and terms in software engineering relate to PM.

5.1.1 Style of Semantics

The denotational semantics of *completions* bears a close resemblance to the *Stable Failures* denotational model used in CSP, as described by Roscoe et al. [70]. A *failure* is a trace (of allowed actions) followed by a set that contains all the actions that could then be refused.[1] This enables distinction to be made between cases where, because of non-determinism, a simple traces model does not suffice. The classic example is shown in Figure 5.1, where two processes have the same traces but different behaviour. The failures model is shown in the middle, and of the completions model (using steps with just *action* and *decision*) to the right. Unlike traces, both of these distinguish the two processes.

[1] The issues of *stability* and *divergence*, which have to form part the CSP model, are not relevant in PM as there is no concept of a silent (internal) action.

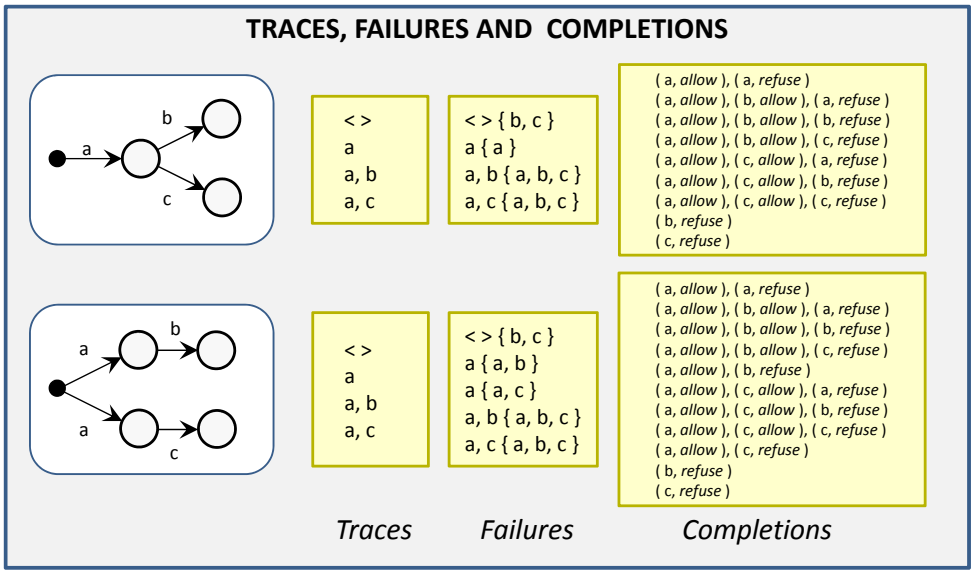


Figure 5.1: Traces, Failures and Completions

The reason that failures and completions succeed where simple traces do not is discussed by Nain and Vardi who articulate a *Principle of Comprehensive Modelling* [63], whereby behaviour is explicitly modelled for all relevant scenarios and not left to inference.

> *From this point of view, certain process-algebraic formalisms are underspecified, since they leave important behavioral aspects unspecified. For example, if the distinction between normal termination and deadlocked termination is relevant to the application, then this distinction ought to be modeled explicitly.*
> **– Sumit Nain and Moshe Vardi [63]**

Their claim is that adopting this principle allows trace semantics to be fully abstract.

This principle has been adopted in PM in two respects:

- The requirement (4.7b) that fixes are defined exhaustively for all elements the universe.
- The requirement (4.10c) that behaviour is defined exhaustively for all elements the universe.

5.1.2 Conditions and Constraints

We think it is important to differentiate between the concepts of a *condition* and a *constraint*. Here we are thinking of **conditions** as used in Hoare Logic [37] and incorporated into the *Design by Contract* method of software development by Meyer [56]; and **constraints** as used in PM for the rules that a protocol machine uses to decide whether to allow or refuse an action.

Conditions. Conditions, specifically *pre-* and *post-conditions*, are used to define a form of **contract** to specify the behaviour of a function. If the pre-condition of a function is true before invocation, then the function must ensure that the post-condition is true after invocation. If, on the other hand, the pre-condition is not true, the result of invocation is **unspecified**. Contracts are a mechanism for formal specification of a function, by placing conditions on what the function returns without constraining the choice of algorithm.

Constraints. Constraints, specifically *pre-* and *post-constraints*, are used to define **protocols** that specify the behaviour of processes that interact with each other. If software that exhibits a protocol is presented with an action, then it will refuse to engage (i.e., will not undergo any permanent change of state) if either a pre-constraint is false before the action and/or a post-constraint is false after the action. Protocols are a mechanism for specifying the behaviour of interactive software by defining the relationships between the states of the software and its ability to accept actions.

There is a clear semantic distinction between these, in particular:

- Violation of a pre-condition results in unspecified behaviour, which could include ungraceful failure (crash). By contrast violation of a constraint, causing an action to be refused, is an essential part of the specified behaviour of a protocol and there is no implication that it entails any kind of failure. Indeed, if violation of a constraint were to cause failure the software would be useless.
- Conditions are constructs used (as Hoare describes) to prove software correctness or (as Meyer advocates) to help ensure correctness in software development. Thus *assertion checking* (checking that conditions are observed) as supported for instance in the Eiffel programming, has use during development and testing but

once correctness has been established it serves no purpose and can be switched off. By contrast the logic that implements a protocol, by checking whether constraints are obeyed, is **essential** to the correct behaviour of the software and cannot be switched off.

This distinction is not generally well made. Consider the specification of the semantics of the UML concept of a *Protocol State Machine*, as described in the UML Superstructure Specification [64]. Given its name and the illustrations of its use in the UML documentation, you would expect it to have semantics similar to that of protocol machines. However the UML specification states:

> *A protocol transition (transition as specialized in the ProtocolStateMachines package) specifies a legal transition for an operation. Transitions of protocol state machines have the following information: a pre-condition (guard), on trigger, and a post-condition. Every protocol transition is associated to zero or one operation (referred BehavioralFeature) that belongs to the context classifier of the protocol state machine.*
> ...
> *The interpretation of the reception of an event in an unexpected situation (current state, state invariant, and pre-condition) is a* semantic variation point*: the event can be ignored, rejected, or deferred; an exception can be raised; or the application can stop on an error. It corresponds semantically to a pre-condition violation, for which no predefined behaviour is defined in UML*
> **– OMG: UML Superstructure [64]** ***pages 546, 547***

This uses the language of contracts (pre- and post-conditions) and seems to equate a pre-condition with a *guard*, which is surely wrong[2]. It then states that the semantics of violation are a "semantic variation point" (in other words, undefined in UML), which consequently means that the semantics of a UML Protocol State Machine are undefined. The underlying problem here is the incorrect choice of *contracts* (as opposed to *constraints*) as the basis for the semantics.

Later, in Chapter 8, we explore the concepts of contracts in software and propose a generalization of the concept that allows protocols to be used to specify contracts while maintaining the proper distinction between constraints and conditions.

[2]The proper distinction between pre-conditions and guards is made, for instance, by Miarka et al. [57]

5.1.3 Parallelism and Concurrency

Terms such as parallelism and concurrency are slippery, with various definitions. Peyton Jones [68] (talking in the context of functional programming, but in general terms) captures generally accepted definitions:

> *I make a sharp distinction between* parallelism *and* concurrency*:*
>
> - *A* parallel *functional program uses multiple processors to gain performance. For example, it may be faster to evaluate $e1 + e2$ by evaluating $e1$ and $e2$ in parallel, and then add the results. Parallelism has no semantic impact at all: the meaning of a program is unchanged whether it is executed sequentially or in parallel. Furthermore, the results are deterministic; there is no possibility that a parallel program will give one result in one run and a different result in a different run.*
> - *In contrast, a* concurrent *program has concurrency as part of its specification. The program must run concurrent threads, each of which can independently perform input/output. The program may be run on many processors, or on one — that is an implementation choice. The behaviour of the program is, necessarily and by design, non-deterministic. Hence, unlike parallelism, concurrency has a substantial semantic impact.*
>
> **– Simon Peyton Jones [68]**

In this, Peyton Jones positions parallelism and concurrency as being concerned with performance and distribution (to concurrent threads) respectively. Both of these are manifested in the way the software is **implemented**, as both require that computational activity happens on different threads; in the first case to evaluate $e1$ and $e2$ in parallel for performance reasons and in the second to allow concurrent parts of the program to perform input/output independently.

> *Process algebra is a widely accepted and much used technique in the specification and verification of parallel and distributed software systems.*
> **– Jos Baeten, Twan Basten, Michel Reniers [4]**

The above quote echoes the common positioning of parallel composition operators in process algebras as a formal way of investigating concurrency and parallelism in software systems. The fact that PM uses a composition operator which has superficial

similarity to the classical CSP || composition, albeit in a synchronous form, could lead a reader to assume that PM also uses composition to model concurrency and parallelism. However this would be wrong. In PM composition is purely about **description** and, in particular, the description of **ordering**. There is absolutely **no** implication in PM modelling that any kind of concurrent computing or parallel programming is involved. Instead, the compositional structure of a PM model aims to achieve:

- Quality of Abstraction.
- Support for Verification.

and for both of these it is important that the model author is free to determine and refine the choice and form of the machines used to express a model independently of issues of performance and distribution.

Quality of Abstraction. In the construction of a protocol model we aim, as in any modelling endeavour, to achieve qualities of simplicity and economy of expression. In Chapter 7 we will discuss how PM supports pursuit of these qualities in behavioural modelling. It is an occupational hazard of modelling that any modelling artefact that expresses complex ideas in a formal language suffers degradation in these qualities as it is built, because decisions on form and structure made early in the process are taken without a full appreciation of all aspects of the problem. The quality of the final result is dependent on work to recover simplicity and economy of expression by revisiting early decisions and refactoring the emerging solution.

Support for Verification. We will see in Chapter 9 how PM can be used in the context of formal verification of the designs of distributed multi-party collaborations. The verification relies on the ability to express a design as a composition, each component of which conforms to certain structural rules. When designing a choreography, in particular, it is crucial to be able to engineer the machines used in the composition so that each, separately, follows the rules that guarantee formal correctness; reorganizing (refactoring) the model where necessary to achieve this. An analogy is the formal definition of an even number as *one that can be expressed as a product of 2 factors, one of which is 2*. The description of 12 as 3*4 would need to be refactored as 2*6 to allow formal verification that it is even.

In both these cases, achieving the required result requires that the author of a model is free to select and evolve (refactor) the number and design of machines in a model according to the dictates of the problem. However, if the number and design of the machines were constrained by the need to represent parallelism (for performance) or concurrency (for distribution) in the senses of the Peyton Jones quote above, this would not be possible. Indeed, it would be incorrect to think that the motivation for expressing the bank account as three parallel machines in Figure 1.2 has anything to do with the performance or distribution of processing. This is a different focus from that normally cited as motivating composition in process algebras, at least as expressed by Baeten et al. at the start of this section.

5.1.4 Data and Topology

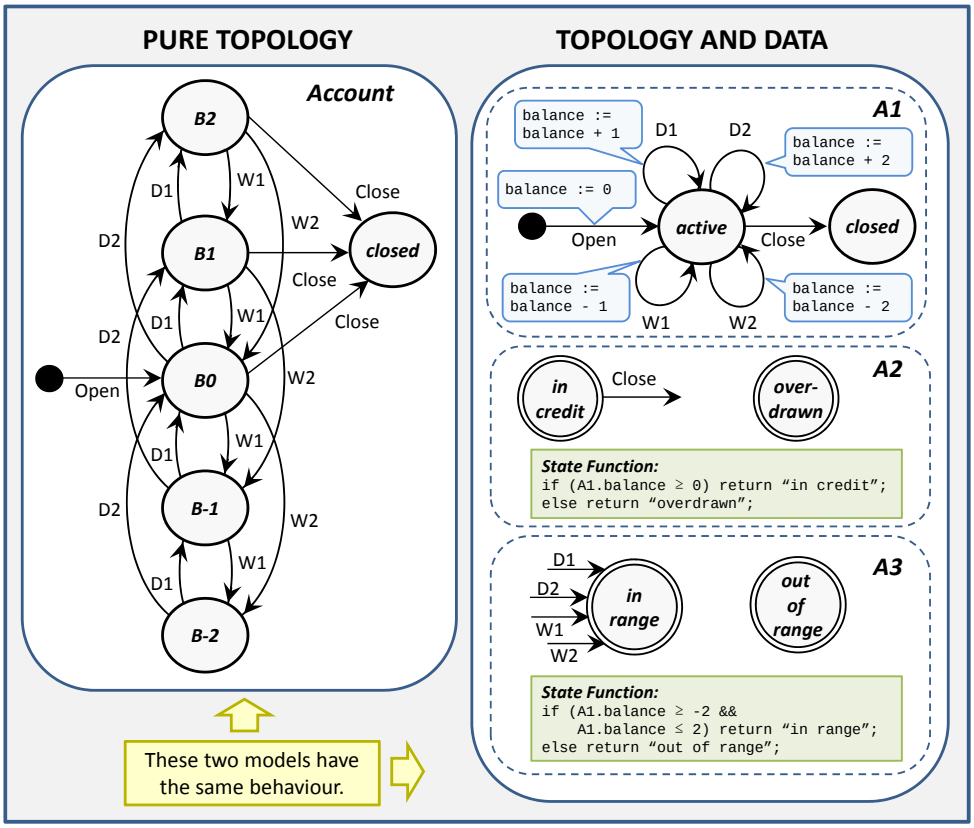


Figure 5.2: Two Models of a Bank Account

Another respect in which PM departs from some traditional thinking in process algebras is its treatment of data. In PM, data and topology are considered as dual:

two ways of expressing the same thing. Consider the two models of a very simple bank account shown in Figure 5.2. This account only supports an integer balance in the range -2 to +2, and only allows one or two units to be deposited, using *D1* or *D2*, or withdrawn, using *W1* or *W2*, at a time. The model on the left shows the account as a single machine; the model on the right shows the same bank account rendered as three composed machines. Note that:

- The account may only be closed when in credit. On the left this is modelled by the fact that *Close* starts from the states $\{B0, B1, B2\}$. The model on the right uses the machine $A2$ to enforce this.
- The model on the right uses post-state constraints in the machine $A3$ to enforce the range (-2 to +2) on the balance.

These two models are both valid PM specifications and, if the states of the model on the left are endowed with attributes to match those in the model on the right, equal in the sense described in Section 4.2.6.2. So one can be regarded as a *refactoring* of the other. The difference between them is that the machine on the left represents the balance *topologically*, whereas the one on the right handles the balance as *data*. It is always possible to refactor a PM model expressed as a composition of machines that use data and derived states as a single purely topological model, by:

- Defining a *state* for every distinct element of $restr(\mathbb{U}, \mathcal{U} \backslash \mathcal{A})$; and
- Defining an *action* (transition label) for every distinct element of $restr(\mathbb{U}, \mathcal{A})$

but in general, the resultant topological model will be huge. The topological model in Figure 5.2 has only been kept small by limiting the possible values of the balance. The price that is paid for this duality is the need for models to reach a stable and well-defined state after an action, otherwise there is no basis for determination of the fate of the next action, and this requires that the semantics of composition is synchronous.

Other process algebras have recognized that a treatment of data is needed in order to be able to make tractably small models of complex problems. However, data has generally been added as a separate semantic layer (as discussed in Chapter 5.2) and there is no idea of the kind of duality suggested here, or it is certainly not emphasized.

5.1.5 Determinism and Repeatability

> *The definition of insanity is doing the same thing over and over again and expecting a different result.*
> **– Albert Einstein (attributed)**

The focus of this thesis is on protocol machines that are deterministic, as specified by the condition (4.10e). Determinism is a complex notion and this section provides a short survey of this concept and its different definitions, to try and isolate our usage here more clearly.

5.1.5.1 Automata Theory and Concurrency

Perhaps the first point to make is that we do not use non-determinism as it is used in automata theory. Lamport makes the following distinction in [47]:

> *This use of non-determinism to model concurrency has caused some confusion, since the type of non-determinism involved is conceptually quite different from the non-determinism studied in automata theory and in the theory of non-deterministic algorithms.*
>
> *In automata theory, a non-deterministic machine is thought of as one that simultaneously pursues all possibilities. The machine is considered to complete its computation successfully if one of these possibilities succeeds. This has led to the study of non-deterministic algorithms, implemented by concurrently executing all possibilities and stopping the entire computation if one succeeds. The theory of branching time is appropriate for reasoning about this type of non-determinacy.*
> [...]
> *Our view of concurrent programs is that the non-determinism represents different possibilities, only one which actually occurs. This suggests that the linear time temporal logic should be more appropriate for reasoning about concurrent programs. Although "appropriateness" is not a provable property, we will give what we feel to be strong arguments that this is indeed the case.*
> **– Leslie Lamport [47]**

In thinking about the evolution of protocol machines we are likewise concerned with *different possibilities*, and a classification of types of non-determinism needs to consider where the decision about what happens next is located.

5.1.5.2 Types of Non-determinism

Figure 5.3 shows the range of different possibilities, expressed in PM notation. In all the cases shown we imagine that the machine responds to action a, after which one of b or c is offered. Considering each case shown:

Case 1. Q determines which of b and c can be accepted based on whether `i` is odd or even. This makes the behaviour of the machine completely determined. If `i` is even but c is presented rather than b, the machine will not advance. This example is completely compatible with (i.e., describable by) PM.

Case 2. In this case the machine is free to accept either b and c and so will advance to states B or C depending on which is presented. This is sometimes called *external non-determinism*, although arguably it is not really non-determinism unless the ordering of b and c is not determined in the environment.[3] Again, this is completely compatible with (describable by) PM.

Case 3. Here we imagine that the machine makes a random choice that determines whether it can receive b or c. As in Case 1, if the machine chooses b but c is presented it will not advance. This is sometimes called *internal non-determinism*, and is *true* non-determinism in that it is inherent and cannot be removed by refinement. The depiction on the right hand side in Figure 5.3 can be used to understand what happens if this process is rendered in PM. Because there are two transitions for a starting at $\bullet$ leading to different states $A1$ and $A2$, there would need to be steps $s1$ and $s2$ following initiation, with ${s1}^{\alpha} = {s2}^{\alpha} = a$ and both allowed, but with ${s1}^{\omega} \neq {s2}^{\omega}$ in violation of (4.10e). The fact that Case 3 cannot be modelled stems from the feature of completions in PM, that they are exhaustive in their description including the description of internal state.

[3] Roscoe [70] suggests that *external non-determinism* should be called *environmental choice* so as "not to confuse it with a form of non-determinism". This seems a sensible suggestion.

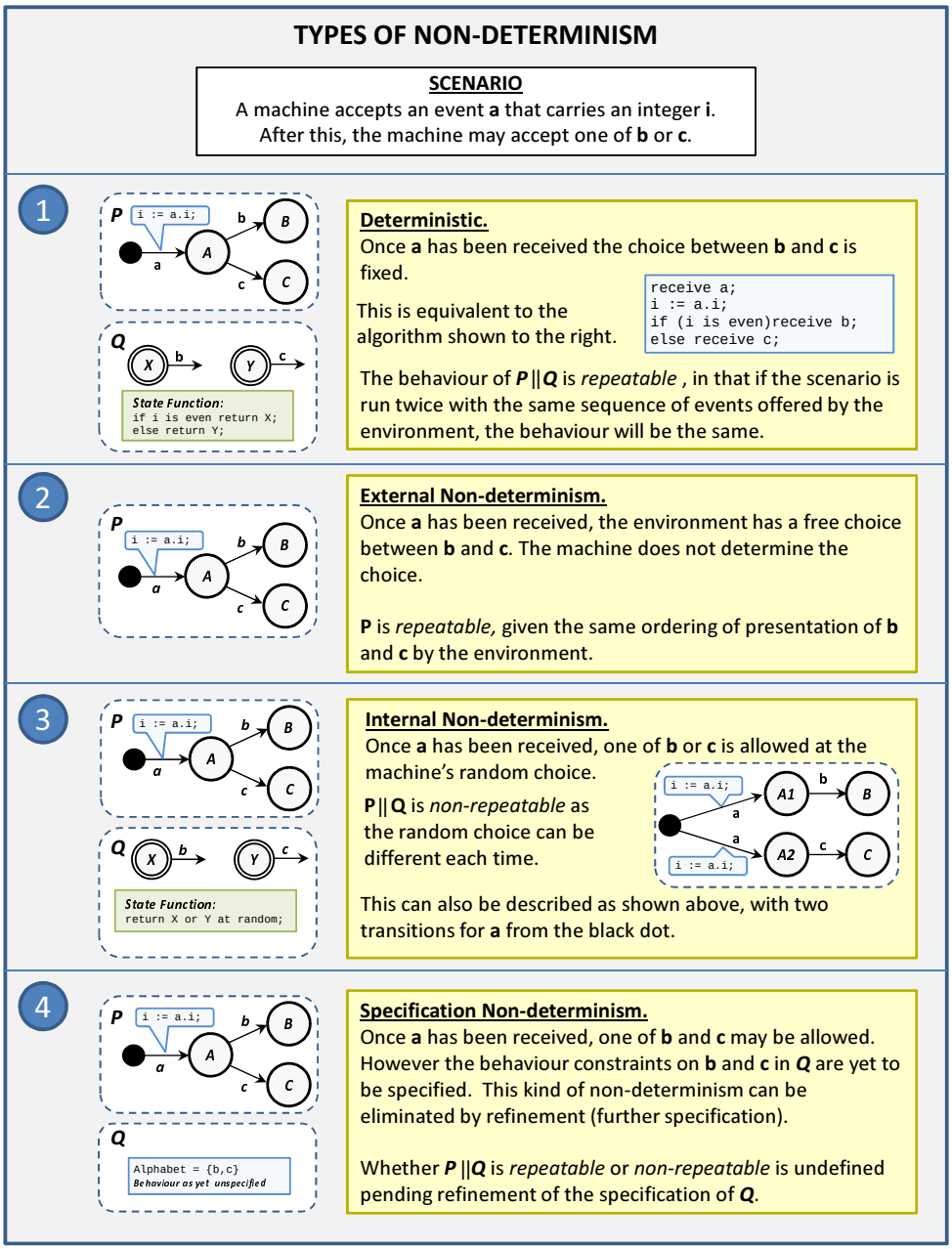


Figure 5.3: Types of Non-determinism

Case 4. Here we suppose that we have a model that is incomplete, in that we expect the behaviour to be refined by the specification of another machine, Q. PM is generally concerned with making partial (incompletely specified) descriptions of behaviour, so there is no incompatibility here.

So the assertion that *PM models are deterministic* relates specifically to Case 3. This can be demonstrated formally as it is not possible to model the machine Q in Case 3 as completions:

- Assuming that $P \parallel Q$ is independent, then the perceives of Q is just the value of `i`.
- The completions of Q must provide a **single** step for each combination of a value of `i` and action in $\{b, c\}$ giving whether the action is allowed or refused.

It is clearly not possible to construct completions for Q that accommodate a non-repeatable random response from the state function. There is no difficulty of this kind in the other cases.

5.1.5.3 Determinization

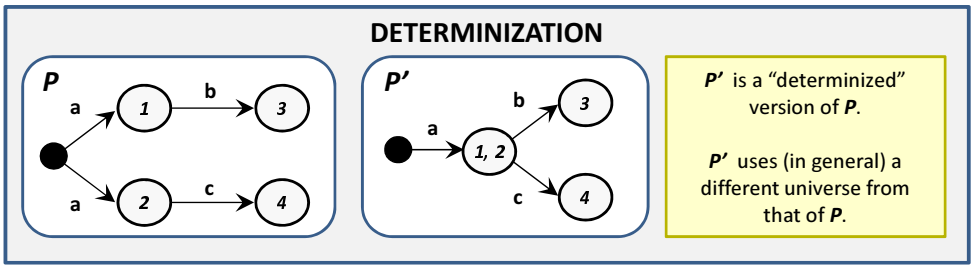


Figure 5.4: Determinization

There are well known transformations that converts a non-deterministic finite automaton into a deterministic one, by using a state space in the determinized version that is (some subset of) the powerset of the state space of the original. A small example is shown in Figure 5.4, where the state $1,2$ in P' represents the possibility that after a either state 1 or state 2 could pertain in P. Given this possibility, it is tempting to ask whether the stipulation that a protocol machine be deterministic per (4.10e) has meaning, as a machine that does not meet it can be transformed mechanically into one that does. This question relates to the definition of *state*, as follows.

Suppose that the machine P on the left in Figure 5.4 were represented as completions. The states marked 1 and 2 would be represented in Ω_P using observations, say: `(state=1)` and `(state=2)`. The attribute `state` will have some range of valid values

defined in the universe $\mathbb{V}$ over which P is defined, and in general we would **not** expect that `(state=1,2)` (as used in the determinized version, P') is in this universe; any more than we would expect the state (*active*, *closed*) (meaning either possibly pertains) to be a valid state in the universe over which the behaviour of a bank account is defined. As $\mathbb{V}$ can be viewed as the typing model used by P, this is equivalent to saying that determinization is prevented by the typing model over which a model is defined.

5.1.5.4 Interaction Non-determinism

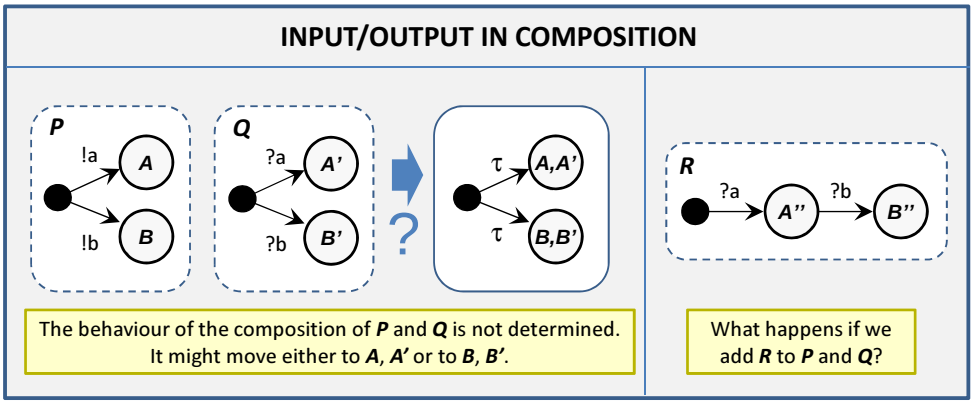


Figure 5.5: Input/Output in Composition

Heterogeneous composition, where one machine can send data that is received by another, leads to consideration of non-determinism resulting from a choice of different possible reactions. Specifically, consider a composition between the machines P and Q shown on the left hand side of Figure 5.5. Starting at $\bullet$ in both P and Q, a send/receive reaction could take place either between $!a$ and $?a$ leading to state (A, A') or between $!b$ and $?b$ leading to state (B, B'). The question here is, does this mean that the composition of P and Q must be viewed as an inherently non-deterministic machine, similar to that in Case 3 in Figure 5.3? The answer to this question is *no*, and this is understood in terms of the distinction between two semantic variants, as explored by Fidge [23] in his comparative analysis of process algebras:

CCS-like. Interaction is *concealed* and *bi-party*, as is the case in CCS and its derivatives such as pi-calculus.

CSP-like. Interaction is *visible* and *multi-party*, as is the case in CSP and LOTOS.

With CCS-like interaction, if two machines interact by sharing actions this shared action can never be seen by any other machine, and to the environment the transition is a spontaneous internal move from one state to another. With CSP-like interaction, each interaction is visible to the environment and can contribute in a wider multi-process composition. The distinction can be seen clearly by considering the behaviour that would result from composing R, shown on the right hand side of Figure 5.5, with P and Q:

- In the case of *CCS-like composition*, adding R introduces the further possibility of a reaction between $!a$ in P and $?a$ in R as the first step of the system. This means that the state of the composition of P, Q and R after the first step could be any of $\langle A, A', \bullet \rangle$, $\langle B, B', \bullet \rangle$ or $\langle A, \bullet, A'' \rangle$. The choice is undetermined and the system will select the new state "at random".

- In the case of *CSP-like composition*, adding R means that $?b$ is not possible as a first step for Q as R refuses $?b$. So the new state of the system after the first step is fully determined to be $\langle A, A', A'' \rangle$.

As this example illustrates, once non-determinism is created using CCS-like composition it is committed and cannot be eliminated by further composition. In fact, further composition may introduce further non-deterministic possibilities as R does here. This gives rise to *true* non-determinism of the kind illustrated in Case 3 in Figure 5.3. By contrast, composition using CSP-like semantics yields a description of possible orderings that is subject to further refinement, by the environment or other composed machines. Such refinement may, as is the case with the addition of R, eliminate the non-determinism completely.

One way of distinguishing the two kinds of composition is by considering whether composition is idempotent. Suppose that we have two processes, P and Q, which both engage in a single action $!a$. By any measure $P = Q$. CSP-like composition of P and Q yields another process that engages just in $!a$. In other words, the composition is idempotent. CCS-like composition yields a process that engages in two $!a$ actions in parallel and could therefore react with two other processes that both engage in $?a$. CCS-like composition is therefore not idempotent. It is almost as if, in the CCS-like case, processes "strongly own" their actions so that if a process is removed its actions go too; whereas, in the CSP-like situation, the actions contributed by each process add

to a shared pool and all processes co-operate in defining the ordering constraints on the actions in the pool.

5.1.5.5 A Possible Unification

As we described in Section 4.3.4, composition of machines where the output of one forms input to the other (heterogeneous composition) requires that we define a combined universe (4.31). Suppose that machines P and Q are defined over universes $\mathfrak{U}$ and $\mathfrak{U}'$ respectively; and that $\mathcal{A} = \mathcal{D} = \mathcal{A}' = \{y\}$, so that P has y as output and Q has y as input. Suppose also that y allows only the values `"foo"` and `"bar"`; and that in some state $P \parallel Q$ allows steps for both these values. Even though the output of y by P is bound by the input of y by Q, y is still an action symbol of the combined universe $\mathfrak{U} \boxplus \mathfrak{U}'$, and so $P \parallel Q$ still has separate steps for `(y = "foo")` and `(y = "bar")`. Only if these steps are identical apart from the value of y, so that this symbol can be removed under the rules of normal form, does the machine becomes autonomous (in the sense of definition (4.19)). This is quite different from the classical CCS composition, where a possible reaction on complementary names causes the reaction to become silent (a "τ-transition"), with the possible introduction of non-determinism as illustrated in Figure 5.5.

Suppose that we stipulate the following further well-formedness rule on a combined universe $\mathfrak{U} \boxplus \mathfrak{U}'$:

$$\mathcal{U} \cap \mathcal{U}' \subseteq \mathcal{D} \cup \mathcal{D}'$$

which requires that all shared symbols of the combined universe are output (derived) in one universe or the other. In this case, CSP-like composition is composition *within a universe* and CCS-like composition is composition *across universes*. By recognizing the need to base machine definitions over a data universe the two styles could be accommodated within a single semantic framework.

5.1.6 Relationship to UML

The Universal Modelling Language (UML) is viewed as the de-facto standard for software modelling and, as such, it is appropriate that we make a comparison between it and PM. Perhaps the starting point is the observation made in Section 1.1.1 that there are significant differences between UML and PM:

- The declared and normal usage of UML is to support the design of software that is to be built using mainstream OO programming languages (C++, Java, etc.). PM does not have this aim and because it is based on a form composition that is not supported in current programming languages, it is arguably not adaptable to this purpose.
- The generalisation/specialisation paradigm of UML, based on refinement or subclassing, is quite different from that of PM, which is compositional as described later in Section 7.3.
- PM has a formal semantics which UML does not (at least as part of the published standard).

Of the UML notations, state machines come the closest to PM; and it is worth noting that the UML state machine notation, which are based on Harel's *statechart* notation [34], offers a construct called *orthogonal regions* that allows a single state machine to own conceptually independent regions which operate in parallel, each hosting its own population of states and transitions. By providing a mechanism for representing parallel activity within a single machine, orthogonal regions achieve a similar effect to composition. However other differences mean that the similarities are outweighed and probably amount to an irreconcilable difference of semantics between the two:

- Orthogonal regions do not have any concept of CSP style synchronisation on shared events and so inter-region message passing is used to achieve synchronisation.
- *Conditions* rather than *constraints* (see Section 5.1.2) are used for the interpretation of unavailable transitions in statecharts.
- There is no concept of derived states in statecharts and guards are used to control the firing of available transitions.

5.2 Related Work

This chapter surveys other work that gives a formal treatment of the interaction between behaviour and data, and makes comparisons with the analogous features of PM's.

5.2.1 Shared Data in Process Algebra

This section explores how the mainstream process algebras (CSP, CCS and variants, LOTOS, and ACP) address the interaction of data and behaviour. These algebras assume asynchronous co-operation and, as this survey shows, the degree to which it is possible to achieve coherent data-based behavioural rules is limited unless extra machinery is introduced.

> *CSP and CCS were designed to model asynchronous processes, running at indeterminate relative speed; the form of parallel composition typical of CCS and CSP allows independent progress to be made by individual processes.*
> **– Stephen Brookes [17]**

All of these algebras, in their seminal form, aimed to enable reasoning about systems of interacting processes running in parallel. Processes were conceived as abstract constructs that advance by engaging in *actions*, possibly shared between two or more processes, represented as symbols. In all of them the emphasis was on representing *behaviour*, so the representation of *data* was minimal or absent. However all them have spawned variants with means of representing data, to extend their application to modelling computation that relies on the interaction between data and behaviour. We examine below some of the main examples of this below, and consider their respective abilities to replicate the system described in Figure 2.1 on page 27 and ensuring that z cannot occur. Of course, preventing z is trivially achieved by refactoring the processes, in particular by combining P and Q into a single process that offers a choice between x or y, so we stipulate in this challenge that P and Q remain nominally independent of each other.

5.2.1.1 CSP Conditional Operators.

CSP [38] includes a mechanism to assign values to variables and a *conditional operator* whereby the assigned values of variables can influence behaviour.

$$(P \lhd \texttt{cond} \rhd Q) \tag{5.1}$$

This means: P is chosen if `cond` is true, otherwise Q chosen. `cond` is called a *conditional variable*.

Hoare says of condition variables, such as `cond`:

> *To deal effectively with assignment in concurrent processes, it is necessary to impose a restriction that no variable assigned in one concurrent process can ever be used in another.*
> **– C.A.R. Hoare [38]**

This is because there is no guarantee that a shared data variable inspected by one process is not being simultaneously updated by another, yielding an undefined result. In particular it would not be possible to represent the example described in Figure 2.1, because R cannot use a conditional variable that gives the state of P (or Q) to determine its behaviour, as it would have to be assigned in P (or Q).

5.2.1.2 The pi-calculus with data.

The pi-calculus, a direct descendent of CCS, has provided fertile ground for the cultivation of various specialised process calculi with the expressive power to model data, including *The polyadic pi-calculus* [60] (an extension that allows tuples of names to be exchanged), *The spi-calculus* [1] (an extension designed for the description and analysis of cryptographic protocols), *The pi-F calculus* [80] (a calculus that supports *fusions*, being explicit identification of names), and *CC-Pi* [18] (an extension supporting concurrent constraints). In their work [9], Bengtson et al. describe the *psi-calculus*, an extension of the pi-calculus with nominal data types for data structures and for logical assertions representing facts about data that is capable of encoding various data-enriched variants of the pi-calculus including those listed above, and we focus on this for comparison with PM.

To look to how pi-calculus might handle the constraints the example in Figure 2.1 we would need to use a constraint calculus, for instance with the following constraints corresponding to R for the "asynchronous" behaviour variant:

a. Either z has not happened or both x and y have happened (added at initiation of the system)

b. x and y have both happened (added when x and y happen together)

c. x has happened but y has not happened (added when x happens by itself)

d. y has happened but x has not happened (added when y happens by itself)

e. z has happened (added when z happens)

where constraint (b) allows x and y to be simultaneous. For "synchronous" behaviour variant, the following constraint set could be used:

a. Either z has not happened or both x and y have happened (added at initiation of the system)

b. x and y have not both happened (added at initiation of the system)

c. x has happened (added when x happens)

d. y has happened (added when y happens)

e. z has happened (added when z happens)

where constraint (b) prevents x and y being simultaneous. It is immediately obvious from constraints (a) and (b) that z cannot occur in the synchronous variant.

> *Most prominently, in the semantics of CC-Pi the fusions resulting from communication are required to be consistent with the store (as defined by the constraint system). In contrast our [psi-calculus] semantics will allow transitions that lead to an inconsistent store.*
> **– Jesper Bengtson, Magnus Johansson, Joachim Parrow and Björn Victor [9]**
>
> *Unless these producers are coordinated they may produce conflicting constraints, causing an inconsistency. For example, one agent may impose the constraint $x = 3$ and another $x = 4$, resulting in uncontrolled behavior since an inconsistent store entails any ask condition.*
> **– Vijay Saraswat [71]**

As Bengtson et al. point out in their paper (see quote above), the psi-calculus does not ensure that the constraints in a system are consistent; and according to Saraswat this precludes the use of constraints to control behaviour. To guarantee that the constraint store is consistent requires that operations that change the store are serialised, but the operational semantics of psi-calculus, with its asynchronous co-operation model adopted from pi-calculus, has no means to do this.

5.2.1.3 LOTOS with Global Variables.

LOTOS (*Language of Temporal Ordering Specification*) is a process algebra based on early versions of CCS and CSP [23]. In 1995, Khoumsi and Bochmann introduced support for global variables into LOTOS [44] by allowing an action σ, to be encased in a *transaction*:

$$<\theta, \sigma, \phi>$$

The action is sandwiched between an *enabling condition*, θ, and an *update function*, ϕ. An action is only enabled if the enabling condition is true (so this is the same as a guard); and execution of the action results also in execution of the update function that gives guards new values.

The authors note that certain *exclusion rules* need to be observed to prevent concurrent transactions interfering with each other:

- Variables on which a guard, θ, depends are not written by another concurrent transaction; and
- Variables written by an update, ϕ, are neither written nor read by another concurrent transaction.

To ensure that this is the case, a locking mechanism is introduced whereby a transaction locks the variables it reads or writes before it actually accesses them; and does not obtain a new lock after it has released a lock. This scheme has the danger of deadlocking and to avoid this, *time-stamps* are used to give lock requests a global priority, so the transaction whose lock has lower priority aborts in the event of a deadlock conflict.

To encode the example in Figure 2.1 would require a *guard* on x in R to prevent it happening if y has already happened. Because this guard is based on the state of P it would be altered by the update function required after x happens in P. As the transactions for x in P and R are concurrent, this violates the condition that transactions do not interfere with each other.

5.2.1.4 ACP Guarded Commands.

ACP [4] includes a concept called a *guarded command*, of the form $\Phi :\rightarrow P$, with the intuitive meaning '*if* Φ *then* P'. This is similar to CSP, so the conditional expression

(5.1) can be expressed in ACP as:

$$(\texttt{cond} :\rightarrow P \ + \ \neg\texttt{cond} :\rightarrow Q)$$

In ACP there is no stipulation, as there is in CSP, that a propositional variable such as Φ is local to a process: it can be given a value in one and used in another. However, observation in one process of a variable set by another process must yield every possible value allowed by any interleaving of the processes. In the context of the example in Figure 2.1, every time there is the possibility of some set of variables being observed in R to have values representing the fact that one of P or Q has advanced, there is also at the same time the possibility of an observation showing that both have advanced so that z is allowed.

5.2.1.5 Summary.

None of these approaches can model the example in Figure 2.1 as a simple composition $S = P \parallel Q \parallel R$ while ensuring that z cannot occur. The point here is that all approaches have **asynchronous co-operation semantics**, and cannot realize or simulate the synchronous behaviour of a PM composition, in which z is prevented, without altering the model or introducing extra machinery to coerce synchronous behaviour. Synchronous behaviour is needed if z is to be blocked because, as Basten et al. [4] put it:

> *The independent execution of parallel processes makes it difficult or impossible to determine the values of global variables at any given moment. It turns out to be simpler to let each process have its own local variables, and to denote exchange of information explicitly via message passing.*
> **– Jos Baeten, Twan Basten, Michel Reniers [4]**

In this quote "independent execution of parallel processes" must be taken to refer to asynchronous co-operation. This conclusion is also reflected by Groote and Ponse [31] in their result that the combination of guards and parallelism (composition) leads to an axiom system that is unsound in any reasonable bi-simulation semantics.[4]

[4] Although they did offer a way round this problem in the form of a *two-phase* calculus in which process terms are rendered free of parallelism in order to prove identity.

5.2.2 Synchronous Process Algebra

Some authors in process algebra have explored the possibility of *synchronous* composition semantics. Three examples are:

- Milner with SCCS [59, 36], a synchronous variant of the (asynchronous) CCS.
- Bergstra and Klop with ASP [10], a synchronous variant of the (asynchronous) ACP.
- Barnes with SCSP [5], a synchronous version of the (asynchronous) CSP.

These accounts require, as does PM, that when two processes are synchronously composed both engage in **all actions**; so there is no concept that one machine may engage in an action and advance to a new state while the other either sleeps or, perhaps, engages in some different concurrent action of its own. The first two use CCS-like composition (see Section 5.1.5.4) and the third uses the CSP-like composition used by PM.

All these synchronous algebras describe action behaviour without any treatment of data, so none provide any treatment of the interaction between data and behaviour comparable to that in PM.

5.2.3 Coordination Schemes

The foregoing sections assume a clean distinction between synchronous and asynchronous co-operation, but some schemes for describing or implementing composed processes use asynchronous co-operation as their default model but supply additional machinery for *coordination* of the execution. The most general such schemes are termed *coordination schemes*, of which the Linda language [29], based on shared tuple spaces is the archetypical example.

> *We will define an* asynchronous ensemble *as a collection of asynchronous activities that communicate. An activity is a program, process, thread or any agent capable in principle of simulating a Turing Machine. It could be a person; it could be (recursively) another whole ensemble.*
> **– David Gelertner and Nicholas Carriero [29]**

Coordination languages aim to address the coordination (communication and synchronisation) needs of the whole spectrum of asynchronous ensembles, from multi-threaded applications executing on a single processor, through tightly-coupled, fine-grained parallel processing applications, to loosely-coupled, coarse-grained distributed applications. Such schemes work by requiring that certain events in the execution of the system require access to a system resource that is subject to *mutual exclusion*, so that two processes cannot access the resource simultaneously. This enables atomicity and serialisation of the events that affect shared data[5].

Coordination schemes can be viewed as hybrid: allowing cooperation between asynchronous computations without incurring the penalty of single-threading all actions in the system, as a purely synchronous implementation would require. The penalty that they pay is the possibility of deadlock (see, for instance, the discussion of transaction processing by Jagannathan and Vitek [42]). This characterisation is instructive, as it positions such schemes as concerned with programmatic enforcement of the ordering of events in concurrent computing or parallel programming environments. However, as explained in Section 5.1.3, PM is not concerned with these issues but only with the description of ordering. In a sense the pure synchronous co-operation semantics used by PM can be viewed as "completely coordinated", not requiring any coordination scheme of the type represented by Linda and its kind.

5.2.4 Abstract State Machines

In this section we expand on the similarities between the work described here and that of *Abstract State Machines* and the related area of *Synchronous Reactive Languages*. A connection with Abstract States Machines has already been noted in Section 4.2.8.

5.2.4.1 Parallel Abstract State Machines

Blass and Gurevitch have described a compositional approach Abstract State Machines for parallel algorithms [13]. The model they describe uses a synchronous paradigm, advancing in steps from one coherent global state to the next, with parallel execution of *proclets* performing the state update. They explicitly rule out the asynchronous model:

[5] LOTOS with Global Variables, with its locking mechanism on shared variables; and concurrent constraint calculi that support constraint store consistency, can be viewed as coordination systems.

> *The algorithms we consider have computations divided logically into a sequence of discrete steps. Within each step, many parallel sub-computations may take place, but they finish before the next step begins. We do not consider the more general notion of distributed computation, where many agents proceed asynchronously, each at its own speed, and where communications between agents may provide the only logical ordering between their actions.*
> **– Andreas Blass and Yuri Gurevitch [13]**

Rather than being about interactive processes, ASMs concern *algorithms*, and there is no concept of *actions* or *protocols*. The composition of the proclets is based on dataflow rather than process algebraic composition. Having said that, some of the fundamental issues in establishing formalization of a synchronous parallel model are the same. In particular, their formalization requires that the proclets of an algorithm obey an *Acyclicity Postulate* whereby the *information flow digraph* that depicts how data flows between the proclets does not contain cycles. This is equivalent to our requirement, given in Section 4.3.2, that a protocol model does not contain circular dependencies.

5.2.4.2 Action Machines

Grieskamp et al. working at Microsoft Research have proposed the use of composition of Abstract State Machines in the context of symbolic execution for program verification [30]. Their concept of composed *Action Machines* bears a close similarity to the concept of composed protocol machines, albeit in a completely different context. The following quote clearly shows that the composition they use for Action Machines has CSP semantics:

> *The parallel composition of two action machines results in a machine that transitions both machines simultaneously on a set of synchronized actions and interleaves transitions on other actions.*
> **– Wolfgang Grieskamp, Nicolas Kicillof and Nikolai Tillmann [30]**

5.2.5 Synchronous Reactive Languages

Our approach also has something in common with *synchronous reactive languages* such as Esterel and Lustre. As described by Berry [12], synchronous reactive languages

can be thought as executing in discrete steps, sometimes called "ticks", and in the introduction to his book [33] Halbwachs relates the concepts in synchronous reactive programming to the work by Milner on synchronous process algebras. A major benefit claimed for the synchronous reactive model is that it is *deterministic*:

> *Esterel is based on the perfect synchrony hypothesis: control transmission, signal broadcasting, and elementary computations are supposed to take no time, making the outputs of a system perfectly synchronous with its inputs. Perfect synchrony has the major advantage of making concurrency and determinism live together in harmony.*
> **– Gérard Berry [12]**

Authors in the area of synchronous reactive languages tend to distinguish *reactive* from the *transformational* and *interactive* paradigms of computing, and claim the first of the three as the domain of applicability of synchronous reactive languages:

> ***Reactive** systems are computer systems that continuously react to their environment at a speed determined by their environment. This class of systems has been introduced in order to distinguish these systems, on the one hand, from **transformational** systems, whose inputs are available at the beginning of the execution and which deliver their outputs when terminating – and, on the other hand, from **interactive** systems, which continuously interact with their environment, but at their own rate (e.g. operating systems).*
> **– Nicholas Halbwachs [33]**

This classification seems dubious, as PM would seem to show that the synchronous approach can successfully be applied to *interactive* systems.

5.3 Summary

In this chapter we have positioned PM with respect to other paradigms and techniques for the formal modelling of behaviour. We emphasize the idea that PM is concerned with behaviour defined by the ordering of actions, and in particular how data and behaviour interact.

The synchronous nature of the composition in PM means that, in principle, every machine in a model engages in every action, with some undergoing a change of state and others "ignoring" the action. Between actions a model has a well-defined state, allowing well-defined inter-machine data access. The resultant behaviour is deterministic, because it is not subject to race conditions between inter-process data access and intra-process state change. In consequence, PM has the following properties:

- Composition is closed, in the sense that the composition of two protocol machine is another protocol machine. Deterministic behaviour and type safety are preserved by composition.
- The behavioural equality of two machines is established independently of how they are expressed as compositions and this means that a composition can be "refactored" without changing its meaning. This can be exploited to engineer specific formal properties, as it is in the approach to choreography described in Chapter 9.
- A finite-state protocol machine can be expressed as a single LTS (without composition), so that its meaning is rendered in pure topology. However, as this requires a state-space with a node for every combination of data value that can occur it is not normally practical; so PM models use data attributes to abstract the state-space and inter-machine access to describe behaviour constraints based on these abstractions. In this sense, data and topology are dual in PM as one can be represented using the other without change of meaning.

This is in contrast to other process algebras where composition is intended to model independent agents or threads of processing, so composed processes are asynchronous and there is no concept that topology and data are somehow equivalent or interchangeable.

In Part III of the thesis we describe applications of PM. These applications address both "object" and "process" modelling and this supports our contention, stated in Chapter 1, that PM successfully spans the two paradigms.

Chapter 6

Topological Transformations

in discussing applications of PM in Part III we will be concerned to reason about behaviour defined as PM models. Generally, the basis for reasoning is **topology** and the results are obtained using arguments based on the graphical representations. To prepare for this, this chapter provides a repertoire of topological transformations that are useful in developing and analyzing the behaviour of protocol machines expressed as labelled transition systems.

6.1 Topological Representation

For the subset of machines that are termed stored-state (see Section 4.4.4) it is possible to use a labelled transition system (LTS) to represent an abstraction of the machine definition. This abstraction captures the actions (s^{α}) and decisions (s^{δ}) parts of each step and the relationship between the two. This is possible for stored-state machines as the machine's decisions are fully dependent on the actions. This form of representation is not defined for derived-state machines as their behaviour cannot be depicted in topological form.

6.1.1 Topology Formalism

We define a *representation* of a stored-state machine as the tuple:

$$P = \langle \Lambda_P, \Sigma_P, \Gamma_P, \Delta_P \rangle$$

where:

Λ_P is the alphabet of P, the set of actions which P does not ignore, as defined in Section 4.5.3. Elements range over a, b, x, y, ...

Σ_P is a finite set of states of P, as defined in Section 4.2.7.1. Elements range over σ_i, σ_j, ...

$\Gamma_P \subseteq (\Lambda_P \times \Sigma_P)$ is a binary relation. $\langle x, \sigma_i \rangle \in \Gamma_P$ means that x is allowed by P when P is in state σ_i. $\langle x, \sigma_j \rangle \notin \Gamma_P$ means that x is not allowed by P when P is in state σ_j.

Δ_P is a total mapping $\Gamma_P \rightarrow power(\Sigma_P) \backslash \{\emptyset\}$ that defines for each member of Γ_P the next state or states that P adopts as a result of allowing an action. So $\Delta_P(\langle x, \sigma_k \rangle) = \{\sigma_l, \sigma_m\}$ means that if P *allows* x when in state σ_k it will then adopt either state σ_l or state σ_m.

If Δ_P maps each member of Γ_P to a single state, then P is *deterministic*. Note that in general we do not assume determinism, although if used to represent a protocol machine an LTS must be deterministic.

We also take it that an LTS for P has a single state $\bullet_P \in \Sigma_P$ that is identified as the *initial state*. This is the state that the machine is in after its initiation step (see Section 4.2.4). We do not allow the initial state to have incoming transitions, so a machine cannot return to it. (Any LTS that has returns to its initial state can be transformed into one that does not by replicating the initial state and its transitions and removing the incoming transitions from one copy).

A state σ that has no outgoing transitions, so $\{x \mid \langle x, \sigma \rangle \in \Gamma_P\} = \emptyset$, is called a *terminal state*. An LTS is not required to have a terminal state and can have more than one.

6.1.2 Transitions, Paths and Traces

It will be useful to identify the transitions in P uniquely. The labels given by Λ_P do not do this as it is quite possible for two transitions in the graph of an LTS to carry the same action label.[1]

[1] Even if the LTS is deterministic, as the transitions carrying the same label can have different starting states.

Transition Identifiers. We give each transition of P a *unique identifier*, using elements of the tuple $\Theta_P \subseteq \Sigma_P \times \Lambda_P \times \Sigma_P$ defined as:

$$\Theta_P = \{\langle \sigma, x, \sigma' \rangle \mid (\langle x, \sigma \rangle \in \Gamma_P) \wedge (\sigma' \in \Delta_P(\langle x, \sigma \rangle))\}$$

and define functions:

- *label* to extract the action label from a transition, and
- *labels* to extract a sequence of action label from a path.

Formal definitions of these are given in Appendix A.6.

Paths. A *path* is a sequence of elements from Θ_P that represent a connected topological route following the direction of the arrows on the transitions. For a representation P we define:

- $\Pi_P(\sigma)$ as the set of all paths starting at state σ.
- $\overrightarrow{\Pi}_P(\sigma)$ as the set of all finite paths starting at state σ but unable to progress beyond a return to a state already visited.
- $\overrightarrow{\Pi}^m_P(\sigma)$ as the set of maximal finite paths starting at state σ but unable to progress beyond a return to a state already visited, so end by reaching a terminal state or a state already visited).

Formal definitions of these are given in Appendix A.6.

Traces. The set $traces_P = labels(\Pi_P(\bullet))$ is the set of traces of P.

Fn: *paths*. To obtain the set of paths that correspond to a given trace we define the function *paths* with signature:

$$paths :: traces_P \to power(\Pi_P(\bullet))$$

as:

$$paths(\tau) = \{\pi \mid labels(\pi) = \tau\}$$

If P is deterministic, so that:

$$\forall\ \tau \in traces_P : \pi 1, \pi 2 \in paths(\tau) \Rightarrow \pi 1 = \pi 2$$

then we use $path(\tau)$ to return the single path for that trace.

6.2 Topology Weaving

Topology weaving constructs the topological representation of the parallel composition of two topological (stored-state) machines. The technique can be used repeatedly to combine multiple machines. This is a well-known technique described elsewhere, for instance Sassone et al. [72].

6.2.1 Weaving Representations

Suppose we have two stored-state machines, P and Q. Using the semantics of CSP $\|$ composition we can create a representation of $P \parallel Q$ as follows:

$$\Lambda_{P\|Q} = \Lambda_P \cup \Lambda_Q$$

$$\Sigma_{P\|Q} = \Sigma_P \times \Sigma_Q \quad \text{(the Cartesian product of } \Sigma_P \text{ and } \Sigma_Q)$$

Given $x \in \Lambda_{P\|Q}$ and a state $\langle \sigma_p, \sigma_q \rangle \in \Sigma_{P\|Q}$, we determine whether $\langle x, \langle \sigma_p, \sigma_q \rangle\rangle \in \Gamma_{P\|Q}$ as follows:

$$\begin{array}{ll}
\langle x, \langle \sigma_p, \sigma_q \rangle\rangle \notin \Gamma_{P\|Q} & \text{if } (x \in \Lambda_P \wedge \langle x, \sigma_p \rangle \notin \Gamma_P) \\
\langle x, \langle \sigma_p, \sigma_q \rangle\rangle \notin \Gamma_{P\|Q} & \text{if } (x \in \Lambda_Q \wedge \langle x, \sigma_q \rangle \notin \Gamma_Q) \\
\langle x, \langle \sigma_p, \sigma_q \rangle\rangle \in \Gamma_{P\|Q} & \text{otherwise}
\end{array}$$

and we construct $\Delta_{P\|Q}(\langle x, \langle \sigma_p, \sigma_q \rangle\rangle)$ as follows:

$$\begin{array}{lll}
\Delta_{P\|Q}(\langle x, \langle \sigma_p, \sigma_q \rangle\rangle) = \{\langle \sigma'_p, \sigma_q \rangle \mid \sigma'_p \in \Delta_P(\langle x, \sigma_p \rangle)\} & & \text{if } (x \in \Lambda_P \wedge x \notin \Lambda_Q) \\
\Delta_{P\|Q}(\langle x, \langle \sigma_p, \sigma_q \rangle\rangle) = \{\langle \sigma_p, \sigma'_q \rangle \mid \sigma'_q \in \Delta_Q(\langle x, \sigma_q \rangle)\} & & \text{if } (x \notin \Lambda_P \wedge x \in \Lambda_Q) \\
\Delta_{P\|Q}(\langle x, \langle \sigma_p, \sigma_q \rangle\rangle) = \{\langle \sigma'_p, \sigma'_q \rangle \mid \sigma'_p \in \Delta_P(\langle x, \sigma_p \rangle) \wedge & & \\
\qquad \sigma'_q \in \Delta_Q(\langle x, \sigma_q \rangle)\} & & \text{otherwise}
\end{array}$$

The initial state of the woven representation is $\bullet_{P\|Q} = \langle \bullet_P, \bullet_Q \rangle$.

6.2.2 Reasoning with Woven Representations

Weaving representations enables topological reasoning about models expressed as a composition. Behaviour of the model as a whole can be deduced based on the single representation obtained by weaving the composites. This technique is used in Chapter 9 to reason about choreography realizability.

6.3 Unwound Form

This chapter describes a topological transformation used in the context of reasoning about choreography realizability. The transformation achieves a single unwinding of all the cycles in an LTS.

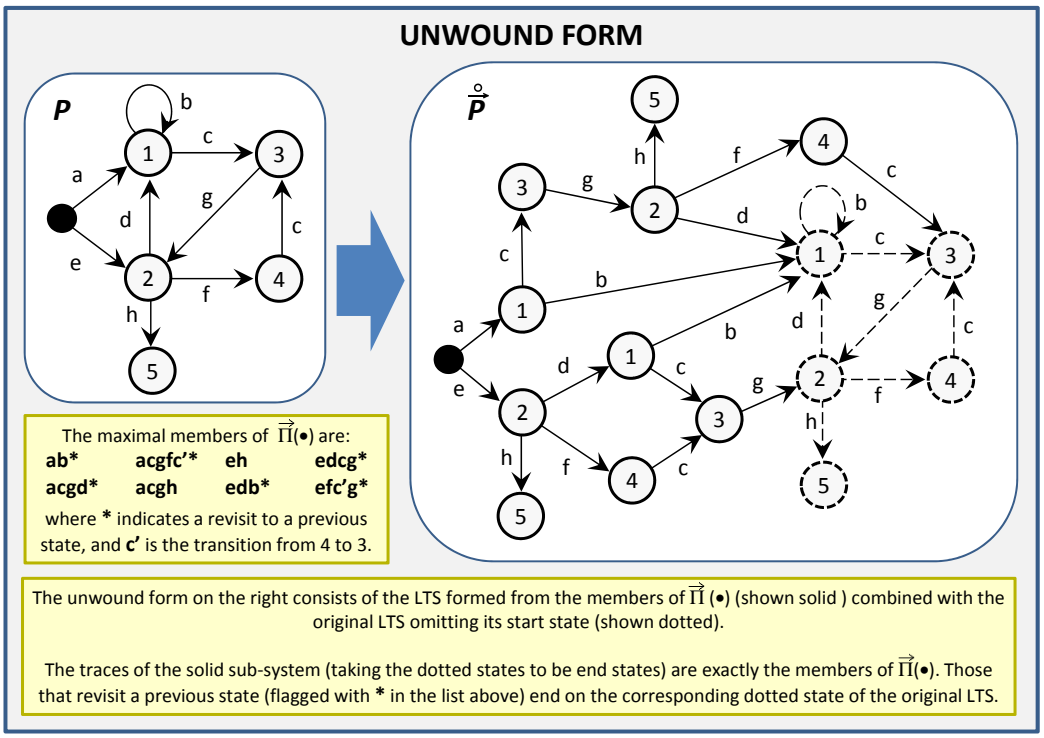


Figure 6.1: Unwound Form

6.3.1 Definition of Unwound Form

Given an LTS P with topological representation $\langle \Lambda_P, \Sigma_P, \Gamma_P, \Delta_P \rangle$ we proceed to create the unwound form $\overrightarrow{\overset{\circ}{P}}$ as follows:

a. Consider the members of $\overrightarrow{\Pi}^m_P(\bullet)$ as the full set of complete traces of an LTS $\overrightarrow{P}$ with $\Lambda_{\overrightarrow{P}} = \Phi_P$. The traces of $\overrightarrow{P}$ never have a repeated element, so $\overrightarrow{P}$ is acyclic. We form $\overrightarrow{P}$ as follows:

   - Use the equivalence relation technique described in Section 4.2.7.1 on $\overrightarrow{\Pi}^m_P(\bullet)$ to define the members of $\Sigma_{\overrightarrow{P}}$ apart from the terminal states.
   - For each $\pi \in \overrightarrow{\Pi}^m_P(\bullet)$ add a fresh member to $\Sigma_{\overrightarrow{P}}$ as a terminal state for π.

b. Label each terminal state in $\Sigma_{\overrightarrow{P}}$ with $end(\pi)$, the end state of the path $\pi \in \overrightarrow{\Pi}^m_P(\bullet)$ for which it was added.

c. Form the LTS $\mathring{P}$ as a copy of P without its start state, so $\Phi_{\mathring{P}} = \Phi_P \backslash \{\theta \mid \theta \in \Phi_P \wedge begin(\theta) = \bullet_P\}$.

d. Form the unwound form $\overrightarrow{\mathring{P}}$ by joining $\overrightarrow{P}$ and $\mathring{P}$. The joining is done by identifying each terminal state of $\overrightarrow{P}$ with a state of $\mathring{P}$ using the labels attached in step b.

e. Convert the transition labels in the unwound form, so far labelled using Φ_P, back to labels from Λ_P by converting θ to $label(\theta)$.

Figure 6.1 shows an example of the unwound form of an LTS. Note first that, by construction, the unwound form $\overrightarrow{\mathring{P}}$ has the same traces as P.

Note also that every state of $\overrightarrow{\mathring{P}}$ maps to a single state of the original LTS P. For $\mathring{P}$ this is obvious as it is based on a copy of P. For $\overrightarrow{P}$ it is because the equivalence relation used to establish the states requires that two partial traces that are equivalent have the same continuations. The begin state of the identifier of the first transition of any such continuation fixes the equivalence class to a given state of the original LTS P.

6.3.2 Reasoning with Unwound Form

The purpose of creating the unwound form of an LTS is to reason about cycles. Suppose a collaboration follows a cycle, so that a pattern of message exchange between the participants is repeated as might happen in an negotiation scenario where participants engage in cycles of bidding and counter-bidding. In such a situation there is sometimes the possibility that messages from different instances of the cycle cannot be distinguished, and this could result in participants misdiagnosing the state of the

collaboration. Perhaps a message from the second iteration of the cycle arrives quickly and is incorrectly interpreted as belonging to the first cycle.

In the unwound form the first iteration of any cycle is represented in $\overrightarrow{P}$ whereas subsequent iterations are represented in $\mathring{P}$. Provided it can be shown that messages that correspond to transitions in $\overrightarrow{P}$ can be reliably distinguished from messages that correspond to exchanges in $\mathring{P}$, this kind of "cycle confusion" cannot take place.

6.4 Topological Reduction

Topological Reduction is a technique that we use in the context of choreography theory, to extract participant contracts from a global definition of the behaviour of a collaboration. Given a machine P and a filter $\mathcal{F} :: \Lambda_P \rightarrow \mathit{boolean}$ on the alphabet of P we wish to define a reduced machine R whose alphabet (set of labels on transitions) is $\{x | x \in \Lambda_P \wedge \mathcal{F}(x)\}$ and which satisfies:

$$t \in \mathit{traces}_P \Rightarrow \mathit{filter}_{\mathcal{F}}(t) \in \mathit{traces}_R \tag{6.2}$$

In other words every trace in P has an equivalent trace in R.

Note that (6.2) in general allows R to introduce new traces, that were not present in P. We will describe two methods of reduction, *simple* and *exact*, where the first can result in new traces in the reduction and the second cannot. In the discussion of reduction in choreography in Section 9.11 we will discuss the merits of exact reduction.

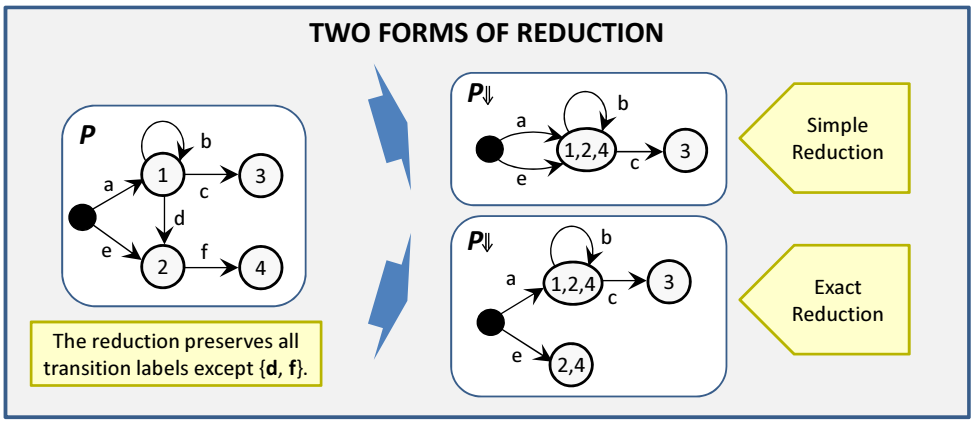


Figure 6.2: Two Forms of Reduction

6.4.1 Simple Reduction

The *simple reduction* involves:

- Removing some of the transitions.
- Merging the states that are disconnected by the removed transitions.

The upper reduction on the right of Figure 6.2 shows an example. The reduced machine is produced by removing transitions carrying the labels d or f.

Formally, simple reduction is defined as follows. Suppose that we have an LTS P with topological representation $\langle \Lambda_P, \Sigma_P, \Gamma_P, \Delta_P \rangle$. We define a binary relation $\Phi_{(\mathcal{F},P)} \subseteq \Sigma_P \times \Sigma_P$ as:

$$\langle \sigma_i, \sigma_j \rangle \in \Phi_{(\mathcal{F},P)} \Leftrightarrow \exists\, x \in \Lambda_P \text{ with}$$
$$\neg \mathcal{F}(x) \wedge (\langle \sigma_i, x, \sigma_j \rangle \in \Theta_P \vee \langle \sigma_j, x, \sigma_i \rangle \in \Theta_P)$$

This means that a pair of states belongs to the relation $\Phi_{(\mathcal{F},P)}$ if there is a transition between the two states in either direction using an action that is not retained by $\mathcal{F}$.

We now define a relation $\mathcal{R}_{(\mathcal{F},P)}$ between states as:

$$\sigma_k \mathcal{R}_{(\mathcal{F},P)} \sigma_l \Leftrightarrow\ (\sigma_k = \sigma_l)\ \vee\ \exists\, t \in {\Sigma_P}^* \text{ with:}$$
$$(t_1 = \sigma_k) \wedge (\mathit{last}(t) = \sigma_l) \wedge (1 \leqslant i < \mathit{length}(t) \Rightarrow \langle t_i, t_{i+1} \rangle \in \Phi_{(\mathcal{F},P)}) \tag{6.3}$$

so $\sigma_k \mathcal{R}_{(\mathcal{F},P)} \sigma_l$ means that either σ_k and σ_l are the same state or there is a chain of states that joins the two and all links of the chain are removed (not retained) by $\mathcal{F}$. Note that the transitions that form the links of the chain do not have to be in the same direction.

It is easy to see that $\mathcal{R}_{(\mathcal{F},P)}$ is an equivalence relation. We designate the set of equivalence classes by $\Sigma_{(\mathcal{F},P)}$ and the quotient function of the relation by: $\mathcal{R}_{(\mathcal{F},P)} :: \Sigma_P \rightarrow \Sigma_{(\mathcal{F},P)}$.

The construction of the reduction has two steps:

Step 1. We create a machine $P{\downarrow}$ as follows:

$$\Lambda_{P\downarrow} = \{\, \theta \mid \theta \in \Theta_P \ \wedge \ \mathcal{F}^{\Theta}(\theta)\} \tag{6.4a}$$

$$\Sigma_{P\downarrow} = \Sigma_{(\mathcal{F},P)} \tag{6.4b}$$

$$\Gamma_{P\downarrow} = \{\langle \theta, \sigma\rangle \mid \mathcal{F}^{\Theta}(\theta) \ \wedge \ \langle \mathit{label}(\theta), \sigma'\rangle \in \Gamma_P \ \wedge \ \mathcal{R}_{(\mathcal{F},P)}(\sigma') = \sigma\} \tag{6.4c}$$

$$\Delta_{P\downarrow}(\langle \theta, \sigma\rangle) = \{\mathcal{R}_{(\mathcal{F},P)}(\sigma') \mid \sigma' \in \Delta_P(\langle \mathit{label}(\theta), \sigma''\rangle) \ \wedge \ \mathcal{R}_{(\mathcal{F},P)}(\sigma'') = \sigma\} \tag{6.4d}$$

where:

- $\mathcal{F}^{\Theta}$ is defined in terms of the reduction filter $\mathcal{F}$ by $\mathcal{F}^{\Theta}(\theta) \Leftrightarrow \mathcal{F}(\mathit{label}(\theta))$
- In (6.4c): $\sigma \in \Sigma_{P\downarrow}$ and $\sigma' \in \Sigma_P$.
- In (6.4d): $\sigma \in \Sigma_{P\downarrow}$ and $\sigma', \sigma'' \in \Sigma_P$.

Step 2. We form the reduction $P{\Downarrow}$ by converting the labels on $P{\downarrow}$ to elements of $\Lambda(P)$ using the *label* function, so a label x in $P{\downarrow}$ is replaced by $\mathit{label}(x)$ in $P{\Downarrow}$. The states and transition topology of $P{\Downarrow}$ is the same as that of $P{\downarrow}$, excepting that two or more transitions using the same start and end states may be combined if they have they same label after conversion.

This construction forms the LTS whose states are the equivalence classes of $\mathcal{R}_{(\mathcal{F},P)}$ and from which transitions not retained by $\mathcal{F}$ have been eliminated. A transition in $P{\Downarrow}$ that is retained in the reduction under $\mathcal{F}$ joins, as its start and end states in $P{\Downarrow}$, the images of the start and end states in P under the quotient function of $\mathcal{F}$. The LTS $P{\Downarrow}$ is called the *simple reduction* of P under the filter $\mathcal{F}$.

6.4.2 Exact Reduction

Exact reduction of a machine is a reduction that does not introduce any new traces. If $P{\Downarrow}$ is an exact reduction of P under a filter $\mathcal{F}$ then $P{\Downarrow}$ satisfies:

$$\mathit{traces}_{P\Downarrow} = \{\mathit{filter}_{\mathcal{F}}(t) \mid t \in \mathit{traces}_P\} \tag{6.5}$$

The key idea in creating $P\Downarrow$ as an exact reduction of P is to use a set of states for $P\Downarrow$ that is the powerset of the set of states of P. A given state in $P\Downarrow$ is the set of states that could pertain in P after a given sequence of actions (transition labels) preserved by the reduction.

Formally, exact reduction is defined as follows. Suppose that we have an LTS P with topological representation $\langle \Lambda_P, \Sigma_P, \Gamma_P, \Delta_P \rangle$ and a boolean filter $\mathcal{F} :: \Lambda_P \rightarrow \mathit{boolean}$. We wish to construct a machine $P\Downarrow$ whose alphabet (set of labels on transitions) is reduced to the set retained by $\mathcal{F}$ and which satisfies (6.5). The construction has two steps:

Step 1. We create a machine $P\downarrow$ as follows:

$$\Lambda_{P\downarrow} = \{\, \theta \mid \theta \in \Theta_P \ \wedge \ \mathcal{F}^{\Theta}(\theta) \} \tag{6.6a}$$

$$\Sigma_{P\downarrow} = \mathit{power}(\Sigma_P) \tag{6.6b}$$

$$\Gamma_{P\downarrow} = \{\, \langle \theta, \sigma \rangle \mid \mathcal{F}^{\Theta}(\theta) \ \wedge \ \mathit{begin}(\theta) \in \sigma \,\} \tag{6.6c}$$

$$\begin{aligned} \Delta_{P\downarrow}(\langle \theta, \sigma \rangle) = \Big\{ \{\, \sigma' \mid \ \exists \quad \sigma'' \in \sigma \quad \text{and} \quad \theta' \in \Theta_P \quad \text{and} \\ \pi \in \mathit{prefixes}(\overrightarrow{\Pi}_P^m(\mathit{end}(\theta'))) \quad \text{with} \\ \mathit{label}(\theta') = \mathit{label}(\theta) \quad \wedge \quad \mathit{begin}(\theta') = \sigma'' \quad \wedge \\ \mathcal{F}^{\Theta}(\pi) = <> \quad \wedge \quad \sigma' \in \mathit{visits}(\pi) \,\} \Big\} \end{aligned} \tag{6.6d}$$

where:

- $\mathcal{F}^{\Theta}$ is defined in terms of the reduction filter $\mathcal{F}$ by $\mathcal{F}^{\Theta}(\theta) \Leftrightarrow \mathcal{F}(\mathit{label}(\theta))$
- In (6.6b) and (6.6c): $\sigma \in \Sigma_{P\downarrow}$.
- In (6.6d): $\sigma \in \Sigma_{P\downarrow}$ and $\sigma', \sigma'' \in \Sigma_P$.

Note that a value of $\Delta_{P\downarrow}(\langle \theta, \sigma \rangle)$ in (6.6d) is always a set containing exactly one member of $\Sigma_{P\downarrow}$.

Step 2. We form the reduction $P\Downarrow$ by converting the labels on $P\downarrow$ to elements of $\Lambda(P)$ using the *label* function, so a label θ in $P\downarrow$ is replaced by $\mathit{label}(\theta)$ in $P\Downarrow$. The states and transition topology of $P\Downarrow$ is the same as that of $P\downarrow$, excepting that two or more transitions using the same start and end states may be combined if they have they same label after conversion.

The LTS $P\Downarrow$ is called the *exact reduction* of P under the filter $\mathcal{F}$. The construction uses a state space defined in (6.6b) using the power set of the set of states of P. Each member of a given element of $\Sigma_{P\Downarrow}$ represents a set of states in P that could pertain, given the actions that have been allowed so far. As defined in (6.6d) a transition for an action x between σ and σ' in $P\Downarrow$ exists if **and only if** one or paths in P obey the following:

- Has a first transition with action label x.
- Starts from a state of P in σ,
- Following the first transition, has one or more continuation paths (possibly of zero length) that are completely removed by the reduction filter, and which continuation paths visit states of P belonging to σ'.

Moreover, all the states of σ' must be accounted for by visitation of a continuation path generated as described above. The idea is that the end state σ' in the reduction represents **exactly** the uncertainty as to what state now pertains in P after an action, given both that there may be more than one transition in P labelled with that action, and the partial knowledge of what has taken place in P because some transitions are filtered out and so "invisible" . Because the representation of uncertainty is exact, no new traces can be introduced and thus the reduction satisfies (6.5).

6.4.3 Path-Deterministic Reduction

In both reduction methods, simple and exact, we define the process as two steps where the first step generates an intermediate LTS $P\downarrow$ that uses transition identifiers from P as its alphabet. This intermediate LTS is called the *path-labelled reduction* of P. This is used to form the final reduction $P\Downarrow$ in the second step.

As the elements of the alphabet of the path-labelled reduction $P\downarrow$ belong to Θ_P (the *transition identifiers* in P), each transition in $P\downarrow$ corresponds to **exactly one** transition in P.

The path-labelled reduction is more than just an artefact of the construction, as it is used in the statement of conditions for realizability of a choreography discussed in Section 9.6. The following definitions will be important in that context.

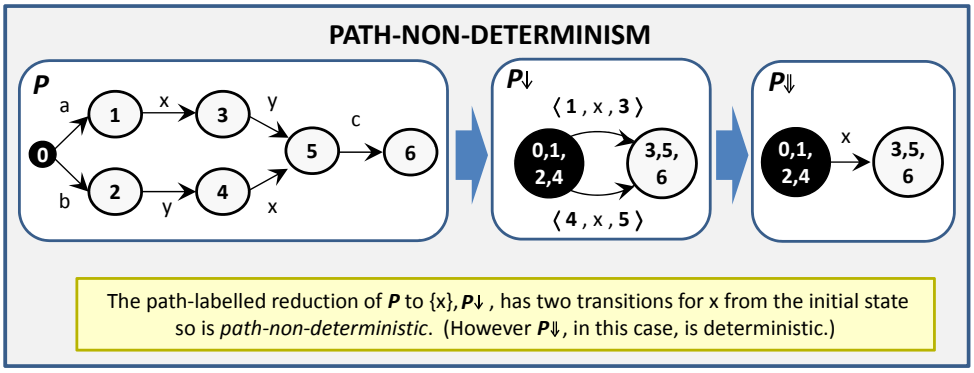


Figure 6.3: Path-non-deterministic Reduction

A reduction $P\Downarrow$ is said to be *path-non-deterministic* if the following condition holds:

$$\text{There is a state in the path-labelled reduction } P\downarrow \text{ with outgoing transitions labelled } \theta_1 \text{ and } \theta_2 \text{ with } \theta_1 \neq \theta_2 \text{ but } \mathit{label}(\theta_1) = \mathit{label}(\theta_2) \tag{6.7}$$

Fig 6.3 shows an example. A reduction that is not path-non-deterministic is said to be *path-deterministic*. Note that if $P\Downarrow$ is path-deterministic then it is also bound to be deterministic in the usual sense that every state has at most one transition for a given label in its alphabet.

Fn: *tranTrace*. If $P\Downarrow$ is path-deterministic then the transitions in $P\downarrow$ and $P\Downarrow$ are in one-to-one correspondence and so the two systems are isomorphic. Moreover, because if $P\Downarrow$ is path-deterministic it is also deterministic, given $\tau \in \mathit{traces}_{P\Downarrow}$ we can obtain the corresponding path as $\mathit{path}_{P\Downarrow}(\tau)$. Under the isomorphism there will then be exactly one corresponding path in $P\downarrow$. If we apply $\mathit{labels}()$ to this path we obtain a trace in $P\downarrow$ corresponding to τ in $P\Downarrow$. This is called the *transition trace* for τ and we define the function $\mathit{tranTrace} :: \mathit{traces}_{P\Downarrow} \rightarrow \mathit{traces}_{P\downarrow}$ to return it. Note that the elements of $\mathit{tranTrace}(\tau)$ are members of Θ_P, the transition identifiers in P.

6.4.4 Ambiguous States

We will be interested in whether a state in a reduction maps to a unique state of the original machine. A state $\sigma \in \Sigma_{P\Downarrow}$ is termed *unambiguous* iff:

$$\forall\, \sigma' \in \sigma : \quad (\sigma'' \in \Sigma_{P\Downarrow}) \wedge (\sigma' \in \sigma'') \Rightarrow (\sigma'' = \{\sigma'\}) \tag{6.8}$$

where $\sigma' \in \Sigma_P$ and $\sigma, \sigma'' \in \Sigma_{P\Downarrow}$.

The condition (6.8) requires that σ contains only a single state of P, and that this state of P is always by itself in any state of $P\Downarrow$ to which it belongs. A state that is not unambiguous is termed *ambiguous*.

6.5 Connected Form

Finally we define the creation of a topologically connected stored-state approximation of a derived-state machine called the *connected form*. The motivation for creating a connected form of a derived-state machine is to allow application of a topological reasoning technique that demands a stored-state form. Examples of using connected form machines in formal reasoning will be found in Chapter 9.

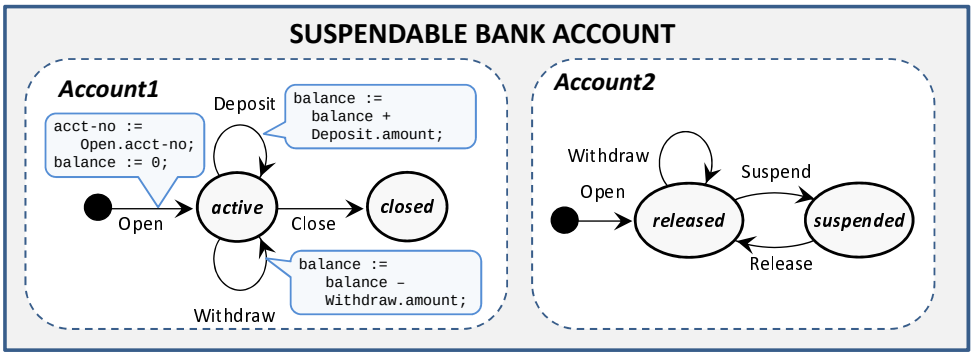


Figure 6.4: Suspendable Bank Account

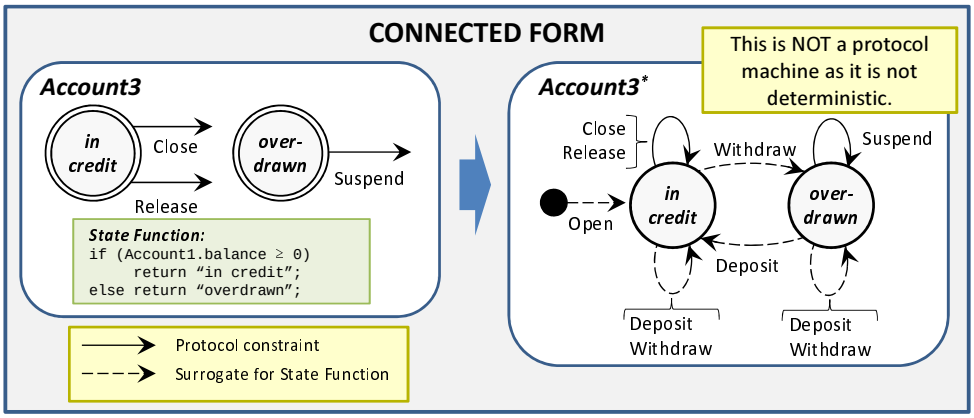


Figure 6.5: Connected Form

Figure 6.4 shows a bank account that can be suspended. When in the suspended state *Account*2 prevents any *Withdraw* action. The left side of Figure 6.5 shows a further derived-state machine *Account*3 which controls when *Close*, *Release* and *Suspend* may take place based on the balance of the account. The right hand side of Figure 6.5 shows *Account*3* which is a connected form representation of *Account*3. While the original uses a derived state, the connected form is a traditional stored-state machine.

The connected form uses the transitions shown as dashed arrows in Figure 6.5 as a surrogate for the state function. Whereas the state function tells us **exactly** the effect of depositing or withdrawing funds on the state of the machine, the surrogate transitions can only show the **possibilities**. Generally the transitions added as surrogates for a state function create a machine that is non-deterministic, so does not qualify as a protocol machine. This is the case here and *Account*3* is non-deterministic. The connected form, however, is a derivative created **only for topological analysis/reasoning purposes**. It does **not** replace the machine from which it was derived.

6.5.1 Creation of Connected Form

Creation of the connected form of a derived-state machine is done in three steps:

Step 1. Form the state space as defined by the machine's state function, but with the addition of a • for the initiated state.

Step 2. Add transitions between the states to create a topological surrogate for the state function.

Step 3. Create transitions representing the constraints of the original machine, but in topologically connected form.

Step 1 identifies the three states: $\{\bullet, \textit{in credit}, \textit{overdrawn}\}$. In step 2, transitions (shown as dashed for graphical emphasis) are added as a surrogate for the state function. In step 3, transitions for *Close*, *Suspend* and *Release* are shown, starting and ending in the same state as they do not affect the state of this machine.

Step 2 merits some elaboration. The transitions that need to be added in this step (those shown as dashed arrows) are identified as follows:

- The state function is examined to determine the set of attributes referenced in the calculation. In this case, it is just `balance` from *Account1*.
- The machine(s) that own these attributes are examined to determine which transitions cause their values to change. In this case it is *Open*, *Deposit* and *Withdraw* as can be seen from the update bubbles attached to these transitions in *Account1*.
- Corresponding transitions are added to each state of the connected form machine being constructed, according to how they can alter the state. For instance, *Withdraw* decreases the value of `balance` so its effect on *Account3** is either to leave the state unchanged or to cause it to change from *in credit* to *overdrawn*.

Finally note also that, in some cases, step 3 may involve removing transitions added in step 2. Suppose that *Account*3 were to constrain *Withdraw* only to happen in the state *in credit*, as it does *Close*, then the dashed transition for *Withdraw* starting from *overdrawn* added in step 2 would be removed in step 3.

6.5.2 Formalization of Connected Form

The notion of a connected form can be formalized as follows. Suppose that we have a model M defined over a universe $\mathfrak{U}$, so that M is an independent machine with $\Omega_M = \mathcal{U} \backslash \mathcal{A}$. Suppose P is a derived state component machine of M. The connected form machine P^* is an independent version of P whose topology can be used to reason about the effect P in M. We assume that P is finite-state protocol machine with a state attribute $state_P \in \Omega_P$.

The connected form P^* is defined as:

$$\Omega_{P*} = \{state_P\} \tag{6.9a}$$

$$\mathcal{B}_{P*} = restr(\mathcal{B}_M\ ,\ \mathcal{A} \cup \{state_P\}) \tag{6.9b}$$

The connected form P^* of P is created per (6.9) by redacting M so that the offered data is only the state of P. The machine so generated will not, in general, conform to the well-formedness conditions (4.10d) and (4.10e) for a protocol machine so an LTS con-

structed using the recipe given in Section 4.4.4 will, in general, be non-deterministic.[2] Nevertheless, the LTS so constructed will show:

- the possible effect of all actions in the model on the state of P, and
- the constraining effect of the state of P on all actions of the model.

and can be used to reason about the effect of P in M, as follows.

By (6.9b) $\mathcal{B}_{P*}$ is an abstraction of $\mathcal{B}_M$ and so any sequence of actions or transitions between states not possible in P^* will not be possible in M either. Thus machine *Account3** in Figure 6.5 can be used to make assertions about the sequencing of actions that *Account3* allows, for instance:

- A *Suspend* cannot immediately follow an *Open*.
- A *Release* or *Close* cannot immediately follow a *Suspend*.

As observed in Section 5.1.4, rendering a full model in pure topological form generally results in a state space too large to handle, a well known problem in topological model proving known as *state space explosion* [51]. Using connected form machines allows a degree of such reasoning while keeping the state space small.

[2]Strictly speaking, this procedure would not generate a topological representation with a • as the initial state. However this is normally added, as it has been in *Account3** in Figure 6.5, to comply with the conventions for LTS topology given in Section 6.1.

Part III

Applications

Chapter 7

Object Modelling

As the introduction to this thesis explains, PM originally emerged from ideas in object modelling and this chapter expands on this theme by describing the use of PM as an object modelling medium. To motivate the use of PM in this context, consider the following example.

A certain bank allows its customers to open and maintain any number of separate accounts, but imposes the rule that a customer may only withdraw funds from an account if the result of the withdraw is that she remains in credit overall (where "overall" means across all of her accounts).

A graphical model for this is shown in Figure 7.1.

7.1 The Challenge

A significant driver in the development of PM was the apparent difficulty of extending earlier methods of object lifecycle modelling to cases such as the one sketched above. Consider the Shlaer/Mellor model of a bank account shown on the left side of Figure 1.1. To adapt this model for the rules given above, our reasoning might proceed as follows:

- Clearly the machines for each account will maintain the individual account balances. The overall balance will need to be kept by the customer machine.

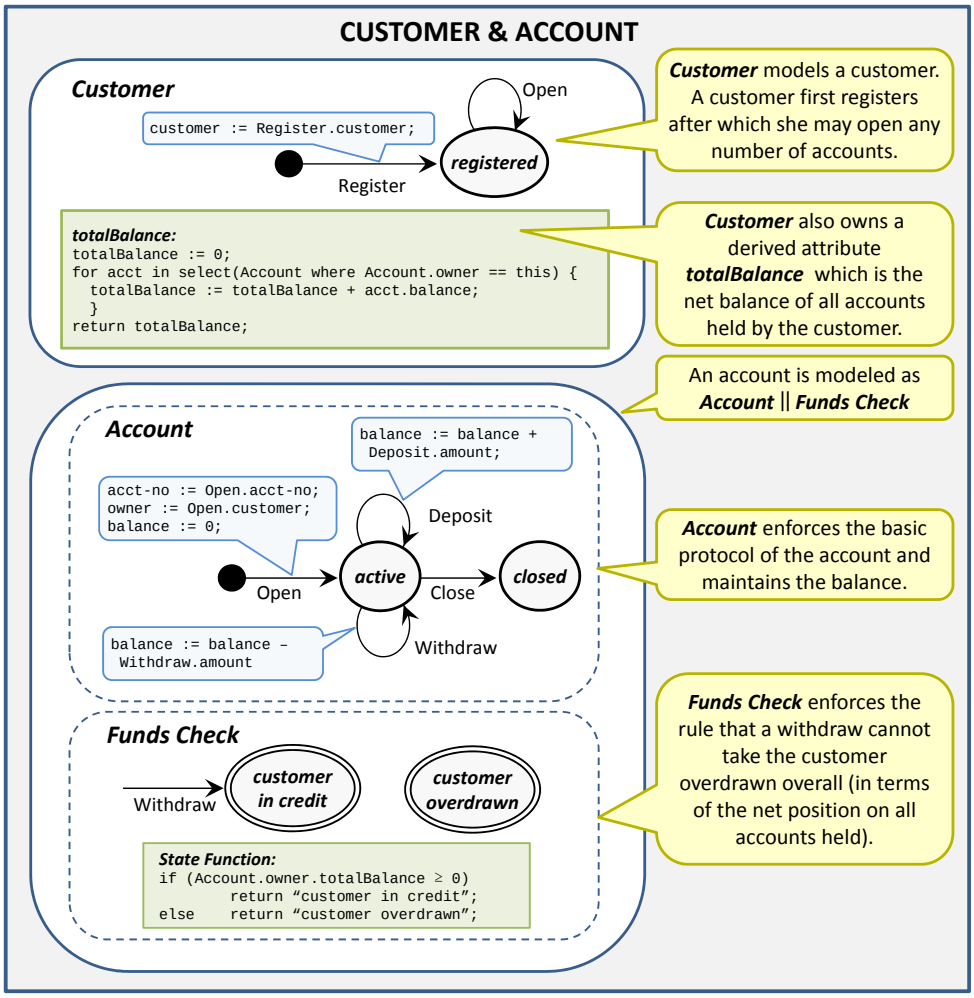


Figure 7.1: Customer and Account

- This means that all actions that affect the customer balance, all the deposits and withdraws for all the customer's accounts, must be fed to the customer machine.
- As only it knows the funds available in the overall balance, the customer machine has to make the determination of whether a withdrawal is allowed. If it allows a withdrawal it must update its own record of the overall customer balance and forward the withdraw to the account to which it applies.

As well as being complex, this scheme violates the DRY ("Don't Repeat Yourself") Principle.[1] This principle, which has no formal basis in computer science, distils accumulated practical experience in building models where data and behaviour interact,

[1] See `http://en.wikipedia.org/wiki/Don't_repeat_yourself`

and holds that models that contain redundancy tend to be more complex, fragile and difficult to amend than models that do not.[2] Keeping a *total balance* at the customer level is not compatible with this principle, as it can be derived from the balances in the individual accounts.

An alternative might be to consider having the customer machine read and sum the balances from the individual accounts, but as the machines in a Shaer/Mellor model operate asynchronously it is hard to see how it would be possible to prevent updates to individual accounts while the summing process is underway. It is then not possible to maintain the discipline that customer remains in credit overall. This is the same issue that was discussed in section 2.1. Perhaps some kind of locking mechanism could be used to prevent updates in the accounts while summing is in progress. This would require a mechanism along the lines of that in "LOTOS with Global Variables" examined in Section 5.2.1 with the problems of complexity noted there.

Adaptation of the Jackson notation shown on the right side of Figure 1.1 presents different issues. The notation used for action ordering is essentially a graphical form of regular expression (using $*$ to represent repetition and $\circ$ for selection) and does not support ordering constraints based on data, such as the constraint on *Withdraw* based on *balance* that we require here.

Data intensive systems are riddled with behavioural rules of the sort exhibited in this example, presenting us with the challenge of being able to represent such rules simply and without the need for redundancy of fact. Meeting this challenge was a key driver in the development of PM.

The use of PM for building such models is an example of the *Constructive* mode of use, as described in Section 4.4.5.

7.2 ModelScope

ModelScope is a language and tool created by the author and a colleague, Nicholas Simons, to explore the possibilities of executable PM models. The main features of ModelScope are:

[2]The author can attest to the value of this principle from his own experience in software engineering.

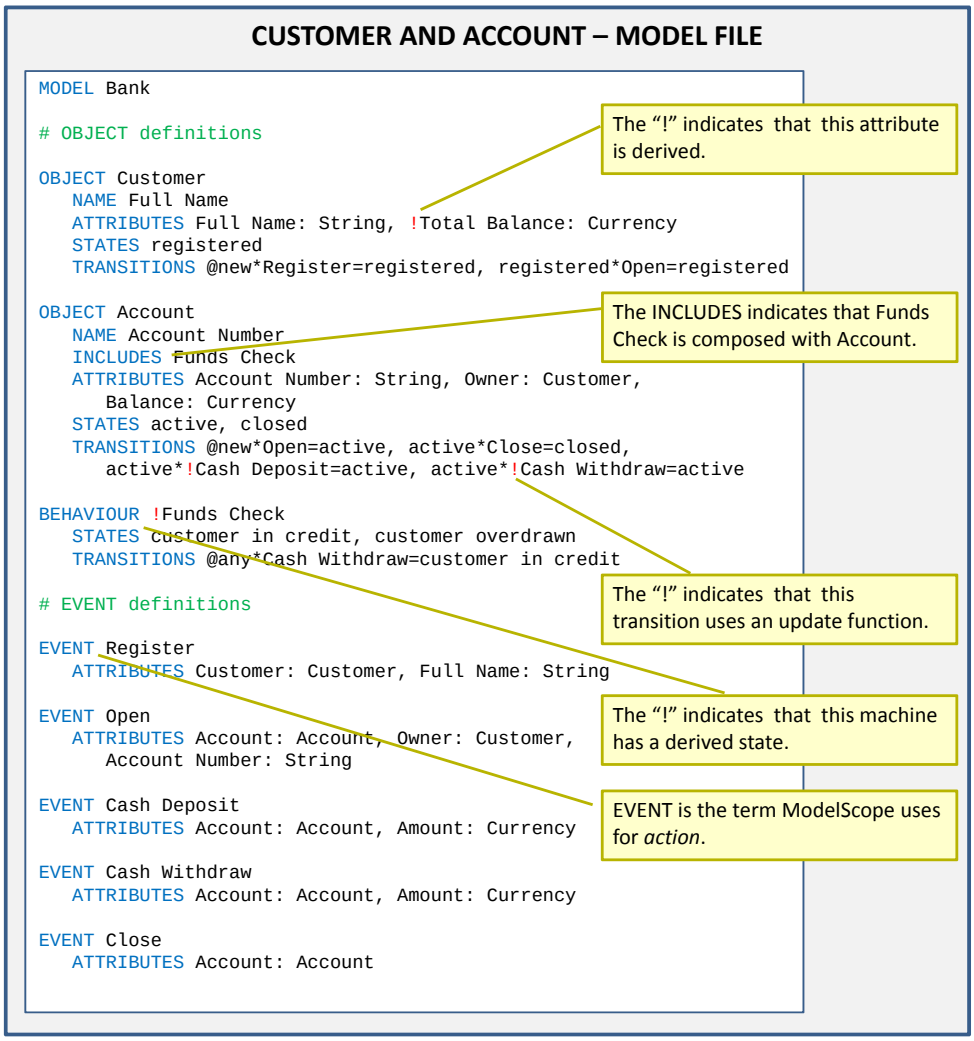


Figure 7.2: Customer and Account - Model File

- A textual modelling medium, using a combination of a new language to describe protocol machines (Figure 7.2 shows the model definition for the Customer and Account example in this language); and Java for defining functions required to perform updates and derive attributes and states (Figure 7.3 shows the Java functions for the Customer and Account example).

- An execution environment that interprets the model to generate a browser based run-time user interface. Figure 7.4 shows a snapshot of the user interface running the Customer and Account model. In the snapshot the user has just attempted a *Cash Withdraw* from Account `001` belonging to customer `Fred` that exceeds Fred's

CUSTOMER AND ACCOUNT – CALL BACKS

```
public class Account extends Behaviour {

// Update Balance for a Deposit
   public void processCashDeposit(Event event, String subscript) {
      int newBalance = this.getCurrency("Balance") + event.getCurrency("Amount");
      this.setCurrency("Balance", newBalance);
      }

// Update Balance for a Withdraw
   public void processCashWithdraw(Event event, String subscript) {
      int newBalance = this.getCurrency("Balance") - event.getCurrency("Amount");
      this.setCurrency("Balance", newBalance);
      }

}
```

These methods implement the update bubbles for *Deposit* and *Withdraw* in *Account*.

```
public class Customer extends Behaviour {

// Calculate the Total Balance for the Customer
   public int getTotalBalance() {
      int totalBal = 0;
      Instance[] myAccounts = this.selectByRef("Account", "Owner");
      for (int i = 0; i < myAccounts.length; i++)
         totalBal = totalBal +  myAccounts[i].getCurrency("Balance");
      return totalBal;
      }

}
```

This select returns an array of all *Account* objects that have attribute *Owner* equal to this (*Customer*).

This method implements the derivation of *TotalBalance* in *Customer*.

```
public class FundsCheck extends Behaviour {

// Return state based on the Total Balance of the Owner of the Account
   public String getState() {
      return (this.getInstance("Owner").getCurrency("Total Balance") < 0) ?
         "customer overdrawn" : "customer in credit";
      }

}
```

This method implements the state function for *Funds Check*.

Figure 7.3: Customer and Account - Call Backs

overall balance. This has taken the account into the state *customer overdrawn* and generating a post-state error in the *Funds Check* machine.

The specification of ModelScope was done before much of the theoretical work on the denotational semantics of PM given in the first part of this thesis; and while it is largely faithful to the theory that has since been established it is not a complete implementation of PM and in some minor ways is incorrect.

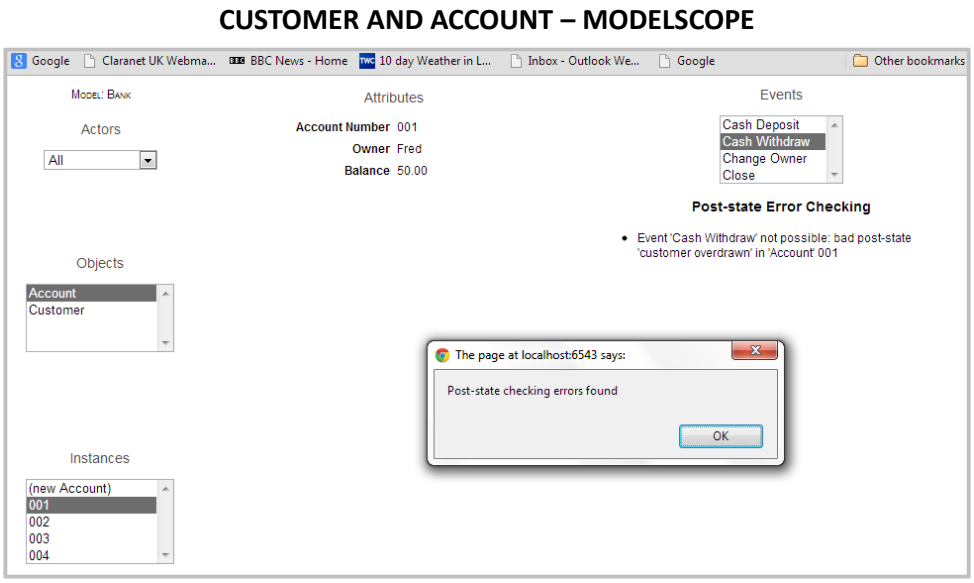


Figure 7.4: Customer and Account - ModelScope

7.3 Modelling Variation

This section is concerned with the question of how to model *behavioural variants*. We examine the traditional inheritance based approach and compare it with the compositional approach in PM. The key question is how inheritance of behaviour should work.

7.3.1 Inheritance of Behaviour

Extending *inheritance* to include behaviour is problematic because the normal mechanisms of inheritance, as applied to *attributes* and *operations* (*methods*), do not apply to *behaviour* in any obvious way. Attributes and operations take the form of a list or collection of individual items, and these lists can be refined by selectively overriding individual members of the list and/or extending the list by adding new members. But the behaviour of an object, if represented as a single modelling artefact such as a UML Statechart, is a single undifferentiated entity. There is no obvious simple way of merging multiple statecharts, or of extending or refining all or part of a single statechart.

Most research into how behaviour should be inherited has focused on making formal statements about the relationship between the behaviour of a child and the behaviour of its parent in a generalization hierarchy. Taking their cue from the work

done by Liskov on typing structures [52], the "Three Amigos" in their *UML Users' Guide* [15] give *substitutability* as the proper determinant of compatibility between parent and child in a generalization hierarchy:

> *A generalization is a relationship between a general thing (called the superclass or parent) and a more specific kind of that thing (called the subclass or child).*
> ...
> *The child may even add new structure and behaviour, or it may modify the behaviour of the parent. In a generalization relationship, instances of the child may be used anywhere instances of the parent apply - meaning that the child is substitutable for the parent.*
> **– Grady Booch, James Rumbaugh and Ivar Jacobson [15]**

However UML gives no rules for Liskov conformant refinement of statecharts, and subsequent academic work to formulate such rules has shown that it is not simple, as evidenced by the following quote from Simons et al. in their paper [75] on this subject:

> *The basic premise of component substitutability is 'no surprises', yet these examples show how difficult it can be to avoid unexpected behaviour or even failure. The syntactic rules for matching interfaces are well known: a component must provide at least as many methods as expected, and the signatures of those it provides must match (be subtypes of, with covariant results and contravariant arguments) the expected signatures. However, previously published semantic rules for matching behaviour have ranged from the cautious to the liberal.*
> **– Anthony Simons, Michael Stannet, Kirill Bogdanov and Michael Holcolme [75]**

Other researchers into behaviour refinement and conformance have concluded that substitutability does not uniquely define conformance and have proposed different kinds of conformance. For instance:

- Ebert and Engels propose *observable compatibility* or *invocable compatibility* [21]
- Schrefl and Stumptner propose *observation consistency*, *weak invocation consistency* or *strong invocation consistency* [73]
- van der Aalst and Basten propose *protocol inheritance*, *projection inheritance*, *protocol/projection inheritance* or *life-cycle inheritance* [76]

It seems that different kinds of conformance align to different circumstances or requirements, although determination of the appropriate alignment is still a matter of debate.

7.3.2 Mixins

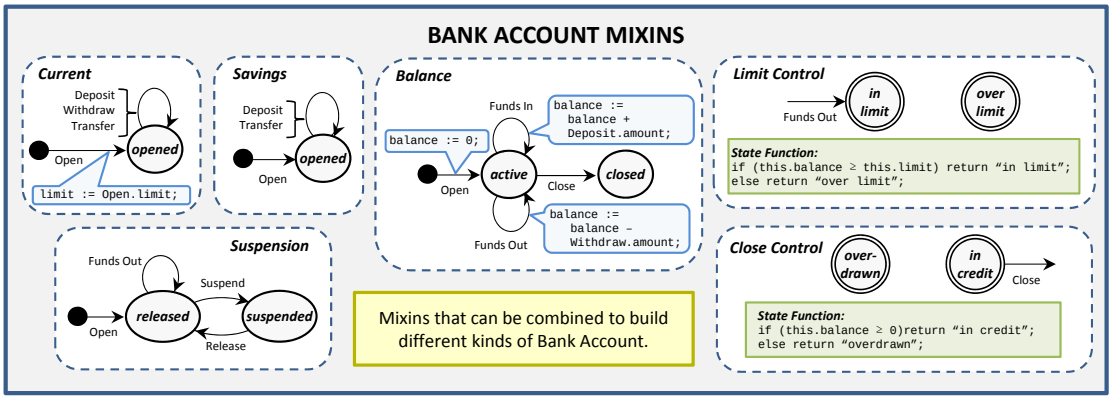
A key observation in PM is that if $R = P \parallel Q$ then R has conformance of the kinds underlined in the list above to both P and Q. This is by virtue of the semantics of composition, and there is no requirement to follow any refinement rules. The natural approach in PM is to create machines that represent partial definitions and to model variation by combining these machines in different combinations in the style of "lego bricks". This corresponds to the paradigm known as *mixins* which first appeared in the *Flavors* programming language developed by Howard Cannon in the late 1970s [19] and discussed from a theoretical point of view by Bracha and Cook [16].

> *Object-oriented programming systems have typically been organized around hierarchical inheritance. Unfortunately, this organization restricts the usefulness of the object-oriented programming paradigm. This paper presents a non-hierarchically organized object-oriented system, an implementation of which has been in practical use on the MIT Lisp Machine since late 1979.*
> **– Howard Cannon [19]**

Others have expressed a similar view that composition has advantages over inheritance.

> *Favor object composition over class inheritance.*
> **– Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides [28]**

Figure 7.5 shows a set of machines that could be used as mixins to build different kinds of bank account. The repertoire of accounts in a system would be determined by the instantiable assemblies seeded in the model at initiation (as described in Section 4.6.3) or, equivalently, the assemblies modelled by the `INCLUDES` structures in ModelScope. The table in the lower part of Figure 7.5 shows the relationship between the actions shown as labels on the transitions in the graphical syntax and how these might be represented as action fields. Note that the mixins in Figure 7.5 are *object machines*, as defined in Section 4.6.2; and that the term *this* used in the state functions and



Graphics		Action Fields					
Machine	**Action Label**	**Account Type**	**Action Type**	**Account**	**Direction**	**Source**	**Target**
Current	*Open*	"Current"	"Open"	this	"Neither"		
	Deposit		"Deposit"	this	"In"		
	Withdraw		"Withdraw"	this	"Out"		
	Transfer		"Transfer"		"Both"		this
	Transfer		"Transfer"		"Both"	this	
Savings	*Open*	"Savings"	"Open"	this	"Neither"		
	Deposit		"Deposit"	this	"In"		
	Transfer		"Transfer"		"Both"		this
	Transfer		"Transfer"		"Both"	this	
Suspension	*Open*		"Open"	this	"Neither"		
	Suspend		"Suspend"	this	"Neither"		
	Funds Out			this	"Out"		
	Funds Out				"Both"	this	
	Release		"Release"	this	"Neither"		
Balance	*Open*		"Open"	this	"Neither"		
	Funds In			this	"In"		
	Funds In				"Both"		this
	Funds Out			this	"Out"		
	Funds Out				"Both"	this	
	Close		"Close"	this	"Neither"		
Limit Control	*Funds Out*			this	"Out"		
	Funds Out				"Both"	this	
Close Control	*Close*		"Close"	this	"Neither"		

Figure 7.5: Bank Account Mixins

in the table in Figure 7.5 refers to the common object id of a composition of mixins that forms a single object.

If used as the replacement for inheritance, mixins provide an expressive power equivalent to multiple inheritance [16]. While mixins have been supported as constructs in programming languages, for example in Lisp, Simula, Python and Smalltalk, they have not generally been seen as a construct in modelling languages, which have tended to employ hierarchical inheritance structures.

PM modelling supports various kinds of re-use and polymorphic behaviour, for instance in the account example:

- The *Balance* machine can be used in any type of account to maintain a *balance*.
- The *Limit Control* and *Close Control* can be used in any account that has a *balance*, whether that balance is maintained by a *Balance* machine or in some other way.
- The *Funds In* and *Funds Out* actions in the Balance machine do not have a specified action type and are thus abstract. If a new action *Charge Fee* were introduced and it was also coded as `(Account=this)` and `(Direction="Out")` it would cause the balance to be decremented without any change to the *Balance* machine.

The ability of PM to model a complex domain and to support evolution of the model as the domain changes is noted by Verheul and Roubtsova in an exercise modelling the Health Insurance Industry [79]:

> *Two sorts of conclusions can be drawn from this study. Firstly, the paper illustrates the property of local changeability of protocol models using the real process of changes taking place in insurance industry. The changes in insurance products introduced in the last six years were modelled locally. Even changes in the business process (*Personal Budget *and* Mandatory Deductibles*) remained local as they were modelled as new protocol machines. New protocol machines were equally composed with the existing protocol machines in the model and the ordering of sequences of accepted events of old machines was preserved in the new model. Localization of changes in the business process is the direct consequence of the CSP parallel composition used in Protocol Modelling and cannot be achieved in many other conventional notations, for example in state machines and activity diagrams.*

> *Secondly, the result of the study is an executable and evolvable reference model of the Health Insurance Applications that combines the types of Insurance Products available at the moment. The model is abstract, technology independent, represents entities, relationships and business processes and embodies the basic goals and ideas of Insurance Applications. Based on these properties . . . the model can be seen as a reference model for the Health Insurance Industry.*
> **– Jaco Verheul, Ella Roubtsova [79]**

7.3.3 Mixins as Aspects

Modelling based on hierarchies is prone to what Ossher and Tarr refer to as the "Tyranny of the Dominant Decomposition" [66], whereby the structure of the model will be dominated by the model architect's view of the key elements of the model and their relationships and connections. The more a core modelling language imposes hierarchical structures on a model, the less likely that all the elements of a problem and their relationships can be accommodated, as the structure chosen for some elements necessarily means that others cross-cut. This problem is immediately manifest with single inheritance structures, where the single hierarchy of the class structure is the tyrant, and the modeller has to resort to using a specialist techniques (such as aspect based modelling languages) to represent those elements that clash with the main structure.

As observed by Filman and Friedman [24] and by Apel et al. [3], a pure mixin approach has no requirement for hierarchy and thus helps avoid this tyranny.

> *In using inheritance to achieve aspects, single superclass inheritance systems require all aspects to match the class structure of the original program, while multiple inheritance systems allow quantification independent of the program's dominant decomposition. Mixins with multiple inheritance are thus a full aspect-oriented programming technology.*
> **– Robert Filman and Daniel Friedman [24]**

The use of PM for aspect-oriented modelling has been examined by the author and Roubtsova in [55].

7.4 Conclusions and Further Work

As presented in the introduction to this thesis, PM has its origins in early notions about object modelling which promoted the importance of modelling object states and lifecycles. PM has built on this ambition by providing a language for modelling behaviour that supports abstraction and polymorphism. The compositional style of PM is in contrast to the mainstream of OO programming, but is arguably well suited to the aim of modelling leading naturally to a "mixin" approach to the creation of behavioural variants.

The ModelScope tool has enabled PM to be used to build and refine object models in industrial applications. Such models are executable and can be used to validate abstractions in the context of creating a software design or creating domain reference models. Ideally the output from such an exercise should feed directly into development of the final software but, given the mismatch with established programming structures mentioned above, this is not straightforward and work is required to make this seamless.

Further work in this area could include investigation of the following questions:

- Could PM ideas be used as the basis for a programming language that supports direct expression of the interaction between state and behaviour, perhaps building on the work done by Aldrich et al. on a paradigm they call "Typestate-Oriented Programming" [2]?
- How does the expressive power of PM mixin-based modelling compare with the more usual inheritance-based approaches, particularly multiple inheritance schemes; and how does the associated typing scheme fit with established formal notions in inheritance and typing, such as the Liskov Substitution Principle?

Chapter 8

Contracts

This chapter explores the idea of *contracts* as partial specifications of software systems. We start by introducing a general framework for defining the semantics of contracts in software engineering. We then describe a form of behavioural contract called a *protocol contract* that uses PM as its form of expression. This form of behavioural contract plays an important role in the work on choreography in Chapter 9.

8.1 The Challenge

The idea of using contracts in the design and testing of software is well established, particularly because of the work of Bertrand Meyer in promoting *Design by Contract*[1] (DbC) [56]. DbC is primarily about ensuring the correctness of the design of *algorithms*, where an algorithm is given a starting state (which can be thought of as the input to the algorithm) and executes to an ending state (which can be thought of as the output from the algorithm). A simple DbC contract is specified in terms of conditions on the starting and ending states termed *pre-* and *post-conditions*. An algorithm satisfying the contract will guarantee that, provided the starting state meets the pre-conditions, the ending state will meet the post-conditions. This is a simple yet powerful concept. For example, the contract for a "sort" algorithm might be specified as follows:

- The pre-condition is that a finite list of items is provided, and the items in the list support a collating operator whereby any two items in the list can be compared

[1] "Design by Contract" is a registered trademark of Eiffel Software in the United States.

to determine whether they are equal or, if not, which is greater.

- The post-condition is that the ending state contains the same items as the start state, but arranged in non-descending sequence by the collating operator.

This completely and formally defines what is required of the sort algorithm, independently of the construction or selection of an algorithm to implement it. DbC works well here because we are able to define the result of the sort without needing to specify or constrain in any way the intermediate states the software may take up during the computation. The concepts and terminology of contracts: *pre-conditions*, *post-conditions*, *assertions* and *invariants* have now become part of the currency of software engineering, and many programming languages provide support for DbC. The entry for Design by Contract in Wikipedia lists over a dozen languages with native support, and numerous products that provide non-native support for DbC.

It is tempting to suppose that the ideas of DbC are universal in computing and can be carried across seamlessly to the world of interactive software, but this is not the case. In a lecture entitled "Turing, Computing and Communication" given by Robin Milner in 2006 celebrating the work of Alan Turing, he calls the world of algorithm the *Old Computing*, and the world of interaction the *New Computing* [61] and advocates that the two worlds should be thought of as different paradigms. Interaction involves *protocol* as well as *computation* and this complicates the idea of contractual conformance. If an interactive software component must obey a certain protocol in communicating with its environment, how do you specify the protocol in the contract? Is it enough that interactive software observes the correct protocol if this ensures that it is compatible with its environment? Or must its contract specify the computation it performs as well as its protocol?

More recently the same question, concerning how to achieve a formal abstract specification of behaviour, has emerged in the arena of executable process languages (BPEL):

> *The WS-BPEL specification also introduces the concept of abstract processes: In contrast to their executable siblings, abstract processes are not executable and can have parts where business logic is disguised. Nevertheless, the WS-BPEL specification introduces a notion of compatibility between such an under-specified* abstract process *and a*

> *fully specified* executable *one. Basically, this compatibility notion defines a set of syntactical rules that can be augmented or restricted by profiles. So far, there exists two such profiles: the Abstract Process Profile for Observable Behavior and the Abstract Process Profile for Templates. None of these profiles defines a concept of behavioral equivalence.*
> ***– Dieter König, Neils Lohmann, Simon Moser, Christian Stahl and Karsten Wolf [45]***

The challenge in this area can be summarized thus: How do we make the concept of contract more general, so that it can be extended from its traditional domain of algorithms to apply to interactive behaviour?

8.2 A Framework for Contract Semantics

> *The paper suggests a general theory of contracts, where a contract is seen as a lemma in the proof of a property of a system. If the lemma is satisfied by a component, then the overall system, embedding that component, should satisfy the property.*
> *Like all good ideas, this seems obvious once you see it.*
> **– Mehmet Akşit** *(In a peer review of the author's paper [53])*

First we propose a conceptual framework for contracts in software. The central idea is that the semantics of a contract must be defined with reference to a *proposition* about the software system as a whole, and that satisfaction of the contract will ensure that it is true. Based on this we propose a *framework* for thinking about contracts, as follows. For a given system S and contract C we require:

- A *formal system of reasoning* $\mathcal{T}$.
- A *proposition* $\mathcal{P}$ that we require to be true of S.
- A set $\mathcal{E}$ of statements true of S and which when combined with the fact that C is satisfied in S enables construction of a *proof* of $\mathcal{P}$ in $\mathcal{T}$.

The idea is that the predicate "the contract C is satisfied" is a required component of the proof of the proposition $\mathcal{P}$. By ensuring that S satisfies C we ensure that $\mathcal{P}$ is met

in S. While framework defines what is guaranteed by satisfaction of C (the proposition $\mathcal{P}$) it says nothing about how we determine whether or not S meets C.

Suppose it is possible to replace part of S in such a way that $\mathcal{E}$ is preserved but C is not necessarily preserved. We can say that a replacement "substitutes" if it ensures that C is satisfied and thereby preserves the truth of $\mathcal{P}$. Substitutability is therefore defined in terms of the proposition P, and a replacement that successfully substitutes for the proposition P may fail to do so for another predicate P'.

As concrete examples of application of this framework, we apply it first to DbC and then consider its application to interactive software.

8.2.1 Application to Design by Contract

DbC has its origins in Hoare Logic [37], a system for formal reasoning about the correctness of algorithms. Hoare Logic is used to establish one of two forms of correctness in an algorithm:

- *Partial Correctness*, which simply requires that if an answer is returned by the algorithm it will be correct.
- *Total Correctness*, which additionally requires that the algorithm terminates.

If we are to base a contract framework on Hoare Logic (as DbC is based), then these forms of correctness are the propositions that can be established. We can therefore think about the semantics of contracts in DbC by equating as follows:

- $\mathcal{T}$ is Hoare Logic.
- $\mathcal{P}$ is a statement of correctness (partial or total) of S.
- $\mathcal{E}$ is a partial proof of $\mathcal{P}$ in the form of a set of statements about S in Hoare Logic.
- C is a contract specified in DbC form.

The notion here is that C relates to a replaceable procedure or function invoked within S; and the proof is an argument in Hoare Logic showing that $\mathcal{P}$ is true of S. Assuming that $\mathcal{E}$ is not affected by the substitution of the procedure or function, then if such replacement meets C we know that $\mathcal{P}$ remains true in S.

It may be that the structure of the proof of $\mathcal{P}$ is composed of many contracts, $C1 + C2 + C3 + \ldots$; and it can be recursively decomposed, so $C1 = C11 + C12 + C13 \ldots$. Exploiting this, of course, is the idea behind the DbC strategy for software development. However, this recursive structure is a property of Hoare Logic and procedural algorithms, and not inherent in the proposed contract framework.

8.2.2 Application to Interactive Software

The key to thinking about contracts for interactive behaviour is to note that we are generally not concerned, or not necessarily concerned, with whether or not a "correctly" computed answer is returned. Consider playing chess against a computer chess game. Both you and the software will expect that a protocol is observed whereby each of you alternates with moves, and that the moves are legal according to the rules of chess. But it doesn't make sense to talk about "correctness" of the interaction, in the sense that this word is used for algorithms.

Consider a player, D, who is completely deterministic in that there is only one move that D will make for a given configuration of the board; and a chess game G against whom D plays. We can ask: what proposition we would like to be true of the system $D + G$, where $+$ represents "plays against", for the purposes of considering what contract the game G should satisfy? Some possibilities are:

a. $D + G$ always executes a valid game of chess, and is deterministic so that there are only two distinct games that ever get played; which one of the two occurs being determined by who has the first move.

b. As above, except that $D + G$ is non-deterministic so that many different games are possible.

c. As above, except that D always wins.

Of these, perhaps the second seems the most useful. In terms of the substitutability of G, the second would mean that, having replaced G with G', you would still be able to play chess, but the games would be different and perhaps better. It is often the case, as it is here, that the protocol is important but the computation is not; so the computation (algorithm) by which the chess game computes its next move from the disposition of pieces on the board does not form part of the contract. This is in contrast with classical DbC, which is entirely concerned with the results of algorithmic computation.

The general implication of our framework is that, if you wish to use contracts as part of a software engineering process, you have to decide on the *proposition* whose truth you want the contract to enforce, and you have to decide on the *reasoning system* that is to be used as the basis for verification. These will then determine the form and nature of the contract you should use. In the context of procedural algorithms, Hoare Logic and the correctness propositions it supports are the obvious choice. In the context of interactive software, because of the complexities of interplay between computation and protocol, there are other choices that can be made. The following sections describe a form of contract based on PM, called a *protocol contract*. In the next chapter, Chapter 9, protocol contracts will be used in the context of a proof of the proposition that a collaboration choreography is realizable; and this represents an example of the framework described above in the domain of interactive software.

8.3 Protocol Contracts

This section describes a kind of behavioural contract called a *protocol contract* defined using PM.

8.3.1 Protocol Contract Formalization

A protocol contract $\mathbb{C}$ defined in a universe $\mathfrak{U}$ is specified as a pair $\langle C, F \rangle$ where:

$$\begin{array}{l} C \text{ is an independent protocol machine defined over } \mathfrak{U}\text{, called the} \\ \textit{contract machine} \text{ of } \mathbb{C} \\ F \text{ is a subset of the alphabet of } C,\ F \subseteq \textit{alphabet}_C\text{, called the } \textit{fully} \\ \textit{constrained actions} \text{ of } \mathbb{C} \end{array} \tag{8.1}$$

An independent protocol machine M satisfies a contract $\mathbb{C}$, written $M \vdash \mathbb{C}$, iff:

$$M \parallel C = M \tag{8.2a}$$

and:

$$\begin{array}{l} \forall\ A \in F :\ \exists \text{ an independent machine } X \text{ such that } M = X \parallel C \text{ and} \\ s \in \textit{asSet}(\mathcal{B}_X) \text{ with } \textit{con}(A \cup s^{\alpha}) \Rightarrow s^{\delta} \neq \textit{refuse} \end{array} \tag{8.2b}$$

The first condition (8.2a) means by (4.45) that $alphabet_C \subseteq alphabet_M$. You would not expect a design (M) to satisfy a contract (C) if its alphabet does not include all the actions in the alphabet of the contract. Note that the composition is heterogeneous, otherwise the equality would not be possible.

The second condition (8.2b) requires that if an action A is *fully constrained* by the contract then contract machine C fully determines whether the action will be refused or not by M. Whereas being allowed by the contract C is always a **necessary** condition for an action $A \in alphabet_C$ to be allowed by a machine $M \vdash \langle C, F \rangle$, if $A \in F$ then being allowed by C is a necessary **and sufficient** condition for A to be allowed by M (assuming that M is robust, so does not crash). We give an example in the next section.

8.3.2 Protocol Contract Example

Consider Figure 8.1. The left hand side specifies a *contract* for the behaviour of a bank account. The right hand side shows the *design* of a bank account, $Account = Account1 \parallel Account2 \parallel Account3$. We now show that the design satisfies the contract.

Account clearly obeys the upper part (state machine) of the contract, as the state diagram *Account1* of the Account is the same as the state diagram, C, of the contract. This means that $Account \parallel C = Account$, thus meeting (8.2a).

We now wish to show that *Account* meets the lower part of the contract, which lists the events that are *fully constrained* by the contract. To do this, consider the *Account* reformulated as shown in Figure 8.2 so that:

$$Account = Account1' \parallel Account1'' \parallel Account2 \parallel Account3$$

In this reformulation, the machine *Account1* has been split into two machines, $Account1'$ and $Account1''$. The first of these specifies the action sequencing on *Open*, *Deposit*, *Withdraw* and *Close*; and the second maintains the *balance* but does not constrain action sequencing.

The machine $Account1'$ is now identical to C, so:

$$Account = C \parallel Account1'' \parallel Account2 \parallel Account3$$

The machine *Account1″* || *Account2* || *Account3* plays the role of *X* for both *Open* and *Deposit* in part (8.2b) of the contract definition above. As none of *Account1″*, *Account2* or *Account3* can ever refuse the actions of type *Open* and *Deposit*, these are fully constrained by the contract. However, as *Account2* can refuse *Close* and *Account3* can refuse *Withdraw*, these are **not** fully constrained by the contract.

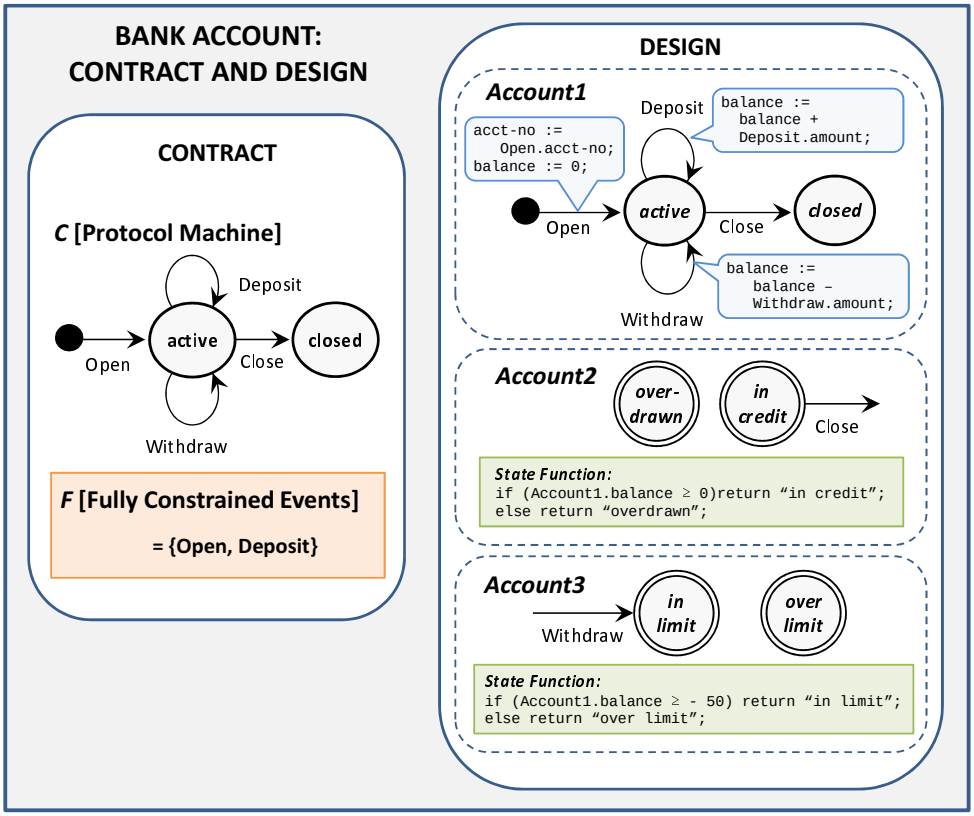


Figure 8.1: Bank Account: Contract and Design

Suppose I have a bank account and I know, because the bank tells me, that the account conforms to the contract shown on the left of Figure 8.1. Suppose also that I know that the account has been opened but not closed. Then I know that, in terms of the contract, it is in the state *active*. From this I can deduce that, for instance:

- I can make a *Deposit* and the account will allow this in its current state. This is because *Deposit* is fully constrained by the contract.
- I **may or may not** be able to *Withdraw* or *Close*, as the contract does not fully constrain these and so does not guarantee that these will be allowed when in the

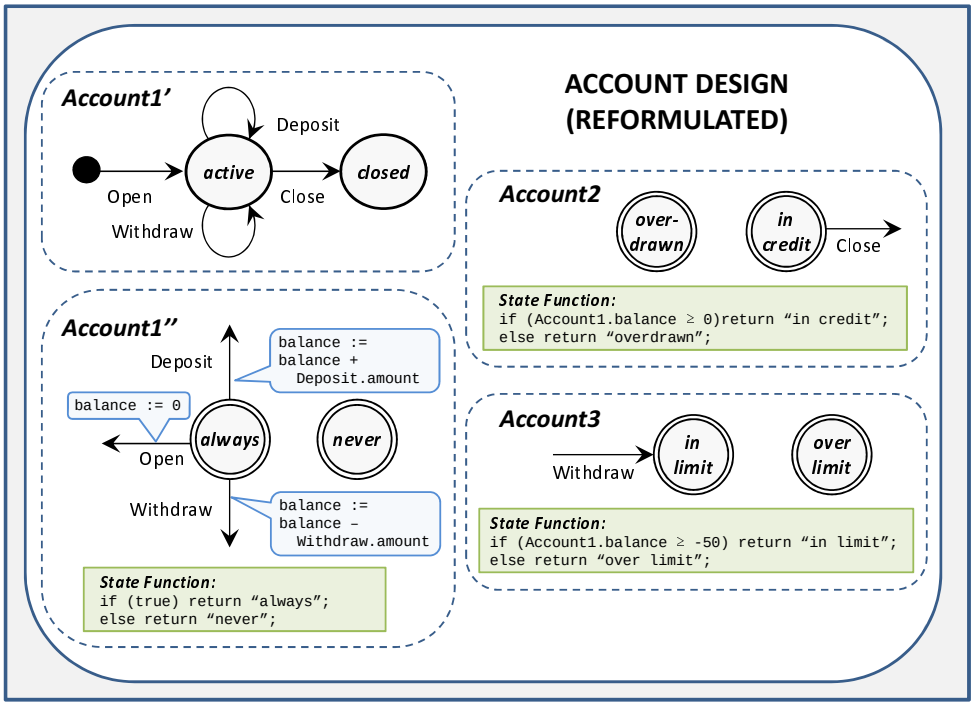


Figure 8.2: Bank Account Reformulated

contract is in the active state. In other words, the bank has rules about when a withdrawal can be made and when an account may be closed, but the contract does not include these rules.

- If I close the account, and the bank permits this operation, I cannot then do any further deposits or withdraws. This is because the sequencing rules of the contract cannot be violated.

In this way, a contract allows determination of what is possible for a machine, based on partial knowledge of its state.

8.3.3 Dependency in Contracts

By definition (8.1) machine part of a protocol contract has to be an independent machine. However, this can be composition that includes dependent machines. Suppose that we want to enhance the contract for the Bank Account to fully constrain *Withdraw*, with the rule that a withdrawal must not take the account overdrawn beyond its limit. The obvious way to do this is to include the machine *Account*3 in the contract, so that

the protocol machine of the contract becomes $C' = C \parallel Account3$. However *Account*3 has a derived state based on *balance*, and the contract needs to know what this is as the contract machine must be independent. This means that *balance* has to be defined within the contract and so we add it as an attribute to the contract, along with the update rules (as shown in the bubbles in *Account1*) that maintain its value. This means that we now have $C' = Account1 \parallel Account3$.

In general, we allow the use of data and dependent machines in the definition of a contract provided that they are in composition with other machines so that the contract machine as a whole is independent. This requires that any inter-machine references used in the contract are fully resolved within the contract.

8.4 Behavioural Conformance

The ability to specify the set of fully constrained actions in a protocol contract gives the power to dictate the degree of behavioural conformance required. Suppose you have a machine M with alphabet $alphabet_M$. Suppose that you have another machine $M^\dagger$ and you want to ensure that this machine has *invocable compatibility* (in the sense used by Ebert and Engels in [21]) with M. Then you require that $M^\dagger$ conforms to the contract $\langle M, alphabet_M \rangle$. This will require that if M allowed a particular trace of invocations (where each invocation is an action in $alphabet_M$) then $M^\dagger$ must allow it too. However if you only want *observable compatibility* then you only require that $M^\dagger$ conforms to $\langle M, \emptyset \rangle$. This only requires that $M^\dagger = M \parallel M^\dagger$ so any trace of invocations to $M^\dagger$ if restricted to actions of $alphabet_M$ will also be a trace of invocations to M.

If you choose a contract $\langle M, \mathbb{A}' \rangle$ where $\mathbb{A}'$ is a non-empty proper subset of $alphabet_M$ then you have a form of conformance that is between *invocable compatibility* and *observable compatibility*. In this way protocol contracts provide a generalised notion of behavioural conformance of which *invocable compatibility* and *observable compatibility* are extreme forms.

8.5 Composition and Decomposition of Contracts

Protocol contracts can be composed and sometimes decomposed, as we describe below. Composition of contracts is important in the context of choreography realizability. Decomposition is addressed too, for sake of completeness.

8.5.1 Composition

If $\mathbb{C}1 = \langle C1,\ F1 \rangle$ and $\mathbb{C}2 = \langle C2,\ F2 \rangle$ where $C1$ and $C2$ are both defined over a universe $\mathfrak{U}$, then their composition is defined as follows:

$$\mathbb{C}1 \oplus \mathbb{C}2 = \langle C1 \parallel C2,\ F1 \cup F2 \rangle$$

Theorem B.14 shows that if a machine P satisfies the two contracts $\mathbb{C}1$ and $\mathbb{C}2$ then P also satisfies $\mathbb{C}1 \oplus \mathbb{C}2$.

8.5.2 Decomposition

Suppose that $\mathbb{C} = \langle C,\ F \rangle$ and that $C = C1 \parallel C2$ where $C1$ and $C2$ and both defined over $\mathfrak{U}$. It is reasonable to ask whether $\mathbb{C}$ can be decomposed into two contracts, $\mathbb{C}1$ and $\mathbb{C}2$, the former using $C1$ and the latter using $C2$, in such a way that:

$$P \vdash \mathbb{C} \Rightarrow (P \vdash \mathbb{C}1) \wedge (P \vdash \mathbb{C}2)$$

This is possible provided that:

$$F \cap \mathit{alphabet}_{C1} \cap \mathit{alphabet}_{C2} = \emptyset \tag{8.3}$$

and the decomposition, giving $\mathbb{C} = \mathbb{C}1 \oplus \mathbb{C}2$, is:

$$\mathbb{C}1 = \langle C1, F \cap \mathit{alphabet}_{C1} \rangle \quad \text{and} \quad \mathbb{C}2 = \langle C2, F \cap \mathit{alphabet}_{C2} \rangle$$

The reason for the stricture (8.3) is as follows. Suppose that $P \vdash \mathbb{C}$ and that $A \in (F \cap \mathit{alphabet}_{C1} \cap \mathit{alphabet}_{C2})$. While the pattern of occurrences of A in P is fully described (i.e., fully constrained) by $C1 \parallel C2$, it may not be fully described by either $C1$ or $C2$ alone because it depends on the way the two processes synchronize on A. In this case A cannot belong to the set of the events fully constrained by a contract defined in terms of either $C1$ or $C2$ alone, so the decomposition fails.

Theorem B.14 shows that, if (8.3) holds, then a machine P satisfies $\mathbb{C}1 \oplus \mathbb{C}2$ then P also satisfies the decomposed parts $\mathbb{C}1$ and $\mathbb{C}2$.

Finally, combining the results for composition and decomposition we note that:

If $F1 \cap F2 = \emptyset$ then $(P \vdash \mathbb{C}1 \oplus \mathbb{C}2) \Leftrightarrow (P \vdash \mathbb{C}1 \wedge P \vdash \mathbb{C}2)$

8.6 Conclusions and Further Work

This chapter proposes a framework for thinking about contracts in software, by seeing satisfaction of the contract as a requirement in the construction of a proof of some proposition, so that a substitution that preserves satisfaction of the contract preserves truth of the proposition. We go on to define *protocol contracts*, a form of behavioural contract based on PM, which we use in the next chapter in the context of choreography theory, where the proposition in question is that enactment of a collaboration is bound to follow its choreography.

Protocol contracts are expressed as protocol machines and rely on the semantics of PM, in particular synchronous composition and the treatment of data. This enables contracts that require conformance on data, in terms of required values for both internal storage attributes and message fields, as well as behaviour, in terms of the required message sequencing protocol. In addition, protocol contracts include a mechanism for specifying for a given protocol event (a message send or receive), whether the strictures it defines for that event represent necessary and sufficient or merely sufficient conditions for the event to occur. Here it differs from the more usual approach where this distinction is made at the meta-level, as part of the semantics of contract protocol conformance, with the implication that you have to make a choice on the semantics.

Over the last few years a number of approaches have been proposed for including a notion of contracts in multiparty collaborations, some of this related to work on choreography (the subject of the next chapter). Some notable examples are:

- In [7] Bartoletti and Zunino propose an abstract theory of contracts which they call "Contract-Oriented Computing". They define a logic called *Propositional Contract Logic* (PCL), that extends intuitionistic logic to capture and reason about contracts; and a calculus for contracting processes that embeds PCL. The operational

semantics of the calculus enable reasoning about whether contracting processes will honour their declared contracts.

- In [14], Bocchi et al. propose a generalisation of the notion of DbC to enable effective specification and verification of distributed multiparty protocols. This proposal allows choreography definitions to be decorated with *assertions*, predicates on data values, and provides a formal mechanism to project these global assertions as obligations on the participants. A proof system makes it possible to establish that the collaboration will respect the global assertions provided that all participants meet their respective local obligations.

- In [6] the authors of the above papers propose a method of reasoning about dishonesty in collaborations, and give a sound criterion for detecting when a participant is honest, even in the presence of dishonest participants. In [49] Lange and Scalas apply this idea to choreographed collaborations, where the choreography is synthesized by composing the contracts advertised by the participants.

There are a number of questions concerning how the work in this thesis relates to these ideas that are worth exploring:

- Can the contract framework of Chapter 8 be used to capture these other models, and if so could this add to the power or utility of contract-oriented computing?

- How does this relate to the ideas set out in Section 9.11, concerning the method used for projection of participant behaviours from a choreography, relate to other work on the ability to recognise "dishonest" behaviour in a collaboration?

Chapter 9

Choreography

This chapter describes the use of PM in the context of *choreographed collaborations*. Here PM provides a medium for describing the *choreography* itself, a depiction of the possible orderings of message exchanges between the participants in the collaboration; and the behavioural *contracts* that describe how each participant should behave to collaborate successfully. We show how the use of compositional descriptions brings new techniques and power to the task of establishing that a choreography is *realizable*, meaning that the collaboration as enacted matches the choreography that was the basis for its design.

9.1 The Challenge

With networks now providing universal connectivity across organizational boundaries, business-to-business e-commerce and cross-organizational workflow applications are increasingly common. In such applications, two or more organizations arrange for their systems to collaborate in a shared business process, communicating by message exchange across a network infrastructure. The interaction between the systems engaged in the services can become complex, involving multiple message types and many different possible patterns of message exchange.

Current software engineering practice does not offer a generally accepted approach to the design of extended multiparty collaborations and consequently industry practice generally resorts to two expedients:

- The use of multiple simple collaborations, each involving only a pair of participants and invoking manual intervention when the collaboration enters a state that, because of the fragmentation of the process, the software is not designed to handle. While this ensures that the complexity of the software is bounded to a level that can be validated using conventional software engineering techniques, in general it results in business processes that are slower, less scalable and more expensive than they need be.

- The use of "packaged" solutions that require business processes to adhere to predefined proprietary templates. The templates are provided by specialist vendors who are able to invest in evolving and validating standard solutions over time. This works for some well standardized applications, but sometimes requires that processes are forced to fit and, more importantly, does not address cases where competitive advantage stems from innovative custom solutions.

This attests to the value in obtaining a general approach to designing extended collaborations, simple and practical enough to be used routinely in the context of custom solution design.

Designing services that can engage successfully in extended collaborations is a challenge. Unlike a program or process that follows a single procedural description, a collaboration involves the concurrent interaction of independent participants and is subject to various unpredictable factors at execution time that can affect its behaviour, progress and outcome; in particular:

- Differences in the relative speed or scheduling of the hardware/software at each participant.

- Variations in latency of message transfer by asynchronous messaging systems.

- Execution choices made by the infrastructure, independently of the application, when handling synchronization across distributed components[1].

Considered as a whole, the behaviour of a collaboration is the emergent behaviour of the interaction between its participants and is non-deterministic in nature. This non-determinism renders conventional testing techniques ineffective, as it is very hard or

[1]Such as using a 2-phase commit mechanism for synchronous message exchange.

impossible to design a test strategy that provides adequate coverage of the execution possibilities, and it is not generally possible to repeat runs to recreate errors. As Owicki and Lamport observe [67]:

> *There is a rather large body of sad experience to indicate that a concurrent program can withstand very careful scrutiny without revealing its errors. The only way we can be sure that a concurrent program does what we think it does is to prove rigorously that it does it.*
>
> **– Susan Owicki and Leslie Lamport [67]**

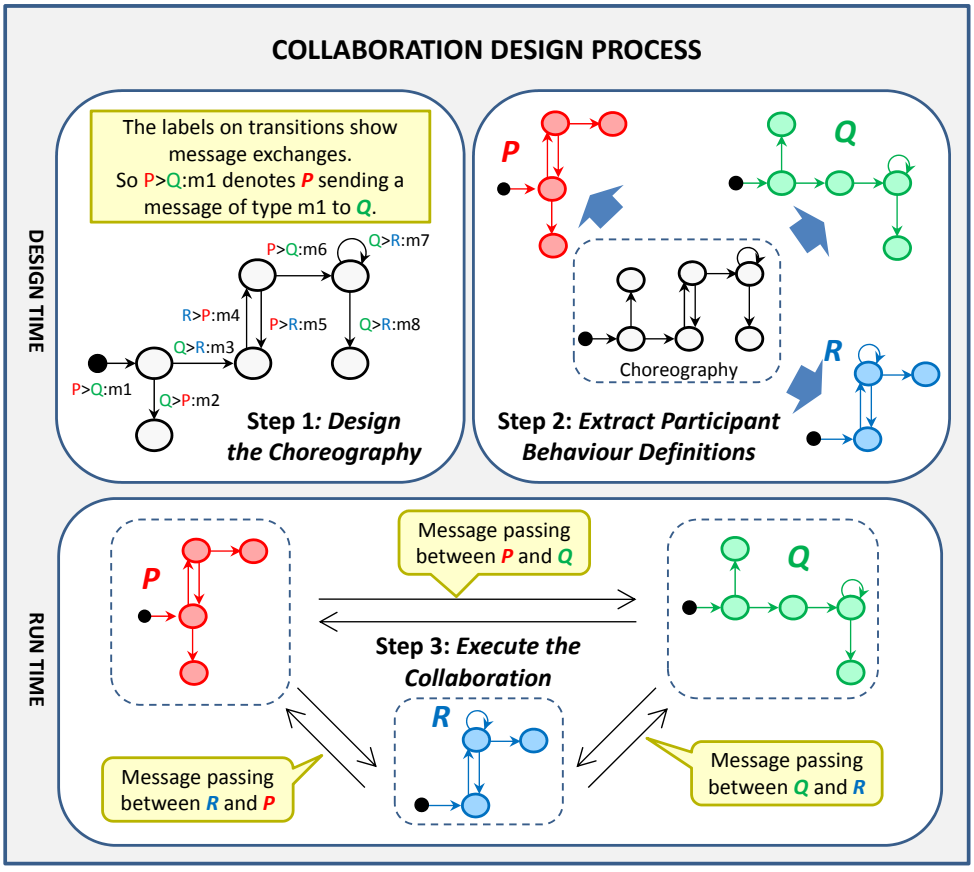


Figure 9.1: Collaboration Design Process

The emerging approach to building extended service collaborations that are provably reliable is to base the design on a *global contract* that specifies the allowable patterns of message exchange between the participants from a global perspective. Such a

global contract is known as a *choreography*. Building a collaboration based on a choreography has three steps as depicted in Figure 9.1:

Step 1. At design-time, a choreography is developed which describes all meaningful sequences of message exchange between the participants.

Step 2. Using some kind of mechanical process, a specification of the behaviour of each participant is extracted from the overall choreography[2].

Step 3. At run-time the participants interact, each behaving according to its own behaviour specification.

At run-time (step 3) the participants behave independently. Each is free to send and receive messages according to its own behavioural specification, without any central orchestrating or controlling entity to enforce conformance to the original choreography.

This procedure guarantees a degree of compatibility between the designs of the behaviours of the participants but, unfortunately, this is not enough by itself to guarantee that the behaviour of the collaboration at step 3 does not depart from the choreography. It may be that the collaboration deadlocks leaving messages unprocessed, or allows patterns of message exchange not envisaged in the original choreography definition resulting in combinations of states in the participants that were not intended and have no meaning. As the choreography specifies the meaningful sequences of exchange it is clearly important that departure from it is not possible, as such departure represents entering an unintended and meaningless state. The behavioural rules embedded in the participants must therefore be strong enough to prevent such departure and this, in turn, requires that the choreography conforms to certain structural conditions. The problem of determining general conditions on the form of a choreography which are sufficient to guarantee that the interaction of extracted behaviours is **bound to adhere** to the choreography is known as *the realizability problem*.

Investigations into choreography realizability address the question: Given a particular communication model (e.g., unbounded FIFO queues in each direction between each pair of participants), and a process for extracting participant behaviours from a

[2]In the jargon of choreography theory this is called "end-point projection".

choreography, what are sufficient conditions on the form of the choreography to guarantee that the collaboration will always follow the choreography?

As discussed by Kazhamiakin and Pistore [43], the rules for realizability depend on the form of *communication model* used for the collaboration, in particular whether communication between participants is synchronous or asynchronous. Our main interest is in collaborations that use *asynchronous* communication. This means that a sender transmits a message on the assumption that the receiver is able to receive it, and the message may take time to transit so that the receive happens at a later time. With this form of communication, the sender and receiver do not engage in a send/receive simultaneously, which would require some kind of transactional infrastructure across the participants. Where the participants in a collaboration are geographically distributed a transactional infrastructure is complex and difficult to operate, so an asynchronous messaging infrastructure is more practical.

We will explore **asynchronous exchange**, where we assume a communication model that:

Provides an unlimited FIFO queue in each direction between each pair of participants. (9.1a)

Has finite network latency, so a message sent will arrive at its recipient in finite time. (9.1b)

A participant cannot be blocked on a send. (9.1c)

9.2 Modelling Choreography

We will be using PM to model both the choreography (as constructed in Step 1 in Figure 9.1) and participant processes (as extracted from the choreography in Step 2). More accurately, we define *protocol contracts* for participant processes.

This means that message sends and receives will be represented as actions. In general, PM allows an action in a protocol machine to be a *combination of input and output*. However for the purposes of choreographed collaborations we will assume that all actions are **either wholly output (send) or wholly input (receive)** as this is required in asynchronous message exchange.

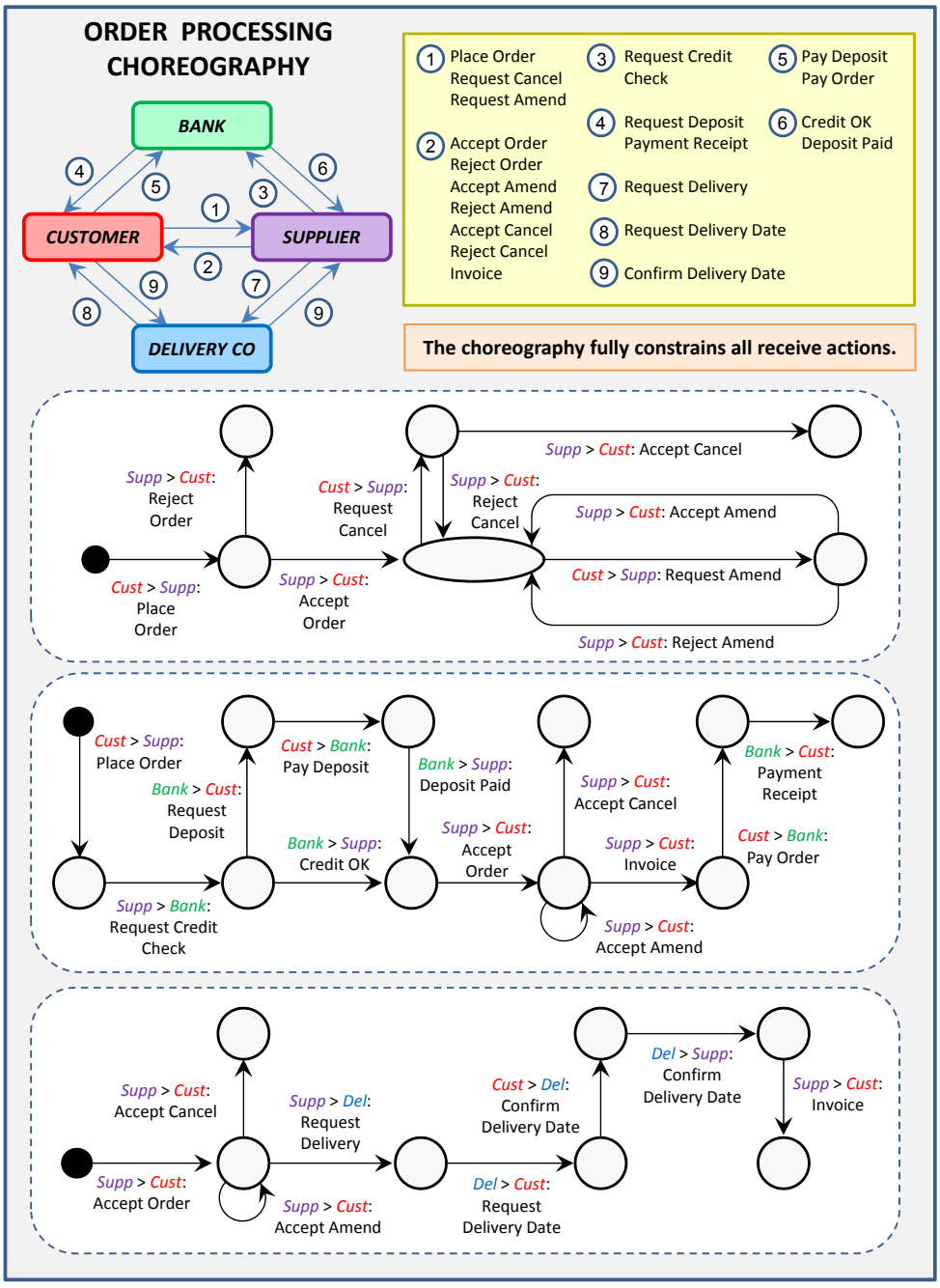


Figure 9.2: Order Processing Example: Choreography

We represent a choreography using the symbol $\mathfrak{C}$ which is a pair $\langle \mathbf{C}, \mathbf{F} \rangle$ where:

- **C** is an independent protocol machine that defines the possible global sequencing of message exchanges in the collaboration, and

- **F** is a statement that specifies which actions in the participants are fully constrained by the choreography.

We assume that the collaboration involves a fixed finite set of participants, $\mathfrak{P}$. These are independent and isolated agents that communicate only by message passing.

Figure 9.2 shows a choreography for collaboration between four participants: a *Customer* (*Cust*) a *Supplier* (*Supp*), a *Bank* (*Bank*) and a *Delivery Company* (*Del*) engaged in placing and processing an order. The choreography is expressed as three composed machines whose actions represent *message exchanges* between the participants. The transitions are labelled with the sender and receiver and message type being exchanged. So in Figure 9.2 a label:

```
Cust>Supp:Place Order
```
(9.2)

represents the exchange of a *Place Order* message sent by participant *Cust* (the Customer) to participant *Supp* (the Supplier). The composition obeys the usual rule, so that an exchange that appears in more than one machine can only take place when allowed by all the machines in which it appears.

9.3 Choreography and Participant Universes

Like all protocol machines both the **C** and the participant processes are defined over *universes* of possible data observations as defined in Section 4.1.5 on page 40. To support its role as the basis for the design of a collaboration, the universe of a choreography has a structure that relates to the universes of the participants. This section describes the structure of the universes of a choreography and of its participants.

We will use:

- $\mathfrak{P}$ to represent the set of participants involved in the collaboration
- P, Q, R to refer to individual participants in $\mathfrak{P}$; and $P_j,\ j = 1 \ldots n$, to index over all participants in $\mathfrak{P}$
- $[\mathfrak{P}]$ to denote $\{\ \{P\} \mid P \in \mathfrak{P}\ \}$
- $[\mathfrak{P}]^2$ to denote $\{\ \{P, Q\} \mid P, Q \in \mathfrak{P} \wedge P \neq Q\ \}$; and $\mathcal{T}$ to refer to an element of $[\mathfrak{P}]^2$
- $\natural$, $\natural 1$, $\natural 2$ to refer to members of $[\mathfrak{P}] \cup [\mathfrak{P}]^2$

9.3.1 Choreography Universe

A choreography is defined over a universe:

$$\mathfrak{U}_{\mathfrak{C}} = \langle\, \mathcal{U}_{\mathfrak{C}},\ \mathcal{A}_{\mathfrak{C}},\ \mathcal{D}_{\mathfrak{C}},\ \mathbb{V}_{\mathfrak{C}},\ \mathit{fixes}_{\mathfrak{C}} \,\rangle$$

and this universe is partitioned as follows:

- $\mathcal{A}_{\mathfrak{C}}$ is partitioned by participant, with the subset for participant P denoted by ${\mathcal{A}_{\mathfrak{C}}}^{\{P\}}$. This partitioning represents which participant is able to calculate (derive) the value of the symbol and therefore use it as part of an output message. It does not affect which participant can have the symbol as part of an input message.
- $\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}}$ is comprises two disjoint parts:
  - One part is partitioned by participant, with the subset for $P \in \mathfrak{P}$ denoted by $(\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\{P\}}$. This part contains symbols that are private to each participant.
  - The other part is partitioned by unordered pair of participants, with the subset for $\mathcal{T} \in [\mathfrak{P}]^2$ denoted by $(\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\mathcal{T}}$. This part contains symbols whose value is synchronised between two participants.

We require that the partitioning obeys:

$$\mathcal{A}_{\mathfrak{C}} = \bigcup_{\{P\} \in [\mathfrak{P}]} {\mathcal{A}_{\mathfrak{C}}}^{\{P\}} \tag{9.3a}$$

$$P, Q \in \mathfrak{P} \text{ with } P \neq Q \quad \Rightarrow \quad {\mathcal{A}_{\mathfrak{C}}}^{\{P\}} \cap {\mathcal{A}_{\mathfrak{C}}}^{\{Q\}} = \emptyset \tag{9.3b}$$

$$\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}} \quad = \bigcup_{\natural \in [\mathfrak{P}] \cup [\mathfrak{P}]^2} (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\natural} \tag{9.3c}$$

$$\natural 1, \natural 2 \in [\mathfrak{P}] \cup [\mathfrak{P}]^2 \text{ with } \natural 1 \neq \natural 2 \quad \Rightarrow \quad (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\natural 1} \cap (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\natural 2} = \emptyset \tag{9.3d}$$

Based on the partitioning we can construct the subset of $\mathcal{U}_{\mathfrak{C}}$ related to a participant P as:

$${\mathcal{U}_{\mathfrak{C}}}^{P} = {\mathcal{A}_{\mathfrak{C}}}^{\{P\}} \cup \bigcup_{\natural \in [\mathfrak{P}] \cup [\mathfrak{P}]^2 \,\wedge\, P \in \natural} (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\natural}$$

Notice that the subsets for different participants are not disjoint, so if $P \neq Q$:

$${\mathcal{U}_{\mathfrak{C}}}^{P} \cap {\mathcal{U}_{\mathfrak{C}}}^{Q} = (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\{P,Q\}}$$

Finally we require that:

$$\mathcal{A}_{\mathfrak{C}} \subseteq \mathcal{D}_{\mathfrak{C}} \tag{9.4a}$$

$$\forall\, P \in \mathfrak{P} \;:\quad closed(restr(\mathbb{V}_{\mathfrak{C}},\, \mathcal{U}_{\mathfrak{C}}{}^{P})) \tag{9.4b}$$

$$\forall\, \mathcal{T} \in [\mathfrak{P}]^2 \;:\quad closed(restr(\mathbb{V}_{\mathfrak{C}},\, (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\mathcal{T}})) \tag{9.4c}$$

These are to be interpreted as follows:

- The first (9.4a) says all actions of a choreography are output per (4.11), as all action symbols are derived.[3]
- The second (9.4b) requires that a derived symbol that belongs to a given participant can be computed entirely within the subset of the universe related to the participant. In particular this is true of derived action symbols, these being the symbols that are sent by that participant.
- The third requires that a partition of the non-action part of the universe indexed by a pair of participants cannot use symbols of another partition of the non-action part of the universe for calculation of a derived symbol. Suppose that the partition indexed by $\{P, Q\}$ uses a symbol in the partition indexed by $\natural \neq \{P, Q\}$. If $\{P, Q\} \cap \natural = \{P\}$ then the derivation can only take place in P and not in Q, as would be required for the symbol to be shared between the two.

9.3.2 Choreography Messages

We assume that the choreography universe $\mathcal{U}_{\mathfrak{C}}$ contains a set of special symbols $\mathfrak{X}_{\mathfrak{C}} \subseteq \mathcal{A}_{\mathfrak{C}}$ used to encode the labels that appear on the transitions in a choreography, such as labels on the transitions in Figure 9.2. This set is defined as:

$$\mathfrak{X}_{\mathfrak{C}} = \{\, \mathtt{exchange}_{\mathfrak{C}}{}^{P_i} \mid P_i \in \mathfrak{P} \,\} \quad \text{with} \quad \mathtt{exchange}_{\mathfrak{C}}{}^{P_i} \in \mathcal{A}_{\mathfrak{C}}{}^{\{P_i\}}$$

with valid values for $\mathtt{exchange}_{\mathfrak{C}}{}^{P} \in \mathfrak{X}_{\mathfrak{C}}$ specified by:

$$\begin{aligned} &val(restr(\mathbb{V}_{\mathfrak{C}}, \mathtt{exchange}_{\mathfrak{C}}{}^{P})) = \\ &\quad \{\, P > P_i : \mathtt{m} \mid P_i \in \mathfrak{P} \;\wedge\; P_i \neq P \;\wedge\; \mathtt{m} \in \mathfrak{M}_{\mathfrak{C}} \,\} \cup \{null\} \end{aligned} \tag{9.5}$$

[3] This is a curious statement. It reflects the fact that the choreography is, in a sense, a merger of all participants and must be able to send all messages.

where $\mathfrak{M}_{\mathfrak{C}}$ is the set of message types in the choreography, for instance `Place Order`, `Confirm Order`; and *null* is a special value used where there is no transition label to be encoded.

The symbols in $\mathfrak{X}_{\mathfrak{C}}$ are referred to as the *exchange symbols* of the choreography. The example (9.2) would be encoded as an action of the choreography as:[4]

$$\{\ (\texttt{exchange}_{\mathfrak{C}}{}^{Cust}\ \texttt{= Cust>Supp:Place Order})\ \}$$

Note that there is no implication that an action in a choreography is limited to a single observation using an exchange symbol: further fields may be specified if it is required to further constrain the message content. For instance we could encode the above action as:

$$\{\ (\texttt{exchange}_{\mathfrak{C}}{}^{Cust}\ \texttt{= Cust>Supp:Place Order}),\ (\texttt{product = Widget})\ \}$$

which would specify that only orders for Widgets are handled. Any attempt by the customer to place an order for a different product would then be a violation of its projected protocol contract, and not be allowed by the choreography.

Fn: *sender, receiver.* We define the following functions on the actions of a choreography universe $\mathfrak{U}_{\mathfrak{C}}$:

- $\mathit{sender} :: \mathit{restr}(\mathfrak{U}_{\mathfrak{C}}, \mathcal{A}_{\mathfrak{C}}) \to \mathfrak{P} \cup \{\mathit{null}\}$
- $\mathit{receiver} :: \mathit{restr}(\mathfrak{U}_{\mathfrak{C}}, \mathcal{A}_{\mathfrak{C}}) \to \mathfrak{P} \cup \{\mathit{null}\}$

with $\mathit{null} \notin \mathfrak{P}$, as:

$$\left.\begin{array}{l} \mathit{sender}(A) = P_j \\ \mathit{receiver}(A) = P_k \end{array}\right\} \quad \begin{array}{l} \text{if } \mathit{val}(\mathit{restr}(A, \mathfrak{X}_{\mathfrak{C}})) \backslash \{\mathit{null}\} = \{P_j \texttt{ > } P_k \texttt{:m}\} \\ \quad \text{where } P_j, P_k \in \mathfrak{P} \text{ and } m \in \mathfrak{M}_{\mathfrak{C}} \end{array}$$

$$\left.\begin{array}{l} \mathit{sender}(A) = \mathit{null} \\ \mathit{receiver}(A) = \mathit{null} \end{array}\right\} \quad \text{otherwise}$$

which gives a non-null value to $\mathit{sender}(A)$ and $\mathit{receiver}(A)$ provided that A contains a single well-formed and non-null exchange label.

[4]This action encoding is simplified, as shortly we will see that actions always contain a full set of exchange symbols, with those not belonging to the sender having value *null*. The null-valued symbols have been omitted here.

Well-formedness conditions for a choreography machine **C** defined over the choreography universe $\mathcal{U}_{\mathfrak{C}}$ are required as follows.

$\forall\, t \in \mathit{stems}(\mathcal{B}_{\mathbf{C}})$ with $t^{\tau} \Subset \mathbb{V}_{\mathfrak{C}}$:

$$\mathit{length}(t) > 1 \quad \Rightarrow \quad \mathit{sender}(t^{\alpha}) \neq \mathit{null} \tag{9.7a}$$

$$\mathit{sender}(t^{\alpha}) = P \quad \Rightarrow \quad \mathit{symb}(t^{\alpha}) \backslash \mathfrak{X}_{\mathfrak{C}} \subseteq \mathcal{A}_{\mathfrak{C}}{}^{\{P\}} \ \wedge \ \mathfrak{X}_{\mathfrak{C}} \subseteq \mathit{symb}(t^{\alpha}) \tag{9.7b}$$

The first (9.7a) requires that every step has a sender. The second (9.7b) requires that the action of a step only uses fields that belong to the sender; and that the action also contains null exchange labels for all participants other than the sender. This last condition ensures that, even when choreography machines are composed, every step in the composition only has a single sender; so that both (9.7a) and (9.7b) are preserved by homogeneous composition of choreographies.

9.3.3 Participant Universe

Figure 9.3 shows a protocol contract for *Customer*, one of the participants, expressed as three composed machines. The other participants will have similar contracts. The actions of a contract represent *message sends* and *message receives*. The transitions are labelled:

```
!>Supp:Place Order
?<Bank:Request Deposit
```
(9.8)

representing a send (to the Supplier) and a receive (from the Bank) respectively. The composition obeys the usual rule, so that a send or receive that appears in more than one machine can only take place when allowed by all the machines in which it appears. The composition of the machines is to be interpreted as the contract machine of a protocol contract for *Customer*. The other component of the contract, the set of fully constrained actions, is shown in the lower part of Figure 9.3 as F_{Cust}. This implements the requirement that the choreography fully constrains all receive actions.

Each participant has its own universe, so $P \in \mathfrak{P}$ has universe:

$$\mathfrak{U}_P = \langle\, \mathcal{U}_P,\ \mathcal{A}_P,\ \mathcal{D}_P,\ \mathbb{V}_P,\ \mathit{fixes}_P \,\rangle$$

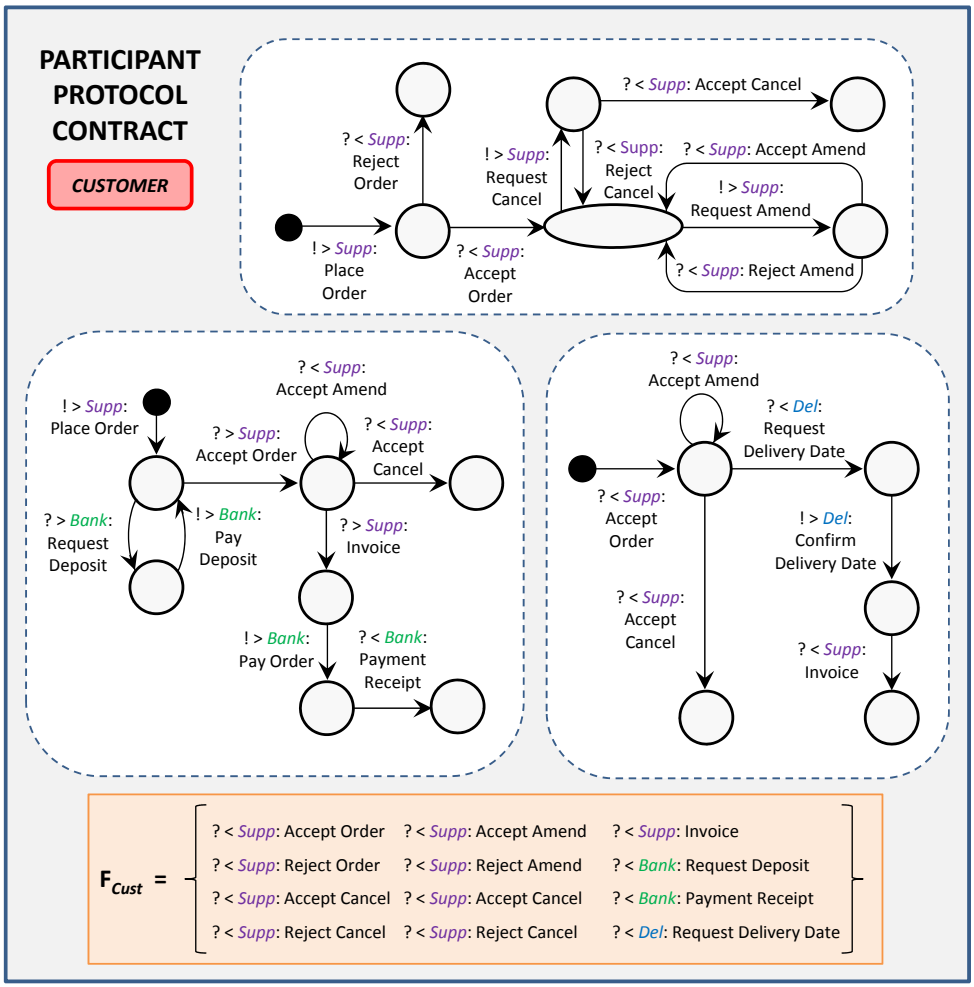


Figure 9.3: Order Processing Example: Customer Contract

The participant universes must obey the following well-formedness constraints:

$$\mathcal{A}_P \cap \mathcal{U}_{\mathfrak{C}} = \mathcal{A}_{\mathfrak{C}} \tag{9.9a}$$

$$(\mathcal{U}_P \backslash \mathcal{A}_P) \cap \mathcal{U}_{\mathfrak{C}} = \bigcup_{\natural \in [\mathfrak{P}] \cup [\mathfrak{P}]^2 \wedge P \in \natural} (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\natural} \tag{9.9b}$$

$$\mathcal{D}_P \cap \mathcal{U}_{\mathfrak{C}} = \mathcal{D}_{\mathfrak{C}} \cap {\mathcal{U}_{\mathfrak{C}}}^{P} \tag{9.9c}$$

$$restr(\mathbb{V}_{\mathfrak{C}}, \mathcal{A}_{\mathfrak{C}} \cup {\mathcal{U}_{\mathfrak{C}}}^{P}) \quad \Subset \quad restr(\mathbb{V}_P, \mathcal{U}_{\mathfrak{C}}) \tag{9.9d}$$

where:

- (9.9a) requires that every participant can handle **all** the actions of the choreography.
- (9.9b) requires that the use of the non-action symbols of a participant universe conforms to the partitioning of the choreography universe.
- (9.9c) requires that the subset of derived symbols of a participant universe that belongs to the choreography matches the set of derived symbols of the choreography, partitioned to that participant. Note that this together with (9.7b) means that if a participant is sender for a given message, the symbols of that message are derived in that participant's universe.
- (9.9d) requires that any set of observations that is valid in the universe of the choreography, restricted to the symbols used by the participant, is also valid in the universe of a participant, restricted to the symbols of the choreography. In particular, this requires that any set of observations using only the symbols of $\mathcal{A}_{\mathfrak{C}} \cup {\mathcal{U}_{\mathfrak{C}}}^{P}$ that is valid in $\mathbb{V}_{\mathfrak{C}}$ is also valid in $\mathbb{V}_P$.

In formulating these constraints we are taking a notational short-cut by equating symbols in the participant universes with symbols in the global universe. For instance, (9.9b) says that a symbol $\mathtt{x}$ in $(\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\{P,Q\}}$ belongs to both $\mathcal{U}_P$ and $\mathcal{U}_Q$. Strictly speaking these are not the same symbol as they are *projections* to different participant universes and should be differentiated, perhaps as $\mathtt{x}_P$ and $\mathtt{x}_Q$. Later, in Section 9.7.2, we justify this short-cut by showing that values projected to different participants from a single symbol are synchronized during enactment of a collaboration.

9.4 Semantics of Projection

In common with other mainstream choreography methods we use *end-point projection*, whereby the behaviour of the participants is extracted in a mechanical fashion from the choreography definition. Thus in Figure 9.3 the three machines that describe the Customer behaviour are projected from the three machines in the choreography. This forms Step 2 of the collaboration design process shown in Figure 9.1 on page 152. In this section we formalize the notion of *projection*.

We define projection for a single independent well-formed choreography machine $\mathbf{C}$ defined over a universe $\mathfrak{U}_{\mathfrak{C}}$. The projection of process has two stages:

- *Global projection* which projects to a process defined in the universe of the choreography, $\mathcal{U}_{\mathfrak{C}}$.
- *Local projection* which casts the global projection to the universe of a participant.

9.4.1 Global Projection

Given a participant $P \in \mathfrak{P}$ we define a boolean *projection filter* $\mathcal{F}_P$ on the actions of a choreography or a participant as:

$$\mathcal{F}_P(A) \Leftrightarrow P \in \{\ sender(A), receiver(A)\ \}$$

Given a choreography machine $\mathbf{C}$ then a *global projection* of $\mathbf{C}$ to P, denoted $\mathbf{C}\Downarrow_P^g$ is any independent protocol machine defined over $\mathfrak{U}_{\mathfrak{C}}$ satisfying:

$$alphabet_{\mathbf{C}\Downarrow_P^g} = \{\ A \mid A \in alphabet_{\mathbf{C}} \ \wedge \ \mathcal{F}_P(A)\ \} \tag{9.10a}$$

$$\mathbf{C}\Downarrow_P^g \parallel \mathbf{C} = \mathbf{C} \tag{9.10b}$$

$$\Omega_{\mathbf{C}\Downarrow_P^g} = \Omega_{\mathbf{C}} \cap {\mathcal{U}_{\mathfrak{C}}}^P \tag{9.10c}$$

where:

- (9.10a) means that every action preserved by the filter is in the alphabet of $\mathbf{C}\Downarrow_P^g$ and all other actions of the choreography are ignored by $\mathbf{C}\Downarrow_P^g$.
- (9.10b) requires that $\mathbf{C}\Downarrow_P^g$ contains all completions that are possible in $\mathbf{C}$.
- (9.10c) restricts the owned data by $\mathbf{C}\Downarrow_P^g$ to the portion of the choreography universe belonging to P.

This is called the *global* (signified by the superscript ^{g}) projection as it is still defined over the *choreography* universe, $\mathfrak{U}_{\mathfrak{C}}$.

In general there is no guarantee that a machine satisfying (9.10) exists or that it is unique. If a projection $\mathbf{C}\Downarrow_P^g$ does exist, then $\mathbf{C}$ is the said to be *projectable* to P.

A consequence of (9.10) is that a message can only update the portion of the owned data $\Omega_{\mathbf{C}}$ of the choreography that "belongs" to either one of the participants involved in the exchange, or is shared by both. For instance suppose that P and Q are sender and receiver of a message instance m and that this exchange leads to an update of the value a symbol $x \in \Omega_R$ with $R \notin \{ P, Q \}$. But by (9.10b) we require that:

$$\mathbf{C}\Downarrow_R^g \parallel \mathbf{C} = \mathbf{C} \tag{9.11}$$

and as the symbol x belongs to the owned data of both machines on the left hand side of (9.11) it must be updated by both. However, by (9.10a), $\mathbf{C}\Downarrow_R^g$ ignores m so cannot update x. This means that m can only update symbols that belong to:

$$(\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\{P\}} \cup (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\{Q\}} \cup (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\{P,Q\}}$$

It is for this reason that the choreography universe is partitioned by $[\mathfrak{P}] \cup [\mathfrak{P}]^2$ as described in Section 9.3.1.

9.4.2 Local Projection

Local projection casts a global projection to the universe of a participant. The *local projection* of $\mathbf{C}\Downarrow_P^g$ to P is denoted by $\mathbf{C}\Downarrow_P$. It is created by adding local actions (as ignored actions, per (4.42)) to ensure that $\mathbf{C}\Downarrow_P$ defines behaviour exhaustively (obeys (4.10c)) for all possible actions in $\mathfrak{U}_P$.

The resultant machine is defined over $\mathfrak{U}_P$. This is possible because:

- By (9.9b) and (9.10c), the offered data of the global projection also belongs to the participant universe.
- All actions of the choreography are retained in the projection, and local actions in the participant that are not part of the choreography are added as ignored actions to the projection according to the definition in Section 4.5.1. This gives exhaustive treatment of the actions in $\mathfrak{U}_P$.
- As projections are independent machines, exhaustive treatment of perceived data is trivially given.

9.5 Topological Projection

The reasoning on realizability, set out in the rest of this chapter, is conducted in the medium of *graphical (state machine) representation* of protocol machines, rather than the formal denotation of completions. This reflects practice and practicality, as this is the medium in which the work of authoring and analyzing choreographies is carried out. We will use topological reduction, as described in Chapter 6.4, as the means of projecting independent choreography machines represented in topological (LTS) form.

9.5.1 Projection Construction

Given **C** expressed as a single independent machine with a topological representation, and a participant $P \in \mathfrak{P}$ to which we want to project, we create the global projection $\mathbf{C}\Downarrow^{g}_{P}$ by using the topological reduction of the representation of **C** using the projection filter $\mathcal{F}_P$ (as defined in Section 9.4.1) to define the set of completions for $\mathbf{C}\Downarrow^{g}_{P}$ up to the offers part of each step. As a projection must be a protocol machine, the reduction used for the projection must be **path-deterministic** (as described in Section 6.4.3). This is because we do not assume that two transitions in a topological representation that carry the same label necessarily perform the same update, so could have different instructions in their attached "bubbles". If the reduction were merely deterministic (as opposed to path-deterministic), two transitions carrying the same label but different update bubbles could not necessarily be distinguished. This gives the following condition:

> The reductions of the topological representation of a choreography **C** to create participant projections must be path-deterministic. (9.12)

For the offers part we require that it is possible to define a family of data constructors to generate the offered data of the choreography, denoted by $\mathcal{C}^{\natural}, \natural \in [\mathfrak{P}] \cup [\mathfrak{P}]^2$, for **C**. The family has one data constructor for each participant and one for each pair of participants, with signatures:

$$\mathcal{C}^{\natural} :: actions(asSet(\mathcal{B}_{\mathbf{C}}))^* \rightarrow restr(\mathbb{U}_P, \Omega_{\mathbf{C}} \cap (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\natural})$$

For any $t \in stems(\mathcal{B}_{\mathbf{C}})$, for any $\{P\} \in [\mathfrak{P}]$, $\mathcal{C}^{\{P\}}$ must satisfy:

$$\begin{aligned} &\mathcal{C}^{\{P\}}(actions(t)) = restr(t^{\omega}, \Omega_{\mathbf{C}} \cap (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\{P\}}) && \text{if } \mathcal{F}_P(t^{\alpha}) \\ &\mathcal{C}^{\{P\}}(actions(t)) = \mathcal{C}^{\{P\}}(actions(trunc(t))) && \text{otherwise} \end{aligned} \tag{9.13a}$$

and for any $\{P, Q\} \in [\mathfrak{P}]^2$, $\mathcal{C}^{\{P,Q\}}$ must satisfy:

$$\begin{aligned} \mathcal{C}^{\{P,Q\}}(actions(t)) &= restr(t^\omega, \Omega_{\mathbf{C}} \cap (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\{P,Q\}}) && \text{if } \mathcal{F}_P(t^\alpha) \wedge \mathcal{F}_Q(t^\alpha) \\ \mathcal{C}^{\{P,Q\}}(actions(t)) &= \mathcal{C}^{\{P,Q\}}(actions(trunc(t))) && \text{otherwise} \end{aligned} \tag{9.13b}$$

The definitions (9.13) mean that:

- When presented with a sequence of actions corresponding to a stem in $\mathbf{C}$, the values generated by $\mathcal{C}^{\{P\}}$ for the offered symbols partitioned to $\{P\}$ match the values that $\Omega_{\mathbf{C}}$ would give those symbols for the same stem.
- Actions in the stem t that do not involve P either as sender or receiver, do not affect the value of $\mathcal{C}^{\{P\}}(actions(t))$.
- When presented with a sequence of actions corresponding to a stem in $\mathbf{C}$, the values generated by $\mathcal{C}^{\{P,Q\}}$ for the offered symbols partitioned to $\{P, Q\}$ match the values that $\Omega_{\mathbf{C}}$ would give those symbols for the same stem.
- Actions in the stem t that do not involve both P and Q, one as sender and the other as receiver, do not affect the value of $\mathcal{C}^{\{P,Q\}}(actions(t))$.

This family of functions allows generation of the offers part of a projection to $P \in \mathfrak{P}$ as follows:

$$t \in prefixes(\mathbf{C} \Downarrow_P^g) \quad \Rightarrow \quad t^\omega = \bigcup_{\natural \in [\mathfrak{P}] \cup [\mathfrak{P}]^2 \wedge P \in \natural} \mathcal{C}^\natural(actions(t)) \tag{9.14}$$

The topological reduction of the choreography as described in Chapter 6.4 combined with the data constructor (9.14) results in a projection that meets (9.10) as follows:

- The topological reduction removes all the actions that are not retained by $\mathcal{F}_P$, so these actions do not affect the behavioural state, as defined in Section 4.2.7.1, of the projection.
- The topological allows (does not remove) any sequence of actions that belongs to $\mathbf{C}$.
- By (9.14), the offers of a step in a prefix of $\mathbf{C} \Downarrow_P^g$ will always be a subset of the offers in the corresponding step of a compatible (per (4.21)) prefix of $\mathcal{B}_{\mathbf{C}}$.

The first of these means that (9.10a) is satisfied, and the second and third mean that (9.10b) and (9.10c) are satisfied.

If (9.12) is met and the offers part of projection can be constructed successfully according to (9.14) then we say that **C** is *projectable* to P.

9.5.2 Participant Contracts

We will use the projections of a choreography as protocol contracts, as described in Chapter 8, for the participants in a collaboration. This recognises that the choreography does not fully describe the behaviour of participants but just defines the protocol that each participant must use to interact with other participants.

As described in Chapter 8, a protocol contract comprises two parts:

- a *contract machine*
- a subset of the alphabet of the contract machine giving the *fully constrained actions*

The first is provided by $\mathbf{C}\Downarrow_P$ the local projection of **C** to P. The second is the set of receive actions in P, generated from the exchanges in $\mathbf{C}\Downarrow_P$ for which P is receiver. Having the choreography fully describe the circumstances under which a message can be received means that if all participants are blocked from sending (so a participant will simply stop when it reaches a send action) then, because a participant cannot otherwise refuse a receive that its contract allows, the collaboration is bound eventually to reach a state with no messages remaining unreceived in the network.[5] This condition is required to establish realizability and we will assume, as standard, that choreographies fully constrain message reception in the participants.

9.6 Realizability

Being able to create participant contracts from a choreography is not enough to guarantee that the choreography "works", in the sense that interaction of projected behaviours is **bound to adhere** to the choreography. Our aim is to be able to construct (design) choreographies that we **know** to be "realizable" and we now proceed to explore how this is done.

[5]This assumes that participant processes progress, so do not crash or deadlock for other reasons, not associated with the choreography.

There are three key results in describing the treatment of choreography realizability using PM:

Result 1. Showing that a choreography that is defined as a **single independent machine** is realizable provided that it obeys conditions termed *asynchronous projectable*. This is addressed in Section 9.7

Result 2. Showing that a choreography that is defined as a **homogeneous composition of independent realizable machines** is itself realizable, in other words "composition preserves realizability". This is addressed in Section 9.8

Result 3. Determining realizability of a choreography that is expressed using a composition involving **dependent machines**. This is addressed in Section 9.9

9.6.1 Definition of Realizable

We start by defining what we mean by *realizable*. Suppose that a collaboration is designed from a choreography according to the scheme in Figure 9.1 on page 152 and that we observe the enactment (Step 3) as a total ordering of the message send action and message receive actions. We assume this ordering is truthful in that respects the causality of the system, as described by Lamport [46], and where actions are observed as simultaneous we order them arbitrarily in a way that respects causality, for instance using timestamps generated by a vector clock mechanism, as described by Fidge and Mattern [22]. We consider realizability in terms of snapshots taken between two successive actions of the ordering. In any snapshot there are, in general, some messages that are "in flight" (sent but not yet received).

We define *realizability* of the choreography to mean:

From any snapshot of the collaboration, it is possible to determine the corresponding state of the choreography. (9.15a)

At any snapshot of the collaboration, those sends allowed in the corresponding state of the choreography may happen as the next action in the collaboration, and only those sends. (9.15b)

If a message is sent it is guaranteed to be received. (9.15c)

9.6.2 Asynchronous Projectable Machines

In this section we define the notion of a machine that is *asynchronous projectable*. We will go on to show that a choreography defined using a machine that is asynchronous projectable is realizable.

Fn: *bufferSet*. As a preliminary we define a function $bufferSet :: \mathfrak{P} \times \mathfrak{P} \to power(\Lambda_{\mathbf{C}})$ defined for distinct P and Q as:

$$bufferSet(P, Q) = \{ x \mid x \in \Lambda_{\mathbf{C}} \wedge (sender(x) = P \vee (sender(x) = Q \wedge receiver(x) = P))\}$$

This definition will be used shortly to ensure that a participant can determine unambiguously which path in a choreography is being followed.

Given a choreography machine $\mathbf{C}$ we establish asynchronous projectable conditions in the unwound form $\overset{\circ}{\overrightarrow{\mathbf{C}}}$ constructed as described in Chapter 6.3 on page 112. $\mathbf{C}$ is asynchronous projectable iff:

In a given state of $\overset{\circ}{\overrightarrow{\mathbf{C}}}$ only one participant may send. (9.16a)

$\overset{\circ}{\overrightarrow{\mathbf{C}}}$ is projectable (as described in Section 9.5.1) to each member of $\mathfrak{P}$ (9.16b)

A participant P only sends from an unambiguous state in $\overset{\circ}{\overrightarrow{\mathbf{C}}}\Downarrow^{g}_{P}$ (9.16c)

For every distinct pair $P, Q \in \mathfrak{P}$ the reduction of $\overset{\circ}{\overrightarrow{\mathbf{C}}}$ using filter $\mathcal{F}_{bufferSet(P,Q)}$ is path-deterministic. (9.16d)

It was noted in Section 6.3.1 that every state of $\overset{\circ}{\overrightarrow{\mathbf{C}}}$ corresponds to exactly one state of $\mathbf{C}$, and that both machines have the same traces. As a result, if any of (9.16a), (9.16b) or (9.16c) were false in $\mathbf{C}$ it would also be false in $\overset{\circ}{\overrightarrow{\mathbf{C}}}$. So if they are all true in $\overset{\circ}{\overrightarrow{\mathbf{C}}}$ they are also all true in $\mathbf{C}$.

9.6.3 Relay Form

A consequence of the conditions for asynchronous projectability is that a choreography has a certain topological form termed *relay form*.

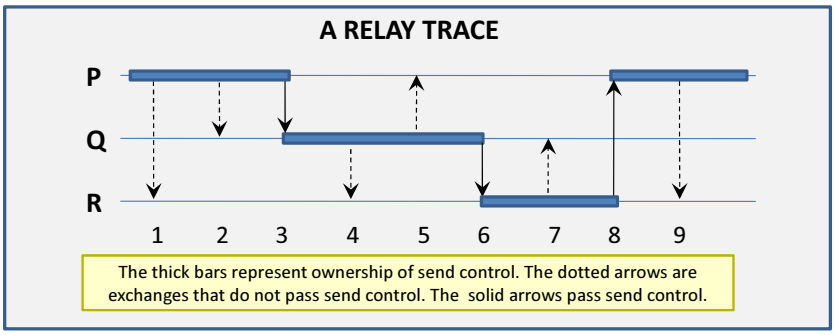


Figure 9.4: Relay Trace

Suppose that σ is a state of **C** and that σ is not a final state, so there is at least one transition starting at σ. By (9.16a) only one participant P may send from σ. If any transition ending at σ is for an exchange that does not involve P as either sender or receiver it is removed in reduction to P. This would make σ an ambiguous state, violating (9.16c). Therefore if σ_1 and σ_2 are adjacent states of **C** so that σ_2 can be reached by a single transition from σ_1, and P is the sender in σ_1, then one the following must be true:

- P is also the sender in σ_2
- Q with $Q \neq P$ is the sender in σ_2 and transitions from σ_1 to σ_2 have Q as receiver.

(9.17)

Any machine that has a single sender in each state, as required by (9.16a), and which conforms to (9.17) is said to have *relay form* or, equivalently, to be a *relay machine*. If **C** is a choreography machine then the predicate:

$$relay(\mathbf{C}) \tag{9.18}$$

evaluates to *true* iff **C** meets (9.16a) and (9.17). Note that the three choreography machines depicted in Figure 9.2 on page 155 all have relay form.

The term "relay" is by analogy with a relay race, in which a baton is passed from one runner to the next. Only the participant "holding the baton" may send, and the holder must "pass the baton" to the next sender[6]. While in possession of the baton

[6]This analogy is not exact, as there is no representation of the baton in the choreography. A send event that "passes the baton" is syntactically and semantically identical to one that does not.

a participant may pass through any number of states as sender, and send any number of messages to different participants, before passing the baton to the next sender. Although all the transitions from a given state must all have the same sender, different transitions from the same state may have different receivers. Figure 9.4 shows an example of a relay trace choreography between three participants.

Note that relay form of itself is not sufficient for asynchronous projectability as it does not require that condition (9.16d) holds.

9.7 Result 1: Single Machine Realizability

This section addresses the first result listed in Section 9.6. We assume in this section that we have a choreography $\mathfrak{C} = \langle \mathbf{C}, \mathbf{F} \rangle$ between a set of participants $\mathfrak{P}$ where:

- The universe of the choreography and the universes of the participants are configured as laid out in Section 9.3.
- **C** is a single independent protocol machine satisfying the asynchronous projectable conditions (9.16).
- **F** requires that receives are fully constrained by the choreography.
- All participants obey a contract obtained by projection of **C** as described in Section 9.5.

9.7.1 Path Matching

In this section we show that, given these assumptions, any enactment of a collaboration is *path matched*, meaning that all participants follow a single completion of **C**. In the next section we will show that this means that the choreography is realizable, as defined above in Section 9.6.1. We use induction over the states of a collaboration to prove that it is path matched, and for this we define the concept of an *enactment*.

An enactment $\mathfrak{E}$ captures the complete history of a collaboration to some point. Conceptually it is a sequence of snapshots of the state of the collaboration. Each snapshot contains:

- The state of each participant's choreography contract (including the values of any owned attributes), and
- The content of every buffer between participants.

A new snapshot is added to the enactment whenever any participant executes a choreography action (a send or a receive), and for this purpose the choreography actions executed in the collaboration are given a global sequence as described in Section 9.6.1. We use *enactment state* to refer to the last state (snapshot) of $\mathfrak{E}$. In addition we will use:

- $\mathfrak{E}^g$ to denote the subsequence of the sequence of sends and receives in $\mathfrak{E}$ consisting of just sends, and
- $\mathfrak{E}^P$ to denote the sequence of choreography actions (message sends and receives of this collaboration)[7] executed so far by participant P.

An enactment $\mathfrak{E}$ is *path matched* iff:

$$\exists\ \pi^g_{\mathfrak{E}} \in \Pi_{\mathbf{C}}(\bullet) \text{ with } path_{\mathbf{C}}(\mathfrak{E}^g) = \pi^g_{\mathfrak{E}} \tag{9.19a}$$

$$\forall\ P \in \mathfrak{P} \quad \exists\ \pi^P_{\mathfrak{E}} \in prefixes(\pi^g_{\mathfrak{E}}) \text{ with } tranTrace_{\mathbf{C}\Downarrow P}(\mathfrak{E}^P) = filter_{\mathcal{F}^{\Theta}_P}(\pi^P_{\mathfrak{E}}) \tag{9.19b}$$

where $\Pi_{\mathbf{C}}(\bullet)$ is a sequence if transitions in C starting from the initial state $\bullet$ as described in Section 6.1.2 on page 109. The first condition (9.19a) requires that there be a path in the choreography C that matches the trace defined by the sequence if sends. The second requires the sequence of choreography actions in each participant is a prefix of this choreography path, filtered to that participant. Note that because the choreography is a protocol machine and therefore deterministic, given $\mathfrak{E}^g$ there is at most one $\pi^g_{\mathfrak{E}} \in \Pi_{\mathbf{C}}(\bullet)$ with $path_{\mathbf{C}}(\mathfrak{E}^g) = \pi^g_{\mathfrak{E}}$.

We now use mathematical induction to show that if the choreography is *asynchronous projectable*, obeying conditions (9.16), the following conditions are sustained in a collaboration:

$$\text{The collaboration is path matched, so obeys (9.19)} \tag{9.20a}$$

[7] The participant may also engage in other actions which are not part of the collaboration. For instance it may also be participating in another, separate, collaboration. These are not visible for the purposes of the current argument.

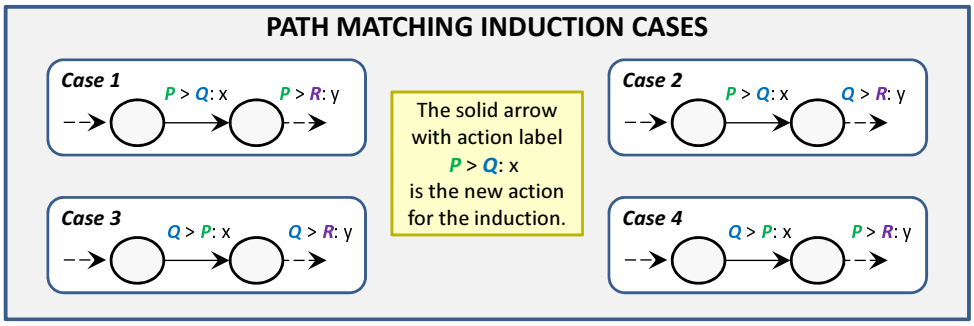


Figure 9.5: Induction Cases

In any enactment state $\mathfrak{E}$ there is at most one sender. (9.20b)

P is sender in enactment state $\mathfrak{E} \Rightarrow length(\pi^P_{\mathfrak{E}}) = length(\pi^g_{\mathfrak{E}})$ (9.20c)

We consider moving from an enactment state $\mathfrak{E}$ to the next enactment state $\mathfrak{E}'$ as a result of a single choreography event (a send or a receive). Without loss of generality, we suppose that the event that moves the collaboration $\mathfrak{E} \rightarrow \mathfrak{E}'$ is performed in participant P, and is a send to participant Q or a receive from participant Q. There are four cases as depicted in Figure 9.5 and considered below.

Case 1. Send and remain sender. As a consequence of (9.16a) and (9.16b):

The only send transitions possible from a given state in a participant are those possible from the corresponding state of the choreography. (9.21)

This is because, in a reduction to P (whether *simple* or *exact*) the outgoing transitions from an unambiguous state from which only P can send are exactly those outgoing transitions from the unique corresponding state in the choreography and thus path matching is preserved.

Case 2. Send and cease to be sender. The argument is the same as Case 1 above, except that P ceases to be sender so that the number of senders is now 0.

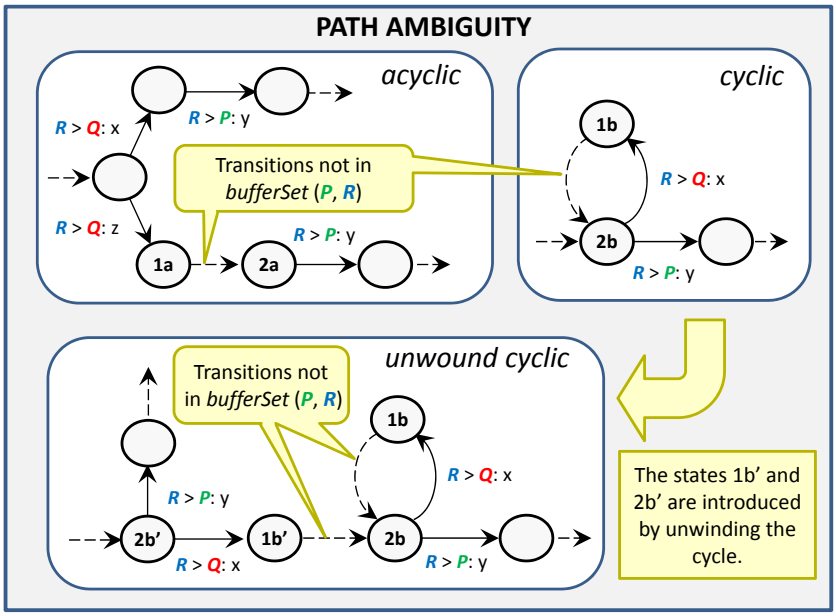


Figure 9.6: Path Ambiguity

Case 3. Receive but don't become sender. In this case we require that the trace $\mathfrak{E}'^{P}$ based on the new enactment state obeys (9.20c). This requires that:

$$\exists\ \pi^{P}_{\mathfrak{E}'} \in \mathit{prefixes}(\pi^{g}_{\mathfrak{E}}) \text{ with } \mathit{tranTrace}_{\mathbf{C}\Downarrow P}(\mathfrak{E}'^{P}) = \mathit{filter}_{\mathcal{F}^{\ominus}_{P}}(\pi^{P}_{\mathfrak{E}'}) \tag{9.22}$$

It is clearly **possible** that (9.22) is met, as follows. Suppose that the message that is received by P from Q was written in the step that ended with enactment state $\mathfrak{E}^{\ddagger}$. All the messages sent to P up to enactment state $\mathfrak{E}^{\ddagger}$ can be received by P because:

- The communication model (9.1b) means that these messages cannot be delayed indefinitely in the network so must reach P.
- The requirement stated in Section 9.5.2 that participant contracts *fully constrain receives* means that P is able to consume any message that is available as the next in one of its buffers as soon as it arrives.
- The communication model (9.1c) means that P cannot be blocked at a send, so cannot be prevented from advancing to its next receive by intervening sends.
- Because, by the induction assumption, (9.20) is met at $\mathfrak{E}^{\ddagger}$ there is a path in the choreography that matches the sequence of sends up to and including $\mathfrak{E}^{\ddagger}$.

- Projection semantics (9.10b) guarantees that the receives in P's projected behaviour can receive the messages sent to P by any other given participant in the order in which they were sent by that participant; and the communication model (9.1a) means that the network preserves this order.

If (9.22) **can** be met in moving from $\mathfrak{E}$ to $\mathfrak{E}'$ and moreover there is only one receive that is possible for P, then (9.22) **must** be met. This is the case when:

- There is only one receive that P can execute, as its current state in $\mathbf{C}\Downarrow_P$ only has a single exit transition.
- There is more than exit transition from the current state in $\mathbf{C}\Downarrow_P$, but only one receivable message can be available at the head of the FIFO buffers by which P receives from other participants.

The only eventuality in which departure from (9.22) is possible is where there is more than exit transition from the current state in $\mathbf{C}\Downarrow_P$ **and** messages satisfying these transitions can be available simultaneously to P at the head of the FIFO buffers by which P receives from other participants. If this were the case, it would lead to *path ambiguity* whereby it is possible for P to depart from the choreography, in violation of path matching. There are two sub-cases of path ambiguity to consider, *acyclic* and *cyclic*, as shown in Figure 9.6.

In the *acyclic* case, shown top left of Figure 9.6, we imagine that lower path in the choreography, involving the z message, is the one that has actually been taken. However we suppose that a different path is possible from this state in which P instead receives a y message from R, shown in the upper path of the choreography. Moreover, we suppose that the lower (true) path has a later message of type y sent to P from R, which we assume without loss of generality to be the first exchange of a message of type y to P from R after state $1a$. This later message could be available already in the buffer between R and P and could be consumed by P in error, taking P along the upper (false) path of the choreography. However this would only be possible if both of the following were true:

- There are no other transitions between states $1a$ and $2a$ in Figure 9.6 involving R as sender and P as receiver, as such messages would appear in the buffer between

R and P before the y message, making the y message unavailable for consumption.

- There are no transitions between states $1a$ and $2a$ that have P as sender as then the choreography, and so also by (9.20c) $\mathbf{C}\Downarrow_P$ itself, would have had to advance through this state before the transition for the y message beyond it could be reached.

These transitions all belong to $\mathit{bufferSet}(P, R)$. If there are no such transitions between states $1a$ and $2a$ and all other transitions were removed, the resultant reduced LTS would have two transitions for label y starting from the same state but representing different transitions in the original choreography. This would be in violation of asynchronous projectability condition (9.16d).

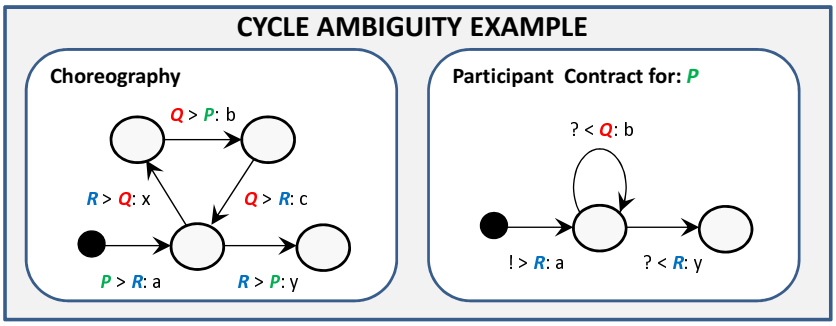


Figure 9.7: Cycle Ambiguity Example

The *cyclic* case, shown at the top right of Figure 9.6, is similar. The possibility here is that the y message that is sent on exit from the circuit is available to P while messages destined for P sent within the loop are still in transit and not yet visible to P. In this case P could falsely assume that it can take the y transition immediately. An example of this is shown in Figure 9.7, where P might receive the y message from R and exit the loop, even though a b message from Q is still on its way. The b message would then never be received and would remain marooned in the network. As in the acyclic case, the confusion is between y messages in different traces; the difference with the acyclic case is that the two messages both relate to the same transition in the choreography. This is addressed by using the **unwound form** of the choreography as the basis for asynchronous projectability so that transitions, instances of which are separated by a circuit, become separate transitions, with different transition identifiers, in

the graphical LTS. The unwound form, in the lower half of Figure 9.6, shows this for the y transition which now appears twice. If these can be distinguished using (9.20c), in the manner described for the acyclic case, then the confusion cannot arise.

Case 4. Receive and become sender. The reasoning that P continues on the correct path of the choreography, so that path matching is preserved, is as for Case 3. We need to establish that (9.20b) and (9.20c) are preserved.

As, by assumption, the enactment has obeyed (9.20) to $\mathfrak{E}$, a send has been executed for every transition in $\pi^g_{\mathfrak{E}}$. This means that at $\mathfrak{E}$ all the sends in $\mathit{filter}_{\mathcal{F}^\Theta_R}(\pi^g_{\mathfrak{E}})$ for all $R \in \mathfrak{P}$ must have been used, so in particular there can be no send in $\mathit{filter}_{\mathcal{F}^\Theta_P}(\pi^g_{\mathfrak{E}})$ beyond $\mathit{filter}_{\mathcal{F}^\Theta_P}(\pi^P_{\mathfrak{E}})$. This could not be the case if $\mathit{length}(\pi^P_{\mathfrak{E}'}) < \mathit{length}(\pi^g_{\mathfrak{E}})$ and P is to become sender at $\mathfrak{E}'$. As this case involves a receive and doesn't advance the choreography state, $\mathit{length}(\pi^g_{\mathfrak{E}'}) = \mathit{length}(\pi^g_{\mathfrak{E}})$. This establishes (9.20c).

Now suppose that R (with possible $R = Q$ but $R \neq P$) is sender in $\mathfrak{E}'$, so that both R and P are senders in $\mathfrak{E}'$. As the states of R and the choreography are both unchanged by $\mathfrak{E} \to \mathfrak{E}'$, this means that R was the sender in $\mathfrak{E}$ and hence that $\mathit{length}(\pi^R_{\mathfrak{E}}) = \mathit{length}(\pi^g_{\mathfrak{E}})$. As, by assumption, P is sender in $\mathfrak{E}'$ this gives $\mathit{length}(\pi^R_{\mathfrak{E}'}) = \mathit{length}(\pi^g_{\mathfrak{E}'}) = \mathit{length}(\pi^P_{\mathfrak{E}'})$ so that P and R are equally advanced in the choreography and agree on its state. If both are senders at $\mathfrak{E}'$ then by (9.16c) both must be in an unambiguous state, establishing both to be at a single state of the choreography. This single state must be the same in both, so it is impossible by (9.16a) for both to be senders. This establishes (9.20b).

The induction starts at the initiated state of the collaboration, at which the choreography and all projected participant machines are at their initial $\bullet$ and all buffers are empty.

9.7.2 Data Synchronization

In this section we consider the degree of data synchronization achieved between the owned data Ω of each participant. First recall from (9.10c) that the owned data of each projection is based on the partitioning of the universe. Now suppose that we have an enactment state $\mathfrak{E}_{\mathit{empty}}$ where all the FIFO buffers between participants are empty, so that every participant is at the same state in the choreography as determined by

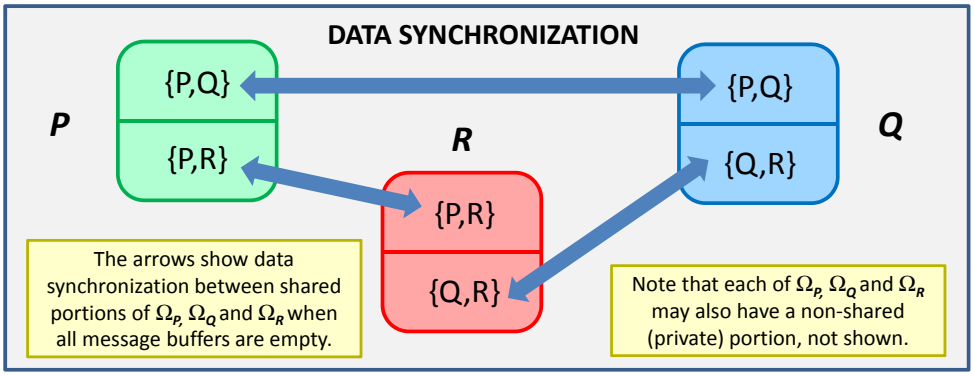


Figure 9.8: Data Synchronization

$path_{\mathbf{C}}(\mathfrak{E}^{g}_{empty})$. Figure 9.8 shows how the owned data of the participant contracts will be synchronized in this enactment state. For instance, if $x \in \Omega_{\mathbf{C}} \cap (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\{P,Q\}}$ then x will have a value in both Ω_P and Ω_Q that is synchronized with the value that would pertain in $\Omega_{\mathbf{C}}$ after following $path_{\mathbf{C}}(\mathfrak{E}^{g}_{empty})$.

In an enactment state $\mathfrak{E}_{\neg\, empty}$ where the buffers are not empty, suppose that P is the sender of $last(\mathfrak{E}^{g}_{\neg\, empty})$. Then P's owned data will be synchronized with the choreography, as will the owned data of any other participant whose input buffers are empty. The owned data of participants whose input buffers are not empty will be synchronized with an **earlier** state of the choreography, depending on their progress along the choreography path $path_{\mathbf{C}}(\mathfrak{E}^{g}_{\neg\, empty})$.

9.7.3 Proof of Realizability

It is now easy to establish that the three conditions for choreography realizability given in Section 9.6.1 are met. Condition (9.15a) follows immediately from (9.19a) as the state of the choreography can be determined from the global sequence of sends. Condition (9.15b) follows from (9.21). Condition (9.15b) can be established by assuming it is untrue and reaching a contradiction as follows:

Suppose that a choreography follows a path π^{g} that results in one or more messages being unreceivable and remaining forever "in flight" in a buffer. There must be a first such message $\mathtt{m}_{fail}$, being the first message instance sent in π^{g} that cannot be received. Without loss of generality, suppose that P is the intended receiver. By (9.19b) P follows

a path that has a receive for every message sent to it in π^{s} in the order sent, and as by assumption all up to $\mathtt{m}_{fail}$ are successfully received P must advance to the point where it can receive $\mathtt{m}_{fail}$. No message written later than $\mathtt{m}_{fail}$ can overtake it in the buffer so $\mathtt{m}_{fail}$ must be available at the head of a buffer for P to receive. As described in Section 9.5.2, the projected choreography contract for P requires that receives cannot be blocked so P **must** therefore receive $\mathtt{m}_{fail}$.

9.8 Result 2: Composite Choreographies

Defining a choreography as a single machine is limiting, in particular because it means that at any point in a collaboration a given participant can either send or receive but not both. In this section we consider choreographies defined as a homogeneous composition of **independent** choreography machines (compositions involving dependent machines are considered in Section 9.9).

With the use of composition a given participant is able to both send messages and to receive messages at a given state of the choreography. For example the choreography in Figure 9.2 on page 155 is a composition of three machines and the following is a possible state:

- In the top machine, the *Customer* is about to send *Request Amend* to the *Supplier*.
- In the middle machine, the *Customer* is ready to receive any of *Accept Amend*, *Accept Cancel* or *Invoice* from the *Supplier*.
- In the lower machine, the *Customer* is about to send *Confirm Delivery Date* to the *Delivery Co.*.

The choreography has three relay machines in composition and each has its own "baton". You can have as many relay machines composed in the choreography as you like. We now show that a composition of realizable choreography machines is also realizable.

9.8.1 Distribution of Reduction over Composition

In general reduction does not distribute over composition. So if we have choreography $\mathfrak{C} = \langle \mathbf{C}, \mathbf{F} \rangle$ with $\mathbf{C} = \mathbf{C1} \parallel \mathbf{C2}$ then, in general:

$$\mathbf{C}{\Downarrow}_P^g \quad \neq \quad (\mathbf{C1}{\Downarrow}_P^g) \parallel (\mathbf{C2}{\Downarrow}_P^g)$$

This follows from Hoare's work on *concealment* in CSP. The relevant result concerns concealment of a set of actions c in the composition of P and Q, and states that:

$$\text{If } \alpha(P) \cap \alpha(Q) \cap c = \emptyset \text{ then } (P \parallel Q)\backslash c = (P\backslash c) \parallel (Q\backslash c) \tag{9.23}$$

where, using Hoare's notation, $\alpha(P)$ denotes alphabet and $\backslash$ denotes concealment. This result shows that in general concealment only distributes over composition if no concealed action belongs to the intersection of the alphabets of the components. We can equate removal of an action by reduction with concealment, so (9.23) implies that reduction only distributes over composition if no action removed by the reduction belongs to the intersection of the alphabets of the components.

The strategy for projection of choreographies expressed as compositions is to project the components individually, and compose local projections. We now show that where the components are individually realizable then this strategy realizes the choreography as a whole.

9.8.2 Composition Preserves Realizability

We assume that the choreography, $\mathfrak{C} = \langle \mathbf{C}, \mathbf{F} \rangle$, between a set of participants, $\mathfrak{P}$, is defined as a homogeneous composition of n **independent** machines over a universe $\mathfrak{U}_{\mathfrak{C}}$:

$$\mathbf{C} = \prod_{i \in \{1 \ldots n\}} \mathbf{C}^i \tag{9.24}$$

where i indexes over the components of the choreography and is called the *choreography index*. In this section we assume that each of these machines $\mathbf{C}^i$ has a topological representation that is the basis for projection. We consider in Section 9.9 the case where a choreography involves dependent machines and so where topological representation is not possible.

The contract for a participant, $P_j \in \mathfrak{P}$, is a composition of independent protocol machines:

$$\mathbf{C}\Downarrow_{P_j} = \prod_{i \in \{1...n\}} \mathbf{C}^i\Downarrow_{Pj} \tag{9.25}$$

where $\mathbf{C}^i\Downarrow_{P_j}$ is the machine for participant P_j projected from choreography machine $\mathbf{C}^i$. This projection, along with the standard stipulation that a choreography fully constrains all receive actions in P_j, forms a protocol contract for each $P_j \in \mathfrak{P}$.

We suppose that every component $\mathbf{C}^i$ of the choreography is asynchronous projectable, so that individually their projections are path matched. We suppose that the collaboration is at enactment state $\mathfrak{E}$ and that the global sequence of sends so far is $\mathfrak{E}^g$. Suppose that each $\mathbf{C}^i$ is in state σ^i, so that the choreography as a whole is in state $< \sigma^1, \sigma^2, \ldots, \sigma^n >$. By the same reasoning used for a single process, realizability conditions (9.15a) and (9.15b) are met.

In order to show that (9.15c) is met, suppose that a message instance $\mathtt{m}_{fail}$ is the first one sent in the global sequence of sends $\mathfrak{E}^g$ that cannot be received. Suppose that $\mathtt{m}_{fail}$ was sent in a transition in P with an exchange symbol value `P>Q:m` and that the set of machines in the choreography that engage in this exchange is given by:

$$\mathbf{C}_{fail} = \prod_{\{(\mathtt{exchange}_{\mathfrak{E}^P} = \mathtt{P>Q:m})\} \Subset alphabet_{\mathbf{C}^i}} \mathbf{C}^i$$

so all other machines in $\mathbf{C}$ ignore this exchange. We now observe that, at a point of sending $\mathtt{m}_{fail}$, the projections of all the component machines in $\mathbf{C}_{fail}$ to P must be synchronized at states able to participate in the send of $\mathtt{m}_{fail}$. This means that the paths followed by each of the projections must reach this send. As all the machines are path matched, the paths followed by the projections to Q must all contain the corresponding receive for $\mathtt{m}_{fail}$. As, by assumption, no previous message sent in the choreography fails to be received, all the projections of component machines in $\mathbf{C}_{fail}$ to Q must reach the receive. Q must obey the "fully constrained actions" part of the protocol contracts projected from the choreography, it cannot refuse to receive a message whose receipt is allowed by the choreography, so $\mathtt{m}_{fail}$ can be received by Q.

9.8.3 Data Synchronization

Where a choreography is defined as a composition of independent machines, the remarks in Section 9.7.2 apply to each machine independently. It must be remembered that the last participant to send will, in general, be different for different machines in the composition.

9.8.4 Refactoring Choreographies

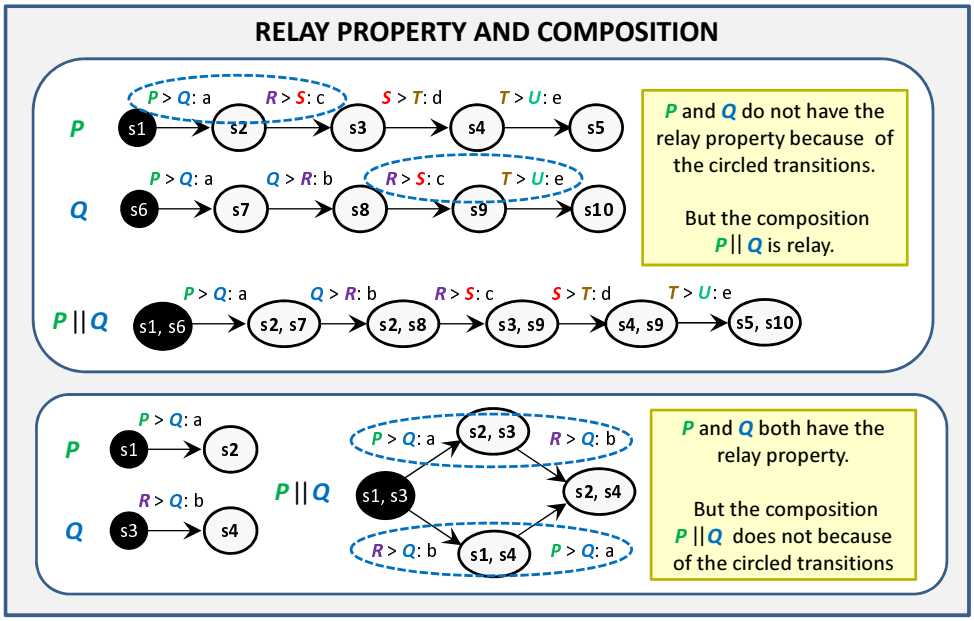


Figure 9.9: Relay Property and Composition

As is probably clear, the use of composition in choreographies is a device that enables realizability to be established easily. The set of components chosen in the composition is not determined or tied to the set of participants, for instance the choreography in Figure 9.2 on page 155 has four participants and is modelled as a composition of three processes.

As Figure 9.9 shows, the relay property can be both created and destroyed by composition. The author of a choreography has the freedom to engineer and refactor the machines of the choreography to achieve realizability. It also important to understand that this freedom to refactor without altering the behavioural semantics of the composition exists because the composition is homogeneous, as the component machines are

all defined over a single universe. Such refactoring is not, in general, possible in the context of heterogeneous composition as the different universes constrain the structure of the composition.

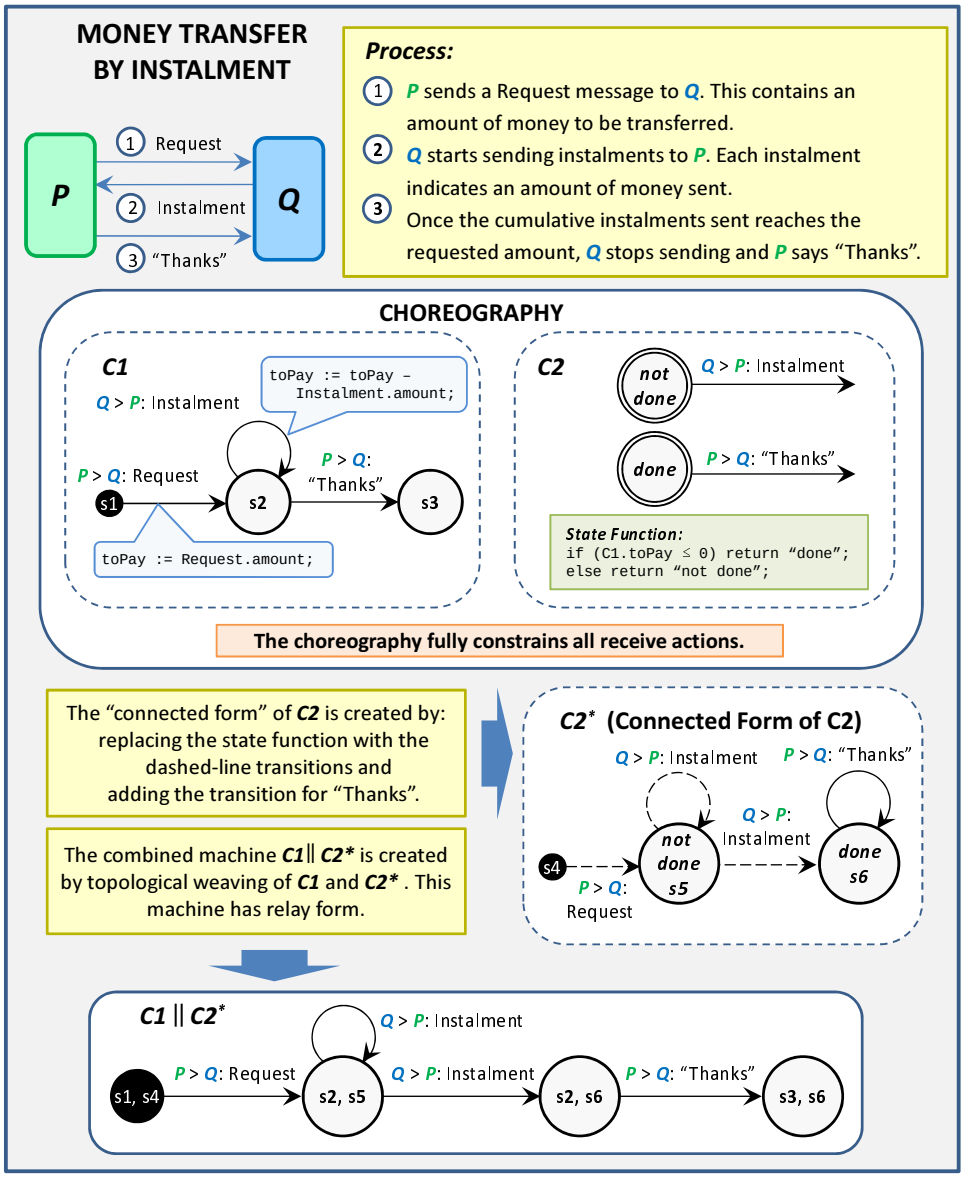


Figure 9.10: Instalments Example

9.9 Result 3: Dependent Choreography Machines

So far we have considered choreographies constructed from independent machines. It is axiomatic in PM that sometimes behavioural rules are not describable just in terms of independent machines (that can be described in pure topology) but need the use of data and derived states. In this section we explore the use of data and derived states to describe a choreography.

9.9.1 Interpretation of Dependency in Choreopgraphies

We consider choreographies defined as homogeneous compositions, where one or more of the composites uses a derived state. Where a message exchange is constrained by a pre-state constraint in a derived-state machine to start at a state σ it has the following semantics:

- The sender can only send when in state σ; and
- The receiver can only receive when in state σ.

Corresponding semantics are given to a post-state constraint.

The reason for this interpretation, whereby sender and receiver are both constrained, is as follows:

- If only the sender were constrained, the constraint should properly be represented as a business rule in the sender rather than as part of the choreography.
- If only the receiver were constrained, the choreography would not in general be realizable as the receiver cannot be guaranteed to obey the constraint.

Realizability requires that *both sender and receiver* be able to determine whether they are in, or will enter, the derived state used to constrain the exchange.

For example, the process shown in Figure 9.10 requires that both P and Q track the amount of money that has been transferred, as P must stop sending and Q says "Thanks" when the requested amount has been reached. Both machines must know that the requested amount has been reached for this protocol to work. Note that machine $C1$ does not have relay form as both P and Q can send from state $s2$. Moreover

the technique for projection described in Chapter 6.4 applies to independent machine and could not be applied to $C2$. The strategy here will be:

- To show that $C1$ and $C2$ **in composition** is realizable.
- To describe a mechanism for projection of derived-state machines.

9.9.2 Realizability Analysis of Derived-State Machines

We argue that the *connected form*, as described in Chapter 6.5 can be used in place of the original derived-state machine in the context of realizability analysis as follows. As the connected form machine describes the transition possibilities of the machine it approximates, its prefixes are a superset of the prefixes that are possible for the original derived-state machine. If the prefixes of the connected form conform to the conditions (9.16) for asynchronous projectability, the subset that constitute the prefixes possible for the original derived-state machine must also conform.

Suppose that in a machine C of choreography $\mathfrak{C}$ is dependent on a symbol $\mathtt{x}$ which is provided by another machine C' of the choreography, so:

$$\mathtt{x} \in \Omega_{C'} \cap \mathit{symb}(\mathit{perceives}(\mathit{asSet}(\mathcal{B}_C))) \tag{9.26}$$

Suppose moreover that $\mathtt{x} \in (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\{P,Q\}}$. The behaviour of C is then only defined in projections to P and Q as, according to (9.9b), $\mathtt{x}$ is not present in the universes of other participants. As we require that the state is computable in both the sender and the receiver of an exchange, C cannot constrain an exchange that involves participants other than P and Q so:

$$\mathit{alphabet}_C \subseteq \{\ A \mid A \in \mathit{restr}(\mathbb{U}_{\mathfrak{C}}, \mathcal{A}_{\mathfrak{C}}) \wedge \{\mathit{sender}(A), \mathit{receiver}(A)\} = \{P, Q\}\ \} \tag{9.27}$$

This implies that the behaviour of C cannot be dependent on any other symbol $\mathtt{y}$ with $\mathtt{y} \notin (\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\{P,Q\}}$ as $\mathtt{y}$ would not be present in projections of C to both P and Q and could not constrain exchanges between these two participants. Noting that $\mathtt{x}$ can be a derived symbol, based on other symbols in $(\mathcal{U}_{\mathfrak{C}} \backslash \mathcal{A}_{\mathfrak{C}})^{\{P,Q\}}$, we can, without loss of generality, take it that C's dependency is confined to $\mathtt{x}$ and that $\mathtt{x}$ represents the state of C.

Using the concepts defined in Section 6.5.1 we form the connected form C^* of C. We use this connected form, combining it with other machines using the technique in Chapter 6.2 where required, to determine that an assembly involving derived-state machines has asynchronous projectability. In Figure 9.10 on page 183 the connected form, $C2^*$, of $C2$ is composed with $C1$ to form $C1 \parallel C2^*$ and this machine, shown at the bottom of Figure 9.10 allows realizability to be established.

9.9.3 Projection of Derived-State Machines

Using connected form enables analysis for realizability, but as the connected form machine is an approximation of the original derived-state machine it cannot be used as the basis for projection, and we need to look for another basis for projection.

By (9.27) the alphabet of C involves only the two participants P and Q. This means that C is completely preserved in projection to P and Q and completely lost in projection to all other participants. In particular, all transitions in C are retained in projection to P and Q. This allows us to invoke Hoare's result (9.23) and distribute projection over composition. Thus we take the projection of C to P and Q to be C (unchanged) and compose locally with the projections from other machines of the choreography.

9.10 Choreography Contract Framework

With the idea that the contracts extracted from a choreography are concerned with ensuring that a collaboration "works", in the sense that it realizes the choreography, we can construct an instance of the framework set out in Section 8.2 as follows:

- $\mathcal{T}$ is a combination of topological and inductive reasoning, based on the properties of parallel composition in PM, the definition of projection, and the well-formedness conditions for realizability.
- $\mathcal{P}$ is a statement that the choreography must be realizable, meaning that the collaboration of participants extracted from it meets (9.15) given in Section 9.6.1 above.
- C is a set of protocol contracts, projected from the choreography, that the participants must satisfy.

- S is an implementation of the participant behaviours such that each satisfies its projected contract.

The motivation for using this contract based approach for the design of participants is that:

- The choreography does not completely specify the behaviour of a participant. This is supported by the use of protocol machines, which specify *possible* orderings of actions but leave the *choice* of action in a given state open.
- A participant can simultaneously engage in two or more completely independent choreographies and, in this case, each choreography gives rise to a contract for the participant. This is supported by the fact that protocol contracts can be composed, as described in Section 8.5, the composition of two being another contract.
- Realizability of a choreography for asynchronous collaboration requires that receive actions in a participant must not be prevented (refused). For instance, suppose that a customer requests a loan from a bank. The bank may either grant or deny the loan, at its own discretion. But the customer has no discretion about choosing which message to receive: it must receive whichever the bank sends. This is specified in the contract using the mechanism for fully constrained actions, F.

Finally, note that a contract is concerned with *protocol* (that a loan once requested is either granted or denied) and not with *computation*. How the bank decides whether to grant or deny a loan is of no consequence for realizability of the choreography, and two banks that both obey the collaboration contract are substitutable for each other with respect to realizability even if the processes and criteria they use for deciding when to grant and when to deny credit are completely different.

9.11 Simple versus Exact Reduction

In Chapter 6.4 we described two forms of reduction: *simple reduction* and *exact reduction*. Both of these can be used to project a choreography to obtain participant contracts. This raises the question of which is "better".

Both techniques model their states using members of the powerset of the set of states of the choreography; and you can view each member of the powerset as a representation of your uncertainty: "I could be any of these states, but I don't know which one I am actually in". Exact reduction gives exactly the right measure of uncertainty after each filtered trace. The member of the powerset you are in after the filtered trace represents **exactly** the set of states of the original choreography you could be in. Simple reduction, on the other hand, can give incorrect uncertainty. Sometimes the member of the powerset you are in after a filtered trace is a superset of states of the original choreography that you could actually be in. It is this inexact representation of uncertainty in simple reduction that then allows impossible traces. As we see below, this has consequences for the safety of a collaboration enacted using the projected behaviours.

9.11.1 Well-Behaved Projections

In Section 4.4.2 we defined a concept of well-behaved models: a model that is guaranteed to preserve data integrity. Here we explore the consequences of assuming that a choreography is well-behaved.

Suppose a choreography machine **C** is both realizable and well-behaved. If **exact reduction** as described in Section 6.4.2 is used to create participant contracts then the projected participant contracts will also be well-behaved. This is because exact reduction guarantees that the behaviour of a projected participant cannot depart from the behaviours allowed by the choreography, as defined in (6.5). If the data image of the choreography remains within the valid universe then the partition of the image belonging to a participant will remain within the valid universe as well. This means that data integrity in the participants is preserved, so that the participant contracts are also well-behaved.

If on the other hand **simple reduction** as described in Section 6.4.1 is used, there may be paths in a projection that were not present in the choreography. Suppose a participant P in a collaboration misbehaves by sending a message to another participant Q that the choreography does not allow. If Q's behaviour was created by simple reduction it could allow receipt of this rogue message and as a result could follow a path in Q not possible in the choreography. This is despite the fact the rogue message conforms to the valid universe $\mathbb{V}_{\mathfrak{C}}$ of the choreography, and therefore by (9.9d) of Q

as well. The well-behaved property of the choreography now provides no guarantees for the participant, which could depart from the valid universe so that data integrity is lost. This means that participant contracts created using simple projection are not, in general, well-behaved. For this reason, exact reduction is to be preferred over simple reduction.

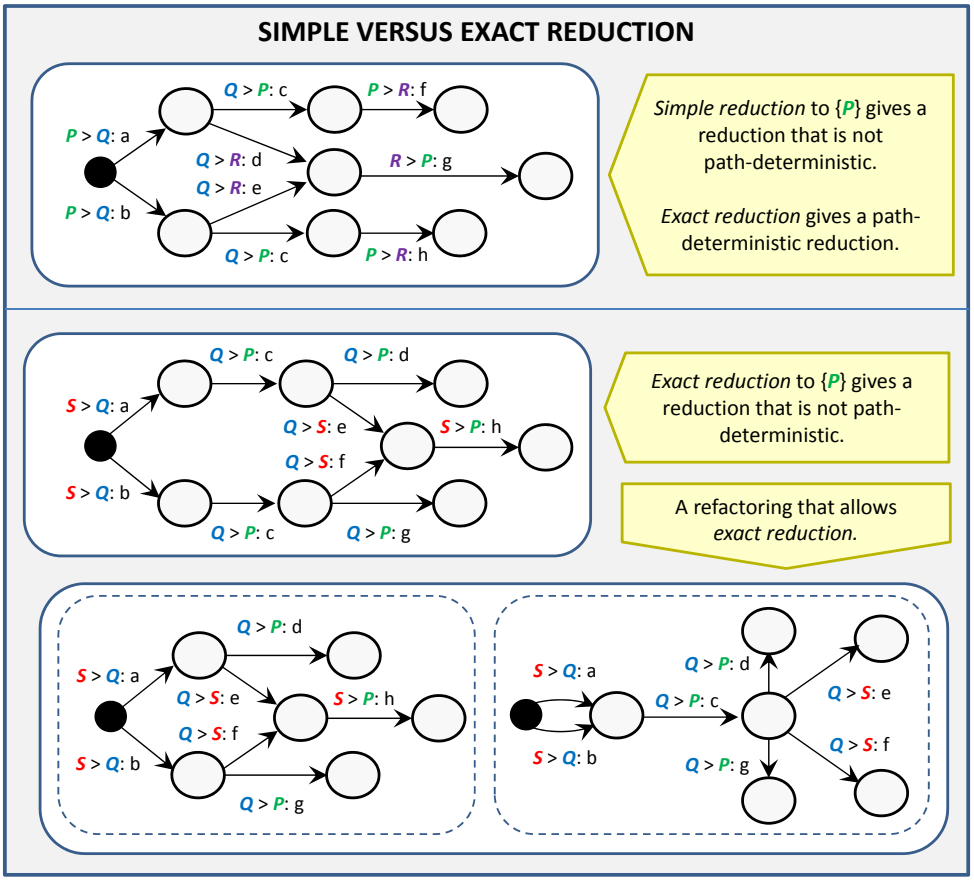


Figure 9.11: Simple versus Exact Reduction

9.11.2 Refactoring For Projectability

As Figure 9.11 shows, it is possible that simple reduction of a choreography is not path-deterministic but exact reduction is. In some cases, as shown in the lower half of Figure 9.11, it may be possible to refactor a choreography that cannot be reduced using exact reduction into one that can, thus allowing the guarantee of well behaviour to be regained. This is a further example of the idea introduced in Section 9.8.4 that a

choreography may be refactored to establish desired properties.

9.12 Choreography and Objects

The choregraphies illustrated so far have been modelled as a fixed population of machines all instantiated at the start of a collaboration. In some cases a choreography requires instantiation of new machines as the collaboration progresses.

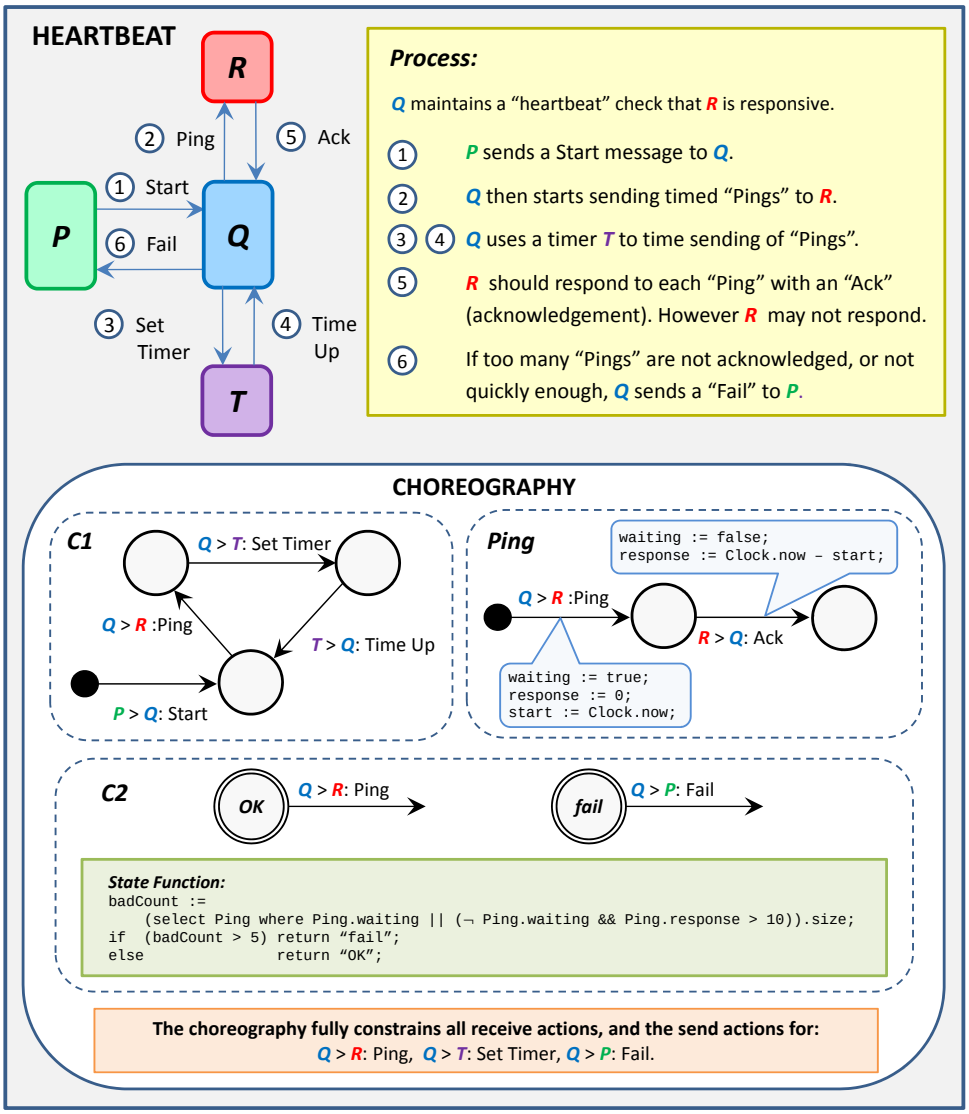


Figure 9.12: Heartbeat Example

Figure 9.12 shows a choreography for a "heartbeat" process, where participant Q monitors whether participant R is functioning by repeatedly sending "pings" to which R is expected to respond within a fixed time. If too many pings fail to get a response within the fixed time then the monitored participant is deemed to have ceased functioning.

Each ping instantiates a machine of the choreography, and so the population of machines changes over the life of the collaboration. In terms of PM, this is supported by the ideas described in Chapter 4.6. As described there, a dynamic population of machines is formally represented as a fixed population whose members are "woken up" by an action that they do not ignore; in this case the exchange of a "ping" message. With this formalization of instantiation, the theory given in the earlier parts of this chapter concerning realizability carries across, without change, to choreographies that involve a dynamic population of machines.

9.13 Related Work

A number of other authors have looked at choreography definition and the rules required to ensure realizability. In this section we discuss related work and provide a commentary on how this work relates to this thesis. We focus on work on asynchronous choreographed collaborations.

Session Typing. Work by Honda et al. [40] addresses the question of realizability of asynchronous multiparty collaborations using a behavioural typing discipline called *Session Typing*. The scheme defines *global types*, which represent choreography; and *local types*, which correspond to our participant contracts. The local types are abstracted from the global type by a projection calculus, similar to that we describe in Chapter 6.4. The conditions we give for asynchronous projectability are captured in their notion of *linearity*, which aims to enforce the discipline that *in two communications, sending actions and receiving actions should respectively be ordered temporally, so that no confusion arises*. Linearity is established by a process called *causality analysis* based on the dependencies between the exchanges of the global type. This analysis requires that there are subsequences in the choreography that observe a similar discipline to that we require in relay machines.

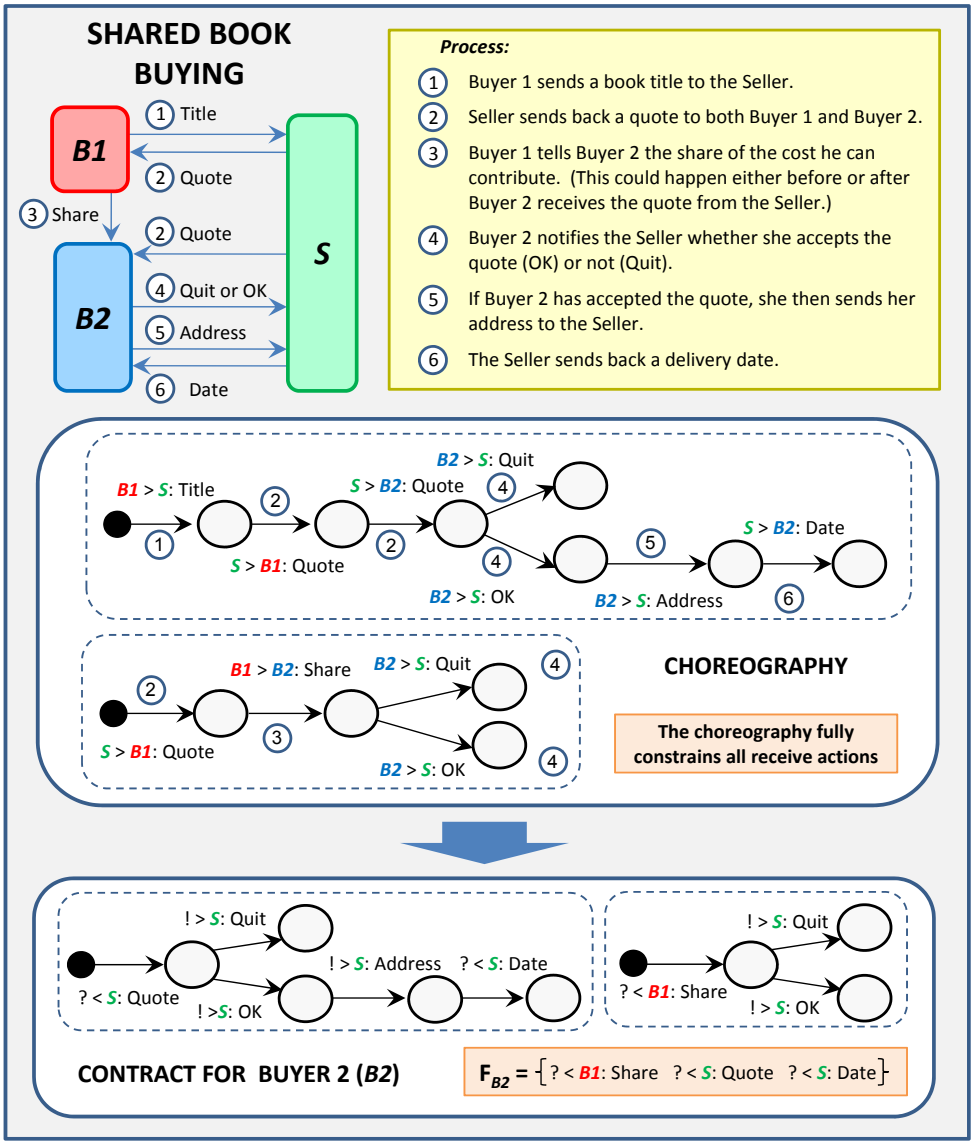


Figure 9.13: Shared Book Buying Example

Where there is indeterminacy of ordering in the global type, sequencing is imposed using a device called a *channel*. Consider the *Shared Book Buying Collaboration* example which is described in [40] and to which we show a protocol modelling solution in Figure 9.13. In the Session Typing solution the channel device is used to address the indeterminacy in the ordering of receipt of the *Quote* and *Share* messages by *B*2 by imposing an ordering: *Quote* followed by *Share*. This is not required in the protocol modelling solution, and it seems that channels are used to remove the natural concurrency of the collaboration which in our approach is expressed explicitly using composition

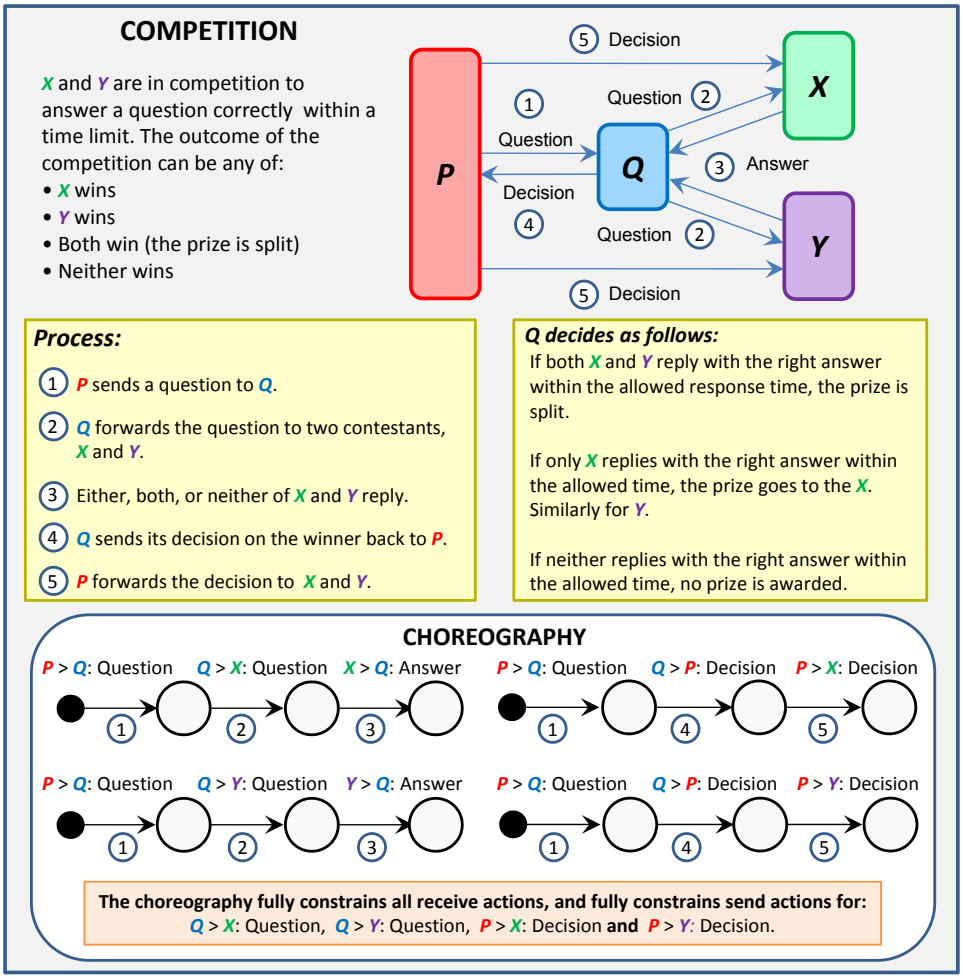


Figure 9.14: Competition Example

in the choreography and the extracted participant contracts. This means that the Session Typing approach, at least as described, is not able to describe collaborations that necessarily entail a race. An example is given in Figure 9.14 which describes a simple competition in which two contestants, X and Y, are asked a question and a prize is awarded for correct answers. A choreography expressed as four composed protocol machines is shown in Figure 9.14, and we note the following:

- The collaboration involves a race, because the sequencing of the receipt of answers from X and Y is unknown in advance and cannot be constrained by the choreography. This is handled using a composition of two machines in the choreography, shown one above the other at the bottom left of the figure, allowing the

ordering of receipt of the answers from X and Y to be unspecified.

- The provision of the *Decision* by Q back to P and then by P on to X and Y is depicted by the two machines at the bottom right of the figure. Note that it is possible for P to send notification of the decision to X or Y before they have responded, as it is possible that either or both failed to respond quickly enough.
- The four machines in the choreography are independent, relay, and deterministic when reduced to pairs. So the choreography is projectable and therefore realizable under our theory.

In attempting to create a global type for this collaboration we have a choice about how to express the sending of the question to X and Y and the subsequent receipt of their answers by Q. One possibility is to use two session types composed in parallel, following the same strategy that is used in Figure 9.14. However, while the syntax of global types allows parallel composition, the projection of a global type that involves parallel composition is undefined if a given participant is involved in both the types being composed, either as the sender or the receiver of a message, so this approach is excluded. The other possibility is to serialize the two exchanges whereby X and Y send their answers to Q in a single global type. In this case we need to address the question: should these two exchanges use the same or different channels? Neither choice appears to be satisfactory. If we choose to use different channels then P is forced to receive the answers in a defined order, as the receives on the channels must be serialized in P's logic. This clearly violates the intention of the collaboration. If we choose to use the same channel then the global type fails to obey the condition for *linearity*[8] (the Session Typing equivalent of our projectability condition) and this means that realizability is not guaranteed. As some collaborations, such as e-auctions, are likely to involve such intentional race conditions the difficulty of modelling this using Session Types seems a weakness.

Session Nets. The syntax used in Session Typing uses a linear syntax that has expressive limitations. This has been explored by Fossati et al. [25] in their paper on "Multiparty Session Nets" where they give the example shown at the top of Figure 9.15. As the authors point out, this flow cannot be represented by the global type syntax

[8]This is because a chain of two receipts on the same channel from different senders, as we would have here, is considered "non-causal" and makes the global type non-linear. See Honda et al. [40]

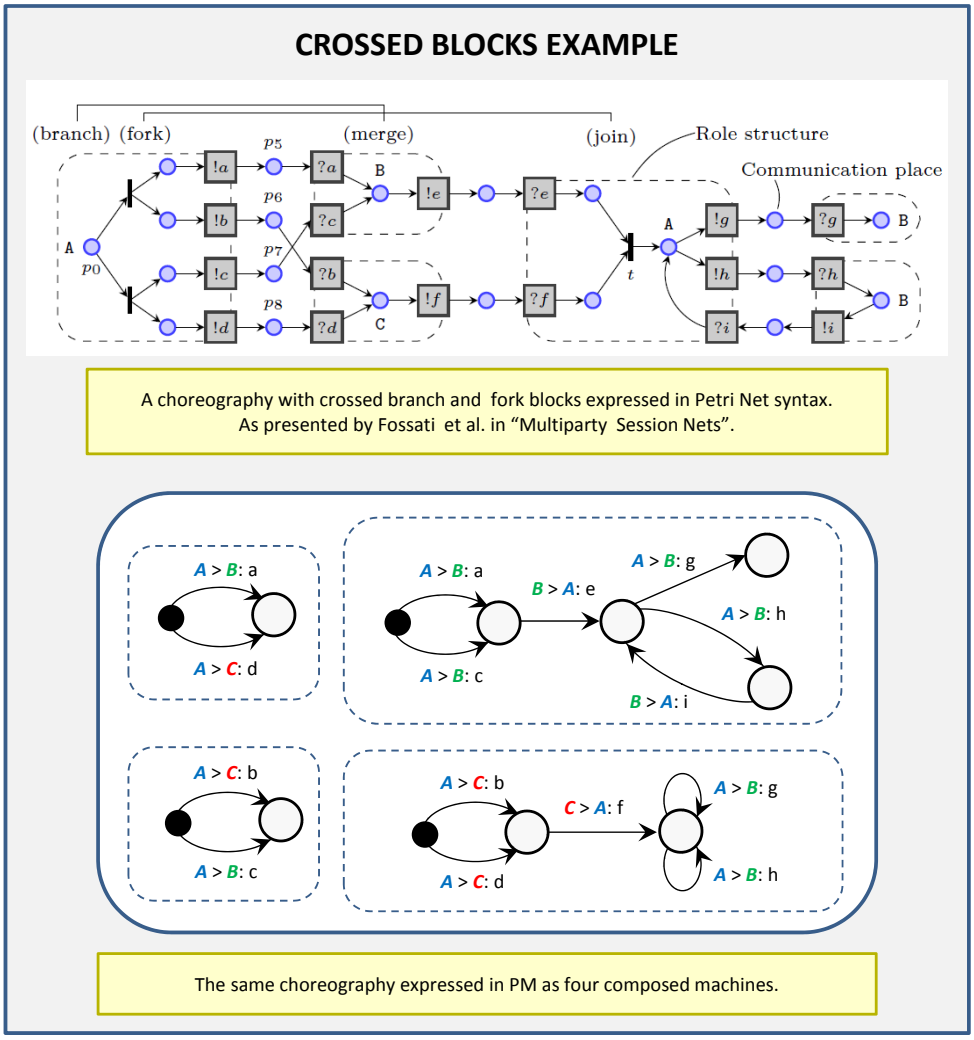


Figure 9.15: Crossed Blocks Example

of Session Typing - firstly because the "criss-crossing" of the middle two of the four paths from $p0$ to t cannot be expressed in the tree structure of a linear syntax; and secondly because each of these paths flows from the initial branch through a fork, but then goes through a merge before the join. This interleaving of choice (branch-merge) and parallel (fork-join) structures is not supported by the nesting of choice and parallel constructors imposed by standard global type syntax. In order to support choreographies with such flow structures the authors describe:

- Formation rules for choreographies expressed using Petri Net style notation displaying branch and fork structures that guarantee realizability. Flows con-

structed according to these rules are referred to as *well-formed nets*.

- A *conformance relation* between syntactic end-point types and well-formed nets which allows each end-point type to be validated against a net independently, while guaranteeing that their behaviour in composition respects the behaviour of the global net.

This approach extends the applicability the use of locally typed processes to a wider class of choreography. However the approach requires that parallel forks are expanded into sequential interleaved outputs in each branch in order render the ordering of sends expressible as a local type. However, if the parallel arms of a fork are serialised differently in different participants deadlock can result and this is addressed by requiring for conformance that outputs are prioritised over inputs. This has two arguably adverse consequences:

- No mechanism is given for deriving local types from the choreography, perhaps because the restructuring required would make it very complex. The paper does not address how end-point processes are obtained.

- The prioritisation rule can introduce unnecessary latency in execution. Suppose a participant cannot perform an output in one arm of a fork (e.g., because it is waiting for information from elsewhere, not part of the choreography) but an input arrives that can be received in the other arm. Serialisation with output priority will delay handling the input until the output can be completed. This delay is an artefact of the methodology as it is not required by the choreography.

The lower half of Figure 9.15 shows the choreography rendered in PM as four composed machines, and it is not difficult to show that the set of traces induced by each of the two representations are the same. Considering the PM representation, the four machines all have relay form and no repeated transition labels, so the choreography is realisable. End-point projections can be obtained from the PM choreography using reduction, and do not entail any serialisation of parallel arms of the forks so no unnecessary latency is incurred, so neither of the consequences identified above pertain.

IOC/POC. Work by Lanese et al. [48] defines an approach similar to that used in Session Typing, but without the use of channels. The scheme defines *Interaction-Oriented*

Choreography (IOC), which represents choreography; and *Process-Oriented Choreography* (POC), which corresponds to our participant contracts. A POC for each participant is abstracted from the IOC by a projection calculus. The authors give realizability conditions, which they call *connectedness conditions*, for both synchronous and asynchronous collaborations. These conditions correspond very closely to those we describe for projectability of a single stored-state choreography machine.

The IOC/POC scheme allows parallel composition of processes in a choreography and, unlike Session Typing, allows the same participants and message types to appear in different composed processes. This means that the IOC/POC scheme has the ability to model concurrency involving races, and could be used to express the choreography for the Competition Example in Figure 9.14 on page 193. However there is no concept of CSP style synchronization or of the use of data and derived states, so it is not possible to articulate a choreography as a set of separate partially synchronized descriptions. This means that the approach would not, as currently described, be able to describe a choreography that requires a combination of stored and derived state spaces such as the Instalments Example in Figure 9.10 on page 183 and, as published so far, the IOC/POC approach does not use data or computation at all.

Conversation Protocols. In their work, Fu et al. [26] describe an approach for the realization of asynchronous collaborations based on *Conversation Protocols*. A conversation protocol corresponds to a choreography and is projected to give peer (i.e., participant) behaviour. Both the choreography and the projected peer behaviours are modelled as Finite State Automata (FSA). The authors identify three conditions on a conversation protocol that must be met for realizability: *Lossless Join*, which requires that the protocol should be complete when projected to individual peers; *Synchronous Compatible*, which ensures that the protocol does not have illegal states; and *Autonomy*, which implies that at any point in the execution each peer is determined on the choice to send, or to receive, or to terminate. Although stated in different terms, these correspond quite closely to our asynchronous projectability conditions. The Autonomy condition allows more than one peer (participant) to be in send mode at the same point in the choreography, so is more relaxed than our projectability condition (9.16a) which limits sending to a single participant. This more relaxed condition is balanced by the stricter requirement of Lossless Join to ensure realizability. As we point out in Section 9.8,

different machines in our compositional approach can have different senders and this gives our approach equivalent expressive power to Conversation Protocols.

This formalism used for Conversation Protocols does not use parallel composition at all. This does limit expressive power. For instance, if:

$$(P{>}Q{:}a) \parallel (Q{>}P{:}b)$$

is expressed as a Conversation Protocol (i.e., with no composition) then it violates Autonomy as P and Q are both sending and receiving in the initial state of the choreography. However both components are clearly asynchronously projectable, so the choreography is realizable under our theory. The same issue would arise in attempting to model the Order Processing choreography in Figure 9.2 on page 155 as a Conversation Protocol as, for instance, there is a state where the *Customer* can send a *Request Cancel* and also receive an *Invoice*.

In later work, [27], the authors have added some capability to model data and computations. This is done by using *Guarded* Finite State Automata (GFSA) for both choreography and projected peer behaviours. The general idea is similar to that which we describe, with data and computation being used to specify rules in the choreography. However, the scheme is significantly less ambitious than ours in that the guard of a transition only expresses the relationships between the message that is being sent and the last message of each class sent or received by the sender; so there is no notion of a choreography owning and maintaining attributes as we have. Updates are confined to changes to message fields and it would not be possible to compute a guard condition on accumulated amounts[9] such as is required for the Instalments Example in Figure 9.10 on page 183. The authors describe an algorithm for checking realizability of a choreography described as a GFSA, and this bears some similarities to the technique described in Section 9.8 but a detailed comparison has not been undertaken.

BPMN2. The *Business Process Model and Notation* (BPMN2) [65] developed by the Object Management Group (OMG) is the latest in a number of attempts to create an industry standard language to describe choreography[10]. The new standard is notable in

[9] At least not without including such amounts as extra fields in the messages. But message content is normally fixed by business considerations or standards, and not open to this kind of change.

[10] Previous attempts include WSCI and WS-CDL.

that it is the first to our knowledge to attempt to include rules for realizability, with the following statement:

> *There are constraints on how Choreography Activities can be sequenced (through Sequence Flow) in a Choreography ... The basic rule of Choreography Activity sequencing is this:*
>
> - *The Initiator of a Choreography Activity MUST have been involved (as Initiator or Receiver) in the previous Choreography Activity.*
>
> **– Object Management Group [65]** ***page 336***

This partly captures our idea of relay form, but is clearly an incomplete treatment of the conditions for realizability.

The BPMN choreography language has no compositional capability of the kind discussed and advocated in this thesis.

9.14 Necessary Conditions for Realizability

All of the treatments of choreography catalogued in the previous section concern themselves with establishing **sufficient** conditions for a choreography to be realizable, and this is also the case for the results presented for PM. In practical terms this allows for false negatives, because a choreography that is, in fact, useful and realizable does not meet the sufficiency criteria required for realizability. This invites the question, what are **necessary** conditions for realizability? Some recent work has addressed this question.

Establishing necessary conditions for realizability enables the determination that a given choreography is *not* realizable, and one conceivable approach to this is exhaustive exploration of possible enactments using model checking. The problem is that model checking on collaborations with unbounded queues between participants is not decidable as the enactment state space is infinite. However in their paper on the decidability of choreography realizability [8], Basu et al. establish that choreography realizability is guaranteed if enactment of projected participant behaviours **connected by queues limited to a single message** is language equivalent to the choreography,

where language equivalence here is based on the possible sequences of sends. Since both the choreography and the enactment with limited queues have finite state spaces, it is possible to use standard model checking tools to establish the required language equivalence between them. This result provides a means of establishing that a choreography is not realizable, by establishing that it fails the language equivalence test, albeit by exhaustive exploration of the state space of the bounded enactments.

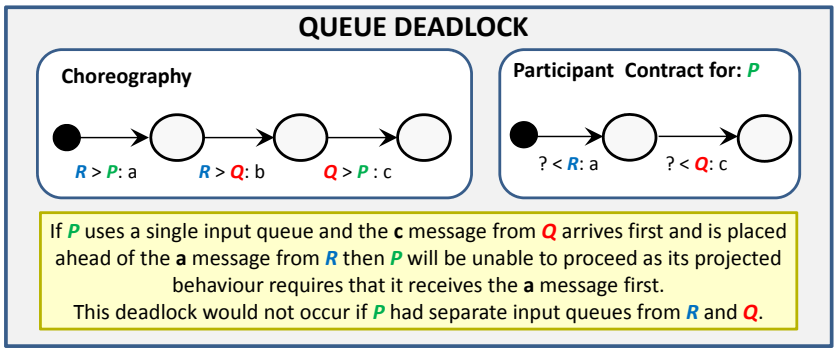


Figure 9.16: Queue Deadlock

The communication model adopted by Basu et al. in this work has a single input message queue for each participant, in contrast to model used in this thesis where each participant has an input queue for each other participant. The choice we have made is deliberate, as using a single queue gives the possibility of "queue deadlock" as shown in Figure 9.16, so we do not think the single queue model is appropriate. However, the proof techniques used by Basu et al. rely on their assumed model, as their model enables departure of an enactment from its choreography to be tied to a single message: the one at the front of the queue for some participant at the point where the departure occurs. It is the ability to tie departure to a single message that allows such departure to be replicated in a bounded system where only one message is allowed in a queue. In our model a participant may have a number of messages visible to it via its multiple input queues, and an enactment's departure from its choreography results from making an incorrect choice on which to process. The context of an incorrect choice is thus constituted of both the "correct" message that was not chosen and the "incorrect" one that was, so cannot be associated with a single message. This means that the result proved by Basu et al. does not carry through to the work described in this thesis.

It is interesting to speculate on whether it is possible to formulate necessary conditions for realizability using the multiple input queue model we have used; and also whether conditions can be formulated that do not rely on exhaustive search of the enactment state space for evaluation. One possibility to attempt to adapt their reasoning to our communication model, but this does not appear straightforward. Another approach is to ask whether any realizable choreography expressed as a single PM machine must be decomposable into a composition of relay machines (as defined in Section 9.6.3 on page 169). If so, being expressible as a composition of relay machine becomes a necessary condition for realizability; and while this does not furnish a complete answer, it is a significant part of it. A possible attack on this question could be along the lines sketched below.

Suppose that a choreography C is expressed as a single LTS. We assert that, if C is realizable, every path of C must be an interleaving of relay "threads", such a thread being a sequence of transitions where the sender in each transition (apart from the first) is either the sender or the receiver of the previous transition. The justification for this assertion can be seen as follows.

First, we identify the possible *starter actions*, those that can start off the collaboration:

$$starters_C = \{\, label(\theta) \mid \theta \in \Theta_C \wedge begin(\theta) = \bullet \,\}$$

Now assume that each participant starts with a Lamport clock set to zero, and consider any transition θ in C with $label(\theta) \notin starters_C$. Because its action cannot be the first, in any enactment θ must carry a Lamport timestamp greater than zero. As only a relay thread can support strictly increasing Lamport timestamps, any path in C that contains θ must have a relay subsequence that contains θ but doesn't start with θ. This identifies relay predecessors for θ. This argument is then applied recursively to the predecessors until an action in $starters_C$ is reached. By this argument every transition, unless it is for a starter action, is in a relay thread.

If all the paths in C are interleaved relay threads, it may be possible to re-express C as a composition of relay machines. We do this first under the simplifying assumption that no two transitions in C share the same label:

$$\forall\, \theta_1, \theta_2 \in \Theta_C : \quad label(\theta_1) = label(\theta_2) \Rightarrow \theta_1 = \theta_2 \tag{9.28}$$

The significance of this assumption is discussed at the end of this section.

We define the set $reductions(C)$ as the set of all exact reductions of C, so for each member $x \in power(labels(\Theta_C))$ there is a member of $reductions(C)$ that is C reduced to retain only those labels in x.

We now construct the machine C^{Π} as follows:

$$C^{\Pi} = \prod_{M \in reductions(C) \wedge relay(M)} M \tag{9.29}$$

and the contention then is that $C^{\Pi} = C$. This contention is supported by the following observations:

- Every relay thread in C is preserved in at least one machine in the composition C^{Π}. This means that every interleaving of relay threads in C is present in C^{Π} and so C^{Π} contains every realizable path in C.
- Every choice made by C must be made as a choice by one participant between two or more sends in one state of the choreography. Because the transitions that represent the different choices have the same sender they conform to relay form and will all be present in at least one machine in the composition C^{Π}. This means that C^{Π} reflects every choice that may be made by a participant in enactment.
- Because exact reduction is used and, as discussed in Section 6.4.2 on page 116, this technique cannot introduce new traces, there is no relay thread in C^{Π} that is not present in C.

This reasoning requires formal verification, and this is identified in the next section as further work required.

The reason for the assumption (9.28) is to ensure that all relay threads are preserved. Without it the reduction of C to the labels that occur on a given relay thread may not be a relay machine, as some subset of these labels also occur in a non-relay relationship elsewhere in C causing the reduction to be non-relay. This can only happen if a given label can appear on more than one transition, so (9.28) ensures that all relay threads are preserved. If a choreography does not meet (9.28) it can be made to do so by renaming

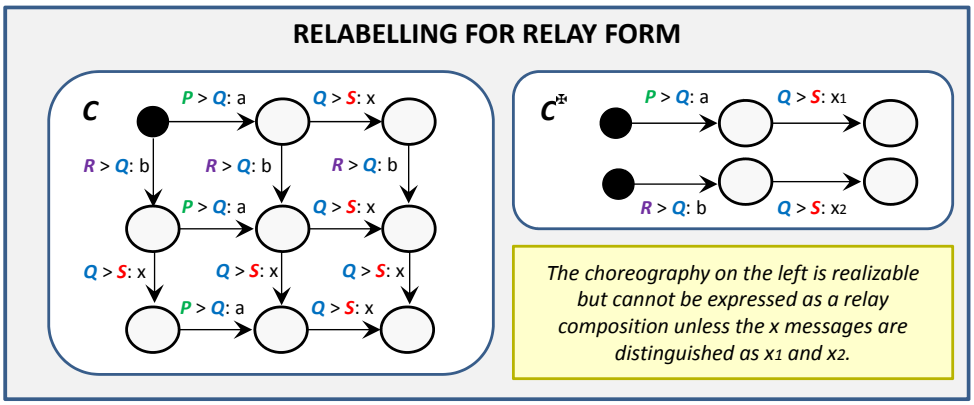


Figure 9.17: Relabelling for Relay Form

message types. So if, for instance, the label:

```
Cust>Supp:Cancel
```

appears on two transitions these can be relabelled:

```
Cust>Supp:Cancel1   and   Cust>Supp:Cancel2
```

Figure 9.17 shows an example of a realizable choreography that requires relabelling of transitions to enable expression as a relay composition. Note that relabelling fully within the choreography C will generate more versions of the labels than is shown in $C^{✠}$, but after refactoring to remove equivalent labels the result is as shown.

The ability to re-express any realizable choreography as a relay composition after suitable relabelling of transitions means that, at least up to message renaming, this form is necessary for realizability.

9.15 Conclusions and Further Work

We claim that the approach to choreography described in this thesis makes a significant advance over other proposals in three respects:

- We use *composition* in the articulation of a choreography, so that a single choreography can be expressed as a composition of partially synchronized descriptions. This gives our approach the expressive power to model cases where the ordering and/or certainty of message receipt is unknown in advance and cannot be

constrained by the choreography. These kinds of collaboration are hard or impossible to handle without composition.

- We demonstrate that realizability of a choreography defined as a composition only needs to be established *individually for the components of the composition*. In other words, *composition preserves realizability*. Determining realizability for a single component is easier than analyzing the choreography as a whole, thus simplifying the analysis.

- We allow the use of *data and computation* in the definition of a choreography, so that the decision on whether it is permissible to engage in a message exchange can be determined by the result of a calculation. This is in contrast with most other published approaches, which only guarantee realizability on the basis of patterns of message exchange.

If sufficiently simple, powerful and robust conditions on the form of a choreography can be formulated, along with a simple procedure for extracting participant behaviour definitions, then designing extended collaborations can become routine. If the choreography is built to obey the rules, the resulting collaboration is bound to work.

We believe that the development of advanced distributed collaborations will require techniques that enable designers to construct solutions that are transparently correct so that, as far as possible, realizability is guaranteed. The modelling formalisms used to represent choreographies and the resultant behaviour contracts must play the key role in this, and the use of compositional modelling appears to be the natural paradigm to express and analyse the inherent parallelism of distributed behaviour.

If choreography-based approaches are to succeed they must provide the expressive power to capture real business collaborations, but also allow realizability to be verified using techniques within the competence of mainstream software engineers. We believe that the ideas we discuss here take us significantly nearer achieving this combination and we hope that this work will help inform future directions in the design of choreography modelling languages. In particular, we suggest that *composition capabilities* should be at the heart of such languages.

A number of lines of further work are possible which we discuss here.

Expressive Power. As was noted in the discussion of the flow depicted in Figure 9.15 on page 195, some of the choreography languages that have been proposed have limitations in their expressive power. To assess where PM stands a number of questions would need to be addressed:

- How should "expressive power" be measured? Could use be made of patterns, such as the workflow pattern catalogue compiled by van der Aalst et al. [77]?
- How does the expressive power of PM compare with other choreography languages? Are there any limitations that would impair its use or value?

Developments in Choreography. There is potential for PM to contribute to new lines of research in choreography. Particular opportunities here are:

- Can the ideas sketched in Section 9.14 lead to a formulation of necessary conditions for choreography realizability that does not require model-checking techniques (exhaustive exploration of the state space) for evaluation?
- Rather than extracting participant behaviours from a choreography it is potentially possible to do the reverse, forming choreography by combining local behaviours. This is known as "synthesis" and has been explored for instance by Lange et al. [50] who use an automata model for building a choreography from a (finite) set of communicating automata expressing local behaviours. Would it be possible to use PM formalisms to reproduce this and extend it with the data dimension?

Distributed Workflow. The author has explored the use of PM to model workflow [54], using a form of deontic modality to model "motivation" in workflow and establish conditions under which a distributed, choreography based, workflow will always progress to completion in finite time. There are a number of ways in which this work can be taken further, by addressing the following questions:

- What possibilities are there to enhance the power of the progress analysis by combining topological reasoning (based on analysis of the state/transition space) with non-topological reasoning?
- How would PM workflow models map to commonly used workflow and collaboration infrastructures?

Appendices

Appendix A

Notations and Utility Functions

A.1 Notations

A.1.1 Notations for Sets

If X is a set and $\mathbb{Y}$ is a set of sets we use $X \Subset \mathbb{Y}$ to mean that there is an member of $\mathbb{Y}$ containing X. So:

$$X \Subset \mathbb{Y} \Leftrightarrow \exists\, Y \in \mathbb{Y} \text{ with } X \subseteq Y \tag{A.1}$$

Similarly we use $X \Subset\!\!\!\!/\ \mathbb{Y}$ to mean that there is no member of $\mathbb{Y}$ that contains X.

We also use $\Subset$ on a set of sets:

$$\mathbb{X} \Subset \mathbb{Y} \quad \Leftrightarrow \quad \forall\, X \in \mathbb{X} :\ X \Subset\ \mathbb{Y} \tag{A.2}$$

We use $\Cup$ to denote union of a set of sets, so:

$$\Cup \mathbb{X} = \bigcup_{X \in \mathbb{X}} X \tag{A.3}$$

and similarly $\Cap$ to denote intersection of a set of sets, so:

$$\Cap \mathbb{X} = \bigcap_{X \in \mathbb{X}} X \tag{A.4}$$

We use $power(X)$ for the power set of X:

$$power(X) = \{ x \mid x \subseteq X \} \tag{A.5}$$

A.1.2 Notations for Sequences

We use:

- X^* for the set of finite sequences of elements from the finite set X.
- X^∞ for the set of all sequences (both finite and infinite) of elements from the finite set X.
- $\frown$ for concatenation. So if t is a sequence of items and s is an item then $t \frown s$ is the sequence obtained by appending s to the end of t. Similarly, if t_1 and t_2 are two sequences, then $t_1 \frown t_2$ is their concatenation.
- t_i is the i^{th} item of the sequence t, numbering starting at 1.

We use:

- $t' \subseteq t$ to mean that t' is a subsequence of t, so $t'_i = t_{n_j}$ where $1 \leqslant n_1 < n_2 \cdots \leqslant length(t)$ is an increasing sequence of indices.
- $t' \subset t$ to mean that t' is a proper subsequence of t, so $t' \subseteq t$ and $t' \neq t$.

A.2 Utility Functions on Sequences

Fn: *length*. To obtain the length of sequence we define the function *length* with signature:

$$length :: X^* \to \mathbb{N}$$

which returns the length of an element of X^*. For instance, $length(<>) = 0$.

Fn: *last*. To return the last element of a non-empty finite sequence we define the function *last* with signature:

$$last :: X^* \backslash \{<>\} \to X$$

If $t \in X^*$ then $last(t) = t_{length(t)}$.

Fn: *prefixes* and *stems*. A *prefix* of a sequence t is any finite sequence that can be extended by successive concatenation zero or more times to the right to equal t. A *stem* of a sequence t is a finite **proper** prefix of t; so a stem is always shorter than the sequence it prefixes. If X is some set of elements then we define functions *prefixes* and *stems* with signatures:

$$prefixes :: X^\infty \to power(X^*)$$
$$stems :: X^\infty \to power(X^*)$$
$$prefixes :: power(X^\infty) \to power(X^*)$$
$$stems :: power(X^\infty) \to power(X^*)$$

as:

$$prefixes(t) = \{\ t' \mid length(t') \leqslant length(t) \ \wedge\ \forall\, i \text{ with } 1 \leqslant i \leqslant length(t') :\ t'_i = t_i\ \}$$
$$stems(t) = \{\ t' \mid length(t') < length(t) \ \wedge\ \forall\, i \text{ with } 1 \leqslant i \leqslant length(t') :\ t'_i = t_i\ \}$$
$$prefixes(S) = \bigcup_{t \in \mathcal{T}} prefixes(t)$$
$$stems(S) = \bigcup_{t \in \mathcal{T}} stems(t)$$

Fn: *trunc*. To remove the last item from a finite sequence we define the function *trunc* with signature:

$$trunc :: X^* \backslash \{<>\} \to X^*$$

as:

$$trunc(t) = t' \text{ where}$$
$$length(t') = length(t) - 1 \ \wedge\ \forall\, i \text{ with } 1 \leqslant i \leqslant length(t') :\ \ t'_i = t_i$$

Fn: *asSet*. To recover the underlying set from a sequence we define functions *asSet* with signatures:

$$asSet :: X^\infty \to X$$
$$asSet :: power(X^\infty) \to X$$

as:

$$asSet(t) = \{e \mid \exists\, i \text{ with } 1 \leqslant\ i \leqslant\ length(t) \text{ and } e = t_i\}$$
$$asSet(S) = \bigcup_{t\, \in\, S} asSet(t)$$

Fn: *filter*. To filter a sequence to those items for which some predicate on items is true we define the function *filter*. Given a set X and a predicate $\phi(e)$ defined on all members of X, we define functions *filter* with signatures:

$$filter_\phi :: X^\infty \to X^\infty$$
$$filter_\phi :: power(X^\infty) \to power(X^\infty)$$

as:

$$filter_\phi(<>) = <>$$
$$filter_\phi(t^\frown e) = filter_\phi(t)^\frown e \qquad \text{if } (\phi(e))$$
$$filter_\phi(t^\frown e) = filter_\phi(t) \qquad \text{otherwise}$$

and:

$$filter_\phi(S) = \{\, filter_\phi(t) \mid t \in S \,\}$$

A.3 Utility Functions on Observations

Fn: *symb* and *val*. To extract the set of symbols used in a set of observations we define functions *symb* with signatures:

$$symb :: \mathcal{E} \to power(\mathcal{Y})$$
$$symb :: power(\mathcal{E}) \to power(\mathcal{Y})$$
$$symb :: power(power(\mathcal{E})) \to power(\mathcal{Y})$$

as:

$$symb(w) = \{w^{symb}\}$$
$$symb(W) = \bigcup_{w \in W} symb(w)$$
$$symb(\mathbb{W}) = \bigcup_{W \in \mathbb{W}} symb(W)$$

The function *val* works in a similar way on the value parts of observations.

Fn: *restr*. To restrict observations to those that use a given set of symbols we define functions *restr* with signatures:

$$restr :: power(\mathcal{E}) \times power(\mathcal{Y}) \to power(\mathcal{E})$$
$$restr :: power(\mathcal{E})^* \times power(\mathcal{Y}) \to \mathcal{E}^*$$
$$restr :: power(power(\mathcal{E})) \times power(\mathcal{Y}) \to power(power(\mathcal{E}))$$

as:

$$restr(W, Y) = \{ w \mid w \in W \quad \wedge \quad w^{symb} \in Y\}$$
$$restr(W^*, Y) \text{ has } length(restr(W^*, Y)) = length(W^*) \text{ and}$$
$$restr(W^*, Y)_i \;=\; restr(W^*{}_i, Y)$$
$$restr(\mathbb{W}, Y) = \{ restr(W, Y) \mid W \in \mathbb{W}\}$$

A.4 Utility Functions on Steps

Fn: *actions*. To extract the action parts from a structure of steps we define functions *actions* with signatures:

$$actions :: \mathcal{S}^* \rightarrow power(\mathcal{E})^*$$
$$actions :: power(\mathcal{S}^*) \rightarrow power(power(\mathcal{E})^*)$$
$$actions :: power(\mathcal{S}) \rightarrow power(power(\mathcal{E}))$$

as:

$$actions(t) \text{ has } length(actions(t)) = length(t) \text{ and } actions(t)_i \;=\; t_i{}^{\alpha}$$
$$actions(\mathcal{B}) = \{actions(t) \mid t \in \mathcal{B}\}$$
$$actions(S) = \{s^{\alpha} \mid s \in S\}$$

Fn: *perceives* and *offers*. Corresponding functions are defined for *perceives* and *offers*.

Fn: *decisions*. To extract the decision parts from a structure of steps we define functions *decisions* with signatures:

$$decisions :: \mathcal{S}^* \rightarrow \{allow, refuse, crash\}^*$$
$$decisions :: power(\mathcal{S}^*) \rightarrow power(\{allow, refuse, crash\}^*)$$
$$decisions :: power(\mathcal{S}) \rightarrow power(\{allow, refuse, crash\})$$

as:

$$decisions(t) \text{ has } length(decisions(t)) = length(t) \text{ and } decisions(t)_i \;=\; t_i{}^{\delta}$$
$$decisions(\mathcal{B}) = \{decisions(t) \mid t \in \mathcal{B}\}$$
$$decisions(S) = \{s^{\delta} \mid s \in S\}$$

Fn: *symb*. To extract the set of symbols used in a structure of steps we define functions *symb* with signatures:

$$symb :: \mathcal{S} \to power(\mathcal{Y})$$
$$symp :: power(\mathcal{S}) \to power(\mathcal{Y})$$
$$symb :: \mathcal{S}^* \to power(\mathcal{Y})$$
$$symb :: power(\mathcal{S}^*) \to power(\mathcal{Y})$$

as:

$$symb(s) = symb(s^\tau)$$
$$symb(S) = \bigcup_{s \in S} symb(s)$$
$$symb(t) = \bigcup_{1 \leqslant i \leqslant length(t)} symb(t_i)$$
$$symb(\mathcal{B}) = \bigcup_{t \in \mathcal{B}} symb(t)$$

Fn: *restr*. To restrict a structure of steps to use a given set of symbols we define functions *restr* with signatures:

$$restr :: \mathcal{S} \times power(\mathcal{Y}) \to \mathcal{S}$$
$$restr :: \mathcal{S}^* \times power(\mathcal{Y}) \to \mathcal{S}^*$$
$$restr :: power(\mathcal{S}^*) \times power(\mathcal{Y}) \to power(\mathcal{S}^*)$$

as:

$$restr(s, Y) = step(restr(s^\alpha, Y), restr(s^\pi, Y), s^\delta, restr(s^\omega, Y))$$
$$restr(t, Y) \text{ has } length(restr(t, Y)) = length(t) \text{ and } restr(t, Y)_i = restr(t_i, Y)$$
$$restr(\mathcal{B}, Y) = \bigcup_{t \in \mathcal{B}} restr(t, Y)$$

A.5 Utility Functions on Machines

Fn: *next*$_P$. To give the set of steps following a given prefix we define the function $next_P$ with signature:

$$next_P :: prefixes(\mathcal{B}_P) \to power(asSet(\mathcal{B}_P))$$

as:

$$next_P(t) = \{\, s \mid s \in asSet(\mathcal{B}_P) \;\wedge\; t^\frown s \in prefixes(\mathcal{B}_P)\}$$

Note that if P is a protocol machine then $t^\delta \neq allow \;\Rightarrow\; next_P(t) = \emptyset$.

Fn: $completions_P$. To obtain the set of completions that all have a given prefix we define the function $completions_P$ with signature:

$$completions_P :: prefixes(\mathcal{B}_P) \to power(\mathcal{B}_P)$$

as:

$$completions_P(t) = \{\, t' \mid t' \in \mathcal{B}_P \wedge t \in prefixes(t') \,\}$$

Fn: $matches_P$. To obtain the prefix of a protocol machine that matches a given sequence of universe elements we define the function $matches_P$ with signature:

$$matches_P :: \mathbb{U}^* \to prefixes(\mathcal{B}_P)$$

as:

$$matches_P(U^*) =$$
$$\begin{array}{ll} <> & \text{if } (U^* = <>) \\ matches_P(trunc(U^*)) & \text{if } (next_P(matches_P(trunc(U^*))) = \emptyset) \\ matches_P(trunc(U^*))^\frown s & \text{otherwise} \end{array}$$
$$\text{where} \quad s \in next_P(matches_P\,(trunc(U^*))) \quad \text{and} \quad s^\tau \subseteq last(U^*)$$

Note that $matches_P$ is only defined if P is a protocol machine so that $\mathcal{B}_P$ obeys (4.10). When this is the case (4.10c) guarantees that the step s used in this definition always exists and (4.10d) guarantees that it is unique.

Fn: $zip_{\mathfrak{U}}$. To create a sequence of steps from three elements:

- A set of symbols that are offered by the steps
- A sequence of elements from the universe $\mathfrak{U}$
- A sequence of decisions

we define the function $zip_{\mathfrak{U}}$ with signature:[1]

$$zip_{\mathfrak{U}} :: power(\mathcal{U}) \times \mathbb{U}^* \times \{allow, refuse, crash\}^* \to \mathcal{S}^*$$

as:

$$length(zip_{\mathfrak{U}}(\Omega, U^*, \delta^*)) = min(length(U^*), length(\delta^*)) \quad \text{and}$$
$$\forall\, i \text{ with } 1 \leqslant i \leqslant length(zip_{\mathfrak{U}}(\Omega, U^*, \delta^*)) :$$
$$zip_{\mathfrak{U}}(\Omega, U^*, \delta^*)_i = step(restr(U^*{}_i, \mathcal{A}), restr(U^*{}_i, \mathcal{U} \backslash (\mathcal{A} \cup \Omega)), \delta^*{}_i, restr(U^*{}_i, \Omega))$$

A.6 Utility Functions on Topological Representations

Fn: *begin, end.* To obtain the beginning state from a structure of transitions we define functions *begin* with signatures:

$$begin :: \Theta \to \Sigma$$
$$begin :: \Theta^\infty \to \Sigma$$

as:

$begin(\theta)$ extracts the first component of θ, the begin state

$$begin(t) = begin(t_1)$$

and similarly for *end*.

Fn: *label.* To obtain the label(s) from a structure of transitions we define functions *label* and *labels* with signatures:

$$label :: \Theta \to \Lambda$$
$$labels :: \Theta^\infty \to \Lambda^\infty$$
$$labels :: power(\Theta^\infty) \to power(\Lambda^\infty)$$
$$labels :: power(\Theta) \to power(\Lambda)$$

The first extracts the middle component of an identifier, the label on the transition. The remainder are defined respectively as follows:

$$\forall\, i \text{ with } 1 \leqslant i \leqslant length(t) :\ labels(t)_i \ = \ label(t_i)$$
$$labels(\mathcal{I}) = \{labels(t) \mid t \in \mathcal{I}\}$$
$$labels(S) = \{label(s) \mid s \in S\}$$

[1] The name *zip* is intended to capture the idea that the function "zips" together two sequences, one of universe elements and one of decisions.

Fn: Π. To obtain the set of all paths starting at a given state we define functions Π with signatures:

$$\Pi_P :: \Sigma_P \rightarrow power({\Theta_P}^{\infty})$$
$$\Pi_P :: \Sigma_P \times \Sigma_P \rightarrow power({\Theta_P}^{\infty})$$

The first defines the set of, possibly infinite, paths starting at a given state σ:

$$\Pi_P(\sigma) = \{\pi \mid \forall\, i \text{ with } 1 \leqslant i \leqslant length(\pi) :\ \pi_i \in \Theta_P \ \ \wedge \ \ begin(\pi_1) = \sigma \ \ \wedge$$
$$\forall\, i \text{ with } 1 \leqslant i < length(\pi) :\ begin(\pi_{i+1}) = end(\pi_i)\}$$

The second defines the set of, possibly infinite, paths between states σ and σ':

$$\Pi_P(\sigma, \sigma') = \{\pi \mid \pi \in \Pi(\sigma) \wedge end(\pi) = \sigma'\}$$

Fn: $\overrightarrow{\Pi}$. To obtain the set of all finite paths starting at a given state but unable to progress beyond a return to a state already visited we define functions $\overrightarrow{\Pi}$ with signatures:

$$\overrightarrow{\Pi}_P :: \Sigma_P \rightarrow power({\Theta_P}^*)$$
$$\overrightarrow{\Pi}_P :: \Sigma_P \times \Sigma_P \rightarrow power({\Theta_P}^*)$$

as:

$$\overrightarrow{\Pi}_P(\sigma) = \{\pi \mid \pi \in \Pi_P(\sigma) \text{ and}$$
$$\forall\, i, j \text{ with } 1 \leqslant i, j \leqslant length(\pi) \text{ and } begin(\pi_i) = begin(\pi_j) :\ i = j\ \}$$

and:

$$\overrightarrow{\Pi}_P(\sigma, \sigma') = \{\pi \mid \pi \in \Pi_P(\sigma, \sigma') \cap \overrightarrow{\Pi}_P(\sigma)\ \}$$

Fn: $\overrightarrow{\Pi}^{\mathbf{m}}$. To obtain the set of maximal finite paths starting at a given state but unable to progress beyond a return to a state already visited we define the function $\overrightarrow{\Pi}^m$ with signature:

$$\overrightarrow{\Pi}^m_P :: \Sigma_P \rightarrow power({\Theta_P}^*)$$

as:

$$\overrightarrow{\Pi}^m_P(\sigma) = \{\pi \mid \pi \in \overrightarrow{\Pi}_P(\sigma) \wedge \ \pi \notin stems(\overrightarrow{\Pi}_P(\sigma))\ \} \tag{A.7}$$

Fn: **visits**. To obtain the set of states visited by a given path we define the function *visits* with signature:

$$visits :: \Theta^{\infty} \rightarrow power(\Sigma)$$

as:

$$visits(\pi) = \bigcup_{\theta \in asSet(\pi)} (\{begin(\theta)\} \cup \{end(\theta)\})$$

Appendix B

Proofs

B.1 Normal Form in *fixes* is unique

Theorem B.1. We show that the normal form of fixes is unique. This is done by showing that any symbol used in normal form representation must be used in any representation.

Proof. We assume that a universe $\mathfrak{U}$ that is well-formed by (4.7) has two fixes defined, *fixes* and *fixes'*, that purport to give the same derivations but possibly have different representations.

If *fixes* and *fixes'* give the same derivations, we must have:

$$\begin{aligned} &\forall\, v, v' \in \mathit{restr}(\mathcal{E}, \mathcal{D}) \text{ with } v^{\mathit{symb}} = v'^{\mathit{symb}} \text{ and } V \in \mathit{fixes}(v), V' \in \mathit{fixes'}(v') \;: \\ &\quad \mathit{con}(V \cup V') \Rightarrow v = v' \end{aligned} \tag{B.1}$$

for otherwise there would be an element $U \in \mathbb{U}$ with $(V \cup V') \subseteq U$ for which *fixes* and *fixes'* give different derivations (fixes) for the same symbol.

We assume that *fixes* and *fixes'* obey (B.1) and that *fixes* is in normal form. Further we assume that in *fixes* a certain derived observation v has a fix V that uses a symbol y, so $y \in \mathit{symb}(V)$, but that the equivalent fix in *fixes'* does not use y. We show that this violates (B.1), giving a contradiction.

As $\mathit{fixes}(v)$ is in normal form, y cannot be removed by fix reduction. This means that there must be some $v^\dagger \neq v$ with ${v^\dagger}^{\mathit{symb}} = v^{\mathit{symb}}$ and some $V^\dagger \in \mathit{fixes}(v^\dagger)$ with $y \in \mathit{symb}(V^\dagger)$ so that:

$$\mathit{con}((V \cup V^\dagger) \backslash (\mathit{restr}(V^\dagger, \{y\}))) \quad \text{but} \quad \neg\mathit{con}(\mathit{restr}(V \cup V^\dagger, \{y\})) \tag{B.2}$$

for if this were not the case changing the value of y in any element of $\mathit{fixes}(v)$ would still fix to v and so y could be removed from $\mathit{fixes}(v)$ by fix reduction.

We now note that there must be an element V' of fixes' that fixes a value for v^{symb} and has:

$$\mathit{con}(V' \cup ((V \cup V^\dagger) \backslash (\mathit{restr}(V^\dagger, \{y\})))) \tag{B.3}$$

for if not there would be a $U \in \mathbb{U}$ where fixes' does not fix any value for v^{symb} in violation of (4.7b). Moreover because V' obeys (B.3) we know that $\mathit{con}(V \cup V')$, and as we assume fixes and fixes' obey (B.1) this means that $V' \in \mathit{fixes}'(v)$.

By assumption $y \notin \mathit{symb}(V')$ and hence $\mathit{con}(V^\dagger \cup V')$. However, $V^\dagger$ fixes $v^\dagger$ and V' fixes v and $v \neq v^\dagger$. So fixes and fixes' do not obey (B.1) and cannot give the same derivations. This means that y must be used to fix v in both fixes and fixes'. If fixes and fixes' use the same symbols it is not possible to construct a normal form different from fixes so normal form is unique. □

B.2 Normal Form in *machines* is unique

Theorem B.2. We show that the normal form of machines is unique. This is done by showing that any symbol used in normal form representation must be used in any representation.

Proof. We assume that $P \cong Q$ and $t \in (\mathit{prefixes}({\mathcal{B}_P}^{\mathit{norm}}) \cap \mathit{prefixes}({\mathcal{B}_Q}^{\mathit{norm}}))$ with possibly $t = <>$, and that P is in normal form. Moreover we assume that a step $sp \in \mathit{next}(t)$ of P uses a symbol y but that the equivalent step of Q does not use y. We show that this violates the assumption that $P \cong Q$, giving a contradiction.

Suppose that $sp' \in next_P(t)$ with $y \in symb(sp) \cap symb(sp')$. Because P is in normal form, y is required to distinguish behaviours, so we must have $U^*, U^{*\prime} \in \mathbb{U}^*$ with:

$$t^\frown sp \in prefixes(matches_P(U^*)) \quad \text{and} \quad t^\frown sp' \in prefixes(matches_P(U^{*\prime})) \tag{B.4a}$$

and, using $k = length(t) + 1$, also:

$$\begin{aligned} 1 \leqslant\ i\ \leqslant \text{length}(U^*) \wedge i \neq k\ \Rightarrow\ {U^*}_i = {U^{*\prime}}_i \text{ and} \\ restr({U^*}_k, \mathcal{U}\backslash\{y\}) = restr({U^{*\prime}}_k, \mathcal{U}\backslash\{y\}) \end{aligned} \tag{B.4b}$$

and:

$$decisions(matches_P(U^*)) \neq decisions(matches_P(U^{*\prime})) \tag{B.4c}$$

for if no such U^* and $U^{*\prime}$ existed then y could be removed from sp and sp' by step reduction.

By (4.10c) we have $sq \in next_Q(t)$ with $t^\frown sq \in prefixes(matches_Q(U^*))$. As by assumption $y \notin symb(sq)$ and because U^* and $U^{*\prime}$ are the same apart from the value of y in their respective k^{th} elements, we also have:

$$t^\frown sq \in prefixes(matches_Q(U^{*\prime}))$$

and as U^* and $U^{*\prime}$ are identical beyond their respective k^{th} elements, we must have:

$$matches_Q(U^*) = matches_Q(U^{*\prime})$$

and so:

$$decisions(matches_Q(U^*)) = decisions(matches_Q(U^{*\prime})) \tag{B.5}$$

but as $P \cong Q$, (4.14) and (B.5) then require that:

$$decisions(matches_P(U^*)) = decisions(matches_P(U^{*\prime}))$$

which contradicts (B.4c). This means that sq must use y as well. But if corresponding steps in P and Q use the same set of symbols it is impossible to construct a normal form different from P, so normal form is unique. □

B.3 Composition can preserve acyclicity

Theorem B.3. We show that it is always possible to choose two nodes of an acyclic graph so that the result of merging the nodes is acyclic.

Proof. Consider a topological ordering $m_1, \ldots, m_n$ of the nodes of the graph so that $m_j \rightarrow m_k \Rightarrow k > j$ and m_1 is a terminal node. Suppose that m_i, $2 \leqslant i \leqslant n$, is the first node in the ordering connected by an edge to m_1. There cannot be any other dependency path from m_1 to m_i otherwise there would have to be a node earlier than m_i in the ordering connected to m_1 for such a path to use as its first link. So composing m_1 and m_i yields a new but smaller (containing fewer nodes) acyclic graph. □

B.4 Composition is abstract

Theorem B.4. We show that the results of composition do not depend on representation, so:

$$P \cong P' \wedge Q \cong Q' \Rightarrow P \parallel Q \cong P' \parallel Q' \tag{B.6}$$

Proof. Suppose that the machines are defined over universes as follows:

- P and P' over $\mathfrak{U}$
- Q and Q' over $\mathfrak{U}'$
- $P \parallel Q$ and $P' \parallel Q'$ over $\mathfrak{U}''$

where:

- For homogeneous composition: $\mathfrak{U}'' = \mathfrak{U}' = \mathfrak{U}$
- For heterogeneous composition: $\mathfrak{U}'' = \mathfrak{U}' \boxplus \mathfrak{U}$

and consider a sequence $U''^* \in \mathbb{U}''^*$.

This gives rise to a unique sequence $U^* = restr(U''^*, \mathcal{U}) \in \mathbb{U}^*$ and a unique sequence $U'^* = restr(U''^*, \mathcal{U}') \in \mathbb{U}'^*$. Because:

$$\begin{aligned} &decisions(matches_{P \parallel Q}(U''^*))_i = \\ &\qquad decisions(matches_P(U^*))_i \parallel decisions(matches_Q(U'^*))_i \end{aligned}$$

and:

$$decisions(matches_{P'\|Q'}(U''^*))_i = \\ decisions(matches_{P'}(U^*))_i \parallel decisions(matches_{Q'}(U'^*))_i$$

then because $P \cong P'$ and $Q \cong Q'$ have by (4.14) that:

$$decisions(matches_{P\|Q}(U''^*))_i = decisions(matches_{P'\|Q'}(U''^*))_i$$

and so $P \parallel Q \cong P' \parallel Q'$. □

B.5 Composition is commutative and associative

Theorem B.5. We show that composition of two protocol machines is commutative and associative. This rests on the *step composition* operator (4.23) having these properties.

Proof. As all parts of the step composition operator are symmetric in $s1$ and $s2$, commutativity of the machine composition construction follows.

If we consider $((s1 \parallel s2) \parallel s3)$ it is obvious that the operator is associative for the *action*, *decision* and *offers* parts. The *perceives* part is not obvious. By (4.23b) and (4.23d) we have:

$$((s1 \parallel s2) \parallel s3)^\pi = (((s1^\pi \cup s2^\pi) \backslash (s1^\omega \cup s2^\omega)) \cup s3^\pi) \backslash ((s1^\omega \cup s2^\omega) \cup s3^\omega)$$

which reduces to:

$$((s1 \parallel s2) \parallel s3)^\pi = ((s1^\pi \cup s2^\pi) \cup s3^\pi) \backslash ((s1^\omega \cup s2^\omega) \cup s3^\omega)$$

As set union is associative, associativity of the perceives part of step composition, and hence of step composition as a whole, follows. This means that the composition construction (4.25) is associative. □

B.6 Composition is closed

Theorem B.6. We show that composition of a set of protocol machines, all having the properties given in (4.10), yields a machine that also qualifies as a protocol machine, so that the composition operator is *closed*.

Suppose that P, Q are protocol machines defined over a universe $\mathfrak{U}$, and suppose that we are constructing a completion of their composition. We use mathematical induction, assuming that all steps used so far satisfy (4.10) and consider a next step, $sp \parallel sq$. Each condition of (4.10) is considered in turn.

Consistency with the universe: (4.10a)**.**

Proof. Because sp and sq obey (4.10a) and by definition of $\Subset$:

$$\exists\, U1, U2 \in \mathbb{U} \text{ with } sp^\tau \subseteq U1,\ sq^\tau \subseteq U2$$

which requires that $symb(U1 \cup U2) \subseteq \mathcal{U}$. As by (4.21), $con(sp^\tau \cup\ sq^\tau)$ and by the definition of the total universe (4.6) we must have $U3 \in \mathbb{U}$ with $sp^\tau \cup sq^\tau \subseteq U3$. □

Correct use of symbol sets: (4.10b)**.**

Proof. This follows directly from the fact that sp and sq obey (4.10b) and from the definition of step composition, (4.23). □

Exhaustiveness of steps: (4.10c)**.**

Proof. Given a universe element U, (4.10c) guarantees that there are steps sp and sq with $sp^\tau \subseteq U$ and $sq^\tau \subseteq U$. This means that $sp \sim sq$ and gives a step $sp \parallel sq$ with $(sp \parallel sq)^\tau \subseteq U$. □

Uniqueness of steps: (4.10d)**.**

Proof. Suppose that two steps, $sp \parallel sq$ and $sp' \parallel sq'$ both match a given universe element U. As all must match U, by (4.10d) for P and Q we have $sp = sq$ and $sp' = sq'$. This means that the composed step is unique. □

Determinism: (4.10e)**.**

Proof. We suppose that $P \parallel Q$ does not obey (4.10e) but that both P and Q do obey and show a contradiction. We will rely on (4.28) and take it that P is independent machine.

We consider steps that match a set of observations $U^\epsilon \Subset restr(\mathbb{U}, \mathcal{U}\backslash(\Omega_P \cup \Omega_Q))$ that fixes the external data for $P \parallel Q$. Because we assume P is independent and obeys (4.10e), this gives a single step sp of P with $sp^\delta = allow$. The only way that $P \parallel Q$ can violate (4.10e) is if there are two steps sq and sq' with $sq^\delta = sq'^\delta = allow$ which are distinct in the sense that $sq^\omega \neq sq'^\omega$ but such that $(sp \parallel sq)^\varepsilon$ and $(sp \parallel sq')^\varepsilon$ both match U^ϵ. This requires that $con((sp \parallel sq)^\pi \cup (sp \parallel sq')^\pi)$, so:

$$con(((sp^\pi \cup sq^\pi)\backslash(sp^\omega \cup sq^\omega)) \quad \cup \quad ((sp^\pi \cup sq'^\pi)\backslash(sp^\omega \cup sq'^\omega))) \tag{B.7}$$

By assumption, Q obeys (4.10e), so we must have $\neg\, con(sq^\pi \cup sq'^\pi)$ otherwise sq and sq' could not be distinct. This requires that we have observations, v and v', with $\neg\, con(\{v\} \cup \{v'\})$ and $v \in sq^\pi$ and $v' \in sq'^\pi$. Then, if v is in the left hand component of the union in (B.7), we must have both of:

- v' is not in the right hand component of (B.7), and this requires $v' \in sp^\omega$, and
- v is in the right hand component of (B.7), and this requires $v \in sp^\pi$.

However, $v' \in sp^\omega$ and $v \in sp^\pi$ cannot both be true as by (4.10a) we must have $con(sp^\tau)$. So v cannot be in the left hand component of (B.7), and this requires that $v \in sp^\omega$. A similar argument shows that we must also have $v' \in sp^\omega$, and this is a contradiction as we must have $con(sp^\omega)$. This means that distinct steps sq and sq' with the same external image cannot exist, and this establishes (4.10e) for $P \parallel Q$. □

B.7 Decomposition is unique

Theorem B.7. We show that there is a unique decomposition of the completions of a composite machine $P \parallel Q$ into the completions of P and Q from which they derive. Uniqueness of decomposition requires that:

$$\begin{aligned} &\forall\, s \in asSet(\mathcal{B}_{P\parallel Q}) \text{ and } sp1, sp2 \in asSet(\mathcal{B}_P) \text{ and } sq1, sq2 \in asSet(\mathcal{B}_Q) \text{ with} \\ &\qquad sp1 \parallel sq1 = sp2 \parallel sq2 = s : \qquad sp1 = sp2 \ \wedge\ sq1 = sq2 \end{aligned} \tag{B.8}$$

Proof. If $sp1 \parallel sq1 = sp2 \parallel sq2$ then (4.23) gives:

$$sp1^\alpha \cup sq1^\alpha = sp2^\alpha \cup sq2^\alpha \tag{B.9a}$$

$$sp1^\omega \cup sq1^\omega = sp2^\omega \cup sq2^\omega \tag{B.9b}$$

$$(sp1^\pi \cup sq1^\pi)\backslash(sp1^\omega \cup sq1^\omega) = (sp2^\pi \cup sq2^\pi)\backslash(sp2^\omega \cup sq2^\omega) \tag{B.9c}$$

Clearly (B.9a) and (B.9b) give:

$$con(sp1^{\alpha} \cup sp2^{\alpha}) \tag{B.10a}$$

$$sp1^{\omega} = sp2^{\omega} \tag{B.10b}$$

Now we assume that we have observations:

$$v1 \in sp1^{\pi} \text{ and } v2 \in sp2^{\pi} \text{ with } \neg\, con(\{v1\} \cup \{v2\}) \tag{B.11}$$

and proceed to show a contradiction. The assumption (B.11) requires that both of the following are true:

- Either $v1$ is in the left hand side of (B.9c) or $v1 \in sp1^{\omega} \cup sq1^{\omega}$
- Either $v2$ is in the right hand side of (B.9c) or $v2 \in sp2^{\omega} \cup sq2^{\omega}$

However, we cannot have $v1$ in the left hand side of (B.9c) and $v2$ in the right hand side of (B.9c). So without loss of generality we can assume that:

$$v1 \text{ in the left hand side of (B.9c)} \tag{B.12a}$$

$$v2 \in sp2^{\omega} \cup sq2^{\omega} \tag{B.12b}$$

By (B.9b) and (B.12b) we have :

$$v2 \in sp1^{\omega} \cup sq1^{\omega} \tag{B.13}$$

But (B.12a) means $v1 \in (sp1 \parallel sq1)^{\pi}$ and (B.13) means $v2 \in (sp1 \parallel sq1)^{\omega}$ and this contradicts (4.10a) for $P \parallel Q$. So assumption (B.11) must be false and we have:

$$con(sp1^{\pi} \cup sp2^{\pi}) \tag{B.14}$$

Combining (B.10a), (B.10b) and (B.14) gives $con(sp1^{\tau} \cup sp2^{\tau})$ and (4.10d) then gives $sp1 = sp2$. A similar argument gives $sq1 = sq2$ and this establishes (B.8). □

B.8 A combined universe is well-formed

Theorem B.8. We show that, under suitable conditions, if $\mathfrak{U}$ and $\mathfrak{U}'$ obey (4.7) then so does $\mathfrak{U} \boxplus \mathfrak{U}'$.

Lemma. First, we show that a acyclic graph always contains a node that is only "points to" leaf nodes. In other words there is always a node n such that:

$$\forall\, n' \text{ with } n \to n' : \nexists\, n'' \text{ with } n' \to n'' \tag{B.15}$$

Proof. We suppose that this is not the case. We can then choose any node and follow the link that navigates to a non-leaf node on which it is dependent. We can repeat this without end, and this leads to an infinite path that never reaches a leaf. In a finite graph this must entail a cycle, contradicting the assumption that the graph is acyclic □

To prove the main result we consider the relation $\mathcal{F}_{\mathfrak{U}}$ between symbols in a universe $\mathfrak{U}$ based on *fixes*:

$$\langle d, y\rangle \in \mathcal{F} \quad \Leftrightarrow \quad d \in \mathcal{D} \quad \wedge \quad y \in \mathit{symb}\Big(\bigcup_{v \in \mathit{restr}(\mathcal{E},\{d\})} \mathit{fixes}(v)\Big) \tag{B.16}$$

which relates a derived symbol d to any symbol used by any fix for a value of d. We refer to this as the *derived symbol relation* for the universe $\mathfrak{U}$.

We then require for a combined universe $\mathfrak{U} \boxplus \mathfrak{U}'$ that:

$$\text{The combined derived symbol relation } \mathcal{F}_{\mathfrak{U}} \cup \mathcal{F}_{\mathfrak{U}'} \text{ is acyclic} \tag{B.17}$$

and we show that the combined universe then obeys (4.7).

The valid universe contained by the total universe: (4.7a)**.**

Proof. This follows directly form the definition of the combined universe (4.31a) and the fact that both component universes obey (4.7a). □

Exhaustiveness of fixes: (4.7b)**.**

Proof. We imagine a directed graph formed from the combined derived symbol relation $\mathcal{F}_{\mathfrak{U}} \cup \mathcal{F}_{\mathfrak{U}'}$ whose nodes are symbols. The leaves of this graph are the symbols in:

$$\mathit{symb}\Big(\bigcup_{v \in \mathit{restr}(\mathcal{E},\{d\})} \mathit{fixes}(v) \quad \cup \bigcup_{v' \in \mathit{restr}(\mathcal{E},\{d'\})} \mathit{fixes}'(v')\Big) \setminus (\mathcal{D} \cup \mathcal{D}') \tag{B.18}$$

and we suppose that these nodes are endowed with values. By the earlier lemma, there is a non-leaf node in this structure which is only fixed by leaves. Without loss of generality suppose that the symbol of this node belongs to $\mathcal{D}$. By (4.7b) in $\mathfrak{U}$ this

node has a fixed value. We now remove the edges that join this node to the leaf nodes used to fix it. The result of this is still acyclic as removing edges cannot create a cycle, so we can repeat the process until every node has a value, establishing (4.7b) for the combined universe. □

Uniqueness of fixes: (4.7c)**.**

Proof. We suppose that both $\mathfrak{U}$ and $\mathfrak{U}'$ obey (4.7c) and consider two graphs of the combined derived symbol relation of the two. These graphs are topologically isomorphic. We suppose that the leaves of both graphs have been endowed with values in the same way, so corresponding leaves have the same values, and that the non-leaf nodes have been given values according to their fixes. We suppose that at least one non-leaf node receives different derived values in the two graphs, and consider the subgraph comprising all the nodes which are given different values in the two graphs. As this subgraph is acyclic it must have a leaf node, and without loss of generality suppose that this node is derived in $\mathfrak{U}$. This node is fixed to two different values by a given set of basis symbol values in the two graphs, which is a contradiction as it violates (4.7c) in $\mathfrak{U}$. This means that $\mathfrak{U} \boxplus \mathfrak{U}'$ obeys (4.7c). □

Correctness of derivation : (4.7d)**.**

Proof. We assume that there is a $V \in \mathbb{V}''$ with $\neg \mathit{closed}(V)$ and show a contradiction. The assumption means that must be a $v^\dagger \in \mathit{restr}(V, \mathcal{D}'')$ such that:

$$\forall\, W \in \mathit{fixes}''(v^\dagger) :\ W \not\subseteq V \tag{B.19}$$

However, as by (4.31b) we know that:

$$\mathit{restr}(V,\mathcal{U}) \in \mathbb{V} \ \wedge\ \mathit{restr}(V,\mathcal{U}') \in \mathbb{V}'$$

and this requires that V contains fixes for all derived symbols in both $\mathcal{D}$ and $\mathcal{D}'$. Moreover, because all these fixes are consistent with V they are consistent with each other. Because the construction of fixes'' by (4.31c) proceeds by recursively combining consistent fixes, it will be able to form an element of $\mathit{fixes}''(v^\dagger)$ that is consistent with V. This contradicts the assumption (B.19) that no such fix exists and completes the proof. □

Derivation uses persistent data: (4.7e)**.**

Proof. This follows directly from the compatibility condition (4.30a) on combined universes. □

Completeness of combined universe : (4.7f)**.**

Proof. We suppose that we have:

$$W1, W2 \Subset \mathbb{V}'' \tag{B.20}$$

with:

$$\mathit{closed}\,(W1) \text{ and } \mathit{closed}\,(W2) \text{ and } \mathit{con}(W1 \cup W2)$$

and show how to construct $V^\dagger \in \mathbb{V}''$ with $W1 \cup W2 \subseteq V^\dagger$.

We start by selecting any $W^\dagger \Subset \mathbb{V}''$ with:

$$\mathit{symb}(W^\dagger) \cap \mathcal{D}'' = \emptyset \quad \wedge \quad \mathit{symb}(W^\dagger \cup W1 \cup W2) \backslash \mathcal{D}'' = \mathcal{U}'' \backslash \mathcal{D}''$$

As $W^\dagger$ contains no derived symbols, and using closed' to mean closed in $\mathfrak{U}'$, this construction gives:

$$\mathit{closed}(\mathit{restr}(W^\dagger, \mathcal{U})) \quad \text{and} \quad \mathit{closed}'(\mathit{restr}(W^\dagger, \mathcal{U}'))$$

and by (4.7e) for $\mathfrak{U}$ and $\mathfrak{U}'$ this means that:

$$\mathit{restr}(W^\dagger \cup W1 \cup W2, \mathcal{U})) \Subset \mathbb{V} \quad \text{and} \quad \mathit{restr}(W^\dagger \cup W1 \cup W2, \mathcal{U}')) \Subset \mathbb{V}'$$

We now imagine a graph of the combined derived symbol relation $\mathcal{F}_\mathfrak{U} \cup \mathcal{F}_{\mathfrak{U}'}$ in which the nodes corresponding to $\mathit{symb}(W^\dagger \cup W1 \cup W2)$ have been endowed with their values. This endowment gives values to all the leaf nodes as $W^\dagger$ is constructed to include all non-derived symbols, and by (B.20) and (4.7f) for $W1$ and $W2$ all derived symbol values will have fixes valid in their respective universes. We can then remove the links that connect the derived values to the nodes that fix them and then, because by the lemma above, we must have a node that is only fixed by leaves and give this node the fix so determined. We repeat this process until nodes are given values and denote by $W^\ddagger$ the set of observations for derived symbols created by this process. This gives $V^\dagger = W^\ddagger \cup W^\dagger \cup W1 \cup W2$ with:

$$W1 \cup W2 \subseteq V^\dagger \text{ and } V^\dagger \in \mathbb{V}''$$

which means that $W1 \cup W2 \Subset \mathbb{V}''$ and establishes (4.7f) for the combined universe. □

B.9 Well-behavedness is abstract

Theorem B.9. We show that if P obeys (4.34) in any representation then it will obey it in normal form.

Proof. We suppose that the representation $\mathcal{B}_P$ of a machine P obeys (4.34) and that $P^\dagger$ is the normal form of P so $\mathcal{B}_{P^\dagger} = \mathcal{B}_P{}^{norm}$. We assume step $s^\dagger \in asSet(\mathcal{B}_{P^\dagger})$ does not obey (4.34) and show a contradiction.

Firstly, we assume $s^{\dagger\delta} = allow$ but that $s^{\dagger\tau} \notin \mathbb{V}$. We suppose that $t^\dagger \in stems(\mathcal{B}_{P^\dagger})$ with $t^\dagger{}^\frown s^\dagger = matches_{P^\dagger}(U^*)$. As $P^\dagger$ is in normal form we cannot remove any symbol in $symb(s^\dagger)$ by step reduction, and as normal form is unique we have:

$$symb(last(matches_P(U^*))) \supseteq symb(s^\dagger)$$

and this means that $last(matches_P(U^*))^\tau \notin \mathbb{V}$. Then, as P obeys (4.34):

$$last(matches_P(U^*))^\delta \neq allow \tag{B.21}$$

As we assume that $P \cong P^\dagger$ then (4.14) requires that:

$$decisions(matches_P(U^*)) = decisions(t^\dagger{}^\frown s^\dagger)$$

and so:

$$last(matches_P(U^*))^\delta = s^{\dagger\delta}$$

and this with (B.21) contradicts the assumption.

Secondly, we assume $s^{\dagger\delta} = allow$ but that $\neg\, closed(s^{\dagger\tau})$. We must then have $v, v' \in restr(\mathbb{U}\mathbb{V}, \mathcal{D})$ with $v \in s^{\dagger\tau}$ and $\neg\, con(\{v\} \cup \{v'\})$ and:

- $W \in fixes(v)$ with $con(W \cup s^{\dagger\tau})$
- $W' \in fixes(v')$ with $con(W' \cup s^{\dagger\tau})$

for if no such v, v', W, W' existed then every derived attribute in $s^\dagger$ would be determined to a single value, but this would require $closed(s^{\dagger\tau})$ and we assume that this is not the case.

We can use step expansion to add to $s^\dagger$ all symbols in $symb(W')\backslash symb(s^\dagger)$ to form a new representation $P^\ddagger$. This must generate a step $s^\ddagger$ in $P^\ddagger$ with $W' \subseteq s^{\ddagger\tau}$ and this means that $s^{\ddagger\tau} \notin \mathbb{V}$ as $s^\ddagger$ contains a fix to v' but also contains v. Suppose:

$$t^\dagger \frown s^\ddagger = matches_{P^\ddagger}(U^{*\ddagger})$$

As $s^\ddagger$ is created from $s^\dagger$ by step expansion we have:

$$t^\dagger \frown s^\dagger = matches_{P^\dagger}(U^{*\ddagger}) \text{ and } s^{\ddagger\delta} = s^{\dagger\delta} = allow \tag{B.22}$$

As we assume that $P \cong P^\dagger$ then (4.14) requires that:

$$decisions(matches_P(U^{*\ddagger})) = decisions(t^\dagger \frown s^\dagger)$$

and this and (B.22) gives:

$$last(matches_P(U^{*\ddagger}))^\delta = s^{\dagger\delta} = allow \tag{B.23}$$

However as $P^\dagger$ is in normal form we cannot remove v^{symb} by step reduction and as normal form is unique, $v^{symb} \in symb(last(matches_P(U^{*\ddagger})))$ as well, so we must have:

$$v \in last(matches_P(U^{*\ddagger}))^\tau$$

By construction of $s^\ddagger$ we know $last(U^{*\ddagger})$ contains W' which fixes v' so we know by (4.7c) that $last(U^{*\ddagger})$ cannot contain a fix for v. This means that:

$$\neg\, closed(last(matches_P(U^{*\ddagger}))^\tau) \tag{B.24}$$

The combination of (B.23) and (B.24) contradicts the assumption that P obeys (4.34). This establishes that $P^\dagger$, the normal form of P, must obey (4.34). □

B.10 Heterogeneous composition preserves well-behavedness

Theorem B.10. We show sufficiency of conditions (4.38) to ensure that heterogeneous composition of well-behaved machines yields another well-behaved machine.

Proof. We suppose that (4.38) holds, and show that $P \parallel Q$ is well-behaved. We suppose that P and Q are both well-behaved and take steps sp and sq respectively giving a composed step $sp \parallel sq$. We show that if sp and sq both obey (4.34) and the conditions (4.38) hold, then so does $sp \parallel sq$. We use:

- *closed* to denote closed in $\mathfrak{U}$
- *closed*$'$ to denote closed in $\mathfrak{U}'$
- *closed*$''$ to denote closed in $\mathfrak{U}'' = \mathfrak{U} \boxplus \mathfrak{U}'$

and assume:

$$(sp \parallel sq)^\delta = allow \tag{B.25}$$

As by (B.25) and (4.23c) we must have $sp^\delta = sq^\delta = allow$, and as P and Q are well-behaved we have:

$$closed(sp^\tau) \quad \wedge \quad sp^\tau \Subset \mathbb{V} \tag{B.26a}$$

$$closed'(sq^\tau) \quad \wedge \quad sq^\tau \Subset \mathbb{V}' \tag{B.26b}$$

The conditions (4.38) allow us to construct sets of observations that only use symbols that are shared between $\mathfrak{U}$ and $\mathfrak{U}'$ and have no derived symbols from $\mathcal{D}$, and such sets are therefore closed and valid in $\mathfrak{U}$. Thus:

$$restr(sq^\alpha \backslash sp^\alpha, \mathcal{U}) \Subset \mathbb{V} \text{ and } closed \text{ by (4.30c) and (4.38b)} \tag{B.27a}$$

$$restr((sq^\omega \cup sq^\pi) \backslash (sp^\omega \cup sp^\pi), \mathcal{U}) \Subset \mathbb{V} \text{ and } closed \text{ by (4.30c) and (4.38a)} \tag{B.27b}$$

Now we use (4.7f) on (B.26a), (B.27a) and (B.27b) to give:

$$restr(sp^\tau \cup (sq^\alpha \backslash sp^\alpha) \cup ((sq^\omega \cup sq^\pi) \backslash (sp^\omega \cup sp^\pi)), \mathcal{U}) \Subset \mathbb{V} \text{ and } closed \tag{B.28}$$

which gives:

$$restr(sp^\tau \cup sq^\tau, \mathcal{U}) \Subset \mathbb{V} \text{ and } closed \tag{B.29}$$

Similarly we have:

$$restr(sp^\alpha \backslash sq^\alpha, \mathcal{U}') \Subset \mathbb{V}' \text{ and } closed' \text{ by (4.30c) and (4.38b)} \tag{B.30a}$$

$$restr((sp^\omega \cup sp^\pi) \backslash (sq^\omega \cup sq^\pi), \mathcal{U}') \Subset \mathbb{V}' \text{ and } closed' \text{ by (4.30c) and (4.38a)} \tag{B.30b}$$

and use (4.7f) on (B.26b), (B.30a) and (B.30b) to give:

$$restr(sq^\tau \cup (sp^\alpha \backslash sq^\alpha) \cup ((sp^\omega \cup sp^\pi) \backslash (sq^\omega \cup sq^\pi)), \mathcal{U}') \Subset \mathbb{V}' \text{ and } closed' \tag{B.31}$$

which gives:

$$\mathit{restr}(\mathit{sp}^\tau \cup \mathit{sq}^\tau, \mathcal{U}') \Subset \mathbb{V}' \text{ and } \mathit{closed}' \tag{B.32}$$

Finally we can use (4.31b) on (B.29) and (B.32) to give:

$$\mathit{sp}^\tau \cup \mathit{sq}^\tau \Subset \mathbb{V}'' \text{ and } \mathit{closed}''$$

which establishes (4.34) for $P \parallel Q$. □

B.11 Derived-state is abstract

Theorem B.11. Given that the universe over which P is defined is in normal form, we show that if P obeys (4.40) in any representations then it will obey it in normal form.

Proof. We assume that P does not obey (4.40) in normal form but that some non-normal representation does obey (4.40) and show a contradiction. There must be a step $s \in {\mathcal{B}_P}^{\mathit{norm}}$ with $s^\delta = \mathit{allow}$ that does not obey (4.40). By (4.34) we have $\mathit{closed}(s^\tau)$ and so there is a $W \in \mathit{fixes}(\mathit{restr}(s^\tau, \mathit{state}_P))$ with $W \subseteq s^\tau$. As s does not obey (4.40) there must then be a $y \in \mathit{symb}(W)$ with $y \notin \Omega_P$. As by assumption the universe is in normal form we cannot use fix reduction on W to remove y. Moreover fix expansion on s cannot remove y and so all other representations of P must also not obey (4.40). This contradicts the assumption that some non-normal representation does obey (4.40). This establishes the result. □

B.12 Ignore is closed

Theorem B.12. We now show that if P and Q both meet conditions (4.42) for an action A then so does $P \parallel Q$.

First we note that if P and Q both have state attributes, state_P and state_Q, then the derived pair $(\mathit{state}_P, \mathit{state}_Q)$ serves as a state attribute $\mathit{state}_{P \parallel Q}$. This is because, as argued in Section 4.3.3, any continuation of $tp \parallel tq$ can be decomposed uniquely into continuations of tp and tq. As the set of continuations of tp and tq is determined by $\mathit{state}_P(tp)$ and $\mathit{state}_Q(tq)$ respectively, the set of possible continuations of $tp \parallel tq$ is determined by $(\mathit{state}_P, \mathit{state}_Q)$.

Now suppose we have action A ignored by both P and Q and:

- stems $tp \in \mathit{stems}(\mathcal{B}_P)$ and $tq \in \mathit{stems}(\mathcal{B}_Q)$ with $tp \sim tq$
- steps $sp \in \mathit{asSet}(\mathcal{B}_P)$ and $sq \in \mathit{asSet}(\mathcal{B}_Q)$ with $sp \sim sq$
- $\mathit{con}(sp^\alpha \cup sq^\alpha \cup A)$

and address homogeneous and heterogeneous composition in turn.

Homogeneous composition.

Proof. We assume that both P and Q are defined over the same universe $\mathfrak{U}$, and that $A \in \mathit{ignores}_P \cap \mathit{ignores}_Q$. We show that $A \in \mathit{ignores}_{P \parallel Q}$.

Suppose that the left hand side of (4.42) holds for $(sp \parallel sq)$. We observe that:

- If either $sp^\tau \notin \mathbb{V}$ or $sq^\tau \notin \mathbb{V}$ then $sp^\tau \cup sq^\tau \notin \mathbb{V}$.
- If either $\mathit{restr}(sp^\omega, \Omega_P \backslash \mathcal{D}) \neq \mathit{restr}(tp^\omega, \Omega_P \backslash \mathcal{D})$ or $\mathit{restr}(sq^\omega, \Omega_P \backslash \mathcal{D}) \neq \mathit{restr}(tq^\omega, \Omega_P \backslash \mathcal{D})$ then $\mathit{restr}(sp^\omega \cup sq^\omega, (\Omega_P \cup \Omega_Q) \backslash \mathcal{D}) \neq \mathit{restr}(tp^\omega \cup tq^\omega, (\Omega_P \cup \Omega_Q) \backslash \mathcal{D})$.

This requires that both sp and sq observe the left hand side of (4.42), so $sp^\delta = sq^\delta = \mathit{allow}$ and this establishes (4.42) for $P \parallel Q$. □

Heterogenous composition.

Proof. For heterogeneous composition, we assume that P is defined over universe $\mathfrak{U}$ and Q over $\mathfrak{U}'$ with $\mathfrak{U}'' = \mathfrak{U} \boxplus \mathfrak{U}'$. We suppose that $\mathit{restr}(A, \mathcal{U}) \in \mathit{ignores}_P$ and $\mathit{restr}(A, \mathcal{U}') \in \mathit{ignores}_Q$. We show that $A \in \mathit{ignores}_{P \parallel Q}$.

We observe that if $y \in (\Omega_P \cup \Omega_Q) \backslash \mathcal{D}''$ then $y \in \Omega_P \wedge y \notin \mathcal{D}$ and/or $y \in \Omega_Q \wedge y \notin \mathcal{D}'$. With this observation the argument given above for homogeneous composition carries across. □

B.13 Composition of contracts preserves satisfaction

Theorem B.13. We show that if two protocol contracts $\mathbb{C}1 = \langle C1,\ F1\rangle$ and $\mathbb{C}2 = \langle C2,\ F2\rangle$ are both satisfied by a machine P then P also satisfies their composition $\mathbb{C}1 \oplus \mathbb{C}2$.

Proof. Firstly we show that P satisfies the behaviour part of the composed contract:

$$P \vdash \mathbb{C}1 \text{ and } P \vdash \mathbb{C}2 \Rightarrow P = C1 \parallel P \text{ and } P = C2 \parallel P$$
$$\text{so } P = (C1 \parallel C2) \parallel P$$

Secondly we show that P satisfies the fully constrained actions part of the combined contract. For this, suppose $A \in F1$. Then $\exists\ X$ with $P = C1 \parallel X$ and X never refuses A.

$$P \vdash \mathbb{C}2 \Rightarrow P = C2 \parallel P$$
$$\text{so } P = (C2 \parallel C1) \parallel X$$

This completes the proof. □

B.14 Decomposition of contracts preserves satisfaction

Theorem B.14. We show that if a machine P satisfies a contract $\mathbb{C}1 \oplus \mathbb{C}2$ and (8.3) holds then P also satisfies the decomposed parts $\mathbb{C}1 = \langle C1,\ F1\rangle$ and $\mathbb{C}2 = \langle C2,\ F2\rangle$.

Proof. Suppose that stricture (8.3) is met and that $P \vdash \mathbb{C}$. We have:

$$P = C \parallel P = C1 \parallel C2 \parallel P \tag{B.33}$$

And by (B.33) and (4.29):

$$\mathit{prefixes}(\mathcal{B}_P) \ \supseteq\ \mathit{prefixes}(\mathcal{B}_{(P\parallel C1)}\restriction_P) \ \supseteq \mathit{prefixes}(\mathcal{B}_{(P\parallel C1\parallel C2)}\restriction_P) = \mathit{prefixes}(\mathcal{B}_P)$$

so:

$$\mathcal{B}_P = \mathcal{B}_{(C1\parallel P)}\restriction_P \tag{B.34}$$

Moreover by (4.20): $\Omega_P \subseteq \Omega_{(P\parallel C1)} \subseteq \Omega_{(P\parallel C1\parallel C2)} = \Omega_P$ so:

$$\Omega_P = \Omega_{(P\parallel C1)} \tag{B.35}$$

Together (B.34) and (B.35) require that:

$$P = C1 \parallel P \tag{B.36}$$

Suppose $A \in F \cap alphabet_{C1}$. As $P \vdash C$, $\exists\ X$ with $P = C \parallel X$ and X never refuses A. So $P = C1 \parallel (C2 \parallel X)$.

As by stricture (8.3) $A \notin alphabet_{C2}$ so $X' = (C2 \parallel X)$ can never refuse A and we have:

$$P = C1 \parallel X' \text{ and } X' \text{ can never refuse } A \tag{B.37}$$

(B.36) and (B.37) $\Rightarrow P \vdash \mathbb{C}1$. Similarly, $P \vdash \mathbb{C}2$. This completes the proof. □

Appendix C

Semantic Function

We develop a semantic function for PM, by providing mapping from concrete syntax to the semantic domain of completions developed in Part II. Because we have not defined a concrete syntax for PM in detail, the semantic function given here is illustrative.

C.1 Decision Function

For development of a semantic function we will work with an alternative definition of the behaviour of a machine, based on the definition of machine equivalence given as (4.14). This alternative definition of a machine using a "decision function" Y so that a machine is defined as:

$$\langle\, \mathfrak{U}, \Omega_P, \mathrm{Y}_P \,\rangle \tag{C.1}$$

where, as previously, Ω_P is the set of offered symbols, but in place of $\mathcal{B}_P$ we use a *decision function* Y_P . This function has following signature:

$$\mathrm{Y}_P :: \mathbb{U}^* \to \{allow, refuse, crash\}^* \tag{C.2}$$

and gives, for any finite sequence of universe elements, the corresponding sequence of decisions that P would make. We can readily convert between the two forms of definition of behaviour, expressed as completions B_P or expressed as a decision function Y_P, as follows:

$$\forall\, U^* \in \mathbb{U}^* : \mathrm{Y}_P(U^*) = decisions(matches_P(U^*)) \tag{C.3a}$$

to go from $\mathcal{B}_P$ to Υ_P and:

$$prefixes(\mathcal{B}_P) = \{zip_{\sqcup}(\Omega_P, U^*, \Upsilon_P(U^*)) \mid U^* \in \mathbb{U}^* \} \tag{C.3b}$$

to go from Υ_P to $\mathcal{B}_P$.

C.2 Execution machine

In order to create a Semantic Function we assume that we have a concrete machine M that has a set Ξ of possible execution states. Moreover we assume that M supports a name-space (a set of names) Λ which is partitioned into two disjoint parts Λ^{store} and Λ^{buffer}, the first used being used to name data elements of persistent data storage and the second used to name data elements in a message buffer.

We assume a semantic mapping $symbols :: power(\Lambda) \rightarrow power(\mathcal{Y})$ from the name-space to the symbols used in the formalisation so that:

$$\begin{aligned} &\mathcal{U} \subseteq symbols[\![\Lambda]\!] \\ &\mathcal{A} \subseteq symbols[\![\Lambda^{buffer}]\!] \end{aligned} \tag{C.4}$$

C.3 Computational Functions

There is no unique way for realizing PM as a computational system and in order to make a semantic mapping we have to assume some choices. A key choice is the way that derivation is handled, where there are two alternatives:

- To embed derivation functionality in the data management system, so that it is separate and independent from the components that implement protocol machine behaviour.
- To embed derivation logic in the components that implement protocol machine behaviour, so that the data management system is "dumb".

The first of these requires that the data management system has a capability to host and execute derivation rules. This is possible in some environments, such as databases that support "views" (derived tables that a created by executing SQL on the fly). However the second approach is more conventional (and is the one adopted in the ModelScope tool described in Section 7.2) so that is the approach we take here.

We assume that the behaviour of M is defined by three deterministic computational functions:

$$\begin{aligned} &\mathit{initialise} :: \Xi \to \mathit{power}(\Xi) \\ &\mathit{update} :: \Xi \to \mathit{power}(\Xi) \\ &\mathit{derive} :: \Xi \to \mathit{power}(\Xi) \end{aligned} \tag{C.5}$$

which together determine how the machine behaves. The first of these initialises storage at the start of execution. The second defines how storage is updated when an action is processed, and in graphical PM representations such as Figures 1.2 on page 20 and 7.1 on page 126 corresponds to the bubbles attached to the arrows and the implicit state updates depicted by the transition arrows. The third defines how derived attributes and fields are calculated, and corresponds to the derivation definitions shown in boxes in Figures 1.2 and 7.1. Each step in the life of the machine consists of executing *update* followed by *derive*, apart from the first which is *initialise* followed by *derive*. The machine only retains the new state achieved by a step if it does not crash and the results of *initialise* (for the first step) or *update* (for subsequent steps) and *derive* agree, so every variable that is given a value by both parts of a step is given the same value; otherwise the machine executes a "rollback" to its state before the step.

The use of the powersets in (C.5) reflects the fact that these computational functions do not fully determine the resultant state of M, so for a given "start state" there is, in general, a set of possible "end states" that are compatible with the computation of the functions. This is not to say that the functions are non-deterministic, only that the data that each can update (its range) is not the full state of M.

We further assume that each of these functions has a *domain*, being the data it uses to calculate its output; and a *range*, being the data that it update as a result of execution. These are given as subsets of Λ by *domain*(*initialise*), *domain*(*update*), *domain*(*derive*), *range*(*initialise*), *range*(*update*) and *range*(*derive*).

We now proceed to construct $[\![M]\!]$. We start with the universe for which we can define:

$$\begin{aligned}\mathcal{U} \;&= \mathit{symbols}[\![\mathit{domain}(\mathit{update})]\!] \;\cup\; \mathit{symbols}[\![\mathit{range}(\mathit{update})]\!] \;\cup \\ &\quad\; \mathit{symbols}[\![\mathit{domain}(\mathit{derive})]\!] \;\cup\; \mathit{symbols}[\![\mathit{range}(\mathit{derive})]\!]\end{aligned} \tag{C.6a}$$

$$\mathcal{A} \;=\; \mathcal{U} \;\cap\; \mathit{symbols}[\![\Lambda^{\mathit{buffer}}]\!] \tag{C.6b}$$

$$\mathcal{D} \;=\; \mathit{symbols}[\![\mathit{range}(\mathit{derive})]\!] \tag{C.6c}$$

$$\Omega \;=\; (\mathit{symbols}[\![\mathit{range}(\mathit{update})]\!] \;\cup \mathcal{D}) \setminus \mathcal{A} \tag{C.6d}$$

For (C.6d) it is assumed in line with (4.38a) that $\mathcal{D} \backslash \mathcal{A} \subseteq \Omega$. Mappings for *fixes* and $\mathbb{V}$ are given in the next section.

Furthermore, we assume that the results of the computational functions *update* and *derive* are only defined for some computational states of M, so that there are sets $\mathit{precon}(\mathit{update}) \subseteq \Xi$ and $\mathit{precon}(\mathit{derive}) \subseteq \Xi$ outside of which their results are undefined. The *initialise* function is defined for all states so has $\mathit{precon}(\mathit{initialise}) = \Xi$.

C.4 Behavioural Mapping

For behavioural mapping we assume a semantic mapping:

$$\mathit{map} :: \Xi \to \mathit{power}(\mathcal{E})$$

$$\mathit{map} :: \mathit{power}(\Xi) \to \mathit{power}(\mathit{power}(\mathcal{E}))$$

which in the first form maps a given stable computational state of M to the corresponding set of observations of values for all the symbols corresponding to Λ; and in the second maps a set of stable computational states of M to the corresponding set of sets of observations, so if $\Xi' \in \mathit{power}(\Xi)$ then:

$$\mathit{map}[\![\Xi']\!] \;=\; \{\; \mathit{map}[\![\chi]\!] \mid \chi \in \Xi' \;\}$$

This gives mappings:

$$\mathbb{U} = \mathit{restr}(\mathit{map}[\![\Xi]\!], \mathcal{U})$$

$$\mathit{fixes}(v) \supseteq \{\; \mathit{restr}(\mathit{map}[\![\chi]\!], \mathcal{U} \backslash \mathcal{D}) \mid \chi \in \mathit{precon}(\mathit{derive}) \;\wedge\; v \in \mathit{map}[\![\mathit{derive}(\chi)]\!] \;\}$$

$$\begin{aligned}\mathbb{V} = \{\; V \mid \exists\, \chi \in (\mathit{precon}(\mathit{update}) \cap \mathit{precon}(\mathit{derive})) \text{ with} \\ V = \mathit{restr}(\mathit{map}[\![\chi]\!], \mathcal{U}) \;\wedge\; \mathit{closed}(V) \;\}\end{aligned}$$

where the *fixes* are not completely determined as we don't care what values are fixed when $\chi \notin precon(derive)$.

We proceed to construct a decision function Υ_M for $[\![M]\!]$, the mapping of M to the semantic domain of PM. As a preliminary we define the function:

$$decide_M :: \mathbb{U} \times \mathbb{U} \rightarrow \{allow, refuse, crash\}$$

as follows:

$$\begin{array}{l}
decide_M(U1, U2) = \\
\quad allow \;\; \text{if } \big(\forall\, \chi 1, \chi 2 \in \Xi \text{ with} \\
\qquad map[\![\chi 1]\!] \supseteq restr(U1, \mathcal{U} \backslash \mathcal{A}) \cup restr(U2, \mathcal{A} \backslash \mathcal{D}) \text{ and} \\
\qquad map[\![\chi 2]\!] \supseteq restr(U2, \mathcal{U} \backslash (\mathcal{A} \cup \mathcal{D})) : \\
\qquad \chi 1 \in precon(update) \wedge \Cap map[\![update(\chi 1)]\!] \subseteq restr(U2, \Omega \cup (\mathcal{A} \cap \mathcal{D})) \wedge \\
\qquad \chi 2 \in precon(derive) \wedge \Cap map[\![derive(\chi 2)]\!] \subseteq restr(U2, \mathcal{D}) \wedge \\
\qquad symb(\Cap map[\![update(\chi 1)]\!] \cup \Cap map[\![derive(\chi 2)]\!]) \supseteq \Omega \big) \\
\quad crash \;\; \text{if } \big(\forall\, \chi 1, \chi 2 \in \Xi \text{ with} \\
\qquad map[\![\chi 1]\!] \supseteq (restr(U1, \mathcal{U} \backslash \mathcal{A}) \cup restr(U2, \mathcal{A} \backslash \mathcal{D})) \text{ and} \\
\qquad map[\![\chi 2]\!] \supseteq restr(U2, \mathcal{U} \backslash (\mathcal{A} \cup \mathcal{D})) : \\
\qquad \chi 1 \notin precon(update) \vee \\
\qquad (\Cap map[\![update(\chi 1)]\!] \subseteq restr(U2, \Omega \cup (\mathcal{A} \cap \mathcal{D})) \wedge \chi 2 \notin precon(derive)) \big) \\
\quad refuse \;\; \text{otherwise}
\end{array} \tag{C.7}$$

In (C.7) $U1$ and $U2$ represent the starting and ending configurations of a step in $[\![M]\!]$. The execution state $\chi 1$ is used to model the update part of a step using *update*, and the execution state $\chi 2$ is used to model the derivation part of a step using *derive*. For a step to be allowed the following must pertain:

- The domain of *update* is the non-action part of the starting image including derived attributes ($restr(U1, \mathcal{U} \backslash \mathcal{A})$), combined with the non-derived fields of the new action ($restr(U2, \mathcal{A} \backslash \mathcal{D})$). These together must satisfy the pre-condition of *update* and the result of *update* must match the offered attributes and output action fields at the end of the step ($restr(U2, \Omega \cup (\mathcal{A} \cap \mathcal{D}))$).

- The domain of *derive* is the set of non-derived and non-action attributes at the end of the step ($restr(U2, \mathcal{U} \backslash (\mathcal{A} \cup \mathcal{D}))$). These must satisfy the pre-condition of

derive and the result of *derive* must match the derived attributes and fields at the end of the step ($restr(U2, \mathcal{D})$).

- The combination of *update* and *derive* must completely populate Ω.

Similarly for initiation of $[\![M]\!]$, we define a function $init_M :: \mathbb{U} \rightarrow \{allow, crash, refuse\}$ as:

$$
\begin{array}{l}
init_M(U) = \\
\quad \begin{array}{ll}
allow & \text{if } \big(\forall\, \chi_1, \chi_2 \in \Xi \text{ with } map[\![\chi_2]\!] \supseteq restr(U, \mathcal{U} \backslash (\mathcal{A} \cup \mathcal{D})) \ : \\
& \quad \Cap\ map[\![initialise(\chi_1)]\!] \subseteq restr(U, \Omega \backslash \mathcal{D}) \ \wedge \\
& \quad \chi_2 \in precon(derive) \ \wedge \ \Cap map[\![derive(\chi_2)]\!] \subseteq restr(U, \mathcal{D}) \ \wedge \\
& \quad symb(\Cap map[\![initialise(\chi_1)]\!] \ \cup \ \Cap map[\![derive(\chi_2)]\!]) \supseteq \Omega \ \big) \\
crash & \text{if } \big(\forall\, \chi_1, \chi_2 \in \Xi \text{ with } map[\![\chi_2]\!] \supseteq restr(U, \mathcal{U} \backslash (\mathcal{A} \cup \mathcal{D})) \ : \\
& \quad \Cap\ map[\![initialise(\chi_1)]\!] \subseteq restr(U, \Omega \backslash \mathcal{D}) \ \wedge \\
& \quad \chi_2 \notin precon(derive) \ \big) \\
refuse & \text{otherwise}
\end{array}
\end{array}
\tag{C.8}
$$

Note the following in (C.8):

- The range of *initialise* is confined to the non-derived offered data of the initialised state. Action data is not included in order to conform to (4.12).
- There is no pre-condition restriction on χ_1 as it must be possible to initialise M from any state.

We use $init_M$ and $decide_M$ to define the decision function Υ_M for $[\![M]\!]$ as follows:

$$
\begin{array}{l}
\Upsilon_M(U^*) = \\
\quad \begin{array}{ll}
<> & \text{if } (length(U^*) = 0) \\
< init_M(U^*_1) > & \text{if } (length(U^*) = 1) \\
\Upsilon_M(trunc(U^*)) \frown decide_M(last(trunc(U^*)), last(U^*)) & \text{if } (length(U^*) > 1 \ \wedge \\
& \qquad last\,(\Upsilon_M(trunc(U^*))) = allow) \\
\Upsilon_M(trunc(U^*)) & \text{otherwise}
\end{array}
\end{array}
$$

which then gives definition of the completions using (C.3b).

Index

Bibliography

[1] M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1 – 70, 1999.

[2] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented Programming. In *Proc. of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1015–1022, 2009.

[3] S. Apel, T. Leich, and G. Saake. Mixin-Based Aspect Inheritance. Technical report, No. 10/2005, University of Magdeburg, Germany, 2005.

[4] J. Baeten, T. Basten, and M. Reniers. *Process Algebra: Equational Theories of Communicating Processes*. Cambridge University Press, New York, NY, USA, 2009.

[5] J. Barnes. A Mathematical Theory of Synchronous Communication. Technical report, PRG-112, University of Oxford, 1993.

[6] M. Bartoletti, E. Tuosto, and R. Zunino. On the Realizability of Contracts in Dishonest Systems. In *Coordination Models and Languages. Proc. of the 14th International Conference, COORDINATION 2012*, pages 245–260, 2012.

[7] M. Bartoletti and R. Zunino. A Calculus of Contracting Processes. In *Proc. of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, Edinburgh, United Kingdom*, pages 332–341, 2010.

[8] S. Basu, T. Bultan, and M. Ouederni. Deciding Choreography Realizability. In *Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 191–202, 2012.

[9] J. Bengtson, M. Johansson, J. Parrow, and B. Victor. Psi-calculi: Mobile Processes, Nominal Data, and Logic. In *Proc. of the 24th Annual IEEE Symposium on Logic In Computer Science, 2009. LICS '09.*, pages 39–48, 2009.

[10] J. Bergstra and J. Klop. Process Algebra for Communication and Mutual Exclusion. Technical report, CS-R8409, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1984.

[11] J. Bergstra, J. Klop, and J. Tucker. Process Algebra with Asynchronous Communication Mechanisms. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 76–95. Springer-Verlag, London, UK, 1985.

[12] G. Berry. The foundations of Esterel. In *Proof, Language, and Interaction: Essays in honour of Robin Milner*, pages 425–454. 2000.

[13] A. Blass and Y. Gurevich. Abstract State Machines Capture Parallel Algorithms: correction and extension. *ACM Trans. Comput. Logic*, 9(3):1–32, 2008.

[14] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *CONCUR 2010 - Concurrency Theory: Proc. of the 21th International Conference, CONCUR 2010, Paris, France*, pages 162–176, 2010.

[15] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide (2nd Edition)*. Addison-Wesley Professional, 2005.

[16] G. Bracha and W. Cook. Mixin-based Inheritance. In *Proc. of ASM conference on Object-Oriented Programming, Systems, Languages, Applications*, pages 179–183, 1990.

[17] S. Brookes. Deconstructing CCS and CSP: Asynchronous communication, fairness and full abstraction. In *MFPS*, volume 16, page 467, 2000.

[18] M. Buscemi and U. Montanari. CC-Pi: A constraint-based language for specifying service level agreements. In *Proc. of Programming Languages and Systems: 16th European Symposium on Programming, ESOP 2007. Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS*, pages 18–32, 2007.

[19] H. Cannon. Flavors: A non-hierarchical approach to object-oriented programming. In *The Computer History Museum: Joseph Wolf collection of Symbolics documentation*. Symbolics, Inc., 1982.

[20] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[21] J. Ebert and G. Engels. Observable or Invocable Behaviour - You have to choose. *Technical report 94-38. Department of Computer Science, Leiden University*, 1994.

[22] C. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *Proc. of the 11th Australian Computer Science Conference*, pages 55–66, University of Queensland, Australia, 1988.

[23] C. Fidge. A Comparative Introduction to CSP, CCS and LOTOS. Technical report, The University of Queensland, Queensland 4072, Australia, 1994.

[24] R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *OOPSLA 2000: Proc. of the Workshop on Advanced Separation of Concerns*. Department of Computer Science, University of Twente, The Netherlands, 2000.

[25] L. Fossati, R. Hu, and N. Yoshida. Multiparty Session Nets. In *Trustworthy Global Computing*, volume 8902 of *Lecture Notes in Computer Science*, pages 112–127. Springer Berlin Heidelberg, 2014.

[26] X. Fu, T. Bultan, and J. Su. Conversation Protocols: A formalism for specification and verification of reactive electronic services. *Theoretical Computer Science*, 328(1-2):19–37, 2004.

[27] X. Fu, T. Bultan, and J. Su. Realizability of Conversation Protocols with Message Contents. In *ICWS '04: Proc. of the 2004 IEEE International Conference on Web Services*, pages 96–103, 2004.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[29] D. Gelernter and N. Carriero. Coordination Languages and Their Significance. *Commun. ACM*, 35(2):97–107, 1992.

[30] W. Grieskamp, F. Kicillof, and N. Tillmann. Action Machines: A framework for encoding and composing partial behaviours. *International Journal of Software Engineering and Knowledge Engineering*, 16(5):705–726, 2006.

[31] J. Groote and A. Ponse. Process Algebra with Guards. In *CONCUR '91*, volume 527 of *Lecture Notes in Computer Science*, pages 235–249. Springer Berlin Heidelberg, 1991.

[32] Y. Gurevich. Sequential Abstract-State Machines Capture Sequential Algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, 2000.

[33] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Dordrecht; Boston: Kluwer Academic Publishers, 1993.

[34] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[35] J. He and C. Hoare. Unifying Theories of Programming. In *Relational Methods in Logic, Algebra and Computer Science: Proc. of the 4th International Seminar RelMiCS, Warsaw, Poland*, 1998.

[36] M. Hennessy. A Term Model for Synchronous Processes. *Information and Control*, 51(1):58–75, 1981.

[37] C. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969.

[38] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[39] C. Hoare. Why ever CSP? *Electronic Notes in Theoretical Computer Science*, 162:209–215, 2006.

[40] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. *SIGPLAN Not.*, 43(1):273–284, 2008.

[41] M. Jackson. *System Development*. Prentice Hall, 1983.

[42] S. Jagannathan and J. Vitek. Optimistic Concurrency Semantics for Transactions in Coordination Languages. In *Coordination Models and Languages: Proc. of the 6th International Conference, COORDINATION 2004, Pisa, Italy*, pages 183–198, Berlin, Heidelberg, 2004.

[43] R. Kazhamiakin and M. Pistore. Analysis of Realizability Conditions for Web Service Choreographies. In *FORTE 2006: Proc. of the 26th IFIP WG 6.1 International*

Conference, Paris, volume 4229 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2006.

[44] A. Khoumsi and G. von Bochmann. Protocol Synthesis using Basic Lotos and Global Variables. In *ICNP '95: Proc. of the 1995 International Conference on Network Protocols*, pages 126–133. IEEE Computer Society, 1995.

[45] D. König, N. Lohmann, S. Moser, C. Stahl, and K. Wolf. Extending the Compatibility Notion for Abstract WS-BPEL Processes. In *Proc. of the 17th International Conference on World Wide Web*, WWW '08, pages 785–794, 2008.

[46] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[47] L. Lamport. "Sometime" is sometimes "Not Never": On the temporal logic of programs. In *Proc. of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '80, pages 174–185, 1980.

[48] I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the Gap between Interaction- and Process-Oriented Choreographies. In *SEFM'08: Proc. of the 6th IEEE International Conferences on Software Engineering and Formal Methods*, pages 323–332, 2008.

[49] J. Lange and A. Scalas. Choreography Synthesis as Contract Agreement. In *Proc. of the 6th Interaction and Concurrency Experience, Florence, Italy*, volume 131 of *Electronic Proceedings in Theoretical Computer Science*, pages 52–67. Open Publishing Association, 2013.

[50] J. Lange, E. Tuosto, and N. Yoshida. From Communicating Machines to Graphical Choreographies. In *Proc. of POPL 2015*, pages 221–232. ACM, 2015.

[51] F. Lin, P. Chu, and M. Liu. Protocol Verification using Reachability Analysis: The state space explosion problem and relief strategies. *SIGCOMM Comput. Commun. Rev.*, 17(5):126–135, August 1987.

[52] B. Liskov. Keynote address - Data Abstraction and Hierarchy. In *Addendum to the proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 17–34, 1987.

[53] A. McNeile. A Framework for the Semantics of Behavioral Contracts. In *Proc. of the 2nd International Workshop on Behaviour Modelling: Foundation and Applications*, BM-FA '10, pages 3:1–3:5, 2010.

[54] A. McNeile. Using Motivation and Choreography to Model Distributed Workflow. In *Proc. of the 5th ACM SIGCHI Annual International Workshop on Behaviour Modelling - Foundations and Applications*, BMFA '13, pages 1:1–1:11, 2013.

[55] A. McNeile and E. Roubtsova. Aspect-Oriented Development Using Protocol Modeling. In *Transactions on Aspect-Oriented Software Development VII*, volume 6210 of *Lecture Notes in Computer Science*, pages 115–150. Springer Berlin / Heidelberg, 2010.

[56] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, March 2000.

[57] R. Miarka, E. Boiten, and J. Derrick. Guards, Preconditions, and Refinement in Z. In *Proc. of the First International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '00, pages 286–303, 2000.

[58] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[59] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25(3):267 – 310, 1983.

[60] R. Milner. The Polyadic pi-Calculus: a Tutorial. In *Logic and Algebra of Specification*. Springer-Verlag New York, Inc., 1991.

[61] R. Milner. Turing, Computing and Communication. In *Interactive Computation*, pages 1–8. Springer Berlin Heidelberg, 2006.

[62] D. Milojicic. Interview with Leslie Lamport. *IEEE Distributed Systems Online*, 3(8), 2002.

[63] S. Nain and M. Vardi. Trace Semantics is Fully Abstract. In *Proc. of the 24th Annual IEEE Symposium on Logic In Computer Science*, LICS '09, pages 59–68, 2009.

[64] Object Management Group. Unified Modeling Language, Superstructure: Version 2.4.1. *OMG Document Number: formal/2011-08-06*, 2011.

[65] Object Management Group. Business Process Model and Notation (BPMN): Version 2.0.2. *OMG Document Number: formal/2013-12-09*, 2013.

[66] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In *Software Architectures and Component Technology*, volume 648 of *The Springer International Series in Engineering and Computer Science*, pages 293–323. 2002.

[67] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.

[68] S. Peyton Jones. Tackling the Awkward Squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *NATO ASI Series: Proc. of Engineering Theories of Software Construction, Marktoberdorf Summer School, 2000*, pages 47–96. IOS Press, 2001.

[69] B. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[70] A. Roscoe, C. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[71] V. Saraswat, K. Kahn, and J. Levy. Janus: A Step Towards Distributed Constraint Programming. In *Proc. of the 1990 North American Conference on Logic Programming*, pages 431–446, 1990.

[72] V. Sassone, M. Nielsen, and G. Winskel. A Classification of Models for Concurrency. In *CONCUR '93, Proc. of the 4th International Conference on Concurrency Theory, Hildesheim, Germany*, volume 715 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1993.

[73] M. Schrefl and M. Stumptner. Behavior Consistent Extension of Object Life Cycles. In *OOER '95: Object-Oriented and Entity-Relationship Modeling*, volume 1021 of *Lecture Notes in Computer Science*, pages 133–145. Springer Berlin Heidelberg, 1995.

[74] S. Shlaer and S. Mellor. *Object Life Cycles - Modeling the World in States*. Yourdon Press/Prentice Hall, 1992.

[75] A. Simons, M. Stannett, K. Bogdanov, and W. Holcombe. Plug And Play Safely: Rules For behavioural compatibility. In *Proc. of the 6th IASTED International Conference Software Engineering and Applications*, pages 263–268, 2002.

[76] W. van der Aalst and T. Basten. Inheritance of Workflows: An approach to tackling problems related to change. *Theoretical Computer Science*, 270(1-2):125–203, 2002.

[77] W. Van der Aalst, A. Ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[78] M. Verhelst. 50 Years of Software Development for Information Systems. Some Milestones. *Review of Business and Economics*, L(4):595–610, 2005.

[79] J. Verheul and E. Roubtsova. An Executable and Changeable Reference Model for the Health Insurance Industry. In *Proc. of the Third Workshop on Behavioural Modelling*, BM-FA '11, pages 33–40, 2011.

[80] L. Wischik and P. Gardner. Explicit Fusions. *Theoretical Computer Science*, 340(3):606–630, 2005.