On the refinement of state-based and event-based models

Christie Bolton
New College

Thesis submitted for the degree of Doctor of Philosophy

Hilary Term, 2002

On the refinement of state-based and event-based models

Christie Bolton, New College

Hilary Term, 2002

Abstract

The specification and development of a large, complex system might involve the construction of a variety of models, each of which might address a different aspect of design. Where two or more models are proposed for the same component, it is useful to be able to compare them, and check that the views they present are consistent. If the models are constructed in the same language, this comparison is a question of refinement. If different languages are employed, then some translation is required.

This thesis investigates the problem of translation between two kinds of model: those in which systems are described in terms of states, and transformations upon them; and those in which systems are described in terms of the occurrence and availability of events. It shows how models constructed in the former, state-based approach can be translated into, and thus compared within, an event-based framework.

The notation chosen for state-based models is that of Z; the event-based framework is derived from the theory of Communicating Sequential Processes (CSP). A theory of event-based refinement, using a new semantic model for CSP, is developed to match the theory of refinement for abstract data types written in Z: in this model, process refinement corresponds exactly to data refinement.

The resulting theory of refinement can be used to show consistency and refinement of object-oriented specifications. The thesis shows how both state-based and event-based descriptions, written in the graphical language UML might be translated into either Z or CSP, and compared using the theory of data refinement.

Acknowledgements

I am greatly indebted to my supervisor Jim Davies, for his guidance, encouragement and enthusiasm throughout the course of my M.Sc and D.Phil; and to Ellie for being a loyal and true friend and enduring more mathematical conversations than she cares to remember. I would also like to extend my gratitude and thanks to Jim Woodcock and David Walker for their encouragement and for always taking an interest in my work and progress. In addition, I am especially grateful to Jeff Sanders for all his helpful comments and useful suggestions. I would also like to thank my train-spotter, Andy, for diligently proof-reading in record time. Railtrack needs you! Thanks also to EPSRC who gave me the funding that enabled me to embark on this quest. I am also most grateful to my examiners Gavin Lowe and Michael Butler for their insightful comments and help in making this a more polished document.

Finally I would like to thank my family and friends for all their love, care and support over the past few years. All that despite the fact most of them still have no clue what I do!

Contents

List of Figures

Chapter 1

Introduction

Computers play a critical role in our lives today and our reliance on them will only increase. They are used by our banks and our hospitals; our planes, trains and automobiles; our chemical plants and air traffic control systems; our phones and our household appliances. As functionality increases so too does complexity; and with it the likelihood of introducing errors into the system. In critical systems these errors can be very expensive. The cost might be measured in the loss of time or money, damage to the environment or even loss of human life [Lev95].

Formal methods have been developed to enable software engineers to reason about systems at the most abstract level, free from language-specific, implementation-level constraints. Mathematical models can be constructed, focusing only on the properties and behaviour of the system. As Hoare [Hoa94] states:

> A specification describes the observable properties and behaviour of some system that does not yet exist in the physical world; and the goal is to design and implement a product which can be predicted by the theory to meet the specification.

During the step-wise hierarchical process of program development a software engineer might use *refinement* to prove formally that each model satisfies the properties of the previous, more abstract, description of the system. De Roever and Engelhardt [dRE98] explain how iterative refinement techniques can be used to determine that a product does indeed meet its specification:

> Designing a large and complex program usually involves application of some refinement method providing a way to *gradually transform* an *abstract* program, possibly a *specification*, into a *concrete* implementation. The main principle of such a method is that if the initial abstract program is correct and the *transformation steps* preserve correctness, then the resulting implementation will be correct by construction.

Refinement must be a reflexive and transitive relation. If each successive model is a refinement of the previous one then it follows automatically that the final implementation will be a refinement of the original specification.

The formal methods community have proposed various approaches to modelling; we often find these classified as either *state-based* or *event-based*. Examples of state-based approaches include those of Z [Spi92], Object-Z [DRS95], B [Abr96] and the Vienna Development Method (VDM) [Jon90]. Examples of event-based approaches include those of Communicating Sequential Processes (CSP) [Hoa85], Algebra for Communicating Processes (ACP) [BK85] and Calculus for Communicating Systems (CCS) [Mil80, Mil89].

The information contained within models describing the same system in state-based and event-based approaches might be the same, but the means of reasoning is different. Each method has its own tools and techniques for analysing models. Thus, the questions we can ask about the system depend upon the approach that we have chosen. We address this issue: we present a framework for comparing state-based and event-based descriptions of the same system and we clarify the relationship between data refinement and process refinement.

The primary focus of our comparison relates the languages Z and CSP. We present a translation from abstract data types expressed in Z to their corresponding processes expressed in CSP and we explore the information content of their respective semantic models. We prove that, contrary to accepted results, for divergence-free systems data refinement is *strictly weaker* than stable failures refinement. Both semantic models record possible sequences of interaction and both record availability of events or operations after each such sequence. The difference between them is that the relational semantics of data types records availability only on an operation-by-operation basis, whereas the stable failures semantic model records availability of *combinations* of events.

Having identified the disparity between these semantic models, there are two options available if we wish to be able to compare data types and processes. We must either identify a failures-divergences semantic model for data types, or we must propose an alternative semantic model for CSP, a model in which availability is recorded on an event-by-event basis. Extensive research has been undertaken on the former approach. We discuss this further in our conclusions and we present sets of sound and jointly complete simulation [HHS87] rules defining such a model. However, our primary focus in this thesis is the previously un-explored problem of identifying a semantic model for Communicating Sequential Processes that coincides with the relational semantics of data types. We define such a model, the *singleton failures* semantic model, and investigate the properties of the associated refinement ordering.

The singleton failures semantic model merits investigation for two reasons. Firstly it gives another means of reasoning about data types. It enables us to ask different questions about their behaviour with the added advantage that, having translated the data types concerned to their corresponding processes, these questions can be answered automatically using the model-checker FDR [For97]. Yet we gain more than that. We gain a means of establishing the refinement of a specific subclass of processes using the simulation rules of data types. For single-threaded programs we do not need the strength of the stable failures, or even the failures-divergences model. The singleton failures model is precisely what we are looking for; hence the simple simulation rules give us all the information we need.

Our investigation into the properties of the singleton failures semantic model brings various issues to the fore. One approach would have been to take a projection of the stable failures semantic model, recording only trace-refusal pairs whose cardinality is at most one. The main drawback of such an approach is that the resulting semantic model does not define a congruence: applying the natural projection function, we cannot express the singleton failures of a process after hiding in terms of the singleton failures of the original process. An alternative approach, and the one adopted in this thesis, is to define a semantic function that remains consistent where possible with the stable failures semantic yet still yields a congruence. We do not define our hiding operator to be true to our definition of the hiding operator within the stable failures model. Rather, we define it to be as close as possible to the intended meaning whilst ensuring that the resulting model defines a congruence. Our resultant definition of hiding brings with it additional complications. As Roscoe [Ros98] observes, within the stable failures model each operator other than the hiding operator is fully distributive[1], and hence monotonic, in every argument; moreover, although not *fully* distributive, the hiding operator is *finitely* distributive, and hence monotonic, over the stable failures model. This property does not extend to the singleton failures semantic model: the hiding operator is neither fully nor finitely distributive. However, more positively, we prove that *at the outermost level* hiding preserves refinement.

We have claimed that our refinement-preserving translation from data types to their corresponding processes has given us another tool for reasoning about state-based systems. We prove this by showing how, following the application of parallel combination, we can use the stable failures refinement ordering—and hence the model-checker FDR [For97]—to verify singleton failures refinement.

In this thesis we do not restrict our comparison of state-based and behavioural models to algebraic specification languages. We turn our attention to the graphical Unified Modeling Language (UML) [RJB97]. We describe the steps necessary for building a modelling tool that can check automatically that the final class description of a system is consistent with diagrams constructed during its specification. Our goal is to give users the strength of more formal specification techniques without requiring them to learn the corresponding syntax and semantics. We present a formal semantics for activity graphs, a notation which combines the features of state charts [Har87] with the intuition of Petri Nets [Pet81] and can be used to describe the flow of behaviour. We then proceed to outline how this, together with our translation from Z to CSP and the work of others in the field, might be combined to realise our goal.

[1] Roscoe [Ros98] defines an operator to be *fully distributive* in each of its arguments over a semantic model M if $F(P \sqcap Q) \equiv_M F(P) \sqcap F(Q)$ and if $F(\bigsqcap S) \equiv_M \bigsqcap\{F(P) \mid P \in S\}$ for each non-empty S where $F(\cdot)$ represents any unary function produced by fixing all arguments of the operator other than that currently under consideration.

He defines an operator to be *finitely distributive* in each of its arguments over a semantic model M if $F(P \sqcap Q) \equiv_M F(P) \sqcap F(Q)$ and if $F(\bigsqcap S) \equiv_M \bigsqcap\{F(P) \mid P \in S\}$ for each *finite*, non-empty S where $F(\cdot)$ represents any unary function produced by fixing all arguments of the operator other than that currently under consideration.

1.1 Semantic interpretation

Intuitively it is understood that one model is a *refinement* of another if the first conforms to the second whilst clarifying some ambiguities; perhaps some of the non-determinism has been resolved. Yet, when modelling systems, we must be more precise about our definition of refinement.

Some operations might be only partially defined and we must be clear about the information that this partial specification tacitly imparts. Does it mean that the operation may not be called outside the current domain of definition? Or does it mean that we have not yet considered the effect of the operation outside the current domain of definition; that we will address that part of the specification at a later stage? These two interpretations are respectively referred to as the blocking and non-blocking semantic models [DB01]. The latter model is often used for descriptions proposed at an early stage of the development process, whereas the former model is used once the domains of all operations have been identified. The non-blocking version of the relational semantics for simple non-communicating data types, as described in Chapter 4, along with its associated sets of sound and jointly complete simulation rules, was proposed by Woodcock and Davies [WD96]. The blocking version of the relational semantics, for the same simple non-communicating data types again with its associated sets of sound and jointly complete simulation rules, was given by the author [BDW99].

In this thesis we perform all of our explorations and provide all of our results in the context of both the blocking and non-blocking models. We present blocking and non-blocking versions of the relational semantics for communicating data types and derive sets of sound and jointly complete simulation rules for each. In addition, we provide translations from data types to processes that are consistent with both interpretations. In each case we prove that data refinement is not equivalent to stable failures refinement but that it is equivalent to refinement within our newly proposed model, the singleton failures semantic model.

1.2 Related work

Many languages have been developed for formal specification. We have observed that two of the predominant approaches for modelling and analysing systems are event-based specification languages, or process algebras, and state-based specification languages. Aptly named, the state of the system is made explicit within state-based approaches. Operations are defined over the state-space and their ordering can only be inferred. Conversely, the ordering of events is made explicit within event-based approaches and the state may not be directly accessed.

1.2.1 Behavioural models

Process algebras can be given one or more of: an algebraic semantics; an operational semantics; or a denotational semantics. An *algebraic*, or *axiomatic*, semantics comprises a collection of laws, or axioms, that capture the properties of the operators: whether they are symmetric, associative, commutative, or idempotent; whether one operator distributes over another. Process equivalence can then be derived from these laws. An example of a process algebra that takes this approach is ACP [BK85].

A *denotational* semantic model describes processes in a different fashion. The model is defined by a function that maps the language onto a mathematical description of its behaviour. A denotational semantic model is normally a *congruence*: given a semantic function $\mathcal{D}$ that defines the model, any (binary) operator $\oplus$ and any processes P and Q, the set $\mathcal{D}(P \oplus Q)$ is normally expressible in terms of the sets $\mathcal{D}(P)$ and $\mathcal{D}(Q)$. The traces model of CSP [Hoa85] is an example of a process algebra that takes this approach.

Within an *operational* semantic model programs are modelled as *labelled transition systems*. A labelled transition system (LTS) can be represented by a directed graph in which machine states are represented by the nodes and events or actions are represented by edges. CCS [Mil80] is an example of a process algebra that has been given an operational semantics. Process refinement within CSP [Ros98] can be established automatically [For97] by exhaustive exploration of the operational semantics of the processes concerned.

A key difference between CSP and both CCS and ACP is that CSP has two choice operators, distinguishing between internal and external choice, whereas CCS and ACP have a single choice operator that represents an external choice. In CCS and ACP the effect of an internal choice can be achieved by preceding the events with the event τ, representing unobservable communication.

The π*-calculus* [SW01] is a process algebra with its roots in CCS. The key difference between the π-calculus and the other process algebras that we have considered is that it allows channel names to be created dynamically and to be passed along other channels. Processes expressed in the π-calculus are said to be *mobile* because their ability to communicate can change dynamically over time.

1.2.2 State-based models

A *predicate* [Nel89] is a subset of states from a given state space with a top element, true, being the whole state space and a bottom element, false, being the empty set. Two predicates are equivalent if the sets are equal and one *entails* another if the first is a subset of the second. A predicate transformer is a function from predicates to predicates.

In 1975 Dijkstra [Dij75] proposed the *guarded command language* for modelling sequential imperative programs. A guard is a predicate that can be evaluated to either true or false; a command is a program fragment. Guarded commands can be given a *weakest precondition* semantics [Dij76]. The weakest precondition predicate transformer takes a statement and corresponding postcondition and returns the largest set of states from which the postcondition is guaranteed after execution of the given statement. Back, Morgan and Morris [Bac80, Mor88, Mor94, Mor87] extend this work to develop the *refinement calculus*. Statement S satisfies specification $(pre, post)$ provided the predicate pre entails the predicate $wp(S, post)$. Refinement might be achieved, for example, by strengthening the postcondition or by weakening the precondition.

Data refinement can be used to compare systems having different state spaces, relating abstract variables and concrete variables with a predicate transformer. Refinement can be established either directly [Bac90, MRG88] or inductively using simulation rules [HHS87].

The state-based specification language Z [Spi92] is based on typed set theory coupled with a structuring mechanism: the schema. Schemas can be used to define the state-space as well as dynamic aspects of the language: the effect of operations on the state. Object-Z [Smi00] is an object-oriented extension of Z that includes notions of class, instance, inheritance and polymorphism. As Z and Object-Z employ state, operation, and class schemas, so B [Abr96] and VDM [Jon90] employ abstract machines.

State-based systems can alternatively be modelled by *transition systems* [Pnu86, Lam89] and represented by a tuple recording: a finite or infinite set of states; a finite or infinite set of transitions; a function mapping each transition to a relation on the state space; and a non-empty set of initial states. Examples of languages that take this approach are Chandy and Misra's UNITY [CM88] and the *actions systems* formalism of Back and Kurki-Suonio [BKS83]. An action, a guarded command as defined by Dijkstra [Dij76], is enabled precisely when its guard evaluates to true. In the execution of an action system first the initialisation is executed: its command is performed. Then, chosen enabled actions are repeatedly executed. If no action is enabled then the system *deadlocks*; if the command of an enabled action aborts then the system *diverges*.

1.2.3 Combining state-based and behavioural formalisms

Combining state-based and behavioural formalisms is by no means a new area of research: these issues have been addressed for well over a decade. In [Jos88] Josephs adopts a state-based approach to the refinement and specification of non-divergent CSP processes; Morgan [Mor90] defines a correspondence between action systems [BKS83] and CSP by expressing the traces, failures and divergences of CSP processes as weakest precondition formulae [Dij76]; Butler [But92] addresses the issue of unbounded nondeterminism and extends Morgan's correspondence [Mor90] to the infinite traces model; and He [He89], like Josephs, expresses CSP processes as labelled transition systems and presents sets of sound and complete simulation rules. Woodcock and Morgan [WM90] show also how refinement can be established using simulation rules, this time in the context of the action systems formalism [BKS83]. In each case refinement within the state-based formalism is shown to be equivalent either to the stable failures refinement or the failures-divergences refinement of CSP.

More recently, Butler [But00] links CSP and B; Galloway, Taguchi and Araki [GS97, TA97] link Z and CCS; and Derrick and Smith [SD97, FS97, Smi97, DS00] link Z or Object-Z with CSP. In [Fis97], Fischer proposes a hybrid language, CSP-OZ in which the CSP components of a specification are incorporated into the appropriate classes. Smith, Derrick and Boiten [SD97, DB01] present two means of comparing processes: refinement can be established either by using the Smith's histories semantic model or by using a set of simulation rules. It turns out that these give rise to different refinement orderings, and we explore their differences further in our conclusions.

In our conclusions we discuss the similarities and differences between the work presented in this thesis and the various means of combining state-based and event-based formalisms mentioned above.

1.3 Scope

Our intention is to provide a framework for comparing abstract data types expressed in Z and processes expressed in CSP. We achieve this by introducing a denotational semantics for processes that corresponds exactly to the relational semantics characterising data refinement. However, there are various issues that remain unresolved: issues beyond the scope of this thesis, that we will discuss in our conclusions but will leave for future work. These issues concern scalability and the comparison of state-based and behavioural models built from smaller components.

When reasoning about large systems it will often be desirable or even necessary to reason at various levels of abstraction. Initial descriptions may involve just a few components and their interactions, but as we approach an implementation-level description of the system we might need to break these components down into subcomponents as

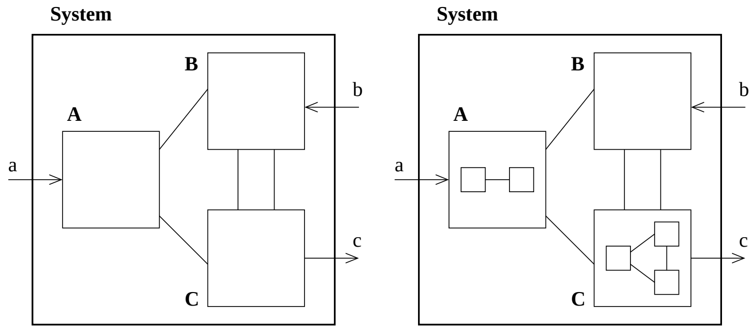

Figure 1.1: Building systems from components.

illustrated in Figure 1.1. To verify that the implementation is consistent with the initial high level description our intuition tells us to put the various components in parallel, to hide any internal events or operations, and then to do an appropriate refinement check. It does not turn out to be that simple.

Comparing state-based and behavioural models of large systems built from the parallel combination of smaller components is relatively straightforward for simple, non-communicating data types and their corresponding processes. However, to cope with passing values within shared events or operations, we need to adapt our definitions of communicating data types and their corresponding processes to cope with passing values. In our current model input is received and then output is given. We need to be more flexible.

1.4 Outline

The objective of this thesis is to provide a framework for comparing state-based and behavioural models of systems, both graphical and algebraic. In achieving our objective we define a new semantic model for CSP.

In Chapter 2 we present the standard notation that we will be using throughout the rest of the thesis. We introduce the syntax used within CSP along with two standard semantic models—the traces model and the stable failures model—and their associated refinement orderings and healthiness conditions. In addition, assuming knowledge of set theory and predicate logic, we introduce the constructs particular to Z that we will assume throughout the rest of the thesis.

We propose in Chapter 3 a new semantic model for Communicating Sequential Processes, the *singleton failures* semantic model. We prove that all properties expressible only in terms of *basic* processes—process constructed using only the prefix operator, the choice operators, parallel composition and the terminating process *Stop*—that hold within the stable failures model hold also within the singleton failures model. We identify problems that arise with hiding, for instance the hiding operator is not distributive with respect to the internal choice operator, but prove the more positive result that hiding preserves refinement. We conclude the chapter by showing how singleton failures refinement might be verified automatically.

In the subsequent two chapters we prove that data refinement is equivalent to singleton failures refinement. Chapter 4 is an exploratory chapter and as such we take a very simplistic approach to our data types. We consider only abstract data types with no explicit notion of input or output. We claim that the *environment*—which provides an initial input and, after performing a sequence of operations, receives a final output—is a convenient fiction through which we can record all communications. We prove first that, for these simple data types, data refinement is not, as had previously been supposed, equivalent to stable failures refinement. We then proceed to prove that it is equivalent to singleton failures refinement.

In Chapter 5 we build on the ideas developed in the previous chapter in the more complicated arena where inputs and outputs are explicit. We define communicating data types, give them a relational semantics and derive sets of sound and jointly complete simulation rules. We provide translations from our communicating data types to their corresponding processes, one in the context of the blocking semantics and one in the context of the non-blocking semantics. We prove that these translations preserve refinement: that refinement of communicating data types is equivalent to the singleton failures refinement of their corresponding processes.

We show, in Chapter 6, how the work presented in the previous chapters might be applied to the formalisation of the graphical modelling language UML. We give a formal semantics to activity graphs, a notation used to describe the flow of behaviour, modelling them as CSP processes. Our long term objective is to provide the means to develop a modelling tool that automatically compares the model encapsulated by the final class description with the behaviour conveyed by diagrams constructed during the development process. Such a tool would then give the user intelligible reports about where or if the specification has been violated.

We conclude the thesis with a discussion of what we have achieved, of related work done by others, and of areas of interest for further investigation.

Chapter 2

Communicating Sequential Processes and the Z Notation

As early as 1978, Tony Hoare [Hoa78] introduced the idea of a *process* as a model to describe the interactions between a computing system and its environment. He and others then went on to develop the ideas presented in this seminal paper leading to the development of various event-based specification languages: languages in which behaviour is described in terms of the ordering of events. Three well known examples of such languages are Hoare's Communicating Sequential Processes (CSP) [Hoa85], Bergstra and Klop's Algebra for Communicating Processes (ACP) [BK85], and Milner's Calculus for Communicating Systems (CCS) [Mil80, Mil89].

A *process*, as defined in [Hoa85], is a pattern of communication. We can use processes to represent components in terms of their communicating behaviour, modularising descriptions using the standard operators of the CSP language. There are various ways of describing the behaviour of CSP processes: we can give them an *algebraic semantics* defined by a set of algebraic laws describing the effects of each of the operators; we can give them an *operational semantics* describing programs using transition diagrams; or we can give them a *denotational semantics* mapping the language into an abstract model based on sets of behaviours.

The state-based specification language Z [Spi92], a language based on typed set theory, was developed in Oxford University's Computing Laboratory in the early eighties and came to the fore during collaboration with IBM Hursley on the CICS system. The language uses propositional and predicate calculus to describe relationships between mathematical objects or components. The components, their associated properties and operations can be captured using *schemas*.

This chapter describes constructs particular to Z as well as two established denotational semantic models for CSP, the traces model and the stable failures model that we will refer to throughout the thesis. The material is standard, and is described in full by Spivey, Woodcock and Davies [Spi92, WD96] and by Hoare and Roscoe [Hoa85, Ros98]. It is included here to facilitate an in-depth comparison between the properties of these established semantic models for CSP and the new model that we propose in Chapter 3.

2.1 CSP Syntax

Processes are defined in terms of *events*: synchronous, atomic communications between a process and its environment. Compound events can be constructed using '.' the dot operator; a family of compound events is called a *channel*. Channels can be used to represent the passing of a value between components. Processes can also be defined by sets of mutually-recursive equations; these equations can be indexed to allow parameterised definitions. Often these will be defined locally using a 'let ... within ...' clause.

Parameters can be used to represent aspects of the process state, and can appear in *guards*: we write *bool* & P to denote the process that behaves as P if Boolean *bool* is true, and can perform no events otherwise. The atomic processes *Skip* and *Stop* both denote termination; *Skip* denotes successful termination—the end of a pattern of communication—whereas *Stop* denotes deadlock. The most basic operator is the *prefix operator*; if P is a process and a an event then $a \rightarrow P$ denotes a process that is initially ready to engage in a. If this event occurs, then subsequent behaviour will be that of P.

There are two types of *choice operators* in CSP: one denoting internal choice and the other denoting external choice. An *internal choice* is resolved between the processes involved without reference to the environment, whereas an *external choice* may be influenced by the environment and is resolved by the first event to occur. These operators each come in two forms: the binary operator; and the choice over an indexed set of processes. Given processes P and Q and a set of processes $R(i)$ indexed over some set I, the following processes are as described:

$P \sqcap Q$	an internal choice between P and Q;
$P \,\square\, Q$	an external choice between P and Q;
$\sqcap\, i : I \bullet R(i)$	an internal choice over the indexed set of processes;
$\square\, i : I \bullet R(i)$	an external choice over the indexed set of processes.

Processes can be composed *sequentially*: if P and Q are both processes, then the process $P \mathbin{;} Q$ first behaves as P and then, if P successfully terminates, behaves as Q. Alternatively processes can be composed in *parallel*. We write $P \;{}_X\|_Y\; Q$ to denote the process in which P and Q can evolve independently but must synchronise on every event in the set $X \cap Y$. The set X is the *alphabet* of P and the set Y is the alphabet of Q. No event in X can occur without the cooperation of P and, likewise, no event in Y can occur without the cooperation of Q. We write $\|\, i : I \bullet [A(i)]\, P(i)$ to denote an *indexed parallel* combination of processes in which each process $P(i)$ can evolve independently but must synchronise upon every event from the set $A(i)$. The combination may terminate only when every process is ready to do so. In an *interleaving* parallel combination no synchronisation is required; in the combination $P \;|||\; Q$, the two processes evolve independently. An indexed version of the interleaving operator is also employed: $|||\, i : I \bullet P(i)$. As with ordinary parallel combination, termination requires the agreement of all parties.

Finally, we introduce the *hiding operator*: if P is a process and A is a set of events then $P \setminus A$ is a process that behaves as P, except that both the requirement to synchronise upon and the ability to observe events from the set A has been removed.

When reasoning about possible sequences of interaction we will use filter ($\upharpoonright$) and hiding ($\setminus$) operators on sequences. Given a sequence s and a set A, the sequence $s \upharpoonright A$ is the longest subsequence of s containing only elements from the set A. Conversely the sequence $s \setminus A$ is the longest subsequence of s containing no elements from the set A.

2.2 Denotational semantics for CSP

In Section 2.1 we introduced the syntax of CSP. In this section we will describe two denotational semantic models, the *traces* model and the *stable failures* model as defined by Roscoe [Ros98]. A *trace* records a history of communication between a process and the environment. It is expressed as a sequence of events in which the process has engaged up to a particular point in time with the head of the sequence being the first event that was communicated. Given a process P, the set $\mathcal{T}\,[\![P]\!]$ is the set of all possible finite traces of P. If *Event* is the set of all possible events, and if $\mathit{Trace} == \mathrm{seq}\,\mathit{Event}$ then the semantic domain of the traces model is the set $\mathbb{P}\,\mathit{Trace}$.

Although the set of all possible traces of a process carries a great deal of information, it does not record the availability of events. This information is preserved in *refusal sets*, sets of events from which a process in a stable state can refuse to communicate anything no matter how long it is offered. Given a process P the set $\mathit{refusals}(P)$ is the set of P's initial refusals. For instance, the process $a \rightarrow P$ can initially refuse any set of events that does not contain the event a; the process $a \rightarrow P \,\Box\, b \rightarrow Q$ can initially refuse any set of events that contains neither a nor b whereas the process $a \rightarrow P \sqcap b \rightarrow Q$ can initially refuse any set of events that does not contain both a and b. The process *Stop* can refuse any set of events. More formally, given the set $\mathit{Refusal} == \mathbb{P}\,\mathit{Event}$,

$$\begin{aligned}
\mathit{refusals}(\mathit{Stop}) &= \mathbb{P}\,\mathit{Event} \\
\mathit{refusals}(a \rightarrow P) &= \{\mathit{ref} : \mathit{Refusal} \mid a \notin \mathit{ref}\} \\
\mathit{refusals}(a \rightarrow P \,\Box\, b \rightarrow Q) &= \{\mathit{ref} : \mathit{Refusal} \mid \{a, b\} \cap \mathit{ref} = \emptyset\} \\
\mathit{refusals}(a \rightarrow P \sqcap b \rightarrow Q) &= \{\mathit{ref} : \mathit{Refusal} \mid \{a, b\} \not\subseteq \mathit{ref}\}.
\end{aligned}$$

A *failure* is a pair in which the first element is the trace of a process and the second is a refusal set of the process after the given trace. Given a process P and the set *Event* of all possible events, the set of failures of P, $\mathcal{F}\,[\![P]\!]$, is formally defined as follows.

$$\mathcal{F}\,[\![P]\!] \;=\; \{\mathit{tr} : \mathit{Trace};\ \mathit{ref} : \mathit{Refusal} \mid \mathit{tr} \in \mathcal{T}\,[\![P]\!] \wedge \mathit{ref} \in \mathit{refusals}\,(P \,/\, \mathit{tr})\}$$

where $P \,/\, \mathit{tr}$ is the process that models the behaviour of P after trace tr has been completed. The semantic domain of the failures model is then the set $\mathbb{P}(\mathit{Trace} \times \mathit{Refusal})$. A *divergence* is the possibility of a process entering into an infinite sequence of internal actions. Divergence is most commonly introduced into a process by hiding.

Roscoe introduces three denotational semantic models for CSP: the traces model ($\mathcal{T}$); the stable failures model ($\mathcal{F}$); and the failures-divergences model ($\mathcal{N}$). In this thesis we consider only divergence-free processes and so will not work within the context of the failures-divergences model.

Traces model

For any given processes P and Q, event a and sets of events X and Y, the semantic function $\mathcal{T}$ satisfies the laws given below. The process *Stop* will never interact with its environment. The process $a \to P$ can initially perform only the event a, and subsequent behaviour is that of P, and the processes $P \,\Box\, Q$ and $P \sqcap Q$ have the same sets of traces. They can both behave as either P or Q. In the parallel combination $P \;{}_X\!\parallel_Y\; Q$, the occurrence of any event in the set X must be agreed by the process P and likewise the occurrence of any event in the set Y must be agreed by the process Q: any possible sequence of interaction restricted to those events in the set X must be a possible trace of P and similarly any possible sequence of interaction restricted to those events in the set Y must be a possible trace of Q. Furthermore, any such sequence must contain only events from the union of sets X and Y. A sequence of events is a possible behaviour of the process $P \setminus X$ only if there is a possible trace of P which when restricted to those events not in X gives the desired trace.

$$
\begin{array}{rcl}
\mathcal{T}\,[\![\mathit{Stop}]\!] & = & \{\langle\rangle\} \\
\mathcal{T}\,[\![a \to P]\!] & = & \{tr : \mathcal{T}\,[\![P]\!] \bullet \langle a \rangle \frown tr\} \;\cup\; \{\langle\rangle\} \\
\mathcal{T}\,[\![P \,\Box\, Q]\!] & = & \mathcal{T}\,[\![P]\!] \;\cup\; \mathcal{T}\,[\![Q]\!] \\
\mathcal{T}\,[\![P \sqcap Q]\!] & = & \mathcal{T}\,[\![P]\!] \;\cup\; \mathcal{T}\,[\![Q]\!] \\
\mathcal{T}\,[\![P \;{}_X\!\parallel_Y\; Q]\!] & = & \{tr : \mathit{Trace} \mid \operatorname{ran} tr \subseteq X \cup Y \;\wedge \\
 & & \qquad tr \upharpoonright X \in \mathcal{T}\,[\![P]\!] \;\wedge\; tr \upharpoonright Y \in \mathcal{T}\,[\![Q]\!]\,\} \\
\mathcal{T}\,[\![P \setminus X]\!] & = & \{tr : \mathit{Trace} \mid tr \in \mathcal{T}\,[\![P]\!] \bullet tr \setminus X\}
\end{array}
$$

Stable failures model

For any given processes P and Q, event a and sets of events X and Y, the function $\mathcal{F}$ satisfies the laws given below. The process *Stop* will never interact with its environment and can initially refuse any set of events. The process $a \to P$ can initially refuse any set of events not containing a. If event a does occur then subsequent behaviour is that of P. The process $P \,\Box\, Q$ can initially refuse only those sets of events that could be refused initially by *both* P and Q whereas the process $P \sqcap Q$ can initially refuse all sets of events that could be refused by *either* P or Q. After the first event has occurred their behaviour is equivalent.

In the process $P \;{}_X\!\parallel_Y\; Q$, process P can influence the occurrence (or refusal) only of those events in set X. Similarly Q can influence the occurrence (or refusal) only of those events in Y. Furthermore, to ensure consistency with the traces model, any possible sequence of interaction restricted to those events in the set X must be a possible trace of P and any possible sequence of interaction restricted to those events in the set Y must be a possible trace of Q. In the failures semantics, a failure of P makes a contribution to the semantic set of $P \setminus A$ only if the refusal set contains every event from A.

$$
\begin{aligned}
\mathcal{F}\,[\![Stop]\!] &= \{ref : Refusal \bullet (\langle\rangle, ref)\}\\
\mathcal{F}\,[\![a \rightarrow P]\!] &= \{ref : Refusal \mid a \notin ref \bullet (\langle\rangle, ref)\}\\
&\quad \cup\\
&\quad \{tr : Trace;\ ref : Refusal \mid (tr, ref) \in \mathcal{F}\,[\![P]\!] \bullet (\langle a\rangle \frown tr, ref)\}\\
\mathcal{F}\,[\![P \mathbin{\Box} Q]\!] &= (\,\mathcal{F}\,[\![P]\!] \cap \mathcal{F}\,[\![Q]\!]\,) \ \cap\ \{tr : Trace;\ ref : Refusal \mid tr = \langle\rangle\}\\
&\quad \cup\\
&\quad (\,\mathcal{F}\,[\![P]\!] \cup \mathcal{F}\,[\![Q]\!]\,) \ \cap\ \{tr : Trace;\ ref : Refusal \mid tr \neq \langle\rangle\}\\
\mathcal{F}\,[\![P \sqcap Q]\!] &= \mathcal{F}\,[\![P]\!] \ \cup\ \mathcal{F}\,[\![Q]\!]\\
\mathcal{F}\,[\![P\,{}_X\!\parallel_Y Q]\!] &= \{tr : Trace;\ ref, ref_P, ref_Q : Refusal \ \mid\\
&\qquad \operatorname{ran} tr \subseteq X \cup Y\\
&\qquad \wedge\\
&\qquad ref \cap (X \cup Y) \ = \ (ref_P \cap X) \cup (ref_Q \cap Y)\\
&\qquad \wedge\\
&\qquad (tr \upharpoonright X, ref_P) \in \mathcal{F}\,[\![P]\!] \wedge (tr \upharpoonright Y, ref_Q) \in \mathcal{F}\,[\![Q]\!] \bullet (tr, ref)\}\\
\mathcal{F}\,[\![P \setminus X]\!] &= \{tr : Trace;\ ref : Refusal \mid\\
&\qquad (tr, X \cup ref) \in \mathcal{F}\,[\![P]\!] \bullet (tr \setminus X, ref)\}
\end{aligned}
$$

2.3 Refinement and equivalence of processes

The refinement orderings induced by both the traces model and the stable failures model are based upon reverse containment. One process is traces-refined by another if every trace of the second is also a trace of the first. Similarly, one process is failures-refined by another if every failure of the second is also a failure of the first. Given processes P and Q we write

$$
\begin{aligned}
P \ \sqsubseteq_{\mathcal{T}} \ Q \ &\Leftrightarrow \ \mathcal{T}\,[\![Q]\!] \ \subseteq \ \mathcal{T}\,[\![P]\!]\\
P \ \sqsubseteq_{\mathcal{F}} \ Q \ &\Leftrightarrow \ \mathcal{F}\,[\![Q]\!] \ \subseteq \ \mathcal{F}\,[\![P]\!]
\end{aligned}
$$

Within both these models processes are defined to be equivalent if each refines the other.

$$
\begin{aligned}
P \ \equiv_{\mathcal{T}} \ Q \ &\Leftrightarrow \ P \sqsubseteq_{\mathcal{T}} Q \ \wedge \ Q \sqsubseteq_{\mathcal{T}} P\\
P \ \equiv_{\mathcal{F}} \ Q \ &\Leftrightarrow \ P \sqsubseteq_{\mathcal{F}} Q \ \wedge \ Q \sqsubseteq_{\mathcal{F}} P
\end{aligned}
$$

The two models are consistent in that the pair (tr, ref) can be a failure of a process P only if the sequence tr is a possible trace of P. Hence, one process is a refinement of another within the failures model only if it is also a refinement of the second within the traces model. It follows directly from this that if two processes are equivalent within the failures model they will also be equivalent within the traces model.

$$
\begin{aligned}
P \sqsubseteq_{\mathcal{F}} Q \ &\Rightarrow \ P \sqsubseteq_{\mathcal{T}} Q\\
P \equiv_{\mathcal{F}} Q \ &\Rightarrow \ P \equiv_{\mathcal{T}} Q
\end{aligned}
$$

2.4 Healthiness conditions for CSP

Given a set T of traces, the healthiness conditions for the traces model are as follows:

$$\begin{aligned}
&\forall\, tr_1, tr_2 : \mathit{Trace} \;\bullet \\
&\qquad \langle\rangle \in T \quad \wedge && \text{(T1)} \\
&\qquad tr_1 \frown tr_2 \in T \;\Rightarrow\; tr_1 \in T && \text{(T2)}
\end{aligned}$$

Condition T1 states that the empty trace is a possible trace of any process, and condition T2 ensures prefix closure. Given a set F of failures, the healthiness conditions for the failures model are as follows:

$$\begin{aligned}
&\forall\, tr, tr_1, tr_2 : \mathit{Trace};\; \mathit{ref}_1, \mathit{ref}_2 : \mathit{Refusal} \;\bullet \\
&\qquad (tr_1 \frown tr_2, \emptyset) \in F \;\Rightarrow\; (tr_1, \emptyset) \in F \quad \wedge && \text{(F1)} \\
&\qquad (tr, \mathit{ref}_1 \cup \mathit{ref}_2) \in F \;\Rightarrow\; (tr, \mathit{ref}_1) \in F \quad \wedge && \text{(F2)} \\
&\qquad (tr, \mathit{ref}_1) \in F \;\wedge\; \forall\, e : \mathit{ref}_2 \bullet (tr \frown \langle e \rangle, \emptyset) \notin F && \text{(F3)} \\
&\qquad\qquad\quad \Rightarrow \\
&\qquad\qquad\qquad (tr, \mathit{ref}_1 \cup \mathit{ref}_2) \in F
\end{aligned}$$

Condition F1 is equivalent to condition T2 in the traces model: it ensures trace prefix closure. Condition F2 ensures that if a given set can be refused after a particular trace then any subset can also be refused. Finally condition F3 ensures that trace and refusal information remains consistent.

2.5 Important properties

Distributivity is an important property of the traces and failures models: the internal choice operator distributes over all other operators within both models. For instance:

$$\begin{aligned}
(\, P \sqcap Q \,) \;\Box\; R \;&\equiv_{\mathcal{F},\mathcal{T}}\; (\, P \;\Box\; R \,) \sqcap (\, Q \;\Box\; R \,) \\
(\, P \sqcap Q \,) \;{}_X\!\parallel_Y\; R \;&\equiv_{\mathcal{F},\mathcal{T}}\; (\, P \;{}_X\!\parallel_Y\; R \,) \sqcap (\, Q \;{}_X\!\parallel_Y\; R \,).
\end{aligned}$$

Distributivity of the internal choice operator facilitates the construction of large systems from smaller components whilst still preserving the properties of the smaller components. If one process is refined by another then the composition of the first process with a third is refined by the composition of the second with the third.

$$P \;\sqsubseteq_{\mathcal{F}}\; Q \;\Rightarrow\; (\, P \;{}_X\!\parallel_Y\; R \,) \;\sqsubseteq_{\mathcal{F}}\; (\, Q \;{}_X\!\parallel_Y\; R \,)$$

Another important property is the preservation of refinement after hiding: given any processes P and Q and any set of events A, if P is refined by Q then $P \setminus A$ will be refined by $Q \setminus A$.

$$P \;\sqsubseteq_{\mathcal{F}}\; Q \;\Rightarrow\; (\, P \setminus A \,) \;\sqsubseteq_{\mathcal{F}}\; (\, Q \setminus A \,)$$

2.6 The Z notation

Types and definitions Every object in a specification must have a *type*, a maximal set to which it belongs. The finite *basic types* can be defined by identifying each of the distinct members, introducing each element by name. Alternatively, a *free type*, either finite or infinite, can be introduced by simply declaring its name.

$$Type ::= element_1 \mid \ldots \mid element_n \qquad [Type]$$

Global constants can be introduced using axiomatic and generic definitions. *Axiomatic definitions* can be used to introduce a global constant which is type dependent. *Generic definitions* are used to define a *family* of definitions that are independent of type. The definition is given in terms of generic types about which no property, such as an ordering of the elements, can be assumed; the only permissible checks are equality and containment. Functions can be defined recursively using either method.

declaration
predicate

$[X_1, \ldots, X_n]$
declaration
predicate

Schema notation A schema is essentially a *pattern* containing a declaration and a constraint. Schemas can be used to describe mathematical objects, types and properties, identifying components by name rather than position. Types can be declared using the schema notation. The state space is described using a schema in which the components of the state and any constraints upon them are declared. Alternatively, schemas can be used as predicates on given types.

Name
declaration
predicate

One of the strengths of the schema notation is the level of abstraction and factorisation it allows: simply by naming the schema we can declare the components of a given type without mentioning them and without breaking the encapsulation. Furthermore, suppose that we have created a scope in which we wish to reason about the components of a particular object of a given type, we can use the operator *theta* (θ) to mention the object as a whole without having to explicitly identify it.

Lambda and mu The lambda notation can be used to introduce functions without having to name them. The type of the domain of the function is introduced first, followed by any constraints. The result is then given. This is expressed in terms of the declared value in the domain. The operator μ is used to identify a unique value. First its type is given followed by the constraints necessary to define it uniquely.

$$(\lambda\ declaration \mid constraint \bullet result) \qquad (\mu\ declaration \mid constraint)$$

2.7 Using schemas to describe abstract data types

We formally introduce *simple abstract data types* in Chapter 4 and *communicating data types* in Chapter 5 but for the time being an intuitive approach to data types will suffice: an abstract data type comprises a notion of *state* together with a set of possible *initial states* and a collection of named *operations*. These operations can be used to obtain information about the current state or to effect a change upon it.

The state space of the data type can be defined using a schema in which all attributes are declared and any constraints to be imposed on them defined. Furthermore an *Init* schema can be used to characterise the initialisation of the data type by identifying a subset of the state space, the set of possible states after initialisation. By convention, an initialisation schema will include a single, decorated copy of the state schema.

$$\begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{State}} \\ \mathit{declarations} \\ \hline \mathit{constraints} \\ \hline \end{array} \qquad \begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{Init}} \\ \mathit{State}' \\ \hline \mathit{predicate} \\ \hline \end{array}$$

Operations on the state space can be represented by a schema containing two copies of the state schema, together with a list of inputs and outputs which might themselves be written as schemas. By convention, one copy of the state is decorated with a prime; this describes the state *after* the operation has taken place. The *predicate* component within a schema describing an operation explains the relationship between the before-state, after-state, inputs, and outputs. The combination of the primed and unprimed versions of the state, *State* and *State*$'$, can be written either $\Xi\,\mathit{State}$ or as $\Delta\,\mathit{State}$. $\Xi\,\mathit{State}$ is used for operations in which the state space is unchanged whereas $\Delta\,\mathit{State}$ for operations in which the state space might be changed.

$$\begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{Op}} \\ \Xi\,\mathit{State} \\ \mathit{In};\ \mathit{Out} \\ \hline \mathit{predicate} \\ \hline \end{array} \qquad \begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{Op}} \\ \Delta\,\mathit{State} \\ \mathit{In};\ \mathit{Out} \\ \hline \mathit{predicate} \\ \hline \end{array}$$

Chapter 3

Singleton Failures Semantics

In this chapter we define an alternative semantic model for CSP, a model whose information content lies between that of the traces and the stable failures models introduced in the previous chapter. In our proposed model, the *singleton failures* semantic model ($\mathcal{S}$), as with the stable failures model, we record both trace and refusal information. The model captures refusal information on an event-by-event basis: whilst in the stable failures model we record all *combinations* of events that might be refused after a given sequence of interaction, in the singleton failures model we record availability only of individual events. Hence the name *singleton* failures.

As with the stable failures model, possible behaviours in our model are described in terms of trace-refusal pairs, the difference being that the cardinality of the refusal set is at most one. The semantic domain of $\mathcal{S}$ is therefore $\mathbb{P}(\mathit{Trace} \times \mathit{Refusal}_1)$, where the set $\mathit{Refusal}_1$ contains the empty set and all sets containing a single event.

$$\mathit{Refusal}_1 \quad == \quad \{\, e : \mathit{Event} \mid \{e\}\,\} \cup \{\emptyset\}$$

3.1 Denotational semantics for basic operators

In this section we define the singleton failures semantic function for the *basic* CSP operators, that is all operators other than the hiding operator. For each of these operators, the obvious and natural definition is indeed the one we want: we prove that for these operators, the singleton failures semantic function is a projection of the stable failures semantic function and show that many desired properties follow directly from this result.

The process *Stop* will never interact with its environment and can initially refuse any event. The process $a \rightarrow P$ can initially refuse any event other than a. If event a does occur then subsequent behaviour is that of P. The process $P \mathrel{\Box} Q$ can initially refuse only those events that could be refused by both components whereas the process $P \sqcap Q$ can initially refuse those that could be refused by either component. After the first event has occurred their behaviour is equivalent. In the parallel combination $P \mathbin{{}_X\|_Y} Q$, process P can influence the occurrence (or refusal) only of those events in set X. Similarly process Q can influence the occurrence (or refusal) only of those events in set Y. Furthermore, any possible sequence of interaction restricted to those events in the set X must be a possible trace of P and any possible sequence of interaction restricted to those events in the set Y must be a possible trace of Q.

$$\mathcal{S}\,[\![Stop]\!] = \{ref : Refusal_1 \bullet (\langle\rangle, ref)\}$$

$$\begin{aligned}\mathcal{S}\,[\![a \to P]\!] = {} & \{ref : Refusal_1 \mid ref \neq \{a\} \bullet (\langle\rangle, ref)\} \\ & \cup \\ & \{tr : Trace;\ ref : Refusal_1 \mid (tr, ref) \in \mathcal{S}\,[\![P]\!] \bullet (\langle a\rangle \frown tr, ref)\}\end{aligned}$$

$$\begin{aligned}\mathcal{S}\,[\![P \,\Box\, Q]\!] = {} & \{ref : Refusal_1 \mid (\langle\rangle, ref) \in \mathcal{S}\,[\![P]\!] \cap \mathcal{S}\,[\![Q]\!] \bullet (\langle\rangle, ref)\} \\ & \cup \\ & \{tr : Trace;\ ref : Refusal_1 \mid (tr, ref) \in \mathcal{S}\,[\![P]\!] \cup \mathcal{S}\,[\![Q]\!] \wedge tr \neq \langle\rangle\}\end{aligned}$$

$$\mathcal{S}\,[\![P \sqcap Q]\!] = \mathcal{S}\,[\![P]\!] \cup \mathcal{S}\,[\![Q]\!]$$

$$\begin{aligned}\mathcal{S}\,[\![P_X \|_Y Q]\!] = \{ & tr : \operatorname{seq} Trace;\ ref, ref_P, ref_Q : Refusal_1 \mid \\ & (\operatorname{ran} tr \subseteq X \cup Y \\ & \ \wedge \\ & \ ref \cap (X \cup Y) = (ref_P \cap X) \cup (ref_Q \cap Y) \\ & \ \wedge \\ & \ (tr \upharpoonright X, ref_P) \in \mathcal{S}\,[\![P]\!] \ \wedge\ (tr \upharpoonright Y, ref_Q) \in \mathcal{S}\,[\![Q]\!]\,) \ \bullet\ (tr, ref)\}\end{aligned}$$

Having defined in the natural way our semantic function for the basic operators we now prove that this function is consistent with the semantic function for the stable failures model. We define a *basic process* to be one that can be expressed only in terms of the process *Stop* and the operators $\to$, $\sqcap$, $\Box$ and ${}_X\|_Y$.

Theorem 1 If P is a basic process then its semantics under $\mathcal{S}$ is the projection of its semantics under $\mathcal{F}$: $\mathcal{S}\,[\![P]\!] = \mathcal{F}\,[\![P]\!] \cap (Trace \times Refusal_1)$.

Proof We can establish this by structural induction on the restricted syntax of these processes. Consider for instance the case of the parallel operator; the result for the other operators is proved similarly. With an inductive hypothesis of

$$\begin{aligned}\mathcal{S}\,[\![P]\!] &= \mathcal{F}\,[\![P]\!] \cap (Trace \times Refusal_1) \ \wedge \\ \mathcal{S}\,[\![Q]\!] &= \mathcal{F}\,[\![Q]\!] \cap (Trace \times Refusal_1)\end{aligned}$$

we prove that our theorem holds for ${}_X\|_Y$. Given trace *tr* and refusal set *ref* we reason as follows:

$$(tr, ref) \in \mathcal{S}\,[\![P\ {}_X\|_Y\ Q]\!]$$

$$\begin{aligned}\Leftrightarrow\ & (tr, ref) \in Trace \times Refusal_1 \ \wedge\ \operatorname{ran} tr \subseteq (X \cup Y) && [\text{definition of } \mathcal{S}] \\ & \wedge \\ & \exists\, ref_P, ref_Q : Refusal_1 \bullet (\, ref \cap (X \cup Y) = (ref_P \cap X) \cup (ref_Q \cap Y) \\ & \qquad \wedge \\ & \qquad (tr \upharpoonright X, ref_P) \in \mathcal{S}\,[\![P]\!] \wedge (tr \upharpoonright Y, ref_Q) \in \mathcal{S}\,[\![Q]\!]\,)\end{aligned}$$

$$\begin{array}{ll}\Leftrightarrow & (tr, ref) \in Trace \times Refusal_1 \ \land \ \mathrm{ran}\, tr \subseteq (X \cup Y) \qquad \text{[inductive hypothesis]} \\ & \land \\ & \exists\, ref_P, ref_Q : Refusal_1 \bullet (\, ref \cap (X \cup Y) = (ref_P \cap X) \cup (ref_Q \cap Y) \\ & \qquad \land \\ & \qquad (tr \upharpoonright X, ref_P) \in \mathcal{F}\,[\![P]\!] \land (tr \upharpoonright Y, ref_Q) \in \mathcal{F}\,[\![Q]\!]\,)\end{array}$$

$$\begin{array}{ll}\Leftrightarrow & (tr, ref) \in Trace \times Refusal_1 \ \land \ \mathrm{ran}\, tr \subseteq (X \cup Y) \qquad \text{[predicate logic and F2]} \\ & \land \\ & \exists\, ref_P, ref_Q : Refusal \bullet (\, ref \cap (X \cup Y) = (ref_P \cap X) \cup (ref_Q \cap Y) \\ & \qquad \land \\ & \qquad (tr \upharpoonright X, ref_P) \in \mathcal{F}\,[\![P]\!] \land (tr \upharpoonright Y, ref_Q) \in \mathcal{F}\,[\![Q]\!]\,)\end{array}$$

$$\Leftrightarrow \ (tr, ref) \in Trace \times Refusal_1 \ \land \ (tr, ref) \in \mathcal{F}\,[\![P \ {}_X\|_Y \ Q]\!] \qquad [\mathrm{def}^n \text{ of } \mathcal{F}]$$

$$\Leftrightarrow \ (tr, ref) \in (\, \mathcal{F}\,[\![P \ {}_X\|_Y \ Q]\!] \ \cap \ Trace \times Refusal_1\,) \qquad \text{[set theory]}$$

The third step warrants a closer look. The implication is obvious in one direction: since the set $Refusal_1$ is a subset of the set $Refusal$ it follows immediately that

$$(\, \exists\, ref_P, ref_Q : Refusal_1 \bullet \ldots) \ \Rightarrow \ (\, \exists\, ref_P, ref_Q : Refusal \bullet \ldots)$$

The implication in the other direction is less obvious. It holds as a result of the second healthiness condition in the failures model: if a set of events can be refused after a given trace then any (singleton) subset of events could also be refused after the given trace. □

As a corollary to the above theorem, we observe that all algebraic properties within the stable failures concerning only basic processes and operators are preserved within the singleton failures semantic model. This result is sufficiently important that we promote it to the status of a theorem.

Theorem 2 Any algebraic property in the failures model that involves only basic process and basic operators is preserved within the singleton failures semantic model.

$$\begin{array}{rcll} Stop \,\Box\, P & \equiv_{\mathcal{S}} & P & [\,\Box - \mathrm{unit}\,] \\ P \,\Box\, P & \equiv_{\mathcal{S}} & P & [\,\Box - \mathrm{idem}\,] \\ P \sqcap P & \equiv_{\mathcal{S}} & P & [\,\sqcap - \mathrm{idem}\,] \\ P \,\Box\, Q & \equiv_{\mathcal{S}} & Q \,\Box\, P & [\,\Box - \mathrm{sym}\,] \\ P \sqcap Q & \equiv_{\mathcal{S}} & Q \,\Box\, P & [\,\sqcap - \mathrm{sym}\,] \\ P \,\Box\, (Q \,\Box\, R) & \equiv_{\mathcal{S}} & (P \,\Box\, Q) \,\Box\, R & [\,\Box - \mathrm{assoc}\,] \\ P \sqcap (Q \sqcap R) & \equiv_{\mathcal{S}} & (P \sqcap Q) \sqcap R & [\,\sqcap - \mathrm{assoc}\,] \\ P \,\Box\, (Q \sqcap R) & \equiv_{\mathcal{S}} & (P \,\Box\, Q) \sqcap (P \,\Box\, R) & [\,\Box - \mathrm{dist}\,] \end{array}$$

$$
\begin{array}{rcll}
P \sqcap (Q \,\Box\, R) & \equiv_{\mathcal{S}} & (P \sqcap Q) \,\Box\, (P \sqcap R) & [\, \sqcap - \mathrm{dist}\,] \\
P \ {}_X\|_Y \ Q & \equiv_{\mathcal{S}} & Q \ {}_X\|_Y \ P & [\ {}_X\|_Y \ - \mathrm{sym}\,] \\
(P \ {}_X\|_Y \ Q) \ {}_{X\cup Y}\|_Z \ R & \equiv_{\mathcal{S}} & P \ {}_X\|_{Y\cup Z} \ (Q \ {}_Y\|_Z \ R) & [\ {}_X\|_Y \ - \mathrm{assoc}\,] \\
P \ {}_X\|_Y \ (Q \sqcap R) & \equiv_{\mathcal{S}} & (P \ {}_X\|_Y \ Q) \ \sqcap \ (P \ {}_X\|_Y \ R) & [\ {}_X\|_Y \ - \mathrm{dist}\,]
\end{array}
$$

Proof For each such algebraic property, this result follows directly from Theorem 1 above. The proof that the external choice operator distributes over the internal choice operator is given below. The proofs for the other laws follow similar lines. Given basic processes P, Q and R, we reason as follows:

$$\mathcal{S}[\![\, P \,\Box\, (\, Q \sqcap R \,) \,]\!]$$

$$= \mathcal{F}[\![\, P \,\Box\, (\, Q \sqcap R \,) \,]\!] \ \cap \ (\mathit{Trace} \times \mathit{Refusal}_1) \qquad \text{[Theorem 1]}$$

$$= \mathcal{F}[\![\, (\, P \,\Box\, Q \,) \ \sqcap \ (\, P \,\Box\, R \,) \,]\!] \qquad \text{[distributivity of } \Box \text{ over } \sqcap \text{ within } \mathcal{F}]$$
$$\cap$$
$$\mathit{Trace} \times \mathit{Refusal}_1$$

$$= \mathcal{S}[\![\, (\, P \,\Box\, Q \,) \sqcap (\, P \,\Box\, R \,) \,]\!] \qquad \text{[Theorem 1]}$$

□

3.2 Refinement and equivalence of basic processes

In this section we define formally the singleton failures refinement ordering and its induced equivalence relation for basic processes. In Section 3.3, having defined the singleton failures semantic function on the hiding operator, we will discuss its effect on refinement.

Refinement within the singleton failures model, as with both the traces and the stable failures model, is reverse containment: one basic process is refined by another process if every singleton failure of the second process is also a singleton failure of the first. Furthermore, two processes are equivalent if each refines the other. Given basic processes P and Q,

$$P \ \sqsubseteq_{\mathcal{S}} \ Q \ \Leftrightarrow \ \mathcal{S}[\![Q]\!] \ \subseteq \ \mathcal{S}[\![P]\!]$$

$$P \ \equiv_{\mathcal{S}} \ Q \ \Leftrightarrow \ P \sqsubseteq_{\mathcal{S}} Q \ \wedge \ Q \sqsubseteq_{\mathcal{S}} P$$

A useful means of establishing refinement within the traces and stable failures model is by considering the internal choice of the two processes. We prove that this result is preserved for basic processes within the singleton failures model.

Lemma 1 One basic process is refined by another basic process within the singleton failures model precisely when the internal choice between the two processes is equivalent to the first process. Given basic processes P and Q, $P \sqsubseteq_{\mathcal{S}} Q \Leftrightarrow (P \sqcap Q) \equiv_{\mathcal{S}} P$.

Proof

$$P \sqsubseteq_{\mathcal{S}} Q$$

$$\Leftrightarrow \mathcal{S}[\![Q]\!] \subseteq \mathcal{S}[\![P]\!] \qquad [\text{def}^{n} \text{ of } \sqsubseteq_{\mathcal{S}}]$$

$$\Leftrightarrow \mathcal{S}[\![P]\!] \cup \mathcal{S}[\![Q]\!] = \mathcal{S}[\![P]\!] \qquad [\text{set theory}]$$

$$\Leftrightarrow \mathcal{S}[\![P \sqcap Q]\!] = \mathcal{S}[\![P]\!] \qquad [\text{def}^{n} \text{ of } \mathcal{S}]$$

$$\Leftrightarrow P \sqcap Q \equiv_{\mathcal{S}} P \qquad [\text{def}^{n} \text{ of } \equiv_{\mathcal{S}}]$$

□

We use Lemma 1 to prove that, for basic processes, singleton failures refinement is preserved by parallel composition. This is an important result as it enables us to build large systems from smaller components whilst ensuring that the properties of each component are preserved.

Theorem 3 Singleton failures refinement is preserved by parallel composition. For any basic processes P, Q, and R, and any sets of events X and Y,

$$P \sqsubseteq_{\mathcal{S}} Q \Rightarrow (P\ {}_X\|_Y\ R) \sqsubseteq_{\mathcal{S}} (Q\ {}_X\|_Y\ R).$$

Proof

$$P \sqsubseteq_{\mathcal{S}} Q$$

$$\Leftrightarrow P \sqcap Q \equiv_{\mathcal{S}} P \qquad [\text{Lemma 1}]$$

$$\Rightarrow (P \sqcap Q)\ {}_X\|_Y\ R \equiv_{\mathcal{S}} P\ {}_X\|_Y\ R \qquad [\text{def}^{n} \text{ of } \mathcal{S}]$$

$$\Leftrightarrow (P\ {}_X\|_Y\ R) \sqcap (Q\ {}_X\|_Y\ R) \equiv_{\mathcal{S}} P\ {}_X\|_Y\ R \qquad [\text{Theorem 2}]$$

$$\Leftrightarrow (P\ {}_X\|_Y\ R) \sqsubseteq_{\mathcal{S}} (Q\ {}_X\|_Y\ R) \qquad [\text{Lemma 1}]$$

□

3.3 Hiding

In Theorem 1 we proved that for internal and external choice and for the prefix and parallel operators, the semantic function $\mathcal{S}$ is a projection of the function $\mathcal{F}$. We shall presently prove that this result does not extend to the hiding operator.

In the stable failures semantics, a failure of P makes a contribution to the semantic set of $P \setminus A$ only if the refusal set contains every event from A: after a given trace, a set *ref* can be refused by $P \setminus A$ only if the union of A and *ref* can be refused by P after the same trace. In the singleton failures model, however, we do not record the availability of combinations of events; we cannot determine whether or not the whole of $A \cup ref$ can be refused. The strongest check that we can make is that each event from the union of A and *ref* can be individually refused after the given trace. And so our definition of $\mathcal{S}$ applied to the hiding operator is as follows:

$$\begin{aligned}\mathcal{S}\,[\![\, P \setminus X \,]\!] \quad = \quad & \{\, tr : Trace;\ ref : Refusal_1 \mid \\ & \qquad (tr, ref) \in \mathcal{S}\,[\![P]\!] \ \wedge\ \forall\, e : X \bullet\ (tr, \{e\}) \in \mathcal{S}\,[\![P]\!] \bullet \\ & \qquad\quad (tr \setminus X, ref)\,\}\end{aligned}$$

Since the semantic function $\mathcal{S}$ applied to the hiding operator is not a projection of the stable failures semantic function we cannot apply Theorem 2 and inherit all the laws of the stable failures semantics. Indeed, as we will show, this weakened definition of the semantic function leads to some complications and undesirable results.

Observation 1 The hiding operator is not distributive with respect to the internal choice operator within the singleton failures semantics. Given processes P and Q and set of events A,

$$(\, P \sqcap Q\,) \setminus A \quad \not\equiv_{\mathcal{S}} \quad (\, P \setminus A\,) \,\sqcap\, (\, Q \setminus A\,)$$

Proof We prove this result with a simple example. Let processes $Q1$ and $Q2$ be as defined below.

$$\begin{aligned} Q1 \ = \ & a \rightarrow b \rightarrow Stop \\ & \sqcap \\ & (\, a \rightarrow b \rightarrow Stop\,) \ \Box\ b \rightarrow Stop \end{aligned} \qquad\qquad Q2 \ = \ b \rightarrow Stop$$

Given set of events $Event = \{a, b\}$, we easily establish the singleton failures of processes $Q1$ and $Q2$.

$$\begin{aligned}\mathcal{S}\,[\![\, Q1 \,]\!] \quad = \quad & \{\,(\langle\rangle, \emptyset), (\langle\rangle, \{b\})\,\} \ \cup\ \{\,(\langle a\rangle, \emptyset), (\langle a\rangle, \{a\})\,\} \\ & \cup \\ & \{\,(\langle a, b\rangle, \emptyset), (\langle a, b\rangle, \{a\}), (\langle a, b\rangle, \{b\})\,\}\end{aligned}$$

$$\begin{aligned}\mathcal{S}\,[\![\, Q2 \,]\!] \quad = \quad & \{\,(\langle\rangle, \emptyset), (\langle\rangle, \{a\})\,\} \\ & \cup \\ & \{\,(\langle b\rangle, \emptyset), (\langle b\rangle, \{a\}), (\langle b\rangle, \{b\})\,\}\end{aligned}$$

Hence we deduce the singleton failures of processes $(Q1 \setminus A)$ and $(Q2 \setminus A)$ where $A = \{a\}$.

$$\begin{array}{rcl} \mathcal{S}[\![\, Q1 \setminus A \,]\!] & = & \{\, (\langle\rangle, \emptyset), (\langle\rangle, \{a\}) \,\} \\ & & \cup \\ & & \{\, (\langle b \rangle, \emptyset), (\langle b \rangle, \{a\}), (\langle b \rangle, \{b\}) \,\} \end{array}$$

$$\begin{array}{rcl} \mathcal{S}[\![\, Q2 \setminus A \,]\!] & = & \{\, (\langle\rangle, \emptyset), (\langle\rangle, \{a\}) \,\} \\ & & \cup \\ & & \{\, (\langle b \rangle, \emptyset), (\langle b \rangle, \{a\}), (\langle b \rangle, \{b\}) \,\} \end{array}$$

We can now calculate the singleton failures of $(Q1 \setminus A) \sqcap (Q2 \setminus A)$ and $(Q1 \sqcap Q2) \setminus A$.

$$\begin{array}{rcl} \mathcal{S}[\![\, (Q1 \setminus A) \sqcap (Q2 \setminus A) \,]\!] & = & \{\, (\langle\rangle, \emptyset), (\langle\rangle, \{a\}) \,\} \\ & & \cup \\ & & \{\, (\langle b \rangle, \emptyset), (\langle b \rangle, \{a\}), (\langle b \rangle, \{b\}) \,\} \\ & & \\ \mathcal{S}[\![\, (Q1 \sqcap Q2) \setminus A \,]\!] & = & \{(\langle\rangle, \emptyset), (\langle\rangle, \{a\}), (\langle\rangle, \{b\}) \,\} \\ & & \cup \\ & & \{\, (\langle b \rangle, \emptyset), (\langle b \rangle, \{a\}), (\langle b \rangle, \{b\}) \,\} \end{array}$$

We see that

$$\mathcal{S}[\![\, (Q1 \sqcap Q2) \setminus A \,]\!] \; - \; \mathcal{S}[\![\, (Q1 \setminus A) \sqcap (Q2 \setminus A) \,]\!] \;\; = \;\; \{\, (\langle\rangle, \{b\}) \,\}$$

and conclude that the singleton failures semantics is not finitely distributive. □

Although this result is disappointing, all is not lost. Another key property *is* respected by the singleton failures semantics: *provided the hiding operator occurs only at the outermost level*, refinement is preserved by encapsulation.

Theorem 4 If P and Q are any processes and A is any set of events, then

$$P \sqsubseteq_{\mathcal{S}} Q \;\; \Rightarrow \;\; (\, P \setminus A \,) \sqsubseteq_{\mathcal{S}} (\, Q \setminus A \,).$$

Proof Suppose that process P is refined by process Q within the singleton failures semantic model. Then, given sequence of events tr and set of events ref containing at most one element, we reason as follows:

$$\begin{array}{ll} (tr, ref) \in \mathcal{S}[\![Q \setminus A]\!] & \\ & \\ \quad \Leftrightarrow \; \exists\, tr' : \operatorname{seq} Event \mid tr' \setminus A = tr \bullet & [\mathrm{def}^{\,n} \text{ of } \mathcal{S}] \\ \qquad\qquad (tr', ref) \in \mathcal{S}[\![Q]\!] \;\wedge\; \forall\, a : A \bullet (tr', \{a\}) \in \mathcal{S}[\![Q]\!] & \\ & \\ \quad \Rightarrow \; \exists\, tr' : \operatorname{seq} Event \mid tr' \setminus A = tr \bullet & [\mathcal{S}[\![Q]\!] \subseteq \mathcal{S}[\![P]\!]] \\ \qquad\qquad (tr', ref) \in \mathcal{S}[\![P]\!] \;\wedge\; \forall\, a : A \bullet (tr', \{a\}) \in \mathcal{S}[\![P]\!] & \\ & \\ \quad \Leftrightarrow \; (tr, ref) \in \mathcal{S}[\![P \setminus A]\!] & [\mathrm{def}^{\,n} \text{ of } \mathcal{S}] \end{array}$$

□

Recall the following laws concerning the hiding operator within both the traces and the stable failures models: hiding the empty set has no effect; hiding one set followed by another is equivalent to hiding the union of the two; and the hiding operator is symmetric. Only the first of these laws is fully preserved within the singleton failures semantics. The second law is partially preserved in that the process obtained by hiding one set of events and then another is refined by the process obtained by hiding the union of the two sets. The third law, however, is simply false within the singleton failures semantics.

$$P \setminus \emptyset \quad \equiv_{\mathcal{S}} \quad P \qquad \text{[null hiding]}$$

$$(P \setminus Y) \setminus X \quad \sqsubseteq_{\mathcal{S}} \quad P \setminus (X \cup Y) \qquad \text{[hide combine]}$$

Theorem 5 Hiding the empty set has no effect within the singleton failures semantic model.

Proof Given process P, sequence of events *tr* and set of events *ref* of cardinality at most one we reason as follows:

$(tr, ref) \in \mathcal{S}\,[\![P \setminus \emptyset]\!]$

$\Leftrightarrow\ \exists\, tr' : Trace \mid tr' \setminus \emptyset = tr \bullet$ $(tr', ref) \in \mathcal{S}\,[\![P]\!] \ \wedge\ \forall\, a : \emptyset \bullet (tr', \{a\}) \in \mathcal{S}\,[\![P]\!]$ [def^n of $\mathcal{S}$]

$\Leftrightarrow\ \exists\, tr' : Trace \mid tr' \setminus \emptyset = tr \bullet$ $(tr', ref) \in \mathcal{S}\,[\![P]\!]$ [predicate logic and property of $\forall$]

$\Leftrightarrow\ \exists\, tr' : Trace \mid tr' = tr \bullet (tr', ref) \in \mathcal{S}\,[\![P]\!]$ [property of $\setminus$]

$\Leftrightarrow\ (tr, ref) \in \mathcal{S}\,[\![P]\!]$ [one point rule]

$\square$

Theorem 6 The process obtained by hiding one set of events and then another is refined by the process obtained by hiding the union of the two sets.

$$(\,P \setminus Y\,) \setminus X \quad \sqsubseteq_{\mathcal{S}} \quad P \setminus (\,X \cup Y\,)$$

Proof Given process P, sequence of events *tr* set of events *ref* of cardinality at most one, and sets of events X and Y we reason as follows:

$(tr, ref) \in \mathcal{S}\,[\![(P \setminus Y) \setminus X]\!]$

$\Leftrightarrow\ \exists\, tr' : Trace \mid tr' \setminus X = tr \bullet$ $(tr', ref) \in \mathcal{S}\,[\![P \setminus Y]\!] \ \wedge\ \forall\, x : X \bullet (tr', \{x\}) \in \mathcal{S}\,[\![P \setminus Y]\!]$ [def^n of $\mathcal{S}$]

$$
\begin{array}{ll}
\Leftrightarrow \; \exists\, tr' : Trace \mid tr' \setminus X = tr \bullet & [\text{def}^{\,n} \text{ of } \mathcal{S}] \\
\qquad \exists\, tr'' : Trace \mid tr'' \setminus Y = tr' \bullet & \\
\qquad\qquad (tr'', ref) \in \mathcal{S}\,[\![P]\!] \;\wedge\; \forall\, y : Y \bullet (tr'', \{y\}) \in \mathcal{S}\,[\![P]\!] & \\
\qquad \wedge & \\
\qquad \forall\, x : X \;\bullet\; \exists\, tr'' : Trace \mid tr'' \setminus Y = tr' \bullet & \\
\qquad\qquad (tr'', \{x\}) \in \mathcal{S}\,[\![P]\!] \;\wedge\; \forall\, y : Y \bullet (tr'', \{y\}) \in \mathcal{S}\,[\![P]\!] &
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow \; \exists\, tr' : Trace \mid tr' \setminus X = tr \bullet & [\text{property of } \forall] \\
\qquad \exists\, tr'' : Trace \mid tr'' \setminus Y = tr' \bullet & \\
\qquad\qquad (tr'', ref) \in \mathcal{S}\,[\![P]\!] \;\wedge\; \forall\, y : Y \bullet (tr'', \{y\}) \in \mathcal{S}\,[\![P]\!] & \\
\qquad \wedge & \\
\qquad \exists\, tr'' : Trace \mid tr'' \setminus Y = tr' \bullet & \\
\qquad\qquad \forall\, x : X \bullet (tr'', \{x\}) \in \mathcal{S}\,[\![P]\!] & \\
\qquad\qquad \wedge & \\
\qquad\qquad \forall\, y : Y \bullet (tr'', \{y\}) \in \mathcal{S}\,[\![P]\!] &
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow \; \exists\, tr' : Trace \mid tr' \setminus X = tr \;\bullet\; \exists\, tr'' : Trace \mid tr'' \setminus Y = tr' \bullet & [\text{property of } \exists] \\
\qquad (tr'', ref) \in \mathcal{S}\,[\![P]\!] & \\
\qquad \wedge & \\
\qquad \forall\, y : Y \bullet (tr'', \{y\}) \in \mathcal{S}\,[\![P]\!] \;\wedge\; \forall\, x : X \bullet (tr'', \{x\}) \in \mathcal{S}\,[\![P]\!] &
\end{array}
$$

$$
\begin{array}{ll}
\Leftrightarrow \; \exists\, tr'' : Trace \mid tr'' \setminus (X \cup Y) = tr \bullet & [\text{property of } \setminus \text{ and } \forall] \\
\qquad (tr'', ref) \in \mathcal{S}\,[\![P]\!] \;\wedge\; \forall\, a : (X \cup Y) \bullet (tr'', \{a\}) \in \mathcal{S}\,[\![P]\!] &
\end{array}
$$

$$
\Leftrightarrow \; (tr, ref) \in \mathcal{S}\,[\![P \setminus (X \cup Y)]\!] \qquad [\text{def}^{\,n} \text{ of } \mathcal{S}]
$$

$\square$

Observation 2 Hiding one process and then another is not necessarily a refinement of hiding the union of the two.

$$
P \setminus (X \cup Y) \quad \not\sqsubseteq_{\mathcal{S}} \quad (P \setminus Y) \setminus X
$$

Proof We prove this with a simple example. Let P be the process defined below:

$$
\begin{array}{rcl}
P & = & y1 \rightarrow (\,(\, x \rightarrow e \rightarrow Stop\,) \\
 & & \qquad \sqcap \\
 & & \qquad (\, x \rightarrow e \rightarrow Stop\,) \,\square\, (\, e \rightarrow Stop\,)\,) \\
 & & \square \\
 & & y2 \rightarrow e \rightarrow Stop
\end{array}
$$

Then, defining X and Y respectively to be the sets $\{x\}$ and $\{y1, y2\}$ and *Event* to be the set $\{x, y1, y2, e\}$, we calculate the semantic sets of P, $P \setminus (X \cup Y)$, $P \setminus Y$, and $(P \setminus Y) \setminus X$.

$$
\begin{array}{rcl}
\mathcal{S}\,[\![P]\!] & = & \{\,(\langle\rangle, \emptyset),\ (\langle\rangle, \{x\}),\ (\langle\rangle, \{e\}), \\
& & \quad (\langle y1\rangle, \emptyset),\ (\langle y1\rangle, \{y1\}),\ (\langle y1\rangle, \{y2\}),\ (\langle y1\rangle, \{e\}), \\
& & \quad (\langle y2\rangle, \emptyset),\ (\langle y2\rangle, \{y1\}),\ (\langle y2\rangle, \{y2\}),\ (\langle y2\rangle, \{x\}), \\
& & \quad (\langle y1, x\rangle, \emptyset),\ (\langle y1, x\rangle, \{y1\}),\ (\langle y1, x\rangle, \{y2\}),\ (\langle y1, x\rangle, \{x\})\,\} \\
& & \cup \\
& & \{\, tr : \mathit{Trace};\ \mathit{ref} : \mathit{Refusal}_1 \mid \\
& & \qquad tr \in \{\langle y1, x, e\rangle, \langle y1, e\rangle, \langle y2, e\rangle\} \ \wedge\ \mathit{ref} \subset \{y1, y2, x, e\}\,\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S}\,[\![P \setminus (X \cup Y)]\!] & = & \{\,(\langle\rangle, \emptyset),\ (\langle\rangle, \{y1\}),\ (\langle\rangle, \{y2\}),\ (\langle\rangle, \{x\})\,\} \\
& & \cup \\
& & \{\, \mathit{ref} : \mathit{Refusal}_1 \mid \mathit{ref} \subset \{y1, y2, x, e\} \bullet (\langle e\rangle, \mathit{ref})\,\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S}\,[\![P \setminus Y]\!] & = & \{\,(\langle\rangle, \emptyset),\ (\langle\rangle, \{y1\}),\ (\langle\rangle, \{y2\}),\ (\langle\rangle, \{x\}),\ (\langle\rangle, \{e\}) \\
& & \quad (\langle x\rangle, \emptyset),\ (\langle x\rangle, \{y1\}),\ (\langle x\rangle, \{y2\}),\ (\langle x\rangle, \{x\})\,\} \\
& & \cup \\
& & \{\, tr : \mathit{Trace};\ \mathit{ref} : \mathit{Refusal}_1 \mid \\
& & \qquad tr \in \{\langle x, e\rangle, \langle e\rangle\} \ \wedge\ \mathit{ref} \subset \{y1, y2, x, e\}\,\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S}\,[\![(P \setminus Y) \setminus X)]\!] & = & \{\,(\langle\rangle, \emptyset),\ (\langle\rangle, \{y1\}),\ (\langle\rangle, \{y2\}),\ (\langle\rangle, \{x\}),\ (\langle\rangle, \{e\})\,\} \\
& & \cup \\
& & \{\, \mathit{ref} : \mathit{Refusal}_1 \mid \mathit{ref} \subset \{y1, y2, x, e\} \bullet (\langle e\rangle, \mathit{ref})\,\}
\end{array}
$$

We see that the singleton failure $(\langle\rangle, \{e\})$ lies in $\mathcal{S}\,[\![(P \setminus Y) \setminus X]\!]$ but not in $\mathcal{S}\,[\![P \setminus (X \cup Y)]\!]$ and hence that $\mathcal{S}\,[\![(P \setminus Y) \setminus X]\!]$ is not a subset of $\mathcal{S}\,[\![P \setminus (X \cup Y)]\!]$. We conclude that $\mathcal{S}\,[\![P \setminus (X \cup Y)]\!]$ is not refined within the singleton failures semantics by $\mathcal{S}\,[\![(P \setminus Y) \setminus X]\!]$. $\Box$

Observation 3 Hiding is not a symmetric operation within the singleton failures semantic model.

$$(P \setminus Y) \setminus X \quad \not\equiv_{\mathcal{S}} \quad (P \setminus X) \setminus Y$$

Proof We prove this with a simple example. Taking process P and sets X and Y to be as defined in Observation 2 above, we can show easily that $(P \setminus X) \setminus Y$ is equivalent to $P \setminus (X \cup Y)$ within the singleton failures model. Hence, since we proved in Observation 2 that $\mathcal{S}\,[\![P \setminus (X \cup Y)]\!]$ is a strict subset of $\mathcal{S}\,[\![(P \setminus Y) \setminus X]\!]$, it follows immediately that $(P \setminus X) \setminus Y$ is not equivalent to $(P \setminus Y) \setminus X$. $\Box$

In Theorem 1 we proved that for basic processes the singleton failures semantic function is a projection of the stable failures semantic function. We now consider the relation between the singleton failures semantic function and the projection of the stable failures semantic function for non-basic processes. We prove that for any process the subset of stable failures in which the cardinality of the refusal set is at most one is a subset of the singleton failures of the process, but that the converse is not necessarily true.

Theorem 7 Given any basic process P and any set of events X,

$$\mathcal{F}\,[\![\, P \setminus X \,]\!] \cap (\mathit{Trace} \times \mathit{Refusal}_1) \quad \subseteq \quad \mathcal{S}\,[\![\, P \setminus X \,]\!]$$

Proof Given basic process P, set of events X, trace tr and refusal set ref_1 of cardinality at most one we reason as follows:

$$(tr, \mathit{ref}_1) \in \mathcal{F}\,[\![\, P \setminus X \,]\!]$$

$$\Leftrightarrow\ \exists\, tr' : \mathit{Trace} \mid tr' \setminus X = tr \bullet (tr', \mathit{ref}_1 \cup X) \in \mathcal{F}\,[\![\, P \,]\!] \qquad [\mathrm{def}^n \text{ of } \mathcal{F}]$$

$$\Rightarrow\ \exists\, tr' : \mathit{Trace} \mid tr' \setminus X = tr \bullet \begin{array}{l} (tr', \mathit{ref}_1) \in \mathcal{F}\,[\![\, P \,]\!] \\ \wedge \\ \forall\, x : X \bullet (tr', \{x\}) \in \mathcal{F}\,[\![\, P \,]\!] \end{array} \qquad [\mathrm{F2}]$$

$$\Leftrightarrow\ \exists\, tr' : \mathit{Trace} \mid tr' \setminus X = tr \bullet \begin{array}{l} (tr', \mathit{ref}_1) \in \mathcal{S}\,[\![\, P \,]\!] \\ \wedge \\ \forall\, x : X \bullet (tr', \{x\}) \in \mathcal{S}\,[\![\, P \,]\!] \end{array} \qquad [\text{Theorem 1}]$$

$$\Leftrightarrow\ (tr, \mathit{ref}_1) \in \mathcal{S}\,[\![\, P \setminus X \,]\!] \qquad [\mathrm{def}^n \text{ of } \mathcal{S}]$$

□

Corollary 1 Given any process P, $\mathcal{F}\,[\![\, P \,]\!] \cap (\mathit{Trace} \times \mathit{Refusal}_1) \subseteq \mathcal{S}\,[\![\, P \,]\!]$.

Proof This result follows directly from Theorems 1 and 7 by structural induction on P.
□

Observation 4 The presence of a trace-refusal pair in the singleton failures semantics of a given non-basic process does not ensure the presence of the same trace-refusal pair in the stable failures semantics of the process.

$$\mathcal{S}\,[\![\, P \setminus X \,]\!] \quad \not\subseteq \quad \mathcal{F}\,[\![\, P \setminus X \,]\!]$$

Proof We prove this with a simple example. Let process P be as defined below:

$$P \;=\; (\, x \to y \to \mathit{Stop} \,) \;\sqcap\; (\, y \to \mathit{Stop} \,)$$

and let X be the set $\{x\}$ and *Event* be the set $\{x, y\}$. We see easily that the singleton failures and stable failures of P are respectively as follows:

$$\begin{aligned}
\mathcal{S}\,[\![\, P \,]\!] \;=\;& \{\, (\langle\rangle, \emptyset), (\langle\rangle, \{x\}), (\langle\rangle, \{y\}), (\langle x\rangle, \emptyset), (\langle x\rangle, \{x\}), \\
& (\langle y\rangle, \emptyset), (\langle y\rangle, \{x\}), (\langle y\rangle, \{y\}), \\
& (\langle x, y\rangle, \emptyset), (\langle x, y\rangle, \{x\}), (\langle x, y\rangle, \{y\}) \,\} \\
\mathcal{F}\,[\![\, P \,]\!] \;=\;& \{\, (\langle\rangle, \emptyset), (\langle\rangle, \{x\}), (\langle\rangle, \{y\}), (\langle x\rangle, \emptyset), (\langle x\rangle, \{x\}), \\
& (\langle y\rangle, \emptyset), (\langle y\rangle, \{x\}), (\langle y\rangle, \{y\}), (\langle y\rangle, \{x, y\}), \\
& (\langle x, y\rangle, \emptyset), (\langle x, y\rangle, \{x\}), (\langle x, y\rangle, \{y\}), (\langle x, y\rangle, \{x, y\}) \,\}
\end{aligned}$$

Hence the singleton failures and stable failures of $P \setminus X$ are respectively as follows:

$$\begin{aligned}
\mathcal{S}\,[\![\, P \setminus X \,]\!] \;=\;& \{\, (\langle\rangle, \emptyset), (\langle\rangle, \{x\}), (\langle\rangle, \{y\}), \\
& (\langle y\rangle, \emptyset), (\langle y\rangle, \{x\}), (\langle y\rangle, \{y\}) \,\} \\
\mathcal{F}\,[\![\, P \setminus X \,]\!] \;=\;& \{\, (\langle\rangle, \emptyset), (\langle\rangle, \{x\}), \\
& (\langle y\rangle, \emptyset), (\langle y\rangle, \{x\}), (\langle y\rangle, \{y\}), (\langle y\rangle, \{x, y\}) \,\}
\end{aligned}$$

Observe that the trace-refusal pair $(\langle\rangle, \{y\})$ lies in $\mathcal{S}\,[\![P]\!]$ but not in $\mathcal{F}\,[\![P]\!]$ and hence that $\mathcal{S}\,[\![P]\!]$ is not a subset of $\mathcal{F}\,[\![P]\!]$. □

3.4 Healthiness conditions

Unsurprisingly, the healthiness conditions for the singleton failures semantic model correspond very closely to the healthiness conditions for the stable failures model. Given a set S of singleton failures, they are:

$$\forall\, tr, tr_1, tr_2 : \mathit{Trace};\; e : \mathit{Event} \;\bullet$$

$$(tr_1 \frown tr_2, \emptyset) \in S \;\Rightarrow\; (tr_1, \emptyset) \in S \quad \wedge \qquad \text{(S1)}$$

$$(tr, \{e\}) \in S \;\Rightarrow\; (tr, \emptyset) \in S \quad \wedge \qquad \text{(S2)}$$

$$(tr, \langle\rangle) \in S \;\Rightarrow\; (\,(tr, \{e\}) \in S \;\vee\; (tr \frown \langle e\rangle, \emptyset) \in S\,) \qquad \text{(S3)}$$

The first condition (S1) is equivalent to the second condition in the traces model (T2) and the first condition in the stable failures model (F1): it ensures trace prefix closure. The second condition (S2) corresponds to the second condition in the stable failures model (F2): it ensures that if a singleton set can be refused after a particular trace then any subset can also be refused. Finally (S3) is a reformulation within the context of singleton failures of the third healthiness condition in the stable failures model (F3): it ensures that trace and refusal information remains consistent. After any given trace, any event can be either performed or refused or both.

3.5 Automatic verification

Reasoning about the properties of a process modelling any realistic system requires the assistance of a refinement-checking tool. The model-checker FDR [For97] has been developed to check for refinement within the traces, stable failures and failures-divergences models. Now we prove that additional procedures do not need to be written in order to check singleton failures refinement: we define a check expressed in terms of stable failures refinement that is equivalent to verifying singleton failures refinement. This check will hold for all basic processes, those expressible in terms of the process *Stop* and the operators $\rightarrow$, $\sqcap$, $\Box$ and ${}_X\|_Y$.

The singleton failures semantics is concerned with the refusal of individual events rather than combinations of events. Given processes P and Q, P is refined by Q within the singleton failures semantic model if every trace of Q is also a trace of P and if every event that can be refused by Q after a given trace could also have been refused by P. It does not matter if, after a given trace, Q can refuse a combination of events that cannot be collectively refused by P.

We can use the stable failures refinement ordering to check singleton failures refinement: we can determine whether P is refined within the singleton failures semantics if we can construct a process P' that has the same traces as P and such that after a given trace can refuse *all possible combinations* of events that can be refused individually by P after the given trace. We prove in Lemma 2 below that such a process is refined by Q within the stable failures model precisely when P is refined by Q within the singleton failures model.

Lemma 2 Given any basic processes P and Q expressible in terms of the process *Stop* and the operators $\rightarrow$, $\sqcap$, $\Box$ and ${}_X\|_Y$, if there exists a process P' that has the same traces as P and that after any given trace can refuse all combinations of events that could be refused individually by P:

$$\begin{aligned}\mathcal{F}\,[\![P']\!] \quad = \quad & \{tr : Trace;\ ref : Refusal \mid \\ & \qquad tr \in \mathcal{T}\,[\![P]\!] \ \wedge\ \forall\, e : ref \bullet (tr, \{e\}) \in \mathcal{F}\,[\![P]\!]\}\end{aligned}$$

then process P is refined by Q within the singleton failures model precisely when P' is refined by Q within the stable failures model.

$$P \ \sqsubseteq_{\mathcal{S}}\ Q \quad \Leftrightarrow \quad P' \ \sqsubseteq_{\mathcal{F}}\ Q$$

Proof Given processes P, Q and P' as defined above, suppose first that P is refined by Q within the singleton failures model. Given the sequence of events $tr : Trace$ and refusal set $ref : Refusal$ we reason as follows:

$(tr, ref) \in \mathcal{F}\,[\![Q]\!]$

$\Rightarrow tr \in \mathcal{T}\,[\![Q]\!] \ \wedge\ \forall\, e : ref \bullet (tr, \{e\}) \in \mathcal{F}\,[\![Q]\!]$ [F2]

$$\Leftrightarrow tr \in \mathcal{T}\,[\![Q]\!] \;\wedge\; \forall\, e : ref \bullet (tr, \{e\}) \in \mathcal{S}\,[\![Q]\!] \qquad \text{[Theorem 1]}$$

$$\Rightarrow tr \in \mathcal{T}\,[\![P]\!] \;\wedge\; \forall\, e : ref \bullet (tr, \{e\}) \in \mathcal{S}\,[\![P]\!] \qquad [P \sqsubseteq_{\mathcal{S}} Q]$$

$$\Leftrightarrow tr \in \mathcal{T}\,[\![P]\!] \;\wedge\; \forall\, e : ref \bullet (tr, \{e\}) \in \mathcal{F}\,[\![P]\!] \qquad \text{[Theorem 1]}$$

$$\Leftrightarrow (tr, ref) \in \mathcal{F}\,[\![P']\!] \qquad [\text{def}^{\,n} \text{ of } \mathcal{F}\,[\![P']\!]]$$

We have shown that if P is refined by Q within $\mathcal{S}$ then P' will be refined by Q within $\mathcal{F}$. Next we prove the implication in the other direction. Suppose now that P' is refined by Q within the stable failures model. Given sequence of events $tr : Trace$ and refusal set $ref_1 : Refusal_1$ we reason as follows:

$$(tr, ref_1) \in \mathcal{S}\,[\![Q]\!]$$

$$\Leftrightarrow (tr, ref_1) \in \mathcal{F}\,[\![Q]\!] \qquad \text{[Theorem 1]}$$

$$\Rightarrow (tr, ref_1) \in \mathcal{F}\,[\![P']\!] \qquad [P' \sqsubseteq_{\mathcal{F}} Q]$$

$$\Leftrightarrow tr \in \mathcal{T}\,[\![P]\!] \;\wedge\; \forall\, e : ref_1 \bullet (tr, \{e\}) \in \mathcal{F}\,[\![P]\!] \qquad [\text{def}^{\,n} \text{ of } P']$$

$$\Leftrightarrow (tr, ref_1) \in \mathcal{S}\,[\![P]\!] \qquad [\#ref_1 \leq 1, \text{ Theorem 1}]$$

□

In Lemmas 3, 4 and Corollary 2 which follow we prove that the process posited in Lemma 2 can be obtained through repeated parallel combination of P with itself.[1]

Lemma 3 Given any process P, if the sequence of processes $\{\, n : \mathbb{N} \mid n > 0 \bullet P^n \,\}$ is defined recursively as follows:

$$P^1 \;=\; P, \qquad\qquad P^{n+1} \;=\; P \;{}_{Event}\|_{Event}\; P^n$$

then, for any $n > 0$,

$$\mathcal{F}\,[\![P^{n+1}]\!] \;=\; \{\, tr : Trace;\; ref_1, ref_n : Refusal \mid (tr, ref_1) \in \mathcal{F}\,[\![P]\!] \wedge (tr, ref_n) \in \mathcal{F}\,[\![P^n]\!] \bullet (tr, ref_1 \cup ref_n) \,\}.$$

Proof

$$\mathcal{F}\,[\![P^{n+1}]\!]$$

$$= \mathcal{F}\,[\![P \;{}_{Event}\|_{Event}\; P^n]\!] \qquad [\text{definition of } P^{n+1}]$$

[1]This method of construction was suggested by Michael Goldsmith.

$$= \quad [\text{definition of } \mathcal{F} \text{ and } (\operatorname{ran} tr \cup ref \cup ref_1 \cup ref_n) \subseteq Event]$$
$$\{ tr : Trace;\ ref_1, ref_n : Refusal \mid (tr, ref_1) \in \mathcal{F}[\![P]\!] \wedge (tr, ref_n) \in \mathcal{F}[\![P^n]\!] \bullet (tr, ref_1 \cup ref_n) \}$$

□

Lemma 4 Given any basic process P the following predicate will hold:

$$\forall n : \mathbb{N} \mid n > 0 \bullet$$
$$\forall tr : Trace;\ ref : Refusal \mid \#ref \leq n \bullet$$
$$(tr \in \mathcal{T}[\![P]\!] \wedge \forall a : ref \bullet (tr, \{a\}) \in \mathcal{F}[\![P]\!]) \Leftrightarrow (tr, ref) \in \mathcal{F}[\![P^n]\!]$$

where P^n is as defined above.

Proof This is trivially true when $ref = \emptyset$. For cases when $\#ref > 0$ we prove the result inductively. Given truth set TRUTH our inductive hypothesis is as follows:

$$\forall k : \mathbb{N}^+ \bullet$$
$$k \in \textsc{Truth} \Leftrightarrow$$
$$(\forall tr : Trace;\ ref : Refusal \mid \#ref \leq k \bullet$$
$$(tr \in \mathcal{T}[\![P]\!] \wedge \forall a : ref \bullet (tr, \{a\}) \in \mathcal{F}[\![P]\!]$$
$$\Leftrightarrow$$
$$(tr, ref) \in \mathcal{F}[\![P^k]\!]))$$

The base case, when $k = 1$ follows directly from Theorem 1. For the inductive step suppose that for some strictly positive integer k, the predicate $k \in \textsc{Truth}$ holds. Suppose also that ref is a set of events such that $0 < \#ref \leq k + 1$ and choose some event e_{k+1} from ref. Defining ref_k to be the set $ref \setminus \{e_{k+1}\}$, and given any trace $tr : Trace$ we reason as follows:

$$tr \in \mathcal{T}[\![P]\!] \ \wedge\ \forall e : ref \bullet (tr, \{e\}) \in \mathcal{F}[\![P]\!]$$

$$\Leftrightarrow \quad [ref = ref_n \cup \{a_{n+1}\}]$$
$$tr \in \mathcal{T}[\![P]\!] \ \wedge\ (\forall e : ref_k \bullet (tr, \{e\}) \in \mathcal{F}[\![P]\!]) \ \wedge\ (tr, \{e_{k+1}\}) \in \mathcal{F}[\![P]\!]$$

$$\Leftrightarrow (tr, ref_k) \in \mathcal{F}[\![P^k]\!] \wedge (tr, \{e_{k+1}\}) \in \mathcal{F}[\![P]\!] \quad [\text{inductive hypothesis}]$$

$$\Leftrightarrow (tr, ref_k \cup \{e_{k+1}\}) \in \mathcal{F}[\![P^{k+1}]\!] \quad [\Rightarrow\text{: Lemma 3, } \Leftarrow\text{: F2}]$$

$$\Leftrightarrow (tr, ref) \in \mathcal{F}[\![P^{k+1}]\!] \quad [ref = ref_k \cup \{e_{k+1}\}]$$

We see that $k \in \textsc{Truth} \Rightarrow (k + 1) \in \textsc{Truth}$ and hence, since $1 \in \textsc{Truth}$ it follows that all positive integers lie in the truth set. We have proved our desired result. □

Corollary 2 If P is a basic process with finite alphabet *Event* then

$$\exists\, n : \mathbb{N} \mid n > 0 \bullet \\ \mathcal{F}\,[\![\, P^n \,]\!] \;=\; \{\, tr : \mathit{Trace};\ \mathit{ref} : \mathit{Refusal} \mid \\ tr \in \mathcal{T}\,[\![P]\!] \wedge \forall\, e : \mathit{ref} \bullet (tr, \{e\}) \in \mathcal{F}\,[\![\, P \,]\!]\,\}$$

Proof By Lemma 4, choosing n to be $\#\mathit{Event}$. □

Theorem 8 If P and Q are basic processes with finite alphabets then

$$P \sqsubseteq_{\mathcal{S}} Q \;\Leftrightarrow\; \exists\, n : \mathbb{N} \mid n > 0 \bullet P^n \sqsubseteq_{\mathcal{F}} Q$$

where P^n is the sequence defined above.

Proof By Lemma 2 and Corollary 2. □

This shows us that, in those situations in which we might wish to establish *singleton* failures refinement of P, it is enough to consider the *stable* failures refinement of P^n. Although an upper bound for n, the number of times we must put *Spec* in parallel with itself, is the size of the set *Event*, the determining factor is the degree of internal choice used in defining the process.

Constructing parallel combinations can be expensive in terms of the time that it takes to compile state-machine representations. For practical applications, it is useful to observe that:

Corollary 3 For any pair of basic processes P and Q with finite alphabets

$$P \sqsubseteq_{\mathcal{S}} Q \;\Leftrightarrow\; \exists\, n : \mathbb{N} \bullet P^{\parallel n} \sqsubseteq_{\mathcal{F}} Q$$

where the sequence $\{\, n : \mathbb{N} \bullet P^{\parallel n} \,\}$ is defined recursively as follows:

$$\begin{aligned} P^{\parallel 0} &= P \\ P^{\parallel (n+1)} &= P^{\parallel n} {}_{\mathit{Event}}\|_{\mathit{Event}}\, P^{\parallel n} \end{aligned}$$

Proof From Theorem 8, as $P^{\parallel n} = P^{2^n}$ and $P^{n+1} \sqsubseteq_{\mathcal{F}} P^n$. □

This reduces the number of parallel combinations required from $O(n)$ to $O(\log n)$, where n is the degree of internal choice employed in the definition of process P.

Theorem 8 showed how *provided the processes in question are basic processes* the stable failures model—and hence the model checker FDR—might be used to verify singleton failures refinement. Next we prove that this result is not extensible to all processes, and in particular not to those expressed in terms of the hiding operator.

Observation 5 Given any processes P and Q it is not necessarily the case that P is refined by Q within the singleton failures model when arbitrarily many parallel combinations of P is refined by Q within the stable failures model.

$$(\,\exists\, n : \mathbb{N} \mid n > 0 \bullet P^n \sqsubseteq_{\mathcal{F}} Q\,) \;\;\not\Rightarrow\;\; P \sqsubseteq_{\mathcal{S}} Q$$

Proof We prove this with a simple example. In Observation 1 it was shown that given set $A = \{a\}$ and processes $Q1$ and $Q2$ as defined below:

$$\begin{aligned} Q1 \;=\;& a \rightarrow b \rightarrow \mathit{Stop} \\ & \sqcap \\ & (\, a \rightarrow b \rightarrow \mathit{Stop}\,) \;\Box\; b \rightarrow \mathit{Stop} \end{aligned} \qquad\qquad Q2 \;=\; b \rightarrow \mathit{Stop}$$

the process $P = (Q1 \setminus A) \sqcap (Q2 \setminus A)$ is not refined by the process $Q = (Q1 \sqcap Q2) \setminus A$ within the singleton failures model. However, we can show that the process obtained by putting P in parallel with itself is refined by Q within the stable failures semantic model.

$$\begin{array}{rcll} P & \not\sqsubseteq_{\mathcal{S}} & Q & [\,\text{Observation } 1\,] \\ (\, P \;{}_{\{a,b\}}\|_{\{a,b\}}\; P\,) & \sqsubseteq_{\mathcal{F}} & Q & [\,\text{FDR}\,] \end{array}$$

□

3.6 Comparison of models

We begin this section with an example that highlights both the similarities and the differences between the traces, stable failures and singleton failures models before formally identifying the relationships between the models.

Example 1 Information content of the traces, stable failures and singleton failures models.

Consider processes $P1$, $P2$ and $P3$ given by:

$$\begin{aligned} P1 \;\widehat{=}\;& a \rightarrow \mathit{STOP} \\ & \Box \\ & b \rightarrow \mathit{STOP} \end{aligned} \qquad \begin{aligned} P2 \;\widehat{=}\;& a \rightarrow \mathit{STOP} \\ & \sqcap \\ & b \rightarrow \mathit{STOP} \end{aligned} \qquad \begin{aligned} P3 \;\widehat{=}\;& a \rightarrow \mathit{STOP} \\ & \sqcap \\ & b \rightarrow \mathit{STOP} \\ & \sqcap \\ & \mathit{STOP} \end{aligned}$$

Our first observation is that these three processes are indistinguishable within the traces model: we do not record the availability of events.

$$\mathcal{T}\,[\![P1]\!] \;=\; \mathcal{T}\,[\![P2]\!] \;=\; \mathcal{T}\,[\![P3]\!] \;=\; \{\langle\rangle, \langle a\rangle, \langle b\rangle\}$$

Our next observation is that the stable failures model distinguishes between all three processes. Initially, process $P1$ can refuse neither a nor b, process $P2$ can refuse either a or b and process $P3$ can refuse both a and b. Given a set of events Event containing both a and b, the set of failures of processes $P1$, $P2$ and $P3$ are as described below.

$$\begin{aligned} \mathcal{F}\,[\![\,P1\,]\!] \;=\;& \{\mathit{ref} : \mathit{Refusal} \mid \{a, b\} \cap \mathit{ref} = \emptyset \bullet (\langle\rangle, \mathit{ref})\} \\ & \cup \\ & \{e : \{a, b\};\; \mathit{ref} : \mathit{Refusal} \bullet (\langle e\rangle, \mathit{ref})\} \end{aligned}$$

$$\begin{aligned}
\mathcal{F}[\![P2]\!] &= \{ref : Refusal \mid \{a, b\} \not\subseteq ref \bullet (\langle\rangle, ref)\} \\
& \cup \\
& \{e : \{a, b\};\ ref : Refusal \bullet (\langle e \rangle, ref)\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}[\![P3]\!] &= \{ref : Refusal \bullet (\langle\rangle, ref)\} \\
& \cup \\
& \{e : \{a, b\};\ ref : Refusal \bullet (\langle e \rangle, ref)\}
\end{aligned}$$

Within the singleton failures model we can distinguish $P1$ from both $P2$ and $P3$: events a and b can be refused by both $P2$ and $P3$ but cannot be refused by $P1$. However we cannot distinguish between processes $P2$ and $P3$: we record the fact that both events might initially be refused by both processes but not whether or not their combination might be refused.

$$\begin{aligned}
\mathcal{S}[\![P1]\!] &= \{ref : Refusal_1 \mid \{a, b\} \cap ref = \emptyset \bullet (\langle\rangle, ref)\} \\
& \cup \\
& \{e : \{a, b\};\ ref : Refusal_1 \bullet (\langle e \rangle, ref)\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}[\![P2]\!] = \mathcal{S}[\![P3]\!] &= \{ref : Refusal_1 \bullet (\langle\rangle, ref)\} \\
& \cup \\
& \{e : \{a, b\};\ ref : Refusal_1 \bullet (\langle e \rangle, ref)\}
\end{aligned}$$

We see that the information content of the singleton failures model lies in between the traces and stable failures models. Whilst the traces model identifies all three processes and the stable failures model distinguishes between all three processes, the singleton failures model identifies processes $P2$ and $P3$ but distinguishes these from process $P1$. We sum up this information in Figure 3.1 below.

	$P1 \equiv P2$	$P2 \equiv P3$	$P1 \equiv P3$
$\mathcal{T}$	✓	✓	✓
$\mathcal{S}$	×	✓	×
$\mathcal{F}$	×	×	×

Figure 3.1: Equivalence of processes $P1$, $P2$ and $P3$ within the various semantic models.

□

We now formally identify the relationship between the singleton failures model and both the traces model and the stable failures model. Although the proof that singleton failures refinement implies traces refinement is trivial the result is of sufficient importance to merit the status of a theorem.

Theorem 9 Singleton failures refinement implies traces refinement. Given any processes P and Q, the following predicate holds.

$$P \sqsubseteq_{\mathcal{S}} Q \quad \Rightarrow \quad P \sqsubseteq_{\mathcal{T}} Q$$

Proof Given sequence of events tr and processes P and Q such that $P \sqsubseteq_{\mathcal{S}} Q$ we reason as follows:

$$
\begin{aligned}
& tr \in \mathcal{T}[\![Q]\!] \\
& \quad \Leftrightarrow (tr, \emptyset) \in \mathcal{S}[\![Q]\!] && [\text{property of } \mathcal{S}] \\
& \quad \Leftrightarrow (tr, \emptyset) \in \mathcal{S}[\![P]\!] && [P \sqsubseteq_{\mathcal{S}} Q] \\
& \quad \Leftrightarrow tr \in \mathcal{T}[\![P]\!] && [\text{property of } \mathcal{S}]
\end{aligned}
$$

□

Observation 6 Stable failures refinement does not imply singleton failures refinement. Given processes P and Q,

$$P \sqsubseteq_{\mathcal{F}} Q \quad \not\Rightarrow \quad P \sqsubseteq_{\mathcal{S}} Q$$

Proof We prove this result with a simple example. Consider processes $P \setminus (X \cup Y)$ and $(P \setminus Y) \setminus X$ where process P and sets X and Y are as defined in Observation 2. We recognise immediately that these processes are equivalent within the context of the stable failures mode and hence conclude that the following predicate holds.

$$P \setminus (X \cup Y) \quad \sqsubseteq_{\mathcal{F}} \quad (P \setminus Y) \setminus X$$

However, we have already shown in Observation 2 that the corresponding result does not hold within the context of the singleton failures semantic model.

$$P \setminus (X \cup Y) \quad \not\sqsubseteq_{\mathcal{S}} \quad (P \setminus Y) \setminus X$$

We therefore conclude our desired result, that stable failures refinement does not imply singleton failures refinement. □

3.7 Interesting examples

We conclude this chapter with a couple of at first surprising examples. These illustrate results that hold within the context of the singleton failures semantic model but not within the context of the stable failures semantic model.

Example 2

$$
\begin{aligned}
& (\, a \rightarrow \mathit{Stop} \,) \sqcap (\, (\, b \rightarrow \mathit{Stop} \,) \;\Box\; (\, c \rightarrow \mathit{Stop} \,)\,) \\
& \quad \equiv_{\mathcal{S}} \\
& \qquad (\, a \rightarrow \mathit{Stop} \,) \sqcap (\, b \rightarrow \mathit{Stop} \,) \sqcap (\, c \rightarrow \mathit{Stop} \,)
\end{aligned}
$$

Proof Both processes share the same singleton failures. If *Event* is the set $\{a, b, c\}$, then both their singleton failures semantic sets are:

$$\begin{array}{l} \{\, (\langle\rangle, \emptyset) \,\} \;\cup\; \{\, e : \{a, b, c\} \bullet (\langle\rangle, \{\, e \,\}) \,\} \\ \cup \\ \{\, e : \{a, b, c\} \bullet (\langle e \rangle, \emptyset) \,\} \;\cup\; \{\, e, e' : \{a, b, c\} \bullet (\langle e \rangle, \{\, e' \,\}) \,\} \end{array}$$

They may each initially perform either a or b or c and they may each initially refuse either a or b or c. The fact that the first process cannot refuse the combination of a and b whereas the second process can is not recorded within the singleton failures semantics. $\square$

Example 3

$$(\, a \rightarrow \mathit{Stop} \,) \;\sqcap\; (\, b \rightarrow \mathit{Stop} \,) \quad \sqsubseteq_{\mathcal{S}} \quad \mathit{Stop}$$

Proof We see immediately that the singleton failures of $(a \rightarrow \mathit{Stop}) \sqcap (b \rightarrow \mathit{Stop})$ and *Stop* are as defined below:

$$\begin{array}{rcl} \mathcal{S}\,[\![\, (a \rightarrow \mathit{Stop}) \sqcap (b \rightarrow \mathit{Stop}) \,]\!] & = & \{\, (\langle\rangle, \emptyset), (\langle\rangle, \{\, a \,\}), (\langle\rangle, \{\, b \,\}), \\ & & \;\; (\langle a \rangle, \emptyset), (\langle a \rangle, \{\, a \,\}), (\langle a \rangle, \{\, b \,\}), \\ & & \;\; (\langle b \rangle, \emptyset), (\langle b \rangle, \{\, a \,\}), (\langle b \rangle, \{\, b \,\}) \,\} \\ \mathcal{S}\,[\![\, \mathit{Stop} \,]\!] & = & \{\, (\langle\rangle, \emptyset), (\langle\rangle, \{\, a \,\}), (\langle\rangle, \{\, b \,\}) \,\} \end{array}$$

Observing that the set of singleton failures of *Stop* is a subset of the set of singleton failures of $(a \rightarrow \mathit{Stop}) \sqcap (b \rightarrow \mathit{Stop})$ and hence conclude our desired result. $\square$

Chapter 4

Simple data types

Abstract data types can be used to model systems, combining a notion of local state with a collection of named operations. In addition a set of possible initial states are identified. In this chapter we will explore the relationship between data refinement and CSP refinement, exposing fallacious results in the literature. For clarity and simplicity whilst exploring these issues we will consider only *simple data types*: data types with no explicit notion of input and output. We will proceed in Chapter 5 to extend these results to *communicating data types*—data types with an explicit notion of input and output. We present our results within the context of two accepted semantic interpretations: the *blocking* and *non-blocking* models.

Although our presentation is different, these simple data types have the same information content as those presented in [WD96]. Woodcock and Davies gave their data types a non-blocking relational semantics and presented sound and jointly complete simulation rules to match this model. In [BDW99], a paper based on [Bol98], we gave Woodcock and Davies' data types a blocking relational semantics, derived a corresponding set of simulation rules, and proved their soundness and joint completeness. We present these sets of rules here both for completeness and to facilitate our understanding when it comes to reasoning about the more complicated communicating data types of Chapter 5.

We present translations from simple abstract data types to their corresponding processes within the context of the blocking and the non-blocking semantic model. The translation within the context of the blocking semantic model, reported in [BDW99], was originally presented in [Bol98] with the intention of formally proving a result that was being propagated through the literature [SD97, Fis00, DB01]: that data refinement, within this model, is equivalent to stable failures refinement. We proved that this result holds for data types *with a single operation* and, incorrectly, concluded that it holds in the general case.

In this chapter we correct the erroneous result of [BDW99], showing also that data refinement is not equivalent to stable failures refinement within the non-blocking semantic model. Having used our counter examples to explore the information content of the various semantic models we then go on to prove that, for simple data types, data refinement is equivalent to singleton failures refinement. We prove this result in the context of both semantic interpretations.

4.1 Syntactic definition

This chapter explores the relationship between data refinement and CSP refinement. To facilitate this exploration we take a most simplistic approach to our data types. Firstly we seemingly ignore any notion of input and output. (In fact, as we will prove in Chapter 5 the giving and receiving of input and output values can be incorporated into our initialisation and finalisation operations, through our fictional environment *Env*.) Secondly, having shown in Section 2.6 that schemas can be used to define the various components of a data type—its internal state space, initialisation, finalisation and named operations—we observe that these can in turn be interpreted in terms of their underlying relations. And so our simplistic definition of a data type will be in terms of those relations.

For the time being, a data type will simply have four components: a set of private internal states *state* of type *State* a total initialisation relation *init* from the environment *Env* to the internal state space; a function *ops* mapping the name of each operation onto the relation on the state space describing the effect of the operation; and a total finalisation relation *final* from the internal state space to the environment *Env*. A simple abstract data type can then be defined generically as follows:

$$
\begin{array}{|l}
\hline
ADT[State, Name, Env] \\
state : \mathbb{P}\, State \\
init : Env \leftrightarrow State \\
ops : Name \nrightarrow (State \leftrightarrow State) \\
final : State \leftrightarrow Env \\
\hline
\mathrm{dom}\, init \;=\; Env \;\wedge \\
\mathrm{dom}\, final \;=\; state \;\wedge \\
\mathrm{ran}\, init \;\subseteq\; state \;\wedge \\
ops \;\in\; Name \nrightarrow (state \leftrightarrow state) \\
\hline
\end{array}
$$

4.2 Relational semantics

We can characterise a data type in terms of the visible effects of different programs or sequences of operations. The set of possible effects of each such program can be modelled as a relation between environment states obtained from the relational composition of the sequence, preceded by initialisation and succeeded by finalisation.

The effect of any operation can be non-deterministic: under identical circumstances, different values might be returned. Furthermore, each operation might be only partially defined. Traditionally two different approaches have been adopted for solving the problem of what happens when a partially defined operation is called outside its precondition [DB01]. The first is the *blocking* or *behavioural* model in which operations cannot be performed outside their preconditions; they are simply blocked. The second approach is the *non-blocking* or *contract* model in which operations may be called outside their

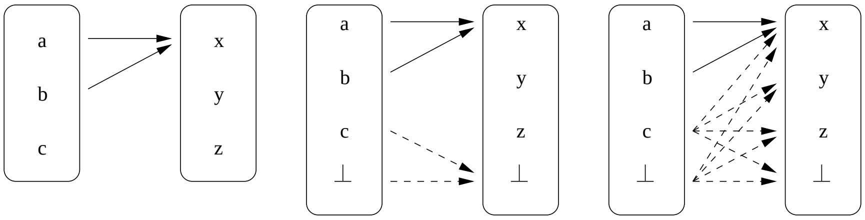

Figure 4.1: The relation $\{a \mapsto x, b \mapsto x\}$ and its blocking and non-blocking totalisations.

preconditions but no guarantees are made about the outcome of that operation or indeed any subsequent operations.

Whichever approach we adopt we need to be able to record the fact that an operation has been called outside its preconditions. We do this by first augmenting the state space with the element bottom ($\perp$)

$$State_\perp \;=\; State \cup \{\perp\} \quad \text{and} \quad Env_\perp \;=\; Env \cup \{\perp\}$$

and then *totalising* the relations appropriately. Given relation r of type $X \leftrightarrow Y$ we use $r^\top$ to denote the totalised version, the relation where $r^\top$ is of type $X_\perp \leftrightarrow Y_\perp$. The way in which we totalise the relation determines whether the program is blocked or whether it can continue:

- the totalisation that gives rise to the blocking semantics requires every element (including the bottom element) outside the domain to be mapped only to the bottom element;

$$r^\top \;=\; r \cup ((X_\perp \setminus \operatorname{dom} r) \times \{\perp\})$$

- the totalisation that gives rise to the non-blocking semantics requires every element (including the bottom element) outside the domain to be mapped to every element in the target state space including the bottom element.

$$r^\top \;=\; r \cup ((X_\perp \setminus \operatorname{dom} r) \times Y_\perp)$$

Figure 4.1 illustrates the effect of both totalisations on the relation $\{a \mapsto x, b \mapsto x\}$ of type $\{a, b, c\} \times \{x, y, z\}$.

Before formally defining our relational semantics we need to introduce the function *run* that composes a sequence of named operations on the same state space. More precisely, it takes a sequence of names of operations and then a function mapping the names of each operation onto the relation describing that operation (*ops*), and then returns the relation describing the sequential composition of the totalised relations corresponding to the named operations. Where necessary we will explicitly state our chosen interpretation—blocking or non-blocking—by tagging *run* accordingly.

$$
\begin{array}{l}
[State, Name] \\
\hline
run_{ADT} : \operatorname{seq} Name \nrightarrow (Name \nrightarrow (State \leftrightarrow State)) \nrightarrow \\
\qquad\qquad (State_\bot \leftrightarrow State_\bot) \\
\hline
\forall\, n : Name;\ ns : \operatorname{seq} Name;\ ops : Name \nrightarrow (State \leftrightarrow State) \bullet \\
\qquad run_{ADT}\ \langle\rangle\ ops = \operatorname{id} State_\bot\ \land \\
\qquad run_{ADT}\ (\langle n \rangle ^\frown ns)\ ops = (ops\ n)^\top \fatsemi run_{ADT}\ ns\ ops
\end{array}
$$

The set of effects of any given sequence of operations is the sequential composition of the totalised versions of the operations (obtained using the function *run*) bracketed by the totalised version of the initialisation and finalisation operations. Once again we may use tagging to indicate whether the semantic model defining the refinement ordering adopts a blocking or a non-blocking approach.

$$
\begin{array}{l}
[State, Name, Env] \\
\hline
\mathcal{R}_{ADT} : ADT[State, Name, Env]\ \rightarrow\ (\operatorname{seq} Name \nrightarrow Env_\bot \leftrightarrow Env_\bot) \\
\hline
\mathcal{R}_{ADT}\ =\ (\ \lambda\, ADT[State, Name, Env] \bullet \\
\qquad\qquad (\lambda\, prog : \operatorname{seq}(\operatorname{dom} ops) \bullet \\
\qquad\qquad\qquad init^\top \fatsemi run_{ADT}\, prog\ ops \fatsemi final^\top)\ )
\end{array}
$$

4.3 Refinement

We use the relational semantics presented in Section 4.2 to define a refinement ordering on our data types. Data refinement, like CSP refinement, is reverse containment. Data type A is refined by data type C if, for each sequence of named operations, the effect upon C is more deterministic than the effect upon A. For every such sequence *prog*, the relation $\mathcal{R}_{ADT}[C]\, prog$ must be a subset of the relation $\mathcal{R}_{ADT}[A]\, prog$. When this is the case, we write $A \sqsubseteq_{\mathcal{R}_{ADT}} C$. Formally,

$$
\begin{array}{l}
[AState, CState, Name, Env] \\
\hline
\_ \sqsubseteq_{\mathcal{R}_{ADT}} \_ : ADT[AState, Name, Env] \leftrightarrow ADT[CState, Name, Env] \\
\hline
\forall\, ADT[AState, Name, Env];\ ADT'[CState, Name, Env] \bullet \\
\qquad \theta ADT \sqsubseteq_{\mathcal{R}_{ADT}} \theta ADT' \\
\qquad\qquad \Leftrightarrow \\
\qquad\qquad\quad \forall\, prog : \operatorname{seq} Name \bullet \\
\qquad\qquad\qquad \mathcal{R}_{ADT}[\,\theta ADT'\,]\, prog\ \subseteq\ \mathcal{R}_{ADT}[\,\theta ADT\,]\, prog
\end{array}
$$

Note that in order to verify that one data type is refined by another the two data types must have the same *interface*, that is, the same combination of *Name* and *Env* that defines the signatures of the operations. They may however have different internal state spaces. Where necessary we will explicitly state our chosen interpretation—blocking or non-blocking—using appropriate tagging.

We say that two data types are equivalent if each refines the other.

$$
\begin{array}{l}
[S, T, Name, Env] \\
\_ \equiv_{\mathcal{R}_{ADT}} \_ : ADT[S, Name, Env] \leftrightarrow ADT[T, Name, Env] \\
\hline
\forall\, ADT[S, Name, Env];\ ADT'[T, Name, Env] \bullet \\
\quad \theta ADT \equiv_{\mathcal{R}_{ADT}} \theta ADT' \\
\qquad \Leftrightarrow \\
\qquad\quad (\theta ADT \sqsubseteq_{\mathcal{R}_{ADT}} \theta ADT') \wedge (\theta ADT' \sqsubseteq_{\mathcal{R}_{ADT}} \theta ADT)
\end{array}
$$

4.4 Simulation

Using the definition of refinement presented in Section 4.3 we could in principle establish that one data type is refined by another by considering every effect of every program of operations. However, since programs can be arbitrarily long this can be an arbitrarily large task. Rather we establish inductively that refinement holds using simple sets of *simulation rules* [HHS87]. In this section we prove that if we can show *operation by operation* that one data type can simulate the behaviour of another then refinement follows. The simulation rules establish that the effect of each operation on the more concrete data type is more deterministic than the effect of the same operation on the more abstract data type. If data type θADT simulates data type $\theta ADT'$ within the context of the blocking semantics then we write $\theta ADT \preceq_b \theta ADT'$. Similarly, if θADT simulates $\theta ADT'$ within the context of the non-blocking semantics then we write $\theta ADT \preceq_n \theta ADT'$.

Simulation rules can be expressed either in terms of the schemas describing the operations on the data types or in terms of the relations describing the operations on the data types. We present both versions of the rules for both semantic models.

Data types with the same state space

If the two data types have the same state space then establishing simulation is a simple task. In the non-blocking semantics, data type θADT simulates data type $\theta ADT'$ if the following rules hold:

$$init' \subseteq init$$

$$\forall\, n : \mathrm{dom}\, ops \bullet \mathrm{dom}\,(ops\ n) \subseteq \mathrm{dom}\,(ops'\ n)$$

$$\forall\, n : \mathrm{dom}\, ops \bullet \mathrm{dom}\,(ops\ n) \lhd ops'\ n \subseteq ops\ n$$

$$final' \subseteq final$$

where $\lhd$ denotes domain restriction of a relation: see [Spi92]. These rules allow the domain of an operation to be extended and the postcondition to be strengthened. This is noticeably weaker than the corresponding rule for the blocking semantics where postconditions can be strengthened but the domain must remain the same.

$$init' \subseteq init$$

$$\forall\, n : \mathrm{dom}\, ops \bullet \mathrm{dom}\,(ops\; n) = \mathrm{dom}\,(ops'\; n)$$

$$\forall\, n : \mathrm{dom}\, ops \bullet ops'\; n \subseteq ops\; n$$

$$final' \subseteq final$$

In either semantics, we can obtain a refinement by *removing* a set of pairs from a relation, as long as the domain of the resulting relation stays the same. In the non-blocking semantics, we can also *add* a set of pairs, provided that each pair starts from a point that is outside the current domain.

Given data types A and C both with state space *State*, with initialisations and finalisations *AInit* and *AFinal*, and *CInit* and *CFinal* respectively and with corresponding operations *AOp* and *COp*, the schema versions of the non-blocking rules are:

$$\forall\, Env;\; State' \bullet CInit \Rightarrow AInit$$

$$\forall\, State \bullet \mathrm{pre} AOp \Rightarrow \mathrm{pre} COp$$

$$\forall\, State;\; State' \bullet \mathrm{pre} AOp \wedge COp \Rightarrow AOp$$

$$\forall\, State;\; Env' \bullet CFinal \Rightarrow AFinal$$

Similarly, the schema versions of the blocking rules for data types with the same state space are as follows.

$$\forall\, Env;\; State' \bullet CInit \Rightarrow AInit$$

$$\forall\, State \bullet \mathrm{pre} AOp \Leftrightarrow \mathrm{pre} COp$$

$$\forall\, State;\; State' \bullet \mathrm{pre} AOp \wedge COp \Rightarrow AOp$$

$$\forall\, State;\; Env' \bullet CFinal \Rightarrow AFinal$$

Forward simulation rules

Unlike the refinement ordering which is independent of the internal state spaces of the data types involved, the definition of a simulation refers to the state spaces of the data types. Where the data types do not have the same state space we need to identify a *retrieve relation*; the retrieve relation explains how the state of one of the data types can be retrieved from the state of the other.

If some of the non-determinism of the more abstract of the two data types has been resolved and if none of the non-determinism has been postponed then we look to establish a forwards simulation. If, however, some of the non-determinism of the more abstract of the two data types has been postponed but none of the non-determinism has been resolved then we look to establish a backwards simulation. Alternatively, if some of the non-determinism of the more abstract of the two data types has been resolved and

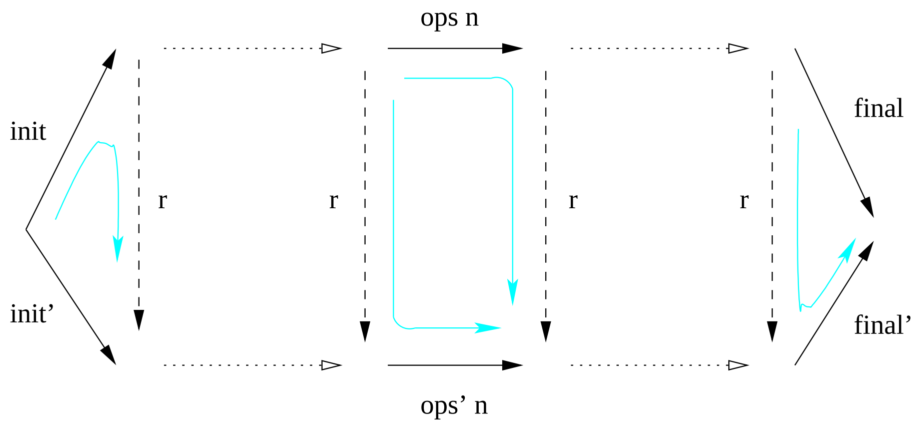

Figure 4.2: A forwards simulation

if some has been postponed then we need to identify an intermediary data type which is simulated by the first (either forwards or backwards) and simulates (either backwards or forwards) the second.

Note that in Z the retrieve relation for a forwards simulation typically maps the state space of the more abstract data type onto the state space of the more concrete data type and the retrieve relation for a backwards simulation typically maps the state space of the more concrete data type onto the state space of the more abstract data type. As they are relations this is merely a convention; indeed when we are considering the schema representation of such a relation, the direction is irrelevant. We will adhere to this convention.

In the non-blocking semantics, data type θADT forwards simulates data type $\theta ADT'$ if we can identify a retrieve relation r from *state* to *state'* such that the following rules hold:

$$init' \subseteq init \mathbin{⨾} r$$

$$\forall\, n : \mathrm{dom}\, ops \bullet r\llparenthesis\, \mathrm{dom}\,(ops\; n)\,\rrparenthesis \subseteq \mathrm{dom}\,(ops'\; n)$$

$$\forall\, n : \mathrm{dom}\, ops \bullet \mathrm{dom}\,(ops\; n) \lhd r \mathbin{⨾} ops'\; n \subseteq ops\; n \mathbin{⨾} r$$

$$r \mathbin{⨾} final' \subseteq final$$

where $\llparenthesis\,\rrparenthesis$ denotes the relational image of a set: see [Spi92]. If this is the case we write $\theta ADT \preceq_{f-n} \theta ADT'$. The first of these conditions states that for every initial concrete state there is a matching initial abstract state. The second insists that the concrete operation is defined whenever the abstract operation would be. The third insists that *whenever the abstract operation is applicable*, the concrete operation produces only concrete states for which an abstract equivalent is reachable using the abstract operation.

Similarly, in the blocking semantics, data type θADT forwards simulates data type $\theta ADT'$, written $\theta ADT \preceq_{f-b} \theta ADT'$, if we can identify a retrieve relation r from *state* to $state'$ such that the following rules hold:

$$init' \subseteq init \mathbin{;} r$$
$$\forall n : \mathrm{dom}\, ops \bullet r(\!|\; \mathrm{dom}\,(ops\; n)\;|\!) \subseteq \mathrm{dom}\,(ops'\; n)$$
$$\forall n : \mathrm{dom}\, ops \bullet r \mathbin{;} ops'\; n \subseteq ops\; n \mathbin{;} r$$
$$r \mathbin{;} final' \subseteq final$$

The first rule states that for every initial concrete state there is a matching initial abstract state. The second insists that the concrete operation is defined whenever the abstract operation would be. The third rule states that for all concrete states reachable from an abstract state, the concrete operation produces only those concrete states for which an abstract equivalent is reachable using the abstract operation. Finally, the fourth rule insists that the finalisation operations are consistent.

Given data types A and C with state spaces $AState$ and $CState$, with initialisations and finalisations $AInit$ and $AFinal$, and $CInit$ and $CFinal$ respectively and with corresponding operations AOp and COp, and given retrieve relation $Retr$, the schema versions of the non-blocking forwards simulation rules are:

$$\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \mid AInit \bullet Retr$$
$$\forall\, AState \mid \mathrm{pre}\; AOp \bullet \forall\, CState \mid Retr \bullet \mathrm{pre}\; COp$$
$$\forall\, AState \mid \mathrm{pre}\; AOp \bullet \forall\, CState \mid Retr \bullet$$
$$\quad \forall\, CState' \mid COp \bullet \exists\, AState' \mid AOp \bullet Retr'$$
$$\forall\, AState;\; Env \bullet Retr \wedge CFinal \Rightarrow AFinal$$

Similarly, the schema versions of the blocking forwards simulation rules are as follows.

$$\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \mid AInit \bullet Retr$$
$$\forall\, AState \mid \mathrm{pre}\; AOp \bullet \forall\, CState \mid Retr \bullet \mathrm{pre}\; COp$$
$$\forall\, AState \bullet \forall\, CState \mid Retr \bullet \forall\, CState' \mid COp \bullet \exists\, AState' \mid AOp \bullet Retr'$$
$$\forall\, AState;\; Env \bullet Retr \wedge CFinal \Rightarrow AFinal$$

Backward simulation rules

In the non-blocking semantics, data type θADT backwards simulates data type $\theta ADT'$ if we can identify a retrieve relation s from $state'$ to $state$ such that the following rules hold:

$$init' \mathbin{;} s \subseteq init$$
$$\forall n : \mathrm{dom}\, ops \bullet state' \setminus \mathrm{dom}\, ops'\; n \subseteq \mathrm{dom}(s \rhd (\mathrm{dom}\, ops\; n))$$
$$\forall n : \mathrm{dom}\, ops \bullet \mathrm{dom}(s \rhd (\mathrm{dom}\, ops\; n)) \lhd (ops'\; n) \mathbin{;} s \subseteq s \mathbin{;} (ops\; n)$$
$$final' \subseteq s \mathbin{;} final$$

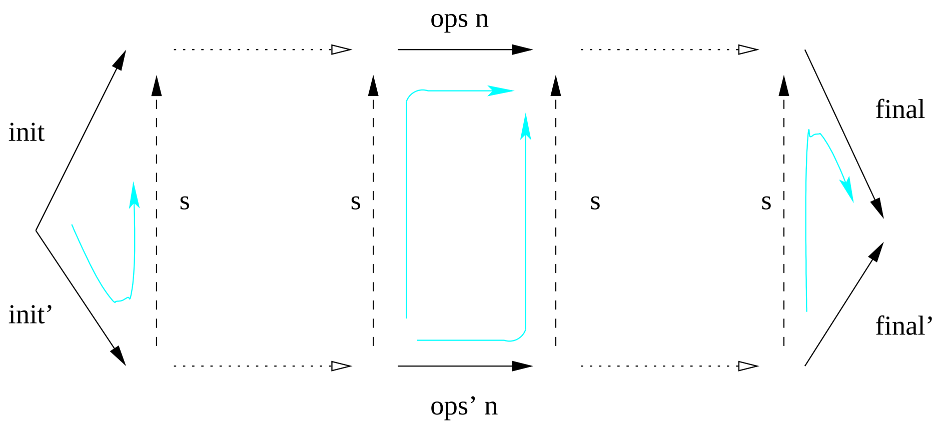

Figure 4.3: A backwards simulation

where $\lhd$ denotes domain subtraction, and $\rhd$ denotes range subtraction: see [Spi92]. If this is the case we write $\theta ADT \preceq_{b-n} \theta ADT'$. The complement condition, the second rule, is a point-free way of expressing the condition that any concrete state for which every corresponding abstract state is in the domain of the abstract operation must lie within the domain of the concrete operation: $\forall\, c : state' \bullet s \,(\!|\, \{c\} \,|\!)\, \subseteq \mathrm{dom}(ops\; n) \Rightarrow c \in dom\,(ops'\; n)$.

Similarly, in the blocking semantics, data type θADT backwards simulates data type $\theta ADT'$, written $\theta ADT \preceq_{b-b} \theta ADT'$, if we can identify a retrieve relation s from $state'$ to $state$ such that the following rules hold:

$$init' \fatsemi s \subseteq init$$

$$\forall\, n : \mathrm{dom}\; ops \bullet state' \setminus \mathrm{dom}\; ops'\; n \subseteq \mathrm{dom}(s \rhd (\mathrm{dom}\; ops\; n))$$

$$\forall\, n : \mathrm{dom}\; ops \bullet (ops'\; n) \fatsemi s \subseteq s \fatsemi (ops\; n)$$

$$final' \subseteq s \fatsemi final$$

Given data types A and C with state spaces $AState$ and $CState$, with initialisations and finalisations $AInit$ and $AFinal$, and $CInit$ and $CFinal$ respectively and with corresponding operations AOp and COp, and given retrieve relation $Retr$, the schema versions of the non-blocking backwards simulation rules are:

$$\forall\, AState';\ CState' \bullet CInit \land Retr' \Rightarrow AInit$$

$$\forall\, CState \bullet (\forall\, AState \bullet Retr \Rightarrow \mathrm{pre}\; AOp) \Rightarrow \mathrm{pre}\; COp$$

$$\forall\, AState';\ CState;\ CState' \mid (\forall\, AState \bullet Retr \Rightarrow \mathrm{pre}\; AOp) \bullet$$
$$\quad COp \land Retr' \Rightarrow \exists\, AState \bullet Retr \land AOp$$

$$\forall\, CState;\ Env \mid CFinal \bullet \exists\, AState \mid Retr \bullet AFinal$$

Similarly, the schema versions of the blocking backwards simulation rules are as follows.

$$\forall\, AState';\ CState' \bullet CInit \land Retr' \Rightarrow AInit$$
$$\forall\, CState \bullet (\forall\, AState \bullet Retr \Rightarrow \text{pre}\ AOp) \Rightarrow \text{pre}\ COp$$
$$\forall\, AState';\ CState;\ CState' \mid COp \land Retr' \bullet (\exists\, AState \bullet Retr \land AOp)$$
$$\forall\, CState;\ Env \mid CFinal \bullet \exists\, AState \mid Retr \bullet AFinal$$

4.4.1 Soundness and completeness of the simulation rules

Now we prove the soundness and joint completeness of the simulation rules for simple data types within both the blocking and the non-blocking semantic models.

Theorem 10 Soundness of the simulation rules for ADTs.
If one data type data type simulates another then the first is refined by the second. Given data types θADT and $\theta ADT'$,

$$\begin{array}{lcl} \theta ADT \preceq_f \theta ADT' & \Rightarrow & \theta ADT \sqsubseteq_{\mathcal{R}_{ADT}} \theta ADT' \\ \theta ADT \preceq_b \theta ADT' & \Rightarrow & \theta ADT \sqsubseteq_{\mathcal{R}_{ADT}} \theta ADT' \\ \theta ADT \preceq \theta ADT' & \Rightarrow & \theta ADT \sqsubseteq_{\mathcal{R}_{ADT}} \theta ADT'. \end{array}$$

Proof This result is proved for data types within the context of the non-blocking semantics in [WD96]. We prove the corresponding result for data types within the context of the blocking semantics in [BDW99]. The result is established for forwards and backwards simulation by induction on the length of the program. It is established for simulation in general by transitivity. □

Theorem 11 Joint completeness of the simulation rules for ADTs.
If one data type data type is refined by another then the first simulates the second. Given data types θADT and $\theta ADT'$,

$$\theta ADT \sqsubseteq_{\mathcal{R}_{ADT}} \theta ADT' \Rightarrow \theta ADT \preceq \theta ADT'.$$

Proof This result is proved for data types within the context of the non-blocking semantics in [WD96]. We prove the corresponding result for data types within the context of the blocking semantics in [BDW99]. The proof of the completeness of the simulation rules merits attention.

We first show how the *backwards normal form* of any given data type can be constructed. This is another data type with the same operations and identical behaviour to the given data type but with all non-determinism delayed to finalisation. The state space of the normal form of a data type is the power set of the state space of the original data type; similarly, the initialisation relation of the normal form maps each element in the environment onto its relational image under the initialisation relation on the given data type; the set of operations on the normal form is the same as the set of operations on the given data type, and each operation on the normal form relates sets of states to their relational image under that operation on the given data type; and the finalisation relation on the normal form of a data type relates sets of states set to every point in their relational image under the finalisation relation on the given data type. More formally, the function that takes a simple abstract data type and returns its normal form is defined as follows.

$$
\begin{array}{l}
[State, Name, Env] \\
\hline
bnf : ADT[State, Name, Env] \rightarrowtail ADT[\mathbb{P}\, State, Name, Env] \\
\hline
\forall\, ADT[State, Name, Env];\ ADT'[\mathbb{P}\, State, Name, Env] \bullet \\
\quad \theta ADT' = bnf\ (\theta ADT\ ) \Leftrightarrow \\
\quad\quad (\, state' = \mathbb{P}\, state \ \land \\
\quad\quad\ \ init' = \{e : Env;\ X : state' \mid X = init(\!|\ \{e\}\ |\!)\} \ \land \\
\quad\quad\ \ final' = \{X : state';\ e : Env \mid e \in final(\!|\ X\ |\!)\} \ \land \\
\quad\quad\ \ \mathrm{dom}\, ops' = \mathrm{dom}\, ops \ \land \\
\quad\quad\ \ \forall\, n : \mathrm{dom}\, ops';\ X : state' \bullet ops'\, n\, X = ops\, n(\!|\ X\ |\!)
\end{array}
$$

We prove that any data type backwards simulates its normal form using a retrieve relation that maps any set of elements in the state space individually onto each element in the set.

$$
\begin{array}{l}
s : \mathbb{P}\, state \leftrightarrow state \\
\hline
s = \{A : \mathbb{P}\, state;\ a : state \mid a \in A\}
\end{array}
$$

We then prove that if one data type is refined by another then the backwards normal form of the first forwards simulates the second. The retrieve relation identifies sets of states of the first data type with individual states of the second if they can each be reached following an equivalent sequence of interaction.

$$
\begin{array}{l}
r : \mathbb{P}\, state \leftrightarrow state' \\
\hline
r = \{A : \mathbb{P}\, state;\ c : state' \mid \\
\quad\quad \exists\, e : Env;\ prog : \mathrm{seq}\, Name \bullet \\
\quad\quad\quad A = init \fatsemi run_{ADT}\, prog\ ops(\!|\ \{e\}\ |\!) \\
\quad\quad\quad \land \\
\quad\quad\quad e \mapsto c \in init' \fatsemi run_{ADT}\, prog\ ops'\ \}
\end{array}
$$

Given any pair of data types such that the first is refined by the second it then follows that we can construct a third data type, the backwards normal form of the first such that the third data type is backwards simulated by the first and forward simulates the second. Hence, by transitivity it follows that the first data type simulates the second. □

4.5 Corresponding CSP processes

Now we define a translation from abstract data types expressed in Z to their process equivalents expressed in CSP. We use this translation to explore the relationship between their respective refinement orderings. This will be done both for the blocking and non-blocking semantic models. The processes corresponding to our data types will be parametrised by the current value of the internal state space. Initially, given an input e, it can be any value from the relational image of e under *init*. The body of the process will contain an external choice of operation followed by an internal choice of after-state and the communication of the associated event.

The process corresponding to the blocking semantics will accept any operation provided the current parameter lies in the domain of the operation. More concretely, if the current parameter is a then the operation *name* can be chosen only if the predicate

$$a \;\in\; \operatorname{dom} \mathit{ops}\,\mathit{name}$$

is true. If this is not the case then the operation *name* simply cannot be chosen—it is blocked. If the predicate $a \in \operatorname{dom} \mathit{ops}\,\mathit{name}$ is true and *name* is the chosen operation, then an after-state a' such that the predicate $a \mapsto a' \in \mathit{ops}\,n$ is true will be chosen internally. After the event op.*name* has been communicated, the process will continue with parameter a'. We see that the body of the process corresponding to a data type within the context of the blocking semantics is as described below:

$$\begin{array}{l} \Box\, \mathit{name} : \operatorname{dom} \mathit{ops} \mid a \in \operatorname{dom} \mathit{ops}\,\mathit{name} \bullet \\ \qquad\qquad \sqcap\, a' : \mathit{state} \mid a \mapsto a' \in \mathit{ops}\,\mathit{name} \bullet \\ \qquad\qquad\qquad\qquad \mathrm{op}.\mathit{name} \rightarrow \mathit{Proc}(a') \end{array}$$

At initialisation, since $\operatorname{dom} \mathit{init} = \mathit{Env}$, we can externally choose any e from the environment. A state a' such that $e \mapsto a' \in \mathit{init}$ will then be chosen internally and the process will continue with parameter a'. The component of the process corresponding to initialisation is therefore as follows:

$$\Box\, e : \mathit{Env} \;\bullet\; \sqcap\, a' : \mathit{State} \mid e \mapsto a' \in \mathit{init} \bullet \mathrm{init}.e \rightarrow \mathit{Proc}(a')$$

Since the finalisation operation is total we can choose to finalise at any time. Any value e' from the environment such that $a \mapsto e' \in \mathit{final}$, where state a is the current parameter, can be internally chosen. The value e' is then communicated on the channel final before the process terminates.

$$\sqcap\, e' : \mathit{Env} \mid a \mapsto e' \in \mathit{final} \bullet \mathrm{final}.e' \rightarrow \mathit{Stop}$$

Combining these observations, we see that the function $\mathit{process}_b$ which takes an abstract data type that has been given a blocking semantics and returns the corresponding CSP process is as defined below:

$$\begin{array}{l}
process_b\,(\,\theta ADT[State, Name, Env]\,) \;=\\
\quad \text{let}\\
\qquad\qquad Proc(a) \;=\\
\qquad\qquad\quad \Box\, name : \mathrm{dom}\, ops \mid a \in \mathrm{dom}\, ops\; name \bullet\\
\qquad\qquad\qquad \sqcap\, a' : State \mid a \mapsto a' \in ops\; name \bullet \mathrm{op}.name \rightarrow Proc(a')\\
\qquad\qquad\quad \Box\\
\qquad\qquad\quad \sqcap\, e' : Env \mid a \mapsto e' \in final \bullet \mathrm{final}.e' \rightarrow Stop\\
\quad \text{within}\\
\qquad \Box\, e : Env \;\bullet\; \sqcap\, a' : State \mid e \mapsto a' \in init \bullet \mathrm{init}.e \rightarrow Proc(a')
\end{array}$$

The process corresponding to the non-blocking semantics is similar to the process corresponding to a blocking semantics except that it might accept any named operation. If the predicate $a \in \mathrm{dom}\, ops\; name$ is true then the behaviour will be the same as the process corresponding to the blocking semantics. However, if the predicate is false then subsequent behaviour is completely undefined.

$$\begin{array}{l}
process_n\,(\,\theta ADT[State, Name, Env]\,) \;=\\
\quad \text{let}\\
\qquad Proc(a) \;=\\
\qquad\quad \Box\, name : \mathrm{dom}\, ops \mid a \in \mathrm{dom}\, ops\; name \bullet\\
\qquad\qquad \sqcap\, a' : State \mid a \mapsto a' \in ops\; name \bullet \mathrm{op}.name \rightarrow Proc(a')\\
\qquad\quad \Box\\
\qquad\quad \Box\, name : \mathrm{dom}\, ops \mid a \notin \mathrm{dom}\, ops\; name \bullet\\
\qquad\qquad Stop \;\sqcap\; \mathrm{op}.name \rightarrow chaos_{ADT}\,(\,\theta ADT[State, Name, Env]\,)\\
\qquad\quad \Box\\
\qquad\quad \sqcap\, e' : Env \mid a \mapsto e' \in final \bullet \mathrm{final}.e' \rightarrow Stop\\
\quad \text{within}\\
\qquad \Box\, e : Env \;\bullet\; \sqcap\, a' : State \mid e \mapsto a' \in init \bullet \mathrm{init}.e \rightarrow Proc(a')
\end{array}$$

The process $chaos_{ADT}$[1] can accept or refuse any sequence of events corresponding to named operations until a *final* event occurs. Once a *final* event has occured it will terminate.

$$\begin{array}{l}
chaos_{ADT}\,(\,\theta ADT[State, Name, Env]\,) \;=\\
\quad (\; \Box\, name : \mathrm{dom}\, ops \bullet \mathrm{op}.name \rightarrow chaos_{ADT}\,(\,\theta ADT[State, Name, Env]\,)\\
\qquad \sqcap\\
\qquad Stop\;)\\
\quad \Box\\
\quad \sqcap\, e' : Env \bullet \mathrm{final}.e' \rightarrow Stop
\end{array}$$

[1] $chaos_{ADT}$ is based upon Roscoe's [Ros98] process *Chaos*.

4.5.1 Data refinement and process refinement

Having identified CSP processes that correspond to our abstract data types, we now investigate the relationship between data refinement and process refinement. We prove first the surprising and important result that data refinement is not equivalent to stable failures refinement. This result is surprising in that it contradicts results in the literature [SD97, BDW99, Fis00, DB01], and important in that it was the catalyst that caused us to investigate the properties of data refinement and to propose the singleton failures refinement ordering. We then prove that data refinement is equivalent to singleton failures refinement.

We prove that data refinement is not equivalent to stable failures refinement with a simple example. Let data types P and Q be defined as follows: both data types have the same internal state space *State*; given environment *Env* containing the single element e, both have the same finalisation operations; and both data types share the operations A, B and C as illustrated in Figure 4.4. Operation A maps state 1 to state 0; operation B

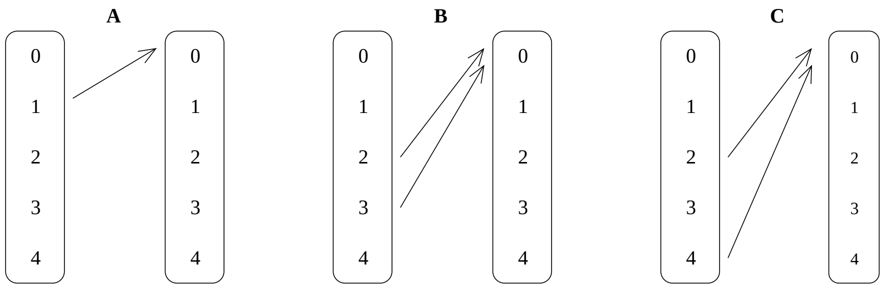

Figure 4.4: Relations describing operations A, B and C on data types P and Q.

maps states 2 and 3 to state 0; and operation C maps states 2 and 4 to state 0. More formally,

State
$s : 0 \,..\, 4$

Final
$State;\ Env'$

A
$\Delta\, State$
$s = 1 \ \wedge\ s' = 0$

B
$\Delta\, State$
$s \in \{2, 3\} \ \wedge\ s' = 0$

C
$\Delta\, State$
$s \in \{2, 4\} \ \wedge\ s' = 0$

The difference between the two data types lies in the range of their initialisation relations:

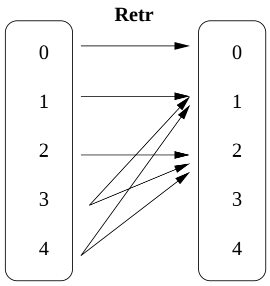

Figure 4.5: The backwards relation *Retr* from $State_Q$ to $State_P$.

data type P can initially be in states 1 or 2 whereas data type Q can initially be in states 1, 3 or 4.

PInit
$Env;\ State'$
$s' \in \{1, 2\}$

QInit
$Env;\ State'$
$s' \in \{1, 3, 4\}$

Lemma 5 Data type P is refined by data type Q where P and Q are as defined above.

Proof We prove that data type P is refined by data type Q within the context of the blocking semantics by proving that P simulates Q; more specifically we identify a retrieve relation *Retr* such that the backwards simulation rules corresponding to the blocking semantics all hold. Refinement follows automatically due to the soundness of the simulation rules. Identical techniques can be used to prove that P is also refined by Q within the context of the non-blocking semantics. Indeed, since three out of the four backwards simulation rules are identical for both contexts, only the rule concerning the availability of operations need be verified.

For clarity, we define $State_P$ and $State_Q$ to be equal to *State*; A_P and A_Q to be equal to A; B_P and B_Q to be equal to B; C_P and C_Q to be equal to C; and $Final_P$ and $Final_Q$ to be equal to *Final*. Then letting relation *Retr* from the state space of Q to the state space of P be as defined below and illustrated in Figure 4.5:

Retr
$p : State_P$ $q : State_Q$
$(\, q \in \{0, 1, 2\} \ \wedge \ p = q \,) \ \vee$ $(\, q \in \{3, 4\} \ \wedge \ p \in \{1, 2\} \,)$

we simply need to prove that the following predicates are true.

$$\forall\, State_P{}';\ State_Q{}' \bullet QInit \wedge Retr' \Rightarrow PInit$$

$$\forall\, State_Q \bullet (\forall\, State_P \bullet Retr \Rightarrow \mathrm{pre}\ A_P) \Rightarrow \mathrm{pre}\ A_Q$$
$$\forall\, State_Q \bullet (\forall\, State_P \bullet Retr \Rightarrow \mathrm{pre}\ B_P) \Rightarrow \mathrm{pre}\ B_Q$$
$$\forall\, State_Q \bullet (\forall\, State_P \bullet Retr \Rightarrow \mathrm{pre}\ C_P) \Rightarrow \mathrm{pre}\ C_Q$$

$$\forall\, State_P{}';\ State_Q;\ State_Q{}' \mid A_Q \wedge Retr' \bullet (\exists\, State_P \bullet Retr \wedge A_P)$$
$$\forall\, State_P{}';\ State_Q;\ State_Q{}' \mid B_Q \wedge Retr' \bullet (\exists\, State_P \bullet Retr \wedge B_P)$$
$$\forall\, State_P{}';\ State_Q;\ State_Q{}' \mid C_Q \wedge Retr' \bullet (\exists\, State_P \bullet Retr \wedge C_P)$$

$$\forall\, State_Q;\ Env \mid Final_Q \bullet \exists\, State_P \mid Retr \bullet Final_P$$

The first rule reduces to

$$\begin{array}{l} \forall\, p', q' : State \bullet \\ \quad ((\, q' = 1 \wedge p' = 1\,) \ \vee ((\, q' = 3 \vee q' = 4\,) \wedge (\, p' = 1 \vee p' = 2\,))) \\ \qquad\qquad \Rightarrow \\ \qquad\qquad\quad (\, p' = 1 \vee p' = 2\,) \end{array}$$

which is equivalent to true.

In the case of operation a, the inner clause of the second rule, or to be precise the predicate $\forall\, State_P \bullet Retr \Rightarrow \mathrm{pre}\ A_P$, is equivalent to

$$\begin{array}{l} \forall\, p : State \bullet \\ \quad ((q = 0 \wedge p = 0) \ \vee \ (q = 1 \wedge p = 1) \ \vee \ (q = 2 \wedge p = 2) \ \vee \\ \quad\ (q = 3 \wedge (p = 1 \vee p = 2)) \ \vee \ (q = 4 \wedge (p = 1 \vee p = 2))) \\ \qquad \Rightarrow \\ \qquad\quad p = 1 \end{array}$$

which reduces to $(q \neq 0) \wedge (q \neq 2) \wedge (q \neq 3) \wedge (q \neq 4)$. Hence, in the case of operation a, the second rule is equivalent to

$$\forall\, q : State \ \bullet \ ((q \neq 0) \wedge (q \neq 2) \wedge (q \neq 3) \wedge (q \neq 4)) \ \Rightarrow \ q = 1$$

which is of course true since $State = 0\,..\,4$. Similarly, in the case of operation b, the inner clause of the second rule is equivalent to

$$\begin{array}{l} \forall\, p : State \bullet \\ \quad ((q = 0 \wedge p = 0) \ \vee \ (q = 1 \wedge p = 1) \ \vee \ (q = 2 \wedge p = 2) \ \vee \\ \quad\ (q = 3 \wedge (p = 1 \vee p = 2)) \ \vee \ (q = 4 \wedge (p = 1 \vee p = 2))) \\ \qquad \Rightarrow \\ \qquad\quad (\, p = 2 \ \vee p = 3\,) \end{array}$$

which reduces to $(q \neq 0) \wedge (q \neq 1) \wedge (q \neq 3) \wedge (q \neq 4)$. Hence, in the case of operation b, the second rule is equivalent to

$$\begin{array}{l} \forall\, q : State \bullet \\ \qquad (\,(q \neq 0) \wedge (q \neq 1) \wedge (q \neq 3) \wedge (q \neq 4)\,) \;\Rightarrow\; (\,(q = 2) \vee (\,q = 3)\,) \end{array}$$

or equivalently $\forall\, q : State \bullet q = 2 \Rightarrow (q = 2 \vee q = 3)$ which is once again true. Furthermore, by symmetry it follows immediately that the second rule must also be true for operation c.

We have shown that rule one holds and that rules two holds for each of the operations on the data types. Next we consider the third rule. In the case of operation a this rule reduces to

$$\begin{array}{l} \forall\, p', q, q' : State \mid q = 1 \wedge q' = 0 \wedge p' = 0 \bullet \\ \qquad \exists\, p : State \bullet p = 1 \wedge p' = 0 \wedge (\,q = 1 \vee q = 3 \vee q = 4\,) \end{array}$$

which is equivalent to true. In the case of operation b, the third rule reduces to

$$\begin{array}{l} \forall\, p', q, q' : State \mid (\,q = 2 \vee q = 3\,) \wedge q' = 0 \wedge p' = 0 \bullet \\ \qquad \exists\, p : State \bullet p = 2 \wedge p' = 0 \wedge (\,q = 2 \vee q = 3 \vee q = 4\,) \\ \qquad\qquad\qquad\quad \vee \\ \qquad\qquad\qquad\quad p = 3 \wedge p' = 0 \wedge \text{false} \end{array}$$

which is also equivalent to true. Once more we appeal to symmetry to establish immediately that the third rule must also be true for operation c.

Finally, observing that the fourth simulation rule is trivially true, we conclude that all the backwards simulation rules hold and hence that P simulates Q. Hence, by Theorem 10, we deduce that data type P is refined by data type Q.

$$P \quad \sqsubseteq_{\mathcal{R}_{ADT}} \quad Q$$

$\square$

We now apply the function $process_b$ to data types P and Q to obtain the corresponding processes.

$$\begin{array}{lll} process_b\,(P) \;=\; \text{let} & & \\ \qquad P(s) & = & (s == 0)\ \&\ Stop \\ & & \square \\ & & (s == 1)\ \&\ \text{op}.a \rightarrow P(0) \\ & & \square \\ & & (s == 2)\ \&\ (\text{op}.b \rightarrow P(0) \,\square\, \text{op}.c \rightarrow P(0)) \\ & & \square \\ & & (s == 3)\ \&\ \text{op}.b \rightarrow P(0) \\ & & \square \\ & & (s == 4)\ \&\ \text{op}.c \rightarrow P(0) \\ & & \square \\ & & \text{final}.e \rightarrow Stop \\ \text{within} & & \\ \qquad \square\, e : \{e\} \;\bullet\; \sqcap\, s : \{1, 2\} \;\bullet\; \text{init}.e \rightarrow P(s) & & \end{array}$$

With a little rearranging and some obvious renaming, this process can equivalently be written as follows:

$$
\begin{array}{l}
i \rightarrow (\ (\ a \rightarrow f \rightarrow \mathit{Stop} \\
\qquad\quad \sqcap \\
\qquad\quad (\ b \rightarrow f \rightarrow \mathit{Stop}\ ) \mathbin{\Box} (\ c \rightarrow f \rightarrow \mathit{Stop}\ )\ ) \\
\qquad \Box \\
\qquad f \rightarrow \mathit{Stop}\ )
\end{array}
$$

Applying the function $\mathit{process}_b$ to Q and re-writing in a similar way, we see that $\mathit{process}_b\,(Q)$ is equivalent to the following process:

$$
\begin{array}{l}
i \rightarrow (\ (\ a \rightarrow f \rightarrow \mathit{Stop} \\
\qquad\quad \sqcap \\
\qquad\quad b \rightarrow f \rightarrow \mathit{Stop} \\
\qquad\quad \sqcap \\
\qquad\quad c \rightarrow f \rightarrow \mathit{Stop}\ ) \\
\qquad \Box \\
\qquad f \rightarrow \mathit{Stop}\ )
\end{array}
$$

Observation 7 The process corresponding to P is not refined within the stable failures model by the process corresponding to Q where data types P and Q are as defined above.

Proof This result holds within the context of either the blocking or the non-blocking semantic models. In the proof presented below we will work within the context of the blocking model (and hence use the processes identified above), however, the same result can be obtained within the context of the non-blocking model. Indeed the trace-refusal pairs identified below that lie in $\mathcal{F}\,[\![\mathit{process}_b(Q)]\!]$ but not $\mathcal{F}\,[\![\mathit{process}_b(P)]\!]$ also lie in $\mathcal{F}\,[\![\mathit{process}_n(Q)]\!]$ but not $\mathcal{F}\,[\![\mathit{process}_n(P)]\!]$.

We see that after event i, $\mathit{process}_b(P)$ can refuse any set of events that does not contain f and that does not contain both a and b, or both a and c, whereas after event i, $\mathit{process}_b(Q)$ can refuse any set of events that does not contain f and that contains at most two of a, b and c. In particular, $\mathit{process}_b(Q)$ can refuse the sets $\{a, b\}$ and $\{a, c\}$ whereas $\mathit{process}_b(P)$ cannot: the trace-refusal pairs $(\langle i \rangle, \{a, b\})$ and $(\langle i \rangle, \{a, c\})$ both lie in the failures of $\mathit{process}_b(Q)$ but not of $\mathit{process}_b(P)$.

We have shown that the set of failures of the process corresponding to Q is not a subset of the set of failures of the process corresponding to P. Hence the process corresponding to P is not refined within the stable failures model by the process corresponding to Q.

$$\mathcal{F}\,[\![\mathit{process}\,(Q)]\!] \nsubseteq \mathcal{F}\,[\![\mathit{process}\,(P)]\!] \quad \text{i.e.} \quad \mathit{process}\,(P) \not\sqsubseteq_{\mathcal{F}} \mathit{process}\,(Q)$$

$\Box$

Observation 8 Data refinement is not equivalent to stable failures refinement.

$$\begin{array}{l} \forall\, ADT;\ ADT' \bullet \\ \quad \theta ADT \sqsubseteq_{\mathcal{R}_{ADT}} \theta ADT' \\ \qquad \not\Rightarrow \\ \qquad\quad \mathit{process}\,(\theta ADT) \sqsubseteq_{\mathcal{F}} \mathit{process}\,(\theta ADT') \end{array}$$

Proof We have identified two data types P and Q. In Lemma 5 we proved that P is refined by Q within the context of both the blocking and non-blocking semantic models. In Observation 7 we showed that the process corresponding to P is not refined within the stable failures semantic model by the process corresponding to Q in either context.

$$P \sqsubseteq_{\mathcal{R}_{ADT}} Q \qquad \text{but} \qquad \mathit{process}\,(P) \not\sqsubseteq_F \mathit{process}\,(Q)$$

From this it follows that data refinement does not imply stable failures refinement: data refinement and stable failures refinement are not equivalent. □

In [BDW99] we proved that *for data types with a single operation* stable failures refinement is equivalent to data refinement. We then incorrectly supposed that this result was extensible and concluded that *for all data types* stable failures refinement is equivalent to data refinement. The result was not extensible because whilst data refinement records refusal information on an operation-by-operation basis it does not record the availability of combinations of operations. This is an important part of the information content of the stable failures model.

Having shown that data refinement is not equivalent to stable failures refinement we now prove that it is equivalent to singleton failures refinement.

Theorem 12 Data refinement is equivalent to singleton failures refinement.

$$\begin{array}{l} \forall\, ADT;\ ADT' \bullet \\ \quad \theta ADT \sqsubseteq_{\mathcal{R}_{ADT}} \theta ADT' \\ \qquad \Leftrightarrow \\ \qquad\quad \mathit{process}\,(\theta ADT) \sqsubseteq_{\mathcal{S}} \mathit{process}\,(\theta ADT') \end{array}$$

Proof We present the proof within the context of the blocking model. Arguing along similar lines gives the corresponding result within the context of the non-blocking semantics, although extra care need to be taken since, as shown above, trace-refusal pairs of the same form (either unfinished or finished) can be produced by different means.

Before we begin the proof we introduce some functions that will translate constructs from the event-based framework to constructs in the relational framework. First we define *Prog* to be the set of all programs and Ref_1 to be any set of names of cardinality of at most one.

$$\begin{array}{rcl} \mathit{Prog} & == & \mathrm{seq}\, \mathit{Name} \\ \mathit{Ref}_1 & == & \{\, n : \mathit{Name} \bullet \{\, n \,\}\,\} \cup \{\, \emptyset \,\} \end{array}$$

Next we introduce two functions, *toTrace* and *toProg*, that relate programs and traces.

$$\begin{array}{|l}
toTrace : Prog \rightarrow Trace \\
toProg : Trace \rightarrowtail Prog \\
\hline
\forall\, prog : Prog \bullet\ toTrace\,(\,prog\,) \ = \ \langle i : 1 \mathinner{..} \#prog \bullet \mathrm{op}.(prog.i)\rangle \ \wedge \\
\qquad\qquad\qquad\quad toProg \ = \ toTrace^{-1}
\end{array}$$

Finally we introduce the function *toEvent* that relates names of operations to their corresponding events.

$$\begin{array}{|l}
toEvent : Name \rightarrow Event \\
\hline
toEvent \ = \ \{name : Name \bullet name \mapsto \mathrm{op}.name\}
\end{array}$$

Our proof strategy is as follows: we first simplify the predicate $\theta ADT \sqsubseteq_{\mathcal{R}_{ADT}} \theta ADT'$ to obtain a set of necessary and sufficient conditions for data refinement; we then simplify the predicate $process\,(\theta ADT) \sqsubseteq_{\mathcal{S}} process\,(\theta ADT')$ to obtain a set of necessary and sufficient conditions for singleton failures refinement; finally we show that these sets of constraints are equivalent.

Data refinement Given data types θADT and $\theta ADT'$ we reason as follows:

$$\theta ADT \ \sqsubseteq_{\mathcal{R}_{ADT}} \ \theta ADT'$$

$\Leftrightarrow$ [def^n of $\sqsubseteq_{\mathcal{R}_{ADT}}$]

$$\forall\, prog : Prog \bullet \mathcal{R}_{ADT}[\,\theta ADT'\,]\,prog \subseteq \mathcal{R}_{ADT}[\,\theta ADT\,]\,prog$$

$\Leftrightarrow$ [def^n of $\mathcal{R}_{ADT}$]

$$\begin{array}{l}
\forall\, prog : Prog \bullet \\
\qquad init'^{\top} \fatsemi run\ prog\ ops' \fatsemi final'^{\top} \subseteq init^{\top} \fatsemi run\ prog\ ops \fatsemi final^{\top}
\end{array}$$

$\Leftrightarrow$ [set theory]

$$\begin{array}{l}
\forall\, prog : Prog \bullet \\
\qquad init'^{\top} \fatsemi run\ prog\ ops' \fatsemi final'^{\top} \rhd Env \\
\qquad\qquad \subseteq \\
\qquad\qquad init^{\top} \fatsemi run\ prog\ ops \fatsemi final^{\top} \rhd Env \\
\qquad \wedge \\
\qquad Env \lhd init'^{\top} \fatsemi run\ prog\ ops' \fatsemi final'^{\top} \rhd \{\bot\} \\
\qquad\qquad \subseteq \\
\qquad\qquad Env \lhd init^{\top} \fatsemi run\ prog\ ops \fatsemi final^{\top} \rhd \{\bot\} \\
\qquad \wedge \\
\qquad \{\bot\} \lhd init'^{\top} \fatsemi run\ prog\ ops' \fatsemi final'^{\top} \\
\qquad\qquad \subseteq \\
\qquad\qquad \{\bot\} \lhd init^{\top} \fatsemi run\ prog\ ops \fatsemi final^{\top}
\end{array}$$

The third clause is trivially true. Hence, observing that $R^{\top} \rhd Env = R$ for any relation R and that $Env \lhd init^{\top} = init$, we continue reasoning as follows:

$$
\begin{array}{ll}
\Leftrightarrow & \text{[property of } \lhd, \rhd \text{ and } ^{\top}] \\
\quad \forall\, prog : Prog \bullet & \\
\qquad init' \mathbin{;} run\ prog\ ops' \mathbin{;} final' & \\
\qquad\quad \subseteq & \\
\qquad\quad init \mathbin{;} run\ prog\ ops \mathbin{;} final & \\
\qquad \wedge & \\
\qquad init' \mathbin{;} run\ prog\ ops' \mathbin{;} final'^{\top} \rhd \{\bot\} & \\
\qquad\quad \subseteq & \\
\qquad\quad init \mathbin{;} run\ prog\ ops \mathbin{;} final^{\top} \rhd \{\bot\} &
\end{array}
$$

Observe that given any $e : Env$,

$$
\begin{array}{l}
\exists\, s : state' \bullet e \mapsto s \in init' \mathbin{;} run\ prog\ ops' \\
\quad \Rightarrow \\
\qquad \exists\, e' : Env \bullet e \mapsto e' \in init' \mathbin{;} run\ prog\ ops' \mathbin{;} final'
\end{array}
$$

since *final* is a total operation, and hence that

$$
\begin{array}{l}
\exists\, s : state' \bullet e \mapsto s \in init' \mathbin{;} run\ prog\ ops' \\
\quad \Rightarrow \\
\qquad \exists\, e' : Env \bullet e \mapsto e' \in init \mathbin{;} run\ prog\ ops \mathbin{;} final
\end{array}
$$

whenever the predicate $init' \mathbin{;} run\ prog\ ops' \mathbin{;} final' \subseteq init \mathbin{;} run\ prog\ ops \mathbin{;} final$ is true. In addition, we see that since

$$
\begin{array}{l}
\exists\, e' : Env \bullet e \mapsto e' \in init \mathbin{;} run\ prog\ ops \mathbin{;} final \\
\quad \Rightarrow \\
\qquad \exists\, s : state \bullet e \mapsto s \in init \mathbin{;} run\ prog\ ops
\end{array}
$$

it follows immediately that

$$
\begin{array}{l}
\exists\, s : state' \bullet e \mapsto s \in init' \mathbin{;} run\ prog\ ops' \\
\quad \Rightarrow \\
\qquad \exists\, s : state \bullet e \mapsto s \in init \mathbin{;} run\ prog\ ops
\end{array}
$$

whenever $init' \mathbin{;} run\ prog\ ops' \mathbin{;} final' \subseteq init \mathbin{;} run\ prog\ ops \mathbin{;} final$ is true. Finally, applying the property that if $p \Rightarrow q$ for any predicates p and q then the predicates p and $p \wedge q$ are equivalent, it follows immediately that the predicates

$$init' \mathbin{;} run\ prog\ ops' \mathbin{;} final' \subseteq init \mathbin{;} run\ prog\ ops \mathbin{;} final$$

and

$$
\begin{array}{l}
init' \mathbin{;} run\ prog\ ops' \mathbin{;} final' \subseteq init \mathbin{;} run\ prog\ ops \mathbin{;} final \\
\wedge \\
\forall\, e : Env \bullet (\, \exists\, s : state' \bullet e \mapsto s \in init' \mathbin{;} run\ prog\ ops' \\
\qquad\qquad\qquad \Rightarrow \\
\qquad\qquad\qquad\quad \exists\, s : state \bullet e \mapsto s \in init \mathbin{;} run\ prog\ ops \,)
\end{array}
$$

are equivalent. We therefore continue reasoning as follows:

$$\theta ADT \sqsubseteq_{\mathcal{R}_{ADT}} \theta ADT'$$

$$\begin{array}{ll}
\Leftrightarrow & \text{[above reasoning]} \\
\quad \forall\, prog : Prog \bullet & \\
\qquad init' \comp run\ prog\ ops' \comp final' \subseteq init \comp run\ prog\ ops \comp final & \\
\qquad \land & \\
\qquad \forall\, e : Env \bullet (\exists\, s : state' \bullet e \mapsto s \in init' \comp run\ prog\ ops' & \\
\qquad\qquad\qquad \Rightarrow & \\
\qquad\qquad\qquad\quad \exists\, s : state \bullet e \mapsto s \in init \comp run\ prog\ ops\,) & \\
\qquad \land & \\
\qquad init' \comp run\ prog\ ops' \comp final'^{\top} \rhd \{\bot\} & \\
\qquad\quad \subseteq & \\
\qquad\quad init \comp run\ prog\ ops \comp final^{\top} \rhd \{\bot\} &
\end{array}$$

$$\begin{array}{ll}
\Leftrightarrow & \text{[property of } final\text{]} \\
\quad \forall\, prog : Prog \bullet & \\
\qquad init' \comp run\ prog\ ops' \comp final' \subseteq init \comp run\ prog\ ops \comp final & \\
\qquad \land & \\
\qquad \forall\, e : Env \bullet (\exists\, s : state' \bullet e \mapsto s \in init' \comp run\ prog\ ops' & \\
\qquad\qquad\qquad \Rightarrow & \\
\qquad\qquad\qquad\quad \exists\, s : state \bullet e \mapsto s \in init \comp run\ prog\ ops\,) & \\
\qquad \land & \\
\qquad \forall\, e : Env \bullet (\, e \mapsto \bot \in init' \comp run\ prog\ ops' & \\
\qquad\qquad\qquad \Rightarrow & \\
\qquad\qquad\qquad\quad e \mapsto \bot \in init \comp run\ prog\ ops\,) &
\end{array}$$

By exploiting the properties of $\forall$ we see that this reduces to the following three predicates.

a.

$$\forall\, prog : Prog \bullet init' \comp run\ prog\ ops' \comp final' \subseteq init \comp run\ prog\ ops \comp final$$

b.

$$\begin{array}{l}
\forall\, prog : Prog;\ e : Env \bullet \\
\quad \exists\, s : state' \bullet e \mapsto s \in init' \comp run\ prog\ ops' \\
\qquad \Rightarrow \\
\qquad\quad \exists\, s : state \bullet e \mapsto s \in init \comp run\ prog\ ops
\end{array}$$

c.

$$\begin{array}{l}
\forall\, prog : Prog;\ e : Env \bullet \\
\quad e \mapsto \bot \in init' \comp run\ prog\ ops' \ \Rightarrow\ e \mapsto \bot \in init \comp run\ prog\ ops
\end{array}$$

Observe first that predicate c is equivalent to predicate c′ below.

c′.

$$\begin{array}{l}
\forall\, prog : Prog;\; e : Env \bullet \\
\quad (\, \exists\, prog' : Prog \mid \#prog' \leq \#prog \;\wedge\; \forall\, i : 1 \,..\, \#prog' \bullet prog'.i = prog.i \;\bullet \\
\qquad\qquad \exists\, s : state' \bullet e \mapsto s \in init' \fatsemi run\,(front\, prog')\; ops' \\
\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad s \notin \mathrm{dom}\, ops'\,(last\, prog')\,) \\
\qquad \Rightarrow \\
\qquad\quad (\, \exists\, prog'' : Prog \mid \\
\qquad\qquad \#prog'' \leq \#prog \;\wedge\; \forall\, i : 1 \,..\, \#prog'' \bullet prog''.i = prog.i \;\bullet \\
\qquad\qquad\quad \exists\, s : state \bullet e \mapsto s \in init \fatsemi run\,(front\, prog'')\; ops \\
\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad s \notin \mathrm{dom}\, ops\,(last\, prog'')\,)
\end{array}$$

From this, by applying predicate b and the fact that c′ holds for *all* programs, we see that $prog'$ and $prog''$ must be the same sequence. Hence we deduce that predicate c′ is equivalent to predicate c″ below.

c″.

$$\begin{array}{l}
\forall\, prog : Prog;\; e : Env;\; n : Name \bullet \\
\quad \exists\, s : state' \bullet e \mapsto s \in init' \fatsemi run\; prog\; ops' \wedge s \notin \mathrm{dom}\, ops'\, n \\
\qquad \Rightarrow \\
\qquad\quad \exists\, s : state \bullet e \mapsto s \in init \fatsemi run\; prog\; ops \wedge s \notin \mathrm{dom}\, ops\, n
\end{array}$$

Having derived three predicates, a, b and c″, whose conjunction is true precisely when θADT is data refined by $\theta ADT'$, we turn our attention to the refinement of their corresponding processes.

Singleton failures Consider the failures of the process corresponding to a data type. These can be classified into three groups: failures after the empty trace; failures after an unfinished trace (an init event and any number of events corresponding to named operations have occurred but no final event has occurred); and failures after a finished trace (a final event has occurred).

After the empty trace any event other than init will be refused. After an unfinished trace an event final.e for some value e can be refused only if another distinct event final.e' could occur. In addition any event corresponding to an operation might be refused after an unfinished trace if a state outside its domain can be reached after the given trace. Finally all init events will be refused after an unfinished trace. After a finished trace any event will be refused. Hence, within the context of the blocking semantic model, the singleton failures of a process corresponding to a data type can be expressed as the disjoint union of three sets as shown below:

$$
\begin{array}{l}
\mathcal{S}\,[\![process_b(\theta ADT)]\!] \; = \\
\quad \{ ref : Refusal_1 \mid \mathrm{init}.Env \cap ref = \emptyset \bullet (\langle\rangle, ref) \} \\
\quad \cup \\
\quad \{ e : Env;\; prog : Prog;\; ref : Refusal_1 \mid \\
\qquad \exists\, s : state \bullet e \mapsto s \in init \mathbin{;} run\, prog\, ops \;\wedge \\
\qquad\qquad\qquad \forall\, n : \mathrm{dom}\, ops \bullet s \in \mathrm{dom}\, ops\, n \Rightarrow toEvent(n) \notin ref \;\wedge \\
\qquad\qquad\qquad \forall\, e' : Env \bullet \\
\qquad\qquad\qquad\quad \mathrm{final}.e' \in ref \;\Rightarrow\; \exists\, e'' : Env \mid e'' \neq e' \bullet e'' \in final(\!|\, \{s\} \,|\!) \;\bullet \\
\qquad\quad (\langle \mathrm{init}.e \rangle \frown toTrace(prog), ref) \} \\
\quad \cup \\
\quad \{ e, e' : Env;\; prog : Prog;\; ref : Refusal_1 \mid e \mapsto e' \in init \mathbin{;} run\, prog\, ops \mathbin{;} final \;\bullet \\
\qquad (\langle \mathrm{init}.e \rangle \frown toTrace(prog) \frown \langle \mathrm{final}.e' \rangle, ref) \}
\end{array}
$$

Within the context of the non-blocking semantic model we get additional trace-refusal pairs. These arise from the continuation of a program after an operation has been called outside its preconditions. We include both unfinished traces (after which only finalisation events may not be refused) and finished traces (after which any event may be refused). Hence, within the context of the non-blocking semantic model, the singleton failures of a process corresponding to a data type can be expressed as follows:

$$
\begin{array}{l}
\mathcal{S}\,[\![process_n(\theta ADT)]\!] \; = \\
\quad \mathcal{S}\,[\![process_b(\theta ADT)]\!] \\
\quad \cup \\
\quad \{ e : Env;\; prog : Prog;\; ref : Refusal_1 \mid \\
\qquad e \mapsto \bot \in init \mathbin{;} run\, prog\, ops \;\wedge\; \mathrm{final}.Env \cap ref = \emptyset \;\bullet \\
\qquad\quad (\langle \mathrm{init}.e \rangle \frown toTrace(prog), ref) \} \\
\quad \cup \\
\quad \{ e, e' : Env;\; prog : Prog;\; ref : Refusal_1 \mid \\
\qquad e \mapsto \bot \in init \mathbin{;} run\, prog\, ops \bullet \\
\qquad\quad (\langle \mathrm{init}.e \rangle \frown toTrace(prog) \frown \langle \mathrm{final}.e' \rangle, ref) \}
\end{array}
$$

Singleton failures refinement One process is refined by another within the singleton failures model if the set of singleton failures of the second process is a subset of the set of failures of the first process. Hence, by considering the disjoint sets above and applying a little set theory we see that the process corresponding to data type θADT is refined by the process corresponding to data type $\theta ADT'$ provided the following predicates are true:

1.

$$
\begin{array}{l}
\{ ref : Refusal_1 \mid \mathrm{init}.Env \cap ref = \emptyset \bullet (\langle\rangle, ref) \} \\
\quad \subseteq \\
\quad \{ ref : Refusal_1 \mid \mathrm{init}.Env \cap ref = \emptyset \bullet (\langle\rangle, ref) \}
\end{array}
$$

2.

$$
\begin{array}{l}
\{e : Env;\ prog : Prog;\ ref : Refusal_1 \mid \\
\qquad \exists\, s : state' \bullet e \mapsto s \in init' \mathbin{;} run\ prog\ ops' \ \wedge \\
\qquad\qquad \forall\, n : \mathrm{dom}\ ops' \bullet s \in \mathrm{dom}\ ops'\ n \Rightarrow toEvent(n) \notin ref \ \wedge \\
\qquad\qquad \forall\, e' : Env \bullet \mathrm{final}.e' \notin ref \ \vee \\
\qquad\qquad\qquad \exists\, e'' : Env \mid e'' \neq e' \bullet e'' \in final'(\!|\ \{s\}\ |\!) \ \bullet \\
\qquad\quad (\langle \mathrm{init}.e \rangle \frown toTrace(prog), ref)\} \\
\quad \subseteq \\
\quad \{e : Env;\ prog : Prog;\ ref : Refusal_1 \mid \\
\qquad \exists\, s : state \bullet e \mapsto s \in init \mathbin{;} run\ prog\ ops \ \wedge \\
\qquad\qquad \forall\, n : \mathrm{dom}\ ops \bullet s \in \mathrm{dom}\ ops\ n \Rightarrow toEvent(n) \notin ref \ \wedge \\
\qquad\qquad \forall\, e' : Env \bullet \mathrm{final}.e' \notin ref \ \vee \\
\qquad\qquad\qquad \exists\, e'' : Env \mid e'' \neq e' \bullet e'' \in final(\!|\ \{s\}\ |\!) \ \bullet \\
\qquad\quad (\langle \mathrm{init}.e \rangle \frown toTrace(prog), ref)\}
\end{array}
$$

3.

$$
\begin{array}{l}
\{e, e' : Env;\ prog : Prog;\ ref : Refusal_1 \mid \\
\qquad e \mapsto e' \in init' \mathbin{;} run\ prog\ ops' \mathbin{;} final' \ \bullet \\
\qquad\quad (\langle \mathrm{init}.e \rangle \frown toTrace(prog) \frown \langle \mathrm{final}.e' \rangle, ref)\} \\
\quad \subseteq \\
\quad \{e, e' : Env;\ prog : Prog;\ ref : Refusal_1 \mid \\
\qquad\quad e \mapsto e' \in init \mathbin{;} run\ prog\ ops \mathbin{;} final \ \bullet \\
\qquad\qquad (\langle \mathrm{init}.e \rangle \frown toTrace(prog) \frown \langle \mathrm{final}.e' \rangle, ref)\}
\end{array}
$$

Observe that predicate 1 concerning the empty trace is trivially true. By applying a little set theory and moving notationally away from the event-based framework towards the relational framework we can simplify the remaining two predicates to those below.

$2'$.

$$
\begin{array}{l}
\forall\, e : Env;\ prog : Prog;\ ref : Refusal_1 \bullet \\
\quad \exists\, s : state' \bullet e \mapsto s \in init' \mathbin{;} run\ prog\ ops' \ \wedge \\
\qquad\qquad \forall\, n : \mathrm{dom}\ ops' \bullet s \in \mathrm{dom}\ ops'\ n \Rightarrow toEvent(n) \notin ref \ \wedge \\
\qquad\qquad \forall\, e' : Env \bullet \mathrm{final}.e' \notin ref \ \vee \\
\qquad\qquad\qquad \exists\, e'' : Env \mid e'' \neq e' \bullet e'' \in final'(\!|\ \{s\}\ |\!) \ \bullet \\
\qquad \Rightarrow \\
\qquad \exists\, s : state \bullet e \mapsto s \in init \mathbin{;} run\ prog\ ops \ \wedge \\
\qquad\qquad \forall\, n : \mathrm{dom}\ ops \bullet s \in \mathrm{dom}\ ops\ n \Rightarrow toEvent(n) \notin ref \ \wedge \\
\qquad\qquad \forall\, e' : Env \bullet \mathrm{final}.e' \notin ref \ \vee \\
\qquad\qquad\qquad \exists\, e'' : Env \mid e'' \neq e' \bullet e'' \in final(\!|\ \{s\}\ |\!) \ \bullet
\end{array}
$$

$3'$.

$$\forall\, prog : Prog \bullet \\ \quad init' \fatsemi run\ prog\ ops' \fatsemi final' \subseteq init \fatsemi run\ prog\ ops \fatsemi final$$

Recalling that the set ref contains at most one event, we can simplify clause $2'$ further by considering separately the following cases: when ref is the empty set or contains only an init event; when ref contains an event corresponding to a named operation; and when ref contains a final event.

$2''$.

$$\begin{array}{l}
\forall\, e : Env;\ prog : Prog \bullet \\
\quad \exists\, s : state' \bullet e \mapsto s \in init' \fatsemi run\ prog\ ops' \\
\qquad \Rightarrow \\
\qquad \exists\, s : state \bullet e \mapsto s \in init \fatsemi run\ prog\ ops \\
\wedge \\
\forall\, e : Env;\ prog : Prog;\ n : Name \bullet \\
\quad \exists\, s : state' \bullet e \mapsto s \in init' \fatsemi run\ prog\ ops' \wedge s \notin \operatorname{dom} ops'\ n \\
\qquad \Rightarrow \\
\qquad \exists\, s : state \bullet e \mapsto s \in init \fatsemi run\ prog\ ops \wedge s \notin \operatorname{dom} ops\ n \\
\wedge \\
\forall\, e, e' : Env;\ prog : Prog \bullet \\
\quad \exists\, s : state';\ e'' : Env \mid e'' \neq e' \bullet e \mapsto s \in init' \fatsemi run\ prog\ ops' \wedge s \mapsto e'' \in final' \\
\qquad \Rightarrow \\
\qquad \exists\, s : state;\ e'' : Env \mid e'' \neq e' \bullet e \mapsto s \in init \fatsemi run\ prog\ ops \wedge s \mapsto e'' \in final
\end{array}$$

Now, the final clause of $2''$ reduces to

$$\begin{array}{l}
\forall\, e, e' : Env;\ prog : Prog \bullet \\
\quad \exists\, e'' : Env \mid e'' \neq e' \bullet e \mapsto e'' \in init' \fatsemi run\ prog\ ops' \fatsemi final' \\
\qquad \Rightarrow \\
\qquad \exists\, e'' : Env \mid e'' \neq e' \bullet e \mapsto e'' \in init \fatsemi run\ prog\ ops \fatsemi final
\end{array}$$

which is implied by clause $3'$. Therefore, since $(p \Rightarrow q) \Rightarrow (\,(p \wedge q) \equiv p\,)$ it follows that this clause is redundant. Hence clause $2''$ is equivalent to clause $2'''$ below.

$2'''$.

$$\begin{array}{l}
\forall\, e : Env;\ prog : Prog \bullet \\
\quad \exists\, s : state' \bullet e \mapsto s \in init' \fatsemi run\ prog\ ops' \\
\qquad \Rightarrow \\
\qquad \exists\, s : state \bullet e \mapsto s \in init \fatsemi run\ prog\ ops \\
\wedge \\
\forall\, e : Env;\ prog : Prog;\ n : Name \bullet \\
\quad \exists\, s : state' \bullet e \mapsto s \in init' \fatsemi run\ prog\ ops' \wedge s \notin \operatorname{dom} ops'\ n \\
\qquad \Rightarrow \\
\qquad \exists\, s : state \bullet e \mapsto s \in init \fatsemi run\ prog\ ops \wedge s \notin \operatorname{dom} ops\ n
\end{array}$$

Combining our observations We have shown that θADT is data refined by $\theta ADT'$ when predicates a, b and c$''$ all hold and that the process corresponding to θADT is refined within the singleton failures semantic framework by the process corresponding to $\theta ADT'$ when predicates $2'''$ and $3'$ both hold. However, since predicate a is equivalent to predicate $3'$, and the conjunction of predicates b and c$''$ is equivalent to predicate $2'''$ our desired result follows directly: data refinement is equivalent to singleton failures refinement. $\square$

Chapter 5

Communicating data types

In Chapter 4 we explored the relationship between data refinement and refinement within CSP: we proved that data refinement is not, as had previously been supposed, equivalent to stable failures refinement but that it is equivalent to singleton failures refinement. To facilitate this exploration we defined our data types in a most simplistic manner, seemingly taking no account of input and output. However, a data type with no notion of input and output is meaningless, and in this chapter we introduce communicating data types, data types with an explicit notion of input and output. We show how results from our simple data types can be extended to our communicating data types.

5.1 Syntactic definition

A communicating data type has three components: its internal state space, a non-empty set of initial states and a set of named operations. Each operation takes an input value and a state and then returns an output value and a state; for the purposes of this thesis we will suppose that all inputs and outputs are of the same type. Given the type of the internal state space *State* and the set of all possible input and output values *Value*, we model the operations on our communicating data types as relations of the type $(State \times Value) \leftrightarrow (State \times Value)$. Then, given the set *Name* of all possible names of operations we define our communicating data types generically in terms of the parameters *State*, *Name* and *Value*.

Communicating data types have three components: *state*, *init* and *ops*. The set *state*, a subset of *State*, is the internal state space of the data type and the set *init* is the set of all possible initial states; *init* is a non-empty subset of *state*. The function *ops* maps the name of each operation on the data type onto the relation describing that operation. Observe that, unlike the simple data types of Chapter 4, communicating data types have no *final* component. This is because finalisation is used as a means of returning outputs for simple data types; it is therefore redundant for communicating data types where communication is explicit. We discuss further the role of *init* and *final* on simple data types in Section 5.2. Our formal definition of communicating data types is as follows.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{CDT[State, Name, Value]} \\
state : \mathbb{P}\, State \\
init : \mathbb{P}\, State \\
ops : Name \pfun ((State \times Value) \leftrightarrow (State \times Value)) \\
\hline
\emptyset \subset init \subseteq state \;\wedge \\
ops \subseteq Name \pfun ((state \times Value) \leftrightarrow (state \times Value)) \\
\hline
\end{array}
$$

The first constraint tells us that the set of initial states is non-empty and that all initial states lie in the internal state space *state*. The second constraint tells us that the state component of each element in the domain or range of each of the named operations must lie in the internal state space *state*.

Example 4 A non-deterministic one place buffer.

We give a non-deterministic one place buffer as an example of a communicating data type. Our buffer satisfies the following properties: it can store at most one value; initially it is empty; if it is given an input when empty then it stores that value; if it is given an input when non-empty then it will non-deterministically either replace its current stored value with the given input or will retain its current stored value ignoring the input; and finally, when non-empty it can output its stored value and then become empty.

We introduce the *null* value for operations that either have no input or have no output and we introduce the named operations *input* and *output*.

$$
\begin{array}{|l}
null : Value \\
input, output : Name
\end{array}
$$

Our buffer is then defined as follows.

$$
\begin{aligned}
BuffND \;\widehat{=}\; (\mu\; CDT[\mathbb{P}\, Value, Name, Value] \bullet \\
&state = \{ins : \mathbb{P}(Value \setminus \{null\}) \mid \#ins \leq 1\} \;\wedge \\
&init = \{\emptyset\} \;\wedge \\
&\mathrm{dom}\, ops = \{input, output\} \;\wedge \\
&ops\; input = \\
&\quad \{v : Value \setminus \{null\} \bullet (\emptyset, v) \mapsto (\{v\}, null)\} \\
&\quad \cup \\
&\quad \{v_1, v_2 : Value \setminus \{null\} \bullet (\{v_1\}, v_2) \mapsto (\{v_1\}, null)\} \\
&\quad \cup \\
&\quad \{v_1, v_2 : Value \setminus \{null\} \bullet (\{v_1\}, v_2) \mapsto (\{v_2\}, null)\} \;\wedge \\
&ops\; output = \{v : Value \setminus \{null\} \bullet (\{v\}, null) \mapsto (\emptyset, v)\})
\end{aligned}
$$

□

5.2 Relational semantics

We give a formal semantics to our data types, describing them in terms of programs, or sequences of operations, and the visible effects, in terms of inputs and outputs, of those programs. As with the simple abstract data types introduced in Chapter 4 we do this by exposing the binary relations that underpin the operation schemas.

5.2.1 Wound data types

Rather than move directly to a semantics which becomes notationally very heavy we define a function *simplify* from our communicating data types (CDTs) to *wound data types* (WDTs) in which all inputs and outputs are *wound*[1] into the state: all inputs are collected at initialisation and all outputs are returned at finalisation. WDTs are similar in structure to the ADTs introduced in Chapter 4; the difference being that ADTs have no notion of input and output.

The wound state space is a triple. The first element of the triple is of type *State* and records the state of the corresponding communicating data type. The second and third elements are sequences of *Value*s and record respectively the inputs that have not yet been consumed and the outputs that have been given thus far.

$$
\begin{array}{l}
WDT[State, Name, Value] \\
\hline
state : \mathbb{P}\,(State \times \mathrm{seq}\ Value \times \mathrm{seq}\ Value) \\
init : \mathrm{seq}\ Value \leftrightarrow (State \times \mathrm{seq}\ Value \times \mathrm{seq}\ Value) \\
ops : Name \rightarrow\!\!\!\!\!\shortmid\;\; ((State \times \mathrm{seq}\ Value \times \mathrm{seq}\ Value) \leftrightarrow \\
\qquad\qquad\qquad\qquad (State \times \mathrm{seq}\ Value \times \mathrm{seq}\ Value)) \\
final : (State \times \mathrm{seq}\ Value \times \mathrm{seq}\ Value) \rightarrow\!\!\!\!\!\shortmid\;\; \mathrm{seq}\ Value \\
\hline
\emptyset \subset \mathrm{ran}\, init \subseteq state \ \wedge \\
ops \subseteq Name \rightarrow\!\!\!\!\!\shortmid\;\; (state \leftrightarrow state) \\
final = \{s : State;\ outs : \mathrm{seq}\ Value \mid (s, \langle\rangle, outs) \in state \bullet (s, \langle\rangle, outs) \mapsto outs\} \\
\hline
\end{array}
$$

The set of operations on the wound data type is the same as the set of operations on the corresponding communicating data type. Moreover, if for values *in* and *out* and for states a and a' from the state space of a given communicating data type, the maplet

$$(a, in) \ \mapsto \ (a', out)$$

lies in the relation describing an operation n on the given data type then for *any* sequences of values *ins* and *outs*, the maplet

$$(a, \langle in\rangle \frown ins, outs) \ \mapsto \ (a', ins, outs \frown \langle out\rangle)$$

lies in the relation describing the operation n on the corresponding wound data type. The first component of the triples has recorded the change in the state space. The second

[1] The term *wound* was first coined in [WD96].

element of the triples has recorded the consuming of input *in* and the last element of the triples has added the output *out* to the sequence of recorded outputs.

At initialisation a sequence of inputs to be consumed is given. This is then recorded as the second element of the wound state; the first element being any state a' from the set of initial states on the corresponding communicating data type and the last element being the empty sequence thereby recording that no outputs have yet been given. The initialisation operation is then a relation from sequences of inputs to the wound state space. The finalisation operation on a wound data type is a (partial functional) relation from the internal wound state space to sequences of values. It returns the third element of the wound state space, the recorded output values.

The first two constraints on a wound data type are equivalent to the constraints on a communicating data type. The first constraint, $\emptyset \subset \operatorname{ran} init \subseteq state$ is equivalent to the constraint concerning initialisation of communicating data types: there must be some initial state. Similarly the second constraint is equivalent to the second constraint of the corresponding communicating data type: operations are not defined outside the internal state space. The third constraint simply defines the finalisation operation: it is defined for the subset of the wound state space in which the second component is the empty sequence (all input values have been consumed) and it returns the third component (the observed outputs).

The function *simplify* takes a communicating data type and returns the corresponding wound data type. It is formally defined as follows:

$$
\begin{array}{l}
[State, Name, Value] \\
\hline
simplify : CDT[State, Name, Value] \nrightarrow WDT[State, Name, Value] \\
\hline
simplify = \\
\quad (\lambda\, CDT[State, Name, Value] \bullet \\
\qquad (\mu\, WDT'[State, Name, Value] \mid \\
\qquad\quad state' = state \times \operatorname{seq} Value \times \operatorname{seq} Value \\
\qquad\quad \land \\
\qquad\quad init' = \{\, is : \operatorname{seq} Value;\ st' : init \bullet is \mapsto (st', is, \langle\rangle) \,\} \\
\qquad\quad \land \\
\qquad\quad \forall\, n : \operatorname{dom} ops' \bullet \\
\qquad\qquad ops'\, n = \{\, is, os : \operatorname{seq} Value;\ i, o : Value;\ s_1, s_2 : state \mid \\
\qquad\qquad\qquad (s_1, i) \mapsto (s_2, o) \in ops\, n \bullet \\
\qquad\qquad\qquad\quad (s_1, \langle i \rangle ^\frown is, os) \mapsto (s_2, is, os ^\frown \langle o \rangle) \,\} \\
\qquad\quad \land \\
\qquad\quad final' = \{\, os : \operatorname{seq} Value;\ st : state \bullet (st, \langle\rangle, os) \mapsto os \,\}\,)\,)
\end{array}
$$

Example 5 A *wound* non-deterministic one place buffer.

In this example we illustrate how the function *simplify* can be applied to communicating data types to obtain the corresponding wound data types. Recalling the definition of *BuffND* as defined in Example 4, we see easily that the wound data type *WoundBuffND* obtained by applying the function *simplify* to *BuffND* is defined as follows:

$$
\begin{aligned}
&\mathit{WoundBuffND} \;\widehat{=}\\
&\quad (\mu\ \mathit{WDT}[\mathbb{P}\ \mathit{Value}, \mathit{Name}, \mathit{Value}] \bullet\\
&\qquad \mathit{state} = \{\mathit{ins} : \mathbb{P}(\mathit{Value} \setminus \{\mathit{null}\});\ \mathit{is}, \mathit{os} : \mathrm{seq}\ \mathit{Value} \mid \#\mathit{ins} \leq 1\} \;\wedge\\
&\qquad \mathit{init} = \{\mathit{is} : \mathrm{seq}\ \mathit{Value} \bullet (\emptyset, \mathit{is}, \langle\rangle)\} \;\wedge\\
&\qquad \mathrm{dom}\ \mathit{ops} = \{\mathit{input}, \mathit{output}\} \;\wedge\\
&\qquad \mathit{ops}\ \mathit{input} =\\
&\qquad\quad \{\mathit{is}, \mathit{os} : \mathrm{seq}\ \mathit{Value};\ v : \mathit{Value} \setminus \{\mathit{null}\} \bullet\\
&\qquad\qquad (\emptyset, \langle v\rangle \frown \mathit{is}, \mathit{os}) \mapsto (\{v\}, \mathit{is}, \mathit{os} \frown \langle \mathit{null}\rangle)\}\\
&\qquad\quad \cup\\
&\qquad\quad \{\mathit{is}, \mathit{os} : \mathrm{seq}\ \mathit{Value};\ v_1, v_2 : \mathit{Value} \setminus \{\mathit{null}\} \bullet\\
&\qquad\qquad (\{v_1\}, \langle v_2\rangle \frown \mathit{is}, \mathit{os}) \mapsto (\{v_1\}, \mathit{is}, \mathit{os} \frown \langle \mathit{null}\rangle)\}\\
&\qquad\quad \cup\\
&\qquad\quad \{\mathit{is}, \mathit{os} : \mathrm{seq}\ \mathit{Value};\ v_1, v_2 : \mathit{Value} \setminus \{\mathit{null}\} \bullet\\
&\qquad\qquad (\{v_1\}, \langle v_2\rangle \frown \mathit{is}, \mathit{os}) \mapsto (\{v_2\}, \mathit{is}, \mathit{os} \frown \langle \mathit{null}\rangle)\} \;\wedge\\
&\qquad \mathit{ops}\ \mathit{output} =\\
&\qquad\quad \{\mathit{is}, \mathit{os} : \mathrm{seq}\ \mathit{Value};\ v : \mathit{Value} \setminus \{\mathit{null}\} \bullet\\
&\qquad\qquad (\{v\}, \mathit{null} \frown \mathit{is}, \mathit{os}) \mapsto (\emptyset, \mathit{is}, \mathit{os} \frown \langle v\rangle)\} \;\wedge\\
&\qquad \mathit{final} = \{\mathit{os} : \mathrm{seq}\ \mathit{Value};\ a : \mathit{state} \bullet (a, \langle\rangle, \mathit{os}) \mapsto \mathit{os}\})
\end{aligned}
$$

□

As with our abstract data types of Chapter 4 we can adopt either a blocking or a non-blocking model for our communicating data types. When totalising the relations describing the operations on those data types it was sufficient to map elements called outside their preconditions onto the bottom element. However, if we were to do the same for our communicating data types we would be throwing away too much information: we need to preserve the information recording the outputs that have already been given since these are an integral part of the operations on the communicating data types. Instead, we map the state component of the wound state space triple onto the bottom element; we consume one of the inputs and retain the output information:

- the totalisation that gives rise to the blocking semantics requires the state component of every element (including the bottom element) outside the domain to be mapped only to the bottom element; an input must be consumed; output information must be preserved.

$$
\begin{aligned}
r^{\top} \;=\;& r\\
& \cup\\
& \{s : \mathit{State}_{\perp};\ \mathit{ins}, \mathit{outs} : \mathrm{seq}\ \mathit{Value};\ \mathit{in} : \mathit{Value} \mid\\
&\quad (s, \langle \mathit{in}\rangle \frown \mathit{ins}, \mathit{outs}) \notin \mathrm{dom}\ r \bullet\\
&\qquad (s, \langle \mathit{in}\rangle \frown \mathit{ins}, \mathit{outs}) \mapsto (\perp, \mathit{ins}, \mathit{outs})\}
\end{aligned}
$$

- the totalisation that gives rise to the non-blocking semantics requires the state component of every element (including the bottom element) outside the domain to be mapped to every element in the target state space including the bottom element. An input must be consumed and an additional output might or might not be given, reflecting the fact that the operation might or might not be blocked.

$$\begin{array}{rcl} r^{\top} & = & r \\ & & \cup \\ & & \{ s : State_{\perp};\ ins, outs : \operatorname{seq} Value;\ in : Value \mid \\ & & \quad (s, \langle in \rangle \frown ins, outs) \notin \operatorname{dom} r \bullet \\ & & \qquad (s, \langle in \rangle \frown ins, outs) \mapsto (\perp, ins, outs) \} \\ & & \cup \\ & & \{ s : State_{\perp};\ ins, outs : \operatorname{seq} Value;\ in, out : Value \mid \\ & & \quad (s, \langle in \rangle \frown ins, outs) \notin \operatorname{dom} r \bullet \\ & & \qquad (s, \langle in \rangle \frown ins, outs) \mapsto (\perp, ins, outs \frown \langle out \rangle) \} \end{array}$$

Once more we define a function *run* that totalises and composes sequences of named operations. For clarity and brevity we will use the following abbreviations. We define *WState* to be the wound state space and overload the operator $_{\perp}$ by defining $WState_{\perp}$ to be the augmented wound state space.

$$\begin{array}{rcl} WState & == & State \times \operatorname{seq} Value \times \operatorname{seq} Value \\ WState_{\perp} & == & State_{\perp} \times \operatorname{seq} Value \times \operatorname{seq} Value \end{array}$$

The function *run* takes a program *prog* of operations and a function *ops* describing the effects of each operation. It then returns the relation obtained by composing the *totalised* versions of the relations corresponding to the operations in the program.

$$\begin{array}{|l} run_{WDT} : \operatorname{seq} Name \ \rightarrow\!\!\!\!\!+ \ (Name \ \rightarrow\!\!\!\!\!+ \ (WState \leftrightarrow WState)) \ \rightarrow\!\!\!\!\!+ \\ \qquad\qquad\qquad\qquad\qquad\qquad (WState_{\perp} \leftrightarrow WState_{\perp}) \\ \hline \forall\, n : Name;\ ns : \operatorname{seq} Name;\ ops : Name \ \rightarrow\!\!\!\!\!+ \ (WState \leftrightarrow WState) \bullet \\ \quad run_{WDT}\ \langle\rangle\ ops = \operatorname{id}[\, WState_{\perp} \,] \\ \quad \wedge \\ \quad run_{WDT}\ (ns \frown \langle n \rangle)\ ops = (run_{WDT}\ ns\ ops) \fatsemi (ops\ n)^{\top} \end{array}$$

To obtain the relational semantics for a simple data type we have only to bracket the result with initialisation and finalisation. As with the relational semantics for ADTs, we may use tagging to indicate whether the semantic model defining the refinement ordering adopts a blocking or a non-blocking approach.

$$\begin{array}{|l} [State, Name, Value] \\ \hline \mathcal{R}_{WDT} : WDT[State, Name, Value] \rightarrow \\ \qquad\qquad \operatorname{seq} Name \ \rightarrow\!\!\!\!\!+ \ (\operatorname{seq} Value \leftrightarrow \operatorname{seq} Value) \\ \hline \mathcal{R}_{WDT} \ = \ (\lambda\, WDT[State, Name, Value] \bullet \\ \qquad\qquad (\lambda\, prog : \operatorname{seq}(\operatorname{dom} ops) \bullet \\ \qquad\qquad\quad init \fatsemi run_{WDT}\ prog\ ops \fatsemi final^{\top})) \\ \hline \end{array}$$

Note that we have overloaded the $\top$ operator. Our "totalisation" of the finalisation simply extends the operation to the augmented state space. We still restrict it to those states whose inputs have all been consumed.

$$final^{\top} \;=\; final \cup \{outs : \mathrm{seq}\, Value \bullet (\perp, \langle\rangle, outs)\}$$

The relational semantics for communicating data types is then given by the composition of *simplify* and $\mathcal{R}_{WDT}$.

$$\begin{array}{l}
[State, Name, Value] \\
\hline
\mathcal{R}_{CDT} : CDT[State, Name, Value] \rightarrow \\
\qquad\qquad (\mathrm{seq}\, Name \rightarrow\!\!\!\!\!\!+ \; (\mathrm{seq}\, Value \times \mathrm{seq}\, Value)) \\
\hline
\mathcal{R}_{CDT} \;=\; simplify \,\fatsemi\, \mathcal{R}_{WDT}
\end{array}$$

5.3 Refinement and equivalence

We use the relational semantics presented in Section 5.2 to define a refinement ordering on our data types. As with our ADTs, refinement is reverse containment. We say that communicating data type A is data refined by communicating data type C if, for each sequence of named operations, the effect upon C is at least as deterministic as the effect upon A. For all such sequences *prog*, the relation $\mathcal{R}_{CDT}\, C\, prog$ must be a subset of the relation $\mathcal{R}_{CDT}\, A\, prog$. When this is the case, we write $A \sqsubseteq_{\mathcal{R}_{CDT}} C$. Formally,

$$\begin{array}{l}
[AState, CState, Name, Value] \\
\hline
_ \sqsubseteq_{\mathcal{R}_{CDT}} _ : CDT[AState, Name, Value] \leftrightarrow CDT[CState, Name, Value] \\
\hline
\forall\, CDT[AState, Name, Value];\; CDT'[CState, Name, Value] \bullet \\
\qquad \theta CDT \sqsubseteq_{\mathcal{R}_{CDT}} \theta CDT' \\
\qquad\qquad \Leftrightarrow \\
\qquad\qquad\quad \forall\, prog : \mathrm{seq}\, Name \bullet \\
\qquad\qquad\qquad \mathcal{R}_{CDT}[\theta CDT']\, prog \;\subseteq\; \mathcal{R}_{CDT}[\theta CDT]\, prog
\end{array}$$

Note that in order to verify that one communicating data type is refined by another the two data types must have the same *interface*, that is, the same combination of *Name* and *Value* that defines the signatures of the operations. They can however have different internal state spaces.

We say that two data types are equivalent if each refines the other.

$$
\begin{array}{|l}
\hline
[S, T, Name, Value] \\
\_ \equiv_{\mathcal{R}_{CDT}} \_ : CDT[S, Name, Value] \leftrightarrow CDT[T, Name, Value] \\
\hline
\forall\, CDT[S, Name, Value];\ CDT'[T, Name, Value] \bullet \\
\quad \theta CDT \equiv_{\mathcal{R}_{CDT}} \theta CDT' \\
\qquad \Leftrightarrow \\
\qquad\quad (\theta CDT \sqsubseteq_{\mathcal{R}_{CDT}} \theta CDT') \wedge (\theta CDT' \sqsubseteq_{\mathcal{R}_{CDT}} \theta CDT) \\
\hline
\end{array}
$$

5.4 Simulation of wound data types

In Chapter 4 we considered ADTs, abstract data types of the form

$$
\begin{array}{|l}
\hline
ADT[State, Name, Env] \\
state : \mathbb{P}\, State \\
init : Env \leftrightarrow State \\
ops : Name \nrightarrow (State \leftrightarrow State) \\
final : State \leftrightarrow Env \\
\hline
\operatorname{dom} init = Env \\
\operatorname{dom} final = state \\
\operatorname{ran} init \subseteq state \\
ops \subseteq Name \nrightarrow (state \leftrightarrow state) \\
\hline
\end{array}
$$

and claimed that the environment Env was "a convenient fiction serving only as a source for initial input and a target for final output". And indeed we see now that there is only a very small step between ADTs and WDTs, the wound version of our communicating data types; their structures are almost identical. The environment Env on an ADT is equivalent to the set $\mathbb{P}\operatorname{seq} Value$ on a wound data type. And the state space of an ADT is equivalent to the triple $State \times \operatorname{seq} Value \times \operatorname{seq} Value$ on a wound data type where $State$ is the state space of the corresponding communicating data type.

In addition we observe that the constraints on simple ADTs are consistent with the constraints on wound data types corresponding to a communicating data type. First we note that since the set *init* on a given communicating data type must be non-empty, by definition of *simplify*, the relation *init* on the corresponding wound data type is defined for all possible sequences of input values; this is consistent with the constraint $\operatorname{dom} init = Env$ on our ADTs. Furthermore the constraint $\operatorname{ran} init \subseteq state$ on our ADTs is consistent with the constraint on communicating data types that the set of initial states must be a subset of the internal state space. Similarly the constraint $ops \subseteq Name \nrightarrow (state \leftrightarrow state)$ on our ADTs is equivalent to the constraint $ops \subseteq Name \nrightarrow ((state \times Value) \leftrightarrow (state \times Value))$ on our communicating data types.

The constraint on the finalisation relation of simple abstract data types requires more attention. It states that finalisation must be available at any time. This is not the case with our wound data types where you can only finalise when the sequence of remaining inputs is empty. In fact this is not a problem: the reason we put the constraints on our ADTs was so that we could access the information about the inputs given thus far at any time; of course this is automatically the case with communicating data types as the outputs are given as they occur, and for the wound data types we could access that same information simply by considering prefixes of the given program and given inputs.

By the above observations we see that the definition of wound data types that correspond to communicating data types is consistent with the definition of simple data types, ADTs, presented in Chapter 4. It follows then that a set of sound and complete simulation rules for WDTs can be obtained by simply adapting the simulation rules for ADTs. In fact, the rules are identical: it is the types of the operations that are different.

5.5 Simulation of communicating data types

Having identified sets of simulation rules for wound data types, we now present sets of simulation rules for communicating data types. We prove that these too are both sound and jointly complete.

Data types with the same state space

In the non-blocking semantics, data type θCDT of type $CDT[State, Name, Value]$ simulates data type $\theta CDT'$ also of type $CDT[State, Name, Value]$ if the following rules hold.

$$init' \subseteq init$$

$$\forall\, n : \operatorname{dom} ops \bullet \operatorname{dom}(ops\ n) \subseteq \operatorname{dom}(ops'\ n)$$

$$\forall\, n : \operatorname{dom} ops \bullet \operatorname{dom}(ops\ n) \lhd ops'\ n \subseteq ops\ n$$

These rules appear identical to the first three simulation rules in the non-blocking semantics for ADTs with the same state space. The difference lies in the types: in the case of ADTs, $init$ is a relation from Env to $State$ whereas for CDTs, $init$ is a set containing elements of type $State$; for any name n, in the case of ADTs, $ops\ n$ is a relation from $State$ to $State$ whereas for CDTs, $ops\ n$ is a relation from $State \times Value$ to $State \times Value$. The rule concerning finalisation is trivially true and so disappears. This corresponds to our claim that the environment is a convenient fiction for providing inputs and receiving outputs.

Similarly, in the blocking semantics the simulation rules for data types with the same state space appear identical to the first three corresponding rules for ADTs: θCDT simulates $\theta CDT'$ if the following rules hold.

$$init' \subseteq init$$

$$\forall\, n : \operatorname{dom} ops \bullet \operatorname{dom}(ops\ n) = \operatorname{dom}(ops'\ n)$$

$$\forall\, n : \operatorname{dom} ops \bullet ops'\ n \subseteq ops\ n$$

Forward simulation rules

For clarity, when unwinding the forwards and backwards simulation rules for our SDTs we use the following parallel operator, as defined in [WD96], on our relations:

$$
\begin{array}{l}
[W, X, Y, Z] \\
\hline
\_ \parallel \_ : (W \leftrightarrow Y) \times (X \leftrightarrow Z) \to (W \times X \leftrightarrow Y \times Z) \\
\hline
\forall r : W \leftrightarrow Y;\ s : X \leftrightarrow Z;\ w : W;\ x : X;\ y : Y;\ z : Z \bullet \\
\quad (w, x) \mapsto (y, z) \in r \parallel s \Leftrightarrow (w \mapsto y \in r \wedge x \mapsto z \in s)
\end{array}
$$

Given this definition, data type θCDT of type $CDT[AState, Name, Value]$ forwards simulates data type $\theta CDT'$ of type $CDT[CState, Name, Value]$ within the non-blocking semantics if we can identify a retrieve relation r from $AState$ to $CState$ such that:

$$init' \subseteq r(\!|\ init\ |\!)$$

$$\forall n : \mathrm{dom}\ ops \bullet (r \parallel \mathrm{id}[Value])(\!|\ \mathrm{dom}\,(ops\ n)\ |\!) \subseteq \mathrm{dom}\,(ops'\ n)$$

$$\begin{array}{l}\forall n : \mathrm{dom}\ ops \bullet \\ \quad \mathrm{dom}\,(ops\ n) \lhd (r \parallel \mathrm{id}[Value]) \mathbin{;} ops'\ n \subseteq ops\ n \mathbin{;} (r \parallel \mathrm{id}[Value])\end{array}$$

$$r(\!|\ AState \setminus state\ |\!) \subseteq (\ CState \setminus state'\ )$$

Similarly, θCDT forwards simulates $\theta CDT'$ within the blocking semantics if we can identify retrieve relation r from $AState$ to $CState$ such that:

$$init' \subseteq r(\!|\ init\ |\!)$$

$$\forall n : \mathrm{dom}\ ops' \bullet (\ r \parallel \mathrm{id}[Value]\ ) \mathbin{;} (ops'\ n) \subseteq (ops\ n) \mathbin{;} (\ r \parallel \mathrm{id}[Value]\ )$$

$$\forall n : \mathrm{dom}\ ops' \bullet (\ r \parallel \mathrm{id}[Value]\ )(\!|\ \mathrm{dom}(ops\ n)\ |\!) \subseteq \mathrm{dom}(ops'\ n)$$

$$r(\!|\ AState \setminus state\ |\!) \subseteq (\ CState \setminus state'\ )$$

Within the context of both the blocking and the non-blocking semantic models we can see easily the correlation between the first three forwards simulation rules for CDTs and the first three forwards simulation rules for WDTs (which are syntactically identical to the simulation rules for ADTs as presented in Section 4.4). The correlation between the fourth simulation rules is not, however, immediately obvious. The restrictions on the state space for communicating data types stem from the restrictions on the finalisation relation for wound and simple abstract data types. We will demonstrate this in Theorem 13.

Backward simulation rules

In the non-blocking semantics, data type θCDT of type $CDT[AState, Name, Value]$ backwards simulates data type $\theta CDT'$ of type $CDT[CState, Name, Value]$ if we can identify a retrieve relation s from *CState* to *AState* such that:

$$s(\!|\; init' \;|\!) \subseteq init$$

$$\forall\, n : \operatorname{dom} ops \bullet state' \operatorname{dom} ops'\, n \subseteq \operatorname{dom}((s \parallel \mathrm{id}[Value]) \rhd (\operatorname{dom} ops\, n))$$

$$\begin{aligned}&\forall\, n : \operatorname{dom} ops \bullet \\ &\quad \operatorname{dom}((s \parallel \mathrm{id}[Value]) \rhd (\operatorname{dom} ops\, n)) \lhd (ops'\, n) \fatsemi (s \parallel \mathrm{id}[Value]) \\ &\qquad \subseteq \\ &\qquad\quad (s \parallel \mathrm{id}[Value]) \fatsemi (ops\, n)\end{aligned}$$

$$state' \subseteq \operatorname{dom}(\, s \rhd state\,)$$

Similarly, in the blocking semantics, θCDT backwards simulates $\theta CDT'$ if we can identify retrieve relation s from *CState* to *AState* such that:

$$s\;(\!|\; init' \;|\!) \;\subseteq\; init$$

$$\forall\, n : \operatorname{dom} ops' \bullet (ops'\, n) \fatsemi (\, s \parallel \mathrm{id}[Value]\,) \;\subseteq\; (\, s \parallel \mathrm{id}[Value]\,) \fatsemi (ops\, n)$$

$$\begin{aligned}&\forall\, n : \operatorname{dom} ops' \bullet \\ &\quad (\, state' \times Value\,) \setminus (\operatorname{dom} ops'\, n\,) \;\subseteq\; \operatorname{dom}(\,(s \parallel \mathrm{id}) \rhd \operatorname{dom} ops\, n\,)\end{aligned}$$

$$state' \;\subseteq\; \operatorname{dom}(\, s \rhd state\,)$$

As with the forwards simulation rule, for both the blocking and the non-blocking semantic models, the constraints on the state space imposed by the final rule arise from the constraints imposed on the finalisation relation by the corresponding rules on the wound data types.

Theorem 13 Equivalence of the simulation rules.

Within both the blocking and the non-blocking semantic models, the simulation rules for communicating data types are equivalent to the simulation rules for simple abstract data types as presented in Chapter 4.

Proof This result follows from algebraic manipulation of the simulation rules for wound data types which we have shown to be identical to the simulation rules for simple abstract data types. To be precise, we *unwind*, or express in terms only of the components of a *CDT*, the simulation rules for wound data types and hence obtain sets of simulation rules for our communicating data types. We present, by way of example, three proofs of equivalence.

First we show that, within the context of the non-blocking semantic model, the forward simulation rule concerning initialisation of *CDT*s is equivalent to the forward simulation rule concerning initialisation of *WDT*s. Consider data types θCDT and $\theta CDT'$ of types $CDT[AState, Name, Value]$ and $CDT[CState, Name, Value]$ respectively, and consider their corresponding wound data types θWDT and $\theta WDT'$ obtained by applying the function *simplify*. Let r be a retrieve relation of the type $AState \leftrightarrow CState$. Defining the corresponding retrieve relation R to be $(r \parallel \mathrm{id}[Value] \parallel \mathrm{id}[Value])$, and tagging our initialisation operations for clarity, we reason as follows:

$$init_{WDT}{}' \subseteq init_{WDT} \mathbin{⨾} R$$

$$\Leftrightarrow \qquad [\text{def}^n\text{s of } \mathit{simplify} \text{ and } R]$$

$$\{ is : \text{seq } Value;\ c' : init_{CDT}{}' \bullet is \mapsto (c', is, \langle\rangle) \} \subseteq \{ is : \text{seq } Value;\ a' : init_{CDT} \bullet is \mapsto (a', is, \langle\rangle) \} \mathbin{⨾} (r \parallel \text{id} \parallel \text{id})$$

$$\Leftrightarrow \qquad [\text{def}^n\text{s of } ⨾,\ \parallel \text{ and id}]$$

$$\{ is : \text{seq } Value;\ c' : init_{CDT}{}' \bullet is \mapsto (c', is, \langle\rangle) \} \subseteq \{ is : \text{seq } Value;\ c' : init_{CDT}{}' \mid (\exists\, a' : init_{CDT} \bullet c' \in r(\!|\, \{a'\} \,|\!)) \bullet is \mapsto (c', is, \langle\rangle) \}$$

$$\Leftrightarrow \qquad [\text{property of } (\!|\,|\!)]$$

$$\{ is : \text{seq } Value;\ c' : init_{CDT}{}' \bullet is \mapsto (c', is, \langle\rangle) \} \subseteq \{ is : \text{seq } Value;\ c' : r(\!|\, init_{CDT} \,|\!) \bullet is \mapsto (c', is, \langle\rangle) \}$$

$$\Leftrightarrow\ init_{CDT}{}' \subseteq r\,(\!|\, init_{CDT} \,|\!) \qquad [\text{set theory}]$$

To verify the equivalence of the backward simulation rules concerning the availability of operations we consider communicating data types θCDT and $\theta CDT'$ and corresponding wound data types θWDT and $\theta WDT'$ as defined above. Given retrieve relation s of type $CState \leftrightarrow AState$, defining corresponding retrieve relation S to be $(s \parallel \text{id}[Value] \parallel \text{id}[Value])$ and given any named operation n, we reason as follows:

$$state_{WDT}{}' \setminus \text{dom } ops_{WDT}{}'\, n \subseteq \text{dom}\,(S \rhd (\text{dom } ops_{WDT}\, n))$$

$$\Leftrightarrow \qquad [\text{set theory and def}^n \text{ of } \rhd]$$

$$\forall\, c : CState;\ ins, outs : \text{seq } Value \bullet (c, ins, outs) \in state_{WDT}{}' \land (c, ins, outs) \notin \text{dom } ops_{WDT}{}'\, n \Rightarrow \exists\, a : AState;\ ins', out' : \text{seq } Value \bullet (c, ins, outs) \mapsto (a, ins', outs') \in S \land (a, ins', outs') \notin \text{dom } ops_{WDT}\, n$$

$$\Leftrightarrow \qquad [\text{def}^n\text{s of } \mathit{simplify},\ S, \text{ and } \mathit{head}]$$

$$\forall\, c : CState;\ ins : \text{seq } Value \mid ins \neq \langle\rangle \bullet c \in state_{CDT}{}' \land (c, head\ ins) \notin \text{dom } ops_{CDT}{}'\, n \Rightarrow \exists\, a : AState \bullet c \mapsto a \in s \land (a, head\ ins) \notin \text{dom } ops_{CDT}\, n$$

$$\Leftrightarrow \qquad \text{[one point rule and def}^n\text{s of } \parallel \text{ and id]}$$
$$\forall\, c : CState;\ in : Value \bullet$$
$$c \in state_{CDT}{}' \land (c, in) \notin \operatorname{dom} ops_{CDT}{}'\, n$$
$$\Rightarrow$$
$$\exists\, a : AState \bullet (c, in) \mapsto (a, in) \in (s \parallel \mathrm{id}) \land (a, in) \notin \operatorname{dom} ops_{CDT}\, n$$

$$\Leftrightarrow \qquad \text{[set theory and } \mathit{defn} \text{ of } \rhd]$$
$$\forall\, c : state_{CDT}{}';\ in : Value \bullet$$
$$(c, in) \notin \operatorname{dom} ops_{CDT}{}'\, n \Rightarrow\ (c, in) \in \operatorname{dom}((s \parallel \mathrm{id}) \rhd \operatorname{dom} ops_{CDT}\, n)$$

$$\Leftrightarrow \qquad \text{[set theory]}$$
$$(state_{CDT}{}' \times\ Value) \setminus \operatorname{dom} ops_{CDT}{}'\, n\ \subseteq\ \operatorname{dom}((s \parallel \mathrm{id}) \rhd \operatorname{dom} ops_{CDT}\, n)$$

Finally, we verify the equivalence of the fourth forward simulation rules. The version of this rule for WDTs is expressed in terms of the finalisation relation whereas for CDTs it is expressed in terms of the applicability of the retrieve relation. In our proof we consider retrieve relations r and R, and communicating data types θCDT and $\theta CDT'$ with corresponding wound data types θWDT and $\theta WDT'$ all as defined above. We reason as follows:

$$R \fatsemi final_{WDT}{}' \subseteq final_{WDT}$$

$$\Leftrightarrow \qquad \text{[set theory and def}^n \text{ of } \fatsemi]$$
$$\forall\, a : AState;\ ins, outs, outs' : \operatorname{seq} Value \bullet$$
$$(\,\exists\, c : CState;\ is, os : \operatorname{seq} Value \bullet$$
$$(a, ins, outs) \mapsto (c, is, os) \in R \land (c, is, os) \mapsto outs' \in final_{WDT}{}'\,)$$
$$\Rightarrow$$
$$(a, ins, outs) \mapsto outs' \in final_{WDT}$$

$$\Leftrightarrow \qquad \text{[one point rule and def}^n\text{s of } \mathit{final} \text{ and } R]$$
$$\forall\, a : AState;\ ins, outs, outs' : \operatorname{seq} Value \bullet$$
$$ins = \langle\rangle \land outs = outs' \land \exists\, c : CState \mid c \in state_{CDT}{}' \bullet a \mapsto c \in r$$
$$\Rightarrow$$
$$ins = \langle\rangle \land outs' = outs \land a \in state_{CDT}$$

$$\Leftrightarrow \qquad \text{[property of } \forall]$$
$$\forall\, a : AState\ \bullet\ (\,\exists\, c : CState \mid c \in state_{CDT}{}' \bullet a \mapsto c \in r\,)\ \Rightarrow\ (\,a \in state_{CDT}\,)$$

$$\Leftrightarrow \qquad \text{[property of } \Rightarrow \text{ and } \forall]$$
$$\forall\, a : AState \setminus state_{CDT}\ \bullet\ \neg\,(\,\exists\, c : CState \mid c \in state_{CDT}{}' \bullet a \mapsto c \in r\,)$$

$$\Leftrightarrow \qquad \text{[property of } \exists]$$
$$\forall\, a : AState \setminus state_{CDT}\ \bullet\ (\,\forall\, c : state_{CDT}{}' \bullet a \mapsto c \notin r\,)$$

$\Leftrightarrow$ [set theory and property of $(\!|\,|\!)$]

$$r(\!|\ AState \setminus state_{CDT}\ |\!) \subseteq CState \setminus state_{CDT}{}'$$

□

Theorem 14 Soundness and joint completeness of the forwards and backwards simulation rules for communicating data types.

Proof This result follows directly from Theorem 13 above and from Theorems 10 and 11 which established the soundness and joint completeness of the simulation rules for *ADT*s. □

Example 6 shows how the simulation rules presented above might be used to verify that one communicating data type refines another.

Example 6 Applying the simulation rules.

In this example we model a *deterministic* one place buffer as a communicating data type. We then use the blocking version of the backwards simulation rules for communicating data types to show that it refines the communicating data type *BuffND* presented in Example 4. For illustrative purposes we define our deterministic buffer to have a different state space to that of our non-deterministic buffer, thereby enabling us to see an example of simulation with a retrieve relation.

Our deterministic one place buffer, *BuffD*, satisfies the following properties: it can store at most one value; initially it is empty; if it is given an input when it is empty then it stores that value; if it is given an input when it is non-empty it will replace its current stored value with the given input; it can output its stored value when it is non-empty and then it becomes empty. It is formally defined as follows:

$$\begin{array}{lll} BuffD & \widehat{=} & (\,\mu\, CDT'[\,\mathrm{seq}\ Value, Name, Value\,] \bullet \\ & & \quad state' \ =\ \{ins : \mathrm{seq}(Value \setminus \{null\}) \mid \#ins \leq 1\} \ \wedge \\ & & \quad init' \ =\ \langle\rangle \ \wedge \\ & & \quad \mathrm{dom}\, ops' \ =\ \{input, output\} \ \wedge \\ & & \quad ops'\, input \ = \\ & & \qquad \{v : Value \setminus \{null\} \bullet (\langle\rangle, v) \mapsto (\langle v\rangle, null)\} \cup \\ & & \qquad \{v_1, v_2 : Value \setminus \{null\} \bullet (\langle v_1\rangle, v_2) \mapsto (\langle v_2\rangle, null)\} \ \wedge \\ & & \quad ops'\, output \ =\ \{\, v : Value \setminus \{null\} \bullet (\langle v\rangle, null) \mapsto (\langle\rangle, v)\,\}\,) \end{array}$$

We will establish that the relational versions of the backwards simulation rules hold by applying the retrieve relation $s = \mathrm{ran}$ to elements in the state space of *BuffD* to get the corresponding elements in the state space of *BuffND*.

Reasoning as follows we see that the first backwards simulation rule holds.

$$s\ (\!|\ init'\ |\!) \subseteq init$$

$\Leftrightarrow \mathrm{ran}\langle\rangle \subseteq \emptyset$ [substitution]

$\Leftrightarrow$ true [property of ran]

Next consider the second backwards simulation rule.

$$\forall n : \mathrm{dom}\, ops' \bullet (ops'\, n) ⨟ (s \parallel \mathrm{id}[Value]) \subseteq (s \parallel \mathrm{id}[Value]) ⨟ (ops\, n)$$

$$\Leftrightarrow \qquad \text{[substitution]}$$
$$(ops'\, input) ⨟ (\mathrm{ran} \parallel \mathrm{id}) \subseteq (\mathrm{ran} \parallel \mathrm{id}) ⨟ (ops\, input)$$
$$\wedge$$
$$(ops'\, output) ⨟ (\mathrm{ran} \parallel \mathrm{id}) \subseteq (\mathrm{ran} \parallel \mathrm{id}) ⨟ (ops\, output)$$

Recalling that the set *ops output* is equal to

$$\{ v : Value \setminus \{null\} \bullet (\{v\}, null) \mapsto (\emptyset, v) \}$$

and considering the effect of the function (ran $\parallel$ id) we observe immediately that the sets $(ops'\, output) ⨟ (\mathrm{ran} \parallel \mathrm{id})$ and $(\mathrm{ran} \parallel \mathrm{id}) ⨟ (ops\, output)$ are equivalent since they are both equal to the set

$$\{ v : Value \setminus \{null\} \bullet (\langle v \rangle, null) \mapsto (\emptyset, v) \}.$$

Hence the second clause of the predicate is trivially true.

Next, recalling the definition of *ops input* and applying the distributivity of relational composition over set union we see that the set $(\mathrm{ran} \parallel \mathrm{id}) ⨟ (ops\, input)$ is equivalent to the following set union.

$$\{v : Value \setminus \{null\} \bullet (\langle\rangle, v) \mapsto (\{v\}, null)\}$$
$$\cup$$
$$\{v_1, v_2 : Value \setminus \{null\} \bullet (\langle v_1 \rangle, v_2) \mapsto (\{v_1\}, null)\}$$
$$\cup$$
$$\{v_1, v_2 : Value \setminus \{null\} \bullet (\langle v_1 \rangle, v_2) \mapsto (\{v_2\}, null)\}$$

Similarly, applying the distributivity of relational composition over set union, we see that $(ops'\, input) ⨟ (\mathrm{ran} \parallel \mathrm{id})$ is equivalent to the set union given below.

$$\{v : Value \setminus \{null\} \bullet (\langle\rangle, v) \mapsto (\{v\}, null)\}$$
$$\cup$$
$$\{v_1, v_2 : Value \setminus \{null\} \bullet (\langle v_1 \rangle, v_2) \mapsto (\{v_2\}, null)\}$$

The two components of the above set union describing the relation $(ops'\, input) ⨟ (\mathrm{ran} \parallel \mathrm{id})$ are equal to the first and third components of the set union describing the relation $(\mathrm{ran} \parallel \mathrm{id}) ⨟ (ops\, input)$. From this we conclude that the following predicate is true

$$(ops'\, input) ⨟ (\mathrm{ran} \parallel \mathrm{id}) \subseteq (\mathrm{ran} \parallel \mathrm{id}) ⨟ (ops\, input)$$

and hence that the second backwards simulation rule holds.

Next consider the third simulation rule.

$$\forall\, n : \mathrm{dom}\, ops' \;\bullet\; (\, state' \times Value\,) \setminus (\,\mathrm{dom}\, ops'\, n\,) \subseteq \mathrm{dom}\,(\,(s \parallel \mathrm{id}) \rhd \mathrm{dom}\, ops\, n\,)$$

$$\Leftrightarrow \qquad [\text{substitution}]$$

$$(\,\{cc : \mathrm{seq}\, Value \mid \#cc \leq 1\} \times Value\,) \setminus (\,\mathrm{dom}\, ops'\, input\,) \subseteq \mathrm{dom}\,(\,(\mathrm{ran} \parallel \mathrm{id}) \rhd \mathrm{dom}\, ops\, input\,)$$
$$\wedge$$
$$(\,\{cc : \mathrm{seq}\, Value \mid \#cc \leq 1\} \times Value\,) \setminus (\,\mathrm{dom}\, ops'\, output\,) \subseteq \mathrm{dom}\,(\,(\mathrm{ran} \parallel \mathrm{id}) \rhd \mathrm{dom}\, ops\, output\,)$$

Since $\mathrm{dom}\, ops'\, input$ is equal to

$$\{v : Value \setminus \{null\} \bullet (\langle\rangle, v)\} \cup \{v_1, v_2 : Value \setminus \{null\} \bullet (\langle v_1 \rangle, v_2)\}$$

and since $\mathrm{dom}\, ops'\, output$ is equal to $\{\, v : Value \setminus \{null\} \bullet (\langle v \rangle, null)\}$ we see immediately that the third simulation rules reduces to the following predicate.

$$\{(\langle\rangle, null)\} \cup \{v : Value \bullet (\langle v \rangle, null)\} \cup \{v : Value \bullet (\langle null \rangle, v)\} \subseteq \mathrm{dom}\,(\,(\mathrm{ran} \parallel \mathrm{id}) \rhd \mathrm{dom}\, ops\, input\,)$$
$$\wedge$$
$$\{(\langle null \rangle, null)\} \cup \{v : Value \bullet (\langle\rangle, v)\} \subseteq \mathrm{dom}\,(\,(\mathrm{ran} \parallel \mathrm{id}) \rhd \mathrm{dom}\, ops\, output\,)$$

Furthermore, since $\mathrm{dom}\, ops\, input$ is equal to

$$\{v : Value \setminus \{null\} \bullet (\emptyset, v)\} \cup \{v_1, v_2 : Value \setminus \{null\} \bullet (\langle v_1 \rangle, v_2)\}$$

and since $\mathrm{dom}\, ops\, output$ is equal to $\{\, v : Value \setminus \{null\} \bullet (\{v\}, null)\}$, by considering the injective function $(\mathrm{ran} \parallel \mathrm{id})$ we see that the sets $\mathrm{dom}\,(\,(\mathrm{ran} \parallel \mathrm{id}) \rhd \mathrm{dom}\, ops\, input\,)$ and $\mathrm{dom}\,(\,(\mathrm{ran} \parallel \mathrm{id}) \rhd \mathrm{dom}\, ops\, output\,)$ are given respectively by the following set unions:

$$\{(\langle\rangle, null)\} \cup \{v : Value \bullet (\langle v \rangle, null)\} \cup \{v : Value \bullet (\langle null \rangle, v)\}$$
and
$$\{(\langle null \rangle, null)\} \cup \{v : Value \bullet (\langle\rangle, v)\}.$$

Substituting back in we conclude that backwards simulation rule three is equivalent to

$$\begin{array}{l}\{(\langle\rangle, null)\} \cup \{v : Value \bullet (\langle v\rangle, null)\} \cup \{v : Value \bullet (\langle null\rangle, v)\}\\ \quad\subseteq\\ \qquad\{(\langle\rangle, null)\} \cup \{v : Value \bullet (\langle v\rangle, null)\} \cup \{v : Value \bullet (\langle null\rangle, v)\}\\ \wedge\\ \{(\langle null\rangle, null)\} \cup \{v : Value \bullet (\langle\rangle, v)\}\\ \quad\subseteq\\ \qquad\{(\langle null\rangle, null)\} \cup \{v : Value \bullet (\langle\rangle, v)\}\end{array}$$

and hence is trivially true. Finally consider the fourth backwards simulation rule.

$$state' \subseteq \operatorname{dom}(s \rhd state)$$

$$\begin{array}{ll}\Leftrightarrow \{ins : \operatorname{seq}(Value \setminus \{null\}) \mid \#ins \leq 1\} & \text{[substitution]}\\ \qquad\subseteq & \\ \qquad\quad \operatorname{dom}(\operatorname{ran} \rhd \{ins : \mathbb{P}(Value \setminus \{null\}) \mid \#ins \leq 1\}) & \\ & \\ \Leftrightarrow \{ins : \operatorname{seq}(Value \setminus \{null\}) \mid \#ins \leq 1\} & \text{[property of ran]}\\ \qquad\subseteq & \\ \qquad\quad \{ins : \operatorname{seq}(Value \setminus \{null\}) \mid \#ins \leq 1\} & \\ & \\ \Leftrightarrow \text{true} & \text{[set theory]}\end{array}$$

We have shown that all four backwards simulation rules are true and therefore conclude that communicating data type *BuffND* simulates communicating data type *BuffD*. It then follows immediately from Theorem 14 that *BuffND* is data refined by *BuffD*.

$$BuffND \quad \sqsubseteq_{\mathcal{R}_{CDT}} \quad BuffD$$

□

5.6 Corresponding CSP processes

In this section we define a translation from communicating data types expressed in Z to their process equivalents expressed in CSP. As we did with simple ADTs in Chapter 4 we then use this translation to explore the relationship between their respective refinement orderings. For both the blocking and the non-blocking semantic models, the structure of the body of the process corresponding to a CDT is very similar to the structure of the body of the process corresponding to an ADT, the main difference being connected with initialisation and finalisation. For CDTs no event is communicated at initialisation—there is no *Env* value to be communicated—simply an internal state is chosen. Similarly the final.e event is redundant. A further difference between ADTs and CDTs is that for CDTs an input value can be chosen (deterministically) when the named operation is chosen and an output value can be chosen (non-deterministically) when the subsequent internal state is chosen.

The function $process_{io\,b}$ which takes a communicating data type that has been given a blocking semantics and returns the corresponding CSP process is defined:

$$
\begin{array}{l}
process_{io\,b}\,(\,\theta CDT[State, Name, Value]\,) \;=\\
\quad \text{let}\\
\qquad Proc(a) \;=\\
\qquad\quad \Box\, name : \mathrm{dom}\, ops;\; in : Value \mid (a, in) \in \mathrm{dom}\, ops\; name \bullet\\
\qquad\qquad \sqcap\, a' : State;\; out : Value \mid\\
\qquad\qquad\qquad (a, in) \mapsto (a', out) \in ops\; name \bullet\\
\qquad\qquad\quad \mathrm{op}.name.in.out \rightarrow Proc(a')\\
\quad \text{within}\\
\qquad \sqcap\, a' : init \bullet Proc(a')
\end{array}
$$

We define similarly the function $process_{io\,n}$ which takes a communicating data type that has been given a non-blocking semantics and returns the corresponding CSP process. In this case, if an event is called from a state outside the domain of the corresponding operation then subsequent behaviour is undefined: the process non-deterministically can stop or can perform any non-deterministically chosen sequence of events of arbitrary length before stopping.

$$
\begin{array}{l}
process_{io\,n}\,(\,\theta CDT[State, Name, Value]\,) \;=\\
\quad \text{let}\\
\qquad Proc(a) \;=\\
\qquad\quad \Box\, name : \mathrm{dom}\, ops;\; in : Value \mid (a, in) \in \mathrm{dom}\, ops\; name \bullet\\
\qquad\qquad \sqcap\, a' : State;\; out : Value \mid (a, in) \mapsto (a', out) \in ops\; name \bullet\\
\qquad\qquad\quad \mathrm{op}.name.in.out \rightarrow Proc(a')\\
\qquad\quad \Box\\
\qquad\quad (\,\Box\, name : \mathrm{dom}\, ops;\; in : Value \mid (a, in) \notin \mathrm{dom}\, ops\; name \bullet\\
\qquad\qquad\quad \sqcap\, out : Value \bullet\\
\qquad\qquad\qquad \mathrm{op}.name.in.out \rightarrow chaos_{CDT}(\,\theta CDT[State, Name, Env]\,)\\
\qquad\qquad \sqcap\\
\qquad\qquad Stop\,)\\
\quad \text{within}\\
\qquad \sqcap\, a' : init \bullet Proc(a')
\end{array}
$$

The process $chaos_{CDT}$, like the process $chaos_{ADT}$, can accept or refuse any sequence of events corresponding to named operations.

$$
\begin{array}{l}
chaos_{CDT}\,(\,\theta CDT[State, Name, Env]\,) \;=\\
\quad \Box\, name : \mathrm{dom}\, ops;\; in, out : Value \bullet\\
\qquad \mathrm{op}.name.in.out \rightarrow chaos_{CDT}\,(\,\theta CDT[State, Name, Env]\,))\\
\quad \sqcap\\
\quad Stop
\end{array}
$$

5.7 Data refinement and process refinement

In this section we consider the relationship between data refinement and process refinement of communicating data types. In Theorem 12 of Chapter 4 we proved that *for simple abstract data types* data refinement is equivalent to singleton failures refinement; we will prove in Corollary 4 that this result extends to communicating data types with deterministic output values. However, as we will illustrate in Observation 9 below, the result does not hold in the general case: simply showing that the process corresponding to one communicating data type is refined within the singleton failures semantic model by the process corresponding to another is not enough to prove that the first data type is data refined by the second.

Observation 9 Singleton failures refinement does not imply data refinement for communicating data types with non-deterministic outputs.

Proof We establish this result in the context of the blocking semantic model but the same result is obtained within the context of the non-blocking semantic model. Consider data types θCDT and $\theta CDT'$ in which

$$\begin{array}{rclcl} \mathit{state} & = & \mathit{state}' & = & \{s, t\} \\ \mathrm{dom}\ \mathit{ops} & = & \mathrm{dom}\ \mathit{ops}' & = & \{n\} \\ \mathit{ops}\ n & = & \mathit{ops}\ n' & = & \{(s,0) \mapsto (t,0), (s,0) \mapsto (t,1)\}. \end{array}$$

They share the same state space with two elements s and t and they share a single operation n. The difference lies in their initialisations.

$$\mathit{init} \ = \ \{s\} \qquad\qquad \mathit{init}' \ = \ \{t\}$$

Data type θCDT is initially in a state from which operation n can be performed with input value 0, and either 0 or 1 will non-deterministically be given as output. Data type $\theta CDT'$ however is initially in a state from which no operation can be performed. We observe that the maplet $\langle n \rangle \mapsto (\langle 0 \rangle, \langle\rangle)$ lies in $\mathcal{R}_{CDT\,b}[\theta CDT']$ the blocking relational semantics of data type $\theta CDT'$ whereas this pair does not lie in $\mathcal{R}_{CDT\,b}[\theta CDT]$, the blocking relational semantics of data type θCDT. It follows then that θCDT is not refined by $\theta CDT'$.

Consider now the processes corresponding to communicating data types θCDT and $\theta CDT'$. Within the blocking semantics the process corresponding to θCDT is given by

$$\begin{array}{l} \mathit{process}_{io\,b}\,(\,\theta CDT\,) \ = \\ \quad \text{let} \\ \quad\quad \mathit{Proc}(a) \ = \ (a == s)\ \&\ \sqcap\ \mathit{out} : \{0,1\} \bullet \mathrm{op}.n.0.\mathit{out} \rightarrow \mathit{Proc}(t) \\ \quad \text{within} \\ \quad\quad \mathit{Proc}(s). \end{array}$$

This can equivalently be expressed as $(\mathrm{op}.n.0.0 \rightarrow \mathit{Stop} \sqcap \mathrm{op}.n.0.1 \rightarrow \mathit{Stop})$. The singleton failures of this process are as follows:

$$\begin{array}{l} \{(\langle\rangle, \emptyset), (\langle op.n.0.0\rangle, \emptyset), (\langle op.n.0.1\rangle, \emptyset)\} \\ \cup \\ \{in, out : \{0,1\} \bullet (\langle\rangle, \{op.n.in.out\})\} \\ \cup \\ \{in, out, out' : \{0,1\} \bullet (\langle op.n.0.out'\rangle, \{op.n.in.out\})\} \end{array}$$

The process corresponding to $\theta CDT'$ is given by

$$\begin{array}{l} process_{io\,b}\,(\,\theta CDT'\,) \;=\; \\ \quad \text{let} \\ \quad\quad Proc(a) \;=\; (a == s)\ \&\ \sqcap\, out : \{0,1\} \bullet \text{op}.n.0.out \rightarrow Proc(t) \\ \quad \text{within} \\ \quad\quad Proc(t) \end{array}$$

and can equivalently be expressed as $Stop$. The singleton failures of this process are given below.

$$\{(\langle\rangle, \emptyset)\} \;\cup\; \{in, out : \{0,1\} \bullet (\langle\rangle, \{op.n.in.out\})\}$$

We observe that the singleton failures of the process corresponding to data type $\theta CDT'$ are a subset of the singleton failures of the process corresponding to data type θCDT. Hence θCDT is refined by $\theta CDT'$ within the singleton failures semantic model. We have proved that the following predicate is true:

$$process_{io}\,(\theta CDT) \sqsubseteq_{\mathcal{S}} process_{io}\,(\theta CDT') \quad\wedge\quad \theta CDT \not\sqsubseteq_{\mathcal{R}_{CDT}} \theta CDT'.$$

Data type θCDT is not refined by data type $\theta CDT'$ but the process corresponding to θCDT is refined within the singleton failures semantic model by the process corresponding to $\theta CDT'$. $\Box$

The reason that singleton failures refinement does not imply data refinement for communicating data types with non-deterministic outputs is because the two semantic models differ in their treatment of output. In the relational model from a given state a the environment calls an operation n with input value in and, if such a value exists, is presented with an output value out. If there is no possible output value from the current state for the chosen operation and input value then we record this by mapping the state component to the bottom element $\perp$: the system is blocked. If there is some possible output value from the current state for the chosen operation and input value then we record whether or not our chosen value out is a possible output by whether or not a contribution is made to the semantic set.

In the singleton failures model we cannot distinguish between the situation in which the environment requiring output out is offered output out' instead and the situation in which no output is offered at all. The environment and the data type synchronise and agree upon a single event op.$name.in.out$. Such an event can be refused either if there is *no* possible output value for the chosen operation and input value from the current state or if both *out* and another distinct value out' are possible outputs for the chosen operation and input value from the current state.

We have shown that for the processes corresponding to communicating data types we cannot distinguish between an operation being refused because an alternative output value is being offered and an operation being refused because no output is being offered. This means that data refinement is not equivalent to process refinement for communicating data types with non-deterministic outputs.

To characterise data refinement in terms of process refinement, we must consider also the behaviour of a second form of corresponding process, one that does not record any output information. For the blocking semantics, we define

$$
\begin{array}{l}
\mathit{inputprocess}_b(\theta\, CDT[\mathit{State}, \mathit{Name}, \mathit{Value}]) = \\
\quad \text{let} \\
\qquad P(a) = \\
\qquad\quad (\,\Box\, \mathit{name} : \operatorname{dom} \mathit{ops};\ \mathit{in} : \mathit{Value} \mid (a, \mathit{in}) \in \operatorname{dom} \mathit{ops}\ \mathit{name} \bullet \\
\qquad\qquad \sqcap\, a' : \mathit{State} \mid (\exists\, \mathit{out} : \mathit{Value} \bullet (a, \mathit{in}) \mapsto (a', \mathit{out}) \in \mathit{ops}\ \mathit{name}) \bullet \\
\qquad\qquad\quad \mathrm{op}_{\mathrm{in}}.\mathit{name}.\mathit{in} \rightarrow P(a')\,) \\
\quad \text{within} \\
\qquad \sqcap\, a' : \mathit{init} \bullet P(a')
\end{array}
$$

Each input process can refuse the event $\mathrm{op}_{\mathrm{in}}.\mathit{name}.\mathit{in}$ precisely when operation *name* could fail, when called with input *in*, upon data type θCDT. This is identical to $\mathit{process}_{io\,b}(\theta CDT[\mathit{State}, \mathit{Name}, \mathit{Value}])$, except that the environment is unable to observe, or influence, the choice of output value. For the non-blocking semantics, we define

$$
\begin{array}{l}
\mathit{inputProcess}_n(\theta\, CDT[\mathit{State}, \mathit{Name}, \mathit{Value}]) = \\
\quad \text{let} \\
\qquad P(a) = \\
\qquad\quad \Box\, \mathit{name} : \operatorname{dom} \mathit{ops};\ \mathit{in} : \mathit{Value} \mid (a, \mathit{in}) \in \operatorname{dom} \mathit{ops}\ \mathit{name} \bullet \\
\qquad\qquad \sqcap\, a' : \mathit{State} \mid (\exists\, \mathit{out} : \mathit{Value} \bullet (a, \mathit{in}) \mapsto (a', \mathit{out}) \in \mathit{ops}\ \mathit{name}) \bullet \\
\qquad\qquad\quad \mathrm{op}_{\mathrm{in}}.\mathit{name}.\mathit{in} \rightarrow P(a') \\
\qquad \Box \\
\qquad (\,\Box\, \mathit{name} : \operatorname{dom} \mathit{ops};\ \mathit{in} : \mathit{Value} \mid (a, \mathit{in}) \notin \operatorname{dom} \mathit{ops}\ \mathit{name} \bullet \\
\qquad\qquad \mathrm{op}_{\mathrm{in}}.\mathit{name}.\mathit{in} \rightarrow \mathit{chaos}_{\mathit{IN}}(\,\theta\, CDT[\mathit{State}, \mathit{Name}, \mathit{Env}]\,) \\
\qquad\ \ \sqcap \\
\qquad\ \ \mathit{Stop}\,) \\
\quad \text{within} \\
\qquad \sqcap\, s' : \mathit{init} \bullet P(s')
\end{array}
$$

The process $chaos_{IN}$, like the process $chaos_{CDT}$, can accept or refuse any sequence of events corresponding to named operations, but unlike $chaos_{CDT}$ elides output information.

$$
\begin{array}{l}
chaos_{IN}\,(\,\theta CDT[State, Name, Env]\,) \;\; = \\
\quad \Box\, name : \mathrm{dom}\, ops;\; in : Value \bullet \\
\qquad \mathrm{op}_{\mathrm{in}}.name.in \rightarrow chaos_{IN}\,(\,\theta CDT[State, Name, Env]\,)) \\
\quad \sqcap \\
\quad Stop
\end{array}
$$

Although the input processes are not enough to characterise data refinement—they do not distinguish between a single, deterministic output, and a non-deterministic choice of outputs—they identify the program input sequences that can fail altogether. This is precisely the information that we need to complete the refinement picture.

Theorem 15 Data refinement of communicating data types is equivalent to the simultaneous traces refinement of the corresponding processes *and* singleton failures refinement of the corresponding input processes.

$$
\begin{array}{l}
\forall\, CDT;\; CDT' \bullet \\
\quad \theta CDT \sqsubseteq_{\mathcal{R}_{CDT}} \theta CDT' \\
\qquad \Leftrightarrow (\, process_{io}\,(\theta CDT) \sqsubseteq_{\mathcal{T}} process_{io}\,(\theta CDT') \\
\qquad\quad \wedge \\
\qquad\quad inputProcess\,(\theta CDT) \sqsubseteq_{\mathcal{S}} inputProcess\,(\theta CDT')\,)
\end{array}
$$

Proof We present the proof within the context of the blocking model. Arguing along similar lines gives the corresponding result within the context of the non-blocking semantics.

Before beginning the proof we recall that $Prog == \mathrm{seq}\, Name$ and we introduce functions similar to those employed in Theorem 12 of Chapter 4 that translate constructs from the relational framework to constructs in the event-based framework.

$$
\begin{array}{|l}
toInEvent_{io} : Name \times Value \rightarrow Event \\
toTrace_{io} : Prog \times (\mathrm{seq}\, Value \times \mathrm{seq}\, Value) \twoheadrightarrow Trace \\
\hline
toInEvent_{io} = \{name : Name;\; in : Value \bullet (name, in) \mapsto \mathrm{op}_{\mathrm{in}}.name.in\} \\
toTrace_{io} = \{prog : Prog;\; ins, outs : \mathrm{seq}\, Value \mid \#ins = \#outs = \#prog \bullet \\
\qquad\qquad (prog, (ins, outs)) \mapsto \\
\qquad\qquad\quad \{i : 1\,..\,\#prog \bullet i \mapsto \mathrm{op}.(prog.i).(ins.i).(outs.i)\}\,\}
\end{array}
$$

Consider data types θCDT and $\theta CDT'$ of types $CDT[AState, Name, Value]$ and $CDT[CState, Name, Value]$ respectively, and consider their corresponding wound data types θWDT and $\theta WDT'$ obtained by applying the function *simplify*. We reason as follows:

$$\theta CDT \sqsubseteq_{\mathcal{R}_{CDT}} \theta CDT'$$

$$\Leftrightarrow \qquad [\text{def}^n \text{ of } \sqsubseteq_{\mathcal{R}_{CDT}}]$$
$$\forall\, prog : Prog \bullet \mathcal{R}_{CDT}[\,\theta CDT'\,]\, prog \subseteq \mathcal{R}_{CDT}[\,\theta CDT\,]\, prog$$

$$\Leftrightarrow \qquad [\text{def}^n \text{ of } \mathcal{R}_{CDT}]$$
$$\forall\, prog : Prog \bullet \mathcal{R}_{WDT}[\,\theta WDT'\,]\, prog \subseteq \mathcal{R}_{WDT}[\,\theta WDT\,]\, prog$$

$$\Leftrightarrow \qquad [\text{def}^n \text{ of } \mathcal{R}_{WDT}]$$
$$\begin{aligned}&\forall\, prog : Prog \bullet\\ &\quad init_{WDT}{}' \mathbin{;} run_{WDT}\ prog\ ops' \mathbin{;} final_{WDT}{}'^{\top}\\ &\qquad \subseteq\\ &\qquad\quad init_{WDT} \mathbin{;} run_{WDT}\ prog\ ops \mathbin{;} final_{WDT}{}^{\top}\end{aligned}$$

$$\Leftrightarrow \qquad [\text{set theory}]$$
$$\begin{aligned}&\forall\, prog : Prog;\ ins, outs : \mathrm{seq}\ Value \mid \#ins = \#prog \wedge \#outs \leq \#prog \bullet\\ &\quad ins \mapsto outs \in init_{WDT}{}' \mathbin{;} run_{WDT}\ prog\ ops' \mathbin{;} final_{WDT}{}'^{\top}\\ &\qquad \Rightarrow\\ &\qquad\quad ins \mapsto outs \in init_{WDT} \mathbin{;} run_{WDT}\ prog\ ops \mathbin{;} final_{WDT}{}^{\top}\end{aligned}$$

$$\Leftrightarrow \qquad [\text{def}^n \text{ of } simplify \text{ and property of } {}^{\top}]$$
$$\begin{aligned}&\forall\, prog : Prog;\ ins, outs : \mathrm{seq}\ Value \mid \#ins = \#prog \wedge \#outs \leq \#prog \bullet\\ &\quad \exists\, c : init_{CDT}{}';\ c' : CState_{\perp} \bullet (c, ins, \langle\rangle) \mapsto (c', \langle\rangle, outs) \in run_{WDT}\ prog\ ops'\\ &\qquad \Rightarrow\\ &\qquad\quad \exists\, a : init_{CDT};\ a' : AState_{\perp} \bullet\\ &\qquad\qquad (a, ins, \langle\rangle) \mapsto (a', \langle\rangle, outs) \in run_{WDT}\ prog\ ops\end{aligned}$$

By exploiting the properties of $\forall$, and by observing that the internal state maps to the bottom element precisely when an operation has been blocked and hence precisely when the number of outputs is strictly less than the number of inputs, we see that this reduces to the following two cases:

1.

$$\begin{aligned}&\forall\, prog : Prog;\ ins, outs : \mathrm{seq}\ Value \mid \#ins = \#outs = \#prog \bullet\\ &\quad \exists\, c : init_{CDT}{}';\ c' : CState \bullet\\ &\qquad\qquad (c, ins, \langle\rangle) \mapsto (c', \langle\rangle, outs) \in run_{WDT}\ prog\ ops'\\ &\qquad \Rightarrow\\ &\qquad\quad \exists\, a : init_{CDT};\ a' : state_{CDT} \bullet\\ &\qquad\qquad (a, ins, \langle\rangle) \mapsto (a', \langle\rangle, outs) \in run_{WDT}\ prog\ ops\end{aligned}$$

2.

$$\begin{array}{l} \forall\, prog : Prog;\ ins, outs : \mathrm{seq}\ Value \mid \#ins = \#prog \land \#outs < \#prog \bullet \\ \quad \exists\, c : init_{CDT}{}' \bullet \\ \qquad\quad (c, ins, \langle\rangle) \mapsto (\bot, \langle\rangle, outs) \in run_{WDT}\ prog\ ops' \\ \qquad \Rightarrow \\ \qquad\quad \exists\, a : init_{CDT} \bullet \\ \qquad\qquad (a, ins, \langle\rangle) \mapsto (\bot, \langle\rangle, outs) \in run_{WDT}\ prog\ ops \end{array}$$

Observing that we can find states $a : init_{CDT}$ and $a' : state_{CDT}$ such that the predicate $(a, ins, \langle\rangle) \mapsto (a', \langle\rangle, outs) \in run_{WDT}\ prog\ ops$ is true precisely when the predicate $toTrace_{io}\,(prog, (ins, outs)) \in \mathcal{T}\,[\![process_{io\,b}\,(\,\theta CDT\,)]\!]$ is also true, we see that clause 1 is equivalent to $1'$ below.

$1'$.

$$\begin{array}{l} \forall\, prog : Prog;\ ins, outs : \mathrm{seq}\ Value \mid \#ins = \#outs = \#prog \bullet \\ \quad toTrace_{io}\,(prog, (ins, outs)) \in \mathcal{T}\,[\![process_{io\,b}\,(\,\theta CDT'\,)]\!] \\ \qquad \Rightarrow \\ \qquad\quad toTrace_{io}\,(prog, (ins, outs)) \in \mathcal{T}\,[\![process_{io\,b}\,(\,\theta CDT\,)]\!] \end{array}$$

This is in turn equivalent to clause $1''$.

$1''$.

$$\mathcal{T}\,[\![process_{io\,b}\,(\,\theta CDT'\,)]\!] \ \subseteq\ \mathcal{T}\,[\![process_{io\,b}\,(\,\theta CDT\,)]\!]$$

Observing that clause $1''$ implies $\mathcal{T}\,[\![inputProcess_b\,(\,\theta CDT'\,)]\!] \ \subseteq\ \mathcal{T}\,[\![inputProcess_b\,(\,\theta CDT\,)]\!]$ and hence implies

$$\begin{array}{l} \mathcal{S}\,[\![inputProcess_b\,(\,\theta CDT'\,)]\!] \cap \{ tr : Trace \bullet (tr, \emptyset) \} \\ \quad \subseteq \\ \qquad \mathcal{S}\,[\![inputProcess_b\,(\,\theta CDT\,)]\!] \cap \{ tr : Trace \bullet (tr, \emptyset) \} \end{array}$$

we conclude that clause $1''$ is equivalent to clause $1'''$ below.

$1'''$.

$$\begin{array}{l} \mathcal{T}\,[\![process_{io\,b}\,(\,\theta CDT'\,)]\!] \ \subseteq\ \mathcal{T}\,[\![process_{io\,b}\,(\,\theta CDT\,)]\!] \\ \land \\ \mathcal{S}\,[\![inputProcess_b\,(\,\theta CDT'\,)]\!] \cap \{ tr : Trace \bullet (tr, \emptyset) \} \\ \quad \subseteq \\ \qquad \mathcal{S}\,[\![inputProcess_b\,(\,\theta CDT\,)]\!] \cap \{ tr : Trace \bullet (tr, \emptyset) \} \end{array}$$

We now turn our attention to the second clause. We may throw away the information about operations that would have been called had an operation not already been called outside its preconditions. Hence we may consider programs, sequences of inputs and sequences of outputs of the same length. We see that clause 2 is equivalent to clause $2'$ below.

$2'$.

$$
\begin{array}{l}
\forall\, prog : Prog;\ ins, outs : \mathrm{seq}\ Value;\ n : Name;\ in : Value \mid \\
\qquad \#ins = \#outs = \#prog \bullet \\
\quad \exists\, c : init_{CDT}{}';\ c' : state_{CDT}{}' \bullet (c, ins, \langle\rangle) \mapsto (c', \langle\rangle, outs) \in run_{WDT}\ prog\ ops' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (c', in) \notin \mathrm{dom}\ ops_{CDT}{}'\ n \\
\qquad \Rightarrow \\
\qquad\quad \exists\, a : init_{CDT};\ a' : state_{CDT} \bullet (a, ins, \langle\rangle) \mapsto (a', \langle\rangle, outs) \in run_{WDT}\ prog\ ops \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (a', in) \notin \mathrm{dom}\ ops_{CDT}\ n
\end{array}
$$

Recalling the definition of *inputProcess* we see that this is in turn equivalent to clause $2''$:

$2''$.

$$
\begin{array}{l}
\forall\, prog : Prog;\ ins, outs : \mathrm{seq}\ Value;\ name : Name;\ in : Value \mid \\
\qquad \#ins = \#outs = \#prog \bullet \\
\quad (toTrace_{io}\,(prog, (ins, outs)), \{toInputEvent\,(name, in)\}) \\
\qquad\qquad\qquad \in\ \mathcal{S}\,[\![inputProcess_{b}\,(\,\theta CDT'\,)]\!] \\
\qquad \Rightarrow \\
\qquad\quad (toTrace_{io}\,(prog, (ins, outs)), \{toInputEvent\,(name, in)\}) \\
\qquad\qquad\qquad\qquad \in\ \mathcal{S}\,[\![inputProcess_{b}\,(\,\theta CDT\,)]\!]
\end{array}
$$

and hence equivalent to clause $2'''$ below.

$2'''$.

$$
\begin{array}{l}
\mathcal{S}\,[\![inputProcess_{b}\,(\,\theta CDT'\,)]\!] \setminus \{tr : Trace \bullet (tr, \emptyset)\} \\
\qquad \subseteq \\
\qquad\quad \mathcal{S}\,[\![inputProcess_{b}\,(\,\theta CDT\,)]\!] \setminus \{tr : Trace \bullet (tr, \emptyset)\}
\end{array}
$$

Our desired result follows directly from the conjunction of clauses $1'''$ and $2'''$. We conclude that one communicating data type is refined by another precisely when the process corresponding to the first is traces refined by the process corresponding to the second *and* the input process corresponding to the first is refined within the singleton failures model by the input process corresponding to the second. $\Box$

We have identified a sound and complete method for determining whether or not one communicating data type is refined by another. This method employs both the traces model and the singleton failures model. However, since the focus of this thesis has been on the singleton failures model we will show how an equivalent result may be established by considering only the singleton failures refinement ordering. For such a result we need first to prove Lemma 6 below.

Lemma 6

$$\begin{array}{l} \forall\, CDT;\ CDT' \bullet \\ \quad (\, process_{io}\,(\theta CDT) \sqsubseteq_{\mathcal{T}} process_{io}\,(\theta CDT') \\ \quad\ \wedge \\ \quad\ inputProcess\,(\theta CDT) \sqsubseteq_{\mathcal{S}} inputProcess\,(\theta CDT')\,) \\ \qquad \Rightarrow \\ \qquad\quad process_{io}\,(\theta CDT) \sqsubseteq_{\mathcal{S}} process_{io}\,(\theta CDT') \end{array}$$

Proof Consider communicating data types θCDT and $\theta CDT'$ respectively of types $CDT[AState, Name, Value]$ and $CDT[CState, Name, Value]$ such that the following predicate holds.

$$\begin{array}{l} process_{io}\,(\theta CDT) \sqsubseteq_{\mathcal{T}} process_{io}\,(\theta CDT') \\ \wedge \\ inputProcess\,(\theta CDT) \sqsubseteq_{\mathcal{S}} inputProcess\,(\theta CDT') \end{array}$$

Given trace tr we reason as follows:

$$\begin{array}{lr} (tr, \emptyset) \in \mathcal{S}\,[\![process_{io}\,(\theta CDT')]\!] & \\ \quad \Leftrightarrow\ tr \in \mathcal{T}\,[\![process_{io}\,(\theta CDT')]\!] & [\text{property of } \mathcal{S}] \\ \quad \Rightarrow\ tr \in \mathcal{T}\,[\![process_{io}\,(\theta CDT)]\!] & [process_{io}\,(\theta CDT) \sqsubseteq_{\mathcal{T}} process_{io}\,(\theta CDT')] \\ \quad \Leftrightarrow\ (tr, \emptyset) \in \mathcal{S}\,[\![process_{io}\,(\theta CDT)]\!] & [\text{property of } \mathcal{S}] \end{array}$$

Next suppose that for some trace tr and some operation $\mathrm{op}.name.in.out$ the predicate $(tr, \{\mathrm{op}.name.in.out\}) \in \mathcal{S}\,[\![process_{io}\,(\theta CDT')]\!]$ holds. This tells us that after trace tr we might be in a state $c : CState$ from which $process_{io}\,(\theta CDT')$ can refuse the event $\mathrm{op}.name.in.out$. This implies that either $(c, in) \notin \mathrm{dom}\, ops'\, name$ or that there is another distinct output out' such that we can find an after-state $c' : CState$ such that the predicate $(c, in) \mapsto (c', out') \in ops\ name$ holds.

If the former is true then, by definition of $inputProcess$, it follows that the predicate $(tr, \{\mathrm{op}_{\mathrm{in}}.name.in\}) \in \mathcal{S}\,[\![inputProcess\,(\theta CDT')]\!]$ must also be true. Hence we infer that $(tr, \{\mathrm{op}_{\mathrm{in}}.name.in\}) \in \mathcal{S}\,[\![inputProcess\,(\theta CDT)]\!]$ since $inputProcess\,(\theta CDT) \sqsubseteq_{\mathcal{S}} inputProcess\,(\theta CDT')$. Furthermore, from this we conclude that there must be a state $a : AState$, reachable after trace tr, such that $(a, in) \notin \mathrm{dom}\, ops\ name$ is true and hence that $(tr, \{\mathrm{op}.name.in.out\}) \in \mathcal{S}\,[\![process_{io}\,(\theta CDT)]\!]$ holds.

If the latter is true then $tr \frown \langle \mathrm{op}.name.in.out' \rangle \in \mathcal{T}\,[\![process_{io}\,(\theta CDT')]\!]$ and hence, since $process_{io}\,(\theta CDT) \sqsubseteq_{\mathcal{T}} process_{io}\,(\theta CDT')$ it follows that $tr \frown \langle \mathrm{op}.name.in.out' \rangle \in \mathcal{T}\,[\![process_{io}\,(\theta CDT)]\!]$. From this we see easily that there must be a state a, reachable after trace tr such that there is another state a' such that $(a, in) \mapsto (a', out) \in ops\ name$ and hence conclude that $(tr, \{\mathrm{op}.name.in.out\}) \in \mathcal{S}\,[\![process_{io}\,(\theta CDT)]\!]$.

We have shown that

$$
\begin{array}{l}
\mathcal{S}\,[\![process_{io}\,(\theta CDT')]\!] \;\cap\; \{tr : Trace \bullet (tr, \langle\rangle)\} \\
\qquad \subseteq \\
\qquad\quad \mathcal{S}\,[\![process_{io}\,(\theta CDT)]\!] \;\cap\; \{tr : Trace \bullet (tr, \langle\rangle)\}
\end{array}
$$

and also that

$$
\begin{array}{l}
\mathcal{S}\,[\![process_{io}\,(\theta CDT')]\!] \;\setminus\; \{tr : Trace \bullet (tr, \langle\rangle)\} \\
\qquad \subseteq \\
\qquad\quad \mathcal{S}\,[\![process_{io}\,(\theta CDT)]\!] \;\setminus\; \{tr : Trace \bullet (tr, \langle\rangle)\}.
\end{array}
$$

Hence we conclude our desired result.

$$process_{io}\,(\theta CDT) \quad \sqsubseteq_{\mathcal{S}} \quad process_{io}\,(\theta CDT')$$

□

We may now prove that data refinement of communicating data types is equivalent to singleton failures refinement both of their corresponding processes and of their corresponding input processes.

Theorem 16 Data refinement of communicating data types is equivalent to the simultaneous singleton failures refinement of the corresponding processes *and* the corresponding input processes.

$$
\begin{array}{l}
\forall\, CDT;\; CDT' \bullet \\
\qquad \theta CDT \;\sqsubseteq_{\mathcal{R}_{CDT}} \theta CDT' \\
\qquad\qquad \Leftrightarrow (\, process_{io}\,(\theta CDT) \;\sqsubseteq_{\mathcal{S}}\; process_{io}\,(\theta CDT') \\
\qquad\qquad\quad\;\; \wedge \\
\qquad\qquad\quad\;\; inputProcess\,(\theta CDT) \;\sqsubseteq_{\mathcal{S}}\; inputProcess\,(\theta CDT')\,)
\end{array}
$$

Proof Lemma 6 above and the fact that singleton failures refinement implies traces refinement tells us that the predicates

$$
\begin{array}{l}
process_{io}\,(\theta CDT) \;\sqsubseteq_{\mathcal{T}}\; process_{io}\,(\theta CDT') \\
\wedge \\
inputProcess\,(\theta CDT) \;\sqsubseteq_{\mathcal{S}}\; inputProcess\,(\theta CDT')
\end{array}
$$

and

$$
\begin{array}{l}
process_{io}\,(\theta CDT) \;\sqsubseteq_{\mathcal{S}}\; process_{io}\,(\theta CDT') \\
\wedge \\
inputProcess\,(\theta CDT) \;\sqsubseteq_{\mathcal{S}}\; inputProcess\,(\theta CDT')
\end{array}
$$

are equivalent. Combining this observation with Theorem 15 gives us our desired result.
□

Corollary 4 For communicating data types with deterministic outputs, data refinement is equivalent to singleton failures refinement.

$$\begin{array}{l}\forall\, CDT;\; CDT' \mid (\forall\, n : \mathrm{dom}\, ops';\; in : Value;\; s : state' \bullet \\ \qquad\qquad\qquad\qquad (\exists\, out_1, out_2 : Value;\; t_1, t_2 : state' \bullet \\ \qquad\qquad\qquad\qquad\qquad\qquad \{(t_1, out_1), (t_2, out_2)\} \subseteq ops'\, n(\!|\; \{(s, in)\}\; |\!)) \\ \qquad\qquad\qquad\qquad\quad \Rightarrow\; out_1 = out_2\,)\; \bullet \\ \quad \theta CDT\; \sqsubseteq_{\mathcal{R}_{CDT}}\; \theta CDT' \\ \qquad \Leftrightarrow \\ \qquad\quad process_{io}\,(\theta CDT)\; \sqsubseteq_{\mathcal{S}}\; process_{io}\,(\theta CDT')\end{array}$$

Proof Our desired result follows directly from Theorem 16 above and the following property of communicating data types with deterministic outputs:

$$\begin{array}{l}process_{io}\,(\theta CDT)\; \sqsubseteq_{\mathcal{S}}\; process_{io}\,(\theta CDT') \\ \quad \Rightarrow \\ \qquad inputProcess\,(\theta CDT)\; \sqsubseteq_{\mathcal{S}}\; inputProcess\,(\theta CDT')\end{array}$$

$\Box$

In Observation 9 we identified two communicating data types θCDT and $\theta CDT'$. We proved that θCDT is not data refined by $\theta CDT'$ although the process corresponding to θCDT is refined by the process corresponding to $\theta CDT'$. In Observation 10 below we confirm the result presented in Theorem 16 above.

Observation 10 The input process corresponding to θCDT is not refined by the input process corresponding to $\theta CDT'$ where θCDT and $\theta CDT'$ are as defined in Observation 9. Hence that we should not expect θCDT to be data refined by $\theta CDT'$.

Proof Within the context of the blocking semantic model, the process corresponding to the inputs of data type θCDT as defined in Observation 9 is as described below:

$$\begin{array}{l}inputprocess_b\,(\,\theta CDT\,)\; = \\ \quad \mathrm{let} \\ \qquad Proc(a)\; =\; (a == s)\;\&\; \mathrm{op}.n.0 \rightarrow Proc(t) \\ \quad \mathrm{within} \\ \qquad Proc(s)\end{array}$$

and hence can equivalently be expressed as $\mathrm{op}.n.0 \rightarrow Stop$. The singleton failures of this process are as follows:

$$\{(\langle\rangle, \emptyset), (\langle\rangle, \{\mathrm{op}.n.1\}), (\langle \mathrm{op}.n.0\rangle, \emptyset), (\langle \mathrm{op}.n.0\rangle, \{\mathrm{op}.n.0\}), (\langle \mathrm{op}.n.0\rangle, \{\mathrm{op}.n.1\})\}.$$

Similarly, the process corresponding to the inputs of data type $\theta CDT'$ as defined in Observation 9 is as given below:

$$
\begin{array}{l}
\mathit{inputprocess}_b\,(\,\theta CDT'\,) \;=\; \\
\quad \text{let} \\
\quad\quad \mathit{Proc}(a) \;=\; (a == s)\ \&\ \text{op}.n.0 \rightarrow \mathit{Proc}(t) \\
\quad \text{within} \\
\quad\quad \mathit{Proc}(t)
\end{array}
$$

This process can equivalently be expressed as Stop, and hence its singleton failures are as follows:

$$\{(\langle\rangle, \emptyset), (\langle\rangle, \{\text{op}.n.0\}), (\langle\rangle, \{\text{op}.n.1\})\}$$

We observe that the pair $(\langle\rangle, \{\text{op}.n.0\})$ lies in the set $\mathcal{S}\,[\![\mathit{inputprocess}_b\,(\,\theta CDT'\,)]\!]$ but not in the set $\mathcal{S}\,[\![\mathit{inputprocess}_b\,(\,\theta CDT\,)]\!]$. Hence, by Theorem 16 above we confirm the result that we had already proved in Observation 9: θCDT is not data refined by $\theta CDT'$. $\Box$

Chapter 6

On formalising UML

The popularity of object-oriented methodologies such as Catalysis [DW98], Fusion [Col93], OMT [RBP$^+$91], SDL [EHS97], UML [RJB97] and XP [Bec99] has had many contributing factors. As the complexity of systems increases, so does the importance of good modelling techniques. Equally, as businesses are putting greater emphasis on "time to market", the ability to re-use components is taking on a more critical role in the software development process. Hence the increased need for adequate documentation.

The Unified Modeling Language (UML) [RJB97] is a graphical specification language that has been widely adopted for object-oriented software development. Indeed it has been accepted as the industry-standard language [OMG99] for specifying, visualising, constructing, and documenting object-oriented software systems. UML comprises a collection of graphical notations each of which can be used to specify different aspects of the structure, state or behaviour of the system at varying levels of rigour and abstraction.

Use case diagrams can be used to identify desired functionality, and CRC cards[1] can be used during brain-storming sessions both to identify methods and classes and to decide which classes should be responsible for which methods. Class diagrams can be used to record data members and methods associated with individual classes as well as the relationships between the classes themselves. Furthermore, state diagrams, based upon Harel state charts [Har87], can be used to show how individual methods alter the data members of the class. Two types of interaction diagram—message sequence diagrams and collaboration diagrams—can be used to record how objects interact. Finally activity graphs, diagrams that combine features of state charts with the intuition of Petri Nets[Pet81], can be used to describe the flow of behaviour.

UML has an extensively-documented [OMG99] syntactic definition, or structural semantics but, as [EFLR98] observe, lacks any formal static or behavioural semantics. In this chapter we explore how formal description techniques can be used to reason about UML specifications. We discuss how descriptions of system behaviour, expressed in terms of abstract data types and processes, can be constructed from individual UML diagrams and explain how the work presented in previous chapters might be exploited to formally,

[1] CRC (Classes, Responsibilities and Collaborator) cards are not an official part of the UML notation but are used widely used during the development process and included in much of the literature on The Unified Modeling Language [SP99, FS99].

and automatically, analyse and verify the consistency of UML specifications.

Our intention is not to give a complete or formal semantics to the whole of UML. Indeed, perhaps that is not an appropriate thing to do. Before we embark on the formalisation process it is important to consider the role of the various notations within the language.

The use of UML in software specification and design can be divided into three, not necessarily distinct, phases: an initial brain-storming period; the development process; and an implementation-level specification. Diagrams constructed in the first phase—for instance CRC cards and use case diagrams—are a means of getting ideas flowing and opening up a dialogue between various members of the team. A formal semantics is less useful here.

We focus our attention on diagrams constructed during the development process and implementation-level specification. We begin the chapter by presenting a detailed formal semantics for activity diagrams describing the behaviour of the system. An important feature of our chosen syntactic interpretation of activity graphs is that it matches closely Rose's interchange format [Rat] thereby facilitating automation of consistency checks.

We do not attempt in this thesis to formalise the state-based notations within UML. This area has been widely researched by others and we discuss their approaches in Section 7.5. Rather, we *assume* that an abstract data type model of a system can be extracted from its final class description (a combination of state diagrams and annotated class diagrams). We show how these formal interpretations, both state-based and behavioural, along with the work presented in previous chapters, might be used to support analysis and to verify consistency of UML specifications. We conclude the chapter with a case study.

6.1 Graphical and syntactic descriptions of activity graphs

An activity graph is a special type of state diagram that is used to model the flow of activities within a procedure. Essentially it describes a collection of use cases. An activity graph is constructed from a combination of action states, (sub)activity states, start and finish states and pseudostates merge, decision, fork and join.

Merge states and join states both have multiple incoming lines and a single outgoing transition, the difference being that an activity following a join state can be performed only once *all* the activities with transitions leading into the join state have occurred. An activity following a merge state can, however, be performed if *any* of the activities with transitions leading into the state have occurred.

Both decision states and fork states have a single incoming line and multiple outgoing transitions. If an activity preceding a decision state has occurred then any *single* activity with transitions from the decision state can occur, whereas if an activity preceding a join state occurs then *all* activities with transitions from the join state can occur. Both fork and join states may sometimes be referred to as *synchronisation bars*.

A table showing the graphical representation of each possible type of state within an

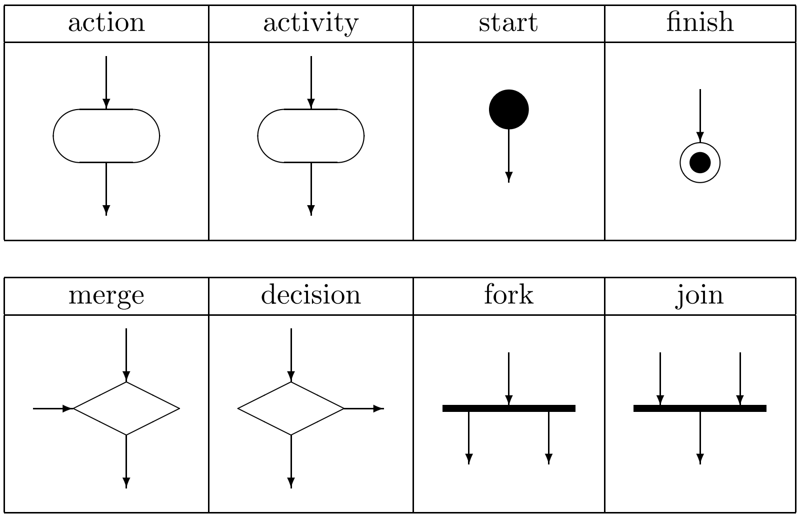

Figure 6.1: States of an activity graph.

activity graph is presented in Figure 6.1. Syntactically we define these states as follows:

$$
\begin{array}{lcl}
\mathit{Type} & ::= & \mathit{action}\langle\!\langle \mathit{Action}\rangle\!\rangle \mid \\
 & & \mathit{activity}\langle\!\langle \mathit{ActivityLabel}\rangle\!\rangle \mid \\
 & & \mathit{start} \mid \mathit{finish} \mid \\
 & & \mathit{merge} \mid \mathit{decision} \mid \mathit{fork} \mid \mathit{join}
\end{array}
$$

where Action and ActivityLabel are given types. Observe that action states and activity states are graphically indistinguishable. Semantically, however, they are very different. An *action* can be thought of as a simple task that cannot be broken down any further, whereas an *activity* is a task that can be broken down into subtasks and can itself be expressed as an activity graph. An *ActivityLabel* within an activity state points to another activity graph within the specification; we illustrate this in Example 7 below.

Example 7 Constructing a house.

Figure 6.2 shows a simple pair of activity graphs that might be used to model the process of constructing a house. This example illustrates how graphs can be nested.

The first activity graph, "Create House", gives a top level description of the main tasks that need to be completed and the order in which they must be done: first the plans must be drawn up; then the house must be built before it is finally decorated.

Within this graph is the activity state "Build House" which points to the second activity graph. This second graph further breaks down the actual building process into four subtasks. First the foundations must be laid and then the walls must be built. Once these tasks have been completed the electricity must be installed and the roof put on. These tasks can be completed concurrently or in either order. □

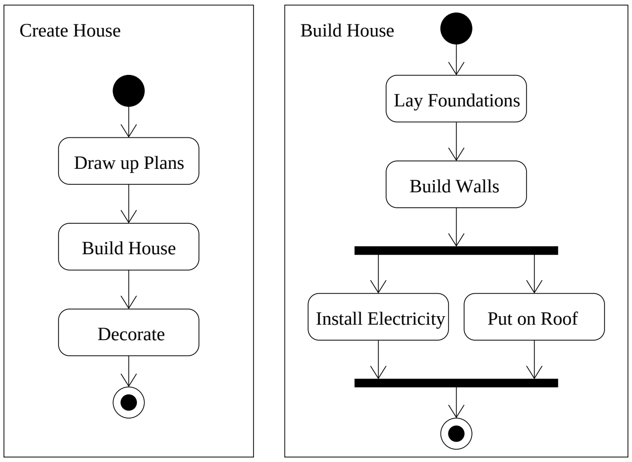

Figure 6.2: A graph nested within a graph.

Transition lines

The states within an activity graph are linked together by transition lines. These lines can have associated guards represented by Booleans and can have associated actions[2] represented by events as illustrated in Figure 6.3. An activity following a guarded tran-

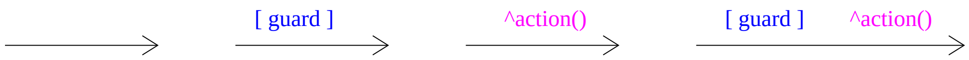

Figure 6.3: A simple transition, a guarded transition, a transition enabled by an action and a guarded transition enabled by an action.

sition can only be offered if the associated guard evaluates to true. Similarly, an activity following a transition labelled with an action can occur only when the trigger event has occurred.

In our formal description of a transition we must record the name of the line as well as any associated guards and actions. Given the types *Line* and *Guard*, the schema definition of a transition is as follows:

Transition
$guard : Guard$
$action : Action$
$line : Line$

[2]In the latest version of the UML documentation [OMG99], greater distinction is being drawn between activity graphs and state diagrams. We would now expect to see transitions labelled with actions in state diagrams, but not in activity graphs which are being used more as flow charts. Nevertheless we have retained them in our semantic treatment to facilitate extension to the language of state diagrams, or state charts.

If the graphical representation shows no guard then the schema description of the transition records the default value *true*. Similarly if the graphical representation shows no action then the schema description records the default value *null*.

$$\mid \; true : Guard \qquad\qquad\qquad \mid \; null : Action$$

Finally, we define such a *SimpleTransition* to be a transition that has no guard and no associated action.

$$SimpleTransition \;\widehat{=}\; [\, Transition \mid guard = true \;\wedge\; action = null \,]$$

Example 8 A lift.

In Figure 6.4 we illustrate how the various types of transition line can be used within an activity graph. On request, that is when the trigger event "request" occurs, the lift

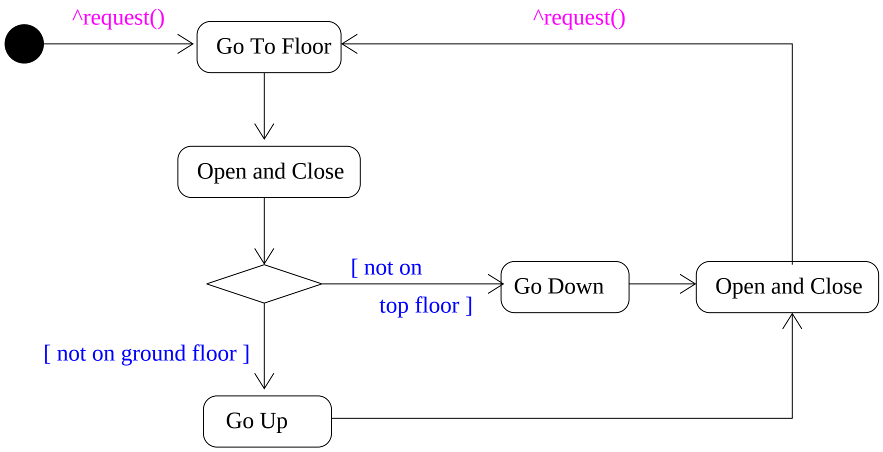

Figure 6.4: A lift.

goes to the desired floor and opens and closes its doors. Provided it is on one of the middle floors, that is the guards "not on ground floor" and "not on top floor" are both true, it can then either go up or go down. If it is on the top floor, and hence the guard "not on top floor" is false, the only option available is to go down. Similarly, if it is on the bottom floor, and hence the guard "not on ground floor" is false, the only option available is to go up. Having moved, the lift opens and closes its doors and is then prepared to receive another floor request. □

States

Each *state* within our schema representation of an activity graph records the type of its contents, the set of incoming lines it requires and the set of outgoing transitions it enables.

$$
\begin{array}{|l}
\textit{State} \\
\hline
\textit{type} : \textit{Type} \\
\textit{lines} : \mathbb{P}\,\textit{Line} \\
\textit{transitions} : \mathbb{P}\,\textit{Transition} \\
\hline
\end{array}
$$

The type *State* allows all possible states, including those which are not permissible within a UML specification. The schema *WellFormedState* describes the subset of possible states that are permissible according to the constraints within the official documentation [OMG99].

$$
\begin{array}{|l}
\textit{WellFormedState} \\
\hline
\textit{State} \\
\hline
\textit{type} = \textit{start} \;\Leftrightarrow\; \textit{lines} = \emptyset \\
\textit{type} = \textit{finish} \;\Leftrightarrow\; \textit{transitions} = \emptyset \\
\textit{type} \in \{\textit{start}, \textit{merge}, \textit{join}\} \;\Rightarrow\; \#\textit{transitions} = 1 \\
\textit{type} \in \{\textit{finish}, \textit{decision}, \textit{fork}\} \;\Rightarrow\; \#\textit{lines} = 1 \\
(\,\exists\, \textit{tr} : \textit{transitions} \bullet \textit{tr}.\textit{line} \in \textit{lines}\,) \\
\qquad \Rightarrow\; \textit{type} \notin \{\textit{start}, \textit{finish}, \textit{merge}, \textit{decision}, \textit{fork}, \textit{join}\} \\
\hline
\end{array}
$$

A start state can have no incoming lines and a finish state can have no outgoing transition. To ensure connectedness, all other types of state must have both in-coming lines and out-going transitions. Start states, merge states and join states all have precisely one out-going transition and finish states, decision states and fork states all have at most one in-coming line. Finally, the only states that can have self-loops, that is share the same incoming line and outgoing transition, are action states and (sub)activity states.

Graphs

An activity graph is built up from a finite set of well-formed states. The environment of a specification is a function mapping the name of each graph to the associated graph. These are described formally below.

$$
\begin{array}{rcl}
\textit{Graph} & ::= & \textit{states}\,\langle\!\langle\, \mathbb{F}\,\textit{WellFormedState}\,\rangle\!\rangle \\
\textit{Env} & == & \textit{ActivityLabel} \nrightarrow \textit{Graph}
\end{array}
$$

We define a well-formed graph to be one in which no two states share incoming lines or outgoing transitions and in which all lines are attached to states at both ends.

$$
\begin{array}{|l}
WellFormedGraph : \mathbb{P}\, Graph \\
\hline
WellFormedGraph\ = \\
\quad \{graph : Graph \mid \\
\qquad \bigcup\{state : states^{\sim} graph \bullet \{tr : state.transitions \bullet tr.line\}\} \\
\qquad\quad = \\
\qquad\qquad \bigcup\{state : states^{\sim} graph \bullet state.lines\} \\
\qquad \wedge \\
\qquad \forall\, state_1, state_2 : states^{\sim} graph \mid state_1 \neq state_2 \bullet \\
\qquad\quad state_1.lines \cap state_2.lines = \emptyset \\
\qquad\quad \wedge \\
\qquad\quad state_1.transitions \cap state_2.transitions = \emptyset\,\}
\end{array}
$$

6.2 Behavioural and semantic descriptions of activity graphs

In this section we define our semantic function which takes the name of an activity graph within a specification and returns the CSP process that models the behaviour of that graph and any of its subgraphs. More specifically, we associate each state within the named graph with its corresponding process. These processes each include events corresponding to the incoming lines and outgoing transitions, with any accompanying actions, as well as the events corresponding to the state itself. The process corresponding to the graph is then the parallel combination of these processes, each synchronising on its own alphabet. The synchronisation of the line events ensures that the actions occur in the correct order.

Given the type *Event*, we define the functions ε_{line} and ε_{action} which, respectively, map line names and actions to events. We insist that events corresponding to each line and to each action are distinct. Having defined an *Action* to be a simple task that cannot be broken down any further we can justifiably model it as a single atomic event. All lines have a corresponding line event and all actions other than *null* have a corresponding action event.

$$
\begin{array}{|l}
\varepsilon_{line} : Line \rightarrowtail Event \\
\varepsilon_{action} : Action \mapsto\!\!\!\!\!\rightarrow Event \\
\hline
\operatorname{ran} \varepsilon_{line} \cap \operatorname{ran} \varepsilon_{action}\ =\ \emptyset \\
\operatorname{dom} \varepsilon_{action}\ =\ Action \setminus null
\end{array}
$$

Alphabets

In order to define the alphabet for each state—the events on which it must synchronise—we must consider the events associated with each line, type and transition. We introduce the functions α_{line}, α_{tran} and α_{type} which respectively take a line, a transition and the type of a state and return the associated set of events. Given these definitions we then define the function α_{state} which takes a state within the graph and returns its alphabet, the set of events corresponding to its incoming lines, outgoing transitions and its type.

The set of events associated with a line contains a single element, the event corresponding to the line. This is simply the relational image of the line under function ε_{line}. The set of events associated with a transition is the set containing the events corresponding to its line and its action.

$$
\begin{array}{|l}
\alpha_{line} : Line \nrightarrow \mathbb{P}\, Event \\
\alpha_{tran} : Transition \nrightarrow \mathbb{P}\, Event \\
\hline
\alpha_{line} \;=\; (\lambda\, line : Line \bullet \varepsilon_{line}(\!|\, \{line\} \,|\!)) \\
\alpha_{tran} \;=\; (\lambda\, Transition \bullet \varepsilon_{line}(\!|\, \{line\} \,|\!) \cup \varepsilon_{action}(\!|\, \{action\} \,|\!))
\end{array}
$$

Observe that, since *null* is not in the domain of the function ε_{action} and hence the set $\varepsilon_{action}(\!|\ \{null\}\ |\!)$ is empty, if the action associated with a transition is *null* then this component does not contribute to the alphabet.

The set of events associated with the type *action* state with given action *actn* is the set containing the singleton event corresponding to *actn*. The set of events associated with the type *activity* state with given ActivityLabel *actL* is the alphabet of the graph corresponding to *actL*; this is simply the union of the alphabets of all the states in the identified graph. The set of events associated with all other types of state is simply the empty set.

$$
\begin{array}{|l}
\alpha_{type} : Type \nrightarrow Env \nrightarrow \mathbb{P}\, Event \\
\hline
\forall\, env : Env;\ type : Type;\ actn : Action;\ actL : ActivityLabel \bullet \\
\quad type \in \{start, finish, merge, decision, fork, join\} \\
\qquad \Rightarrow \\
\qquad \alpha_{type}\ type\ env = \emptyset \\
\quad \wedge \\
\quad \alpha_{type}\ (action\ actn)\ env \;=\; \{\varepsilon_{action}\ actn\} \\
\quad \wedge \\
\quad \alpha_{type}\ (activity\ actL)\ env \;=\; \bigcup\ \{\, state : State \mid state \in states^{\sim}(env\ actL) \bullet \\
\qquad\qquad\qquad\qquad\qquad\qquad \alpha_{state}\ state\ env \,\}
\end{array}
$$

The alphabet of any given state is then the union of the alphabets of all of its incoming lines, the alphabet of its type and the alphabets of all of its outgoing transitions.

$$
\begin{array}{|l}
\alpha_{state} : State \nrightarrow Env \nrightarrow \mathbb{P}\, Event \\
\hline
\alpha_{state} \;=\; (\lambda\, State \bullet \\
\qquad (\lambda\, env : Env \bullet (\bigcup(\alpha_{line}(\!|\ lines\ |\!)) \\
\qquad\qquad \cup \\
\qquad\qquad \bigcup(\alpha_{tran}(\!|\ transitions\ |\!)) \\
\qquad\qquad \cup \\
\qquad\qquad \alpha_{type}\ type\ env)))
\end{array}
$$

Processes corresponding to graphs, lines, transitions and states

We have observed that the process corresponding to the behaviour of a named activity graph within its specified environment is the parallel combination of the processes corresponding to its constituent states, each synchronising on its own alphabet. All line events are then hidden. Given the type CSP, we formally define as follows the function $toProcess$ that takes an ActivityLabel, the name of a graph, and takes the function defining the environment and returns the process that models the behaviour of the graph.

$$
\begin{array}{|l}
toProcess : ActivityLabel \rightarrow Env \rightarrow CSP \\
\hline
\forall\, env : Env;\ actL : ActivityLabel \bullet \\
\quad toProcess\ actL\ env\ = \\
\qquad (\|\ st : states^{\sim}\,(env\ actL)\ \bullet\ [\,\alpha_{state}\ st\ env\,]\ \rho_{state}\ st\ env) \\
\qquad\quad \setminus\ \bigcup(\,\{st : states^{\sim}\,(env\ actL) \bullet \varepsilon_{line}(\!|\ st.lines\ |\!)\}\,)
\end{array}
$$

Recalling that $Env == ActivityLabel \rightarrow Graph$, given an activity label $actL$ we can obtain the associated graph by applying the function env of type Env. Hence, by applying the inverse function $states^{\sim}$ to the graph $env\ actL$, we obtain the set of states contained within the activity graph labelled $actL$. We put the processes corresponding to each such state[3], st, in parallel each synchronising on its own alphabet, $\alpha_{state}\ st\ env$.

The function ρ_{state} that we used in the above definition returns the process corresponding to the behaviour of the given state. ρ_{state} is essentially the sequential composition of the processes corresponding to the lines, type and transitions of the given state. In order to define it formally we must consider these constituent processes and the effect of loops on the system.

The process that corresponds to the behaviour of the type action state with given action $actn$, is simply the event associated with $actn$ followed by $Skip$, whilst the process that corresponds to the behaviour of the type activity state with given ActivityLabel $actL$, is the process corresponding to the graph associated with $actL$. The process that corresponds to the behaviour of all other types of states is simply $Skip$.

$$
\begin{array}{|l}
\rho_{type} : State \rightarrow Env \rightarrow CSP \\
\hline
\forall\, state : State;\ env : Env;\ actn : Action;\ actL : ActivityLabel \bullet \\
\quad state.type = action\ actn \\
\qquad \Rightarrow \\
\qquad \rho_{type}\ state\ env = (\varepsilon_{action}\ actn) \rightarrow Skip \\
\quad \wedge \\
\quad state.type = activity\ actL \\
\qquad \Rightarrow \\
\qquad \rho_{type}\ state\ env = toProcess\ actL\ env \\
\quad \wedge \\
\quad state.type \in \{start, finish, merge, decision, fork, join\} \\
\qquad \Rightarrow \\
\qquad \rho_{type}\ state\ env = Skip
\end{array}
$$

[3]These processes are obtained by applying the function ρ_{state} which we will define formally later in this section.

We observed that the difference between a *merge* state and a *join* state is that a *merge* state can be reached provided any one of its incoming lines is enabled whereas a *join* state can be reached only when *all* of its incoming lines have been enabled. In order to reflect these differences we define the processes *RequireAny* and *RequireAll* that exhibit the necessary behaviours.

$$RequireAny(ls) \;=\; \Box\, e : \varepsilon_{line} (\!|\; ls \;|\!) \bullet e \rightarrow Skip$$

$$RequireAll(ls) \;=\; |||\; e : \varepsilon_{line} (\!|\; ls \;|\!) \bullet e \rightarrow Skip$$

Similarly, in order to reflect the difference between a *fork* state where all outgoing transitions are enabled and other states where only one transition can be chosen, we define the processes *Fork* and *Choice*.

$$\begin{aligned} Fork(ts) \;=\; ||| \; & Transition \mid \theta Transition \in ts \bullet \\ & guard \;\&\; (\mathsf{if}\ (action = null)\ \mathsf{then}\ (\varepsilon_{line}\, line) \rightarrow Skip \\ & \quad \mathsf{else} \\ & \qquad (\varepsilon_{action}\, action) \rightarrow (\varepsilon_{line}\, line) \rightarrow Skip)) \end{aligned}$$

$$\begin{aligned} Choice(ts) \;=\; \Box \; & Transition \mid \theta Transition \in ts \bullet \\ & guard \;\&\; (\mathsf{if}\ (action = null)\ \mathsf{then}\ (\varepsilon_{line}\, line) \rightarrow Skip \\ & \quad \mathsf{else} \\ & \qquad (\varepsilon_{action}\, action) \rightarrow (\varepsilon_{line}\, line) \rightarrow Skip)) \end{aligned}$$

Observe that if there is no action event on the transition line, that is $action = null$, then the corresponding process has no action event.

Our definition of *Choice* merits closer inspection. The intention behind activity graphs is that stimuli given by the user are indicated by actions associated with transitions, and that responses from the system are modelled using action states and activity states. This suggests a model in which, for those transitions whose guards evaluate to true, trigger actions can be deterministically chosen whereas, line events from a transition with no associated action can be chosen non-deterministically.

$$\begin{aligned} & \Box\, Transition \;\mid\; \theta Transition \in ts \wedge guard = \text{true} \wedge action = null \bullet \\ & \qquad (\varepsilon_{line}\, line) \rightarrow Skip \\ & \Box \\ & \sqcap\, Transition \;\mid\; \theta Transition \in ts \wedge guard = \text{true} \wedge action \neq null \bullet \\ & \qquad (\varepsilon_{action}\, action) \rightarrow (\varepsilon_{line}\, line) \rightarrow Skip \end{aligned}$$

In fact, this is precisely the behaviour that our chosen model exhibits when the whole system has been put together and the line events have been hidden.

Before we define the functions ρ_{lines} and ρ_{trans}, consider the graph illustrated in Figure 6.5. Since *line*2 is a self-transition, that is, it loops back directly to the same state, the event $\varepsilon_{line}\, line2$ will not be in the alphabet of the process corresponding to

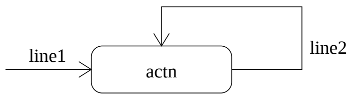

Figure 6.5: An activity looping back to itself.

any other state. This event will therefore not be constrained by synchronisation and can occur at any time. Thus we must treat incoming lines which are part of a self-transition separately. We define the function *loops* to take a state and return the set of all self-transitions, and the function α_{loops} to return the set of all events that correspond to a line contained in a self-transition.

$$\alpha_{loops} : State \rightarrow\!\!\!\!\!+ \mathbb{P}\, Event$$
$$loops : State \rightarrow\!\!\!\!\!+ \mathbb{P}\, Transition$$

$$loops = (\lambda\, state : State \bullet \{tr : state.transitions \mid tr.line \in state.lines\})$$
$$\alpha_{loops} = (\lambda\, state : State \bullet \{tr : loops\ state \bullet \varepsilon_{line}\,(tr.line)\})$$

Given these definitions, and recalling that only *action* states and *activity* states can have direct loops back to themselves in a well-formed graph, we define the function ρ_{lines} which takes a state and returns the process describing the behaviour of all the incoming lines as follows.

$$\rho_{lines} : State \rightarrow\!\!\!\!\!+ CSP$$

$$\begin{aligned} \rho_{lines} = (\lambda\, state : State \bullet \\ &\mathsf{if}\ (type = join)\ \mathsf{then}\ RequireAll(lines) \\ &\mathsf{else} \\ &\quad RequireAny(lines \setminus (\alpha_{loops}\ state))) \end{aligned}$$

Similarly, the function ρ_{trans} takes a state and returns the process that corresponds to the behaviour of its outgoing transitions that are not self loops:

$$\rho_{trans} : State \rightarrow\!\!\!\!\!+ CSP$$

$$\begin{aligned} \rho_{trans} = (\lambda\, state : State \bullet \\ &\mathsf{if}\ (type = finish)\ \mathsf{then}\ Skip \\ &\mathsf{else} \\ &\quad \mathsf{if}\ (type = fork)\ \mathsf{then}\ Fork(transitions) \\ &\quad \mathsf{else} \\ &\quad\quad Choice(transitions \setminus (loops\ state))) \end{aligned}$$

Note that we define the process corresponding to the transitions of a *finish* state to be *Skip*. This avoids having to choose one from an empty set of transitions that would give rise to deadlock.

Our intuition tells us that (with the exception of the start state which may be executed only once) the process corresponding to any given state should be the process corresponding to its incoming lines composed with the process corresponding to its type composed with the process corresponding to its outgoing transitions all composed with itself. In fact, as we will illustrate in Example 9 below, that is too restrictive a definition and can lead to a restricted subset of the desired behaviour.

Example 9 A simple machine.

Consider the simple machine illustrated in Figure 6.6. For each coin inserted the machine does something. The intuition behind the diagram is that arbitrarily many

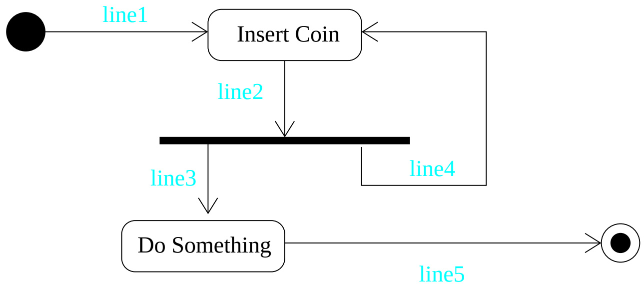

Figure 6.6: An activity graph describing a simple machine.

coins may be inserted. However, if the processes corresponding to the five states are as described below:

$$\begin{array}{rcl}
Start & = & \mu\, X \bullet line.1 \rightarrow Skip \\
Insert & = & \mu\, X \bullet line.1 \rightarrow insert \rightarrow line.2 \rightarrow X \\
 & & \qquad \Box \\
 & & \qquad line.4 \rightarrow insert \rightarrow line.2 \rightarrow X \\
Fork & = & \mu\, X \bullet line.2 \rightarrow (\, line.3 \rightarrow Skip \\
 & & \qquad\qquad\qquad ||| \\
 & & \qquad\qquad\qquad line.4 \rightarrow Skip\,) \,\S\, X \\
DoSomething & = & \mu\, X \bullet line.3 \rightarrow doSomething \rightarrow line.5 \rightarrow X \\
Finish & = & \mu\, X \bullet line.5 \rightarrow X
\end{array}$$

each with their appropriate alphabet, and if *SimpleMachine* is the process obtained by putting these processes in parallel and then hiding the line events, then the refinement check

$$SimpleMachine \quad \sqsubseteq_{\mathcal{T}} \quad Insert$$

where $Insert = insert \rightarrow Insert$ fails. The process *SimpleMachine* can communicate at most three *insert* events before a *doSomething* event must occur. $\Box$

To avoid the error introduced in the above example we must allow multiple synchronous executions of each state. With this in mind, we define as follows the function ρ_{state} which takes a state within a given specification environment and returns the process which models the behaviour of the entire state: the incoming lines, the type of state and the outgoing transitions.

$$\rho_{state} : State \rightarrow\!\!\!\!+ Env \rightarrow\!\!\!\!+ CSP$$

$$
\begin{array}{l}
\rho_{state} \; = \\
\quad (\, \lambda\, state : State \bullet \; (\, \lambda\, env : Env \bullet \\
\qquad \mathsf{if} \;\; (type = start) \;\; \mathsf{then} \;\; \rho_{trans}\, state \\
\qquad \mathsf{else} \\
\qquad\quad \mathbf{let} \\
\qquad\qquad Y = \rho_{lines}\, state \,\text{⨾}\, (Z \;|||\; Y) \\
\qquad\qquad Z = \rho_{type}\, state\; env \;\text{⨾}\; (\, \rho_{trans}\, state \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \Box \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad Choice\,(loops\; state) \;\text{⨾}\; Z\,) \\
\qquad\quad \mathbf{within} \\
\qquad\qquad Y\,)\,)
\end{array}
$$

Observe that the process corresponding to a start state is the only non-recursive process; a start activity can occur only once. The process corresponding to a start state has no component corresponding to incoming lines or the state itself, its only component corresponds to its outgoing transition. For all other states, through interleaving the process with itself after the events corresponding to the incoming lines have occurred, we enable the incoming line events to occur many times before the events corresponding to the state type and the state transitions occur[4], thereby mirroring the intended behaviour of the graph and hence the system. If the state has no self-transitions then the process modelling its behaviour is the process corresponding to incoming lines followed by the process itself interleaved with the process corresponding to the type of the state followed by the process corresponding to the outgoing transitions. If the state has self-transitions it is the same process but after the process corresponding to the state has occurred there can be arbitrarily many self-loops followed by the process corresponding to the state once more before a transition to another state occurs.

In Example 10 below we re-visit the lift of Example 8. We present the simple activity graph corresponding to the activity state “Open and Close” and determine the process corresponding to this graph.

Example 10 Opening and closing the lift doors.

The activity graph describing fully the activity state “Open and Close” of the activity graph “Lift” as illustrated in Figure 6.4 of Example 8 is given in Figure 6.7 below. The lift doors simply open and then close. We have labelled the lines in our graph and we identify the actions *open* and *close* within the graph. Furthermore we identify *OpenAndClose*, the ActivityLabel corresponding to the graph, and also the necessary events.

[4]This corresponds closely to Petri nets [Pet81] in which each place can contain multiple tokens.

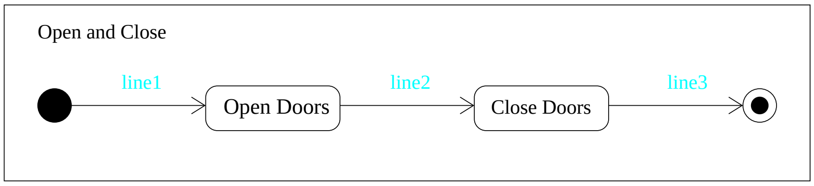

Figure 6.7: Opening and closing a lift's doors.

$$
\begin{array}{|l}
line1, line2, line3 : Line \\
open, close : Action \\
OpenAndClose : ActivityLabel \\
line.1, line.2, line.3, open, close : Event
\end{array}
$$

Finally we introduce the functions that identify the actions and lines with their associated events.

$$
\begin{aligned}
\varepsilon_{line} &= \{line1 \mapsto line.1, line2 \mapsto line.2, line3 \mapsto line.3\} \\
\varepsilon_{action} &= \{openDoor \mapsto open, closeDoor \mapsto close\}
\end{aligned}
$$

Given the above definitions and recalling that a *SimpleTranstion* is one with no associated guard or action, we model the transitions in the above graph as follows:

$$
\begin{aligned}
Trans1 &\mathrel{\widehat=} [\, SimpleTransition \mid line = line1 \,] \\
Trans2 &\mathrel{\widehat=} [\, SimpleTransition \mid line = line2 \,] \\
Trans3 &\mathrel{\widehat=} [\, SimpleTransition \mid line = line3 \,]
\end{aligned}
$$

Hence we see easily that the formal descriptions of the four states within the activity graph illustrated in Figure 6.7 are those presented below.

Start

$$
\begin{array}{|l}
WellFormedState \\
\hline
type = start \;\wedge \\
transitions = \{Trans1\}
\end{array}
$$

Finish

$$
\begin{array}{|l}
WellFormedState \\
\hline
type = finish \;\wedge \\
lines = \{line3\}
\end{array}
$$

OpenDoors

$$
\begin{array}{|l}
WellFormedState \\
\hline
type = action\langle\!\langle openDoor \rangle\!\rangle \;\wedge \\
lines = \{line1\} \;\wedge \\
transitions = \{Trans2\}
\end{array}
$$

CloseDoors

$$
\begin{array}{|l}
WellFormedState \\
\hline
type = action\langle\!\langle closeDoor \rangle\!\rangle \;\wedge \\
lines = \{line2\} \;\wedge \\
transitions = \{Trans3\}
\end{array}
$$

Recall that we need not state explicitly that the start state has no incoming lines and that the finish state has no outgoing transitions since those constraints are already included in the definition of a *WellFormedState*.

The environment describing the graph illustrated in Figure 6.7 can then be formally modelled thus:

$$env : Env$$

$$env \;=\; \{ OpenAndClose \mapsto states \langle\!\langle \{ Start, OpenDoors, CloseDoors, Finish \} \rangle\!\rangle$$

Having formally modelled our given activity graph we can now determine the corresponding CSP process. We reason as follows:

$$toProcess\ \ OpenAndClose\ \ env$$

$$= \qquad [\text{def}^{\,n} \text{ of } toProcess]$$

$$(\| \; state : states^{\sim} (env\ \ OpenAndClose) \; \bullet \; [\, \alpha_{state}\; state\; env \,]\; \rho_{state}\; state\; env) \setminus \bigcup (\, \{ st : states^{\sim} (env\ \ OpenAndClose) \bullet \varepsilon_{line} (\!| \; st.lines \; |\!) \} \,)$$

$$= \qquad [\text{def}^{\,n}\text{s of } states,\ \alpha_{state},\ \rho_{state} \text{ and } \varepsilon_{line}]$$

$$(\, line.1 \rightarrow Skip \underset{\{line.1\}}{\|} (\,(\,\mu\, X \bullet line.1 \rightarrow (\, open \rightarrow line.2 \rightarrow Skip \;|||\; X \,)\,) \underset{\{line.2\}}{\|} (\,(\,\mu\, X \bullet line.2 \rightarrow (\, close \rightarrow line.3 \rightarrow Skip \;|||\; X \,)\,) \underset{\{line.3\}}{\|} (\,(\,\mu\, X \bullet line.3 \rightarrow X \,)\,)\,)\,)\,) \;\; \setminus \;\; \{line.1, line.2, line.3\}$$

$$= \qquad [\text{def}^{\,n}\text{s of } \| \text{ and } \setminus]$$

$$open \rightarrow close \rightarrow Stop$$

We see that the process that describes the activity graph “Open and Close” illustrated in Figure 6.7 is, as we would expect, ($open \rightarrow close \rightarrow Stop$).

□

6.3 Analysis

In order to compare the final class description of a model with the activity graphs constructed during the development process we first translate the final class description to its corresponding process. We rely on the work of others—discussed further in Section 7.5—to make the initial step from the final class description to an abstract data type modelling that system. Applying the function $process_{io}$ as defined in Section 5.6 to the resultant data type then yields the corresponding process. We can apply the methods presented in Sections 6.1 and 6.2 to obtain the processes corresponding to any activity diagrams.

Having translated both the final class description and any activity graphs to their corresponding CSP processes, it is simply a question of refinement to confirm that the final object model is consistent with its specification. But the nature of the refinement check we do is not always immediately obvious. Consider, for instance, the activity graph illustrated in Figure 6.8 below. Does this diagram mean that whenever *methodA* is

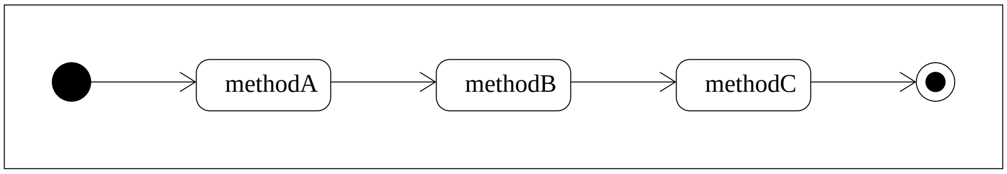

Figure 6.8: A simple activity graph

called, *methodB* must be called and then *methodC* must be called? Or does it mean that whenever *methodA* is called *methodC* cannot be called before *methodB* has been called? Or does it simply mean that *methodA* followed by *methodB* followed by *methodC* is a possible sequence of interaction?

At present the information content of each diagram is conveyed (possibly tacitly) by its context within the specification. But this information content dictates the nature of the refinement check we make: do we use the traces model which simply checks that the behaviour described in the diagram is a possible sequence of interaction or do we perform our refinement check using the failures model thereby checking availability information? Before we are in a position to automate the process of verifying the consistency of object models we must settle the issue of the information content of each of the diagrams.

It seems most appropriate to take activity graphs at face value; to assume that they record possible sequences of interaction and to infer no information about availability. Hence we opt for the traces model. Furthermore, since our CSP model of the final class description—determined by applying the function $process_{io}$ to the Z description of the model—is correct up to the singleton failures semantic model, and since refinement within the singleton failures model implies refinement within the traces model, the traces model is a suitable choice.

Definition 1 Consistency within a UML specification.

If *ClassDesc* is an abstract data type modelling the final class description of a UML specification and if *GraphLabel* is the name of an activity graph constructed within the environment *env* within the same specification, then we say that the final class description is *consistent* with the activity graph if the following refinement check holds.

$$process_{io}\ ClassDesc \quad \sqsubseteq_{\mathcal{T}} \quad toProcess\ GraphLabel\ env$$

□

Automation

Verification of consistency within a UML specification, as illustrated in Figure 6.9, requires four steps:

- Translation of the UML class description to an equivalent model expressed as an abstract data type in Z;
- Translation of the Z specification of the final class description to its corresponding CSP process;
- Translation of any activity graphs constructed during development to their corresponding processes;
- Verification of consistency between the processes corresponding to activity graphs and the process corresponding to the final class description.

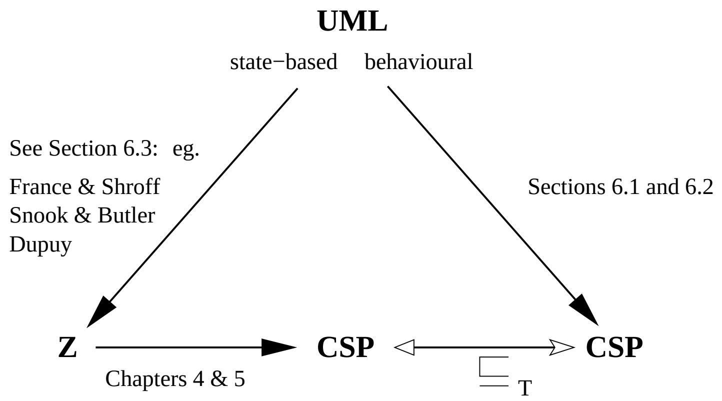

Figure 6.9: Verification of consistency

The steps towards automating this process are almost, but not quite, complete. As we discuss further in Section 7.5, the step from the class model to its equivalent Z specification has not yet been fully automated. The tool RoZ [DLCP00] considers only class diagrams annotated with OCL [JBW99] or Z rather than extracting information from state diagrams. U2B considers both annotated class diagrams and state diagrams; however this tool translates to B rather than to Z and, perhaps more significantly, can handle only a strict subset of UML. Provided the first step yields an abstract data type in Z the second step, translating this to its corresponding CSP process, should be easily automatable. We simply need to apply the function $process_{io}$ as defined in Chapter 5.

Within the third step, obtaining a process corresponding to an activity graph *that is suitable for providing as an argument to the model-checker FDR [For97]* is a little complicated. Whilst the function mapping activity graphs to processes defined in Section 6.2 is entirely correct it cannot be used in practice when we come to analysing the system using a model checker. The interleaving of the process corresponding to each state with itself causes the system to be infinite state. This interleaving is, however, necessary since we must be able to handle multiple, possibly synchronous, executions of each state.

To ensure a finite state space we must, in Petri net terms, put an upper limit on the number of tokens allowed in any place; that is, for each action state and activity state we must only allow a finite number of events corresponding to an incoming line to occur before an event corresponding to an outgoing transition occurs.

We extend the definition of *State* to incorporate the variables *dynamicMultiplity* and *isDynamic* as defined in the official documentation [OMG99]. The Boolean *isDynamic* is *True* for action states and activity states in which the state's actions can be executed concurrently. If this Boolean is *True* then the integer *dynamicMultiplicity* limits the number of parallel executions of the actions of the state. If the Boolean *isDynamic* is *False*, then *dynamicMultiplicity* is ignored.

$$
\begin{array}{|l}
\hline
\textit{PracticalState} \\
\hline
\textit{WellFormedState} \\
\textit{isDynamic} : \textit{Boolean} \\
\textit{dynamicMultiplicity} : \mathbb{N} \\
\hline
\textit{type} \in \{\textit{start}, \textit{finish}, \textit{merge}, \textit{decision}, \textit{fork}, \textit{join}\} \Rightarrow \textit{isDynamic} = \textit{False} \\
\hline
\end{array}
$$

We define the functions *inComing* and *outGoing* which take a *PracticalState* and return respectively the set of events corresponding to the incoming non-looped lines and the set of events corresponding to the lines within the non-looped outgoing transitions. Given these definitions, we define the functions *single* and *constraint* which respectively return the processes which limits the number of parallel executions of each state to one and to the bound set by the variable *dynamicMultiplicity*.

$$
\begin{array}{|l}
inComing : PracticalState \rightarrow \mathbb{P}\, Event \\
outGoing : PracticalState \rightarrow \mathbb{P}\, Event \\
single : PracticalState \rightarrow Process \\
constraint : PracticalState \rightarrow Process \\
\hline
\forall\, state : PracticalState \bullet \\
\quad inComing\ state \ = \ \varepsilon_{line} (\!|\ state.lines\ |\!) \ \setminus \ \alpha_{loops}\, state \\
\quad outGoing\ state \ = \ \{tr : state.transitions \bullet \varepsilon_{line}\,(tr.line)\} \ \setminus \ \alpha_{loops}\, state \\
\quad single\ state \ = \ \mu\, X \bullet \Box\, in : inComing\ state \bullet in \rightarrow \\
\qquad\qquad\qquad\qquad \Box\, out : outGoing\ state \bullet out \rightarrow X \\
\quad constraint\ state \ = \\
\qquad \mathbf{let} \\
\qquad\quad Con(n) = \\
\qquad\qquad (n < state.dynamicMultiplicity)\ \& \\
\qquad\qquad\quad \Box\, in : inComing\ state \bullet in \rightarrow Con(n+1) \\
\qquad\qquad \Box \\
\qquad\qquad \Box\, out : outGoing\ state \bullet out \rightarrow Con(n-1) \\
\qquad \mathbf{within} \\
\qquad\quad Con(0)
\end{array}
$$

Using these definitions, the process that, *for practical purposes*, we should use to model the behaviour of each state is given below.

$$
\begin{array}{|l}
\rho_{practical\_state} : PracticalState \rightarrow Env \rightarrow Process \\
\hline
\rho_{practical\_state} \ = \ (\,\lambda\, state : State \bullet \ (\,\lambda\, env : Env \bullet \\
\quad \mathsf{if}\ (isDynamic = False)\ \mathsf{then} \\
\qquad \rho_{state}\ state\ env \ |[\, (inComing\ state) \cup (outGoing\ state)\,]|\ single\ state \\
\quad \mathsf{else} \\
\qquad \rho_{state}\ state\ env \ |[\, (inComing\ state) \cup (outGoing\ state)\,]|\ constraint\ state
\end{array}
$$

Hence it follows that, for practical purposes, our semantic function should be

$$
\begin{array}{|l}
toProcessP : ActivityLabel \rightarrow EnvP \rightarrow Process \\
\hline
\forall\, env : EnvP;\ actL : ActivityLabel \bullet \\
\quad toProcessP\ actL\ env \ = \\
\qquad (\|\ stateP : statesP^{\sim}\,(env\ actL) \ \bullet \\
\qquad\quad [\,\alpha_{state}\ stateP\ env\,]\ \rho_{practical\_state}\ stateP\ env) \\
\qquad\qquad \setminus\ \bigcup(\,\{stp : statesP^{\sim}\,(env\ actL) \bullet \varepsilon_{line} (\!|\ stp.lines\ |\!)\}\,)
\end{array}
$$

where a $EnvP$ maps activity labels onto "practical" graphs and where practical graphs are a collection of $PracticalState$s.

$$EnvP \ == \ ActivityLabel \rightarrow GraphP$$

$$GraphP \ ::= \ statesP \langle\!\langle\ \mathbb{F}\, PracticalState\ \rangle\!\rangle$$

Having achieved the first three steps and automatically generated processes corresponding to the class description and to activity graphs constructed during development, the fourth and final step, automatically verifying consistency using FDR should be relatively easy. We simply need to verify that the process corresponding to the class description is refined, within the traces model, by the processes corresponding to all activity graphs constructed during its development.

Before concluding this chapter with a simple case study we make an important observation. Our intention is that these checks are done automatically and behind the scenes; that the user need have no knowledge of Z or CSP. And this goal is easily achievable. The user will get one of two responses: either their specification is consistent; or the tool will identify firstly a sequence of operations that the final class description cannot perform and secondly the activity graph that claimed that it should be possible. The sequence of operations corresponds to the trace identified by FDR establishing that refinement does not hold. It is then up to the user to decide which of their initial description and the relevant activity graph is incorrect.

6.4 Case study: a ticket machine

In this section we illustrate how a final class description might be verified for consistency with diagrams constructed during its development. We will present a seemingly sensible model of a system and show how an error is detected by comparing it with an activity diagram that might also have been constructed during development.

Class description

Consider the superficially plausible class description of a simple ticket machine presented in Figure 6.10 and suppose that these fragments had been proposed during development. The class diagram describes a single class with four data members, *coins*, *tickets*, *toReturn* and *toIssue*, all integers. It also introduces five methods, *insertCoin*, *pressRed*, *pressGreen*, *issueTicket* and *returnCoin*, but reveals only that they expect no arguments. Suppose also that in the accompanying text the variables were described as follows:

- *coins* records the number of coins that have been inserted but neither allocated a ticket nor put in the tray to be returned;
- *tickets* records the number of tickets remaining in the machine;
- *toReturn* records the number of coins that have been put in the tray to be returned;
- *toIssue* records the number of tickets that have been allocated for issuing.

The accompanying state diagram describes the methods in greater detail, presenting information about their availability and their effect upon the state. Each of the boxes contains a predicate written in OCL [JBW99] describing a constraint upon the combination of values taken by the data members of the state. Within this notation, the expressions `x += n` and `x -= n` are respectively equivalent to the expressions $x' = x + n$

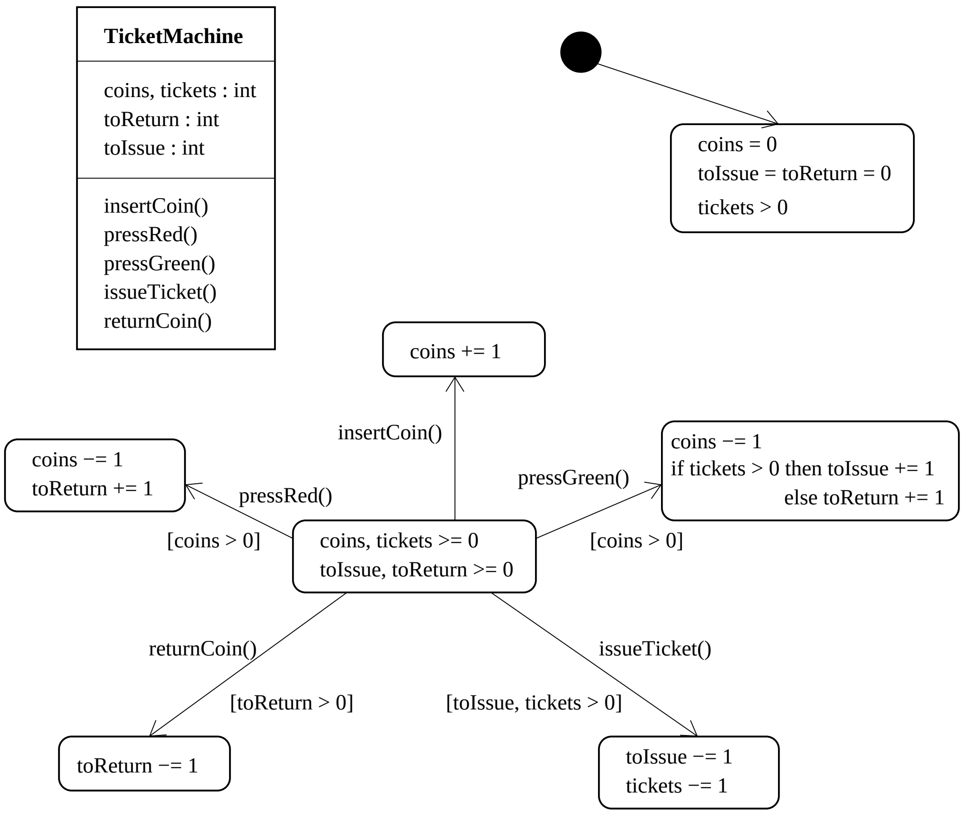

Figure 6.10: Fragments of a class description.

and $x' = x - n$ within Z. We note that no operation can occur unless all the state variables are non-negative. Furthermore *pressRed* and *pressGreen* can occur only if *coins* is strictly positive; *returnCoin* can only occur if *toReturn* is strictly positive; and *issueTicket* can occur only if both *tickets* and *toIssue* are strictly positive.

Initially the variable *tickets* must be strictly positive and the other variables must be zero. If *insertCoin* occurs then *coins* is incremented; this variable is subsequently decremented if either *pressRed* or *pressGreen* occur. If the former happens then *toReturn* is incremented. If the latter happens then if *tickets* is non-negative then *toIssue* is incremented. Otherwise *toReturn* is incremented. When *returnCoin* occurs the variable *toReturn* is decremented and when *issueTicket* occurs both the variables *tickets* and *toIssue* are decremented.

At first glance this hypothetical model seems to be a sensible one. However, there is a serious flaw in it that will be identified when we compare it with the associated activity diagram. The error arises from the definition of the state variable *tickets*. The specification states that this variable "records the number of tickets remaining in the machine". Hence, in this specification, *tickets* is decremented when a ticket is issued, that is the operation *issueTicket* occurs.

In a correct specification, the variable *tickets* should denote the number of tickets that *have not yet been allocated*. Hence it should be decremented at the same time as the *toIssue* variable is incremented within the *pressGreen* operation. We will see where this loose definition leads to problems when we come to comparing the process corresponding to the class description and the process corresponding to the activity graph.

Z description

As we have observed it is not our intention to provide a formal translation to Z from the state-based notations within UML: we leave this work to others. For the time being it is sufficient to recognise that the following Z description matches perfectly the class description of a ticket machine as presented in Figure 6.10.

TicketMachine

$coins, tickets : \mathbb{N}$
$toReturn, toIssue : \mathbb{N}$

Init

$TicketMachine'$

$coins' = 0 \ \wedge \ tickets' > 0$
$toReturn' = 0 \ \wedge \ toIssue' = 0$

InsertCoin

$\Delta\, TicketMachine$

$coins' = coins + 1$
$tickets' = tickets$
$toReturn' = toReturn$
$toIssue' = toIssue$

PressRed

$\Delta\, TicketMachine$

$coins' = coins - 1$
$tickets' = tickets$
$toReturn' = toReturn + 1$
$toIssue' = toIssue$

IssueTicket

$\Delta\, TicketMachine$

$coins' = coins$
$tickets' = tickets - 1$
$toReturn' = toReturn$
$toIssue' = toIssue - 1$

ReturnCoin

$\Delta\, TicketMachine$

$coins' = coins$
$tickets' = tickets$
$toReturn' = toReturn - 1$
$toIssue' = toIssue$

PressGreen

$\Delta\, TicketMachine$

$coins' = coins - 1$
$tickets' = tickets$
$tickets > 0 \Rightarrow ((toIssue' = toIssue + 1) \wedge (toReturn' = toReturn))$
$tickets = 0 \Rightarrow ((toIssue' = toIssue) \wedge (toReturn' = toReturn + 1))$

Applying the function $process_{io}$ as defined in Chapter 5 to the corresponding data type—and ignoring inputs and outputs since these are all null in our simple example—we see that the process that describes the behaviour of the ticket machine is as illustrated in Figure 6.10, that is the process corresponding to the data type *TicketMachine* is as described below. We will name this process *ClassDescProcess*.

$$
\begin{array}{l}
ClassDescProcess \;=\\
\quad \textbf{let}\\
\qquad P(c, t, ret, iss) \;=\\
\qquad\quad insert \rightarrow P(c+1, t, ret, iss)\\
\qquad\quad \Box\\
\qquad\quad c > 0 \;\&\; red \rightarrow P(c-1, t, ret+1, iss)\\
\qquad\quad \Box\\
\qquad\quad c > 0 \;\&\; green \rightarrow (\ \mathsf{if}\ \ t > 0\ \mathsf{then}\ \ P(c-1, t, ret, iss+1)\\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ P(c-1, t, ret+1, iss)\,)\\
\qquad\quad \Box\\
\qquad\quad (t > 0 \wedge iss > 0) \;\&\; issue \rightarrow P(c, t-1, ret, iss-1)\\
\qquad\quad \Box\\
\qquad\quad ret > 0 \;\&\; return \rightarrow P(c, t, ret-1, iss)\\
\quad \textbf{within}\\
\qquad \sqcap\, t : \{1 \,..\, T\} \bullet P(0, t, 0, 0)
\end{array}
$$

The integer T is the machine's ticket capacity. On initialisation it might be filled with any number of tickets not exceeding that capacity. Initially no coins have been inserted, no coins need returning and there are no tickets waiting to be issued.

Activity graph

Next suppose that the following activity graph was constructed during the development process. We observe that a coin can be inserted at any time, and that for each coin

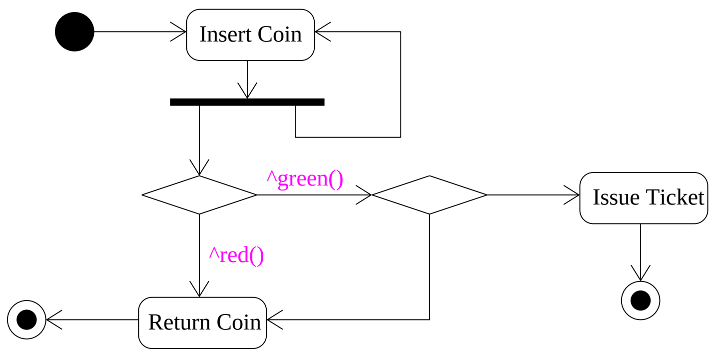

Figure 6.11: An activity graph describing the ticket machine.

that has been inserted either the red button or the green button can be pressed. If the red button is pressed then a coin will be returned. If the green button is pressed then the machine will non-deterministically choose either to issue a ticket or to return a coin. We are supposing that when this graph was constructed early on in the development phase, at a time when the conditions determining the outcome of pressing the green button—whether a ticket should be issued or a coin returned—had not yet been formally identified.

Reasoning as described earlier on in this chapter, and as illustrated in Example 10 we see that the process corresponding to this activity graph is the process *GraphProcess* described below.

$$
\begin{array}{lll}
GraphProcess & = & insert \rightarrow (\, GraphProcess \\
& & \qquad\qquad\quad ||| \\
& & \qquad\qquad\quad (\, red \rightarrow return \rightarrow Skip \\
& & \qquad\qquad\quad\;\; \Box \\
& & \qquad\qquad\quad\;\; green \rightarrow (\, issue \rightarrow Skip \\
& & \qquad\qquad\qquad\qquad\qquad\;\; \sqcap \\
& & \qquad\qquad\qquad\qquad\qquad\;\; return \rightarrow Skip\,)\,)\,)
\end{array}
$$

Verifying consistency

For simplicity we assume that the ticket machine has only one ticket; we set T to 1. Our definition of *consistency* states that the final class description is a correct implementation of the given activity diagram provided the following refinement check holds.

$$ClassDescProcess \sqsubseteq_{\mathcal{T}} GraphProcess$$

We see immediately that this is not the case. The sequence

$$\langle insert, insert, green, green, issue, return\rangle$$

is (correctly) a possible trace of the process corresponding to the activity graph of the ticket machine that contains only one ticket. However, it is not a trace of the process corresponding to the class description of the ticket machine with only one ticket. For that process, after the trace

$$\langle insert, insert, green, green, issue\rangle$$

the only event that is now available is the *insert* event. Following the progression of states illustrated in Figure 6.12 we see that after this sequence of interaction the ticket machine is in the state $P(0,0,0,1)$: it has no coins unaccounted for; no tickets remaining; no coins due to be returned; but a ticket that should be issued.

After the coins had been inserted it was in the state $P(2,1,0,0)$. After *green* had occurred a correct implementation would have left it in the state $P(1,0,1,0)$ rather than the state $P(1,1,1,0)$ indicating that there were no remaining tickets unaccounted for. This would have meant that after the next *green* the correct implementation would leave the system in the state $P(0,0,1,1)$, with one ticket to be issued and a single coin to be returned.

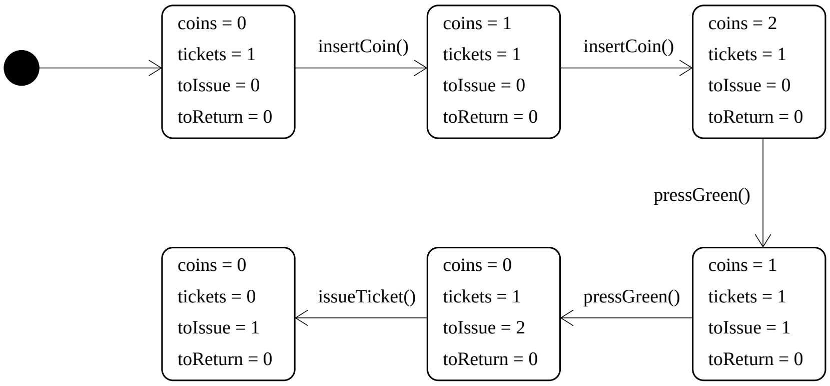

Figure 6.12: A state diagram for the incorrect implementation

Chapter 7

Conclusions

In this final chapter we reflect on the ideas presented in the previous chapters: we summarise the work and draw some conclusions; we discuss important issues exposed during the research; we make comparisons with related work; and we propose interesting avenues for future research.

7.1 Summary

In this thesis we have provided a framework for comparing state-based and behavioural models of the same system. The models in question might be graphical or algebraic. Furthermore, by considering two distinct semantic interpretations, we enable this comparison to be made in an environment in which operations called outside their domain of definition either might be blocked or might occur but without any guarantees concerning the outcome. In the course of our work we have defined and investigated the properties of a new semantic model for Communicating Sequential Processes, as well as giving a formal semantics to activity graphs.

We began by proposing the singleton failures semantic model for CSP: a model whose information content lies between that of the traces model and the stable failures model. This new model records possible sequences of interaction as well as the availability of events after each such sequence. Simply taking a projection of the stable failures model, considering only those trace-refusal pairs in which the cardinality of the refusal set is at most one, does not yield a congruence: we cannot express this subset of the failures of a process after hiding in terms of the subset of the failures of the original process. We therefore defined the hiding operator to be as close as possible to the intended meaning whilst ensuring that the resulting model defines a congruence.

In the stable failures model, a failure of a process P makes a contribution to the semantic set of $P \setminus A$ only if the refusal set contains every element from A. In the singleton failures model we can check only that each event from A can be refused individually after the relevant trace. We investigated the properties of the hiding operator within the singleton failures model: we showed that it is not distributive with respect to the internal choice operator; that it is not symmetric; and that hiding one set of events and then another is not equivalent to hiding the union of the two sets. More positively, we proved that *at the outermost level* hiding preserves refinement.

Defining a *basic process* to be one involving only the process *Stop*, the prefix operator, the parallel operator and the choice operators, we proved that all algebraic properties of basic processes that hold within the stable failures model hold also within the singleton failures model. This result followed from the property that *for basic processes* the singleton failures semantic model is a projection of the stable failures model. We showed how singleton failures refinement of basic processes could be expressed in terms of stable failures refinement and hence we derived a means for determining automatically, using the model-checker FDR [For97], whether one process is refined by another within the singleton failures model.

The purpose of Chapter 4 was to explore the relationship between data refinement and process refinement. To facilitate this exploration we adopted a very simple model for our data types: one with no explicit notion of input and output. We considered two semantic models: the standard non-blocking model [WD96] in which operations can be called outside their preconditions with undefined results; and the less commonly used non-blocking model [BD99] in which operations are blocked outside their preconditions. We described the simulation rules for each of these models.

We presented two translations from data types to CSP processes, one corresponding to the blocking model and the other corresponding to the non-blocking model. Using these translations we observed that, in either context, data refinement is not equivalent to stable failures refinement. Having explored the difference between the information content of the relational semantics of data types and of the stable failures semantics, we went on to prove that data refinement is equivalent to singleton failures refinement.

In Chapter 5 we complicated matters by considering *communicating data types*, data types with an explicit notion of input and output. We identified a correlation between these communicating data types and the simple data types of Chapter 4 and exploited this correlation to derive, and establish the soundness and joint completeness of, the simulation rules for communicating data types. Again we considered both the blocking and non-blocking models.

Having presented translations from communicating data types to CSP processes, one corresponding to the blocking model and the other corresponding to the non-blocking model, we showed that results established for simple data types and their corresponding processes do not automatically follow for communicating data types and their corresponding processes. These inconsistencies stem from the different ways in which the relational semantics and the behavioural semantics handle the choice of output. We proved that, for systems with non-deterministic outputs, data refinement is not equivalent to singleton failures refinement: in order to verify data refinement we need to do an additional check that determines whether an operation is being refused because there is *no* possible output or because there is a *choice* of possible outputs. With this additional check, the singleton failures model can be used to determine whether one data type is refined by another.

In Chapter 6 we showed how the theory presented in the previous chapters might be used to reason about object models developed using The Unified Modeling Language. We presented a formal semantics for activity graphs in terms of CSP. Our graph syntax was chosen to match closely the Rose interchange format [Rat] thereby facilitating automation. We alluded to the work of others (to be discussed further in Section 7.5) in formalising final class descriptions—class diagrams and annotated state diagrams—and illustrated how this work, along with our formalisation of activity graphs, might be used to formally verify consistency of object models. The translation from state-based to event-based models enables comparison between the final class description of an object model and behavioural descriptions constructed during its development. In the example presented, this comparison exposed a flaw in the final class description that had arisen from a loose definition of one of the state variables.

7.2 Interesting results

Aside from providing a means for comparing state-based and behavioural models, we have made two significant contributions to the work that has already been done in this field. Firstly we have disproved an important and fallacious result that had been rippling through the literature: we provided an example that showed that data refinement is not equivalent to stable failures refinement. We did not simply leave this negative result: we went on to propose an alternative semantic model for CSP and to prove that refinement within this model is equivalent to data refinement.

Our second interesting finding is that it is not sufficient to prove results for simple data types—data types with all inputs *wound* into an initialisation operation and all outputs wound into a finalisation operation—and assume that the corresponding results hold for communicating data types. We proved that, for simple data types, data refinement is equivalent to singleton failures refinement but that this is not the case for communicating data types. For data types with non-deterministic outputs we need to do an additional check. We need to distinguish between processes that refuse an event because there is a *choice* of outputs and processes that refuse an event because there is *no* output.

7.3 Related work: a comparison of the information content of simulation rules

As we observed in Section 1.2, combining state-based and behavioural formalisms is by no means a new area of research. Josephs [Jos88] adopted a state-based approach to the refinement and specification of non-divergent CSP processes; Morgan [Mor90] defined a correspondence between action systems [BKS83] and CSP by expressing the traces, failures and divergences of CSP processes as weakest precondition formulae [Dij76]; and He [He89], like Josephs, expressed CSP processes as labelled transition systems and presented sets of sound and complete simulation rules. In this section we discuss these three approaches and in particular we compare the information content of the simulation rules.

Action systems

The *actions systems* formalism of Back and Kurki-Suonio [BKS83] is a state-based approach to distributed computing combining a state, an initialisation command, and some labelled actions on the state. Here an action is a guarded command as defined by Dijkstra [Dij76]: a guard is a predicate that can be evaluated to either true or false and a command is a program fragment. An action is *enabled* precisely when its guard evaluates to true. In the execution of an action system first the initialisation is executed: its command is performed. Then, chosen enabled actions are repeatedly executed. If no action is enabled then the systems *deadlocks*; if the command of an enabled action aborts then the system *diverges*.

Rather than defining formally the syntax and semantic of action systems, we give a simple example. For a more detailed description, see [But92]. We give an action systems description of a yo-yo. The yo-yo goes down and up repeatedly until it starts spinning. After that no more *Up* or *Down* actions can occur; the system deadlocks.

Example 11 An action systems description of a yo-yo.

Given the type *Position* that has three values *up*, *down* and *spin* we model a yo-yo as follows.

$$YoYo \quad = \quad \left(\begin{array}{ll} \mathbf{var} & pos : Position \\ \mathbf{initially} & pos := up \\ Down :- & pos = up \;\rightarrow\; pos :\in \{down, spin\} \\ Up :- & pos = down \;\rightarrow\; pos :\in \{up, spin\} \end{array} \right)$$

The yo-yo has a single state variable *pos* that records its position. Either it is in the up position or the down position or it has started spinning.

Initially the yo-yo is assigned the position *up* and whenever it is in the *up* position, and hence the guard $pos = up$ evaluates to true, the action *Down* is enabled and can be executed. Similarly, whenever it is in the *down* position, and hence the guard $pos = down$ evaluates to true, the action *Up* is enabled and can be executed. If the position of the yo-yo has become the state *spin* neither operation is enabled and the system deadlocks.

The execution of action *Up* non-deterministically sets the position either to *up* or to *spin*. Similarly the execution of action *Down* non-deterministically sets the position either to *down* or to *spin*.

□

The action systems formalism can be given a weakest precondition semantics [Mor88, Nel89]. Given command *com* and postcondition *post* the weakest precondition of command *com* with respect to postcondition *post*, written $wp(com, post)$, is the set of all initial states from which the execution of command *com* guarantees termination in a state satisfying postcondition *post*. Similarly we can calculate the weakest precondition of a guarded command. Given guard G, command *com* and postcondition *post* the weakest precondition of guarded command $G \rightarrow com$ with respect to postcondition post is defined as follows:

$$wp(G \rightarrow com, post) \quad \widehat{=} \quad G \Rightarrow wp(com, post).$$

Statement S satisfies a specification $(pre, post)$ precisely when $pre \Rightarrow wp(S, post)$, and statement S_C is said to *refine* statement S_A if any specification satisfied by S_A is also satisfied by S_C.

Morgan [Mor90] exploits the weakest precondition semantics enabling action systems to be seen as CSP processes by expressing the traces, failures and divergences of CSP processes as weakest precondition formulae. Butler [But92] then extends this to the infinite-traces model within CSP. They reason in terms of the *conjugate weakest precondition*, written $\overline{wp}\,(com, post)$, and defined as follows:

$$\overline{wp}\,(com, post) \quad \widehat{=} \quad \neg\; wp(com, \neg\, post).$$

The conjugate weakest precondition identifies those initial states from which the execution of command *com* might result in termination in a state satisfying postcondition *post*. Given this definition the set of *traces* of an action system with initialisation A_i and set of labelled operations $\mathcal{A}$ is defined

$$\{\, tr : \operatorname{seq} \mathcal{A} \;\mid\; \overline{wp}\,(A_i;\; tr, \mathrm{true}) \,\}$$

where $\overline{wp}\,(\langle\rangle, \mathrm{true})$ is equivalent to $\overline{wp}\,(Skip, \mathrm{true})$ and where $\overline{wp}\,(act1;\; act2, \mathrm{true})$ is equivalent to $\overline{wp}\,(act1, \overline{wp}\,(act2, \mathrm{true}))$.

Similarly the *failures* of an action system can be defined using weakest precondition formulae as follows:

$$\{\, tr : \operatorname{seq} \mathcal{A};\; ref : \mathbb{P}\, \mathcal{A} \;\mid\; \overline{wp}\,(A_i;\; tr, \neg\, gd(ref)) \,\}$$

where $gd(ref)$ is the disjunction of the guards of the actions in the set *ref* and the guard $gd(A_x)$ of an action A_x is given by $\overline{wp}\,(A_x, \mathrm{true})$.

Finally, the divergences of an action system are defined as follows:

$$\{\, tr : \operatorname{seq} \mathcal{A} \;\mid\; \overline{wp}\,(A_i;\; tr, \mathrm{false}) \,\}$$

We know that the traces, failures and divergences are sufficient to characterise a CSP process; having identified these sets for an action system we can write down the CSP process that corresponds to the action system. We illustrate this in Example 12 below.

Example 12 A CSP description of a yo-yo.

In this example we determine the traces, failures and divergences of the action system *YoYo* as defined in Example 11 and use these to write down the CSP process that describes the behaviour of *YoYo*.

We see that the traces of *YoYo* are sequences of names of actions beginning with *Down* and then alternating between *Up* and *Down*.

$$\{\, \langle\rangle, \langle Down \rangle,\; \langle Down, Up \rangle,\; \langle Down, Up, Down \rangle,\; \ldots \,\}$$

The yo-yo has no divergences. If Up and $Down$ are the only known actions then the failures of $YoYo$ are given by:

$$\begin{array}{l}
\{\, (\langle\rangle, \emptyset),\ (\langle\rangle, \{\, Up\}) \,\} \\
\cup \\
\{\, (\langle Down\rangle, \emptyset),\ (\langle Down\rangle, \{\, Up\}),\ (\langle Down\rangle, \{Down\}),\ (\langle Down\rangle, \{\, Up, Down\}) \,\} \\
\cup \\
\{\, (\langle Down, Up\rangle, \emptyset),\ (\langle Down, Up\rangle, \{\, Up\}), \\
\quad (\langle Down, Up\rangle, \{Down\}),\ (\langle Down, Up\rangle, \{\, Up, Down\}) \,\} \\
\cup \\
\{\, (\langle Down, Up, Down\rangle, \emptyset),\ (\langle Down, Up, Down\rangle, \{\, Up\}), \\
\quad (\langle Down, Up, Down\rangle, \{Down\}),\ (\langle Down, Up, Down\rangle, \{\, Up, Down\}) \,\} \\
\vdots
\end{array}$$

We see then that the process corresponding to the action system $YoYo$ is equivalent to the process $YoYoProc$ given below.

$$\begin{array}{lcll}
YoYoProc & = & \mu X \ \bullet \ Down \rightarrow (& Stop \\
 & & & \sqcap \\
 & & & Up \rightarrow (\, Stop \sqcap X \,)\,)
\end{array}$$

□

Woodcock and Morgan [WM90] introduce the notion of data refinement on action systems; refinement can be established inductively using a set of simulation rules that are both sound and complete. Given action systems A and C, relation I from the state space of A to the state space of C, and any predicate β such that no element a in the state space of A is free in β the *forwards simulation rules* are given below:

1. $\exists\, c \bullet \overline{wp}\ (C_i, \beta)) \ \Rightarrow\ \exists\, a \bullet \overline{wp}\ (A_i, (\exists\, c \bullet I \wedge \beta))$
2. For all actions x,
   $\exists\, c \bullet I \wedge \overline{wp}\ (C_x, \beta) \ \Rightarrow\ \overline{wp}\ (A_x, (\exists\, c \bullet I \wedge \beta))$
3. For all actions x,
   $\exists\, a \bullet I \wedge gd\ A_x \ \Rightarrow\ gd\ C_x.$

Similarly, for any predicate α such that no element c in the state space of C is free in α, the *backwards simulation rules* are as given below:

1. $\exists\, c \bullet \overline{wp}\ (C_i, (\exists\, a \bullet I \wedge \alpha)) \ \Rightarrow\ \exists\, a \bullet \overline{wp}\ (A_i, \alpha)$
2. For all actions x,
   $\overline{wp}\ (C_x, (\exists\, a \bullet I \wedge \alpha)) \ \Rightarrow\ \exists\, a \bullet I \wedge \overline{wp}\ (A_x, \alpha)$
3. For all sets of actions, ref,
   $\neg\, (gd_C\ ref) \ \Rightarrow\ \exists\, a \bullet I \wedge \neg\, (gd_A\ ref).$

Labelled transition systems

A labelled state-transition system can be represented by a tuple of the form $T = (A, S, I, T_A)$ where A is a collection of labels, or names of transitions, S is the state space, initialisation I identifies possible initial states, and $T_A = \{T_a \mid a \in A\}$ is an indexed set containing the relations on the state space that describe the named transitions.

Modelling events as transitions, He [He89] and Josephs [Jos88] both show that CSP processes can be expressed as labelled transition systems. However, their definitions are structurally slightly different: Josephs' initialisation I simply identifies a set of initial states whereas He's initialisation I is a function from a single element $i \notin S$ onto each of the possible initial states; furthermore He introduces the element $\perp$ to handle divergence and includes the pair $(\perp, \perp)$ in the relation T_a for each a in A.

As well as giving a CSP semantics to labelled transition systems, He and Josephs both present sets of sound and complete simulation rules. Josephs' simulation rules are expressed in terms of the function *next* which, for any process $P = (A, S, I, P_A)$ and state $\sigma \in S$, returns the set containing those events in which process P can engage from state σ. The function *next* is formally defined as follows:

$$next_P(\sigma) \quad \widehat{=} \quad \{a \in A \mid \exists \sigma' : S \bullet (\sigma, \sigma') \in P_a\}.$$

Given this definition and a relation D from the state space of a transition system P_1 to the state space of a transition system P_2, Josephs' *downwards* (forwards) *simulation rules* are defined as follows:

1. $\forall \sigma_2 \in I_2 \;.\; \exists \sigma_1 \in I_1 \;.\; (\sigma_1, \sigma_2) \in D$

2. $\forall \sigma_1 \in S_1,\ \sigma_2, \sigma_2' \in S_2,\ a \in A \;.$
   $(\sigma_1, \sigma_2) \in D \wedge (\sigma_2, \sigma_2') \in P_{2_a}$
   $\Rightarrow$
   $\exists \sigma_1' \in S_1 \;.\; (\sigma_1, \sigma_1') \in P_{1_a} \wedge (\sigma_1', \sigma_2') \in D$

3. $\forall \sigma_1 \in S_1,\ \sigma_2 \in S_2 \;\;.\;\; (\sigma_1, \sigma_2) \in D \;\Rightarrow\; next_{P_1}(\sigma_1) = next_{P_2}(\sigma_2)$

Similarly, given a relation U from the state space of a transition system P_2 to the state space of a transition system P_1, Josephs' *upwards* (backwards) *simulation rules* are defined as follows:

1. $\forall \sigma_1 \in S_1,\ \sigma_2 \in I_2 \;.\; (\sigma_2, \sigma_1) \in U \;\Rightarrow\; \sigma_1 \in I_1$

2. $\forall \sigma_1' \in S_1,\ \sigma_2, \sigma_2' \in S_2,\ a \in A \;.$
   $(\sigma_2, \sigma_2') \in P_{2_a} \wedge (\sigma_2', \sigma_1') \in U$
   $\Rightarrow$
   $\exists \sigma_1' \in S_1 \;.\; (\sigma_2, \sigma_1) \in U \wedge (\sigma_1, \sigma_1') \in P_{1_a}$

3. $\forall \sigma_2 \in S_2 \;\;.\;\; \exists \sigma_1 \in S_1 \;\;.\;\; (\sigma_2, \sigma_1) \in U \;\wedge\; next_{P_1}(\sigma_1) \subseteq next_{P_2}(\sigma_2).$

Given state transition systems P_1 and P_2, and a relation d from the state space of P_1 to the state space of P_2, He's *downwards* (forwards) *simulation rules* are defined as follows:

1. $I_2 \subseteq I_1 \mathbin{⨾} d$

2. $\forall\, a \in A \;\; . \;\; d \mathbin{⨾} P_{2_a} \subseteq P_{1_a} \mathbin{⨾} d$

3. $\forall\, a \in A \;\; . \;\; (\mathrm{dom}\, P_{1_a}) \mathbin{⨾} d \subseteq \mathrm{dom}\, P_{2_a}$

Similarly, given state transition systems P_1 and P_2, and a relation u from the state space of P_2 to the state space of P_1, He's *upwards* (backwards) *simulation rules* are defined as follows:

1. $I_2 \mathbin{⨾} u \subseteq I_1$

2. $\forall\, a \in A \;\; . \;\; P_{2_a} \mathbin{⨾} u \subseteq u \mathbin{⨾} P_{1_a}$

3. $\forall\, s_2 \in S_2 \;\; . \;\; \exists\, s_1 \in S_1 \; .$
   $(s_2, s_1) \in u \;\wedge\; \forall\, a \in A \;.\; (s_1 \in \mathrm{dom}\, P_{1_a} \Rightarrow s_2 \in \mathrm{dom}\, P_{2_a})$

Information content of the simulation rules

One of the key results of this thesis, the result that motivates us to define $\mathcal{S}$, is the observation that the relational semantics of abstract data types does not correspond to the stable failures semantics of CSP processes.

Recalling that Josephs [Jos88] proves that refinement of state transition systems is equivalent to failures refinement within CSP and similarly Morgan [Mor90] proves that the weakest precondition semantics of action systems corresponds precisely to the failures semantics of CSP processes, an obvious question is why do we not get the same result? How is the information content of our relational semantics for abstract data types different to Josephs' semantics for state transition semantics and Morgan's weakest precondition semantics for action systems?

The difference can be seen easily in the respective simulation rules: the simulation rules for data types address the availability only of individual operations whereas the simulation rules for state-transition systems and action systems address the availability of combinations of operations. We illustrate first at the abstract level and then with a simple example how the information content of the backwards (upwards) simulation rules of these formalisms differ. In both cases, for clarity, we consider only systems that require no inputs and give no outputs.

The third backwards simulation rule for our abstract data types (with no inputs and no outputs) is given by:

$$\forall\, n : \mathrm{dom}\, ops' \;\bullet\; state' \setminus (\,\mathrm{dom}\, ops'\, n\,) \;\subseteq\; \mathrm{dom}\,(\, s \rhd \mathrm{dom}\, ops\, n\,).$$

This states that *for each individual operation* on the data types any point that lies outside the domain of the operation on the more concrete data type must correspond to a point that lies outside the domain of the operation on the more abstract data type. Josephs' corresponding upwards simulation rule is:

$$\forall\, \sigma_2 \in S_2 \;\; . \;\; \exists\, \sigma_1 \in S_1 \;\; . \;\; \sigma_2\, U \sigma_1 \;\wedge\; next_{P_1}(\sigma_1) \;\subseteq\; next_{P_2}(\sigma_2).$$

This states that for every state of $P2$, the more concrete model, there is a corresponding state of $P1$, the more abstract model, such that if $P1$ can next engage in an event e then so can $P2$. Notice that here we are considering combinations of events: the set $next_{P_1}(\sigma_1)$ contains all those events that $P1$ is able to engage in from state σ_1. Similarly in [WM90], Woodcock and Morgan's corresponding backwards simulation rules for action systems gives information about combinations of operations. For all *sets* of action names *ref*,

$$\neg\,(\,gd_C\ ref\,) \;\Rightarrow\; \exists\, a \;\bullet\; I \wedge \neg\,(gd_A\ ref).$$

We illustrate the differences between these simulation rules more concretely in Example 13 below. We model two systems P and Q first as data types, then as action systems, and then as state transition systems, and we identify a retrieve relation from the state space of P to the state space of Q. We show that, given this relation, the backwards simulation rule concerning the availability of operations holds for the data types but not for the state transition systems or for the action systems. Note that this example does not *prove* that the action system modelling P does not simulate the action system modelling Q and that the state transition modelling P does not simulate the the state transition modelling Q. We could do these proofs, showing that no such retrieve relation can exists. However, the *purpose* of the example is rather to explore the differences between the rules.

Example 13

System P has three possible states: 0, 1 and 4. Initially it can be either in state 1 or in state 4. Three operations, *aop*, *bop* and *cop* are defined on P. The domain of operation *aop* contains the single element 1 and the range of the operation contains the single element 0. Operations *bop* and *cop* share the same domain, the set containing the single element 4, and share the same range, the set containing the single element 0.

System Q has four possible states: 0, 1, 2 and 3, and can initially be in any state other than 0. Like system P, Q has three operations: *aop*, *bop* and *cop*. The domain of *aop* contains the single element 1; the domain of *bop* contains the single element 2; and the domain of *cop* contains the single element 3. The range of all three operations is the set containing the single element 0. Finally, the retrieve relation mapping the state space of Q onto the state space of P is $\mathrm{id}[\{0,1\}] \cup \{2,3\} \times \{1,4\}$.

Data types Given environment *Env*, we define the data types *DataP* and *DataQ*:

$$\begin{aligned}
DataP \;\widehat{=}\; (\,\mu\; ADT\,[\mathbb{N}, \{aop, bop, cop\}, Env] \;\bullet\; & \\
& state \;=\; \{\,0,1,4\,\} \;\wedge \\
& init \;=\; Env \times \{1,4\} \;\wedge \\
& ops \;=\; \{aop \mapsto \{(1,0)\}, bop \mapsto \{(4,0)\}, cop \mapsto \{(4,0)\}\} \;\wedge \\
& final \;=\; state \times Env\,)
\end{aligned}$$

$$\begin{array}{rcl} DataQ & \widehat{=} & (\,\mu\ ADT'\,[\mathbb{N}, \{aop, bop, cop\}, Env] \bullet \\ & & \quad state' \ = \ \{\,0,1,2,3\,\} \wedge \\ & & \quad init' \ = \ Env \times \{\,1,2,3\,\} \wedge \\ & & \quad ops' \ = \ \{aop \mapsto \{(1,0)\}, bop \mapsto \{(2,0)\}, cop \mapsto \{(3,0)\}\} \ \wedge \\ & & \quad final' \ = \ state \times Env\,) \end{array}$$

Given retrieve relation $s = \{(0,0),(1,1),(2,1),(2,4),(3,1),(3,4)\}$ we can show easily that all of the blocking backwards simulation rules of Section 4.4 hold and hence conclude that data type $DataP$ is refined by data type $DataQ$. However, since the purpose of this example is to explore the difference between the simulation rules for action systems, state transition systems and data types we consider only the crucial simulation rule: the backwards simulation rule concerning availability. We reason as follows:

$$\forall\, n : \mathrm{dom}\, ops \bullet state' \setminus \mathrm{dom}\, ops'\, n \subseteq \mathrm{dom}(s \rhd (\mathrm{dom}\, ops\, n))$$

$$\Leftrightarrow \ (\,\{0,1,2,3\} \setminus \{1\} \ \subseteq \ \mathrm{dom}(s \rhd \{1\}) \ \wedge \quad \{0,1,2,3\} \setminus \{2\} \ \subseteq \ \mathrm{dom}(s \rhd \{4\}) \ \wedge \quad \{0,1,2,3\} \setminus \{3\} \ \subseteq \ \mathrm{dom}(s \rhd \{4\})\,) \qquad [\mathrm{def}^n\mathrm{s\ of\ } P \mathrm{\ and\ } Q]$$

$$\Leftrightarrow \ (\,\{0,2,3\} \ \subseteq \ \{0,2,3\} \ \wedge \quad \{0,1,3\} \ \subseteq \ \{0,1,2,3\} \ \wedge \quad \{0,1,2\} \ \subseteq \ \{0,1,2,3\}\,) \qquad [\mathrm{set\ theory\ and\ def}^n \mathrm{\ of\ } s]$$

$$\Leftrightarrow \ \mathrm{true} \qquad [\mathrm{set\ theory}]$$

The backwards simulation rule concerning the availability of events holds for data types modelling P and Q.

Action systems We define action systems $ActP$ and $ActQ$ as follows:

$$ActP \ \widehat{=} \ \left(\begin{array}{l} \mathbf{var}\ p : \{0,1,4\} \\ \mathbf{initially}\ p :\in \{1,4\} \\ aop : -\ p = 1 \ \rightarrow \ p := 0 \\ bop : -\ p = 4 \ \rightarrow \ p := 0 \\ cop : -\ p = 4 \ \rightarrow \ p := 0 \end{array}\right) \quad ActQ \ \widehat{=} \ \left(\begin{array}{l} \mathbf{var}\ q : \{0,1,2,3\} \\ \mathbf{initially}\ q :\in \{1,2,3\} \\ aop : -\ q = 1 \ \rightarrow \ q := 0 \\ bop : -\ q = 2 \ \rightarrow \ q := 0 \\ cop : -\ q = 3 \ \rightarrow \ q := 0. \end{array}\right)$$

Given abstract invariant $I \ = \ (q \in \{0,1\} \wedge p = q) \ \vee \ (q \in \{2,3\} \wedge p \in \{1,4\})$ we show that the third backwards simulation rule does not hold:

$$\forall\, ref : \mathbb{P}\{aop, bop, cop\} \ \bullet \ \neg\,(\,gd_{ActQ}\ ref\,) \Rightarrow \exists\, a \ \bullet \ I \wedge \neg\,(gd_{ActP}\ ref)$$

$$\Rightarrow \qquad [\mathrm{property\ of\ } \forall]$$

$$\neg\,(\,gd_{ActQ}\ \{aop, bop\}\,) \ \Rightarrow \ \exists\, a \ \bullet \ I \wedge \neg\,(gd_{ActP}\ \{aop, bop\})$$

$\Leftrightarrow$ [def^n of gd]

$\neg\, q \in \{1,2\} \;\Rightarrow\; \exists\, a \;\bullet\; I \wedge \neg\, p \in \{1,4\}$

$\Rightarrow$ [set theory]

$q = 3 \;\Rightarrow\; \exists\, a \;\bullet\; I \wedge p \in \{0\}$

$\Leftrightarrow$ [def^n of I]

false

The element 3 in the state space of action system $ActQ$ lies outside the domain of both aop and bop. However, there is no corresponding state that lies outside the domain of both these operation on action system $ActP$.

State-transition systems We define state transition systems $TransP$ and $TransQ$:

$$\begin{aligned} TransP \;&=\; (\,\{aop, bop, cop\},\ \{0,1,4\},\ \{1,4\},\\ &\qquad \{aop \mapsto \{(1,0)\}, bop \mapsto \{(4,0)\}, cop \mapsto \{(4,0)\}\}\,)\\ TransQ \;&=\; (\,\{aop, bop, cop\},\ \{0,1,2,3\},\ \{1,2,3\},\\ &\qquad \{aop \mapsto \{(1,0)\}, bop \mapsto \{(2,0)\}, cop \mapsto \{(3,0)\}\}\,). \end{aligned}$$

Given retrieve relation $U = \{(0,0),(1,1),(2,1),(2,4),(3,1),(3,4)\}$ we show that the third backwards simulation rule does not hold.

$$\forall\, \sigma_q \in S_Q \;.\; \exists\, \sigma_p \in S_P \;.\; \sigma_q\, U \sigma_p \;\wedge\; next_P\,(\sigma_p) \;\subseteq\; next_Q\,(\sigma_q)$$

$\Rightarrow \exists\, \sigma_p \in S_P \;.\; 3\, U \sigma_p \;\wedge\; next_P\,(\sigma_p) \;\subseteq\; next_Q\,(3)$ [property of $\forall$]

$\Rightarrow (\, next_P\,(1) \;\subseteq\; next_Q\,(3)\,) \;\vee\; (\, next_P\,(4) \;\subseteq\; next_Q\,(3)\,)$ [def^n of U]

$\Rightarrow (\,\{aop\} \;\subseteq\; \{cop\}\,) \;\vee\; (\,\{bop, cop\} \;\subseteq\; \{cop\}\,)$ [def^n of $next$]

$\Leftrightarrow$ false [set theory]

From state 3 on $TransQ$ only operation cop is available. There is no equivalent state on $TransP$ from which only cop is available.

Summary We have shown that the backwards simulation rule concerning the availability of events holds for abstract data types modelling P and Q: for each operation, every state outside the domain of the operation on the data type modelling Q corresponds to to a state outside the domain of the same operation on the data type modelling P. However, we have also shown that the backwards simulation rules concerning the availability of events for action systems and state transition systems do not hold: we can find a state that lies outside the domains of both aop and bop on Q but we cannot find a corresponding state that lies outside the domain of both these operations on P. □

7.4 Related work: refinement and simulation of classes in Object-Z

As we observed in Section 1.2, Object-Z [Smi00] is an object-oriented extension of Z that includes notions of class, instance, inheritance and polymorphism. It differs from Z in that operations called outside their preconditions are always blocked. In this section we discuss two means of determining whether one Object-Z class is refined by another: the first using a set of simulation rules [DB01, SD97]; and the second by considering the *histories* of the class. We contradict Derrick, Boiten, Smith and Fischer [DB01, SD97, Fis00] by showing that these refinement orderings are not equivalent.

Relational semantics The accepted simulation rules [DB01, SD97] for Object-Z, although syntactically slightly different, are semantically identical to the blocking versions of the forwards and backwards simulation rules for data types as presented in Section 4.4. They are derived from the relational semantics given to Object-Z [DB01]. If A and C are Object-Z classes with the same set of operation names X, then given a retrieve relation *Retr* relating the state spaces of the two classes, the forwards/downwards and backwards/upwards simulation rules are respectively as follows:

$$\forall\, C.\textsc{State} \bullet C.\textsc{Init} \Rightarrow (\exists\, A.\textsc{State} \bullet A.\textsc{Init} \wedge \mathit{Retr})$$

$$\forall\, \textsc{Op} : X;\ A.\textsc{State};\ C.\textsc{State} \bullet \mathit{Retr} \Rightarrow (\mathrm{pre}\ A.\textsc{Op} \Leftrightarrow \mathrm{pre}\ C.\textsc{Op})$$

$$\forall\, \textsc{Op} : X;\ A.\textsc{State};\ C.\textsc{State};\ C.\textsc{State}' \bullet$$
$$(\mathit{Retr} \wedge C.\textsc{Op} \Rightarrow \exists\, A.\textsc{State}' \bullet \mathit{Retr}' \wedge A.\textsc{Op})$$

$$\forall\, A.\textsc{State};\ C.\textsc{State} \bullet C.\textsc{Init} \wedge \mathit{Retr} \Rightarrow A.\textsc{Init}$$

$$\forall\, \textsc{Op} : X;\ C.\textsc{State} \bullet \exists\, A.\textsc{State} \bullet \mathit{Retr} \wedge (\mathrm{pre}\ A.\textsc{Op} \Rightarrow \mathrm{pre}\ C.\textsc{Op})$$

$$\forall\, \textsc{Op} : X;\ A.\textsc{State}';\ C.\textsc{State};\ C.\textsc{State}' \bullet$$
$$(\mathit{Retr}' \wedge C.\textsc{Op} \Rightarrow \exists\, A.\textsc{State} \bullet \mathit{Retr} \wedge A.\textsc{Op}).$$

These rules are sound and jointly complete with respect to the relational semantics given to Object-Z: see [DB01].

Failures-divergences semantics Smith [Smi95] defines a semantic model for Object-Z classes in terms of their *histories*. Unlike Z, Object-Z has a reference-based semantics. Given the set *Id* of all possible identifiers and the set *Value* of all possible values that could be assigned to an identifier, the states of a class can be represented by a set S such that $S \subseteq (\mathit{Id} \mathrel{+\!\!\!\!\twoheadrightarrow} \mathit{Value})$. Similarly operations on a class can be represented by a set O such that $O \subseteq \mathit{Id} \times (\mathit{Id} \mathrel{+\!\!\!\!\twoheadrightarrow} \mathit{Value})$. The first identifier specifies the operation. The following set relates state parameters to the values to be assigned to them.

Smith defines the *history* of an object to be the sequence of states through which it has passed together with the sequence of operations that brought about these changes in state. Constraints ensure that: the sequence of events is non-empty; either both sequences are infinite or the length of the state sequence is precisely one more than the length of the associated operation sequence; and the set of histories is prefix-closed. More formally, the histories of a class can be represented by a set H such that $H \subseteq (S^{\omega} \times O^{\omega})$ and such the following predicate holds:

$$
\begin{array}{l}
\forall\, s, s_1, s_2 : S^{\omega};\ o, o_1, o_2 : O^{\omega} \bullet \\
\quad (s, o) \in H \ \Rightarrow\ s \neq \langle\rangle\ \land \\
\quad (s, o) \in H \land s \notin S^* \ \Rightarrow\ o \notin O^*\ \land \\
\quad (s, o) \in H \land s \in S^* \ \Rightarrow\ \#s = \#o + 1\ \land \\
\quad (s_1 \frown s_2, o_1 \frown o_2) \in H \land \#s_1 = \#o_1 + 1 \ \Rightarrow\ (s_1, o_1) \in H
\end{array}
$$

Given a function *event* that relates operations and events in the natural way[1] and a function *events* that relates sequences of operations and sequences of events:

$$
\begin{array}{rcl}
events\,(\langle\rangle) & = & \langle\rangle \\
events\,(\langle op\rangle \frown o) & = & \langle event\,(op)\rangle \frown events\,(o)
\end{array}
$$

Smith [Smi97] defines the failures of an object C_i with initial state i and of class C with history H as follows:

$$
\begin{array}{l}
failures\,(C_i) \ = \\
\quad \{\, (t, X) \mid \\
\qquad \exists (s, o) \in H \bullet (\, s \in S^* \ \land\ i \subseteq s(1)\ \land\ t = events\,(o) \\
\qquad\qquad \land \\
\qquad\qquad \forall\, e \in X \bullet \nexists\, st \in S, op \in O \bullet (\, e = event\,(op) \\
\qquad\qquad\qquad \land \\
\qquad\qquad\qquad (s \frown \langle st\rangle, o \frown \langle op\rangle) \in H\,)\,)\,\}.
\end{array}
$$

He proves that they satisfy the healthiness conditions for the stable failures semantic model [Ros98]. It follows that one class is refined by another when the set of failures of the first class is a subset of the set of failures of the second.

Discussion Derrick, Smith and Boiten [DB01, SD97] argue that the simulation rules are equivalent to those of Josephs [Jos88] and hence are sound and jointly complete with respect to the failures-divergences refinement ordering. We have shown in Observation 8 (and give a detailed proof in the context of Object-Z in [BD02]) that this is not the case: the relational and failures-divergences semantic models for Object-Z are not equivalent.

In the main body of this thesis we proved that data refinement in Z is not equivalent to stable failures refinement within CSP. As a result we went on to define an alternative semantic model for CSP: a model whose refinement ordering coincided with data refinement. For Object-Z, however, we have taken the opposite approach. We present a set of simulation rules that coincide with the failures-divergences refinement ordering. If A and

[1] For details see [Smi95].

C are Object-Z classes with the same set of operation names X, then given a retrieve relation *Retr* relating the state spaces of the two classes, the *correct* forwards/downwards and backwards/upwards simulation rules are respectively as follows:

$$\forall\, C.\textsc{State} \bullet C.\textsc{Init} \Rightarrow (\exists\, A.\textsc{State} \bullet A.\textsc{Init} \wedge \mathit{Retr})$$

$$\forall\, \textsc{Op} : X;\ A.\textsc{State};\ C.\textsc{State} \bullet \mathit{Retr} \Rightarrow (\text{pre } A.\textsc{Op} \Leftrightarrow \text{pre } C.\textsc{Op})$$

$$\begin{aligned}&\forall\, \textsc{Op} : X;\ A.\textsc{State};\ C.\textsc{State};\ C.\textsc{State}' \bullet\\ &\qquad\qquad \mathit{Retr} \wedge C.\textsc{Op} \Rightarrow \exists\, A.\textsc{State}' \bullet \mathit{Retr}' \wedge A.\textsc{Op}\end{aligned}$$

$$\forall\, A.\textsc{State};\ C.\textsc{State} \bullet C.\textsc{Init} \wedge \mathit{Retr} \Rightarrow A.\textsc{Init}$$

$$\forall\, C.\textsc{State} \bullet \exists\, A.\textsc{State} \bullet (\forall\, \textsc{Op} : X \bullet \mathit{Retr} \wedge (\text{pre } A.\textsc{Op} \Rightarrow \text{pre } C.\textsc{Op}))$$

$$\begin{aligned}&\forall\, \textsc{Op} : X;\ A.\textsc{State}';\ C.\textsc{State};\ C.\textsc{State}' \bullet\\ &\qquad\qquad \mathit{Retr}' \wedge C.\textsc{Op} \Rightarrow \exists\, A.\textsc{State} \bullet \mathit{Retr} \wedge A.\textsc{Op}.\end{aligned}$$

These rules are very similar to the previously accepted simulation rules. Indeed five out of the six rules are identical. The only rule that is different is the second backwards simulation rule, concerning the applicability of operations. The difference lies in the scope of the quantifications. In the original version we have the universal quantification of all operations over the existential quantification of all abstract states ($\forall\, Op \bullet \exists\, A.\textsc{State} \bullet \ldots$) whereas in the corrected version we have a stronger predicate: the existential quantification of all abstract states over the universal quantification of all operations ($\exists\, A.\textsc{State} \bullet \forall\, Op \bullet \ldots$). The former provides information about the availability of individual operations but the latter provides information about the availability of combinations of operations. We conclude this section with a simple observation, confirming an expected result.

Observation 11 The revised backwards simulation rule concerning the availability of operations does not hold for Object-Z classes corresponding to data types P and Q of Observation 8.

Proof Given Object-Z classes corresponding to data types P, Q as defined in Lemma 5, and given retrieve relation *Retr* also defined in Lemma 5 we reason as follows:

$$\forall\, Q.\textsc{State} \bullet \exists\, P.\textsc{State} \bullet (\forall\, \textsc{Op} \bullet \mathit{Retr} \wedge (\text{pre } P.\textsc{Op} \Rightarrow \text{pre } Q.\textsc{Op}))$$

$$\begin{aligned}\Leftrightarrow\ & \forall\, q : \mathit{State} \bullet \qquad\qquad [\text{def}^n \text{ of } P \text{ and } Q]\\ &\quad \exists\, p : \mathit{State} \bullet\\ &\qquad ((q = 0 \wedge p = 0) \vee (q = 1 \wedge p = 1) \vee (q = 2 \wedge p = 2) \vee\\ &\qquad\ (q = 3 \wedge (p = 1 \vee p = 2)) \vee (q = 4 \wedge (p = 1 \vee p = 2)))\\ &\qquad \wedge\\ &\qquad ((p = 1 \Rightarrow q = 1) \wedge\\ &\qquad\ (p \in \{2,3\} \Rightarrow q \in \{2,3\}) \wedge (p \in \{2,4\} \Rightarrow q \in \{2,4\}))\end{aligned}$$

$$\begin{array}{lll}\Leftrightarrow \forall q : State \bullet & ((q = 0 \Rightarrow \exists p : State \bullet p = 0) \land & \text{[predicate calculus]}\\ & (q = 1 \Rightarrow \exists p : State \bullet p = 1) \land & \\ & (q = 2 \Rightarrow \exists p : State \bullet p = 2) \land & \\ & (q = 3 \Rightarrow \exists p : State \bullet \text{false}) \land & \\ & (q = 4 \Rightarrow \exists p : State \bullet \text{false})) & \end{array}$$

$$\Leftrightarrow \text{false} \qquad \text{[predicate calculus]}$$

□

It is interesting to note that the failures-divergences refinement ordering for Object-Z is *defined* by the above sets of simulation rules whereas the simulation rules presented in [DB01, SD97] are *derived* to reflect the relational, behavioural (blocking) semantic model that can be given to Object-Z.

7.5 Related work: formalising graphical notations

A significant amount of work has been done by others on formalising state-based models within UML. This work has been carried out in the context of various state-based languages but in each case the intent and underlying principles are the same. It is not our intention to repeat this work; rather we will discuss what has already been achieved by others, and how a combination of their work and the work presented in this thesis can be used to to verify consistency within UML specifications.

A combination of notations can be used to model an implementation level class description of a system within a UML specification. Annotations written in the Object Constraint Language (OCL) [JBW99] can be included in class diagrams. These can give class invariants or instance invariants. Furthermore, class diagrams can be accompanied by a collection of state diagrams providing information about the availability of methods and about the effect of operation calls on the state.

A class model containing only a single class, and reasoning only about a single object within that class is simple enough to express as an abstract data type. Each data member becomes a component of the state space and each method an operation on the data type. Hall [Hal94] addresses the issue of inheritance; how a model of class hierarchy can be created within Z. Also in [BH94], Hammond [Ham94] considers the formalisation of class models considering multiple classes and their associations. Each object is given a unique, static identifier and associations are recorded as functions or relations between identifiers.

France, Shroff, Bruel and Larrondo-Petrie [SF97, FBLPS97] developed the work of Hammond, focusing on constructs specific to UML. They present a type semantics for UML class models, translating class diagrams into Z. They address the issue of multiple classes and associations using promotion, again giving unique identifiers to objects and recording associations as functions or relations. Since refinement distributes through promotion [WD96] these descriptions can then be simplified if necessary.

Butler and Snook [SB01] are working in the context of UML and B [Abr96], a state-based specification language that models system behaviour in terms of *machines* whose state changes under operations. They have developed a prototype tool [SB] that translates annotated UML class descriptions into B machines; the translation of classes, attributes and operations to B is derived from the work of Meyer and Souquieres [MS99].

Although the underlying principles are similar, their motivation is slightly different. It is not their intention to formalise UML; rather their goal is to use graphical notations to facilitate formal specification within B. The modelling tool Rose [Rat] can be used to reduce *viscosity* [Gre89]; to easily and quickly explore the design of a system and hence to identify a workable model with appropriate abstractions before translating it to B. Since their immediate goal is not the formalisation of UML they are not constrained to work with the whole of UML but restrict themselves to a subset of the language. In addition they require textual constraints to be expressed in B.

Dupuy, Ledru and Chabre-Peccoud [DLCP00] have also built a prototype tool, RoZ along similar lines except that it works within the context of Z and that it does not consider state diagrams. It takes class diagrams written in the Rational Rose environment with formal annotations in Z or OCL and produces the corresponding specification in Z. It can then do simple consistency checks; validating operation guards and verifying the consistency of constraints.

Börger, Cavarra and Riccobene [BCR00b] have given a formal semantics to statecharts. Their work is done in the context of Abstract State Machines [Gur93], systems of finitely many transition rules which transform abstract states. They have built on this work to provide a formal semantics for activity graphs [BCR00a]. Although working in a different paradigm, this work is very similar in essence to that presented in Section 6.2.

7.6 Future research

We have shown how data types, both with implicit and explicit notions of communication, can be translated to their corresponding processes and hence compared with behavioural models of the same system. However, our ultimate goal is not the comparison of individual data types. We wish to be able to compare *large systems* built up from smaller, interacting components. To achieve this, we must have consistent definitions of the parallel operator and consistent definitions of the hiding operator for data types and processes. Larger systems can then be built from smaller components by putting these components in parallel, synchronising on their shared events, and hiding internal events where necessary.

For *simple* data types, as defined in Chapter 4, our definition of the parallel operator is relatively straightforward. The state space of the resulting data type is the cross-product of the state spaces of the data types to be combined. Similarly the initialisation operation, finalisation operation and all shared operations are the cross-product of the corresponding operations on the data types concerned. Operations defined only on one of the data types do not change the component of the state that corresponds to the other data type. Operations defined on both data types change the components of the state space according to the operation on the data type concerned. Verifying the equivalence of the parallel operator for simple data types and the parallel operator for their corresponding processes will be trivial.

The definition of a parallel operator for *communicating* data types will not be as straight-forward as the definition of the parallel operator for simple data types. Indeed, we cannot define such an operator with our current definitions. In the process corresponding to a communicating data type, an output value is chosen only after an input value has been given. Two such processes will not be able to pass values as they both insist on first accepting their input value and then giving their output value.

Suppose that a client and a server share an operation *request* in which the client requests a service and the server responds with a confirmation or refusal. We wish for the process corresponding to the *client* to contain a clause along the following lines:

$$\sqcap\, serv : Service \bullet \Box\, resp : Response \bullet request.serv.resp \rightarrow \ldots$$

and we wish for the process corresponding to the *server* to contain a complementary clause along the following lines:

$$\Box\, serv : Service \bullet \sqcap\, resp : Response \bullet request.serv.resp \rightarrow \ldots$$

To overcome the problem of synchronisation we must redefine our communicating data types, differentiating between operations that first accept an input value and operations that first give an output value. Our tentative first definition of *passing data types*, data types that pass values, splits the set *ops* into four disjoint sets: *opsIO* representing those operations for which an input is first accepted and then an output given; *opsOI* representing those operations for which output is first given and then input accepted; *opsII* representing those operations that accept two inputs; and *opsOO* representing those operations that give two outputs. Such a definition will probably be as follows:

$$\begin{array}{l}
PDT[State, Name, Value] \\
\hline
state : \mathbb{P}\, State \\
init : \mathbb{P}\, State \\
opsIO, opsOI, opsII, opsOO : Name \pfun (\,(State \times Value) \leftrightarrow (State \times Value)\,) \\
\hline
\emptyset \subset init \subseteq state \;\wedge \\
(opsIO \cup opsOI \cup opsII \cup opsOO) \subseteq \\
\qquad Name \pfun (\,(state \times Value) \leftrightarrow (state \times Value)\,) \;\wedge \\
\mathrm{dom}\, opsIO \cap (\mathrm{dom}\, opsOI \cup \mathrm{dom}\, opsII \cup \mathrm{dom}\, opsOO) = \emptyset \;\wedge \\
\mathrm{dom}\, opsOI \cap (\mathrm{dom}\, opsII \cup \mathrm{dom}\, opsOO) = \emptyset \;\wedge \\
\mathrm{dom}\, opsII \cap \mathrm{dom}\, opsOO = \emptyset
\end{array}$$

An example of an operation that might accept two inputs is the *request* operation on a *monitor* in the client-server model mentioned above: it might record which services have been requested and whether they are going to be given. We would wish for the process corresponding to such a monitor to contain a clause along the following lines:

$$\Box\, serv : Service \bullet \Box\, resp : Response \bullet request.serv.resp \rightarrow \ldots$$

An example of an operation that might give two outputs is the *request* operation on the data type corresponding to the parallel combination of the client and server.

Having distinguished between operations that give and then receive values and operations that receive and then give values we can reflect this in the corresponding processes. In this exposition we consider only the blocking versions. Such a process might be defined thus:

$$
\begin{array}{l}
\mathit{process}_{PDT}\,(\,\theta PDT[\mathit{State}, \mathit{Name}, \mathit{Value}]\,) \;=\\
\quad \text{let}\\
\quad\quad \mathit{Proc}(a) \;=\\
\quad\quad\quad \Box\, \mathit{name} : \operatorname{dom} \mathit{opsIO};\; \mathit{in} : \mathit{Value} \mid (a, \mathit{in}) \in \operatorname{dom} \mathit{opsIO}\ \mathit{name} \bullet\\
\quad\quad\quad\quad \sqcap\, a' : \mathit{State};\; \mathit{out} : \mathit{Value} \mid (a, \mathit{in}) \mapsto (a', \mathit{out}) \in \mathit{opsIO}\ \mathit{name} \bullet\\
\quad\quad\quad\quad\quad \text{op}.\mathit{name}.\mathit{in}.\mathit{out} \rightarrow \mathit{Proc}(a')\\
\quad\quad\quad \Box\\
\quad\quad\quad \Box\, \mathit{name} : \operatorname{dom} \mathit{opsOI} \mid (\exists\, \mathit{out} : \mathit{Value} \bullet (a, \mathit{out}) \in \operatorname{dom} \mathit{opsOI}\ \mathit{name}) \bullet\\
\quad\quad\quad\quad \sqcap\, \mathit{out} : \mathit{Value} \mid (a, \mathit{out}) \in \operatorname{dom} \mathit{opsOI}\ \mathit{name} \;\bullet\\
\quad\quad\quad\quad\quad \Box\, \mathit{in} : \mathit{Value} \mid (\exists\, a' : \mathit{state} \bullet (a, \mathit{out}) \mapsto (a', \mathit{in}) \in \mathit{opsOI}\ \mathit{name}) \bullet\\
\quad\quad\quad\quad\quad\quad \sqcap\, a' : \mathit{state} \mid (a, \mathit{out}) \mapsto (a', \mathit{in}) \in \mathit{opsOI}\ \mathit{name} \bullet\\
\quad\quad\quad\quad\quad\quad\quad \text{op}.\mathit{name}.\mathit{out}.\mathit{in} \rightarrow \mathit{Proc}(a')\\
\quad\quad\quad \Box\\
\quad\quad\quad \Box\, \mathit{name} : \operatorname{dom} \mathit{opsII};\; \mathit{in}_1 : \mathit{Value} \mid (a, \mathit{in}_1) \in \operatorname{dom} \mathit{opsIO}\ \mathit{name} \bullet\\
\quad\quad\quad\quad \Box\, \mathit{in}_2 : \mathit{Value} \mid (\exists\, a' : \mathit{state} \bullet (a, \mathit{in}_1) \mapsto (a', \mathit{in}_2) \in \mathit{opsII}\ \mathit{name}) \bullet\\
\quad\quad\quad\quad\quad \sqcap\, a' : \mathit{state} \mid (a, \mathit{in}_1) \mapsto (a', \mathit{in}_2) \in \mathit{opsII}\ \mathit{name} \bullet\\
\quad\quad\quad\quad\quad\quad \text{op}.\mathit{name}.\mathit{in}_1.\mathit{in}_2 \rightarrow \mathit{Proc}(a')\\
\quad\quad\quad \Box\\
\quad\quad\quad \Box\, \mathit{name} : \operatorname{dom} \mathit{opsOO} \mid (\exists\, \mathit{out} : \mathit{Value} \bullet (a, \mathit{out}) \in \operatorname{dom} \mathit{opsOI}\ \mathit{name}) \bullet\\
\quad\quad\quad\quad \sqcap\, \mathit{out}_1 : \mathit{Value} \mid (a, \mathit{out}_1) \in \operatorname{dom} \mathit{opsOO}\ \mathit{name} \;\bullet\\
\quad\quad\quad\quad\quad \sqcap\, a' : \mathit{State};\; \mathit{out}_2 : \mathit{Value} \mid (a, \mathit{out}_1) \mapsto (a', \mathit{out}_2) \in \mathit{opsOO}\ \mathit{name} \bullet\\
\quad\quad\quad\quad\quad\quad \text{op}.\mathit{name}.\mathit{out}_1.\mathit{out}_2 \rightarrow \mathit{Proc}(a')\\
\quad \text{within}\\
\quad\quad \sqcap\, a' : \mathit{init} \bullet \mathit{Proc}(a')
\end{array}
$$

The first clause, concerning those operations in which input is accepted and then output is given, is identical to the predicate in the process corresponding to communicating data types. The second clause, concerning those operations in which output is given and then input accepted, needs close attention. To reflect the blocking semantics, an operation can be chosen only if an output for that operation can be given from the current state. If this is the case, then such an output can be chosen non-deterministically given the deterministically chosen operation. An input value will be accepted only if there is some matching after state. Such a state will be chosen non-deterministically according to the deterministically chosen input value.

The third clause concerns those operations that accept two inputs. The operation and first input will be accepted only if the current state permits such a combination. The second input will be accepted provided there is a matching after-state. Such a state will be chosen non-deterministically according to the given inputs. The final clause concerns those operations that give two outputs. An operation can be chosen only if some output can be given for that operation from the current state. If this is the case then this first output is chosen non-deterministically. A second output and an after-state can then be non-deterministically chosen.

Consider the client, server and monitor described above and suppose that the only operation they have in common is *request*. This operation would be in the set *opsOI* of the client, the set *opsIO* of the server and the set *opsOO* of the monitor. We could put the client and the server in parallel. The client would pass the name of the requested service to the server and the server would return with its response. We could then put the monitor in parallel with the resultant process. The monitor would then "listen" to requests and their responses.

We will need to consider carefully the constraints that we wish to impose on pairs of data types to be put in parallel. For instance we might insist that if two data types share an operation then either one must give an output and then receive an input and the other must receive an input and then give an output, or one must give two outputs and the other must receive two inputs. If we adopt this model then, in either case the corresponding operation on the resulting data type will give two outputs. Alternative, and more flexible, models might allow the sharing of an operation if, for instance, one of the data types gives an output and then requires an input and the other requires two outputs. Were we to adopt this model then the resulting operation would give an output and then require an input.

Whatever our chosen model we will need to consider carefully how we handle the issue of non-deterministic outputs. If two CSP processes with non-deterministic outputs are put in parallel then the system might deadlock. We need to obtain a corresponding result when we put two data types with non-deterministic outputs in parallel. Butler [But92] addresses this issue, when defining a parallel operator for action systems, by incorporating a Boolean into the state that flags whenever the system might deadlock.

7.7 Conclusions

In this thesis we have investigated the relationship between the refinement of state-based and event-based systems. We have identified a correspondence between abstract data types expressed in Z and processes expressed in CSP. We have considered data types both with an implicit and an explicit treatment of communication and have worked in the context of two complementary semantic interpretations. The first, the *blocking model* in which operations cannot be called outside their preconditions, is typically used for the latter stage of the development process when the domain of definition has been identified for all operations. This semantic interpretation has been given to Object-Z. The *non-blocking* semantics is used for models of systems in which the domain of definition is still being explored.

In the process of our investigation we defined, and explored the properties of, a new semantic model for CSP whose information content lies between that of the traces model and the stable failures model: to coincide with the relational semantics of data types, availability is recorded only on an event-by-event basis. We identified the shortcomings of our proposed semantic model—care needs to be taken when applying the hiding operator—and presented a sound and complete rule for automating refinement checking within this model.

A significant amount of work has been done by others on obtaining a formal state-based description of state diagrams and annotated class diagrams in UML. We discussed this work and presented a formal semantics in terms of CSP for activity graphs, one of the notations used in UML to model the behaviour of a system. Having obtained state-based and event-based descriptions of the same model we showed how our refinement-preserving translation from data types to processes might be used to verify consistency of UML specifications. We discussed how this process might be automated so that software engineers developing models using UML might gain the strength of more formal modelling techniques without having to grapple with the syntax and semantics.

Bibliography

[Abr96] J. R. Abrial. *The B Book: assigning programs to meanings.* Cambridge University Press, 1996.

[Bac80] R. J. R. Back. Correctness preserving program refinements: Proof theory and applications, 1980. Tract 131, Mathematisch Centrum, Amsterdam.

[Bac90] R. J. R. Back. Refinement calculus I: Sequential and nondeterministic programs. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop "Stepwise Refinement of Distributed Systems"*. Springer-Verlag, 1990.

[BCR00a] E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In T. Rus, editor, *Proceedings of Algebraic Methodology and Software Technology (AMAST 2000)*, volume 1816 of *Lecture Notes in Computer Science*. Springer, 2000.

[BCR00b] E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML state machines. In T. Rus, editor, *Proceedings of Abstract State Machines, Theory and Technology Applications, (ASM 2000)*, volume 1912 of *Lecture Notes in Computer Science*. Springer, 2000.

[BD99] H. Bowman and J. Derrick. A junction between state-based and behavioural specification. In A. Fantechi P. Ciancarini and R. Gorrieri, editors, *Proceedings of Formal Methods for Open Object-Based Distributed Systems (FMOODS '99)*. Kluwer, 1999.

[BD02] C. Bolton and J. Davies. Refinement in Object-Z and CSP, 2002. Submitted to Integrated Formal Methods (IFM '02).

[BDW99] C. Bolton, J. Davies, and J. Woodcock. On the refinement and simulation of data types and processes. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of Integrated Formal Methods (IFM'99)*. Springer, 1999.

[Bec99] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[BH94] J. P. Bowen and J. A. Hall, editors. *Z User Workshop, Cambridge 1994*, Workshops in Computing. Springer-Verlag, 1994.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1), 1985.

[BKS83] R. J. R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *In 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1983.

[Bol98] C. Bolton. Objects, processes and states. Master's thesis, Oxford University Computing Laboratory, 1998.

[Bol00] C. Bolton. IoData types and processes. Technical Report PRG-TR-01-00, University of Oxford, 2000.

[But92] M. J. Butler. *A CSP approach to Action Systems*. PhD thesis, University of Oxford, 1992.

[But00] M. J. Butler. csp2b: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12, 2000.

[CM88] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley, 1988.

[Col93] D. Coleman. *Object-oriented development: The Fusion Method*. Prentice Hall, 1993.

[DB01] J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Springer, 2001.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminancy, and formal derivation of programs. *Communications of the ACM*, 18, 1975.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall International Series in Computer Science, 1976.

[DLCP00] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. An overview of RoZ: a tool for integrating UML and Z specifications. In B. Wangler and L. Bergman, editors, *Proceedings of 12th Conference on Advanced information Systems Engineering (CAiSE'2000)*, volume 1789 of *Lecture Notes in Computer Science*. Springer, 2000. http://www-lsr.imag.fr/Les.Groupes/pfl/RoZ/.

[dRE98] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science, 1998.

[DRS95] R. Duke, G. Rose, and G. Smith. Object-Z: a specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511 – 533, 1995.

[DS00] J. Derrick and G. Smith. Structural refinement in object-z/csp. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Proceedings of Integrated Formal Methods (IFM '00)*, volume 1945 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[DW98] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML : The Catalysis Approach*. Addison-Wesley, 1998.

[EFLR98] A. Evans, R. B. France, K. C. Lano, and B. Rumpe. Developing the UML as a formal modelling notation. In J. Bezivin and P.-A. Muller, editors, *Proceedings of The Unified Modeling Language, (UML '98): Beyond the notation*. Springer-Verlag, 1998.

[EHS97] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL: Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1997.

[FBLPS97] R. B. France, J.-M. Bruel, M. M. Larrondo-Petrie, and M. Shroff. Exploring the semantics of UML type structures with Z. In H. Bowman and J. Derrick, editors, *Proceedings of Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2. Chapman and Hall, 1997.

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Proceedings of Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2. Chapman and Hall, 1997.

[Fis00] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, University of Oldenburg, 2000.

[For97] Formal Systems (Europe) Ltd. *Failures-divergences refinement: FDR 2*, 1997. FDR 2 user manual.

[FS97] C. Fischer and G. Smith. Combining CSP and Object-Z: Finite or infinite trace semantics? In T. Mizuno, T. Higashino N. Shiratori, and A. Togashi, editors, *Proceedings of FORTE/PSTV'97*, pages 503 – 518. Chapman and Hall, 1997.

[FS99] M. Fowler and K. Scott. *UML Distilled*. Addison-Wesley, 1999.

[Gre89] T. R. G. Green. Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay, editors, *People and computers V: Proceedings of the Fifth Conference of the British Computer Society Human-Computer Interaction Specialist Group*. Cambridge University Press, 1989.

[GS97] A. J. Galloway and W. Stoddart. An operational semantics for ZCCS. In M. Hinchey and Shaoying Liu, editors, *International Conference of Formal Engineering Methods (ICFEM)*, pages 272–282. IEEE Computer Press, 1997.

[Gur93] Y. Gurevich. Evolving algebras: an attempt to discover semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*. World Scientific Publishing, 1993.

[Hal94] J. A. Hall. Specifying and interpreting class heirarchies in Z. In J. Bowen and J. A. Hall, editors, *Proceedings of Z User Workshop, Cambridge 1994*. Springer-Verlag, 1994.

[Ham94] J. A. R. Hammond. Producing Z specifications from object-oriented analysis. In J. Bowen and J. A. Hall, editors, *Proceedings of Z User Workshop, Cambridge 1994*. Springer-Verlag, 1994.

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[He89] J. He. Process refinement. In J. McDermid, editor, *The Theory and Practice of Refinement*, pages 511 – 533. Butterworths, 1989.

[HHS87] C. A. R. Hoare, J. He, and J. W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 1987.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:511 – 533, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hoa94] C. A. R. Hoare. Unified theories of programming. Monograph, Oxford University Computing Laboratory, 1994.

[JBW99] A. G. Kleppe J. B. Warmer. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, 1990.

[Jos88] M. B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1), 1989.

[Lev95] N. G. Leveson. *Safeware : System Safety and Computers*. Addison-Wesley, 1995.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mil89] R. Milner. *Communications and concurrency*. Prentice Hall, 1989.

[Mor87] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3), 1987.

[Mor88] C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages*, 10(3), 1988.

[Mor90] C.C. Morgan. Of wp and CSP. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our Business: a Birthday Salute to Edsger W. Dijkstra*. Springer-Verlag, 1990.

[Mor94] C. C. Morgan. *Programming from Specifications (2nd Edition)*. Prentice-Hall, 1994.

[MRG88] C. C. Morgan, K. A. Robinson, and P. H. B. Gardiner. On the refinement calculus, 1988. Programming Research Group Technical Monograph PRG-70, Oxford University.

[MS99] E. Meyer and J. Souquieres. A systematic approach to transform OMT diagrams to a B specification. In J.M. Wing, J.C.P. Woodcock, and J. Davies, editors, *Proceedings of World Congress on Formal Methods in the Development of Computing Systems, (FM '99)*, volume 1708 of *Lecture Notes in Computer Science*. Springer, 1999.

[Nel89] G. Nelson. A generalization of dijkstra's calculus. *ACM Transactions on Programming Languages*, 11(4), 1989.

[OMG99] Object Management Group. *Unified Modeling Language Specification, version 1.4*. Rational Software Corporation, Santa Clara, CA 95051, USA, 1999.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall International, 1981.

[Pnu86] A. Pnueli. Specification and development of reactive systems. In H.-J. Kugler, editor, *Proceedings of ()*, IFIP, North Holland, 1986. Information Processing.

[Rat] Rational Software Corporation. *Using Rose: Rational Rose 2000e*. Part number 800-023321-000.

[RBP+91] J. Rumbaugh, M Blaha, W. Premerlani, F. Eddy, and B. Lorensen. *Object-oriented Modeling and Design*. Prentice Hall, 1991.

[RJB97] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1997.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science, 1998.

[SB] C. Snook and M. Butler. *U2B Class Diagram to B Translator*. Manual for tool.

[SB01] C. Snook and M. Butler. Using a graphical design tool for formal specification. In Dumke and Abran, editors, *Proceedings of 13th Annual Workshop of the Psychology of Programming Interest Group*, 2001.

[SD97] G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M. Hinchey and Shaoying Liu, editors, *Proceedings of First IEEE International Conference on Formal Engineering Methods (ICFEM '97)*. IEEE Computer Society, 1997.

[SF97] M. Shroff and R. B. France. Towards a formalization of UML class structures in Z. In *Proceedings of 21st International Computer Software and Applications Conference (COMPSAC '97)*. IEEE Computer Society Press, 1997.

[Smi95] G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7:30 – 65, 1995.

[Smi97] G. Smith. A semantic integration of Object-Z and CSP. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Proceedings of Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[SP99] P. Stevens and R. Pooley. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 1999.

[Spi92] J. M. Spivey. *The Z Notation: a Reference Manual*. Prentice-Hall International, 1992.

[SW01] D. Sangiorgi and D. Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[TA97] K. Taguchi and K. Araki. Specifying concurrent systems by Z and CCS. In *Int'l Symposium on Future Software Technology (ISFST)*, pages 101–108. Software Engineers Association, 1997.

[WD96] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[WM90] J. C. P. Woodcock and C. C. Morgan. Refinement of state-based concurrent systems. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM and Z - Formal Methods in Software Development*. Springer, 1990.