Extending Model Checking to Object Process Validation

Rick van Rein

University of Twente
P.O. Box 217 - 7500 AE Enschede - The Netherlands
telephone +31-53-4893690 / fax +31-53-4892927

Acknowledgement. I wish to express my gratitude to my friends and collegues, my family and my promotion committee for supporting me in this work. I am particularly grateful for the patient guidance of Maarten Fokkinga, who has been the best mentor I could have asked for, and a friend at the same time.

Keywords: object orientation, process algebra, model checking, theorem proving, database transactions, components.

Printed by Ipskamp PrintPartners, Enschede, The Netherlands.
ISBN 90-365-17273
ISSN 1381-3617; no 02-43 (CTIT Ph.D-thesis series)

EXTENDING MODEL CHECKING TO OBJECT PROCESS VALIDATION

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. F. A. van Vught,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 2 mei 2002 te 13.15 uur.

door

Henderikus van Rein

geboren op 24 maart 1969
te Rolde

Dit proefschrift is goedgekeurd door:

Prof. dr. P. M. G. Apers

Contents

Chapter 1

Introduction

It is the mark of an educated mind
to be able to entertain a thought
without accepting it.

Aristotle

Object-oriented techniques allow the gathering and modelling of system requirements in terms of an application area. The expression of data and process models at that level is a great asset in communication with non-technical people in that area, but it does not necessarily lead to consistent models. To avoid that inconsistencies are only discovered during the latest phases of development, it is helpful if analysis models can be verified, especially if this is done in an automated process and if it gives feedback that helps repair the models. This thesis presents such an approach for the process aspects of analysis models.

The development of (object-oriented) software usually goes through the phases of analysis, design and finally implementation. The purpose of analysis is to capture conceptual models, the purpose of design is to outline the technical infrastructure, and the purpose of implementation is to obtain an executable application. It is the intention of analysis models and design models to be precise descriptions of a system, and there is great interest to employ software to seek for inconsistencies and other errors in these models; the alternative being that errors are not found before the implementation phase, during which repairs are usually costly. To accommodate this desire, this thesis work focuses on the construction of precise analysis models and their verification. The extent to which this is done is limited to process models, such as state charts of classes, or workflow diagrams.

To achieve this goal, this thesis work must integrate mathematical and practical process modelling approaches. In practical approaches, an easy-to-read, expressive notation is considered important because it is felt to express a designer's instincts most directly. In the mathematical world, precision is the key to success. These two emphasises do not necessarily interfere, but they are hardly ever combined. This may be because one perspective is usually expressed so heavily that the other is sacrificed. Process theories do not provide all concepts required for practical models, and practical models tend to lack the precision required for a mathematical approach.

This work therefore establishes a graphical notation that is sufficiently close to practical models to be useful for the practical model maker. This notation is translated to a formal notation, effectively providing a formal semantics for the graphical notation. Based on these formal semantics, correctness proofs can be conducted, and when the prover stumbles over a problem, it reports an error. Care has been taken to allow expression of such error reports to the language of the original model, so that the model maker can observe in his own models what goes wrong.

The remainder of this chapter is structured as follows. Section 1.1 addresses the central question of this thesis, Section 1.2 gives an example application of the theory developed in this thesis, Section 1.3 describes how life cycle systems are structures in general, Section 1.4 explains the sources of inspiration for this work, and Section 1.5 explains how the thesis work is divided over the chapters.

1.1 Central thesis question

The central question of this work is *Can process checking techniques be applied to practical models?* and the answer is yes — provided that both the practical and the formal models are changed to meet this desire. The required changes are perhaps the most interesting result in this work. They spell out the deficiencies that we found in current process models, and give a direction for bringing these theories together.

An important change to the mathematical models and provers is a method to allow unbounded numbers of instances in a system under proof. An important change to the practical models has been letting go of causal relationships between events. All the work has been performed with feedback in the original modelling language in mind.

Our theoretical area is model checking of process algebras; we have enhanced model checking with a bit of theorem proving where this was required to support quantified properties. Our practical setting was object-oriented modelling; no language in particular, but state charts have influenced our work to shape a graphical notation.

Our approach, in which we try to satisfy goals in two areas of expertise, is not common. The state of the art in object-oriented modelling is that parts of UML state charts are formalised with conducing proofs as a goal, but the models are usually not captured completely, and choices of the meaning of the notational standard must be made. The state of the art in model checking is that it can efficiently check, and provide good feedback on, designs with a large but fixed number of states, whereas theorem proving can hardly be fully automated and yields output which is unpractical.

In the evaluation chapter we will compare our work to work in both areas, starting from the goals of each area; if we succeed in fulfilling the goals of both areas to a sufficient degree, we can conclude that we have made a useful connection between the areas.

1.2 An example application

In this section we introduce an example of our process language, called *life cycles*. Consider the diagram in Figure 1.1. This is a life cycle, where transitions selectively cause transitions in instances. For example, the marry(this,?partner) transitions respond to the occurrence of a marry actual event with concrete parameter values. Only the life cycles with this equal to the first parameter make a transition and store the value of the second parameter in the partner variable.

The transition birth(this) is also a normal transition, responding to the occurrence of a birth event. We imagine an infinite number of new Person instances to be ready in a ‘prenatal’ state, and only the one with the right identity stored in the special variable this makes the transition to the Single state; this is also known as instance creation.

Formally, the order of parameters is important, but for brevity’s sake

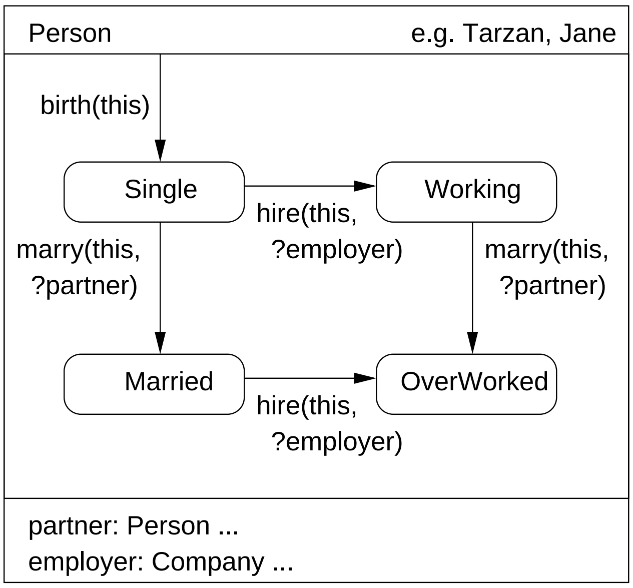


Figure 1.1: Life cycle for a Person, who can marry and be hired.

we shall assume the marry event to be an exception. Then, if a marry event with concrete parameters occurs, like marry(Tarzan,Jane), it is observed by both the life cycle instances Tarzan and Jane that respond by each making a transition, either from Single to Married or from Working to OverWorked.

If one life cycle can observe the occurrence of events, then so can two or more. For instance, the life cycle in Figure 1.2 observes an event hire that is also observed by persons; if two life cycles respond to the same event, they are said to *communicate* about hiring personnel.

This company life cycle sets strict procedures regarding hiring of new personnel, in an attempt to avoid that partners can form an information leak to competitors. So, Single people can be hired, or married people with jobless (that is, Married instead of OverWorked) partners, or people whose partner already works for the hiring company. These constraints are captured in the constraints in square brackets; a transition can only take place if that condition is not violated.

Process verification. It is our intention to perform verification on process structures such as our life cycles. In our example, it would be interest-

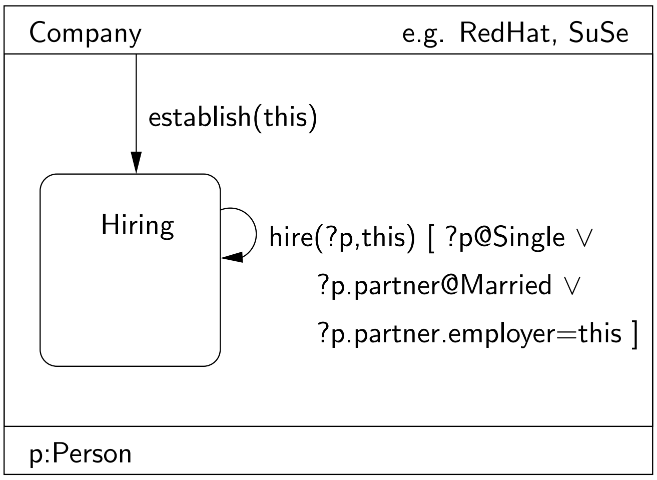


Figure 1.2: Life cycle of a Company that fights spying.

ing to find out if the hiring procedures practiced by companies ensure that no married couple can ever have partners working for different companies. In more formal notation, we would require that

$$\begin{aligned}\mathit{NoSpies} \;\equiv\;& AG\,\forall p, p' : \mathsf{Person} \cdot p.\mathsf{partner} = p' \;\Rightarrow\\ & p@\mathsf{Married} \vee p'@\mathsf{Married} \vee (p.\mathsf{employer} = p'.\mathsf{employer})\end{aligned}$$

where $p@s$ indicates that life cycle instance p is in state s.

In this case, there is one way of invalidating the constraint, and a good model checker could produce output as in Figure 1.3. This trace shows what can go wrong: If two persons are first hired by different companies and then marry, the constraint is invalidated. Such feedback is quite useful to improve designs or constraints, or perhaps make assumptions explicit, until they are accepted by the verifier.

1.3 Infrastructure of the life cycle system

This thesis introduces the concept of *life cycles* to bridge the domains of practical process modelling and mathematical process modelling. The intention of this bridge is to make process checking techniques available in practical settings, so that process checking can be performed on models just like type checking is now performed.

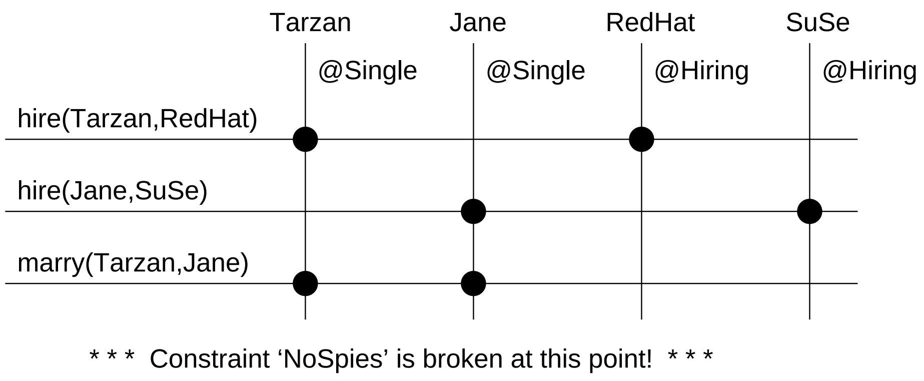


Figure 1.3: Possible feedback from a good constraint verifier.

A life cycle system is based on the practical (but minimalistic) process notation of life cycles, like in Figure 1.1. These can be translated to a temporal logic, named Bill, the Basic Inference Logic for Life cycles.

We translate life cycles to a number of axioms, as well as a number of requirements, both expressed in Bill. Furthermore, there can be manually specified requirements and axioms in William, the Weedy Inference Logic for Life cycles Including Appealing Meta-phrases, which is a syntactically sugared flavour of Bill. The axioms are used to prove or disprove the requirements, and a counter-example is constructed when the proof fails. This counter-example can be translated back to a graphical notation like in Figure 1.3, to make it practically usable (and used).

1.4 Inspiration from related work

This section discusses several sources of inspiration for this work. Its goal is to sketch the background for this research. More detail can be found in Chapter 2.

Object modelling. To improve reusability and maintainability of software, the object paradigm aims to localise concepts in exactly one place. Concepts are therefore represented in a design as directly as possible. The concepts can be instantiated, and reference each other over a globally unique identifier. State diagrams assume that a finite number of states suffices in all situations, because (a) inherently infinite types such as integers need not be modelled in states but just use another syntax, and (b) when unbounded state counts are needed, this is modelled by either grouping states, or by instantiating an unbounded number of instances of some class. Similar assumptions are also made for the research presented in this thesis.

Formal specification. We approached system specifications from the angle of object modelling. A different approach to specification is *formal specification* in a mathematical-style language. These languages have some properties that are absent yet desirable in object models, notably their support for *redundancy* and *non-determinism*. With redundancy, we mean the ability to state the same thing more than once without changing the expressed meaning; this is simple for relations, like $x' = x + 1$, but it is incorrect for assignments, like $x := x + 1$. With non-determinism, we mean the ability to specify partially undefined results; for example, $x' > y$ does not specify exactly one value for x', but it describes a *set* of possible values for x'. The combination of redundancy and non-determinism makes it possible to combine partial models which partially overlap (without contradiction), to form a more complete specification. These concepts are rather helpful in achieving composable specifications.

Another important aspect of formal models is that they are *precise*. Precision means that only one interpretation of a specification exists, avoiding common miscommunications on the interpretation of model notations, such as the meaning of names of classes and associations in a class diagram.

Furthermore, precise systems can often be verified and counter-examples generated (for example, think of type checking and the work on protocol checking presented in Chapter 2). For such reasons, several attempts are being made to formalise state charts [Bee94] [LMM99] as used in UML-ish modelling languages; often such work is incapable of covering every aspect of the complex models. And this brings us to a last important aspect of formal models: they often have inherent elegance and simplicity, which aids in developing a good intuition for a concept.

At this point, we should remark that a few object models exist, like ROOM [SGW94] and KISS [Kri94], that are supported with tools to generate code and check model consistency. Unfortunately, these modelling languages have their own deficiencies: ROOM models systems that cannot deal with an unbounded number of instances of classes, and KISS's object interaction diagrams fixate the number of collaborators in an operation.

Components. A modern approach to distribution of object systems is to plug components onto an object bus, such as CORBA [OMG95]. The idea roughly is that a *component* is a unit of distribution, ready to be composed with other components that were developed independently.

To improve extensibility of component systems, a common approach is to send around *events* to anyone interested. The idea of an event, based on the observer design pattern [GHJV86], is that any (dynamically extensible) number of parties can subscribe to them. Unfortunately, events are one-way traffic only, not making it possible for a receiver to specify if it is capable of processing events. As a result, there is no real guarantee to be drawn from having sent an event, not even that every interested party has processed it.

1.5 Thesis overview

Each chapter in this section sets out to answer a number of central questions. These are as follows:

1. **Introduction:** What is this work about?
2. **Historical background:** What are the experiences and assumptions behind this work?
3. **Life cycle syntax:** What is the life cycle notation? Informally, how do life cycle systems execute?
4. **Life cycle semantics:** What are the formal semantics of life cycles? What is the mathematically expressive power of life cycles?
5. **Conducting proofs:** How to prove properties about life cycles?
6. **Practical applications:** How to translate life cycles to an implementation? How close are life cycles to commonly used practical notations?
7. **Evaluation:** How well do life cycles fit in the object paradigm? How do life cycles compare to process algebras? What conclusions can be drawn?

Chapter 2

Historical background

Plan to throw one away.

F.P. Brooks Jr.

This chapter presents initial work of protocol checking. This work has served as a learning experience and provides a useful background for the remainder of this thesis. We first introduce our Protocol Assuring Universal Language, or Paul for short, and then we discuss the lessons learnt from this work.

2.1 Introduction to Paul

The Protocol Assuring Universal Language, or Paul, was introduced in our earlier work [RF99], and it has been a source of inspiration for the work in this thesis. This chapter discusses the previous work, and explains what lessons we learnt from working on Paul.

Interfaces exported by objects are usually unsorted collections of messages, annotated with types. The types help in assuring type correctness, either at run-time or compile-time, depending on the language. Type correctness is considered important because it eliminates a large amount of conceptual errors from designs. This is also why dynamically binding backbone architectures, such as CORBA [OMG95], offer typed interfaces (IDL in the case of CORBA).

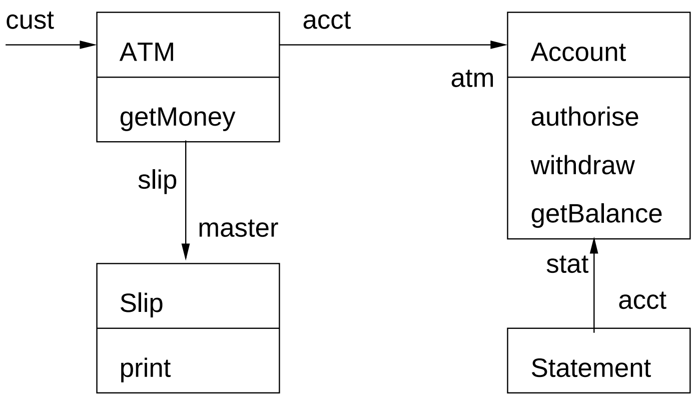


Figure 2.1: Class diagram for the banking example.

Type disciplines enable generic reasoning on data aspects of designs. The care with which type checking establishes integrity of data manipulations is in sharp contrast with the ad hoc approaches with which process aspects are verified in designs. This chapter is a step on the road of integrating process aspects into object-oriented designs more systematically. The specific aspect of processes dealt with in this chapter is the order in which invocations over object interfaces take place. We refer to such an ordering aspect with the term *protocol*.

We will demonstrate the failure of type checking to detect certain conceptual errors that *can* be detected with protocol checking. We will introduce a formalism for protocol correctness inside a single class as well as between classes, together resulting in protocol correctness of an entire application.

> **Example.** Throughout this chapter, we use the example of a bank account, which is accessed by an automated teller machine, or ATM, given in Figure 2.1. The application is an ATM that accesses an Account to withdraw money from it. When this is done, a withdrawal slip is printed by the Slip class. Independent of this, the Account may be accessed by the Statement class in an attempt to retrieve the current balance for printing a monthly statement.

Normally, associations between classes represent a data relation, which is a thought that originates from the entity-relationship background of object orientation. In this chapter however, we will interpret an association as a process, that consists of messages passing over the association, and is described by the protocols at both sides of the association. We limit ourselves to directed 1:1 associations (from the invoking client to the invoked server, though with a provision for feedback from the server to the client); in the diagrams they are drawn as arrows, pointing from client to server.

Lacking a specification of message order on the interfaces, no algorithm can ever decide whether classes can be associated in such a way that expected and offered orders match. We therefore believe we introduce a fundamental improvement of interface descriptions by extending them with *protocols*, or *order descriptions of messages.*

> **Example.** Consider the messages supported by the Account class. These include authorise and withdraw. The conventional interface toward the environment does not reveal *any* required or allowed order. Only by applying real world knowledge is it obvious that an authorise must be invoked before a withdraw.
>
> But even knowing this, after an authorise and a subsequent withdraw, it is undefined whether another withdraw is possible without a preceding authorise. This varies in the real world, thereby introducing a possible problem when the Account and ATM are not implemented at the same time by the same person.

A single class may offer different behaviours to different clients; it plays, so to say, different *roles* for different clients. Therefore, a class in general has several roles and each role is an interface (including a protocol). The idea of multiple roles on classes is not new; see [HO93] [KØ96] [LM95].

> **Example.** Class Statement should be allowed to retrieve balance information from Account. Some banks do not allow an ATM to retrieve such information, for reasons of privacy and security. We model this with role Account:atm which is specifically meant for the association to the ATM. Since a role describes which messages are acceptable at what time, there must be a separate protocol for each role.

Protocols have are used in the OO world informally, hardly supported by language, theory and tools. State diagramming techniques, for instance, are widely used in OO developments; our protocols are state diagrams per role. As another example, consider interaction diagrams in the design patterns world. For many of the patterns, Gamma et al. [GHJV86] give type specifications with comments, but because that is not sufficient to understand the class, they give an interaction diagram. If an interaction diagram is so important and useful, why is not something like it part of the language? Our protocols will do.

Note that, in general, not only the method names in the history of method calls, but also the parameter values determine the state of an object, and therefore what next messages are expected and allowed. Our approach covers this situation, since each element (formally called 'message name') in a protocol may be interpreted as a method name *plus* its parameter values.

Before delving into the details, let us first describe informally the main concepts, their names, and their interrelations. We call the interfaces that offer services to client classes the *export roles* of the server class. At the other end of the association, the client side, there is an *import role*. In Paul, each role has its own protocol. In addition, each class has a class protocol, describing the life cycle of the class' objects.

Seen from within an association, the protocol I of an import role expresses the behaviour that is expected from the client; the protocol E of an export role expresses behaviour that is offered by the server. An important check performed on Paul designs is whether I *correctly uses* E, denoted by $I \leq E$. All such checks together assure *association protocol correctness* of a design.

Seen from within a class, a protocol E of an export role is a perspective on the class protocol C; and C in turn describes in what order the class' methods will be called, and thus how use is made of an import protocol I. A check done on Paul designs is whether E is a correct use of C, and whether C via its methods correctly uses I, both being formalised by means of $\leq$ again. All such checks together assure *class protocol correctness*.

The combination of association and class correctness leads to *overall protocol correctness* of a design.

By deliberate choice we do all checking locally. Consequently, when an object has several export roles, there is no check whether the messages over the associations arrive at the object arbitrarily interleaved or in a specific order determined by the environment. In order to relieve the designer from the obligation to choose each class protocol as the interleaving of its export protocols, we will assume that each object has a *scheduler*: It lets the messages over the (protocol correct!) links pass into the object in a way that is acceptable to the class protocol. The scheduler is to be generated *automatically*.

Finally, in Paul, messages have no return values. In order to express *feedback*, we will interpret a nondeterministic choice $a.p + a.q$ as a single message a with two return values; one leading to continuation p and the other to q. This interpretation is nicely formalised with CSP [Hoa85], and fits well with conventional use of state transition diagrams, as explained in the next section.

The results as presented in this chapter are not at a level suitable for practical application. At first, there is no proof of termination of a program, even if it is proven correct — our checker tool Paul does not perform the global analysis required for this certainty. Second, the cardinality of all associations is assumed to be 1:1, which is not usable in practice — although it is possible at this time to generalise the checker to cover more general associations, it results in more detailed specifications. In this chapter, we shall accept these shortcomings; it is the purpose of the following chapters to uncover a practically usable notation.

The remainder of this chapter is structured as follows. Section 2.2 introduces the syntax and intuitive meaning of Paul, including the formulation of part of the ATM example. Section 2.3 gives a formal semantics to Paul, by translating protocols to CSP. Then Section 2.4 defines protocol checking. The final sections discuss related and future research, and draw conclusions.

This work has previously been published in a paper [RF99]; it has been performed in the scope of the Quantum project, in which Compuware's UNIFACE lab and the University of Twente cooperate. UNIFACE is a leading component-based development tool for mission-critical applications. The goal of our research is to add the protocol concept to UNIFACE's component model.

2.2 Syntax of Paul

"Programs" in Paul express protocol aspects of designs (so we will speak of *design* rather than program). That means Paul contains no things like int variables or addition operators, but everything that plays a role in protocols is expressed in Paul. There are looping and choice constructs (abstracting from the actual conditions in those constructs), there are message sends, there are (behavioural) links between objects, and there are import and export roles.

A design in Paul contains several protocols: One for each class, one for each import role and one for each export role, and the main task of the Paul protocol checker is to ensure the absence of conflicts between these protocols. The checking rules are discussed in Section 2.4; this section introduces only the syntax of Paul.

Syntax. In Paul's grammar we use C for class names, R for role names (both imported and exported), M for message names, and X for protocol variables; their syntax is not elaborated. Symbol ? denotes a test whose outcome is determined by the method (client) itself, whereas the outcome of a test $R.M()$ is determined by the server. We use '...' to denote zero or more occurrences of the preceding nonterminal. The grammar is given in Figure 2.2.

In the current version of Paul, methods are parameterless and do not return a result. The conditions that control the loops and conditional statements are abstracted out of Paul when they are based upon data. Thus we cannot distinguish between 'while (3>2) s' and 'while (3<2) s': Both are represented in Paul by 'while ? do s end'. Method invocations as tests are not abstracted away; here it is the server who decides the outcome.

General recursion is not possible for method definitions *within* a class, since each method invocation 'invoke $R.M()$' in a method body refers to method M via role R. Directed loops of associations could introduce recursion outside the class, which is therefore not an approved-of design structure.

We shall leave out some parentheses in protocols under the convention that the binding strength of the operators is given in decreasing order in Figure 2.2; so . binds stronger than +.

Example. Figure 2.3 expresses the classes ATM and Account in Paul.

Consider class ATM. In its first line the class protocol is declared; it describes the possible orders in which its own method getMoney is called. (Here the class protocol getMoney* is rather trivial, but in class Account it is not.) The protocols of the import roles slip and acct are declared with an imports clause. The "typing" acct:(authorise. withdraw*)*, for instance, expresses the possible orders of invocations of the imported methods acct.authorise and acct.withdraw. We shall

design	::=	*def* ...	
def	::=	*class* | *assoc*	
assoc	::=	$C : R$ — $C : R$	// association between roles
class	::=	class C : *proto* is *decl* ... end	
decl	::=	exports R : *proto*	// export of a role, as a server
	|	imports R : *proto*	// import of a role, as a client
	|	method $M()$ is *stat* end	// method definition
stat	::=	while *test* do *stat* end	// while loop
	|	if *test* then *stat* else *stat* end	// if statement
	|	*stat*; *stat*	// sequential composition
	|	invoke $R.M()$	// method invocation
	|	ε	// (no text) no action
test	::=	$R.M()$ | ?	// decision information retrieval
proto	::=	(*proto**)	// arbitrary repetition
	|	(*proto* . *proto*)	// sequence
	|	(*proto* + *proto*)	// choice, made by client when deterministic
	|	letrec *pdef*... in *proto* end	// scoping of definitions
	|	var X	// protocol variable use
	|	M	// message send
pdef	::=	X = *proto*;	// recursive protocol definition

It is required that recursion in protocols (via letrec) is tail recursion only.

Figure 2.2: Paul's grammar

later see that this acct protocol does not correctly use the protocol of Account's export role atm. The protocol for the export role cust is declared with an exports clause; it constrains the order in which a client may invoke getMoney on the ATM.

The bottom line defines the association between the two classes.

Given the class definitions only, a Paul protocol checker can verify for *class* protocol correctness. *Association* protocol correctness can be checked when the associations are known too, as in the bottom line of Figure 2.3. A complete Paul *design* is a combination of class and assoc definitions.

Example. The check performed for association ATM:acct — Account:atm compares protocol (authorise.withdraw*)* of the import role

```
class ATM:getMoney* is
  imports slip:print*
  exports cust:getMoney*
  imports acct:(authorise.withdraw*)*
  method getMoney () is
    invoke acct.authorise ();
    while ? do
      invoke acct.withdraw ();
      invoke slip.print ();
    end
  end
end

class Account:(authorise.withdraw + getBalance)* is
  exports atm:(authorise.withdraw)*
  exports stat:getBalance*
  method ...
end

ATM:acct — Account:atm
```

Figure 2.3: Definition of the ATM and Account class, and their association

acct of class ATM with protocol (authorise.withdraw)* of the export role atm of class Account. We shall see that the check yields the result 'incorrect.'

The check performed within class ATM between its export protocol and the class protocol, compares getMoney* with getMoney*. The outcome will be 'correct'.

The check performed within class ATM between its class protocol *together with* the method body at one hand and the import role acct at the other hand, compares the protocol (authorise.(withdraw.print)*)* —with concealment of print— with the protocol (authorise.withdraw*)*. The outcome will be 'correct'.

The same check for the import role slip compares (authorise.(withdraw.

print)*)* —with concealment of authorise and withdraw— with print*. The outcome will be 'correct'.

2.3 CSP semantics of Paul

In the next section protocol checking is defined in terms of CSP processes. CSP is the well-known theory of Communicating Sequential Processes [Hoa85]. In this section we define the translation of Paul protocols to CSP processes. We start with a brief exposition of CSP (just enough to follow the main line of our exposition), and an explanation of how feedback from server to client is realised in Paul.

CSP. CSP is an elaborate theory about possibly nondeterministic processes, with a precisely defined notion of equality. Processes P, Q, R are built, at least syntactically, from *events* a, b, c by means of the *then* operator $\rightarrow$ (having the highest priority in parsing), and two "branching" constructs $\Box$ and $\sqcap$. An important operator is the *concurrency* (or in-parallel-communicating) operator $\|$.

An event denotes "something that may happen," and a communication between two processes is defined as the occurrence of the same event in both processes. In the process $P = a{\rightarrow}Q \Box b{\rightarrow}R$ it is the environment (the communicating counter-part of P) which influences the branch that P will take; thus $\Box$ denotes *external choice*. In the process $P = a{\rightarrow}Q \sqcap b{\rightarrow}R$ it is P itself that makes the choice; thus $\sqcap$ is called *internal choice*. There are several laws describing the semantics more precisely; in particular:

$$(a{\rightarrow}P) \Box (a{\rightarrow}Q) \quad = \quad a{\rightarrow}(P \sqcap Q) \tag{2.1}$$

$$(a{\rightarrow}P) \sqcap (a{\rightarrow}Q) \quad = \quad a{\rightarrow}(P \sqcap Q) \tag{2.2}$$

Each process P has an alphabet, denoted αP, consisting of all the events in which it possibly can participate. Concealment, or hiding, of a set A of events is denoted $P \backslash A$, and means that the events A within P occur autonomously without an opportunity for the environment to participate.

Successful termination of a process is represented by the event $\surd$. Because of its interpretation, the event often gets special treatment. The process that just terminates successfully is denoted Skip; so Skip $= \surd{\rightarrow} \ldots$, where the remainder process on the ellipses is irrelevant. Except for this special interpretation, $\surd$ is a normal event, so it is part of alphabets of processes under the customary rules.

A recursively defined process like $X = \ldots X \ldots$ is written $\mu X \cdot \ldots X \ldots$. For example, $\mu X \cdot a \rightarrow X$ is the process that is only and forever able to participate in event a.

The mathematical semantics of the constructs and notions above is described in terms of *traces* and *failures*. A trace of a process is a sequence of events in which the process may possibly participate, in succession, during a run. A failure A of a process P after a trace s, denoted $A \in \mathit{Failures}(P, s)$, is a set of events that P may refuse to participate in after trace s. Of particular importance are the facts (definitions!) that for the empty trace ε:

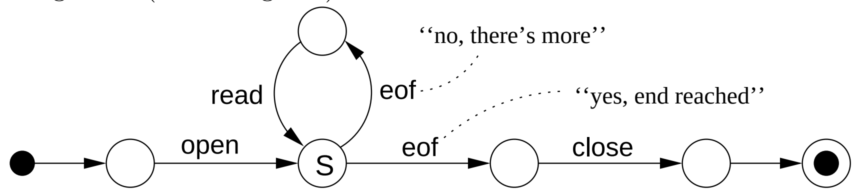
$$\begin{aligned} \mathit{Failures}(P \sqcap Q, \varepsilon) &= \mathit{Failures}(P, \varepsilon) \cup \mathit{Failures}(Q, \varepsilon) \\ \mathit{Failures}(P \,\Box\, Q, \varepsilon) &= \mathit{Failures}(P, \varepsilon) \cap \mathit{Failures}(Q, \varepsilon) \end{aligned}$$

Feedback. Recall that in the version of Paul considered in this paper all associations are directed, from client to server, meaning that the messages over the associations are sent by the client and received by the server. Moreover, in our formalism messages do not return resulting values. Together with the approach to perform checks locally, this implies that a dialogue between two objects will be hard to model in a way that passes the protocol checks.

Our solution to this weakness is to model the process aspects of *feedback* from server to client. This is achieved by our interpretation of nondeterministic choice in a protocol (typical example: A sub-expression $a.p + a.q$) as a server choice, and of other choices (such as $a.p + b.q$ with distinct a and b) as client choices. This interpretation fits well with conventional use of state transition diagrams, as shown by Figure 2.4 below. Thus the designer, the Paul user, is freed from explicitly indicating at various places (export, class, and import protocol) whether the choice is a client choice or a server choice.

Let us show this feedback by two typical examples (in anticipation of the formal definitions below). Consider the first state transition diagram in Figure 2.4. It expresses the protocol for opening a file and reading to end-of-file: the two `eof`-transitions indicate a single invocation of the `eof`-method with a negative and a positive return value, respectively. Here the decision whether the end of file has been reached (the negative or positive return value) is be made by the server, the object that exports the methods to access the file. After our translation into CSP, this is reflected by an internal choice $\sqcap$ in the server side protocol.

Reading until (server signals) end-of-file:



in Paul: $\mathsf{open}.S$ where $S = (\mathsf{eof}.\mathsf{read})^*.\mathsf{eof}.\mathsf{close}$

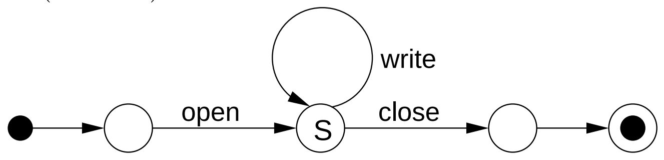
$= \mathsf{eof}.\mathsf{read}.S + \mathsf{eof}.\mathsf{close}$

in CSP: $\mathsf{open}{\rightarrow}S$ where $S = (\mathsf{eof}{\rightarrow}\mathsf{read}{\rightarrow}S) \,\Box\, (\mathsf{eof}{\rightarrow}\mathsf{close}{\rightarrow}\mathsf{Skip})$

$= \mathsf{eof}{\rightarrow}(\mathsf{read}{\rightarrow}S \sqcap \mathsf{close}{\rightarrow}\mathsf{Skip})$ [law (2.1)]

The CSP formulation is from the *server*'s perspective.

Writing until (client is) done:



in Paul: $\mathsf{open}.S$ where $S = \mathsf{write}^*.\mathsf{close}$

$= \mathsf{write}.S + \mathsf{close}$

in CSP: $\mathsf{open}{\rightarrow}S$ where $S = \mathsf{write}{\rightarrow}S \,\Box\, \mathsf{close}{\rightarrow}\mathsf{Skip}$

The CSP formulation is from the *server*'s perspective.

Figure 2.4: Protocols with/without feedback

Now consider the second state transition diagram in Figure 2.4. It expresses the protocol for writing a series of data onto a file. There is no nondeterminism: the decision whether to continue writing is made by the client, the object that imports the write method. After our translation into CSP, this is reflected by an external choice $\Box$ at the *server* side of the association.

Switching of perspective. To change the perspective from server to client or vice versa we now define overbar operation $\overline{}$ (pronounced *switch*). Thanks to this operation, one translation $\mathcal{T}$ from protocols to processes suffices, both for 'server side' protocols and for 'client side' protocols. In principle, operation $\overline{}$ simply interchanges the choices $\Box$ and $\sqcap$ in a CSP process. The exchange is defined as a purely syntactic operation, and must only be applied when the process has the so-called *determined choice form*, that is, no sub-expression has the form of the left hand sides of CSP laws (2.1) or (2.2) (repeated below), nor can it be brought in this form using associativity and commutativity of $\Box$ and of $\sqcap$:

$$(a\to P)\,\Box\,(a\to Q) \quad = \quad a\to(P \sqcap Q) \tag{2.3}$$

$$(a\to P)\sqcap(a\to Q) \quad = \quad a\to(P \sqcap Q) \tag{2.4}$$

There are two reasons for the condition on the form on which the replacement takes place. First, without the condition, the replacement would sometimes not have the intended effect; for example, the replacement in the left hand side of (2.2) has no semantic effect due to law (2.1). Second, without the condition on the form, semantically equal expressions would result in semantically distinct expressions; for example, this would happen when the replacement takes place throughout law (2.1), or (2.2).

Thus the definition of operation $\overline{}$ on CSP processes reads:

```
repeatedly apply laws (2.1) and (2.2) from left to right
   (until no subexpression has the form of the lhs of these laws
   modulo associativity and commutativity of ⊓ and □);
replace each □ by ⊓, and each ⊓ by □.
```

Clearly, the operation satisfies the property that $\overline{\overline{P}} = P$. A particularly noteworthy idiom that we shall use is:

$$\overline{\overline{P} \sqcap \overline{Q}}$$

In this way we express a “really external choice between P and Q,” even though P and Q may start with the same event. For instance, take $P = a{\to}P'$ and $Q = a{\to}Q'$. Then $P \Box Q$ equals $a{\to}(P' \sqcap Q')$, due to law (2.1), thus failing to express an external choice. However, assuming that P' and Q' do not start with the same event, we have:

$$\begin{array}{rcll}
\overline{\overline{a{\to}P'} \sqcap \overline{a{\to}Q'}} & = & \overline{a{\to}\overline{P'} \sqcap a{\to}\overline{Q'}} & \\
 & = & \overline{a{\to}(\overline{P'} \sqcap \overline{Q'})} & \text{[law (2.2)]} \\
 & = & a{\to}(\overline{\overline{P'} \sqcap \overline{Q'}}) & \\
 & = & a{\to}(\overline{\overline{P'}} \Box \overline{\overline{Q'}}) & \\
 & = & a{\to}(P' \Box Q') &
\end{array}$$

So the choice is really external.

Paul protocol to CSP process. $\mathcal{T}$ is the function that translates a Paul protocol (*proto* in Figure 2.2) to a CSP process with CSP's $\Box$ for Paul's $+$. As explained above, it is defined by induction on the structure of the protocol, using a continuation parameter P for “what comes after the protocol”:

$$\begin{array}{lcl}
\mathcal{T}(m, P) & = & m{\to}P \\
\mathcal{T}(p.q, P) & = & \mathcal{T}(p, \mathcal{T}(q, P)) \\
\mathcal{T}(p + q, P) & = & \mathcal{T}(p, P) \Box \mathcal{T}(q, P) \\
\mathcal{T}(p^*, P) & = & \mu X \cdot \mathcal{T}(p, X) \Box P \\
\mathcal{T}(\mathsf{letrec}\ X{=}p\ \mathsf{in}\ q\ \mathsf{end}, P) & = & \mathcal{T}(q[\mathsf{var}\ X/(\mu X \cdot p)], P) \\
\mathcal{T}((\mu X \cdot p), P) & = & \mu X \cdot \mathcal{T}(p, P) \\
\mathcal{T}(\mathsf{var}\ X, P) & = & X
\end{array}$$

In the letrec clause, each occurrence of var X within q is replaced by the recursive protocol $\mu X \cdot p$ (which requires an extension of the syntax with μ expressions). In the last clause, we see that a recursive call *recurs* and therefore discards the continuation P.

Observe that $\mathcal{T}(a.p + a.q, \mathsf{Skip}) = (a{\to}P) \Box (a{\to}Q) = a{\to}(P \sqcap Q)$, where $P{=}\mathcal{T}(p, \mathsf{Skip})$ and $Q = \mathcal{T}(q, \mathsf{Skip})$, as required.

A Paul protocol p in isolation is then translated to CSP as: $\mathcal{T}(p, \mathsf{Skip})$.

Paul statement to CSP process. $\mathcal{S}$ is the function that translates a Paul statement (*stat* in Figure 2.2) to a CSP process (where ‘internal’ is 'client side'). Here we have to bear in mind that a test ? denotes a client choice ($\sqcap$), whereas a test $R.m()$ denotes a server choice. The latter is expressed by means of the idiom $\overline{\overline{P} \sqcap \overline{Q}}$ explained above. For simplicity we assume that the names of all imported methods are distinct, so that the role name R in $R.m()$ is superfluous. With these remarks in mind, the definition suggests itself:

$$\begin{array}{lcl}
\mathcal{S}(\varepsilon, P) & = & P \\
\mathcal{S}(R.m(), P) & = & m \rightarrow P \\
\mathcal{S}(s; t, P) & = & \mathcal{S}(s, \mathcal{S}(t, P)) \\
\mathcal{S}(\mathsf{if}\ ?\ \mathsf{then}\ s\ \mathsf{else}\ t\ \mathsf{end}, P) & = & \mathcal{S}(s, P) \ \sqcap\ \mathcal{S}(t, P) \\
\mathcal{S}(\mathsf{if}\ R.m()\ \mathsf{then}\ s\ \mathsf{else}\ t\ \mathsf{end}, P) & = & m \rightarrow \overline{\overline{\mathcal{S}(s, P)} \ \sqcap\ \overline{\mathcal{S}(t, P)}} \\
\mathcal{S}(\mathsf{while}\ ?\ \mathsf{do}\ s\ \mathsf{end}, P) & = & \mu X \cdot\ \mathcal{S}(s, X) \sqcap P \\
\mathcal{S}(\mathsf{while}\ R.m()\ \mathsf{do}\ s\ \mathsf{end}, P) & = & \mu X \cdot\ m \rightarrow \overline{\overline{\mathcal{S}(s, X)} \sqcap \overline{P}}
\end{array}$$

The translation of a statement s in isolation is defined to be: $\mathcal{S}(s, \mathsf{Skip})$.

Paul class protocol plus methods to CSP process. In order to compare a class protocol with the import protocols, the method definitions have to be taken into account. Function $\mathcal{M}$ translates a class protocol *together with the method definitions* to a CSP process. We assume that for each method name m the body is given by *body* m.

The definition differs from $\mathcal{T}$'s definition only in the clause for a method name m and in the use of $\sqcap$ instead of $\Box$, since, as explained earlier, when a class protocol is related to its import roles, the class is considered the client and the import role is considered the server:

$$\begin{array}{lcl}
\mathcal{M}(m, P) & = & \mathcal{S}(\mathit{body}\ m, P) \\
\mathcal{M}(p.q, P) & = & \mathcal{M}(p, \mathcal{M}(q, P)) \\
\mathcal{M}(p + q, P) & = & \mathcal{M}(p, P) \sqcap \mathcal{M}(q, P) \\
\mathcal{M}(p^*, P) & = & \mu X \cdot \mathcal{M}(p, X) \sqcap P \\
\mathcal{M}(\mathsf{letrec}\ X{=}p\ \mathsf{in}\ q\ \mathsf{end}, P) & = & \mathcal{M}(q[\mathsf{var\ X}/(\mu X \cdot p)], P) \\
\mathcal{M}((\mu X \cdot p), P) & = & \mu X \cdot \mathcal{M}(p, P) \\
\mathcal{M}(\mathsf{var\ X}, P) & = & X
\end{array}$$

The translation of a class protocol p in isolation is defined to be: $\mathcal{M}(p, \mathsf{Skip})$.

2.4 Protocol checking

Within the framework of CSP, a relation *correctly uses*, denoted $\leq$, is defined between processes. The checks performed for each association, and for each class, are based on the relation $\leq$.

In order to avoid complications in the formulæ, we assume that within each class the alphabets of the export protocols are disjoint and contained in the alphabet of the class protocol, and similarly that the alphabets of the import protocols are disjoint and jointly contain all method names in the method bodies of the class. Relaxing the disjointness assumption would introduce some explicit renaming in the formulæ and thus obscure the essentials.

Recall further that another, serious, restriction is that associations are 1:1 only. The life cycle diagrams introduced in the next few chapters will loosen this constraint (in a different language, not in an extension of Paul).

Association correctness. Consider an association with protocols C and S at the end points. Both C and S have been obtained by translating the Paul protocols into CSP expressions by function $\mathcal{T}$. We shall assume $\overline{C}$ as a correct description of the client's behaviour over the association, and S as that of the server: $\overline{C}$ and S correctly describe independently what messages occur and who takes decisions. (This assumption is checked for in the paragraphs on Class correctness below.)

The question is now: When does the client correctly use the server? The answer is: When during each possible execution of $\overline{C} \parallel S$ (in which $\overline{C}$ and S proceed in a lock-step synchronised way, making the decisions as indicated by $\Box$ and $\sqcap$) it never happens that a party insists on participating in another action, yet the other party can refuse the actions offered. So, as long as there is no successful termination, the concurrent composition makes progress, and semantically there is no deadlock in this composition (though deadlock may still arise in the entire system). In CSP terminology, given that C and S have the same alphabet, A say, we define that C *correctly uses* S precisely when $\overline{C} \parallel S$ does not have a pair (s, A) in its failures whenever trace s has not signalled successful termination:

$$C \leq S \quad \equiv \quad \overline{C} \leftrightarrow S \tag{2.5}$$

$$P \leftrightarrow Q \quad \equiv \quad \forall s \cdot \quad \langle \surd \rangle \mathsf{\ in\ } s \quad \vee \quad (s, A) \notin \mathit{Failures}\ (P \parallel Q) \tag{2.6}$$

$$\text{where } A = \alpha P = \alpha Q$$

Relation $\leftrightarrow$ is pronounced 'co-operate well'. So, leaving the translation $\mathcal{T}$ from Paul to CSP implicit, we define that the check performed for each association is:

$$C \quad \leq \quad S$$

where C is the client's import protocol, and S is the server's export protocol.

Example. To illustrate the relation $\leq$, here are some examples:

authorise	$\leq$	authorise + getBalance
authorise + getBalance	$\not\leq$	authorise
authorise + getBalance	$\leq$	authorise + getBalance
authorise.(withdraw + getShot)	$\not\leq$	authorise.getShot
authorise.withdraw + authorise.getShot	$\leq$	authorise.getShot
(authorise.withdraw*)*	$\not\leq$	(authorise.withdraw)*

The latter example is the faulty association ATM:acct — Account:atm in the ATM example. When translated to CSP its client and server side protocols $\overline{C}$ and S become (with a for authorise, and w for withdraw):

$$\begin{aligned} \overline{C} &= (a \to P) \sqcap \mathsf{Skip} \quad \text{where} \quad P = (a \to P) \sqcap (w \to P) \sqcap \mathsf{Skip} \\ S &= (a \to w \to S) \,\Box\, \mathsf{Skip} \end{aligned}$$

Taking trace s to be $\langle a \rangle$, we see that $\neg(\langle\checkmark\rangle \text{ in } s)$ and $(s, \{a, w, \checkmark\}) \in$ *Failures* $(\overline{C} \parallel S)$. This proves $C \not\leq S$, meaning that the ATM example contains a protocol error.

Class correctness I. Here we consider what relation should hold between the export protocols and class protocol (in order to declare the class protocol correct), and in the next paragraph the relation between the class protocol and import protocols.

Suppose a class with class protocol $a.b$ has two export roles with protocols a and b, respectively. See the left side of Figure 2.5. Clearly, serving b first followed by a is in conflict with the class protocol. Even though it might follow from the actual environment that a will occur before b, it seems that the class is *not* protocol correct since, by deliberate choice, protocol correctness is a local property.

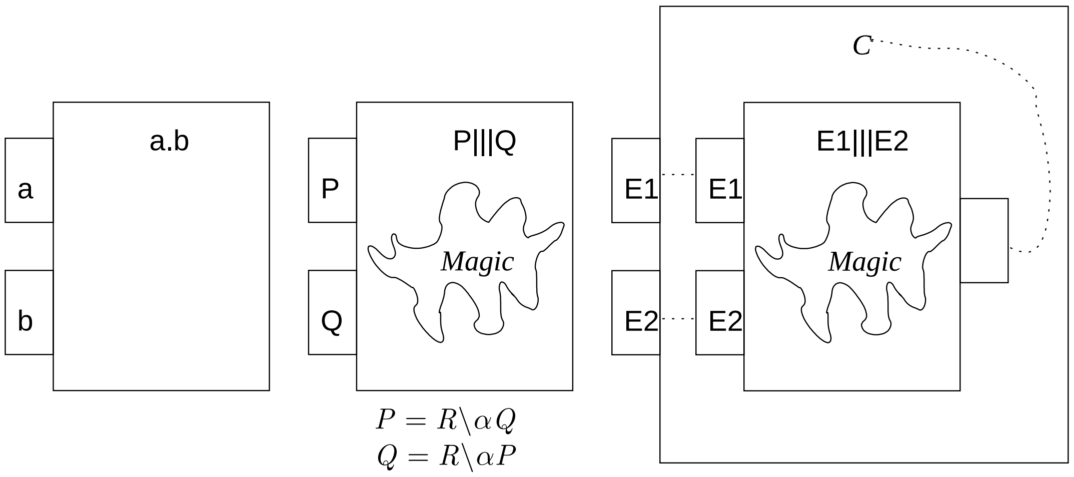


Figure 2.5: Left: a class; mid: a scheduler; right: class with scheduler

Our solution to this weakness is to assume an implicit *scheduler* in a class. A scheduler of protocols P and Q is an object that is able to serve *all* interleavings of P and Q, and that only requests *some* interleaving of these from its server. (It does so, apparently, by alternately holding messages of some roles while propagating messages of other roles.) Formally, a scheduler of P and Q has export protocols P and Q, class protocol $P \interleave Q$ (P *interleaved* with Q), and an import protocol R that satisfies $P = R\backslash\alpha Q$ and $Q = R\backslash\alpha P$; see Figure 2.5. The internals of the scheduler are just *magic* and of no concern to the designer: It is to be generated *automatically*. For example, the tuple $(P, Q; R)$ with $P{=}a$, $Q{=}b$ and $R{=}a.b$ specifies a scheduler: Assuming that it is implicitly placed into the class as in the left of Figure 2.5, the class is protocol correct regarding the export roles and class protocol. Thus, once more, the schedulers must occur in an actual object system, and so our object systems differ from what is conventional.

The formal correctness requirement between the export protocols $\vec{E}$ and the class protocol C reads now as follows: There exists some scheduler $(\vec{E}; E')$ that serves all interleaving of the class' export protocols and whose request correctly uses C:

$$\exists E' \cdot (\forall i \cdot E_i = E' \lceil \alpha E_i) \wedge E' \leq C \tag{2.7}$$

Here we use $P \lceil X$ as an abbreviation for $P\backslash(\alpha P - X)$. Again we have left the translation $\mathcal{T}$ from Paul to CSP implicit.

Class correctness II. Now we consider the protocol correctness requirement regarding the class protocol C and the import roles $\vec{I}$. This will again be expressed in terms of correct use of protocols derived from C and $\vec{I}$.

Recall that all choices expressed within C are decided by either the clients of this class or by the class' internal workings; only choices within the method bodies can possibly be decided through the import roles. This interpretation is given by translation $\mathcal{M}$.

Now, observe that the actual use of the class' methods is described by the class protocol C. The import roles $\vec{I}$ say what services have been imported, for use by the method bodies. So, process $\mathcal{M}(C)$ should cooperate well with the processes of the import roles. Given that the import roles have disjoint alphabets, this requirement reads:

$$\overline{\mathcal{M}(C) \upharpoonright \alpha I_i} \quad \leq \quad I_i \qquad \text{for all } i \tag{2.8}$$

Note the use of $^{-}$ to change the class' protocol from client side to server side protocol, which is expected by the $\leq$ relation.

2.5 Tool support for Paul

An implementation of Paul exists. The syntax from Figure 2.2 (with some trivial extensions) is supported by Paul 1.0. The protocol checker can be obtained from ftp://ftp.cs.utwente.nl/pub/doc/Quantum/Paul.

The implementation is a C-program with bits of Lex and Yacc. The tool functions quite well, but it has not been designed as a verifier tool from the start. It merely 'grew' in a spontaneous process that started with the idea of a syntax checker.

The tool prints a counter-example in case of detected protocol errors. For the faulty associations under paragraph "Association correctness," the counter-examples generated are ⟨getBalance . . .⟩, ⟨authorise.withdraw . . .⟩ and ⟨authorise⟩, respectively.

The tool has been used to demonstrate workflow checking for a toy case (quite literally: A wish list administration for Santa Claus); this demonstration can be obtained from ftp://ftp.cs.utwente.nl/pub/doc/Quantum/Paul/Santa-1.0-2.tar.gz, and it is described in a published workshop paper [Rei99]. The tool and this demo demonstrate the importance of good feedback, such as the traces above. Without such concrete information, it is extremely hard to debug a process specification.

2.6 Extensions to Paul

Based on the above, a number of extensions were made to the Paul theory and tool. We shortly describe each of these.

Proof of termination. The checks in definitions 2.5 and 2.6 are sufficient conditions for correct communication on a link. It is not possible to derive termination of the whole system from these definitions, only termination of the association being tested, and only when the environment (the other processes in a design) cooperate.

This means that the Paul prover derives no knowledge about global termination. Such a check is sufficient for batch programs, but modern programs are increasingly interactive, and therefore demand stronger proofs. What we tried to add to Paul to solve this, is an additional check if the application may deadlock. The complete protocol check then becomes the current check conjugated with deadlock freedom. This work is described in an MSc thesis [Str98] but has not been fully integrated into Paul.

Instance handling. Several practical aspects of object programs still remain unsolved: Multiple instances of the same role should be allowed, as opposed to exactly one. Methods should support parameters, as far as these are of influence on protocols. Assignment to role references and inheritance between Paul classes are also absent.

This topic can exploit the richness of protocols in Paul to solve the problems that remain in the work of Kurshan and McMillian [KM89] who presented an induction theorem that makes it possible to check compositions of an unspecified number of protocols. This basically means that Paul administers instances as sets of instances with similar properties, but it also means that far less programs get accepted.

Synchronisation, transactions, workflow. Protocols are one aspect of a more complicated notion of process that we wish to integrate in the object paradigm. Additional aspects to cover with these processes are (automatic) synchronisation and transactional/workflow aspects to deal with failing executions. Extensions in all these directions have had our attention.

2.7 Lessons learnt

So far, this chapter studied Paul, a language for assuring protocol correctness in object designs. Protocol-checking is complementary to type-checking and catches a different class of conceptual errors.

To introduce protocols in an object model which features roles, it is necessary to define a class protocol, import role protocols and export role protocols. None of these can be omitted without sacrificing features of Paul.

Paul's protocol checking is based on a 'correctly uses' relation. This relation was defined in terms of CSP semantics in this paper. A Paul design can be extracted from class diagrams with sufficient semantics, enabling protocol checks on these software representations. A Paul design can express decisions based on return values from a server.

Concluding, Paul offers flexible, precise and practical protocol checking facilities for use in object languages with roles.

The tool support for Paul has evolved, rather than that it was designed. Perhaps that is why we learnt so much from our efforts with Paul. As in software design, where it is considered smart to plan to throw one away [Bro95], we ended our work on Paul and set up a next generation process language based on experiences with Paul. To support the introduction of these so-called Life cycles in the next chapter, we discuss the invaluable lessons we learnt from Paul in this section.

Having a tool. Paul has grown from an initial idea to make a simple syntax checker, all the way to a protocol checker. The reason why it kept evolving was mainly our joy with having a tool that could verify the things we set out in theory. It has often been helpful for our research to be able to let a tool do a particular form of analysis that would have been quite long-winding to do manually.

The existence of a tool also turned out particularly interesting for Compuware, the company for which the initial part of this research was performed. The presentation of a tool, and the ability to challenge it with smart exceptional cases was a very important means of communication, far more useful than a well-crafted set of formulæ could ever have been.

The moment of checking. The level at which the Paul tool checks is a bit late. It verifies mainly programmatic structures; and although higher-level models can be represented in it, they still deal with message sends, and are thus not strictly analysis models. As a result, the feedback from the tool may come too late. We prefer a tool that is able to handle analysis models from the earliest stages that are concrete and accurate.

Feedback generation. One thing that was 'hacked into' the Paul checker was the generation of feedback traces with counter-examples. When comparing processes, the checker would follow such traces, and by just tracing back its steps, it could produce these counter-examples, which we experienced as a dramatic improvement in the usability of the tool for repairing errors that it found [Rei99], and even when we considered theorem provers at a later time, we still were concerned mainly with providing good feedback [Rei00].

The checker traversed all possible traces in a depth-first manner, and as a result, it would not always find the shortest possible counter-example. Although this was no problem for the initial version of Paul (which did not produce counter-examples), this traversal order led to counter-examples that were longer than strictly necessary. Apparently, it would have been better, had the Paul tool traversed the tree of possible traces in a width-first manner. This is also the driving force behind other model checking applications [BCCZ].

Automated scheduling. Since the Paul checker traverses all possible execution orders between a client and server protocol, it could see ahead in what would eventually block, and what would not. In some cases, a future block would not necessarily lead to a counter-example, namely when the choice to take that branch would not be made by either client or server. In such situations, it is possible to insert a 'scheduler' in between the client and server that makes choices based on the look-ahead work done by Paul.

The idea of using the analysis results to think ahead for the purpose of block-free scheduling makes it possible to accept more client/server combinations than without this idea. We like this idea, and think a next generation process model should benefit from it if possible.

Procedural structure. Paul started out with the intention of describing designs, as they are entered in case tools. For syntactical convenience, we chose to support a textual notation rather than state diagrammatic input.

Paul has since evolved to a (simple) programming language, which expresses program flow to pion down when and where choice on program continuations are made. Paul started to look like a programming language on its own. And although this was helpful to verify programs, our original intent was to verify models, so we could find errors in an early phase of program design.

The cause of this development was the procedural style of Paul specifications, where objects send messages to each other (thereby *invoking* methods) and receive replies. To facilitate this structure, and help passing around decisions, we were almost forced to introduce loops, choices, and so on. We developed a new programming language. Please note that a similar thing has happened to state charts. They too are intended for high-level specification, but to allow the expression of the paths actually traversed, there is a need for embedded code, usually in C++. Needless to say, that C++ code is not beneficial for analysability of the diagrams; although Paul was a tad better at that, we would still prefer to keep those details out altogether.

The lesson we learnt was to avoid certain procedural aspects. One such aspect was 'I am causing this event' — we had seen how process algebras refrain from this causal information, and how well this works. As a part of this causal relationship, there often is a direction of sending information from the invoker to the invoked. If we want to get rid of causal relationships, we need to find a solution for that as well.

Communication structure. In a procedural system, communication is usually one-to-one. Several process algebras support a similar notion. Particularly CCS [Mil80] and its successor, the π-calculus [Mil91], mimic this by pairing events such as x and $\overline{x}$, and when these communicate combine them into a hidden τ step. In effect, this means that event x cannot be observed after it has been combined with its partner event $\overline{x}$. This behaves like a one-to-one communication, where only sender and receiver are aware of the communication, and the remainder of the system can at best infer a change of state of the system (represented by the τ step).

Other process algebras, such as ACP [BW90], focus on globally visible events. In these theories, events x and y can jointly make a transition when they are defined as communication partners in a commutative and associative function γ on events. When we take $\gamma = \{x \times x \mapsto x | x \text{ is an event}\}$, we make x globally visible, even after communication with events x in other processes. In CSP, this particular γ function is even implicit in the theory's semantics.

A common pattern with theories like CCS is that an event is accepted, and *because of* that, other events are sent out — clearly an imperative structure. When such imperative concepts are introduced in a language, it is usually also necessary to introduce choices, loops and other programming constructs. As stated before, we find that problematic.

Another problem with pairwise communication is that it is hard to plug new components into a system. A new component would need to receive events to be able to act upon them with the additional behaviour offered in the new component. A pairwise communication structure makes this particularly hard: Before adding the new component, there must already be an event-sender that sends the information needed by the new component. This means that a possible extension component must be foreseen in the to-be-extended system, often not what you want with a component-based system. The Observer design pattern [GHJV86] is a popular[1] way to overcome the problem of pairwise communication, but the problem of predicting which events are possibly of use to new components remains.

Although Paul used CSP as a basis, it describes communication between a client and server object, thus constraining the global communication possibilities of CSP to pairwise communication. We learnt that a model with global, rather than pairwise communication was more suited to our needs, and that a 'bus' architecture was a possible way to achieve this goal.

Refinement. We have a preference for models that do not contain procedural concepts. Especially the model of methods invoking other methods seems to lead to a full-blown programming language. The expressive power of such modelling constructs can be replaced with another concept with better connectivity to mathematical theory, namely *refinement*.

The main reason why models express method invocations is to make relationships between behaviour visible; either because parties cooperate at the same level, or because they have a part-of relationship. The first aspect can be captured with the aforementioned communication schemes found in process algebras, the second aspect can be captured well using refinement.

The same design can be modelled at different levels of refinement, such that the levels of refinement are related by a partial order. A suitable

[1]Its inclusion in the root class of Java [AGH00] clearly shows its popularity.

refinement relation can be used to verify that the behaviour specified at different, ordered levels have a particular (refining) relationship.

We have worked on extending Paul with refinement [Rei97b] [Esh98], and have learnt that refinement is a powerful help in design. For this reason, we want any process model following up Paul to support refinement, or at least be prepared for it.

Parameter types. We worked on extending our Paul systems with parameters to methods, but decided to stay away from general types such as floats, strings and records. Instead, we kept (identities of) processes as the only possible type in Paul. It appeared to us that this sufficed to express any data type. Not just because Peano arithmetic proves that more complex data types can theoretically be folded into this form, but also because many interesting cases in practical applications mainly, or only, require these identities. Furthermore, it appeared to us that most interesting cases for process modelling could be *easily* expressed with just these process references as parameter values; in this, we were backed by our own experiments and case studies.

We learnt that the use of just process references was a justifiable simplification of our work; it did not seem to be an oversimplification (rendering the selected problem domain unpractical) and at the same time it seemed to make the problem domain crackable.

Non-determinism. In general, process algebras [BW90] have been an important source of inspiration to our work. Specifically, the work of Basten et al [BA96b] [AB97] has been a great source of inspiration.

Process algebras often use bisimulation semantics, which is a structural equivalence. We preferred the (weaker) observational equivalence for the purposes of Paul, and failure equivalence specifically provided a notion of non-deterministic choice that was usable to denote a choice made by another process. This is one reason for our choice of CSP.

In principle, the use of non-determinism to denote a choice made by ‘the other’ process worked well to distinguish client and server processes in a communication. However, the disadvantage of how this was done in Paul is that there can only be two parties in a communication. If knowledge of a server choice is represented as non-determinism in its client, and that drives client-choices over another association to the same server, then the server sees these derived client-choices as though they were not his, possibly causing rejection of a model in Paul.

Clearly, some information was lost. Our conclusion from this experience was that non-determinism may be a powerful tool, but the knowledge that a choice is non-deterministic should not be passed around from process to process. The only other way to get knowledge about non-determinism globally available therefore seems to be that it is global, directly accessible by any other process.

CSP semantics. Two widely known and researched process theories with failure semantics are CSP and CCS. These language are quite close [Gla86], with a main difference being the representation of non-determinism. CSP uses different choice operators to distinguish 'internal choice' and 'external choice', and CCS uses an internal step τ to model a transition caused by an uncontrollable factor[2]. Although both could formally express Paul's failure-based processes, the model of CSP is closer to Paul and has therefore been used.

For Paul, failure semantics were fine, but the relationship between state diagrams and Paul's failure semantics relied on the non-occurrence of two non-deterministic choices in immediate succession. A state in between two such choices provides information about the choice that was made, but that information is lost when the trace leading to that state is all that is known. This demonstrates that states are not observable from a failure semantics, and it may not be what we want for a practical process concept, where state diagrams are likely to be the means of communication.

In conclusion, if a process model is based on state charts and if it allows non-determinism arbitrarily, then we want it to make the current state of a process observable. This means we need a semantics which is a bit more specific than failures.

Model checking. Model checkers (of which Paul is an example) typically address systems with bounded numbers of instances of processes. In the case of Paul, we considered classes with one-to-one relationships, which avoided the typical problems with instances. As soon as other association

[2]One possible factor is radio-active decay of isotopes, which results in dualistic situations under the Copenhagen interpretation of quantum mechanics — the meaning of τ here models the situation of Schrödinger's cat: It cannot be determined whether or not a τ step has taken place until the next observation.

This remark should *not* be interpreted as a defence of the Copenhagen interpretation of Quantum mechanics — note that we do prefer the alternative theory of processes!

cardinalities are allowed, instances and instantiation must be taken into account.

There are extensions to model checking that support more elaborate systems, including those with multiple, sometimes even unbounded numbers, of instances of a process, but these usually either require additional manual work from the designer [KM89] or they would be so conservative that most reasonable specifications would be disapproved of.

Our approach with Paul was based on the realisation that Paul specifications contain quite a lot of protocols, which also requires manual work. Furthermore, we realised that more dynamicity would result in fewer accepted designs, perhaps even too small to make the tool usable. Because of these problems, we decided to consider theorem proving (the next step up from model checking) as a technique for process checking in our later attempts.

Implementation Language. Paul was implemented in C, with parts in Lex and Yacc. These languages were not a deliberate choice, but just because Paul grew from a syntax checker constructed with Yacc. When we started adding tests and features to the checker, we noticed that the code was increasingly getting harder to maintain. One reason was the data structures and manipulation functions for them are quite primitive when programmed in C.

The advantage of the strong touch-base between C and hardware, which is quite desirable for systems programming, was a disadvantage for Paul. Among other things, there were many concerns at the systems level while debugging, and avoiding these would have been quite beneficial for the prototype system that Paul is. We learnt that C is not at all a suitable language for verifiers like Paul, and that we were much better with higher level languages, including scripting languages and functional languages to do the manipulations and verifications on our processes, and a high-level syntax such as XML (where the parser need not be manually crafted) was a better choice as an input/output language than a self-crafted language.

2.8 Related work

Protocols. Our work on protocols in object designs is related to work on protocols in the networking sense in Lotos [EVD89]. The work on Lotos however, does not apply directly since it was designed from a networking perspective, rather than an object-oriented perspective.

Other related work on protocol checking is the development of Spin [Hol97], a versatile model checker. The philosophy behind Spin is very much like that behind Paul, generating counter-examples. If the tool Paul would have been designed with care, it might have been a syntactical wrapper around Spin. Notwithstanding that, the lessons learnt from doing it ourselves have been quite valuable for our continued research.

Many other model checkers deal with CTL [Hoa85], the Computation Tree Logic for branching-time models of execution. Although CTL itself ignores method names on transitions, there are extensions such as ACTL [DNV90] [DNFGR93] that extend CTL in that direction. Checkers for ACTL could also have been used to drive Paul.

Objects and roles. Roles are end-points of associations, whose intent is to describe objects from a specific perspective. Several authors have attacked, and sometimes attempted to solve, the lack of semantics related with associations in major object methods: [GBHS97] [Tan95] [McC97] [LH89]. Other authors adopted roles as a solution to this problem, and have worked specifically on this area: [KØ96] [HO93] [LM95].

Objects and processes. Work which is similar in spirit to our own has been conducted by Nierstrasz [Nie93] [Nie95]. This work focuses on substitutability of protocol aspects of objects, while our work focuses on protocol aspects over associations and internally in a single class; our work further distinguishes itself by contributing a model for roles, for feedback from server to client, and by the availability of an implementation.

Many object methods (such as Catalysis [DW98] and UML [Cor97]) describe object life cycles in terms of state charts, or a flavour of Harel's state charts [Har87]. Therefore, we have striven for a strong relation with that 'formalism,' but with some second thoughts [Rei97a] about them.

The work on the object-oriented method KISS [Kri94] contains an interesting process notion *between* objects, which resembles our notion of the communicated process over an association. It does not distinguish client and server protocols.

The ROOM method [SGW94] for real-time object design supports an idea similar to our roles, each having a state diagram protocol description. Its angle is that of real-time system design.

We expect that class protocols can also be formulated in terms of

PSL [LM95] instead of CSP. In fact, we consider any construct that limits the possible orders of execution of implementation fragments as a candidate formalism for class protocols.

2.9 Chapter summary

The following list of conclusions has functioned as *design principles* in constructing another process notation, namely the life cycles introduced in the following chapters.

1. We have seen useful applications of the verification of programs, but find verification of earlier stage models more useful.
2. Tool support is desirable for verification.
3. Feedback in the form of counter-examples is desired from a verifier. Some degree of minimisation on these counter-examples is useful.
4. Try to generate automated code for process aspects like scheduling concurrency and transactions.
5. Avoid procedural aspects of the verified models, such as loops and choice, by removing causality between events in processes.
6. Prefer globally broadcasted events over point-to-point communication.
7. Support refinement.
8. Initially support process identities as types.
9. Support non-determinism, preferably even globally observable.
10. Process semantics must allow observation of states.
11. Do not constrain the verifier to model checking techniques only.
12. Program a verifier tool in a 'high level' language.

Chapter 3

Life cycle syntax

Miultiplicitas non ponenda sine necessitate.
(Never use many when one is enough.)

William of Ockham

As part of our life cycle research, we evaluated object-oriented modelling concepts as well as concepts from process algebra. Integrating these is not straightforward, as some elements of these areas are hard to combine. In this chapter we therefore introduce our own graphical notation of *life cycles* because having our own notation makes it better possible to effect such an integration. In the evaluation chapter, we will set this out against object-oriented and process-algebraic techniques.

People that use a modelling language to draw up a design enjoy their modelling language most when they have powerful modelling constructs, when the same thing need not be expressed twice or with overlap, and when the model's structure can adhere closely to the world it represents. These desires are common to makers of object-oriented models (or 'designers') and makers of mathematical models (or 'formalists').

There are also differences between the preferences of these classes of model makers. One difference is syntactical: Object designers usually express their opinions in diagrams (which are easier to work with) while formalists prefer formulæ (which are usually precise).

Life cycles provide a process notation that looks like those commonly used in object design, and at the same time they do have the precision of mathematical models. This provides the benefits of both worlds: Although the core syntax of the models is minimal, some syntactic sugar can provide the expressive notation power that is liked about object-oriented models.

The integration of object-oriented and mathematical modelling languages is not straightforward. Existing object-oriented models can be so imprecise, that attempts to assign semantics lead to a variety of possible interpretations of the standardised syntax [Bee94], and the model languages are so complicated that formalisations usually cover only part of the syntax [LMM99]. Similarly, many formal models and tools [Hol97] [RF99] are quite useful, but confined to a somewhat too limited class of specifications. Such limitations are caused by a limit of what can be proven: It is our intention to broaden this to make process verification more usable in everyday object modelling practice. Our central goal for broadening is to allow automated proofs over systems with unbounded numbers of (life cycle) instances.

For the aforementioned reasons, we choose to define our own process language. Life cycles are intended as a replacement of *state charts* as originally defined by Harel [Har87] and used with different semantics in UML [Cor97]. A few object methods have more or less rigorous semantics, and nevertheless manage to be practically feasible [SGW94] [Kri94] [DW98]. Our work takes a similar approach, distinguishing itself by aiming for verification of more general object-oriented systems.

3.1 Tutorial introduction

This section introduces our concept of life cycles in the style of a tutorial. A series of problems is presented, each introduced with an example, to guide the definition of the graphical syntax for life cycles. Where textual syntax is given, BNF syntax representation is used. The semantics follow later, in Chapter 4, but an informal impression will be given here.

The running example used in this tutorial introduction is that of a simple library, inspired on the course materials for Catalysis [DW98]. A class diagram for this system, in which members, books and reservations play a role, can be found in Figure 3.1. The life cycle notation will be developed while addressing different modelling issues of this example.

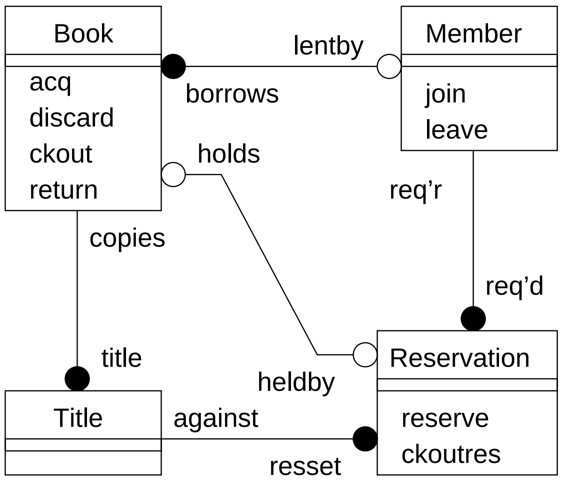


Figure 3.1: Class diagram for a simple library.

3.1.1 Undirected events

Problem. There are multiple ways to relate events in state diagrams to 'the real world' in which we humans live and experience events. State charts by Harel [Har87] and in UML [Cor97] usually have a notion of events dropping into the computer from outside the state charts' domain. To provide state charts with the ability of initiating further actions, they can also send out events. This approach is also taken in ROOM [SGW94] and in our earlier protocol checking work [RF99].

This approach is not free of problems. It is often unclear [LMM99] [Bee94] what the semantics are if a multitude of events arrives at the same time, and whether a nesting of events is allowed. Judging from the debates on these semantical issues, it seems that no reasonable choice covers all cases of practical interest.

Discussion. As stated in the principle of procedural aspects (item 5 in section 2.9), we believe that this problem of a suitable semantics is caused by the over-specification caused by the procedural idea of *sending* an event. We therefore prefer to avoid a causal relationship between events, much like the approach taken in process algebras [BW90] [Hoa85]. These theories demonstrate that it is possible to be accurate without causal relations between events, and also that leaving such relations out simplifies semantics and gives better opportunities for automated verification of the specified processes.

Solution. Our choice is to avoid expressing the direction of event communication. Instead, we define *formal events* in our life cycles, which are the formal descriptions of events that relate to *actual events* that are the most concrete form of event; an actual event describes an occurrence of a real world event, and the relationship between formal and actual events may be compared to formal and actual parameters. As a result, every formal event responds to a certain class of actual events. More on that follows in the coming subsections.

A formal event is defined as a name plus a list of parameters (the latter of which are introduced in Subsection 3.1.4). An actual event may be the simultaneous occurrence of multiple things; this leads to the definition of an actual event as a set of events.

$$\begin{array}{lcl}
\mathit{formalEvent} & ::= & \mathit{eventName}\ (\ \mathit{formalParams}\ ) \\
\mathit{eventName} & ::= & \mathit{name} \\
\mathit{formalParams} & ::= & \mathit{formalParam}\ ,\ \mathit{formalParams} \mid \mathit{formalParam} \\
\mathit{actualEvent} & ::= & \{\ \mathit{actEvents}\ \} \\
\mathit{actEvents} & ::= & \mathit{actEvent}\ ,\ \mathit{actEvents} \\
 & \mid & \mathit{eventName}\ (\ \mathit{actualParams}\ ) \\
\mathit{actualParams} & ::= & \mathit{actualParams}\ ,\ \mathit{actualParams} \mid \mathit{actualParam}
\end{array}$$

Actual events may not contain the same event name more than once.

Based on these definitions, we define a life cycle as a structure containing a life cycle name, a set of states (which we define as names that are assumed to be unique within a life cycle), and a set of transitions. The types *name* and *state* remain unspecified, and the definition of *transition* is deferred to Subsection 3.1.6.

Example. Figure 3.2 defines two life cycles. An example of an actual event in this system is $\{\mathsf{ack}(\ldots)\}$, which matches only a formal event in Book and is ignored by all instances of Member. In contrast, the actual event $\{\mathsf{ckout}(\ldots)\}$ is matched by a formal event in instances of both the Member and Book life cycles, and it therefore represents a joint step in which books and members co-operate. It would also be correct (but not meaningful in this application) to consider $\{\mathsf{acq}(\ldots),\mathsf{leave}(\ldots)\}$ as an actual event, matched by the formal event acq(. . .) in the Book life cycle and by the formal event leave(. . .) in Member. This is another example of two life cycles that jointly make a transition in response to the same actual event. These joint transitions are similar to the state charts concept of events triggering new events, but the semantics of the life cycle form of co-operation are simpler.

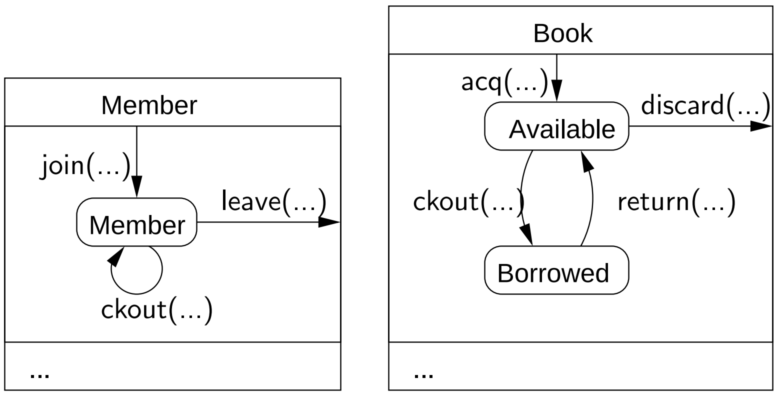


Figure 3.2: Library in state diagram notation.

3.1.2 Dynamic communication structures

Problem. The intention of modular, or *component-based* development is to allow independent development and compilation of components, or subsystems. To support modularity, it is necessary to adapt communication patterns between components at the time they are put together, meaning after they were compiled.

One thing that occurs when implementing such components is the strict pairing of caller to callee in procedural-styled languages, including object languages. The problem with this pairing is that it is hard to add or remove one party in such communication when the other party is to be left untouched.

Discussion. As explained in the principle of broadcasting (item 6 in section 2.9), a scheme of communication that we prefer is one where *any number* of parties may participate in an event. That thought also underpins the observer design pattern [GHJV86], which is mainstream in several object libraries and/or languages in practical use. A global communication scheme is possible, but only when the rules for synchronisation are accurately defined. We follow process algebras by disallowing events that are contained in participant's alphabets, but for which one or more participants are not in the proper state. For example, when the Book life cycle in Figure 3.2 matches a formal event return(...), it blocks (or disallows) the return operation if it is in state Available. We choose to ignore events that are not named in a life cycle; for example, any event named return is ignored by the Member life cycle. This choice makes it more practical to use components; if actual events occur that were not taken into account during a component's design, then these actual events can be ignored.

We find it important to block actual events that are too early or too late, because this allows *synchronisation* between life cycles. Similar ideas can also be found in LOTOS [EVD89] as ||, and in ACP [BW90, §4.2], CSP [Hoa85] and CCS [Mil80] as |. Collaborative processing of events is known in some programming languages as *rendez-vous synchronisation*. It is found in the languages Ada [ISO94] and in Occam[1] [Ltd88]. Finally, this concept of joint transitions in multiple instances is closely related to the concept of *distributed transactions* [X/O96].

Solution. As in process algebra, we define an alphabet on a life cycle diagram, containing all the formal events it recognises. Formal events may be related to actual events in three possible ways, namely:

- matching means that an actual event causes a state transition in the matched life cycle instances;
- ignoring means that an actual event is currently not considered interesting; the ignoring formal event does not respond to the actual event;
- blocking means that an actual event cannot be ignored, but there also is no transition that matches it.

Different life cycles need not be aware of each other, because they independently observe the same actual events occurring. This greatly advances modular design of a life cycle system. Furthermore, even instances of life cycles need not be aware of each other, for the same reason. This is not only helpful for composition, but also for distribution.

[1]Occam is a registered trademark of INMOS Limited.

Example. The actual event {acq(. . .)} is matched by a formal event acq (. . .) and it is ignored by a formal event ckout(. . .). Some examples of the dynamicity of this communication structure is presented in Subsections 3.1.5 and 3.1.6, where more syntax will have been introduced.

3.1.3 Creation and deletion of instances

Problem. To be practical, a life cycle system must deal with any number of *life cycle instances*, which we also call *objects*: Some Books in our library will be Borrowed, while others are Available. Using the name *object* for life cycle instances suggests a relationship to class models, but it is only that — a suggestion.

Every object has a current state. This information is of influence on the decision which of the formal events influence an object's behaviour in terms of matching, ignoring or blocking an actual event; the combination of these influences from selected formal events is used to determine the reaction of an object to an actual event, which may in turn be either matching, ignoring or blocking it. Details follow later.

Every object is an instance of a life cycle, and it can assume that actual events take place in the order specified in that life cycle. This is helpful to resolve the problem of polymorphic object creation.

Generic (or polymorphic) interfaces to objects are a popular means of hiding their implementation, but when creating an object, such a generic interface does not provide sufficient information. Because of this, the implementation of an object must usually be provided when creating a new instance, meaning a loss of genericity in the program. This problem has a solution in the factory design pattern [GHJV86], but the additional process knowledge in our life cycles makes it straightforward to support generic creation without a need for such an intermediate.

Discussion. Because of the principle of procedural aspects (item 5 in section 2.9), we prefer to describe creation and deletion of objects without explicit new operations that must be invoked from somewhere. One reason is that this would re-introduce procedural style, another reason is that it treats creation as something special, and not polymorphically. We prefer creation and deletion to be a response of a life cycle system to a plain event occurrence.

Such approaches can be found in process algebras. For example, concurrent processes in ACP [BW90, table 35] are silently removed when they finish. Another process algebra, the π-calculus [Mil91], introduces a similarly elegant construct for process creation: A process P may be prefixed with a *replication operator* !, with the interesting property that $!P = P||!P$, or in words, $!P$ is any number of P processes interleaved. This behaves as an infinite pool of instances of process P ready to be created upon communication with a P process. With such a pool, there are always enough processes ready to start after observing a particular event; this even enables polymorphism, as the implementation of the created process need not be mentioned as part of the observed event.

Solution. We define an *object* or *life cycle instance* as a quadruple consisting of a *lifecycle*, the instance's identity (unique per life cycle), a current *state* which may take a value allowed by the life cycle, and a binding relation that maps all currently known variables to an instance's identity, as will be explained in Subsection 3.1.4:

$$\begin{array}{lcl} \mathit{object} & ::= & \langle \mathit{lifecycle}, \mathit{id}, \mathit{state}, \mathit{binding} \rangle \\ \mathsf{Nirvana} & : & \mathit{state} \end{array}$$

The type *id* is not specified here; it can be any type on which an equivalence relation is defined, including numbers, hieroglyphs and hexadecimal memory addresses.

The special *state* Nirvana is used to represent objects that are not currently considered instantiated. Objects in Nirvana state are willing to communicate in the transitions that depart from the borders of the rectangle enclosing its states. Such transitions are called the *creating transitions* of a life cycle. The opposite, the transitions going to the rectangular border around the states, are called the *terminating transitions* of a life cycle. When an object is not in a state drawn inside the life cycle rectangle, it is in Nirvana state.

Example. A practical implementation need not (and usually will not) consume any form of storage for objects in Nirvana state, but the semantics do make this assumption. This makes creation and deletion conceptually simple, objects simply exist forever.

It is similarly simple to specify redundant properties with this concept. The life cycle in Figure 3.3 redundantly expresses the constraint that every ckout of a Book must be followed with the corresponding return. There is no need to explicitly create an object for this redundant property; it simply creates itself when the proper actual event is observed.

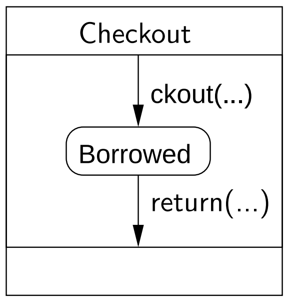


Figure 3.3: The checkout life cycle.

3.1.4 Parameters and variables

Problem. As explained before, actual events are observed by all life cycle instances. It is necessary to distinguish the objects that should, and should not, respond to an actual event, to avoid such things as a state transition from Available to Borrowed state by all Book objects in response to the ckout of a single Book.

Discussion. Objects must be distinguishable, and giving each a unique identity is the customary way of doing this. An object's identity is symbolically represented as this in life cycle diagrams, just like in many object-oriented languages. The value of this is defined and unique for every object from the moment the system is initiated. The value of this always remains the same for all objects, even while they are in the Nirvana state.

Life cycles respond to each other as well as to their own identity, so actual events are extended with parameters that each hold an identify of a life cycle instance. Formal events can read or match values of instances with variables stored in the instance.

Solution. Every object defines a set of variables, each of which holds a reference to an object, conforming to the principle of types (item 8 in section 2.9). Actual events have parameters which are identities of objects, and formal events have parameters which are variables that are either read or written.

At every moment in an instance's existence, there is a mapping from the variable names or formal parameters to the instance identities or actual parameters. Although the naming of directions is mainly suggestive, it is merely an interpretation of the way that matching, blocking and ignoring work on actual events.

- Input, denoted with ? graphically, for variables whose value appears to be read from the actual event. Formally, if a formal parameter is an input parameter, it matches every possible value in an actual event. Furthermore, the value matched will be available in the named variable after the event has taken place. This new value for a variable v is denoted with ?v.

- Output, graphically denoted by absence of an annotation. Formally, if a formal parameter is an output parameter, it matches only the value currently stored in the named variable, and ignores all other values at that position.

- Force, graphically represented by a ! before the variable name. Formally, if a formal parameter is a force parameter, it matches only the value currently stored in the named variable, and blocks all other variables at that position. An example of a force parameter is presented in Subsection 3.1.5.

The syntax of formal parameters *param*, as used in formal events on a transition in a life cycle diagram, and the syntax of a variable declaration *vardecl*, as used in the bottom box of a life cycle diagram, are defined as follows:

$$\begin{array}{lcl} \mathit{formalParam} & ::= & \mathit{name} \mid ?\mathit{name} \mid !\mathit{name} \\ \mathit{actualParam} & ::= & \mathit{id} \\ \mathit{vardecl} & ::= & \mathit{name} : \mathit{name}\ \texttt{@}\ \mathit{states} \\ \mathit{states} & ::= & \mathit{state}\ ,\ \mathit{states} \mid \end{array}$$

Where *id* is, as before, any type with an equivalence relation defined on it.

Variables are local to a life cycle instance, and their names are unique per life cycle. Every life cycle defines a variable named this which is predefined and must not be used as an input parameter, not even on creating transitions. The variables (except this) that are used in a life cycle must all be declared in the bottom box; not only the type, but also the states in which the variable exists must be mentioned. The state Nirvana must not occur in those declarations.

The variable this exists in every state, including Nirvana. Every creating transition must mention this as an output parameter. All other variables can only exist in a state if for all transitions leading to it, either the variable is an input parameter, or the variable existed in the state from which that transition originated. A transition may only use variables for output or force parameters if they exist in the state from which the transition originates. This set of constraints allow for static dataflow analysis.

Example. The library system in Figure 3.2 can now be completed with parameters on the events, leading to Figure 3.4. This new design is more realistic than the previous version; an actual event occurrence ckout(CamelBook,Tarzan) causes a transition in Book CamelBook and Member Tarzan, so that only a single Book instance and a single Member instance respond to the event occurrence. Other instances are not altered. If a Book Bible happens to be in the Borrowed state, then this event occurrence is not blocked.

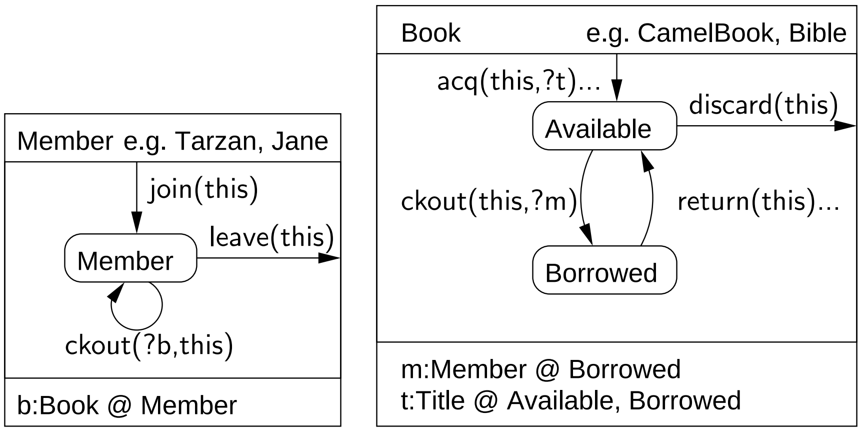


Figure 3.4: Two library life cycles with parameters.

Intermezzo: Dynamic alphabets. The combination of alphabets with variables is not trivial. When a variable's value changes, the set of actual events matched by a life cycle changes. This also means that the events that might get blocked (because they are only matched in states other than the current state) change. The alphabet is the description of what is blocked if not currently acceptable, so that should change as well.

To reflect this, we define life cycle alphabets as a set of formal events rather than the customary choice of actual events. This means that blocking, ignoring or matching of an actual event depends on the binding of variable values.

Except for this, a variable is only defined in some states, as declared in the bottom box of a life cycle diagram. For instance, in Figure 3.4, the variable m in Book is only defined in state Borrowed. This means that no formal event depending on the value of m is in the alphabet for instances in state Available or Nirvana. In this diagram, there is no such event (ckout does not depend on m, so it is in the alphabet for state Available, as may be reasonably expected). But a transition relax(m) from Borrowed to Borrowed would not show up in the alphabet in state Available, and would therefore not be blocked by an instance in that state. Simply because it cannot be determined to what member the blocking would apply.

This means that the alphabet of a life cycle depends on the variables, as well as on the current state of a life cycle. In other words, life cycles employ a dynamic alphabet.

3.1.5 Enforcing synchronisation

Problem. Imagine extending the library system presented in Figure 3.4 with reservations. For simplicity's sake, we shall first reserve Books rather than Titles. When a book is returned for which reservations exist, it must be put on hold for one of the reservations. The common approach to this in other process models (such as state charts) is the use of a boolean guard that splits the arrow for the return transition. This has the disadvantage that the Book life cycle must contain knowledge about reservations, which makes it impossible to extend the library system in Figure 3.4 with reservations by just adding new life cycles.

Discussion. To enforce an identity of a parameter, such as the member that is allowed to checkout a book, it is necessary to block all actual checkout events with a different member parameter. But, neither input nor output parameters are capable of expressing blocks based on the identity of a parameter. An input parameter matches everything, and an output parameter ignores actual events with unmatched values. We therefore introduce a third kind of parameter, the *force parameter*, to enforce identity of that parameter.

Solution. In graphical notation, a parameter that enforces the identity found in variable x is written as !x; these accept the current value of the x parameter as an x parameter would and it blocks all other parameter values at that place in an actual event. If an actual event e matches an event on name, parameter count and all output parameters, so that e is not ignored, then there still is the question whether e should be matched or blocked; blocking is what happens when a force parameter has a value that differs from the corresponding value in e.

Example. Figure 3.5 defines the Reservation life cycle. An instance of this life cycle is created when a reserve(?b,?m) event occurs, and that instance subsequently awaits the return of Book b. When that book is returned, an instance of the Reservation life cycle claims it, and adds a constraint on the next ckout of b, stating that it may *only* be checked out by Member m who made the Reservation initially. This form of synchronisation, dynamic because it depends on the variables stored in an object, is useful and, as far as we have seen, it is not commonplace in graphical process languages. This dynamicity is needed for life cycles, to counter the strong synchronisation behaviour of blocking events if the current state is not the proper one — to compare, state charts in UML ignore an event if it belongs to another state. In this case, the dynamicity deals with the added constraint due to the Reservation; and it retains modularity, because the Book need not be aware of this additional constraint.

Note how the asymmetry between the book and the member in ckout corresponds to a conceptual asymmetry: Member m should be able to check out other books than b, but Book b may not be checked out by any other Member than m.

3.1.6 Multiple perspectives on events

Problem. In the previous solution, we introduced a Reservation on Books. It is better to reserve Titles than Books, to support the situation where two copies (two Books) of the same Title are owned by the library. If a

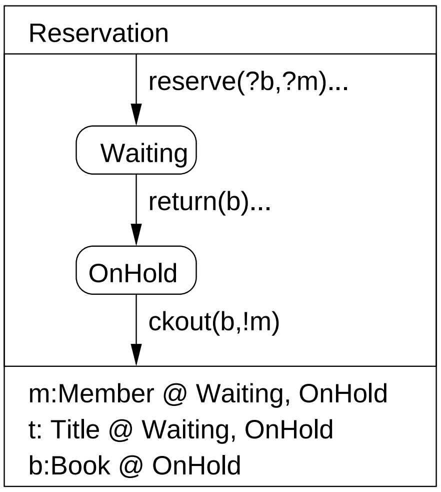


Figure 3.5: Enforcing a certain Member parameter value.

Reservation life cycle must wait for a Title, then that title must also occur in the parameter list of the events that make a book available, thus also on return. The Title has until now not been a parameter to the return event, and it would be a demonstration of bad extensibility of life cycles if we would need to add it now.

Discussion. This is a common problem; from one perspective, there should be certain parameters to an event occurrence, and from another perspective there should be other parameters. This is caused by different perspectives on the same event occurrence. From the perspective of a Book, only the Book returned is of interest. From the perspective of a Reservation, the Title is also of interest.

The support of multiple perspectives on an actual event is taken care of by making every actual event a *set* of events. Every set element can cover a different perspective. Some life cycles will be designed with more than one perspective in mind, and those must be able to express multiple formal events taking place on the same transition; to support this a transition on a life cycle can be annotated with a *logic combination of formal events* rather than just a single formal event.

Where state charts use nested states (a strictly hierarchical ordering of states), the approach with life cycles is to define more than one process, each from a different perspective, and synchronise using the same formal events to make the perspectives jointly respond to the same actual events. This synchronised behaviour is a more flexible scheme than a hierarchy of states, where other perspectives (with different hierarchies) usually reveal themselves as transitions that cross hierarchical levels; and indeed, such transitions are supported in state chart diagrams. We find that this breaks the nested structuring concept, and therefore believe that hierarchical states are undesirable for life cycles.

Solution. A transition in a life cycle is annotated with any number of events:

$$\begin{array}{rcl} \mathit{transit} & ::= & \mathit{formalEvent} \\ & \mid & \neg \mathit{transit} \\ & \mid & \mathit{transit} \wedge \mathit{transit} \\ & \mid & \mathit{transit} \vee \mathit{transit} \end{array}$$

So, a formal event can logically express which actual events will match, and which will not. Informally, a match occurs when the logic composition of separate tests of a formal event against an actual event matches.

Example. In the life cycle for a Book, it is a good design choice to introduce a formal event avail(this,t) along the transition that now holds the event return(this) only, resulting in the formal event return(this)∧avail(this,t). The avail event, which represents the concept of a book being made available for checkout, can be matched in the Reservation life cycle. The concept of a book being made available is only a fraction of what a return event does; that same fraction is also part of the acquisition of a new book, so an avail(this,t) event should be added to the creating transition of a Book. The resulting life cycles for Book and Reservation are therefore as given in Figure 3.6. The Title life cycle is an opaque type; its life cycle diagram is not needed for our presentation, only its name suffices.

3.1.7 Guards on transitions

Problem. There are situations in which a boolean condition is needed to make the decision whether or not a transition should match an actual event. As demonstrated in the previous subsection, some of these situations can be addressed in other ways, but not all — notably, constraints that demand

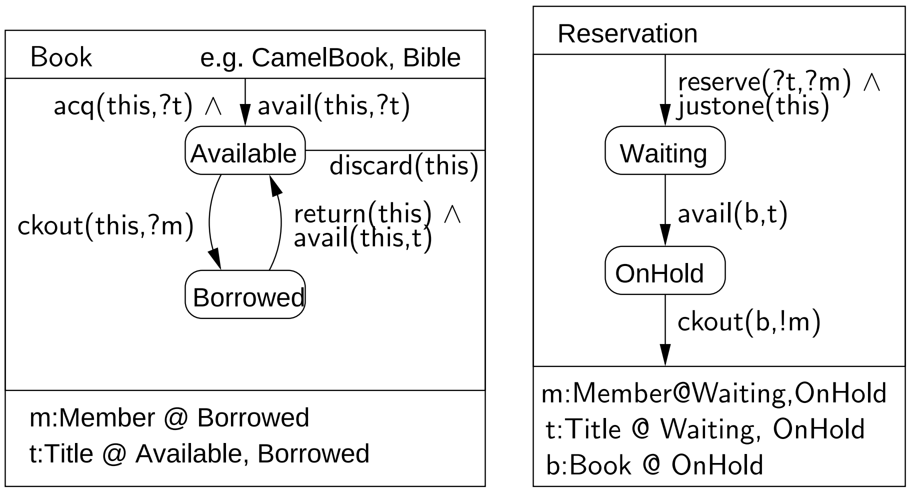


Figure 3.6: The use of multiple events per transition.

that an instance referenced by a variable x is in a state s cannot be expressed without explicit conditions.

Discussion. Transitions describe a condition on the actual event, and they are a reasonable place to add further constraints. UML does the same thing, so it would seem that practical systems can use such a facility.

Although there are alternative ways of formulating some of the guard conditions, it is still useful to also have this notation — as long as it is treated in an orthogonal way, meaning that the additional notation can be translated in full to an already-existing notation with the same meaning.

Solution. When the event expression *transit* on a transition is subject to an additional constraint, it will be formulated in terms of a *guard expression*, which is graphically distinguished in the life cycle notation by surrounding it with square brackets.

$$\begin{array}{lcl} \textit{guard} & ::= & \textit{name}@\textit{state} \\ & | & \textit{name} = \textit{name} \\ & | & \textit{name} \neq \textit{name} \\ & | & \neg \textit{guard} \end{array}$$

$$\begin{array}{ll} | & \mathit{guard} \wedge \mathit{guard} \\ | & \mathit{guard} \vee \mathit{guard} \end{array}$$

Example. These guards allow for extra possibilities, such as stating the effects of an undo of a reservation r with event stopres(r) in a Book life cycle. This operation has a result that depends on the state of r, so there are two transitions with different annotations:

stopres(r) $\wedge$ avail (...) [r@OnHold]
stopres(r) [r@Waiting]

This expresses that a stopres(r) must in some cases be combined with avail(...), namely when r is in the state OnHold. As a result, the availability notification can inform other interested parties that the reserved item is available for others.

3.2 Graphical syntax

The notation of a single life cycle diagram is a rectangle split into three 'stacked boxes'. The top box contains the names of the life cycle and possibly example names of instances, the middle box contains the process diagram, and the bottom box contains declarations of variables:

$$\mathit{lifecycle} \quad ::= \quad \begin{array}{|c|} \hline \mathit{topbox} \\ \hline \mathit{process} \\ \hline \mathit{botbox} \\ \hline \end{array}$$

The syntax in the top box is as follows:

$$\begin{array}{lcl} \mathit{topbox} & ::= & \mathit{name} \\ & | & \mathit{name} \quad \textsf{e.g.}\ \mathit{instances} \\ \mathit{instances} & ::= & \mathit{name}\ ,\ \mathit{instances} \mid \mathit{name} \end{array}$$

The list of *name*s following e.g. is interpreted as a list of instance names. These are provided to make feedback optimally readable (instances names such as Tarzan and Jane are easier to follow than Person0 and Person1).

The variable declarations in the bottom box use the syntax defined in Subsection 3.1.4, as follows:

$$\mathit{botbox} \quad ::= \quad \mathit{vardecl}\ \mathit{botbox} \mid$$

The first *name* in a *vardecl* is a variable name, unequal to this, and the second *name* in the same *vardecl* is its type. The variable is declared to exist precisely in the states mentioned after @. All variables declared in a *vardecl*, plus this, may be used in the *process* box. All occurring *state* names must occur in the *process* box. The name Nirvana is assumed to not occur as a *state* name in life cycle diagrams.

The middle box of a life cycle contains the *process*. In this box, there may be any number of non-overlapping *stateoval*s with a *state* name in each:

$$\textit{stateoval} \quad ::= \quad \boxed{\textit{state}}$$

The box' sides and the ovals may be connected in any way by arrows, representing transitions. Alongside each transition, there is an *annotation*, which is a *transit* as defined in Subsection 3.1.6, optionally followed by a *guard* as defined in Subsection 3.1.7:

$$\textit{annotation} \quad ::= \quad \textit{transit} \mid \textit{transit}\ [\ \textit{guard}\ ]$$

Nothing more is allowed in life cycle diagrams. Multiple life cycle diagrams, with different names for different life cycles, can be combined to specify a (software) system at a given level of refinement. The same system may be described at different levels of refinement, provided that the levels have a partial order defined on them. The same life cycle name may occur at different levels of refinement.

3.3 Normalisation of life cycle syntax

The syntax of life cycles is designed to be practical, and for that reason it contains a few notation forms that are redundant. Because orthogonality and completeness of concepts have been important in the definitions of life cycle notation and semantics, these forms can be rewritten to a simpler form, effectively treating them as syntactical sugar. The normalisation given here preserves the intuitive semantics given in the previous sections of this chapter.

In short, a normalised life cycle diagram is free of $!x$ parameters, free of guards other than on variable states, and has formal events simplified to a standard form. Below, we describe a procedure to normalise a life cycle diagram.

Step 1: Remove 'force' parameters. Force parameters $!x$ can be removed by conjoining the containing formal event with itself; but in the first, replace $!x$ with x to require the parameter value to equal x, in the other replace $!x$ with $?x$ to ensure that all possible parameter values for x are in the alphabet. This describes the same behaviour. For example, $\mathsf{e(!x,y)} = \mathsf{e(x,y)} \wedge \mathsf{e(?x,y)}$.

After this step, the definition of *formalParam* in Subsection 3.1.4 has been reduced to

$$\mathit{formalParam} \quad ::= \quad \mathit{name} \mid ?\mathit{name}$$

Step 2: Simplify guards. First, we simplify guard expressions by applying the following rules to each guard until no left side form matches anymore:

$$\begin{array}{lcl}
\neg\neg p & \rightarrow & p \\
\neg(x = y) & \rightarrow & x \neq y \\
\neg(x \neq y) & \rightarrow & x = y \\
\neg(p \wedge q) & \rightarrow & (\neg p) \vee (\neg q) \\
\neg(p \vee q) & \rightarrow & (\neg p) \wedge (\neg q) \\
(p \vee q) \wedge r & \rightarrow & (p \wedge r) \vee (q \wedge r) \\
p \wedge (q \vee r) & \rightarrow & (p \wedge q) \vee (p \wedge r)
\end{array}$$

The result of these standard rules is that all $\vee$ operators are at the outermost level, and all $\wedge$ operators fall inside it, and all negations have disappeared, except for ones directly prefixing $x\texttt{@}s$ primitive expressions. Even the negations in the form $\neg x\texttt{@}s$ could be removed, because an instance not in s must be in one of the known other states in which x can be; however, this could cause a dependcy on other life cycles, which we prefer to avoid here.

Then, for all transitions that have a form $p \vee q$ in their guards, split the transitions — that is, clone it with the same originating and destination states and *annotation*s — but adapt the guard by replacing the form $p \vee q$ with p in one and with q in the other. Because the destination states of the two transitions are the same, there is no newly introduced non-determinism; furthermore, the two transitions together catch no more or less actual events than the original transition did, so the intuition of the notation is preserved by this rewrite. Repeat this procedure until the $\vee$ operator is removed from all guards.

After this step, the syntax of *guard* in Subsection 3.1.7 has been reduced to:

$$\begin{array}{lcl} \mathit{guard} & ::= & \mathit{name}\texttt{@}\mathit{state} \\ & | & \neg\mathit{name}\texttt{@}\mathit{state} \\ & | & \mathit{name} = \mathit{name} \\ & | & \mathit{name} \neq \mathit{name} \\ & | & \mathit{guard} \wedge \mathit{guard} \end{array}$$

Step 3: Remove equality guards. Guards are now a ∧-separated list of constraints on the matching of a transition. From this list, some constraints can be rewritten to *transit* expressions on this transition, some others have to move to another transition.

Imagine rewriting e(x) [x=y] to e(x)∧e(y). The latter form matches the desired actual events and nothing more, but it also introduces e(y) in the alphabet of the life cycle, which can cause unintended blocks of actual events. This problem does not occur when e(?x) [?x=y] is rewritten to e(?x)∧e(y) because the existence of e(?x) in the alphabet already ensures that no actual events e(...) are ever ignored; adding e(y) to the alphabet will not change that.

To ensure that the alphabet of a life cycle remains unchanged when rewriting it, we must therefore move constraints like $x{=}y$ and $x{\neq}y$ in guards to the last previous state where either x or y was read, or where the previous state already implies the (in)equality. For the purposes of this normalisation step, assume that every state is related to knowledge about equal and unequal pairs of variables. Initially, this knowledge is not present, but due to splitting states in this step, more such knowledge will be added.

Moving a constraint to a previous state (say, s) can be achieved by duplicating the originating state of the transition to form s_1 and s_2 (as in Figure 3.7; the knowledge about one is extended to imply the equality (like $x{=}y$), the knowledge in the other is extended to imply the opposite ($x{\neq}y$). All transitions from and to this state are also duplicated, with a few changes. Transitions from s to itself also become transitions from s_1 to s_2 and vice versa in the duplication process. Transitions with state s_1 or s_2 as their destination get an extra constraint in their guards, namely the just-added knowledge for their destination state, but with variables that are read on the transition prefixed with a ?. Transitions departing from s_1

or s_2 are erased when their guards exclude the just-added knowledge for those states. When a guard in a state departing from s_1 or s_2 contains the just-added knowledge for those states, this constraint is removed from the guard.

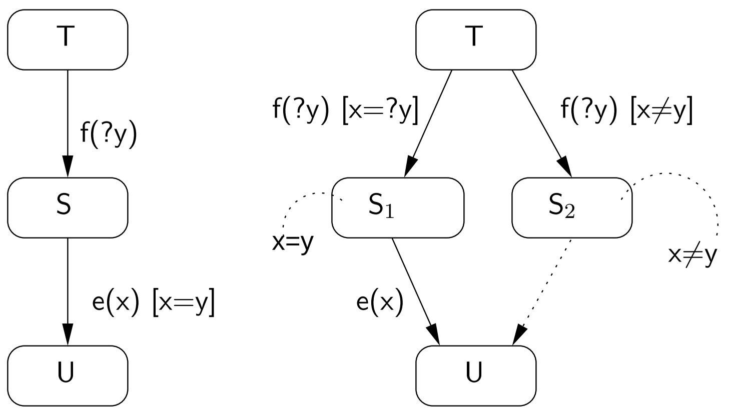


Figure 3.7: Example of splitting in normalisation step 3

When this procedure cannot be repeated anymore, all constraints have either been removed, or moved 'back' to the transition(s) where at least one of the mentioned variables serves as an input parameter. Removing the guards is now simply done by rewriting the events with the knowledge from each constraint in mind. We present the rules for a guard with one constraint; multiple constraints require these rules to be applied for each constraint iteratively, and constraints of the form $x@s$ cannot be removed with this procedure.

$$\begin{array}{lcl} E\ [?x = y] & \rightarrow & E \wedge E_{?x \leftarrow y} \\ E\ [?x \neq y] & \rightarrow & E \wedge \neg E_{?x \leftarrow y} \\ \neg P\ [g] & \rightarrow & \neg(P\ [g]) \\ P \wedge Q\ [g] & \rightarrow & (P\ [g]) \wedge (Q\ [g]) \\ P \vee Q\ [g] & \rightarrow & (P\ [g]) \vee (Q\ [g]) \end{array}$$

Here, E denotes a single formal event, P and Q denote formal event logic expressions, and g denotes a guard condition; a question-marked variable is assumed to be the first (since $=$ and $\neq$ are symmetric, that can always arranged) and $?x \leftarrow y$ replaces syntactical occurrences of $?x$ with y; the [_] symbol was loosely placed at intermediate positions in the expression.

After this step, the syntax of guards from the previous step is further reduced to the following form:

$$\begin{array}{lcl} guard & ::= & name\texttt{@}state \\ & | & \neg name\texttt{@}state \\ & | & guard \wedge guard \end{array}$$

Step 4: Simplifying transition annotations. The *transit* expressions in a transition's *annotation* can still have an arbitrary logic form, and this can be simplified, by applying the following rules from left to right as often as possible:

$$\begin{array}{lcl} \neg\neg P & \rightarrow & P \\ \neg(P \wedge Q) & \rightarrow & (\neg P) \vee (\neg Q) \\ \neg(P \vee Q) & \rightarrow & (\neg P) \wedge (\neg Q) \\ (P \vee Q) \wedge R & \rightarrow & (P \wedge R) \vee (Q \wedge R) \\ P \wedge (Q \vee R) & \rightarrow & (P \wedge Q) \vee (P \wedge R) \end{array}$$

Here, P, Q and R represent expressions in the formal event logic.

After these reductions have been exhaustively applied, all $P \vee Q$ forms are on the 'outside' of the formal event expressions, making it possible to split transitions. This is done, where $P \vee Q$ is replaced by P on the first clone, and with Q on the other. This is also repeated until not possible anymore; then, the formal event expressions contain no $\vee$ operators, and the are of the form $E \wedge F \wedge \ldots \wedge \neg G \wedge \neg H \wedge \ldots$, which will aid in translation to the semantics for life cycles.

After this step, the syntax of *transit* from Subsection 3.1.6 has been reduced to the following form:

$$\begin{array}{lcl} transit & ::= & formalEvent \\ & | & \neg formalEvent \\ & | & transit \wedge transit \end{array}$$

Upper bound and optimality. The number of states in the new diagram is always bounded to a maximum, namely the number of states multiplied by the number of possible partitions of variable values. A similar maximum applies to the number of transitions after normalisation. It would have been easier to devise a rewriting strategy that directly constructs such a 'maximal' diagram from the original form, but this would have created a diagram which always has the maximum number of states. The rewriting procedure sketched above is more complicated, but it only splits states when it must, keeping the number of states to reason about to a minimum, so it is to be expected that proofs can be conducted faster.

Feedback preservation. The feedback proposed in Figure 1.3 in the introduction can still be generated after the normalisation described above. The states are split, but retain their name (and add partial knowledge about variable equalities, which may be useful to add to the feedback). The actual event sequences are still the same as before, because the feedback names actual events; the more complex formal events capturing them are of no influence on that. The added formal events for constraints always have the name of an already-existing formal event, so the set of event names remains the same on all transitions.

For example, a formal event hire(?person,!company) [?person$\neq$company.director] matches hire(Tarzan,RedHat), and so will the normalised form hire(?person,company)$\wedge$hire(?person,?company)$\wedge\neg$hire(company.director,company).

3.4 Life cycle meta model

After the normalisation of the previous section, the syntax of the life cycle notation is described by the class diagram in Figure 3.8. This is a meta-model, so instances of these classes represent a concept that shows up in one or more places in life cycle diagrams.

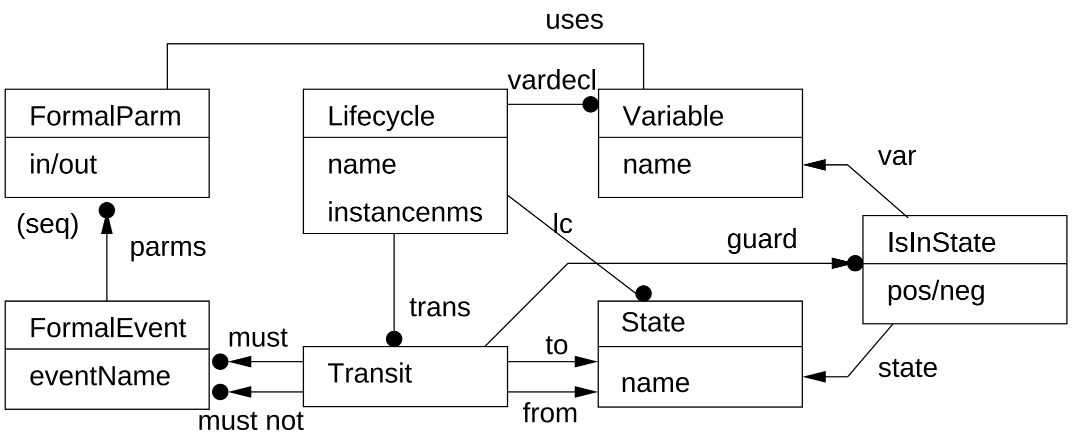


Figure 3.8: Meta-model of the normalised life cycle notation.

3.5 Chapter summary

This chapter introduced a variation on state diagrams called life cycles. We presented a graphical notation, and deferred the definition of semantics to the next chapter.

Our life cycles present some features that we did not find in comparable work: Life cycles support polymorphic creation and deletion of instances as a side-effect of normal steps. Life cycles systems are straightforwardly composed to larger systems. Life cycles can express dynamically changing synchronisation constraints. Life cycles allow redundancy in models.

Life cycles can express multiple perspectives on events; we consider this a good replacement for hierarchical state charts, whose precise meaning is not trivial.

Chapter 4

Life cycle semantics

The truth is usually just an excuse
for lack of imagination.

Elim Garak

This chapter defines the semantics of life cycles. To do so, it first introduces a temporal logic named Bill, and defines μ-calculus semantics for it. Then, the semantics of the graphical life cycle notation of the previous chapter is defined in terms of Bill. Finally, we demonstrate the expressiveness of our life cycles.

In what follows, we define a formal language called *Bill*, short for Behavioural Inference Logic for Life cycles. By adding syntactical sugar, this language can be extended to one called *William*, short for Weedy Inference Logic for Life cycles, Including Appealing Metaphrases. Then we introduce the *μ-calculus*, and we define Bill semantics in those terms. Finally, we use the Bill language to define the semantics of life cycles.

In our definitions, we will regularly postulate sets; some of these require no structure for the treatment here, in other cases the sets are initially postulated only because they can only be defined once a life cycle application to model has been chosen.

4.1 The language Bill

We postulate a number of lexemes for Bill to represent names in a Bill expression. We shall assume that for any concrete expression in Bill, no name occurs in more than one of the sets defined by the lexemes. The postulated lexemes are:

- A set of life cycle names *Lifecyclenm*; we let L, M vary over *Lifecyclenm*. This corresponds to the Lifecycle.name attribute of the meta-model in Figure 3.8.

- A set of state names *Statenm*; we let S, T vary over *Statenm*. This corresponds to the State.name attribute in Figure 3.8. The special element $\mathit{Nirvana} \in \mathit{Statenm}$ shall represent the 'not-created' state that a life cycle instance can be in.

- A set of event names *Eventnm*; we let a, b vary over *Eventnm*. This corresponds to the attribute FormalEvent.eventName in Figure 3.8.

- A set of Bill variables *BillVar*; we let x, y vary over *BillVar*. A Bill variable name x is bound by the Bill expression $\forall x : L \cdot \_$ and does not correspond to any element of Figure 3.8.

- A set of link names *Linknm*; we let l, m vary over *Linknm*. This corresponds to the attribute Variable.name in Figure 3.8. There is one life cycle variable about which more is known than about others, namely this.

- A set of loop names *Loopnm*; we let I, J, K vary over *Loopnm*. This does not correspond to any element of Figure 3.8.

In addition to these lexemes, we postulate a number of functions to represent relationships between some of the above. These relationships can be defined when a concrete life cycle application is turned into Bill expression form. The functions are:

$$\begin{aligned} \mathit{linktp} &: \mathit{Lifecyclenm} \times \mathit{Linknm} \rightarrow \mathit{Lifecyclenm} \\ \mathit{states} &: \mathit{Lifecyclenm} \rightarrow \mathbf{P}\ \mathit{Statenm} \end{aligned}$$

Given the introduction of *Nirvana* above, we know that for every life cycle name L,

$$\mathit{Nirvana} \in \mathit{states}(L)$$

A string in the Bill language is called a *Bill expression*. The formal syntax of a Bill expression is defined as *Expr* in Figure 4.1. Perhaps a bit surprising is the syntax of *Formalevents*, which considers ,,, as a proper text, but that is on purpose. It avoids cumbersome constraints in later sections of this chapter, when we say that $E{,}F$ is an example of *Formalevents* when E and F are.

$$\begin{array}{lcl}
\mathit{Expr} & ::= & \mathit{Var}\ @\ \mathit{Statenm} \\
 & \mid & \mathit{Var} = \mathit{Var} \\
 & \mid & \neg\ \mathit{Expr} \\
 & \mid & \mathit{Expr} \wedge \mathit{Expr} \\
 & \mid & \forall\ \mathit{Var} : \mathit{Lifecyclenm} \cdot \mathit{Expr} \\
 & \mid & \forall\ \mathit{Action}\ \mathit{Expr} \\
 & \mid & \mu\ \mathit{Loopnm} \cdot \mathit{Expr} \\
 & \mid & \mathit{Loopnm} \\
\mathit{Action} & ::= & [\ \mathit{Formalevents}\ /\ \mathit{Formalevents}\ ] \\
\mathit{Var} & ::= & \mathit{BillVar} \\
 & \mid & \mathit{Var}\ .\ \mathit{Linknm} \\
\mathit{Formalevents} & ::= & \\
 & \mid & \mathit{FormalEvent} \\
 & \mid & \mathit{Formalevents}\ ,\ \mathit{Formalevents} \\
\mathit{FormalEvent} & ::= & \mathit{Eventnm}\ \ (\ \mathit{Parms}\ ) \\
\mathit{Parms} & ::= & \mathit{Var} \\
 & \mid & \mathit{Var}\ ,\ \mathit{Parms}
\end{array}$$

A *Loopnm* may only occur positively in its binding $\mu \mathit{Loopnm} \cdot \_$, meaning that it may only be negated an even number of times.

Figure 4.1: Formal syntax of Bill.

Note that the form $\forall[\_/\_]\_$ does not bind any *BillVar*, only the form $\forall\_{:}\_\cdot\_$ does that.

We let p, q vary over *Expr*. We let e, f, g, h vary over *FormalEvent*. We let E, F, G vary over *FormalEvents*. We let A vary over *Action*.

4.2 The language William

William is the name of a sugar-coated version of Bill. The distinction between those flavours is only made in this chapter.

The language William is the language Bill plus the left hand side forms of the equations that follow in this section. The equations in this section can be read from left to right as definitions. The right hand sides only refer to prior definitions; as a result, there are no recursive definitions in this section. Furthermore, every annotation added by William can be rewritten in the core language Bill.

Logic complements. Many of the logic operators have a complementary version, which we introduce by definition in the customary way:

$$\begin{aligned}
p \vee q &= \neg(\neg p \wedge \neg q) \\
p \Rightarrow q &= (\neg p) \vee q \\
p \neq q &= \neg(p = q) \\
\exists x{:}L \cdot p &= \neg \forall x{:}L \cdot \neg p \\
\nu x \cdot p_x &= \neg \mu x \cdot \neg p_{\neg x} && (4.1) \\
\exists\, A\, p &= \neg \forall\, A\, \neg p && (4.2)
\end{aligned}$$

Definition 4.1 uses the form p_x to indicate that p is a function of x; the notation $p_{\neg x}$ is a function of $\neg x$, indicating that all references to x in p_x are replaced with $\neg x$.

CTL. The *Computation Tree Logic* [BCM+90], or CTL for short, is a notation of practical use; the following definitions express CTL operators in terms of William.

$$\begin{aligned}
EX\ p &= \exists\, [/]\, p \\
AX\ p &= \neg EX\ \neg p \\
EG\ p &= \nu I \cdot p \wedge \exists [/]\ I \\
AF\ p &= \neg EG\ \neg p \\
p\ EU\ q &= \mu I \cdot q \vee (p \wedge \exists [/]\ I) \\
p\ AW\ q &= \neg(\neg q\ EU\ \neg(p \vee q)) \\
p\ AU\ q &= (p\ AW\ q) \wedge (AF\ q) \\
p\ EW\ q &= (p\ EU\ q) \vee (EG\ p) \\
EF\ p &= \neg p\ EU\ p \\
AG\ p &= \neg EF\ \neg p
\end{aligned}$$

The proof that these definitions conform to standard CTL semantics is given later in this chapter.

Notational conveniences. The following rules are convenient for manually crafted and computer-generated specifications.

$$\begin{array}{lcl}
\mathit{false} & = & \mu I \cdot I \\
\mathit{true} & = & \neg \mathit{false} \\
[e/f]\ p & = & (\forall [e/f]\ p)\ \wedge\ (\exists [e/f]\ \mathit{true}) \\
x@S_1, S_2, \ldots & = & x@S_1 \vee x@S_2 \vee \ldots \\
x\dagger & = & x@\mathit{Nirvana} \\
\text{if } p \text{ then } q \text{ else } r & = & (p \Rightarrow q)\ \wedge (\neg p \Rightarrow r)
\end{array}$$

4.3 The μ-calculus, sec

Bill's syntax has been defined in the above. The semantics of Bill are defined in terms of the μ-calculus, which is defined in this section. The next sections apply the μ-calculus as a semantics language underpinning Bill. We follow the lines of literature [BCM$^+$90] with the syntactic variations of Appendix B and some other minor refinements.

Assume a finite set $\mathcal{V}$ of variable names and a finite set $\mathcal{R}$ of relation names. These are used to define the syntax of $\mathcal{E}$, a μ-calculus expression, in Figure 4.2; the customary constraints on arity apply; the μ-abstraction is formally monotone, meaning that the $\mathcal{R}$ inside a $\mu\mathcal{R}$ may only be negated an even number of time.

$$\begin{array}{lcl}
\mathcal{E} & ::= & \mathcal{V} = \mathcal{V} \\
 & | & \neg \mathcal{E} \\
 & | & \mathcal{E} \vee \mathcal{E} \\
 & | & \mathcal{X}(\mathcal{V}, \mathcal{V}, \ldots) \\
 & | & \exists \mathcal{V} \cdot \mathcal{E} \\
\mathcal{X} & ::= & \mathcal{R} \\
 & | & \lambda \mathcal{V} \cdot \lambda \mathcal{V} \cdot \ldots \cdot \mathcal{E} \\
 & | & \mu \mathcal{R} \cdot \mathcal{X}
\end{array}$$

Figure 4.2: The μ-calculus syntax, sec.

The μ-calculus has a domain $\mathcal{D}$, and variables from $\mathcal{V}$ translate to one

$$
\begin{array}{lcl}
\mathcal{I}\left(\alpha = \beta\right) & = & \mathcal{I}_{\mathcal{D}}\left(\alpha\right) = \mathcal{I}_{\mathcal{D}}\left(\beta\right) \\
\mathcal{I}\left(\neg\varphi\right) & = & \neg\mathcal{I}\left(\varphi\right) \\
\mathcal{I}\left(\varphi \vee \psi\right) & = & \mathcal{I}\left(\varphi\right) \vee \mathcal{I}\left(\psi\right) \\
\mathcal{I}\left(\exists\alpha \cdot \varphi\right) & = & \exists\kappa \in \mathcal{D} \cdot \text{let } \mathcal{I}_{\mathcal{D}}\left(\alpha\right){:=}\kappa \text{ in } \mathcal{I}\left(\varphi\right) \\
\mathcal{I}\left(\eta(\alpha, \beta, \ldots)\right) & = & \langle \mathcal{I}_{\mathcal{D}}\left(\alpha\right), \mathcal{I}_{\mathcal{D}}\left(\beta\right), \ldots\rangle \in \mathcal{I}\left(\eta\right) \\
\mathcal{I}\left(\zeta\right) & = & \mathcal{I}_{\mathcal{R}}\left(\zeta\right) \\
\mathcal{I}\left(\lambda\alpha \cdot \lambda\beta \cdot \ldots \cdot \varphi\right) & = & \{\langle \iota, \kappa, \ldots\rangle \in \mathcal{D} \times \mathcal{D} \times \ldots \cdot \\
 & & \quad \text{let } \mathcal{I}_{\mathcal{D}}\left(\alpha\right){:=}\iota, \mathcal{I}_{\mathcal{D}}\left(\beta\right){:=}\kappa, \ldots \text{ in } \mathcal{I}\left(\varphi\right)\} \\
\mathcal{I}\left(\mu\zeta \cdot \varphi\right) & = & \text{lfp } r \cdot \text{let } \mathcal{I}_{\mathcal{R}}\left(\zeta\right){:=}r \text{ in } \mathcal{I}\left(\varphi\right) \\
 & & \quad \text{(where lfp=least fixpoint w.r.t. } \subseteq\text{)}
\end{array}
$$

Figure 4.3: The μ-calculus semantics, sec.

of these with a function $\mathcal{I}_{\mathcal{D}} : \mathcal{V} \to \mathcal{D}$. The relations from $\mathcal{R}$ are interpreted by $\mathcal{I}_{\mathcal{R}}$ as a relation between an arity-dependent number of elements from $\mathcal{D}$, as in $\mathcal{I}_{\mathcal{R}} : \mathcal{R} \to \mathbf{P}\left(\mathcal{D} \times \mathcal{D} \times \ldots\right)$

The interpretation of a μ-calculus formula with respect to the structure $\langle \mathcal{D}, \mathcal{I}_{\mathcal{D}}, \mathcal{I}_{\mathcal{R}}\rangle$ of an expression $\mathcal{E}$ is defined in the function $\mathcal{I}$ that refers to $\mathcal{I}_{\mathcal{D}}$ and $\mathcal{I}_{\mathcal{R}}$, as defined in Figure 4.3. This definition assumes $\alpha, \beta \in \mathcal{V}$ and $\zeta \in \mathcal{R}$ and $\varphi, \psi \in \mathcal{E}$ and $\eta \in \mathcal{X}$ and $\iota, \kappa \in \mathcal{D}$.

4.4 Our own μ-calculus flavour

The previous sections defined Bill and the μ-calculus; we now turn to an application of the μ-calculus and define Bill semantics in its turn.

4.4.1 New sets

In addition to the sets defined by lexemes for the Bill language, there are some more sets required for the definition of the semantics.

- A set of life cycle instances *Life*; we let i, j, k vary over *Life*.
- A set of snapshots *Snap*; we let W vary over *Snap*.
- A set of snapshot variables *SnapVar*; we let Q, R vary over *SnapVar*.

- A set of actual event sets *ActualEvents*; we let C vary over *ActualEvents*.
- A set of actual event set variables *ActEvVar*; we let O vary over *ActEvVar*.

These sets are assumed to be disjoint from the sets of lexemes introduced before.

This thesis distinguishes itself from customary model checking by allowing these three sets to be countably infinite in size, both in theory *and in the prover*.

We postulate the following function:

$$\mathit{lifetp} \quad : \quad \mathit{Life} \rightarrow \mathit{Lifecyclenm}$$

The definition of this function is dependent on the actual life cycle system being modelled. We make an assumption that every life cycle has at least one instance, or formally

$$\mathit{Lifecyclenm} \quad = \quad \mathit{Ran}(\mathit{lifetp})$$

Where *Ran* determines the range of the *lifetp* function.

4.4.2 Using functions in our μ-calculus

The μ-calculus is a calculus of relations; in this application however, some relations are functions. We use functional notation in our μ-calculus formulations, such as

$$y = f(x)$$

In terms of our μ-calculus, this is the use of a relation

$$f(x, y)$$

We only use the functional notation on functions, or in other words when the following holds:

$$f(x, y) \wedge f(x, z) \Rightarrow y = z$$

We will also use this form with multiple relational parameters instead of single ones; where useful, our meta-notation uses vector notation for an arbitrary number of parameters.

We will write $\twoheadrightarrow$ to denote a partial function; all other functions mentioned in this chapter are complete.

4.4.3 Language of our μ-calculus

The μ-calculus language is of the form Expr_μ according to the standard notation in Figure 4.4. In addition to this, we shall assume $\wedge$ and $\Rightarrow$ operators with the usual definitions. This definition is in accordance with existing work [BCM+90]. An expression in this language is considered as a

$$
\begin{array}{lcll}
\mathit{Expr}_\mu & ::= & \mathit{BillVar} = \mathit{BillVar} & \\
 & \mid & \neg \mathit{Expr}_\mu & \\
 & \mid & \mathit{Expr}_\mu \vee \mathit{Expr}_\mu & \\
 & \mid & \mathit{Rel}_\mu(\mathit{Term}_\mu, \ldots, \mathit{Term}_\mu) & \\
 & \mid & \exists \mathit{BillVar} \cdot \mathit{Expr}_\mu & \\
 & \mid & \exists \mathit{ActEvVar} \cdot \mathit{Expr}_\mu & \\
\mathit{Rel}_\mu & ::= & \mathit{St} \mid \mathit{Tr} \mid \mathit{Ln} & \\
 & \mid & \mathit{Eventnm} & \\
 & \mid & \mathit{Loopnm} \quad \text{(bound by } \mu\text{)} & \\
 & \mid & \lambda \mathit{SnapVar} \cdot \mathit{Expr}_\mu & \\
 & \mid & \mu \mathit{Loopnm} \cdot \mathit{Rel}_\mu & \\
\mathit{Term}_\mu & ::= & \mathit{BillVar} \mid \mathit{SnapVar} \mid \mathit{ActEvVar} \mid \mathit{Linknm} \mid \mathit{Statenm} &
\end{array}
$$

Figure 4.4: Syntax of our μ-calculus flavour's expressions.

part of a larger μ-calculus expression according to Section 4.3; the relation names St, Tr and Ln are considered constants in our flavour of μ-calculus in Figure 4.4 but they can also be thought of as defined in the context added by that larger expression.

A simple comparison between Figures 4.2 and 4.4 shows that Expr_μ expressions are a subset of $\mathcal{E}$ expressions. As a result, the same interpretation function $\mathcal{I}$ can be used; only the functions $\mathcal{I}_\mathcal{D}$ and $\mathcal{I}_\mathcal{R}$ need more refinement for our flavour of the μ-calculus. These interpretation functions are defined in the following subsections.

4.4.4 Domain of our μ-calculus

For our semantics of Bill, we define a μ-calculus domain as the union of a number of disjoint sets:

$$\mathcal{D} \quad = \quad \mathit{Life} \cup \mathit{Snap} \cup \mathit{ActualEvents} \cup \mathit{Linknm} \cup \mathit{Statenm}$$

Recall that the last two depend on the life cycle application being expressed in terms of Bill.

The domain interpretation functions $\mathcal{I}_{\mathcal{D}}$ is an identity for elements of *Linknm* and *Statenm*; after all, these are constants. Furthermore, a mapping $\mathit{BillVar} \rightarrow \mathit{Life}$ is part of $\mathcal{I}_{\mathcal{D}}$, plus a mapping $\mathit{SnapVar} \rightarrow \mathit{Snap}$, plus a mapping $\mathit{ActEvVar} \rightarrow \mathit{ActualEvents}$. Nothing else is part of $\mathcal{I}_{\mathcal{D}}$.

Note how this definition avoids that $(x \mapsto W) \in \mathcal{I}_{\mathcal{D}}$ if $x : \mathit{BillVar}$ and $W : \mathit{Snap}$ — this is effectively a type discipline.

4.4.5 Relations and functions in our μ-calculus

The relations in our μ-calculus depend on the life cycle system being modelled. Therefore, the following is a type declaration of the relations for our μ-calculus, but not a complete definition.

We postulate the following interpretations of functions and relations for our μ-calculus:

$$\begin{aligned}
\mathcal{I}_{\mathcal{R}}\,(\mathit{St}) &: \mathit{Snap} \times \mathit{Life} \rightarrow \mathit{Statenm} \\
\mathcal{I}_{\mathcal{R}}\,(\mathit{Tr}) &: \mathbf{P}\,(\mathit{Snap} \times \mathit{ActualEvents} \times \mathit{Snap}) \\
\mathcal{I}_{\mathcal{R}}\,(\mathit{Ln}) &: \mathit{Snap} \times \mathit{Life} \rightarrow \mathit{Linknm} \twoheadrightarrow \mathit{Life}
\end{aligned}$$

These functions describe runs of a life cycle system; their definition is dependent on the actual life cycle system being modelled. We will write *Ln* as a ternary function in what follows.

A few constraints apply to these functions:

$$\begin{aligned}
\mathcal{I}_{\mathcal{R}}\,(\mathit{Ln})\,(W, i, \mathsf{this}) &= i \\
\mathit{lifetp}(\mathcal{I}_{\mathcal{R}}\,(\mathit{Ln})\,(W, i, l)) &= \mathit{linktp}\,(\mathit{lifetp}\,(i), l) \\
\mathcal{I}_{\mathcal{R}}\,(\mathit{St})\,(W, i) &\in \mathit{states}\,(\mathit{lifetp}\,(i))
\end{aligned}$$

In addition to the foregoing, there are relations *Eventnm* that represent the occurrence of an event with that name. For every $a : \mathit{Eventnm}$, there are expected parameter types $L_1, L_2, \ldots : \mathit{Lifecyclenm}$. For such a there is a function

$$\mathcal{I}_{\mathcal{R}}\,(a) \quad : \quad \mathit{ActualEvents} \twoheadrightarrow X_1 \times X_2 \times \ldots$$

Where $X_k = \{i : \mathit{Life} | \mathit{lifetp}(i) = L_k\}$ for $k \in 1, 2, \ldots$ Nothing else is mapped by the $\mathcal{I}_{\mathcal{R}}$ function.

Functions like these define the parameters of a named event. This is dependent on an *ActualEvents* parameter to express that different *ActualEvents* can have different arguments for the event. The dependency of an *ActualEvents* parameter is partial because not every *ActualEvents* contains every event name.

4.5 Semantics of Bill

This section starts by introducing three auxiliary functions that express when an *ActualEvents* member is matched, ignored or blocked by a *FormalEvents* member. Given these functions, the semantics of Bill can be defined in terms of the μ-calculus from the previous section.

4.5.1 Definition of match, ignore, block

The functions *match*, *ignore* and *block* define the three possible responses that an action $[E/F]$ has to an actual event represented with $O : \mathit{ActEvVar}$.

$$\begin{aligned}
\mathit{match}(O, [E/F]) &= \bigwedge_{a(\vec{x}) \in E} a(O) = \vec{x} \ \wedge \bigwedge_{b(\vec{y}) \in F} b(O) \neq \vec{y} \\
\mathit{ignore}(O, [E/F]) &= E \neq \emptyset \ \wedge \bigwedge_{b(\vec{y}) \in E \cup F} b(O) \neq \vec{y} \\
\mathit{block}(O, [E/F]) &= \neg\mathit{match}(O, [E/F]) \wedge \neg\mathit{ignore}(O, [E/F])
\end{aligned}$$

These definitions make a minor leap of faith, using the form $a(\vec{x}) \in E$ to express that the syntax in E is interpreted as a set of *FormalEvent* values, and by splitting each of those in an *Eventnm* a and its parameters $\vec{x}$, which is a vector of *BillVar*.

Proposition 1 (Partition) *The functions match, ignore and block partition the space ActualEvents* $\times$ *Action.*

Proof 1 *We show that every point* $(O, [E/F])$ *in the space ActualEvents* $\times$ *Action, when supplied as argument to the three functions, makes precisely one of the functions true.*

First, if $E = \emptyset$*, then clearly* $\mathit{ignore}(O, [E/F]) = \mathit{false}$*, which reduces the definition of block to the negation of match. So, for such points in the space, the proposition holds.*

All other points have $E \neq \emptyset$, meaning that there exists some $a(\vec{x}) \in E$. The statements $(a(O) = \vec{x}) \wedge (a'(O) = \vec{x}') \wedge \ldots$ and $(\neg a(O) = \vec{x}) \wedge (\neg a'(O) = \vec{x}') \wedge \ldots$ cannot both be true. It may happen that neither of these lists evaluates to true causing neither match nor ignore return true; another cause for neither of these results can be due to the $b(\vec{y}) \in F$. In these cases, block returns true. Moreover, block only *returns true when both match and ignore return false, so it never causes overlap.* □

Proposition 2 (Empty action) *The extreme action $[/]$ leads to the following constant behaviour:*

$$\begin{aligned} \mathit{match}(O,[/]) &= \mathit{true} \\ \mathit{ignore}(O,[/]) &= \mathit{false} \\ \mathit{block}(O,[/]) &= \mathit{false} \end{aligned}$$

Proof 2 *A mere calculation.* □

4.5.2 Translating Bill to μ-calculus

The previous sections defined the syntax of Bill, as well as a μ-calculus. The semantics of a Bill formula is defined as a μ-calculus expression Expr_μ in Figure 4.5. This scheme generally yields a function that binds a *Snap* member to a *SnapVar* member such as Q. A point deserving special attention is the form for loops, $\mu I \cdot p_I$. The translation yields a loop named I, and a *SnapVar* Q is bound inside the loop to reflect that a new *Snap* can be bound on every loop 'iteration'.

Every linked instance $x.l$ is translated to a new name y such that $\mathit{Ln}(Q, x, l) = y$, so that only *BillVar* members need to be considered to handle *Life*s. The members of *BillVar* are literally translated to our μ-calculus syntax; the $\mathcal{I}_\mathcal{D}$ function will bind them to *Life* members.

The form $\forall[E/F]\,p$ is the most complicated one; its semantics depends on the relationship between the action $[E/F]$ and an *ActualEvent* bound by *ActEvVar* O. This relation is expressed with the *match* and *ignore* functions; when O is ignored, the $\forall[E/F]\,p$ still applies; when O is matched, then p will apply for the *Snap* bound to R.

Perhaps useful is to give the inverted form of this line, which reads:

$$\begin{aligned} [\![\exists[E/F]\,p]\!] \quad = \quad & \mu I \cdot \lambda Q \cdot \exists O, R \cdot \mathit{Tr}(Q, O, R) \wedge \\ & \qquad \mathit{match}(O, [E/F]) \wedge [\![p]\!](R) \\ & \qquad \vee \mathit{ignore}(O, [E/F]) \wedge I(R) \end{aligned}$$

$$
\begin{array}{lcl}
[\![x@S]\!] & = & \lambda Q \cdot St(Q,x) = S \\
[\![x = y]\!] & = & \lambda Q \cdot x = y \\
[\![\neg p]\!] & = & \lambda Q \cdot \neg [\![p]\!](Q) \\
[\![p \wedge q]\!] & = & \lambda Q \cdot [\![p]\!](Q) \wedge [\![q]\!](Q) \\
[\![\forall x{:}L \cdot p]\!] & = & \lambda Q \cdot \forall x \cdot [\![p]\!](Q) \quad (\text{where } \mathit{lifetp}(x) = L) \\
[\![\forall [E/F]\, p]\!] & = & \nu I \cdot \lambda Q \cdot \forall O, R \cdot \mathit{Tr}(Q,O,R) \Rightarrow \\
 & & \qquad \mathit{match}(O,[E/F]) \Rightarrow [\![p]\!](R) \\
 & & \quad \wedge \mathit{ignore}(O,[E/F]) \Rightarrow I(R) \\
[\![\mu I \cdot p_I]\!] & = & \mu I \cdot \lambda Q \cdot [\![p_I]\!](Q) \\
[\![I]\!] & = & I \\
[\![p_{x.l}]\!] & = & \lambda Q \cdot \exists y \cdot Ln(Q,x,l) = y \wedge \\
 & & \bigvee_{s \in \mathit{states}(\mathit{linktp}(\mathit{lifetp}(x),l))} St(Q,x) = s \wedge [\![p_y]\!] \quad (\text{fresh } y)
\end{array}
$$

Figure 4.5: Formal semantics of Bill.

4.6 Model theory

This section describes the structure of a model theory for the aforementioned languages.

4.6.1 Labelled Kripke structures

Definition 1 (Labelled Kripke) *A labelled Kripke structure is a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{T}, \mathcal{P} \rangle$ where*

- *$\mathcal{S}$ is a set (representing the possible snapshots of an execution);*
- *$\mathcal{T}$ is a ternary relation of type $\boldsymbol{P}(\mathit{Snap} \times \mathit{ActualEvents} \times \mathit{Snap})$ (representing labelled transitions of the form $\langle Q, O, R \rangle$ to denote that ActualEvents O leads from snapshot Q to snapshot R); and*
- *$\mathcal{P}$ is a function from Snap (representing the 'current moment') to a set of elementary relationships $\langle i, S \rangle$ to indicate that Life i is in state S, and to a set of elementary relationships $\langle i, l, j \rangle$ to indicate that Life i has a link named l to Life j.*

4.6.2 Semantics of labelled Kripke structures

The labelled Kripke structure contains all possible executions of a running life cycle system. We will not completely specify the Kripke structure, but instead define functions that extract perspectives on its internal structure. These functions can be used as the model for Expr_μ expressions.

Definition 2 (Domain derivation) *Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{T}, \mathcal{P} \rangle$ be a labelled Kripke structure. We overload some previously introduced names to define functions with a similar intention, but derived from a Kripke structure.*

$$\begin{array}{lcl}
\mathit{Life}(\langle \mathcal{S}, \mathcal{T}, \mathcal{P} \rangle) & = & \textit{all actual parameters in } \mathcal{T} \\
\mathit{Snap}(\langle \mathcal{S}, \mathcal{T}, \mathcal{P} \rangle) & = & \mathcal{S} \\
\mathit{ActualEvents}(\langle \mathcal{S}, \mathcal{T}, \mathcal{P} \rangle) & = & \{ \; W \mid \langle Q, W, R \rangle \in \mathcal{T} \; \} \\
\mathit{Linknm}(\langle \mathcal{S}, \mathcal{T}, \mathcal{P} \rangle) & = & \{ \; l \mid (Q \mapsto X) \in \mathcal{P}, \langle i, l, j \rangle \in X \; \} \\
\mathit{Statenm}(\langle \mathcal{S}, \mathcal{T}, \mathcal{P} \rangle) & = & \{ \; S \mid (Q \mapsto X) \in \mathcal{P}, \langle i, S \rangle \in X \; \} \\
\mathit{St}(\langle \mathcal{S}, \mathcal{T}, \mathcal{P} \rangle) & = & \{ \; \langle Q, i, S \rangle \mid (Q \mapsto X) \in \mathcal{P}, \langle i, S \rangle \in X \; \} \\
\mathit{Tr}(\langle \mathcal{S}, \mathcal{T}, \mathcal{P} \rangle) & = & \mathcal{T} \\
\mathit{Ln}(\langle \mathcal{S}, \mathcal{T}, \mathcal{P} \rangle) & = & \{ \; \langle Q, i, l, j \rangle \mid (Q \mapsto X) \in \mathcal{P}, \langle i, l, j \rangle \in X \; \}
\end{array}$$

The domain and relations of the μ-calculus model are now, respectively:

$$\begin{array}{lcl}
\mathcal{D}(\mathcal{M}) & = & \mathit{Life}(\mathcal{M}), \mathit{Snap}(\mathcal{M}), \mathit{ActualEvents}(\mathcal{M}), \mathit{Linknm}(\mathcal{M}), \mathit{Statenm}(\mathcal{M}) \\
\mathcal{R}(\mathcal{M}) & = & \mathit{St}(\mathcal{M}), \mathit{Tr}(\mathcal{M}), \mathit{Ln}(\mathcal{M})
\end{array}$$

4.6.3 Model theory

Definition 3 (Bill and Kripke) *Given a Bill expression P, a labelled Kripke structure $\mathcal{M}$ and an initial snapshot Q, the ternary relation $\models$ is defined by*

$$\mathcal{M}, Q \models P \quad \textit{if and only if} \quad \mathcal{D}(\mathcal{M}), \mathcal{R}(\mathcal{M}) \models_\mu [\![P]\!](Q)$$

Bill can be classified as a multi-modal language, where the modalities are defined as follows for a Bill property p:

$$\begin{array}{lcl}
\Box_{[E/F]} p & = & \forall [E/F] \, p \\
\Diamond_{[E/F]} p & = & \exists [E/F] \, p
\end{array}$$

Apparently, a relationship between Bill and multi-modal languages can be made — but we don't work it out here, as it does not serve the purposes of this thesis.

4.6.4 Conservative extension of CTL

The temporal constraint language CTL [BCM+90] and subsets of CTL are often used in model checkers. Like William, it can be translated to μ-calculus, but CTL does not quantify over variables, and is unaware of the actions that take place.

Proposition 3 (Bill extends CTL) *William is a conservative extension of CTL.*

Proof 3 *CTL operators are defined in Section 4.2. The operators EX, EG and EU form a minimal basis for CTL, the other CTL operators are defined in terms of these as customary [BCM+90]. Because of this, we only need to prove the conservative extension for the minimal basis.*

This proof is performed by starting from CTL expressions, using the equations in Section 4.2 to translate them to William, translate this result to Bill, and translate Bill to μ-calculus. To prove that William is a conservative extension of CTL, the result must be the same as found in literature.

$$\begin{aligned}
[\![EX\ p]\!] &= [\![\exists\ [/]\ p]\!] \\
&= \lambda Q \cdot \exists O \cdot \exists R \cdot Tr(Q,O,R) \wedge [\![p]\!](R) \\
[\![EG\ p]\!] &= [\![\nu I \cdot p \wedge \exists[/]\ I]\!] \\
&= \nu I \cdot \lambda Q \cdot [\![p \wedge \exists[/]\ I]\!](Q) \\
&= \nu I \cdot \lambda Q \cdot [\![p]\!](Q) \wedge \exists O \cdot \exists R \cdot Tr(Q,O,R) \wedge I(R) \\
[\![p\ EU\ q]\!] &= [\![\mu I \cdot q \vee (p \wedge \exists[/]\ I)]\!] \\
&= \mu I \cdot \lambda Q \cdot [\![q \vee (p \wedge \exists[/]\ I)]\!](Q) \\
&= \mu I \cdot \lambda Q \cdot [\![q]\!](Q) \vee ([\![p]\!](Q) \wedge \exists O, R \cdot Tr(Q,O,R) \wedge I(R)
\end{aligned}$$

These results conform to accepted literature about CTL [BCM+90]. Specifically note that the form $Tr(Q,O,R)$ corresponds to the 'neighbouring' relation $N(Q,R)$ — the additional parameter O is not used in the results above, and O is always bound together with R, so that there are no differences due to quantification either.

We conclude that the CTL operator syntax in Bill has the semantics commonly defined for CTL operators, so that Bill is capable of expressing everything that CTL can express. This makes Bill an extension of CTL; and because the operators EX, EG and EU cover all operators of CTL, we conclude that Bill is a conservative extension of CTL. □

4.7 Life cycle semantics

This section details how life cycle diagrams are translated to expressions in William. It starts with some preliminaries, and gradually builds up a translation scheme for life cycles.

4.7.1 Life cycle states

The semantics of a life cycle diagram are defined as a conjunction of propositions for the states in the life cycle. For a life cycle named L with a state $S \in \mathit{states}(L)$, such propositions take the form

$$AG\,\forall x : L \cdot x@S \Rightarrow p$$

Note that *Nirvana* is also a state name in $\mathit{states}(L)$. The form of p shall be described in what follows; it is concerned with the pre/postcondition relationship for transitions.

The algorithms presented below calculate each of these forms and present them by prefixing them with `we know`. Any free variables in these formulæ are universally qualified — this is left implicit, as usual. The result of all such William expressions (now without free variables) are composed with conjunction to form the total knowledge about a life cycle system.

4.7.2 Dynamic alphabets

Process algebras [BW90] usually define a fixed alphabet for the entire duration of a process; in terms of life cycles, that would be throughout the existence of a life, namely forever. Most process algebras do not really deal with parameters. Instead of modelling the parameter x in $a(x)$, all possible values of x will be considered, and all possibly resulting 'actual events' $a(x)$ are considered part of the static alphabet.

Claim 1 (Parameters cannot be static) *A process theory with parameters is counter-intuitive if its alphabet is static.*

Motivation 4 *Imagine a process with a transition $a(x)$, where x is a parameter whose value can either be i or j. All possibly occurring events must be in the process's static alphabet, including a_i and a_j.*
Now consider one instance of this process in a state from which only the $a(x)$ transition departs; assume that variable x has value i. To this process, $a(x)$ is the same as a_i. In that state, the process can make a step a_i but

not a step a_j. This means that this process, when it communicates with any other process, excludes a_j as a possible step.
If another process instance is in the same state, except that variable x is set to value j, then that process can make a step a_j but not a step a_i. When these two processes in those states communicate, than the composition can neither make a step a_i nor a step a_j.
When a practical concurrent system offers blocking as an option, it is usually with the intention to support synchronisation concepts of some sort (and none seems to exist for the blocking of a_i and a_j). Blocking too much is undesirable, because it easily leads to deadlock. The blocking of events without an apparent reason makes these semantics counter-intuitive, and it is certainly unsuitable for life cycles. $\square$

A solution to this problem is to make an alphabet dependent on variable values. Such an alphabet would contain variables such as x, and rely on a binding function to determine which values belong to it.

An instance of a life cycle 'reads' variables with ?x parameters, and before doing this the value of the variable x is unknown. This means that the binding function cannot provide a value for every variable in every state. We therefore need to know the set of known variables per life cycle state; this has been incorporated in the life cycle syntax presented in Chapter 3.

Life cycle variables are modelled in William as link names *Linknm*, because they define a link from one instance to another.

Definition 4 (States and links) *The function*

$$\mathit{statelinks} : \mathit{Lifecyclenm} \times \mathit{Statenm} \rightarrow \boldsymbol{P}\, \mathit{Linknm}$$

is the smallest function such that for every Lifecyclenm L,

1. $\mathit{statelinks}(L, \mathit{Nirvana}) = \{\mathsf{this}\}$*; and*
2. *for every* $S \in \mathit{states}(L)$, $\mathsf{this} \in \mathit{statelinks}(L, S)$*; and*
3. *for every life cycle variable declaration* $x{:}L@\ldots S\ldots$ *it holds that* $x \in \mathit{statelinks}(L, S)$.

The value states(L) gives the states mentioned in the diagram for life cycle L, plus the state Nirvana.

For example, in Figure 3.6, we find $\mathit{states}(\mathsf{Book}) = \{\mathsf{Available}, \mathsf{Borrowed}, \mathit{Nirvana}\}$ and $\mathit{statelinks}(\mathsf{Book}, \mathsf{Available}) = \{\mathsf{this}, \mathsf{t}\}$.

Definition 5 (Static alphabet) *The static alphabet α_L of the life cycle named L is the set of all the formal events mentioned in its graphical representation.*

This definition means that formal parameters are literally mentioned in the alphabet; connectives between formal events such as $\wedge$ are not. For example, in Figure 3.6, we find $\alpha_{\mathsf{Book}} = \{$acq(this,?t), ckout(this,?m), return(this), discard(this), avail(this,t), avail(this,?t)$\}$.

Definition 6 (Dynamic alphabet) *The dynamic alphabet $\beta_L(S)$ of the life cycle named L, and in state $S \in states(L)$ is defined as*

$$\beta_L(S) \quad = \quad \{a(\vec{l}) \in \alpha_L \mid outvars(\vec{l}) \subseteq statelinks(L, S)\}$$

where $outvars(\vec{l})$ is the list of variables that occur as output variables in $\vec{l}$.

For example, in Figure 3.6, we find $\beta_{\mathsf{Book}}(\mathsf{Available}) = \{$acq(this,?t), ckout (this,?m), return(this), discard(this), avail(this,t), avail(this,?t) $\}$.

In addition to these functions, we define a few helper functions for use in the following sections.

Definition 7 $evnames(E) = \{a \mid a(\vec{l}) \in E\}$

Definition 8 *$\beta_L^{-E}(S)$ contains the formal events from $\beta_L(S)$ except those whose name occurs in $evnames(E)$.*

4.7.3 The action for a transition

A transition contains an expression with formal events. After the normalisation in Section 3.3, these expressions are shaped as

$$\bigwedge_{e \in E} e \;\wedge\; \bigwedge_{f \in F} \neg f$$

Based on this form, a first thought might be to translate this directly to a Bill action $[E/F]$; this approach is wrong because there is a dynamic alphabet *per state*, not per transition.

Definition 9 (Transition translation) *Let E and F represent the 'demanded' and 'forbidden' formal events on a transition t. Let S be the starting state for t, and let L be the name of the containing life cycle. Then the action that matches the transition is*

$$act(t) \quad = \quad [E/F, \beta_L^{-E \cup F}(S)]$$

Below, we introduce algorithms to generate (a) possibilities and (b) certainties. For every transition, there will be a we know statement expressing the possibility of the transition. Depending on the existence of alternative transitions with overlap in matched actual events, there will be more or less certainty of the we know statement that captures the transition. (This is in support of non-deterministic process specifications.)

4.7.4 Blocking in a state

The action that describes what a transition matches has been defined in the previous section. The *transition* will ignore everything else, but the *originating state* may cause a block of certain formal events.

Below, we introduce an algorithm to generate certainties; this algorithm will generate the condition for blocking, in the form of a transition with post condition *false*.

4.7.5 Ignoring event occurrences

Every life cycle instance has its alphabet $\beta_L(S)$ and a variable binding defining which actual events are matched or blocked; all the others are ignored. When an *ActualEvents* member comprises only of actual events to be ignored, the whole actual event may be ignored.

This is part of the we know statements generated by the following algorithms, because $act(t)$ ever contains more than currently accepted in $\beta_L(S)$. William semantics in turn, define how ignoring of an actual event by an $act(t)$ is handled.

4.7.6 Internal steps

Life cycles can describe *internal transitions* (commonly written as τ steps). Such transitions match nothing of the current alphabet, so $act(\tau) = [/\beta_L(S)]$. This rolls out of the $act(t)$ definition for transitions with no event annotation.

The semantics of internal steps involve uncertainty of whether the step has taken place; this uncertainty can be captured using a non-deterministic alternative transition back to the originating state. For this reason, we shall silently add a transition from S to S annotated with no events to every life cycle diagram before we generate its semantical structures.

4.7.7 Treatment of unchanged variables

Consider a transition as a state change; life cycle variables have a prior value and a value-after, represented as l and $?l$, respectively. The question-mark is considered part of the name. If these symbols are universally quantified with a single transition as scope, then it is not problematic to use the same naming for other transitions influencing the same variable.

When a life cycle instance makes a transition, it can read a value for a variable l with a parameter $?l$. Nothing can be derived from the fact that a new variable binding is adopted. The opposite situation, where no new variable binding is adopted for a variable m, does allow for a conclusion, namely that $?m = m$. This knowledge will be derived for any transition that does not contain a $?m$ parameter.

Definition 10 *$\mathcal{U}_t$ is the set of unchanged variables over transition t.*

Where links from life cycle variables are concerned, only the start of the traversal is treated this way. It is up to a reasoning system with more overview over process cooperation to draw conclusions on unchanged variables in referred instances.

4.7.8 Preconditions, postconditions

Every transition has a precondition and a postcondition, and this subsection defines them as functions $pre(t,x)$ and $post(t,x)$, where t is the transition, and x is a Bill variable representing the affected life cycle instance.

The precondition need not include anything related to the events on t, because the action $act(t)$ already handles those; but there may also be a guard on a transition, and that should go into the precondition. After the normalisation of Section 3.3, the form of a guard is a conjunction of (possibly negated) constraints on states of life cycle variables.

Definition 11 *The translation scheme $\mathcal{P}_l$ translates a guard expression into a William expression:*

$$\begin{array}{lcl}
\mathcal{P}_x\left[p \wedge q\right] & = & \mathcal{P}_x\left[p\right] \ \wedge \ \mathcal{P}_x\left[q\right] \\
\mathcal{P}_x\left[\mathsf{y@S}\right] & = & x.\mathsf{y}@\mathsf{S} \\
\mathcal{P}_x\left[\neg \mathsf{y@S}\right] & = & \neg\mathcal{P}_x\left[\mathsf{y@S}\right]
\end{array}$$

Definition 12 *The William precondition $pre(t,x)$ for transition t of life cycle instance x is*

$$pre(t,x) \quad = \quad \mathcal{P}_x\left[guard(t)\right]$$

where $guard(t)$ is the guard expression on t in the life cycle diagram.

The postcondition of a transition need only express unchanged variables over a transition.

Definition 13 *The postcondition of a transition t on a life cycle instance x with unchanging variables $\mathcal{U}_t$ is*

$$post(t,x) \quad = \quad \bigwedge_{\mathsf{x} \in \mathcal{U}_t} l.\mathsf{x} = ?l.\mathsf{x} \ \wedge \ x@destination(t)$$

4.7.9 Generating knowledge about life cycles

Knowledge about life cycles is split in *possibilities* (or existentially quantified statements) and *certainties* (or universally quantified statements). These statements are independently generated for every state named S in every life cycle named L. The prefix we know indicates a statement to be added to a conjugated set of properties that express knowledge about life cycles.

The possibilities express only that a transition can occur, and the algorithm in Figure 4.6 generates these statements in terms of William.

```
for all transitions $t$ departing from $S$:
  we know $AG\,\forall x{:}L \cdot x@S \wedge pre(t,x) \Rightarrow \exists act(t)\, post(t,x)$
```

Figure 4.6: Algorithm to generate possibilities.

It is more complicated to generate the certainties after occurrence of an actual event, because multiple transitions may match it, leading to non-deterministic results. Furthermore, this non-determinism can depend on variable binding and preconditions to transitions.

To handle all situations, we construct an algorithm that generates a we know property for all thinkable combinations of matchable events and sets of matching transitions. Many such combinations will be impossible, but for reasons of clarity we leave out the optimisations that would avoid those combinations.

Each matchable event is described with a set $E_Y \subseteq \beta_L(S)$, or in terms of actions, $[E_Y/E_N]$ where E_Y, E_N partitions $\beta_L(S)$.

The sets of matching transitions can be derived by partitioning the transitions that depart from S in sets T_Y, T_N for matching and non-matching transitions, respectively. For a transition $t \in T_Y$ to match, its precondition $pre(t, x)$ must hold and its action $act(t)$ must match. The conclusion afterwards is that the postcondition $post(t, x)$ for *one* of the $t \in T_Y$ holds.

The transitions in $t \in T_N$ can fail for two reasons. Either the precondition $pre(t, x)$ fails, or the action $act(t)$ does not match. The algorithm reflects this by partitioning T_N into T_{N1} and T_{N2}.

The algorithm that takes all these issues into account is presented in Figure 4.7.

```
for all $E_Y, E_N$ that partition $\beta_L(S)$:
  for all $T_Y$, $T_{N1}$, $T_{N2}$ partitioning transitions from $S$:
    let $pre_x = \bigwedge_{t \in T_Y} pre(t, x) \;\wedge\; \bigwedge_{t \in T_{N1}} \neg pre(t, x)$
    let $post_x = \bigvee_{t \in T_Y} post(t, x)$
    for all $miss \in \{\; nope(i, elt(j, T_{N2}))$
                         $\mid i \in \mathbf{N}, j \in 1..size(T_{N2}) \;\}$:
      let $act = matchall(\{[E_Y/E_N]\} \cup miss \cup map\;\; act\;\; T_Y)$
      we know $AG\,\forall x{:}L \cdot x@S \wedge pre_x \Rightarrow \forall act\, post_x$
def $nope(i,S)$:
  the $i$th counter-example to $[E/F]$ from
    $[/e]$ for $e \in E$; and
    $[f/]$ for $f \in F$.
def $matchall(\{[E_1/F_1], \ldots, [E_n/F_n]\})$:
  $[E_1, \ldots, E_n/F_1, \ldots, F_n]$
```

Figure 4.7: Algorithm to generate certainties.

When a new system is created from scratch, then all its instances are in *Nirvana* state. For such a situation, the addition knowledge of Figure 4.8 may be added to the currently known facts.

```
we know $\forall x{:}L \cdot x@Nirvana$
```

Figure 4.8: Bootstrapping knowledge.

4.7.10 Properties of life cycle knowledge

Definition 14 (Impossible statements) *We shall call a* we know *statement of the form $AG\,\forall x{:}L \cdot l@S \wedge pre \Rightarrow \forall act\; post$ an impossible statement* *when either one of the following holds:*

- $pre = false$
- *act contradicts itself; this happens when it is of the form* $[e,\ldots/e,\ldots]$

If a we know *statement is not impossible, we call it possible.*

Proposition 4 (Impossible is trivial) *Some of the* we know *statements in the outcome of the algorithm in Figure 4.7 describe impossible transitions. Those can be removed without a change of meaning of the algorithm's outcome.*

Proof 5 *The statements from the algorithm are impossible when* $pre = false$ *or when act contradicts itself.*

In the case where $pre = false$, *the whole formula equals* $AG\,\forall x{:}L{\cdot} false \Rightarrow \ldots$ *which equals* $true$. *The* we know *statements are conjugated, so this outcome is neutral and may be discarded without any influence.*

In the case where act contradicts itself, it must be of the form $[e,\ldots/e,\ldots]$ *which causes the* $\forall\, act$ *to range over an empty set. This reduces the* we know *formula to* $AG\,\forall x{:}L \cdot x@S \wedge pre \Rightarrow true$, *which equals* $true$. *Again, this outcome can be discarded without any influence.*

This proves that the we know *statement can be removed without a change of meaning to the algorithm's outcome.* □

Proposition 5 (Predictable actions) *In the algorithm of Figure 4.7, all possible* we know *statements have* $act = [E_Y/E_N]$.

Proof 6 *It is always true that* $E_Y \cup E_N = \beta_L(S)$. *Because no formal events outside* $\beta_L(S)$ *can occur in state* S, *the only way to construct a different act value is by creating an overlap between* E_Y *and* E_N, *leading to an act of the form* $[e,\ldots/e,\ldots]$; *but this contradicts the requirement that the transition should be possible.*

So, no possible transitions can be constructed with $act \neq [E_Y/E_N]$. □

Proposition 6 (No match) *When none of the possible* we know *statement from the algorithm in Figure 4.7 matches, then it will be blocked altogether.*

Proof 7 *If no possible statement matches, then* $T_Y = \emptyset$. *As a result, post is a* $\vee$ *summation over zero terms, thereby reducing to* $false$. *A statement* $\forall A\, false$ *makes everything matching action* A *inconsistent, which is a formal way of blocking* A *altogether.* □

Proposition 7 (Unique match) *If precisely one possible* we know *statement from the algorithm in Figure 4.7 matches, then the outcome is deterministic.*

Proof 8 *Call the matching transition t_Y, then $T_Y = \{t\}$ and post is a $\vee$ summation over precisely one term, namely $post(t_Y, x)$, so $post = post(t_Y, x)$. This is precisely one term, so the outcome (the postcondition of the actual event that triggers the transition) is deterministic.* $\square$

Proposition 8 (Non-deterministic match) *If more than one possible* we know *statement generated by the algorithm of Figure 4.7 matches, then the post-condition selects non-deterministically between the post-conditions of the matched transitions.*

Proof 9 *If multiple transitions match, then $T_Y = \{t_1, t_2, \ldots\}$ and so $post = post(t_1, x) \vee post(t_2, x) \ldots$ with $t_1 \neq t_2$. The $\vee$ operator makes a choice between the alternatives that cannot be influenced. This lack of influence in the selection of a transition represents a non-deterministic match.* $\square$

4.7.11 Towards a prover

The sections above define (or, in part, postulate) the structures involved in the semantics of life cycles and their underlying mathematical language, William. We will now explain how these definitions are used to construct a prover for William.

Form of proofs. The general form of a proof derives one William formula from several others, for a given Kripke structure $\mathcal{M}$. The different formulæ may start in different snapshots because William is a temporal logic. A proof therefore looks like this:

$$\frac{\mathcal{M}, Q_1 \models p_1 \qquad \mathcal{M}, Q_2 \models p_2 \qquad \ldots}{\mathcal{M}, Q \models q}$$

Proofs in theory. The theoretic approach to these proofs is to exploit the equivalence from Section 4.6.3 to reduce a William proof to a proof in the μ-calculus:

$$\begin{array}{ll} & \mathcal{M}, Q_1 \models p_1 \text{ and } \mathcal{M}, Q_2 \models p_2 \ldots \\ \text{iff} & \mathcal{D}(\mathcal{M}), \mathcal{R}(\mathcal{M}) \models_\mu [\![p_1]\!](Q_1) \text{ and } \mathcal{D}(\mathcal{M}), \mathcal{R}(\mathcal{M}) \models_\mu [\![p_2]\!](Q_2) \ldots \\ \text{if} & \mathcal{D}(\mathcal{M}), \mathcal{R}(\mathcal{M}) \models_\mu [\![q]\!](Q) \\ \text{iff} & \mathcal{M}, Q \models q \end{array}$$

The step indicated with 'if' in the above proof is the place where the actual proof is carried out, in the well-known μ-calculus theory.

Proof algorithm. The proof algorithm proves from a 'knowledge base' of facts p_i that a desirable property q holds, or formally $\wedge_i p_i \Rightarrow q$, where all p_i and q are William expressions. The prover approaches this problem by searching for *candidate counter-examples* of this implication; when indecisive, a counter-example will be assumed, to make the prover conservative.

Counter-examples are examples of the inverse statement $\neg((\wedge_i p_i) \Rightarrow q)$, which equals $(\wedge_i p_i) \wedge \neg q$. If no inconsistency can be found in this expression, then any example of that expression must be considered as a candidate counter-example.

Generating feedback. For practical reasons, the prover aims at generating feedback, and does so by printing out candidate counter-examples. Basically, what it tries to do is construct examples of $\mathcal{M}, Q$ in

$$\mathcal{M}, Q \models (\wedge_i p_i) \wedge \neg q$$

Algorithmically, the problem is approached by generating models for all p_i and for $\neg q$, and 'merging' these to a model that makes all these formulæ *true*. Some executions will only work if an inconsistency is *true*; those executions are removed from the resulting model. If no executions remain after this pruning, then q must be *true* whenever all p_i are *true*.

It is not a requirement of a prover to generate *all* candidate counter-examples; as long as at least one is generated if some exist. This means that the model $\mathcal{M}$ need not be a complete (infinite) Kripke structure, but merely a model that enables the generation of one or more counter-examples.

4.8 Supportive of components

Life cycle diagrams were designed from a software engineering perspective, and one of the claims is that they support precompiled *components*. An interesting distinction can be made between *open systems* and *closed systems* — the former being systems that can still be extended by more components (components at *deployment time*, to use a popular phrase), the latter representing a system that is complete (operating at *runtime* in the same popular phrasing).

We can define the difference between an open and a closed system in terms of framing — a closed system is an open system with framing properties added. This principle can easily be applied to life cycles. The translation process treats states orthogonally, so treating whole life cycle diagrams orthogonally is trivial. Each life cycle is translated to a list of we know statements, to be composed with conjunction. Life cycles in turn are also composed with conjunction. The framing that closes a system simply adds and that is all we know.

Interestingly, every life cycle state is already to some degree closed. For every transition, the set of variables that change are known, and all others are known not to change, as has been explained in Subsection 4.7.7. However, this only applies to life cycle variables declared in the life cycle being translated. For instances reached over links, there is no such knowledge, and it takes a smart tool to gain an overview of a life cycle system to reason about such linked-to instances. Such global reasoning must often assume that nothing changes *except* what has been explicitly specified to change: the framing assumption. So, closing a system allows a global reasoning system to make additional assumptions about a specification in comparison to an open system. We will see this point back in the next chapter, when we present a prover for life cycle systems.

Please note the usefulness in practice of conjunction as a compositional operator. Conjunction is associative, commutative and idempotent, with the following advantages:

- *Associativity of composition* allows an arbitrary hierarchy of splitting components into sub-components. An architect is given total freedom to structure the system.

- *Commutativity of composition* allows compositions of components in any order. This allows total freedom for business units to purchase a component, add some new life cycles, and resell the new component, without worrying whether certain sale/resale chains are technically possible.

- *Idempotency of composition* allows the use of the same component multiple times. This allows separately developed components to include sub-components at will, without worrying about other components including the same sub-component. The addition of a sub-component incorporates new knowledge into a component, and can therefore be helpful towards a correctness proof.
- *Allowing redundancy in composition*, a property related to idempotency of composition, allows the expression of a constraint multiple times. This enables co-operation across boundaries of trust, because double-checking each other's actions is not a problem.

4.9 Turing completeness of life cycles

This section demonstrates the Turing completeness of life cycles by translating Turing programs to life cycles. According to the Turing thesis, Turing machine programs can represent all computable programs, and by translating Turing machine programs to life cycles, it follows that life cycles can also express all computable programs.

This section first describes the Turing machine and its programs, and then translates the programs to a life cycle model.

Turing machine. The Turing machine to be translated to life cycles consists of a tape with cells, each of which may either have a zero or one value. The machine has a 'head' pointing to a 'current position' on the tape, and can test the value at the head. The head can be moved left or right. Initially, the tape comprises of only one cell, but moving the head beyond the last cell adds a new cell to that side. New cells are always created with zero value. Cells under the head can be removed only if they are at one end of the tape, and the head will be moved to the (only) neighbour of the removed cell.

A program for this Turing machine takes the shape of a sequence of lines. Lines start with a label, then a colon, and then one of three possible forms; the complete line has one of the following forms:

l: o ; goto l'
l: if t then goto l' else goto l''
l: halt

The first form executes an operation o and continues on the line starting with l', which should exist. All possible operations o of this Turing machine are summarised in Figure 4.9. The second form performs a test t and continues on the line starting with l' if the test succeeds, or on the line starting with l'' if it fails; both labelled lines should exist. All possible tests t of this Turing machine are summarised in Figure 4.10. The third form is used to indicate the termination of a program. There should be precisely one line starting with l for every possible label l. There should be precisely one line starting with start which acts as the start of the program.

Operation o	**Meaning**
zero	Set cell under head to zero value.
one	Set cell under head to one value.
left	Move the head one position to the left.
right	Move the head one position to the right.
remove	If at one end of the tape, remove it and move head.

Figure 4.9: Operations supported by the Turing machine.

Test t	**Meaning**
atLeft	Test if the head is the leftmost position of the tape.
atRight	Test if the head is the rightmost position of the tape.
isZero	Test if the head points to a cell of zero value.
isOne	Test if the head points to a cell of one value.

Figure 4.10: Tests supported by the Turing machine.

Translation. The translation of a Turing machine program to life cycles consists of two parts, namely a compiler scheme and a ‘library’ of predefined life cycles. The predefined life cycles describe the behaviour of single cells and of neighbouring relationships between cells. See Figure 4.11 for these two life cycles. The Cell life cycle represents a single cell on the tape, the Neighbours life cycle represents the neighbouring relationship between two such cells.

The Turing machine program is translated to another life cycle that communicates with the standard ones in Figure 4.11. Let’s turn to each form of line and translate it.

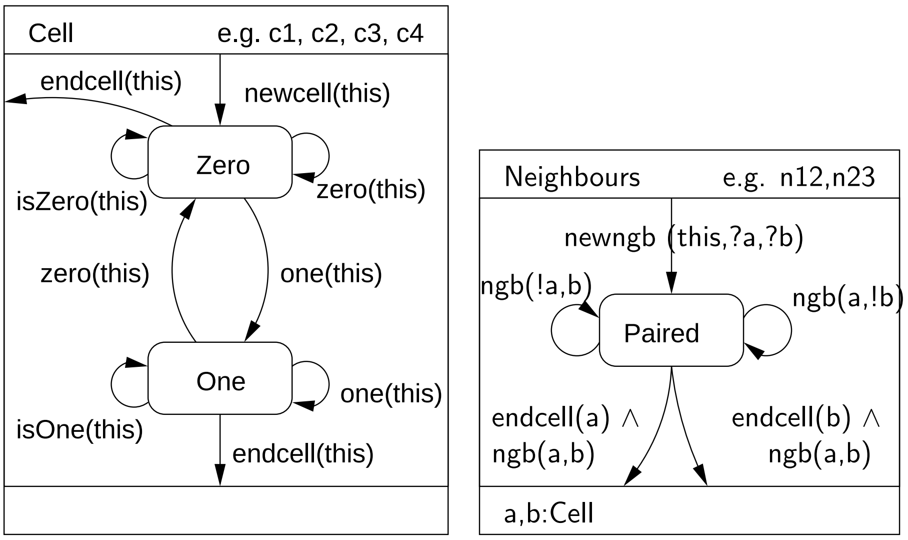


Figure 4.11: Standard life cycles supporting translated Turing machines.

The form l:o;goto l' translates to a state l, from which a translation to state l' departs with action $\mathbf{O}[\![o]\!]$ on it. If multiple definitions apply, multiple such transitions are to be generated. The form l:if t then goto l' else goto l'' translates to a state l with two transitions, one with action $\mathbf{P}[\![t]\!]$ to state l', and another with action $\mathbf{N}[\![t]\!]$ to state l''. The form l:halt translates to a state l from which a transition to Nirvana departs with event stop(this,l,h,r) on it. Finally, there is a transition from Nirvana to state start with action start(this), newcell(?h), newcell(?l), newcell(?r). The Turing machine variables are used as follows: h for the head cell, l for the leftmost cell and r for the rightmost cell; the effect of observing the same parameter on the same actual event causes them to be equal after this initialising transition. The Turing machine furthermore uses an operation nop(this) that is ignored by the other life cycles, and ngb(a,b) to confirm that cell a is the left side neighbour of cell b.

The translation schemes mentioned above are defined as follows; note that the first two forms of remove have their guard conditions only for clarity; life cycle semantics dictate precisely those anyway:

$\mathbf{O}[\![\text{zero}]\!]$	$\rightarrow$	zero(h)
$\mathbf{O}[\![\text{one}]\!]$	$\rightarrow$	one(h)
$\mathbf{O}[\![\text{left}]\!]$	$\rightarrow$	newcell(?l), newngb(?n,?l,l), newcell(?h) [h=l]

$\mathbf{O}[\![\mathsf{left}]\!]$	$\rightarrow$	ngb(?h,h) [h$\neq$l]
$\mathbf{O}[\![\mathsf{right}]\!]$	$\rightarrow$	newcell(?r), newngb(?n,r,?r), newcell(?h) [h=r]
$\mathbf{O}[\![\mathsf{right}]\!]$	$\rightarrow$	ngb(h,?h) [h$\neq$r]
$\mathbf{O}[\![\mathsf{remove}]\!]$	$\rightarrow$	endcell(h), ngb(h,?h) [h=l AND h$\neq$r]
$\mathbf{O}[\![\mathsf{remove}]\!]$	$\rightarrow$	endcell(h), ngb(?h,h) [h$\neq$l AND h=r]
$\mathbf{O}[\![\mathsf{remove}]\!]$	$\rightarrow$	nop(this) [h$\neq$l AND h$\neq$r]

$\mathbf{P}[\![\mathsf{atLeft}]\!]$	$\rightarrow$	nop(this) [h=l]
$\mathbf{N}[\![\mathsf{atLeft}]\!]$	$\rightarrow$	nop(this) [h$\neq$l]
$\mathbf{P}[\![\mathsf{atRight}]\!]$	$\rightarrow$	nop(this) [h=r]
$\mathbf{N}[\![\mathsf{atRight}]\!]$	$\rightarrow$	nop(this) [h$\neq$r]
$\mathbf{P}[\![\mathsf{isZero}]\!]$	$\rightarrow$	isZero(h)
$\mathbf{N}[\![\mathsf{isZero}]\!]$	$\rightarrow$	isOne(h)
$\mathbf{P}[\![\mathsf{isOne}]\!]$	$\rightarrow$	isOne(h)
$\mathbf{N}[\![\mathsf{isOne}]\!]$	$\rightarrow$	isZero(h)

Example. An example transition of a Turing machine may help to clarify the above definitions. Figure 4.12 shows a state with some cells at the right end of the tape, and the head at the rightmost position. The execution of a right operation creates a new cell on the tape, and positions the head on it. Figure 4.13 shows the same system as a static state diagram in UML notation. The left side is the situation before the transition, the right side is the situation after the transition.

The event on the transition is, according to the above definitions, newcell(?r), newngb(?n,r,?r), newcell(?h) because h=l is true in this situation. The occurrence of newcell means that *some* instance of Cell is created and gives out its identity, referred to as c4 in the diagram. The parameter ?r in newcell(?r) indicates that this identity is the new value for variable r in the Turing machine myTM. Since the same actual event is also observed by newcell(?h), this same identity is also stored in variable h of myTM. Finally, the occurrence of newngb(?n,r,?r) indicates that a new neighbour is instantiated (and its identity is stored in n but never used) and that it should respond to future ngb events by coupling the old value of r to the new one, ?r in the event specification. The result is given in Figure 4.13.

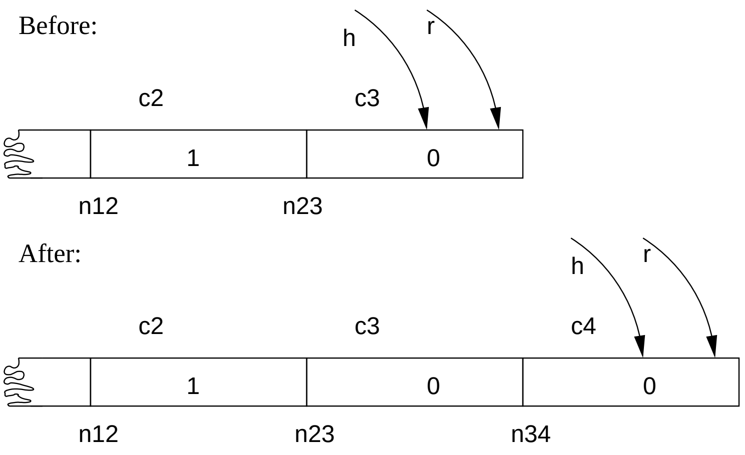


Figure 4.12: A right transition by a Turing machine.

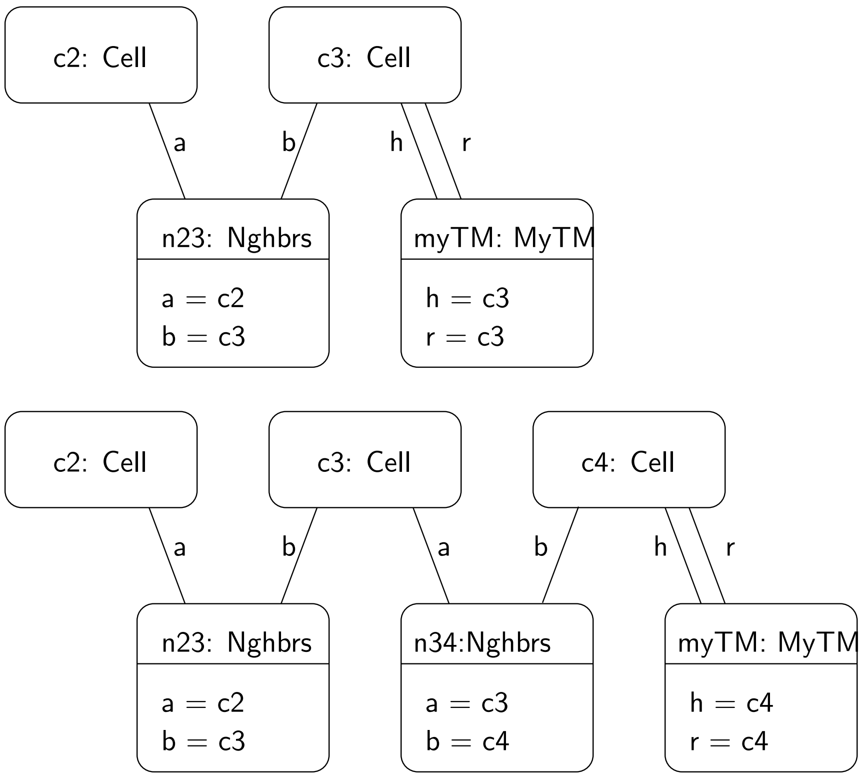


Figure 4.13: A right transition represented in instances, before/after.

Chapter 5

Conducting proofs

There is no why for a rose,
it blooms because it blooms.

Angelus Silesius

Up to here, this thesis has presented a process specification language that integrates a mathematical theory with practical notation. Although interesting in itself, this integration merely serves as appetiser. The process verification approach described in this chapter forms the main course of this thesis. In contrast with common approaches to model checking, this approach can handle a variety of infinite structures in the models. In contrast with theorem proving, this approach is automatic, and can give feedback in terms of the concepts in the designer's mind.

A prover tool must have some internal representation of the processes that it verifies. It is possible to represent either all correct executions or all executions that prove processes wrong, and since we are looking for feedback, we prefer to represent incorrect executions. This approach is also taken in Spin [Hol97] and many similar tools. In this approach, the correctness of a temporal expression E is validated by constructing all possible executions of $\neg E$, and if E is not proven wrong by any execution, then $\neg E$ will not construct anything.

The process representation in our prover approach must support a delicate combination of theorem proving and model checking strategies. Although our work is based on μ-calculus, which is commonly used in model checking applications [BCM$^+$90], we do not make the common assumption underlying model checkers, that the domain is finite. We allow the quantifiers that bind Bill variables to range over unbounded sets of instances (of life cycles). This degree of freedom makes it undoable to unroll all possible executions, and so a model checking approach will not suffice. On the other hand, proving everything with theorem proving techniques makes it less likely to find solutions to problems; general theorem provers lack the domain knowledge that allows model checkers to infer more solutions and to provide practically usable feedback.

For these reasons, we aim at a prover which behaves like a model checker where it can, and like a theorem prover where it must. This approach is intended to retain the benefits of model checkers, and add the additional abilities of an automatic theorem prover to extend the scope of solvable problems.

Unfortunately, this choice also means that existing software is unusable. Model checker software has built-in assumptions about finiteness of its domain, and checking all the code for these implicit assumptions seems much more error-prone than constructing a new prover. The exploitation of a theorem prover in such a way that it represents and manipulates knowledge at the application domain level seems even harder, and is considered out of our reach. The only solution we found was to design our own tool, specifically for our application domain. Unfortunately, this path has introduced problems that make it impossible to present an implemention alongside this thesis.

Model checker aspects. Model checkers usually construct a *state space*, where every state represents a possible *snapshot*, or system-wide state, during a possible execution. It is not uncommon [Hol97] to represent executions that possibly prove a property wrong, rather than the ones for which it is correct, in order to construct feedback. The introduction and Chapter 2 of this thesis also states the importance of feedback; concrete examples of wrong executions seem to be good forms of feedback [Rei99].

Our prover approach therefore constructs an internal representation of wrong executions, but it classifies them to avoid explosion of the state space, especially because this explosion becomes unbounded in Bill's unbounded domain. Recall that Bill allows literals based on the states of variables, and on the equality of variables; hence, the states will be distinguished based on equality of instances, and their states. So, as far as variable values are concerned, a state where $x = \mathsf{Tarzan}$, $y = \mathsf{Jane}$, $z = \mathsf{Tarzan}$ is considered equivalent to a state where $x = \mathsf{Rick}$, $y = \mathsf{Roelof}$, $z = \mathsf{Rick}$, because in both cases, $x \neq y \wedge x = z$.

To optimise this scheme of equivalence classes (or partitions) of variable values, the prover will allow partial knowledge about the partitioning of variables, and split into alternatives when such need arises. This is very much like partial order optimisations [God96].

As a further refinement, possible states of each Bill variable are tracked. The total knowledge available in a snapshot is a collection of equations and inequations between variables, and possible states for each variable. We shall present a structure model for this information, so that it is possible to directly infer that $x@S \wedge y@T \wedge x{=}y$ is inconsistent for different S and T, and that the process assuming this is not a possible counter-example. This model is an abstraction of the snapshots in the extended Kripke structures presented in Section 4.6.3 in the previous chapter.

Theorem prover aspects. When the aforementioned abstraction of snapshots is applied to a classical model checker problem, it is a mere optimisation, and therefore an optional part of a model checker. But for life cycle checking, these classifications are a necessity, because it allows unbounded numbers of instances to be represented with a bounded model. The ability to handle unbounded instantiation allows the introduction of quantifiers that range over unbounded sets (of instances of life cycles). The rules involved in proving such quantifiers correct are theorem prover rules.

Rather than unfolding quantifiers for all possible instances, our internal model represents quantifiers in its internal structure. This keeps a model bounded, but it also requires some techniques from theorem proving. These techniques help to decide when a quantified formula is *false*: An expression $\exists x{:}L \cdot p$ is *false* when p does not allow any value for x, and an expression $\forall x{:}L \cdot p$ is *false* when only one possible value for x is disallowed by p.

Outline of the prover algorithm. The algorithm for our prover approach sets up an initial structure with candidate counter-examples, each of which represents a possibly wrong execution, and then applies axioms (life cycle inferred knowledge and explicitly made assumptions) in the hope to remove counter-examples. If any candidate counter-examples remain after applying all axioms, then the prover will report that it cannot falsify all thinkable counter-examples, and offer the remaining ones as feedback to the life cycle designer.

Every Bill expression is rewritten to a tree-form, and the trees will be merged, along the lines of the algorithm in Figure 5.1. In this algorithm, bill2tree changes a Bill expression into a tree representation, mark adds a flag 'this is part of a counter-example' to every tree node, merge determines a sort of conjugation of an axiom in tree representation with the tree representation of candidate counter-examples, and prune cuts off the branches of the tree that represent execution alternatives that are known not to be counter-examples (anymore), usually because they evaluate to *false*. Although the treatment of the tree structures is based on the mathematical properties of Bill and thus on standard μ-calculus, it is worthwhile to describe these tree manipulations in the remainder of this chapter, because the structures maintained internally adhere to more constraining invariants than Bill expressions, and algorithms updating the trees demonstrate how these constraints can be maintained.

```
# Return candidate counter-examples against axioms $\Rightarrow$ prop
def prove (prop, axioms):
   countex $\leftarrow$ prune (mark (bill2tree ($\neg$prop)))
   for ax $\in$ axioms:
      countex $\leftarrow$ prune (merge (countex, bill2tree (ax)))
   return countex
```

Figure 5.1: High-level algorithm of our prover approach.

Finiteness of proofs. Not all proofs can be conducted in finite time, and it is important to terminate infinite proofs in time. The situation in which infinity is introduced, is when a quantified statement like $\forall \ldots \exists \ldots$ is applied to the existential quantification contained in itself, either directly or indirectly, because that causes a cycle of quantifiers introductions; to the best of our knowledge, this is the only form of infinity inherent in theorem provers.

Theorem provers commonly solve this problem by restricting the number of introductions of quantifiers. When this is implemented as a constraint on allowable logical constructs, it may lead to results that a designer, thinking in terms of life cycles, finds unexpectable. It is better to restrict a concept in that designer's mind to a maximum, as long as it has the desired effect of forced-finite proofs.

Where a Bill variable quantified in $\forall x{:}L$ which has a link $x.l{:}L$, the infinity can be broken in life cycle terms by restricting the maximum number of traversals of the same link. For other repetitive quantifier introductions we see no such life cycle related concept, meaning that we must resolve to the purely logical constraints of a maximum number of introductions.

5.1 Properties of Bill

This section works toward a canonical form of Bill expressions in their tree representation, in search for a format that makes algorithmic manipulation of the structures simpler.

Unrolling loops. The desired Bill tree structure is approached by pushing the μ and ν operators 'down', so the loops that are expressed by these constructs must be unrolled. The idea is that the operators $\vee$ and $\wedge$ are distributed 'outward' through these loop bindings.

Quantifiers over Bill variables can be pulled out of loop bindings according to the following equations:

Proposition 9

$$\begin{aligned}
\mu I \cdot \forall x{:}L \cdot E_I &= \forall x{:}L \cdot E_{\mu I \cdot \forall x:L \cdot E_I} && (5.1)\\
\mu I \cdot \exists x{:}L \cdot E_I &= \exists x{:}L \cdot E_{\mu I \cdot \exists x:L \cdot E_I} && (5.2)\\
\nu I \cdot \forall x{:}L \cdot E_I &= \forall x{:}L \cdot E_{\nu I \cdot \forall x:L \cdot E_I} && (5.3)\\
\nu I \cdot \exists x{:}L \cdot E_I &= \exists x{:}L \cdot E_{\nu I \cdot \exists x:L \cdot E_I} && (5.4)
\end{aligned}$$

Proof 10 *The right hand side of equation 5.1 is obtained by unrolling loop I in the left hand side using the standard law for μ-expressions that states*

$$\mu I \cdot p_I = p_{\mu I \cdot p_I}$$

The proof of equations 5.2, 5.3 and 5.4 is similar. □

Logical operators $\wedge$ and $\vee$ can be 'pushed out' of loops as follows:

Proposition 10

$$\mu I \cdot p_I \wedge q_I = \mu J \cdot p_{J \wedge \mu K \cdot q_{J \wedge K}} \wedge \mu K \cdot q_{\mu J \cdot p_{J \wedge K} \wedge K} \tag{5.5}$$
$$\mu I \cdot p_I \vee q_I = \mu J \cdot p_{J \vee \mu K \cdot q_{J \vee K}} \vee \mu K \cdot q_{\mu J \cdot p_{J \vee K} \vee K} \tag{5.6}$$
$$\nu I \cdot p_I \wedge q_I = \nu J \cdot p_{J \wedge \nu K \cdot q_{J \wedge K}} \wedge \nu K \cdot q_{\nu J \cdot p_{J \wedge K} \wedge K} \tag{5.7}$$
$$\nu I \cdot p_I \vee q_I = \nu J \cdot p_{J \vee \nu K \cdot q_{J \vee K}} \vee \nu K \cdot q_{\nu J \cdot p_{J \vee K} \vee K} \tag{5.8}$$

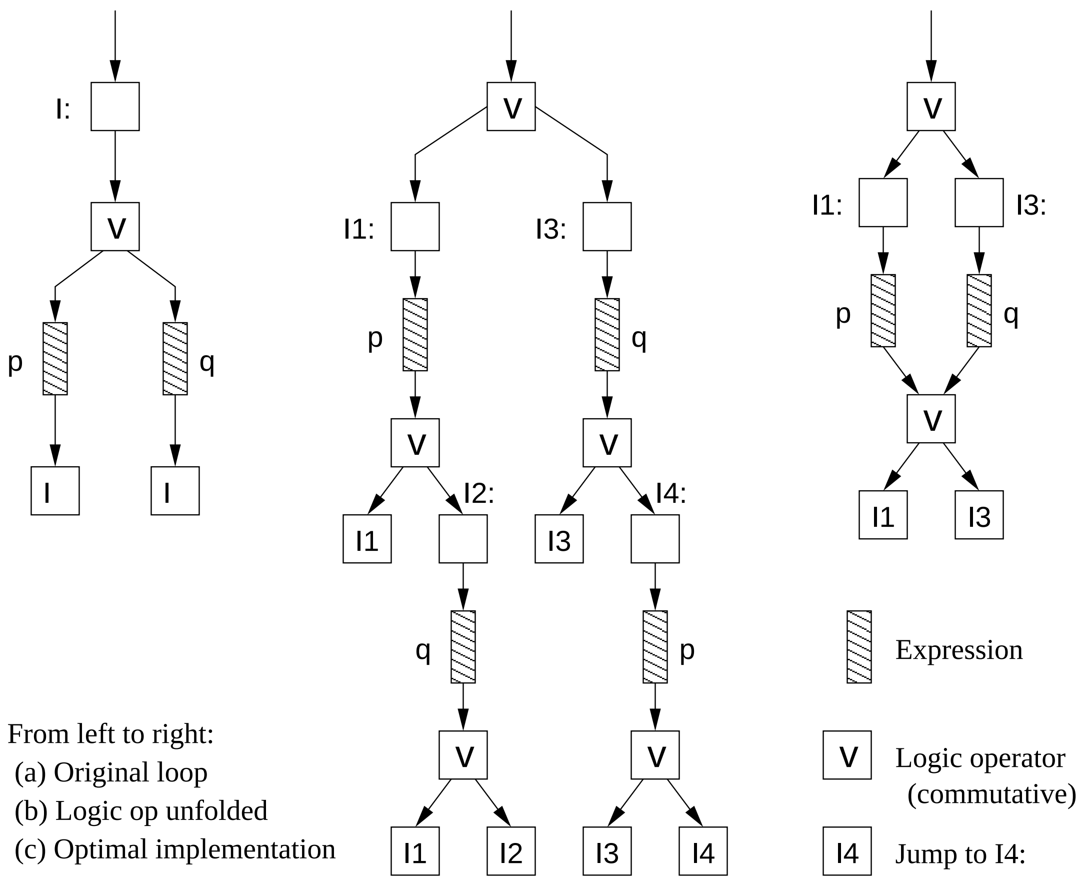


Figure 5.2: Unrolling loops to push logic operators out.

Proof 11 (Sketch) *Figure 5.2 demonstrates this graphically; the (a) and (b) parts are the left and right hand sides of the equations. In an imple-*

mentation, where scoping rules for structural references (or 'pointers') are usually more relaxed, the left hand forms can be represented with better optimisation, as in Figure 5.2(c).

5.2 Representing Bill expressions

Consider the language Bill, extended with *true* and *false* and with the customary logic complements (existential quantification over steps and Bill variables and the $\vee$ operator) as they are defined for William. This language is clearly equivalent with Bill.

This slightly extended syntax is more usable from a prover's perspective. The addition of *true* and *false* is often useful in automated applications, and the addition of logic complements makes it possible to remove negation on anything but the basic propositions.

The removal of negation on composite constructs simplifies the prover. A prover cannot always infer with certainty whether a complex proposition holds or not — and it must make a safe assumption in such cases. The assumption that is safest is the one that does not relinquish candidate counter-examples. If the sign of a formula flips due to a negation, the safe assumption also flips, which would complicate the prover. To avoid this, it is useful to have internal prover structures without negations, except on the basic propositions.

Figure 5.3 shows a number of possibilities for distributing operators (listed on top of the columns) outward over other operators (listed at the left of the rows). The annotation $Log : prover.tex, v$ The annotation Revision 1.2 2002/02/25 12:34:58 vanrein The annotation Thesis as sent to committee The annotation is used to refer to distribution laws from common logic, *Cmb* is used to denote that some combined/altered form of the operators exists according to common logic, and the numbers are references to formulæ in this thesis. The open boxes indicate that the distributive law is not interesting for the purposes of this chapter, regardless of whether it exists.

The normalisation of Bill expressions now proceeds as follows:

1. **Inversion:** The row for $\neg$ demonstrates that this operator can be pushed into any of the given operators; when this is done, every $\neg$ ends up in front of an elementary expression $x{=}y$ or $x@S$. A loop back-reference will never be inverted because of the monotonicity constraint on Bill and μ-calculus expressions. We will refer to possibly

	$\wedge$	$\vee$	$\neg$	$\forall[\_]\ \_$	$\exists[\_]\ \_$	$\forall\_ : \_ \cdot \_$	$\exists\_ : \_ \cdot \_$	$\mu\_ \cdot \_$	$\nu\_ \cdot \_$
$\wedge$	*Cmb*	*Log*				*Log*	*Log*		
$\vee$		*Cmb*				*Log*	*Log*		
$\neg$	*Log*	*Log*	*Cmb*	(4.2)	(4.2)	*Log*	*Log*	*Log*	*Log*
$\forall[\_]\ \_$									
$\exists[\_]\ \_$									
$\forall\_ : \_ \cdot \_$									
$\exists\_ : \_ \cdot \_$									
$\mu\_ \cdot \_$	(5.5)	(5.6)				(5.1)	(5.2)		
$\nu\_ \cdot \_$	(5.7)	(5.8)				(5.3)	(5.4)		

In the above, *Log* means "according to the laws of logic" and *Cmb* means "combined according to the laws of logic".

Figure 5.3: Distributing row-operators through column-operators.

inverted elementary expressions as *Literals* in what follows.

2. **Loop unrolling:** By unrolling loops, it is possible to 'push loop operators into' expressions, until they are positioned immediately in front of action quantifiers. The rows for the loop operators in Figure 5.3 contain just the right distribution laws to make this possible. Whenever the form $\mu I \cdot I$ or $\nu I \cdot I$ turns up, it can be replaced with *false* or *true*, respectively; this is also possible with forms such as $\mu I \cdot \nu J \cdot \nu K \cdot I$.

3. **Variable scope enlargement:** Quantifiers that bind Bill variables can now be 'pushed out' of an expression until they group together as immediate sub-expressions of action quantifications. This is possible because they distribute through the only remaining operators underway, $\vee$ and $\wedge$. We assume an implicit renaming of variables where this is needed to avoid name clashes.

4. **And/or:** The operators $\vee$ and $\wedge$ are now the only ones remaining that are not ordered. It is possible to distribute $\wedge$ operators over $\vee$ operators to get the latter 'on the outside'.

The $\wedge$-separated list of Bill expressions gives a fair chance of finding an inconsistency in the counter-example that it represents; the $\vee$-separated list of Bill expressions makes it possible to describe any number of possible counter-examples.

These steps will be referred to as the normalise function in what follows.

$$\begin{array}{lcl}
\mathit{BillExpr} & ::= & \forall\, \mathit{Varnm} : \mathit{Lifecyclenm} \cdot \mathit{BillExpr} \\
 & | & \exists\, \mathit{Varnm} : \mathit{Lifecyclenm} \cdot \mathit{BillExpr} \\
 & | & \mathit{Alternative} \\
\mathit{Alternative} & ::= & \mathit{Alternative} \vee \mathit{Alternative} \\
 & | & \mathit{Constraint} \\
\mathit{Constraint} & ::= & \mathit{Constraint} \wedge \mathit{Constraint} \\
 & | & \mathit{Literal} \\
 & | & \mathit{Loophd} \\
 & | & \mathit{Loopnm} \\
\mathit{Loophd} & ::= & \mu\, \mathit{Loopnm} \cdot \mathit{Loophd} \\
 & | & \nu\, \mathit{Loopnm} \cdot \mathit{Loophd} \\
 & | & \mathit{Action} \\
\mathit{Action} & ::= & \forall[\ldots]\, \mathit{BillExpr} \\
 & | & \exists[\ldots]\, \mathit{BillExpr} \\
\mathit{Literal} & ::= & \, \mathit{Varnm} = \mathit{Varnm} \\
 & | & \neg\, \mathit{Varnm} = \mathit{Varnm} \\
 & | & \, \mathit{Varnm} \mathbin{@} \mathit{Statenm} \\
 & | & \neg\, \mathit{Varnm} \mathbin{@} \mathit{Statenm} \\
 & | & \mathit{true} \\
 & | & \mathit{false}
\end{array}$$

Figure 5.4: BNF syntax of the prover's representation of Bill expressions.

Proposition 11 (Bill normalisation) *After the* normalise *function has been applied to an arbitrary Bill expression, it conforms to the syntax in Figure 5.4.*

Proof 12 *We shall consider an arbitrary path from top to a tip of the syntax tree. The actual tree manipulations are assumed to be an efficient way to deal with overlapping parts of many such paths at once. We will describe the path as a regular expression of a few syntactical classes.*

Let syntactical class A represent a quantified action such as $\forall[e(x)]$; let V represent a variable quantifier prefix such as $\forall x{:}L\cdot$; let L represent a loop declaration such as $\mu I\cdot$; let R represent a loop reference such as L; let N represent a negation operator $\neg$; let C represent the selection of an operand from a conjunction; let D represent the selection of an operand from a disjunction; let B represent a basic expression such as $x@S$; and, let T denote either true *or* false*.*
A path over the syntax tree of a Bill expression that serves as input to normalise *has the format $(A|V|L|N|C|D)^*\cdot(T|R|B)$.*
After the first step, this syntax is reduced to $(A|V|L|C|D)^*\cdot N^*\cdot(T|R|B)$ which can be simplified to $(A|V|L|C|D)^*\cdot N^?\cdot(T|R|B)$ which, due to monotonicity of loops, is equivalent to $(A|V|L|C|D)^*\cdot(T|R|N^?\cdot B)$.
After the second step, this syntax is reduced to $(V|C|D)^*\cdot L^*\cdot(A\cdot(V|C|D)^*\cdot L^*)^*\cdot(T|R|N^?\cdot B)$ which we prefer to write as $(V|C|D)^*\cdot(L^*\cdot A\cdot(V|C|D)^*)^*\cdot L^*\cdot(T|R|N^?\cdot B)$, or $(V|C|D)^*\cdot(L^*\cdot A\cdot(V|C|D)^*)^*\cdot(L^*\cdot T|L^*\cdot R|L^*\cdot N^?\cdot B)$. In the latter form, $L^*\cdot R$ can simply be reduced to false *when $\mu R\cdot$ is one of the L, or to* true *if $\nu R\cdot$ is one of the L, or to R if neither of these two is true. Furthermore, $L^*\cdot N^?\cdot B$ can be reduced to $N^?\cdot B$ since the loop names in the L are never referenced, and $L^*\cdot T$ can similarly be reduced to T. As a result, the second step yields the syntax $(V|C|D)^*\cdot(L^*\cdot A\cdot(V|C|D)^*)^*\cdot(T|R|N^?\cdot B)$.*
After the third step, this syntax is reduced to $V^*\cdot(C|D)^*\cdot(L^*\cdot A\cdot V^*\cdot(C|D)^*)^*\cdot(T|R|N^?\cdot B)$.
After the fourth step, this syntax is reduced to $V^*\cdot D^*\cdot C^*\cdot(L^*\cdot A\cdot V^*\cdot D^*\cdot C^*)^*\cdot(T|R|N^?\cdot B)$, and since (1) nested D can be seen as one D and (2) no D can be seen as one D, and similarly for C, this syntax can be rewritten as $V^*\cdot D\cdot C\cdot(L^*\cdot A\cdot V^*\cdot D\cdot C)^*\cdot(T|R|N^?\cdot B)$.
This follows the syntax of BillExpr in Figure 5.4. □

In the syntax of Figure 5.4, please note that a *BillExpr* can be just an *Alternative*; that an *Alternative* can be just a *Constraint*; that a *Constraint* can be just a *Loophd*; and that a *Loophd* can be just an *Action*. The only element that necessarily introduces a new element in the syntax, and introduces precisely one element, is an *Action*. This makes transitions the basis of the internal prover model, and the rest can be seen as 'lumped around' the transition as optional extensions. Without these lumps, the model would look much like a traditional model checker's internal representation. The extra lumps make it possible to express generalisations that require theorem proving techniques.

These lumps also imply that the rules for merging and pruning the prover's data structures are more complicated than a traditional model checker. These operations on the data structures deserve detailed treatment, which is the goal of the remainder of this chapter. The remainder of this section shall be used to define the bill2tree algorithm in Figure 5.1. The syntax of Figure 5.4 can be poured into a data structure that can be manipulated efficiently by an algorithm.

Data structures. The data structure representing a *Constraint* is a LiteralLogic, representing all the *Literal* terms, in conjunction with a set of the *Loophd* expressions. A LiteralLogic represents knowledge about Bill variables, but it is not necessarily complete knowledge. The knowledge derivable from a *Constraint* such as $x \neq y \wedge y \neq z$ does not specify whether $x = z$ or not. In general, it is therefore necessary to represent *partial knowledge* in a LiteralLogic structure.

A complete representation of equality information would define for every pair of Bill variables whether they represent the same instance, or different ones, and it would define a function mapping each instance to a state. A partial representation of this knowledge is a bit more loose. Partial equality knowledge can be represented as a partition of Bill variables. Two Bill variables occurring in the same partition imply that they represent the same instance; every Bill variable in scope occurs in at most one partition. The instances represented by the Bill variables in different partitions are *possibly* unequal.

In addition to this partition, there is a list of pairs of Bill variables in scope, and each pair indicates a *certain* inequality relation between the variables. Furthermore, partial knowledge about the current state of a Bill variable in scope is represented with a function from a Bill variable to a *set* of possible states.

The data structure LiteralLogic represents the *Literal* knowledge contained in a *Constraint*, and it is defined as follows:

Definition 15 (Literal logic)

$$\mathit{LiteralLogic} \equiv \underbrace{\boldsymbol{P}\boldsymbol{P}\mathrm{Varnm}}_{\textit{equalities}} \times \underbrace{\boldsymbol{P}(\mathrm{Varnm} \times \mathrm{Varnm})}_{\textit{inequalities}} \times \underbrace{\mathrm{Varnm} \rightarrow \boldsymbol{P}\mathrm{Statenm}}_{\textit{possible states}}$$

The interpretation of a LiteralLogic structure is that it poses certain constraints on a Kripke model in a certain snapshot. If $\langle E, I, S\rangle$: LiteralLogic *in snapshot Q of Kripke model $\mathcal{M}$, then all of the following hold:*

$$\mathcal{M}, Q \models \bigwedge_{X \in E} \bigwedge_{x,y \in X} x = y \tag{5.9}$$

$$\mathcal{M}, Q \models \bigwedge_{\langle x,y \rangle \in I} x \neq y \tag{5.10}$$

$$\mathcal{M}, Q \models \bigwedge_{(x \mapsto Y) \in S} \bigvee_{T \in Y} x@T \tag{5.11}$$

As may be expected, this structure implies that the equations are closed under symmetry, reflexivity and transitivity, making this model for the equations an equivalence relation. Please note that the inequalities are *not* closed under symmetry; but that will not lead to problems in the prover.

Now assume that pointers are used to link a loop back-reference with the point where the loop is declared. Then the complete data structure of Bill expressions is as given in the class diagram in Figure 5.5.

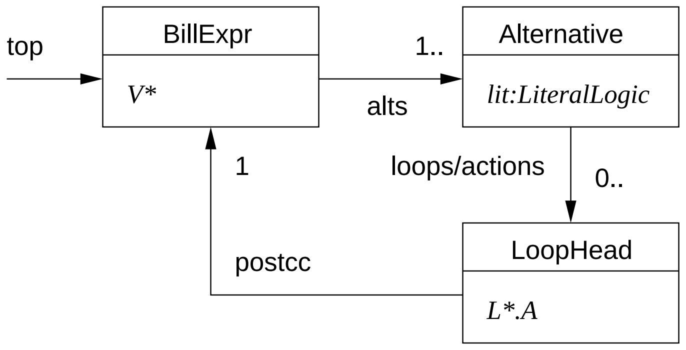


Figure 5.5: Classes used internally by the prover.

The function bill2tree that was assumed in the algorithm in Figure 5.1 basically builds a data structure like that. Although the structure given in Figure 5.5 is not really a tree because it suggests sharing common sub-expressions and because it assumes back-references, we will loosely refer to the internal representation of the prover as the 'tree' of counter-examples.

The prover ensures an important invariant, namely that terms in conjugations in Bill expression trees are always *closed under merging*; merging is defined in Section 5.4. This means that multiple Loophds under the same Alternative never need to be merged anymore by the prover. When we discuss the merge operation, we will see how this is established; it is part of the bill2tree operation but not treated here.

5.3 Pruning inconsistencies

Our prover approach is based on a tree of candidate counter-examples, and if any of these counter-examples can be proven inconsistent, it can be removed from the tree. This usually means pruning a branch off, hence the name prune for this action in Figure 5.1. Boldly removing any inconsistencies is possible because of the following theorem.

Proposition 12 (Black hole) *Once inconsistent, a tree representation with candidate counter-examples will nevermore become consistent in the prover algorithm of Figure 5.1.*

Proof 13 (Sketch) *The prover algorithm in Figure 5.1 has the intention to generate candidate counter-examples for* $\mathsf{axioms}[\,] \Rightarrow \mathsf{prop}$ *or, in other words, for* $\mathsf{prop} \vee \neg(\bigwedge_i \mathsf{axioms}[i])$*. The prover constructs candidate counter-examples by generating examples for the inverse property,* $(\neg\mathsf{prop}) \wedge (\bigwedge_i \mathsf{axioms}[i])$*. The initial* $\neg$ *is performed before the internal representation of the prover exists; therefore, the only operation of importance for construction of the counter-examples is the associative* $\wedge$ *operator. Inconsistencies are represented in logic as false, and since* $\mathit{false} \wedge x = \mathit{false}$*, a tree representing candidate counter-examples that is inconsistent will always remain inconsistent in the algorithm of Figure 5.1.* □

Every Bill expression can be expressed in the form of a BillExpr, including inconsistencies. The whole prover algorithm is aimed at finding inconsistencies, because that would eliminate candidate counter-examples; when all candidate counter-examples are pruned from the BillExpr, the property prop has been proven. The goal of the design of the BillExpr tree structure has therefore been to allow the detection of inconsistencies.

To be on the safe side, it is important not to discard candidate counter-examples in case of uncertainty. Therefore, the treatment below will introduce possible inconsistencies that can be contained in counter-examples, but the treatment will not attempt to be complete.

Proposition 13 (Self-inequality) *A* $\mathsf{LiteralLogic}\,\langle E, I, S\rangle$ *is inconsistent if, for some* $\langle x, y\rangle \in I$*, there is a* $X \in E$ *such that* $x \in X$ *and* $y \in X$*.*

Proof 14 *Trivial — but we present it as an example calculation. For the given variable names, we can infer the start of the following reasoning from (5.10) and (5.9):*

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathcal{M}, Q \models x \neq y \textit{ and } \mathcal{M}, Q \models x = y} \\
\textit{iff} & \mathcal{M}, Q & \models & (x \neq y) \wedge (x = y) \\
\textit{iff} & \mathcal{D}(\mathcal{M}), \mathcal{R}(\mathcal{M}) & \models_\mu & [\![(x \neq y) \wedge (x = y)]\!](Q) \\
\textit{iff} & \mathcal{D}(\mathcal{M}), \mathcal{R}(\mathcal{M}) & \models_\mu & [\![x \neq y]\!](Q) \wedge [\![x = y]\!](Q) \\
\textit{iff} & \mathcal{D}(\mathcal{M}), \mathcal{R}(\mathcal{M}) & \models_\mu & [\![\neg(x = y)]\!](Q) \wedge [\![x = y]\!](Q) \\
\textit{iff} & \mathcal{D}(\mathcal{M}), \mathcal{R}(\mathcal{M}) & \models_\mu & \neg [\![x = y]\!](Q) \wedge [\![x = y]\!](Q) \\
\textit{iff} & \mathcal{D}(\mathcal{M}), \mathcal{R}(\mathcal{M}) & \models_\mu & \neg(x = y) \wedge (x = y) \\
\textit{iff} & \mathcal{D}(\mathcal{M}), \mathcal{R}(\mathcal{M}) & \models_\mu & \textit{false}
\end{array}$$

□

In case there is no known equality or no known inequality between x and y, this form of inconsistency will not be noted. It may require another equation or inequation to be added (or ‘merged into’ the BillExpr) before an inconsistency can be derived. The updates to the LiteralLogic structure, notably to its set E of partitions, will be helpful to make this form of inconsistency easily derivable in that new situation.

Proposition 14 (No possible state) *A LiteralLogic $\langle E, I, S \rangle$ is inconsistent if $\emptyset \in Ran(S)$.*

Proof 15 *If $\emptyset \in Ran(S)$, then there is some v: Varnm for which $(v \mapsto \emptyset) \in S$. According to (5.11) this requires something inconsistent from a Kripke model $\mathcal{M}$ in snapshot Q:*

$$\mathcal{M}, Q \quad \models \quad \ldots \quad \wedge \quad \bigvee_{S \in \emptyset} v@S$$

Since the $\vee$ ranges over zero terms, its result is false, and so this constraint requires the model to make false hold — clearly an inconsistency. □

Assumption 1 (Bottom-up) *Pruning inconsistencies from the tree of counter-examples is a bottom-up operation.*

As a result of this assumption, the pruning process need not consider sub-expressions that are *false* while dealing with the expressions containing those.

Proposition 15 (Variable quantification) *While pruning, a quantification $\exists x{:}L \cdot \bigvee_{i\in 1..n} p_i$ or $\forall x{:}L \cdot \bigvee_{i\in 1..n} p_i$ is known to be inconsistent only when $n = 0$*

Proof 16 *We give the proof for existential quantification; it applies to universal quantification in the same way. The existential quantification over Bill variables is certainly false if the following certainly holds, regardless of the Snapshot Q described by the formulæ:*

$$\begin{array}{cll} & [\![\neg\exists x{:}L \cdot \bigvee_{i\in 1..n} p_i]\!](Q) & \\ = & [\![\forall x{:}L \cdot \bigwedge_{i\in 1..n} \neg p_i]\!](Q) & \\ = & \forall x \cdot \bigwedge_{i\in 1..n} \neg[\![p_i]\!](Q) & (\textit{where lifetp}(x) = L) \\ = & n = 0 \ \vee \ \forall i \in 1..n \cdot \neg[\![p_i]\!](Q) & (\textit{where lifetp}(x) = L) \end{array}$$

The set $\{x \mid \textit{lifetp}(x) = L\}$ is never empty for any Lifecyclenm L, because of the assumption made in Section 4.4.1.

Because of assumption 1, no $i{\in}1..n$ will make $\neg[\![p_i]\!](Q)$ hold, so the constraint $\forall i \in 1..n \cdot \neg[\![p_i]\!](Q)$ never holds. Therefore, the only remaining way for an existential quantification over Bill variables to be inconsistent, is when $n = 0$, meaning the set alts *of* Alternatives *is empty.* □

Proposition 16 (Universal variable quantification) *A universally quantified expression $p = \forall x{:}L \cdot q$ is inconsistent if any of the following causes it to be inconsistent:*

- *For some free variable y of p, with $x = y$;*
- *For some free variable y of p, with $x \neq y$;*

Proof 17 *If any of the bulleted items is inconsistent with q, then the variable x is not free to take on any identity it wishes; as a result, the quantification over L is not universal, so p is false.* □

Proposition 17 (Existential step quantification) *An existentially quantified step $\exists[E/F], p$ is inconsistent if p is inconsistent.*

Proof 18 *Trivial, by calculating the semantics of $\exists[E/F], \textit{false}$.* □

Idiom 1 (Universal step quantification) *The notation $\forall[E/F]\,\textit{false}$ implies that no actual event matching $[E/F]$ can occur at the moment in time described by this temporal formula.*

The following properties deal with unrolling of loops; they ensure that this will never become an infinite exercise.

Lemma 2 (Finite tree depth) *To represent all possible execution states, the depth of a tree of counter-examples need not become infinite.*

Proof 19 (Sketch) *The number of different Snapshots is at most the number of different* LiteralLogic *values on the known Bill variables. Since these variables are declared in the finite syntactical context of an expression, this number of variations is finite. if a loop is unrolled this number of times, there must have been a point which has occurred before.* □

Lemma 3 (Finite tree width) *To represent all possible execution states, the width of a tree of counter-examples need not become infinite.*

Proof 20 (Sketch) *Similar to the latter proof; the number of* LiteralLogic *alternatives is finite at every snapshot; the number of* Loophd *entries for a* LiteralLogic *is constrained by the finite syntax of a Bill expression; since no more possibilities exist to widen a tree, the lemma is thereby proven.* □

Definition 16 (Ko) *The point in a prover tree where an unrolled loop repeats an earlier point in the Bill tree will be referred to as* ko *in what follows*[1]*.*

Proposition 18 (LoopHead ko) *The unrolling of a loop will not make the tree of candidate counter-examples infinite if the definition of ko would only be applied to* Loophd *points.*

Proof 21 (Sketch) *When a point in a Bill tree is ko during the unrolling of a loop, then everything that comes after there is also ko; after all, the* LiteralLogic *that matches an earlier situation defines the complete knowledge at that point, and therefore allows precisely the same continuations as before.* □

Proposition 19 (Pruning ν loops) *When an* Alternative *contains a ν loop which is ko in its list of* Loophd *entries, then removing that loop does not alter the meaning of the* Alternative*.*

[1]In the Japanese game of Go, a ko situation is one that has occurred (just) before in the game. This is not allowed because it makes the game repeat itself. Similarly for the prover, no new developments occur after a ko point.

Proof 22 *A point of ko in a loop signifies infinite looping, and in the case of a ν loop that means that the value of ko is true. Because the* Loophds *under an* Alternative *are composed with $\wedge$, and because that operator has true as its neutral operand, the* Loophd *that evaluates to true can simply be removed from the list.* □

Proposition 20 (Pruning a μ loop) *If a* Loophd *for a μ loop is ko, its containing* Alternative *may be removed from the list of the containing* BillExpr*; if this makes the list of* Alternatives *empty, then the* BillExpr *as a whole is false.*

Proof 23 *A point of ko in a loop signifies infinite looping, and in the case of a μ loop that means that the value of ko is false. Because the* Loophds *under an* Alternative *and the knowledge in its* LiteralLogic *are all composed with $\wedge$, the whole* Alternative *is false. The* Alternative *in turn is listed under* BillExpr*, composed with a $\vee$ operator whose zero value is false. As a result, an* Alternative *containing a ko point for a μ* Loophd *can be removed from the list for the containing* BillExpr*.* □

When this happens, the BillExpr will be pruned from its containing Loophd, or in case the inconsistency is the top BillExpr, the whole tree of candidate counter-examples is *false*, meaning no counter-example exists.

Assumption 1 (Sufficient in practice) *Sufficiently many practical occurrences of inconsistency are removed with the aforementioned propositions.*

5.4 Merging tree representations

This section is dedicated to the step of *merging* information from two tree representations to form one. The *merge operator* is written as $⩕$, and it is designed to be equivalent to the normal $\wedge$ operator from a semantical point of view:

$$[\![p \wedge q]\!](Q) \quad = \quad [\![p ⩕ q]\!](Q)$$

Together with the properties of the $\wedge$ operator, this implies that the $⩕$ operator is commutative, associative, idempotent and that it has *true* as its neutral operand.

Aside from its semantical side, for which the merge operator is nothing new, there is the matter of syntax, or in terms of this chapter, data structures. The merge operator is designed to operate on Bill tree representations such that it maintains an invariant. To avoid that the prover must constantly be aware of the influences that several Bill tree expressions (notably, quantifiers in a conjunction) have on each other, the invariant maintained for every tree representation is that it is closed under such influences between Bill tree expressions. The syntactical responsibility of the $\Wedge$ operator is that it must turn two tree representations for which this invariant holds into one tree representation for which it also holds.

5.4.1 Merging logic operators

The simplest part of merging is dealing with standard logic operators; after all, $\Wedge$ is just a syntactical variation on $\wedge$ and its behaviour therefore need not surprise anyone. We merely pose the following equations, that can be read left-to-right as reduction rules.

$$\begin{array}{lcll}
E \Wedge E' & = & E \wedge E' & \\
(p \vee q) \Wedge r & = & (p \Wedge r) \vee (q \Wedge r) & \\
(p \wedge q) \Wedge r & = & (p \Wedge r) \wedge (q \Wedge r) & \\
(\forall x{:}L \cdot p) \Wedge (\forall x{:}L \cdot q) & = & \forall x{:}L \cdot (p \Wedge q) & \\
(\forall x{:}L \cdot p) \Wedge (\forall x{:}M \cdot q) & = & (\forall x{:}L \cdot p) \ \wedge \ (\forall x{:}M \cdot q) & (\text{for } L \neq M) \\
(\forall x{:}L \cdot p) \Wedge (\exists x{:}L \cdot q) & = & (\exists x{:}L \cdot (p \Wedge q)) \ \wedge \ (\forall x{:}L \cdot p) & \\
(\forall x{:}L \cdot p) \Wedge (\exists x{:}M \cdot q) & = & (\exists x{:}L \cdot p) \ \wedge \ (\forall x{:}L \cdot p) & (\text{for } L \neq M) \\
(\exists x{:}L \cdot p) \Wedge (\exists x{:}M \cdot q) & = & (\exists x{:}L \cdot p) \ \wedge \ (\exists x{:}M \cdot q) &
\end{array}$$

In the above, E and E' are *LiteralLogic* instances. We have left out symmetric variants from the formulæ above (and below).

Please note that for the portion of Bill tree expressions without actions, these reduction steps remove all $\Wedge$ operators, so that the resulting Bill tree is one according to the syntax of Figure 5.4.

5.4.2 Merging actions

Merging actions is a complicated matter, mainly due to the quantifiers prefixing them in Bill. The basis, however, is straightforward. An action $[E/F]$ matches certain actual events, and an action $[E'/F']$ matches certain others. The action that matches the conjunction of these two sets of actual events is described by $[E, E'/F, F']$. Interestingly, such a ‘conjunction of actions’ may lead to impossible combinations.

Definition 17 (Impossible action) *The meta-function impossible maps an Action to a LiteralLogic that expresses the circumstances under which the part of the Action before and after the slash overlap.*

$$\mathit{impossible}\ [E/F] \quad = \quad \bigvee\{\vec{x} = \vec{y} \mid a(\vec{x}) \in E, a(\vec{y}) \in F\}$$

In what follows, we look at combinations of actions with quantifiers, where both actions match. This is used to define how Bill expressions with quantified actions merge. Later, we turn to the situation where actions ignore actual events. The table presented in Figure 5.6 summarises the results from this subsection.

p	q	$p ⩕ q$
$\forall[E/F]\,p$	$\forall[E'/F']\,q$	$\forall[E,E'/F,F']\,(p ⩕ q) \ \wedge$ $\forall[E/F]\,p \ \wedge$ $\forall[E'/F']\,q$
$\forall[E/F]\,p$	$\exists[E'/F']\,q$	$\neg\mathit{impossible}\ [E,E'/F,F']$ $\quad\Rightarrow \exists[E,E'/F,F']\,(p ⩕ q) \ \wedge$ $\mathit{impossible}\ [E,E'/F,F']$ $\quad\Rightarrow \exists[E'/E,F,F']\,q \ \wedge$ $\forall[E/F]\,p$
$\exists[E/F]\,p$	$\exists[E'/F']\,q$	$\exists[E/F]\,p \ \wedge$ $\exists[E'/F']\,q$

Figure 5.6: Merging actions.

Merging universally quantified actions. The Bill expressions $\forall[E/F]\,p$ and $\forall[E'/F']\,q$ merge to $\forall[E,E'/F,F']\,(p \wedge q)$. In cases where *impossible* $[E,E'/F,F']$ holds, the set of future snapshots over which the quantifier ranges is empty; since that makes the $\forall$ statement hold in general, it does not lead to unforeseen candidate counter-example pruning, so for the prover remains its conservativeness if the *impossible* condition is ignored. Taking it into account may help to optimise the prover, though.

Merging existentially quantified actions. Existentially quantified actions cannot be merged to form a description with the same semantics; this is a normal result of logic laws for $\exists$ quantifiers. So, the form $\exists[E/F]\,p \ \wedge \ \exists[E'/F']\,q$ cannot be reduced when it arises during matching.

Merging mixed-quantified actions. When one action is universally quantified and another is existentially quantified, as in merging $\forall[E/F]\,p$ with $\exists[E'/F']\,q$, an interesting situation arises. The tendency arises to construct the merged form $\exists[E,E'/F,F']\,(p \wedge q)$, but this may break the conservativeness of the prover due to overzealous pruning of candidate counter-examples.

If *impossible* $[E,E'/F,F']$ holds, there cannot be an actual event leading to a next snapshot where $p \wedge q$ would hold. In other words, the existential quantification yields *false*, and the candidate counter-example would be pruned, and along with it the conjunction of terms that it is a part of. This would be wrong. There may be an $\exists[E'/F']\,q$, and there may be nothing against $\forall[E/F]\,p$ either, so a counter-example is still possible.

The correct merge of the given mixed-quantified actions therefore incorporates the *impossible* outcome to construct the following merged action:

$$\begin{aligned}&\neg \textit{impossible}\ [E,E'/F,F'] \wedge \exists[E,E'/F,F']\,(p \Wedge q)\ \ \vee\\ &\ \ \textit{impossible}\ [E,E'/F,F'] \wedge \exists[E'/E,F,F']\,q\end{aligned}$$

Other combinations. Not all actual events are matched by both actions, as assumed in the foregoing part of this subsection. Any combination of matching and ignoring may occur; for the separate actions, the semantics express the ignoring behaviour, but when we combine actions we should take care of these variations. As before, the semantics takes care of a completely ignored merged action, so only the combinations where one action is matched and the other is ignored must be taken into account.

Assuming that $E \neq \emptyset$, the action $[E/F]$ is ignored if $[/E,F]$ is matched (if $E = \emptyset$, then no new knowledge is added). Applying this variation first to $[E/F]$ in the above explanation, and then to $[E'/F']$ in the same explanation, yields the other terms of interest.

Please note that blocking of a life cycle's transitions is taken care of by the translation of life cycles to William; it therefore requires no special attention here.

Complete action merging. The above explained the things that matter when actions are merged; in Figure 5.6 we specify it more formally.

5.4.3 Making an initial Bill tree comply

The foregoing part of this section explained how to derive a merged Bill tree in which all possible influences between quantifiers has been expanded.

This was done with the assumption that the Bill trees that are merged also make this assumption true.

The prover algorithm receives a number of Bill expressions which are turned into Bill trees by the bill2tree procedure, and the outcome of that procedure need not make this assumption hold yet. This is resolved by extending the bill2tree procedure with a post-processing stage, in which all $\wedge$ operators are replaced by $\mathbin{/\!\!\!\wedge\!\!\!\backslash}$ operators, followed by a reduction of the merge operators along the lines of the foregoing subsections.

5.5 Open and closed specifications

Life cycles are intended as components, and the prover therefore reasons about a component-based system. This means that a specification should by default be treated as an *open specification*, meaning one that is open to extension with more life cycles. Every value that is stored in the variable countex in Figure 5.1 is in fact one such open specification; the intermediate results are the ones that can be extended later on.

When it is certain that no more components will be added to a specification, it is safe to call the system a *closed specification*. After a specification is closed, it is possible to remove a number of constructs that are only kept in the Bill tree in countex in the hope that they lead to a contradiction when combined with later components.

One kind of construct that is only needed in open specifications is what comes after a $\forall[\ldots]$ operator. These operators do not claim the existence of anything in the counter-example, and may therefore be removed when the specification is closed. This reasoning does not apply to the form $\forall x{:}L$ because the set L is not empty.

The second and last construct that need only be maintained in open specifications is an existential quantifier that did not result from the original $\neg$prop value. The following subsection deals with these cases in more detail.

These removals that are possible at the time of closing a specification are not part of the prune operation because that operation is (also) applied to open specifications; this means that the prover returns an open specification, and the points above merely illustrate issues to take into account when presenting counter-examples to end users.

5.5.1 Candidate counter-examples only

In the prover algorithm, a different treatment must be given to the property to be proven and to the axioms that are applied to it in an attempt to destroy counter-examples. For example, consider a statement

$$\exists[e(x)]\,\mathit{true}$$

If this expression occurs in the inverted property to be proven, then it states that the event $e(x)$ leads to a counter-example. When that same expression occurs in an axiom, it does not describe a possible counter-example. The only reason why it is interesting, coming from an axiom, is that it can destroy a phrase like

$$\forall[e(x)]\,\mathit{false}$$

The general version of the problem at hand is that snapshots in a Bill tree can either be part of a counter-example, or not. If it is not, it will not be useful in generating counter-examples, at best in finding cases against them. This problem can be solved with a special predicate **c** that can be attached to the knowledge in a snapshot; this would mean that the aforementioned different forms would be distinguishable as

$$\begin{array}{l}\exists[e(x)]\ \ \mathit{true}\\ \exists[e(x)]\,(\mathit{true}\wedge\mathbf{c})\end{array}$$

We define **c** as a constant whose value is *true*, but we do not use this property anywhere in the prover, so that it behaves as an unknown value. In terms of implementation, the presence of the **c** predicate can be represented as a flag in the tree nodes. The mark algorithm in Figure 5.1 is just a routine that sets this bit on all the tree nodes, to override an initial value of ‘unmarked’ or *false* for that bit.

This predicate **c** can be handled by $\wedge$ and $\bigwedge\!\!\!\wedge$ operators as any logical value, so that it ends up in all the nodes that arise from combinations that are derived from the original counter-example. When it must be removed, it can be considered as *true*. This also explains why it is ‘safe’ to determine the conjugation of the original counter-example with **c**.

5.6 Depth of proof

The foregoing sections sketched a prover algorithm, but there is one more issue that is orthogonal to all these issues. When quantifiers are taken into account, it becomes possible that they are introduced an infinite number of times.

These unbounded introductions occur for quantified expressions that apply to themselves, directly or indirectly. The simplest form of this situation is

$$\forall x{:}L \cdot \exists y{:}L \cdot p$$

Where the thing stated about x's in general also applies to the y, introducing a new y, and so on, without end.

This form cannot be evaluated until it is either *true* or *false* by a prover; at least, not according to currently existing theorem provers. We cannot solve this problem here, and choose the same solution as is customary in the world of theorem proving, namely to restrict the number of introductions (per quantifier). It is a practical, albeit somewhat unpleasant, solution to a seemingly unsolvable problem.

Unbounded introductions also occur when link traversals introduce new ways to refer to an instance, as in

$$\forall x{:}L \cdot \ldots x.l \ldots$$

where $x.l$ is an instance of L; applying this statement to that linked variable introduces $x.l.l$, which is also an instance of L, and so on, without end.

It is likely that a pattern can be detected in these constructs; the advantage over the first form being that a link is traversed, so that existing loop resolving approaches may apply. A practical solution would again be to limit the number of links traversed. This means that the knowledge stored in a Bill tree, notably that stored in a *LiteralLogic*, must be sorted for the number of links that must be traversed to derive its value. The proof then proceeds for a certain 'depth' of link traversal; it may even make multiple passes of increasing depth, to achieve the quickest possible removal of counter-examples.

The last place where unbounded introductions can occur is for $\forall[\ldots]$ quantifiers, but these are resolved by treating loops as cyclical structures in the Bill "tree" structures. This is a normal approach in model checkers, and nothing new is added in this research. It should only be noted that life cycles are more subtle when loops are merged, because actions may or may not match, leading to more variations in looping alternatives and loop lengths.

5.7 The complete prover

The algorithm in Figure 5.1 gives the outline of the complete prover, and the sections following it specify details. This section is dedicated to some overall reflections over the prover.

In some cases, the prover will be certain that a counter-example does not exist, and therefore be able to agree with a specification. In other cases, the prover will have a definite counter-example and be certain that a counter-example exists. But, as the previous section discussed, some proofs cannot be terminated because they introduce infinite structures; in such cases, the prover takes the conservative approach and generates counter-examples; but it could be noted that the prover is less certain than in the cases that are finite yet retain counter-examples.

The prover has not been designed for efficiency, it has been designed to allow automated finite reasoning over certain infinite and unbounded executions of processes. Loops, specifying infinite traces and/or trace alternatives, are customary handled in a finite way in existing model checkers. What is added by this research is a way to abstract from unbounded numbers of instances, by abstracting from their state, without loosing more detail than would be practical. This applies to the use of states, and of (partial) partitions on variables, based on their equality.

The prover has several places where it exponentially unfolds cases. Although this usually applies to a small exponent, it doest imply severe running times and memory footprints for less comfortable cases. Even so, exponential executions are finite — this is the most fundamental step, what remains is optimisation.

The counter-examples derived by the prover take the shape of a Bill tree, and this is not the most probable format to present to an end-user. As stated before, we believe that a counter-example should ideally be as concrete as possible. This means that concrete instances must be combined, according to the rules of the Bill tree holding the counter-example, and the e.g. annotation in life cycles diagrams has been designed to accommodate that process with plausible instance names. It is so much simpler to view counter-examples about Tarzan and Jane than one about Person1 and Person2! These instance names not only occur as life cycle instances, but also as parameters to actual events; links between these occurrences of the names are easier drawn when the end-user can control the instance naming to provoke a mental image.

Rather than presenting the tree of alternative ways that something can go wrong, it would be good to find one or a few smallest counter-examples. There are two values to minimise in a counter-example, namely by path length, or by number of instances used. The path length is the number of actual events that must occur before a contradicting snapshot is found; the number of instances is the number of life cycle instances mentioned, and this can be lowered by attempting to make as many same-type Bill variables equal to each other as possible. After all, the *LiteralLogic*s contain *partial* equational knowledge, and simplifications are still possible by equating Bill variables.

About implementation. The implementation of this prover is a non-trivial task. As presented here, the prover is finite, and conceptually simple; but to implement it is still very hard.

An important problem is that the data structures grow exponentially, mainly due to distributive laws that unfold to much greater forms, followed by even more expansion due to merging. This fact was known; but it was not expected to be so prevalent as it was in our test implementation. This problem severely constrains the area of examples that can be directly proven with this approach. Our implementation work in a functional programming language led to a process that consumes far over 3GB of memory to conduct proofs about the Tarzan/Jane example in the introduction. A severe indirect effect is the inability to test the code properly; and for this reason it is not part of this thesis work.

To make the prover into a useful tool for realistic examples, it should consider the prover described here as a basis, and extend it with some of the optimisation techniques from the field of model checking; we expect that partiality may yield very good results. Applying such techniques is new research, because it requires a reconsideration of the principles underpinning the optimisation; after all, it is not precisely model checking what is done here. Optimisation is considered to fall out of the scope of this thesis, and therefore the construction of a practically usable tool must also fall outside this scope.

5.8 Conclusions

This section worked out the theoretic models from the previous chapter, turning them into data structures. A procedure for proving correctness of properties in the mathematical Bill language was described, and as a result, it is possible to prove properties about life cycles, at least in theory.

This theoretical prover can generalise over unbounded instantiation of processes, which is a new result. It took the integration of some theorem prover technology with a model checker, and as demonstrated, this integration is possible.

The resulting algorithm can handle unbounded instantiation gracefully, albeit that the algorithm was not designed to be optimal, nor to cover all thinkable applications. We believe however, that the set of life cycle applications over which proofs are possible encompasses many, if not most, practical applications.

We are certain that the class of applications to which this theoretical provability applies is larger than the class of applications covered by a model checker, because (1) all model checking techniques can be combined with our prover, and (2) in addition to model checker techniques, there are techniques to deal with unbounded instantiation, which is not covered by model checkers.

Chapter 6

Practical applications

To invent something, you need
a bit of imagination and a pile of junk.

Thomas Alpha Edison

This chapter demonstrates the practical usability of the previously proposed life cycles. It shows how code generation can generate a web interface to a database. It shows how the precise definitions of behaviour are beneficial for code optimisation. It shows that life cycle notation is fairly close to common notations for object behaviour and for workflow, as used in UML.

6.1 Translation to SQL

This section discusses how to translate life cycles to SQL. On first thought, it may seem logical to map the behavioural semantics of life cycles onto an object database instead of a relational database. However, the cooperation schemes of life cycles sometimes call for bulk updates, and these are most intuitive on relational databases.

Our life cycle processes map to a standard SQL implementation and can thus be used as wrappers around a normal database, possibly one that already exists. For instance, the tables in Figure 6.1 form an administration of once reigning pharaohs and their pyramids, sometime in ancient Egypt. Note that we distinguish between a Person and his possible role as Pharaoh.

If a pharaoh is in a Reign state, which is noted in the state attribute, he is reigning the country, but no son was born to him yet; the son attribute is set to null. In contrast, if the pharaoh would be in the state Son and Reign, he would have been blessed with a son to follow him up after his death.

Process aspects of such a schema involve the crowning of the son when a pharaoh dies in the Son and Reign state, and not crowning anyone when he is in the Reign state when he dies. See Figure 6.2 for the Pharaoh life cycle that specifies such process properties. We assume an opaque type Person; persons can play the role of Pharaoh if crowned as such; persons die, or have a son; a Pyramid is ordered by the Pharaoh to bury the Person in, (hopefuly) after he has died. Such process knowledge is usually available in the SQL procedures accessing the database, albeit scattered over code fragments.

Pharaoh			
this	state	person	son
	Reign	Tutankhamen	—

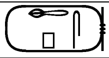
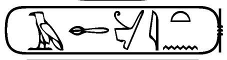
Pyramid			
this	state	pharaoh	person
Chez Cheops	Closed		Cheops
Kings' Valley 2	Closed		Akhenaton
Kings' Valley 3^{A}	Ready		Tutankhamen

Figure 6.1: Database tables for Pharaoh and Pyramid.

6.1.1 Examples of translation to SQL

Before exploring the formal translation procedure, this chapter presents a few examples, to explain the main issues involved in the translation.

Schema translation. We intend to translate life cycles to relational databases, both because their stability and persistence[1] are desirable prop-

[1]The pharaohs were interested in persistence too. Their name, represented in hieroglyphs (or *divine writings*) had to remain written down to ensure continuation of

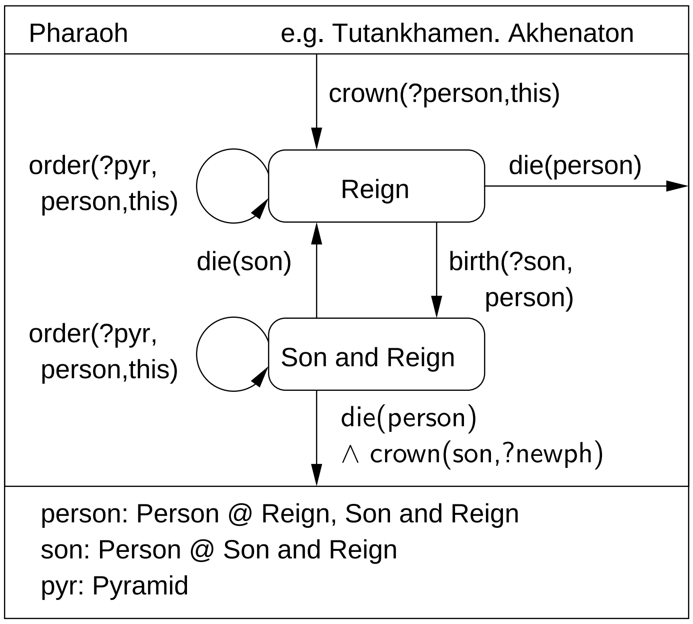


Figure 6.2: The Pharaoh life cycle, 𓀼

erties, and because SQL turns out to be a good implementation language for life cycles.

The most direct way to translate life cycle models to a relational database schema is to consider a life cycle as a table, and life cycle instances as tuples. Life cycles have variables, which are represented in columns of these tables. Variables are not visible in every state of a life cycle (see the variable declarations in Figure 6.2), which is dealt with by using null values.

Figure 6.1 shows the database table for the Pharaoh life cycle, with the identity stored in this, the current state in state and two variables, person and son. The variables pyr and newph are not visible in any state and have therefore been left out of the Pharaoh's table.

Simple transitions. A simple example of an actual event is die(Tutankhamen), representing the occurrence of a die event in reality. That actual

their Afterlife. This has led enemies of buried pharaohs to crudely systematic removal of carved names from tombs.

event matches a formal event on the transition from state Son and Reign to Reign. As a result, that transition occurs for those tuples in the Pharaoh table that have the value Tutankhamen stored in the son attribute[2]. This description can be expressed in SQL, with a where clause to select the correct state and variable values:

```
update Pharaoh
set state='Reign'
where state='Son and Reign' and son='Tutankhamen'
```

The where clause only looks at the attributes that occur as output parameters on the transition, and nothing forces the use of a table's identity. This means that a where clause can be used to select many life cycle instances, which jointly undergo this transition. Such joint transitions employ the efficient bulk updates that SQL has to offer, which is clearly advantageous. However, for this particular transition we know that only one Pharaoh tuple is the father of Tutankhamen, so this is not a bulk update.

The same actual event may at the same time cause another transition on other tuples, namely the transition annotated with formal event die(person), which has the following SQL formulation:

```
delete from Pharaoh
where state='Reign' and person='Tutankhamen'
```

This is a delete statement instead of an update statement because it leads to the Nirvana state. These transitions are combined in one transaction so that the actual event is either entirely accepted or entirely rejected. To avoid a tuple to step through both transitions in response to a single actual event, the transaction should do the delete before the update.

Reading a variable. Some transitions contain formal events like birth (?son,person), with a question-marked parameter indicating that the variable is read from the actual event. An actual event birth(Tutankhamen, Akhenaton) matches when the person variable is equal to Akhenaton, and the Tutankhamen value is of no influence on the match:

[2]This example is not historically correct, as Tutankhamen did not die before his father, Akhenaton. Instead, Tutankhamen became a pharaoh at the age of 9, when his father died. The death of Akhenaton was sudden and mysterious, and was probably caused by Tutankhamen's uncle Ay, who got to rule Egypt for Tutankhamen while he was a boy [Tel00].

```
update Pharaoh
set state='Son and Reign', son='Tutankhamen'
where state='Reign' and person='Akhenaton'
```

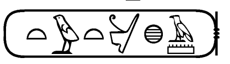
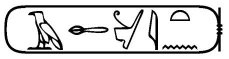
Blocking actual events. A computer model can only model reality under certain assumptions. An implementation based on life cycles will be proven correct, under the assumption that reality behaves according to the specified behavioural constraints. When these would be violated, the software could misbehave. Some orders of execution are blocked in a life cycle implementation, and an implementation is assumed to support such blocking in some way.

We employ preconditions that block a transition to assert the behavioural constraints. It is an advantage of life cycles, caused by bringing process aspects together in specific process models, that behavioural constraints can be easily overviewed by humans as well as algorithms. This makes it straightforward to generate preconditions from life cycle models.

For example, if a pharaoh tuple with identity[3] 𓏏𓅱𓏏 is in state Reign, there cannot be an actual event crown(..., 𓏏𓅱𓏏) because such a transition does not depart from the current state.

As a point of caution, a computer model cannot capture all events that may occur in reality. In our life cycles, we consider the *alphabet* of events to be the formal events mentioned in the state diagram, including their parameter matching constraints (where null attributes match nothing), to be the part of the real world the model knows about. Blocking only occurs for unforeseen orders of actual events from the alphabet. For an actual event to be matched, there may not be any tuple that matches it only from another state than the current state.

This can be implemented in SQL with preconditions tested in the beginning of transactions that implement the life cycle response the the occurrence of actual events. These preconditions make the transaction fail if they are false. For example, a birth(Tutankhamen,Akhenaton) actual event can be matched by the birth(?son,person) formal event by any matching pharaoh in a state where the person variable is visible, so in states Reign as well as Son and Reign. The Reign state is the state from which the actual

[3]The 𓏏𓅱𓏏 hieroglyph is pronounced as Tut, short for Tutankhamen:
 The only long Pharaoh identity used in the remainder of this paper is Akhenaton's, 

transition departs, so the precondition only blocks for tuples in state Son and Reign. The transaction will therefore fail, and thus block the actual event, when the following query is false:

```
select count(*)=0
from Pharaoh
where state='Son and Reign' and person='Akhenaton'
```

Creation of new tuples. The previous example showed that the transition to the Nirvana state resulted in tuple deletion rather than update. Similarly, a creating transition is a transition departing from the Nirvana state.

Conceptually, we imagine an infinite number of life cycle instances in the Nirvana state at the beginning of each transaction, which is very close to the practice with replication in π-calculus [Mil91] [PT97]. Every tuple is assigned a unique identity, referred to by the variable this. When an actual event matches a creating transition including its this parameter(s), a new tuple is created with that this value as its identity. To avoid infinite creations, we require that a this parameter occurs on every creating transition.

The conceptual model of an infinite number of tuples 'standing by' in the Nirvana state cannot be implemented directly. Our SQL implementation delays the insertion of a new tuple in the life cycle's table until a creating transition occurs. At this time, the value of this must be set too, which is somewhat different from its treatment as output variables in the graphical notation. A transition like crown(?person,this) creates at most one Pharaoh tuple during a transaction, and the identity is set to the value 𓏏𓅱𓏏 for an actual event crown(Tutankhamen,𓏏𓅱𓏏).

```
insert into Pharaoh (state,person,this)
values ('Reign', 'Tutankhamen', '𓏏𓅱𓏏 ')
```

To avoid the same-time existence of multiple life cycle instances with the same identity, we should block any such attempts with a query that acts as a precondition and may therefore not yield false:

```
select count(*)=0
from Pharaoh
where this='𓏏𓅱𓏏 '
```

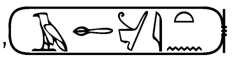
Multiple formal events. A formal event is an abstraction of an actual event. Since abstractions are generally intended to leave out irrelevant details, and since irrelevance depends on perspective, the same actual event may be described with different formal events for different perspectives. Life cycles support this with multiple formal events on every transition. The transition marked with events die(person) | crown(son,?newphar) is an example of a transition that captures an event occurrence from two perspectives: that of the dying person and that of his son. Another transition captures only die(person) to reflect the absence of the perspective of the son: In state Reign, the pharaoh has no son.

We describe actual events as a set, such as die(Akhenaton),crown(Tutankhamen,), to reflect both perspectives — but for readability, we often leave off the braces in case of a singular set. Recall that life cycles do not model causal relationships, and certainly not procedural caller/callee relationships. In many cases, the causal relationship is unknown[4].

Communicating life cycles. The added value of life cycles to object models is a result of how they communicate. The scheme that makes them cooperate is designed for easy composition. Life cycles *independently* observe (and decide to match, ignore or block) the actual events.

For the SQL translation this means that an actual event is implemented as the orthogonal update of the applying transitions of all life cycles.

The addition of a new life cycle to the current system with only the Pharaoh life cycle demonstrates this idea. Figure 6.3 shows the life cycle of a Pyramid[5], which may be ordered by a reigning pharaoh, using the order event. Such an actual event triggers a normal transition in the Pharaoh life cycle, and it triggers the creation of a new Pyramid instance. This is one of the things we like about life cycles: Creation, deletion, and 'internal' transitions all look similar. There is no need for an explicit instantiation operation, which helps to isolate life cycles from each other and thus simplify their composition.

The translation of the creation of a new Pyramid instance in response to an order(King's Valley 2,) actual event is modelled with this SQL query:

[4]Tutankhamen was only 18 when he died and he may have been murdered by his uncle Ay who probably did not want to lose power. Ay married Tutankhamen's widow Ankhsenoomun, even though he was her grandfather, probably to retain power [Tel00].

[5]The hieroglyphs read: *Great house of the pharaoh, giving [him] life forever.*

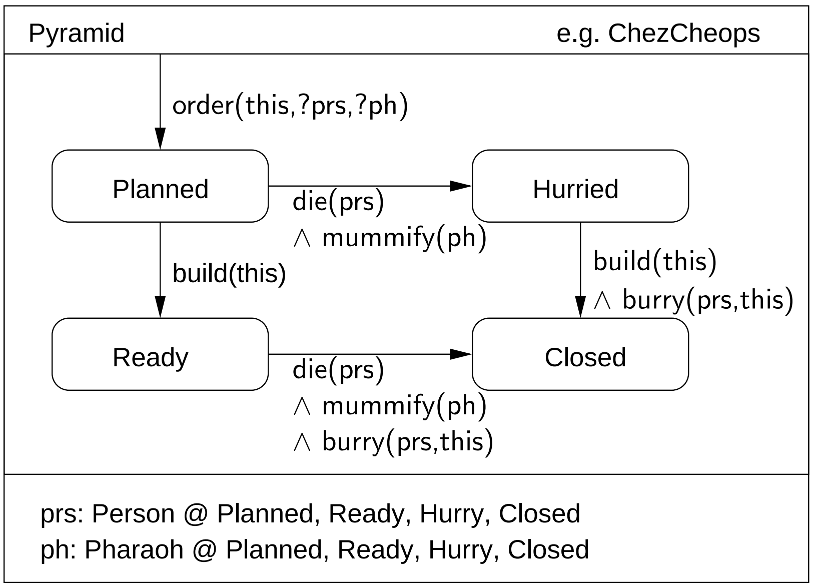


Figure 6.3: The Pyramid life cycle,

```
insert into Pyramid (state, this, person)
values ("Planned", "King's Valley 2", "                ")
```

under the usual precondition:

```
select count(*)=0
from Pyramid
where this="King's Valley 2"
```

The precondition blocks the use of any King's Valley 2 identities that are already in use, meaning that the Pharaoh life cycle's attempts to order a pyramid is limited to new ones. The Pharaoh life cycle does not express this constraint at all, and yet it is subject to the same constraint, because the two life cycles are joint and perform the response to the occurrence of an order event in the same transaction.

The result of this scheme of composing life cycles is that they are able to communicate in spite of their orthogonality.

6.1.2 Translation concepts

The previous subsection demonstrated a few translations of life cycles to SQL. This section treats the general translation scheme behind them. This general schema starts with some preliminary definitions of formal notations and concepts that define life cycles. This is followed by an explanation of how schema's, preconditions, and transition implementations are generated from life cycle models. We finish this section with a discussion of combining multiple life cycles, both 'at compile-time' and as components.

Preliminary definitions. We define an *actual event* as a set O of jointly occurring events, and all $ev \in O$ as an abstract data type with a function $nm(ev)$ defining the (distinct) names, and a function $parm(ev, i)$ obtaining its ith parameter value.

We define a *formal event te* as an abstract data type with a function $nm(te)$ defining its name and a function $parm(te, i)$ obtaining the name of the variable used as the ith parameter, and a function $out(te, i)$ that is only true when the ith parameter is an output parameter, and a function $in(te, i)$ that is only true when the ith parameter is an input parameter.

We define a *transition tr* as an abstract data type with a function $from(tr)$ to determine its origin state ($\neq$ Nirvana) and a function $to(tr)$ to determine its destination state ($\neq$ Nirvana) and a function $events(tr)$ to determine the set of its formal events.

We define a *life cycle* with a name L as a set $\mathcal{T}_L$ of transitions, with at least one transition from Nirvana, and where all transitions from Nirvana have at least one output parameter that refers to variable this, and where no input variable on any transition refers to variable this.

Visibility. Variables are not visible in all states. A variable must be *at least* visible in states from which some future transition uses its variable without having read it first; it must be *at most* visible in states which cannot be reached without reading the variable first; an exception is this, which is visible in all states. Because there is an upper and lower boundary to the visible variables in a state, we allow designers to specify a mapping $\mathcal{V}$ from state names to a set of visible variables between those boundaries. In the life cycle diagrams presented so far, we annotated with @ to define in which states variables were visible. Our scheme for translation to SQL ensures that for all tuples in state s, the attributes unequal to null coincides with $\mathcal{V}(s)$.

Because variables are invisible in some states, we have a way to avoid the use of explicitly visible null values in our designs. This may be interpreted as a dynamic form of the not null constraint on database attributes, similar to integrity constraints like:

```
create assertion VinS
check state<>'s' or v is not null
```

but the advantage in our situation is that they are defined as *model semantics* for life cycles; this means that the information is available to model interpreting tools such as correctness checkers. Extracting information like this from an implementation would be much harder.

The visibility of variables in only part of the states has its impact on visibility of formal events. For example, a formal event like die(son) is defined as a response to the die actual event with a particular variable value, namely the value found in variable son. In states where the son variable is invisible, this event cannot be tested for and the event is thus not visible. A read variable like son in birth(?son,person) does not constrain visibility of the containing formal event, because the value need not be known prior to the read; would we choose otherwise, then the transition would not even be visible in the state from which it transits!

Let $evvars(te) = \{parm(te, i) \mid out(te, i)\}$ represent the variables used as output parameters for formal event te. Then the visibility of te in state s is defined as follows:

$$visible(te, s) \quad = \quad evvars(te) \subseteq \mathcal{V}(s)$$

This definition shows that $\mathcal{V}$ is of influence on event visibility and thus (as we shall see later) on blocking of actual events; this is why it is important to allow the designer to manually define $\mathcal{V}$ between its boundaries.

Recall that every life cycle has an *alphabet* of events that it considers interesting. Since this depends on state (notably, on which variables are visible), the alphabet may differ between instances of the same life cycle. We define the alphabet of a life cycle instance as the set of its visible formal events.

The implementation in SQL presented here makes it particularly straightforward to detect if a transition is visible for a certain tuple: All the variables that are not in $\mathcal{V}(s)$ are null and because SQL will always return false for comparisons with null, much of the SQL implementation implicitly deals with transition visibility.

Matching. The concept of matching differs from the concept of visibility. The visibility of a transition is defined without reference to an actual event. The influence of an actual event on a system is defined by matching. An actual event may or may not match a visible transition dependent on the formal events that trigger it.

A life cycle instance *matches* an actual event O on a transition tr when the events in O, as far as their names occur in the alphabet, coincide with all events on tr, meaning that the same event names occur and that all output parameters on tr have the correct value.

We explained above that the alphabet can be considered as the set of all event names occurring in a life cycle. This means that the question whether O matches tr is answered with false when the event names in O, constrained to the event names in the life cycle containing tr, is not the same as the set of event names on tr.

The inverse situation is more interesting. When the name of an actual event $ev \in O$ matches that of a formal event $te \in events(tr)$, the values of all output parameters of te must match those of the corresponding parameters in ev:

$$\begin{aligned}&\forall ev \in O, te \in events(tr) \cdot nm(ev) = nm(te)\\ &\quad\Rightarrow \quad \forall i \cdot out(te,i) \Rightarrow \underbrace{parm(te,i) = parm(ev,i)}_{\text{SQL}}\end{aligned}$$

The part marked with ‘SQL’ can be directly translated to SQL; the quantifiers surrounding it range over syntactical constructs, that can be unrolled while generating the SQL condition for matching without loss of finiteness. We shall assume this unrolled condition in the remainder of this section, and denote it as $M_L^O(tr)$ with $\mathcal{T}_L$ as the function's domain.

Reading input parameters. When a transition tr of a life cycle L takes place, we assume that it matches the actual event O that it responds to. Aside from the obvious state change, the result of the transition tr is that all its input parameters are assigned the values from the corresponding parameter in O. We can write this as a map to be established:

$$\begin{aligned}R_L^O(tr) \quad = \quad &\{parm(te,i) \mapsto parm(ev,i) \mid\\ &\quad ev \in O, te \in events(tr), nm(ev) = nm(te), in(te,i)\}\end{aligned}$$

6.1.3 Complete translation to SQL

The translation of life cycle models to a database schema is as follows: Every life cycle is translated to a table. Each tuple in such a table describes an instance of the life cycle. The identity of such instances is stored in an attribute, say this. There is another attribute, say state, which contains the current state. Life cycle variables are translated to attributes, which are set to null in the states in which they are not visible. The life cycle variable this maps to the identity of the life cycle instance, and we assume that variables named state do not occur.

Blocking invalid event orders. A visible formal event may block an actual event. This is useful to regulate the processes on a system, to avoid unforeseen execution orders. In our SQL implementation, we implement this synchronisation with preconditions that block (rollback) a transaction when they evaluate to false. Because such rollbacks take place before any updates are performed, the cost should be marginal on efficient transactional database implementations.

In response to an actual event O, some transitions $tr \in \mathcal{T}_L$ of a life cycle L matches under condition $M_L^O(tr)$. Let $T_L(s)$ be the transitions of L that originate in s, defined by:

$$T_L(s) \quad = \quad \{tr \mid tr \in \mathcal{T}_L, \mathit{from}(tr) = s\}$$

So, the matching of an actual event O by any transition from a state s of life cycle L is modelled with $M_L^O(T_L(s))$.

Similar to this, let $E_L(s)$ be the set of formal events of L that originate in s, defined by:

$$E_L(s) \quad = \quad \{\{te\} \mid tr \in T_L(s), te \in \mathit{events}(tr)\}$$

So, if we take the freedom to treat $E_L(s)$ as a set of transitions, the matching of an actual event O by any formal event from a state s of life cycle L is modelled with $M_L^O(E_L(s))$.

In general, a life cycle instance in a state s blocks an actual event O when it cannot perform a transition from s in response to O, and when at least one matching formal event exists in $E_L(s_2)$ for some $s_2 \neq s$. With the above definitions of T_L and E_L, the SQL code to verify this precondition is:

```
select count(*)=0
from L
```

```
where $\exists s\cdot$ state='$s$' and not $M_L^O(T_L(s))$
                and $\exists s_2 \neq s\cdot M_L^O(E_L(S))$
```

where the quantifiers can be unrolled based on the syntax of the life cycles like before. The first line of the where clause selects the tuples that cannot perform a transition in response to O, the second line selects further tuples that have a matching transition in another state. Statements like these can be optimally implemented with techniques from model checking [Eme90] but this falls outside the scope of this paper.

Recall that every creating transition must mention an output parameter this and, because no other variables are known, no other output parameters are possible. Since the value of this is known in all states, we conclude that a creating transition is visible in every state. As a result, one form of this blocking condition is the creational precondition that avoids that multiple instances of a life cycle have the same identity.

Performing transitions. Transitions are either update, insert, or delete statements, or even no code at all. We will first treat the update statements here, and explain the differences for the other versions afterwards.

An update is performed for 'internal' transitions that match and that are in the correct state. In terms of SQL, some transition $tr \in \mathcal{T}_L$ in life cycle L for which $\mathit{from}(tr) \neq$ Nirvana and $\mathit{to}(tr) \neq$ Nirvana is therefore translated to:

```
update $L$
set state='$to(tr)$',
    $v$=null,...
    $w$='$x$',...
where state='$from(tr)$' and ($M_L^O(tr)$)
```

where L is the life cycles table's name, and v=null is repeated for every $v \in \mathcal{V}(\mathit{from}(tr))\backslash\mathcal{V}(\mathit{to}(tr))$, and where w='x' is repeated for every $(w \mapsto x) \in R_L^O(tr)$ for which $w \in \mathcal{V}(\mathit{to}(tr))$.

When $\mathit{from}(tr) =$ Nirvana and $\mathit{to}(tr) \neq$ Nirvana, an insert query must be constructed instead of the update above; this leads to the following simplified form:

```
insert into $L$(this, state, $w$...)
values ($id$, $to(tr)$, $x$...)
```

where *id* is the newly created identity, as assumed by $M_L^O(tr)$. Note that we use the SQL property that non-inserted attributes are set to null.

When $from(tr) \neq$ Nirvana and $to(tr) =$ Nirvana, an insert query must be constructed instead of the update above; this leads to the following simplified form:

```
delete from $L$
where state='$from(tr)$' and ($M_L^O(tr)$)
```

Note that we left out the variable reading from the update code; such statements would be useless since the tuples selected by the (same) where clause are deleted.

A transition for which both $from(tr) =$ Nirvana and $to(tr) =$ Nirvana is even simpler: No update has to be generated for such transitions, only the blocking constraints, if any.

Transactions for a single instance. The previous subsections introduced the preconditions for a life cycle update and the actual update. Obviously, all preconditions are tested before any update.

A few remarks must be made on the order of the updates. For example, if the Pharaoh life cycle in Figure 6.2 experiences a die actual event, then it must be avoided that one instance steps from the state Son and Reign, through Reign, to Nirvana in one transaction. So, when it cannot be inferred that son and person are distinct values for all tuples, we must take some precautions.

The ordering of the updates that represent the separate transitions may often help, but not in general. When necessary, it is always possible to update by setting a temporary state name s on the first transition, then perform the other transitions, and finally change all tuples with state s to the actual destination state.

Compositions of life cycles. A number of life cycles can be composed to cooperate. The cooperation scheme that we selected was inspired on CSP [Hoa85], a process model in which all processes observe 'globally' visible actual events. The advantage of this scheme is that all processes are independent.

As a result, composition of life cycles is straightforward. It means that the transactions of two independent life cycles are mixed in any order. From the viewpoint of efficiency (minimising work to be done on rollback) it is advisable to check preconditions of all life cycles before making any change to any life cycle, but in theory any interleaving of the sequences of SQL statements from different life cycles would be correct.

Extension with a new component. We will now demonstrate how life cycles can be treated as components, that can be plugged together at the time of their utilisation.

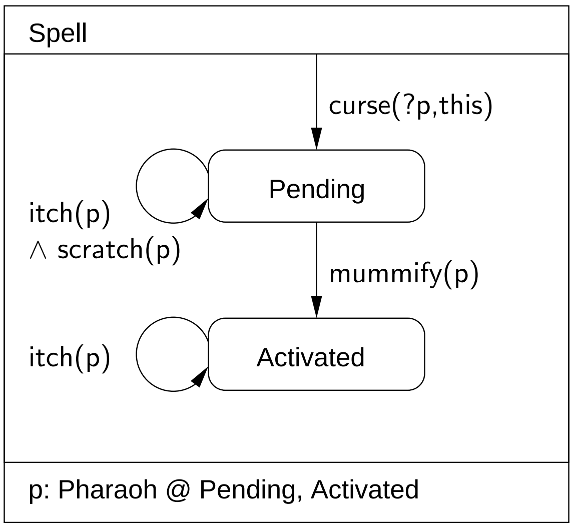
Life cycles have the ability to block actual events if they conflict with the orders that they allow. On the other hand, life cycles have a limited alphabet, making it simple to add new life cycles with new events without any blocking; this is loosened further because the alphabet only works as far as matched by the set of defined variables in each life cycle instance's current state.

This means that a life cycle system can be extended, as long as ordering constraints on the already recognised events are not invalidated. Such extension is commonly referred to as 'component composition' and is considered desirable in modern software architectures [Szy98]. Regarding the process aspects of software, our life cycle approach directly enables such composition.

We demonstrate life cycle system extension by adding a new life cycle to the system consisting of the Pharaoh and Pyramid life cycles. In Figure 6.4 we introduce a Spell life cycle, which is instantiated by cursing a pharaoh with an actual event like curse(𓏏𓀭𓏏 ,'2'), leading to the second tuple in the example table in Figure 6.4. In the Pending state, when 𓏏𓀭𓏏 is itched, he instantly scratches himself to be relieved. However, after picking up the same mummify(𓏏𓀭𓏏) actual event that also influences the Pyramid life cycle, the Spell will become Activated. In that state, Tutankhamen can still experience itch(𓏏𓀭𓏏) actual events, but since scratch(𓏏𓀭𓏏) is still visible but not on the itch(p) transition, his powers to scratch himself are blocked.[6]

When implementing this component on databases, the addition of a life cycle means introducing a new database table. This new table could be located in the original database, or perhaps in another database, which may run on another host. Since the joint transitions of life cycles demands atomic execution of the updates on all database tables, this requires the use of (distributed) transactions.

[6]The hieroglyphs, added to enhance the incantation with magical powers, read *The pharaoh, given [him] work forever.* The ancient Egyptians could be tremendously cruel!



Spell		
this	**state**	**p**
1	Activated	
2	Pending	

Figure 6.4: The Spell life cycle, and an example database table.

6.1.4 Conclusions

We have shown that the implementation of a process system on top of a relational database is possible. The SQL language has proven quite useful to implement the principles of our life cycle models. We were particularly pleased (and surprised) by the straightforward translation of joint transitions of sets of life cycle instances to typical SQL bulk updates.

We therefore conclude that SQL is not only a possible, but even a suitable language for life cycle implementation. Or, stating the same inversely, life cycles form a very suitable process wrapping around SQL databases. Possible applications of such process wrappers include workflow systems as well as object behavioural specifications.

6.2 Replacing garbage collection

Life cycles provide accurate behavioural definitions, and this detailed knowledge can be used to achieve new results. This section gives an example. It explains how the knowledge contained in life cycles can be used to resolve the conflict between garbage collection and explicit deletion of objects.

Most software expects some form of dynamic allocation for objects, and requires mechanisms to delete those objects after their life span. Deletion of such ‘garbage’ can be thought of in three steps, namely detecting it, finalising it (‘cleaning up the data structure’) and deallocating the memory it occupies. Two requirements that usually apply to a deletion mechanism follow. First, it is considered desirable to perform finalisation immediately when an object becomes garbage, so that any locks on resources are freed as soon as possible. Second, it is considered desirable to automate garbage deletion, to avoid the risk of human error. Two common human errors are memory leaks, where some garbage is never deleted, and dangling references, which reference objects that were deleted, thereby breaking referential integrity. Both errors require intensive debugging to locate the bug.

Traditional software contains manually coded garbage detection, which usually has the advantage of immediate deallocation, but which also risks memory leaks and dangling references. Many programming languages, including C and C++, assume this approach to deletion.

Some modern languages employ garbage collection, which deletes objects sometime after they have become garbage. Garbage detection is automatic in this approach, but to keep it efficient, detection is a batch process. Garbage collection is an automated approach and therefore avoids human errors, but its batch nature introduces the disadvantage of deferring the finalisation of objects.

The only thing more advantageous than automated deletion in the form of garbage collection would be automated deletion at the precise moment when a memory structure is no longer used. Such approaches are not feasible in general, because it requires in-depth analysis of a program, which is too hard to do. However, the system overview provided in analysis models of (object-oriented) software can help to make this possible. With life cycles, it is certainly possible.

Code generation. Life cycles have precise semantics that enable the automatic generation of code for the communication schemes set out above;

a programmer must only implement each of the events as a method. Similar results exist for ROOM diagrams [SGW94] and UML state diagrams [HG97]. ROOM processes are statically allocated (not instantiated), so object deletion is not an issue there. The Rhapsody tool for generating UML state chart implementations can only model creation and deletion in fragments of C++ code, which is usually considered too complex for formal analysis. The precise and simple notion of creation and deletion of life cycle instances makes it straightforward to generate code that creates and deletes instances. The only thing that we have to beware of is breaking referential integrity: Variables should not refer to objects that are in Nirvana state.

Constraints. Life cycles allow the precise specification of processes; it is often a good idea to challenge such specifications by stating constraints that are supposed to hold for the designs. Automated verification can help to ensure that the process designs are indeed behaving properly, and if they fail, their feedback can help to trace back the point where things can go wrong, a great asset in improving the process diagrams.

We formulate such constraints in Bill. An example of a constraint is that a ringing tone must be indicative of a ringing phone at the other end of an attempted connection, or more formally,

$$AG \quad \forall p : \mathsf{Phone} \cdot p@\mathsf{Ringtone} \ \Rightarrow \ p.\mathsf{oth}@\mathsf{Ringing}$$

In the above, an actual event dial(4291,3711) is matched by the formal event dial(this,?oth) for object 4291, setting 4291.oth=3711, and by the formal event dial(?oth,this) for object 3711. After this actual event, 4291@Ringtone and 3711@Ringing, so the above constraint holds in this situation. Now note that the dial events rely only on variable this which is known in all states, so that it cannot be ignored from any Phone, not even those that are in Nirvana. As a result, actual events like dial(4291,3711) can only occur without being blocked when the object serving as the first parameter is in state Dialtone and the other is in Nirvana. Since we used no special properties of the actual instances, our reasoning applies to Phone instances in general.

Our approach to garbage deletion Life cycles can be exploited to ensure referential integrity, which is the only proof obligation to ensure that objects can indeed be deleted when they make their final transition.

While processing a life cycle to transform it to Bill, we can generate proof obligations that ensure referential integrity. These would state that, for all variables known, and for all states in which these are defined, they may not refer to an object in Nirvana state. The special, predefined variable this is exempt from this requirement.

Consider the variable son defined in Figure 6.2. This variable is defined in state Son and Reign. This leads to the following proof obligation:

$$AG \;\; \forall p : \mathsf{Pharaoh} \cdot (p@\mathsf{SonandReign}) \Rightarrow \neg(p.\mathsf{son}^{\dagger})$$

If this property is proven to hold (and similarly for all other variables and states in which they are defined), then no variable ever refers to a non-existent instance, or in other words, referential integrity is achieved in all possible runs of the system.

Why this approach works. Our approach to object deletion enables an ideal solution, and it is therefore worthwhile to study the aspects that make it work in comparison to other modelling techniques. One cause is the precision that makes it possible to get a grasp of the models with formal tools. Our choice to make creation and deletion simple enough for formal tools is crucial to this solution. Finally, the formal tools will only search the executions allowed by the life cycles, and hence we assume any implementation to block anything outside these orders.

Conclusion. This section discussed object deletion as an example of the specific information of creation and deletion in life cycles. This information made it possible to automatically delete instances exactly at the moment when their life ends. This resolves a debate between garbage collection and explicit deletion that is fed by less expressive modelling concepts.

6.3 Object behaviour modelling

State charts are presented in UML to capture the behaviour of a single object. Life cycles, which capture the behaviour of a single process instance, serve a similar purpose. The notation of both notations is based on state diagrams, so there is a lot of overlap: Both are based on states and transitions with formal events on them.

In spite of commonalities, the translation of state charts to life cycles is not trivial, due to a quite distinct philosophy behind the notations. The aim of UML state charts is to provide expressive notation, but without much concern about semantics, whereas life cycles have been designed as a minimal notation with accurate, orthogonal and complete semantics, with a close relationship to process algebra.

We cannot formulate a precise translation here, because the semantics of state charts are not stable. Instead, we sketch how the translation works.

Creation and deletion. Every change to a life cycle is the result of observing an actual event, and that includes its creation and deletion. In state charts, creation and deletion is sometimes performed by another object in an imperative statement. This means that the cause of creation and deletion must be extracted from a state charts' environment if it is not noted on its corresponding transitions yet.

Translating actions. An event on a state chart transition can cause actions to be invoked. This introduces causal (and procedural) knowledge in state charts that were cautiously removed from life cycles. In life cycles, it is only possible to indicate that a number of events occur at the same time, or, to express nested behaviour with an event refinement approach; event refinement for life cycles is possible because transitions in life cycles take some amount of time to take place (they are transactional, rather than instantaneous). One added possibility in life cycles is therefore that parameters are passed around in any direction (like in a declarative specification).

States in state charts also express entry and exit actions, as well as internal actions caused by events. All these are most easily handled in a separate life cycle.

Nested states. In object designs, it is important to model the same thing only once, although multiple occurrences in different models are usually accepted. Life cycle diagrams take this a step further; as long as there are no non-deterministic choices, it is no problem to duplicate a process or part of it in another life cycle. This gives the freedom to zoom in or out on behaviour in different life cycles.

State charts handle zooming in and out in the same state chart, by supporting nested states, i.e. a state chart inside the state of a state chart. Although this helps to express overview and detail at the same time, this distinction is always from a specific perspective, and there is no way to express different nesting for different perspectives. To avoid this weakness, life cycles do not work with nested states.

To express multiple perspectives in life cycles, there should be multiple life cycle models. The behaviour of nested states can be expressed by observing events in one life cycle and not in another. The life cycle with the nested behaviour should observe the events in the other life cycle to determine when to create and delete itself.

History states in state charts reflect an inner diagram that 'remembers' its last state. A true hack! The approach with a separate life cycle for the nested behaviour makes it possible to remain in some waiting state until the nested behaviour is called for again, thus making the concept of history states unneeded in life cycles.

Translating life cycles to state charts. State charts are quite expressive, but not capable to express all life cycles. One problem in state charts is that concurrency is constrained separately drawn sub-diagrams; there is no way to express the forking off of any number of instances with the same behaviour. The impossibility of this translation is generally not a problem, because it is undesirable: It looses information because it translates a precise notation into a (currently) imprecise notation.

Translating life cycles to class diagrams. Perhaps surprisingly, much of the knowledge contained in a class diagram is available in a life cycle. Life cycles may reference other life cycles, which expresses the same information as an association in a class diagram. Associations in class diagrams are usually less informative, because they represent a state-dependent association as an optional link. This is not due to the modelling notation (at least not in UML) but to most designer's preferences. This enables a simple and useful way to cross-check the consistency between life cycle and class models, or to transport information between the models, comparable to the ideas behind Catalysis [DW98].

Conclusion. Life cycles and state charts serve the same purpose. Life cycles are more expressive and accurate and resolve some restricted or clumsy notations in state charts elegantly, at the expense of defining simpler process models for a single, complex state chart model. Life cycles contain the knowledge of class models. The extensions of the life cycles in this thesis with event refinement and more data types is needed to make life cycles a complete replacement of state charts and class diagrams.

6.4 Workflow modelling

Workflow models capture process knowledge spanning multiple persons or roles, and fulfilling a single goal. This differs a bit from life cycles, which usually capture a single perspective on a system-wide process and compose these perspectives. But the diagramming languages are not very different, so a comparison seems proper.

A commonly used notation for workflow is the UML activity diagram, which comprises of activities (round-sided rectangles), states (rectangles with rounded edges), choices and the corresponding joins (diamonds), and concurrency through Petri-net notation. All these ingredients are covered in life cycles, making it likely that a straightforward translation exists.

The creative part of translating an activity diagram is to identify threads of execution that will be represented in separate life cycles; there are usually multiple ways of threading, and the selection of the best alternative takes some inspiration and perspiration. In what follows, we sketch a translation from activity diagrams to life cycles which is sufficiently accurate to conclude how the diagrams relate to each other.

Translation. One way of marking threads is by drawing 'paths' (say, of different colours) over an activity diagram, in such a way that all arrows (except the ones from start state and to end states) are covered in at least one colour. Such paths start in an activity or state, and end in an activity, state, or Petri-bar. The threads are split when a choice is made, and joined when alternative branches are joined. It is even possible to draw intermittent threads, meaning that unconnected segments of an activity diagram are marked for the same thread, in such a way that each segments follows the rules of a whole thread, and that alphabet-rules of life cycles are taken into account.

Each thread is now turned into a separate life cycle diagram. A state in an activity diagram becomes a state in the life cycle. An activity becomes a transition (marked with the proper event) in the life cycle, leading to a new state (which is Nirvana if the thread ends after this activity). The Petri-net notation for forking and merging does not translate to notation in life cycles; synchronisation is taken care of by communication rules between life cycles. A join of previously made choices, even when implicit in the diagram, translates to a common state. A determinable choice translates to a guard on the arrows of outgoing transitions that represent the following activities. A non-determinable choice translates to a duplication of the transition of the previous activity; these transitions lead to different states, each representing that a certain alternative was chosen; comments in the guards on choice alternatives can be used to inspire the names of these states.

Extra information in life cycles. Compared to activity diagrams, the life cycle diagrams contains more information. There is a choice of how a global process is split into perspectives. Furthermore, guards in life cycles are usually more pronounced, more accurate. And of course, life cycles have more accurately defined communication semantics. These are forms of additional meaning that represent additional design choices captured in life cycles.

Additional possibilities of life cycles include the forking off of an unbounded number of process instances, enabling construction of overall company processes that initiate other processes. Furthermore, activity diagrams are not supposed to communicate, which is certainly possible with life cycles. This means that life cycles make it possible to construct 'overview' processes such as resource allocation processes.

Quite useful is also the component focus of life cycles, and its relationship with Bill. In terms of activity diagrams, it is only possible to specify accurate processes, but Bill enables more dosed knowledge. For example, if a set of swim-lanes (groups of activities) in an activity diagram is mapped to a life cycle component, then the assumptions about other components can be more finely dosed with Bill expressions than would be possible with complete process specifications in either notation.

Conclusion. Workflow processes such as UML activity diagrams represent processes globally, whereas life cycles tend to look at a single perspective on that process. Life cycles can capture the same knowledge as workflow diagrams, and they are more expressive and more powerful, but also more detailed. This makes life cycles less suitable for the global-overview phase of workflow modelling, but very suitable for the next step. After all, life cycles enable behavioural verification, whereas UML activity diagrams do not.

Chapter 7

Evaluation and Conclusions

A'j altied grómt,
dan he'j een hóndeleven.

Drentse volkswijsheid

The previous chapters of this thesis addressed the concept of life cycles; this chapter evaluates that concept. We end with the conclusions over all the work reported on in this thesis.

This thesis builds a bridge between two areas of expertise, namely object modelling practice and mathematical models such as process algebras. This chapter evaluates the bridge, by checking inhowfar the goals of these two areas are met by the bridging work. If the goals in both areas are met, we can say that the bridge is a good one.

The following sections therefore evaluate life cycles from the perspectives of object modelling and process algebra; in addition to this, we evaluate how well life cycles integrate with technological approaches that are commonplace in modern computer science. Finally, we turn to what we see as useful future work, and we draw conclusions from the thesis work as a whole.

7.1 From the angle of object modelling

The most commonly used notation for object-oriented diagrams is UML, and therefore the most commonly used process notation for object-oriented applications are the state charts and activity diagrams from that notation.

This section evaluates how life cycles relate to this popular notation, starting from the goals of object modelling, looking at notational expressiveness, and we compare our approach to the common mechanism of inheritance.

Goals of object orientation. A driving force behind object-oriented concepts are properties such as improved reusability of models, maintainability and a help in obtaining an overview of the design at hand. This is usually addressed by splitting up conceptually different topics in independent diagrams.

First of all, it is generally simpler to split different conceptual aspects over different life cycle diagrams than is the case for UML state charts. Lacking redundant events and enforcing event causality, a state chart cannot communicate with other state charts as easily as life cycles can; this sometimes forces diagrams to be combined, ultimately leading to activity diagrams that describe a (usually large and complex) process in all its intricacies. Allowing co-operations between split diagrams helps to fulfil the goal of overview over a design.

Secondly, when objects implemented in classical languages communicate, they must necessarily know the parties to send a message to; this makes it harder to extend upon any given design; the goal of reusability is at stake. Life cycles support dynamic communication, and thanks to redundancy, a (deterministic) process in one life cycle can even be followed by another to ensure required developments over time; this makes it possible in most situations to extend a given set of life cycles without touching them — life cycles are good for reusability.

Thirdly, the goal of maintainability demands a well-structured system, even after many changes have been made to its original design. In this respect, it is quite useful that life cycle systems are tested for compliance with a set of propositions in Bill. A design may be altered, and if all propositions still hold afterwards, it can be assumed that the system still behaves well — that is, assuming that sufficient propositions have been formulated. To improve maintainability even more, these propositions can be formulated over partial life cycle systems (or ‘components’) as well as over complete life cycle systems. In comparison, UML provides a language called OCL, which is so complex that it is not really usable for verification in practice. We therefore conclude that life cycles improve maintainability of designs with respect to UML process diagrams.

Fourthly, it is generally considered useful to receive feedback during the earliest phases of system specification. The models of life cycles have a precise semantics that make it possible to derive most inconsistencies, and even to prove arbitrary Bill properties, so that there is a better chance of earlier feedback. Moreover, life cycles have been designed with feedback thath hints towards repairs in mind, so the results should not distract a designer's mind from the content of the application he is designing, which should make repairing the diagrams simpler.

Expressiveness. The syntax of state charts [Cor97] [Har87] is quite extensive. This means that the notational wishes of many practical users are accommodated. Unfortunately, this also means that attempts to define a semantics for UML state charts have had to focus on partial semantics because the whole syntax is too complicated to capture in an undisputable way [LMM99].

By conscious choice, the number of syntactical constructs in life cycles is lower than in UML process diagrams; this may imply that idiom will develop when used intensively, but at least the semantics will be clear, and verification is supported. As demonstrated in Sections 6.3 and 6.4, it is not particularly difficult to construct similar things in life cycles as in UML process notations. This means that, in spite of fewer syntactical constructs, the semantics provide comparable expressive power. In other words, life cycles achieve the same notational wishes with a cleaner notation.

State charts in UML were recently extended with a standard notation for creation of new instances and delete old ones; but this is done with inlined code, a form that does not lend itself well for the kind of analysis described here. This also means that the easily gained advantage of instant garbage collection as described in Section 6.2 is not likely to ever be ported to this notation. Interestingly, state charts were quite late to introduce a standard notation to capture creation and deletion of objects. Even with the newly added creation and deletion constructs, they are not designed to make it easy for verification tools to handle them; we therefore think that the way creation and deletion as treated in life cycles is better usable.

Also interestingly, state charts communicate through a queue which is not specified at all. This leads to a form of uncertainty that makes verification of co-operating objects unfeasible. Life cycle semantics are specifically clear in this, and therefore enable provers to reason about communicating objects. In the modelling language ROOM [SGW94] on the other hand, there is a clear definition of communication between processes, but the method does not support creation of new processes (it aims at control systems).

One important point that limits the capability of life cycles in object practice is its use of rather simple constraints — the OCL part of UML is much more powerful. Even though Section 4.9 proves that anything can be expressed in the constraints supported by the life cycles as presented in this thesis, this is merely a theoretical point, and says nothing about practical usefulness.

Inheritance and refinement. Inheritance is usually considered important in object-oriented modelling notations, because it enables the distinct specification of a concept at different levels of abstraction. But when constructing a semantics for inheritance, it is hard to pin down a single formalism.

In the work of van der Aalst [AB96] [AB97] and Basten [BA96b], a distinction is made between possible interpretations of inheritance. These distinguish the behaviour of an instance of a class is used instead of its superclass; notably, whether additional methods are to be called over another API or if they are to be blocked. UML seems to take the approach of the additional API, without recognising this alternative interpretation, which might be useful in certain applications (for instance, think of security).

In the work of Castagna [Cas97], two type theoretic approaches to inheritance are distinguished; one for overriding a method in a subclass, another to refine an abstract definition. The first is most useful during design and implementation phases, and the second is the most suitable choice during conceptual modelling. It is therefore surprising that UML hints toward the first interpretation of inheritance for its analysis diagrams.

We conclude that the inheritance construct in UML is mainly targeted at design and implementation, rather than at conceptual modelling. Inheritance in UML is formulated in terms of methods, and methods incorporate several programming language constructs that we have evaded for life cycles, in line with the principle of procedural aspects (item 5 in section 2.9). As stated in the principle of early verification (item 1 in section 2.9), we prefer verifying analysis models over later stage models.

Life cycles evade inheritance altogether. The flexible composition of life cycles makes it possible to describe behaviour at different levels of abstraction and let it co-operate in a more flexible fashion than possible with inheritance; notably, redundant specifications and dynamic communication. In addition to this, life cycles have been designed for easy integration with a refinement approach, which allows definitions at different levels of abstractions. This has an option of being more flexible than the strict tree structure, or branch-sharing tree structure, that is usually allowed for inheritance. Given these powerful features, there seems to be no need to incorporate inheritance in life cycles, neither between the diagrams, nor between singular states.

Please note that at least a few object methodologies with refinement have proven their worth, notably Catalysis [DW98] and ROOM [SGW94]. A promising advantage of refinement is that it enables formal verification that one model refines another [BA96b] [BA96a].

Conclusion. Life cycles provide a better implementation for the goals of object modelling than UML, and do so with a cleaner notation. Even though life cycles deliberately lack support for inheritance, the alternative of refinement suggests more powerful modelling features.

7.2 From the angle of process algebra

The formal theories that are closest to life cycles are process algebras. The term process algebra [BW90] does not refer to a single theory, but to a family of related theories. We will mention specific theories where they illustrate this evaluation.

This section evaluates life cycle properties by first looking at the goals of process algebras. Specific attention is then given to expressiveness of the language, to communication structures, process creation, dynamic alphabets, multi-actions and refinemtn.

Goals of process algebras. The intention of process algebras is to unambiguously capture process semantics. Since not all processes are defined with the same idea in mind, there exist many flavours of process algebras. Process algebra research includes model checking approaches, where the aim is to prove the presence or absence of properties, either fixed or user defined. Life cycles answer to all these goals, by defining a semantics and an approach to conducting proofs.

Expressiveness of processes. In its most general form, process algebra can handle irregular processes, meaning semantical tree structures which are infinite and have no repetitive structure. Since we consider a practical application, where predictable behaviour is important, life cycles concentrate on the large subclass of regular process structures. This is sufficient for object modelling practice.

Process algebras can express parameters, but not in a way suitable for object-oriented practises. The usual trick is to replace a parameterised transition $\mathsf{a}(\mathsf{x})$ with one for $\mathsf{a}(1)$, another for $\mathsf{a}(2)$ and so on, which may make the model infinite-sized. Life cycles represent parameters explicitly in the syntax, and even in the alphabets, to avoid this trick. This leads to finite alphabets, which are easier to handle in executing systems.

Communication structure. Most process algebras include a form of communication between processes. This is usually captured in a binary operator that composes two processes into one communicating process. Life cycles make the choice to define this operator as a 'broadcasting' communication operator, as possible in CSP and ACP, and unlike CCS and π-calculus which describe point-to-point communication structures.

This means that life cycles inmplicitly exploit the popular Observer design pattern. Furthermore, it avoids the metaphor of message sending which is commonplace in CCS and π-calculus applications, and this leads to stronger support for composition of components.

Process creation. The ease with which new objects can be created in an object system is not mirrored by easy process instantiation in most process algebras. The more recent theory of π-calculus [Mil91] was the first to introduce a truly elegant and practical [PT97] construct for process creation which blends in well with the rest of the calculus. The π-calculus shares the property of point-to-point communication with its predecessor CCS [Mil80]. We prefer broadcasting communication over point-to-point communication for reasons of composability of specifications.

The support of life cycles for this broadcasting style of communication made it simple to introduce a method for creation and destruction of instances that very smoothly blends in with the remainder of the semantics. In practice, this means that even object creation can be performed in a polymorphic way.

Alphabets and parameters. When ‘running,’ each process instance is in a certain state, and in those states a number of events can be accepted. Every process also has an alphabet, containing all possibly acceptable events. The events contained in the alphabet of process P but *not acceptable in the current state* of P are *blocked* or disallowed. This is intended behaviour.

It is customary in process algebras not to dedicate much attention to parameters. If an action a accepts a parameter which can be 1, 2 or 3, then the alphabet for that process is considered to include a(1), a(2) and a(3), all of which are distinct atomic actions. We find that this overlooks an important problem. Imagine a process with a variable x, encoding event a(x) as a possible transition. If at that time x=3, then a(x) can be read as a(3) and it is clear that event a(3) is accepted at that time. Another process may at the same time be accepting the same transition on event a(2) because it has a variable x=2. The problem is that both processes have a(1), a(2) and a(3) in their alphabet, and so for every element in the alphabet, there is always at least one process blocking it. So, all of the a(x) events are blocked. This is behaviour that we consider undesirable, because it easily causes deadlock in a tightly collaborating system of processes.

Life cycles solve this problem by employing *dynamic* alphabets, meaning that they evolve over time. Although not discussed in this thesis, it is straightforward to transform a process with a dynamic alphabet to one with a static alphabet, but the result looks kludgey and is certainly not presentable as a syntax to present to the designer, because it doesn’t directly match concepts in the design directly: Life cycles, practical as they are, reflect the concept of dynamic alphabets.

Multi-actions. We decided to support multiple views on a life cycle system. Different perspectives often imply different vocabularies, and different interest in parameters. To make it possible to observe the same event occurrences in different ways, life cycles support multiple events on a transition, which jointly occur, instead of just a single one. This is possible with the concept of multi-actions, which is documented in process algebra literature [Bla96] [BW90].

Refinement. Another topic addressed in our discussion of object modelling was refinement, see the principle of refinement (item 7 in section 2.9). This approach to modelling is also well known in formal theories [Spi92], and applications [RG99] [BA96b] [Esh98] to process algebra exist.

Conclusion. Life cycles take the approach of globally communicated events; this avoids the metaphor of message sending, and evades the need to employ common design patterns for subscribed-party notification and polymorphic creation. Dynamic alphabets make life cycles practical because they avoid dangers of deadlocking when event parameters are involved. Finally, life cycles welcome theoretical concepts such as multi-actions and refinement, making them usable for practical applications.

7.3 Integration with technology

The previous sections discussed several problems that arose during our attempts to integrate process algebra and object models. The solutions we found show a possible path toward integration of these two disciplines.

In what follows, life cycles will be reviewed from the angles of composition of components and support of transactions.

Components. In general, we define a component as a composition of at least one life cycle, with at least one interface. An interface is an external access point on which a dynamic alphabet (or *API*) is visible.

In Figure 7.1 a single component is shown with several life cycle instances, and the instance with `this` set to `17` is currently in state `S`. At the bottom of this figure, there is an *interface* intended for plugging components together. This interface reveals the current alphabet of a process. For this diagram, we can expect `dosth(17,*)` in the alphabet, where `*` is written to indicate that any value at that place matches (its value is to be read).

Any number of such components run in an environment which invokes actions from the alphabet onto the components. Each component can then process it internally, perhaps leading to creation and deletion of life cycle instances, and depending on design choices they may or may not communicate with a persistent store. If an event is offered over the interface but a life cycle instance decides it should be blocked, then the interface reports failure back to the caller, who rolls back its attempt to initiate the action. The component interfaces hide all such rudimentary details. After processing is done, the only visible change over the external interface will be a new alphabet, intended for subsequent actions.

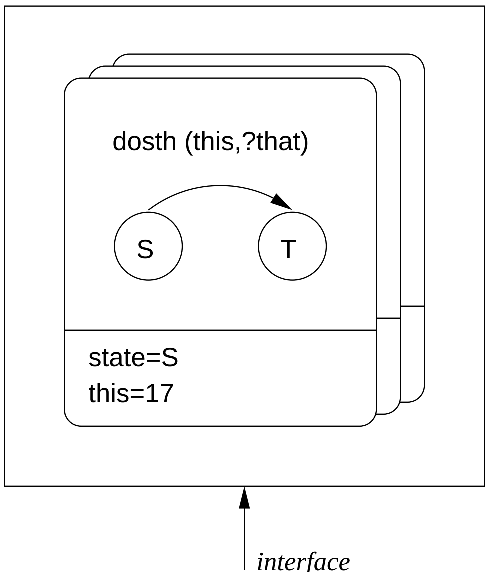


Figure 7.1: A component with many instances and one interface.

Because the internals are hidden, a life cycle component may also contain more than one life cycle composed with the $\|$ operator; similarly, the addition of more than one interface to a component is not a problem (and can serve practical purposes rather well).

Composition. To make multiple life cycle components communicate, their interfaces must be connected. This means that some linking code will realise an action by accessing all connected component interfaces with that action in their current alphabet.

The example in Figure 7.2 demonstrates a composition of components, where we have written only the alphabets inside the components. Note how the left hand component is the only one to mention the `e(7,*)` action; this means that an action like `e(7,2)` may occur, but the right hand component will ignore it. Also note how the action `f(8)` in the left side component is a special form of the action `f(*)` on the right. This means that an action `f(8)` is *jointly* processed by both components, but $\mathsf{f}(x)$ for $x \neq 8$ is ignored by the left side component.

The $\|$ operator is once again suitable for this composition. Combined

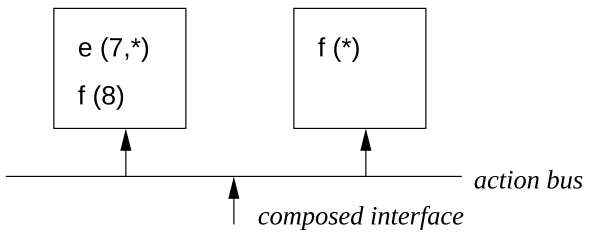


Figure 7.2: Components composed by plugging their interfaces into a ‘bus’.

with the same operator to compose multiple life cycles in components, and the associative and commutative laws we know that for any components A, B and C it holds that

$$\begin{aligned} A\|(B\|C) &= (A\|B)\|C \\ A\|B &= B\|A \end{aligned}$$

So, order and brackets are irrelevant. This means that components may themselves be compositions of components, in any thinkable way; this nested encapsulation structures are of *no influence* on the behaviour of the composition. Consequently, the distribution of behaviour over components may be based on any goal or desire, including managerial decisions in response to market need for components.

Note how the components are *unaware* of each other. Every component processes the actions it is informed about and publishes its new alphabet after having processed it, and some generic code for component linking does the rest. Or, when one component signals that an actual event should be blocked, then all components rollback to the previous state; this can also be taken care of by generic code for component linking.

Transactions. Programs can fail to perform their assigned tasks for various reasons. One accepted means of signalling such failure is by throwing exceptions. Process algebraic actions do not model failure, because actions either succeed or simply do not occur at all. The same concept is the intention of transactions. Therefore, a plausible implementation of process algebraic actions in the face of failure would be to respond to exceptions by triggering a transaction rollback.

Because the failure of an action in one component necessarily leads to the failure of the same action in other components, some component-spanning transaction model is needed. And if components are used as a unit of distribution as sometimes proposed, this transaction model must be a distributed model.

A distributed transaction model used a lot in practice is the X/Open Distributed Transaction Protocol [X/O96]. This defines a *transaction monitor* which controls system-wide transactions. These transactions are distributed over a number of *resource managers* by means of a 2-phase commit (or 2pc) protocol. Aside from their transactional responsibilities, the resource managers also have a responsibility to an *application program*, which manipulates the resources taken care of by the resources managers.

In terms of the component model in Figure 7.3, the resource managers are the life cycle components which may or may not be implemented to save their state in a database. The linking code between the components behaves as the application program, and the transaction monitor is an off-the-shelf program.

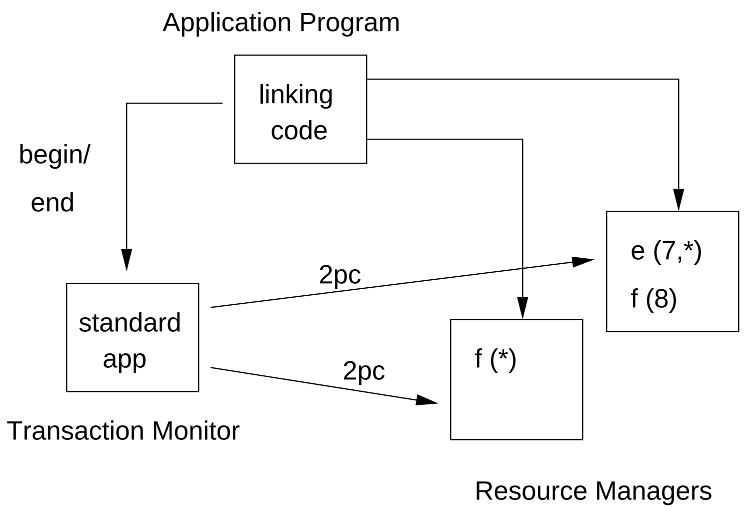


Figure 7.3: Distributed transaction processing.

The X/Open model obviously blends in the component scheme naturally. Given that the components provide transaction support, which may often be automatically generated, the only necessity is that the generic linking code between the components informs a transaction monitor of the start and end of a transaction.

The transaction mechanism is not visible in the object-oriented models, but is instead treated as an *implementation detail*. This pleasant property is due to the elegant communication model provided by the $\|$ operator; it defines joint transitions, which are very close to the concept of a distributed transaction.

Conclusions. Life cycles offer good solutions to some practical problems. Composition of components is well possible as a result of the mathematical foundation of process algebra. Support for transactions, including complex distributed transaction models, is a natural property in much the same way.

7.4 Future work

Our work makes it possible to conduct proofs over a larger range of applications than supported with model checking techniques, but there is still a lot of room for improvement.

Inductive proofs. There are still proofs we cannot perform, including ones that are feasible in theory. One important class of problems is the one that requires inductive proofs. For example, reasoning about 'the people in charge of me' requires induction, because my boss has a boss, who in turn has another boss, and so on, until the executive in charge of my company; proofs about such structures can never be described in general with a fixed, known number of links to traverse, and induction seems to be the natural solution. Our work does not include such an inductive proof approach, but it seems to be a viable extension.

Data types. The life cycle diagrams described in this thesis work have only references to (other) life cycle instances as data types. This kept our approach simple, and allowed us to focus on the proof strategy that we had in mind. Adding more data types, even simple ones such as integers, introduces complex problems that easily require theorem proving powers. Needless to say, additional data types are an improvement of our life cycles, but at the same time they bring about a lot of complexity. Smart techniques [ACD93] may help to solve this complexity, however.

Refinement. We stated early in our work that we find refinement an important asset for a process language, because it is much cleaner than inheritance as a modelling concept. We specifically think of refinement as a relationship between whole components of multiple life cycles, not so much between single life cycles. Although we have gone through lengths to allow easy integration of refinement with our life cycles, we have not actually introduced it yet. Future work should certainly take this path.

7.5 Thesis conclusions

This thesis has introduced the concept of life cycles, from the angles of object process modelling and mathematical process modelling.

1. *It is possible, at least in theory, to conduct automated proofs over systems with unbounded numbers of instances.* This was achieved in Chapter 5 by incorporating quantified annotations in the internal representation of our prover, and handling them with tecniques from theorem proving.

2. *It is possible to prove practically interesting temporal properties on object-styled designs.* Model checking has succesfully been applied to control applications. Control applications mainly distinguish themselves from customary object designs by their fixed number of instances. By generalising this fixed number of instances to an unbounded number, we can verify a much larger class of object designs; sufficient to conclude that common applications can be verified. Chapter 6 and Section 7.3 support this conclusion.

3. *An object process model need not express causality between events to be usable in practice.* As demonstrated in Chapter 6 and Section 7.3, we can handle practical situations, even though Chapter 4 does not introduce causality in the semantics.

4. *When parameters are part of a process algebra, and notably when they can be 'read', the processes should have an alphabet that changes over time, to avoid deadlock in any but the most trivial applications.* This is stated most clearly in Section 7.2, where we stated that deadlock is a great risk if parameters are not modelled explicitly. Life cycles employ a dynamic alphabet to overcome these risks of deadlock.

5. *The controversy between garbage collection and explicit deletion can be resolved for objects whose processes are known and enforced in their implementation.* As described in Section 6.2, it is possible to generate code that automatically deletes objects at the end of their life, provided that the process flow of the object is known and enforced.

The first two conclusions give a posistive answer to the central question for this thesis as formulated in Section 1.1, namely *Can process checking techniques be applied to practical models?* As stated in that section, we have taken object models as representatives of practical models.

Appendix A

In gewoon Nederlands

Het is natuurlijk niet mijn bedoeling om een ondoorgrondelijk proefschrift te schijven. Ik vind het belangrijk om ook aan niet-wetenschappers duidelijk te maken waar ik me tijdens mijn promotie mee bezig heb gehouden. Dat doe ik hier. Ik heb gekozen voor alledaagse voorbeelden, omdat dat veel sprekender is dan wanneer ik op de gebruikelijke wijze termen uit de informatica uitleg: We gaan het spel en de regels van monopoly proberen te beschrijven in relatie tot het onderwerp van dit proefschrift.

A.1 Een berg informatie

Een spel monopoly kun je zien als een grote berg informatie. Bijvoorbeeld de hoeveelheid geld in het bezit van elke speler, wie bezit welke straten, wie is er aan zet, wat is de volgorde van de vakjes op het bord, ... Informatici brengen in die berg informatie graag wat structuur door het in kleinere hoopjes bijeen te vegen, en te kijken hoe die hoopjes met elkaar gerelateerd zijn.

Om de draad niet kwijt te raken is het handig hoopjes informatie te maken voor elk 'ding' in het spel: Een hoopje voor de informatie van elke speler (naam, banksaldo, ...), een hoopje voor elke straat, eentje voor elk huis, elk hotel, ... En dan zijn er relaties tussen de nu overzichtelijke hoopjes informatie: Een speler heeft een pion op een bepaalde straat, een straat kan met hotels bebouwd zijn, ...

Zo, dat ruimt op. Alle informatie is nu gerangschikt en met elkaar in relatie gebracht. Elk hoopje informatie heet in computertermen een *object*, en een relatie heet een *link*. Als je nu voor elk object een kaartje maakt met de desbetreffende links er bij op dan kan het zo in een kaartenbaksysteem worden gestopt; of in de computer natuurlijk.

Helaas zijn de spelregels veel minder overzichtelijk. Als speler Rick met de dobbelstenen 3 gooit, dan zie je op het bord zó wel wat er gebeurt (alhoewel, houd wel je straten in de gaten!) maar in een kaartenbaksysteem moet je toch flink zoeken voor je alle kaarten hebt bijgewerkt. In een computer gaat het net zo als in de kaartenbak, dus ook dat is tamelijk bewerkelijk.

Om dat spitten in de berg kaartjes duidelijker te maken worden ook de beschrijvingen van 'dingen om te doen' in kleinere brokjes beschreven. Die brokjes heten in informaticatermen *processen*. Het is gebruikelijk om een apart proces te beschrijven voor elk object zoals we dat al hadden gevonden.

A.2 Stapje voor stapje

Als we willen weten hoe monopoly er in termen van processen uitziet, dan kunnen we het beste kijken welke losse stapjes je tijdens het spel neemt, en later kunnen we dan kijken welke volgordes van zulke stapjes toegestaan zijn.

Een voorbeeld. Stel, speler Rick gooit 3. Zulke gebeurtenissen schrijf ik in mijn proefschrift op als worp(Rick,3); de *naam van de gebeurtenis*, hier worp, geeft aan wat voor soort gebeurtenis dit is, de rest is aanvullende informatie.

Nou blijft het niet bij die ene gebeurtenis, want er wordt een reeks andere gebeurtenissen ontketend, die denkbeeldig allemaal *tijdens* de worp gebeuren. Misschien is de worp eigenlijk wel een dubbelworp, en dan gebeurt die event gewoon óók, tegelijkertijd. En, de dingen die uit worp(Rick,3) kunnen volgen zijn bijvoorbeeld verlaat(Rick,CoolSingel) (Rick verlaat de Coolsingel) en aankomst(Rick,LeidscheStraat). En als de Leidschestraat eigendom van iemand anders is, dan volgt er ook nog een betalings-gebeurtenis (snif). De stapel gebeurtenissen die elkaar zo veroorzaken vormen samen een elementaire aktie in ons monopolyspel.

Het afhankelijk zijn van andere dingen elders in het spel, is één van de punten waarop mijn aanpak zich onderscheidt van de gangbare processen voor objecten: Ik knoop alle betrokken processen (de lopende speler, het aankomstvakje en de eigenaar daarvan) aan elkaar en laat ze samenwerken. Dat kan ik doen doordat ze eigenlijk allemaal dezelfde gebeurtenissen te zien krijgen, en ze nemen elk op hun eigen houtje actie als gevolg daarvan.

Er zijn best veel gebeurtenissen van belang in zo'n monopolyspel, maar die mogen natuurlijk niet zomaar in elke volgorde gebeuren. Daarom leggen mijn processen, wederom stringenter dan soortgelijke praktische modellen, beperkingen op aan welke gebeurtenissen mogelijk zijn, en in welk volgorde. Bijvoorbeeld, de volgorde van wie er gooit moet netjes vastgelegd worden, eerst Rick, dan Maarten, en dan weer overnieuw. Een ander voorbeeld is als Maarten in de gevangenis zit, die mag daar pas uit als hij dubbel gooit. Dus als Maarten in de gevangenis zit dan is hij in een speciale toestand, dus worp(Maarten,6) zonder gelijktijdige dubbelworp mag nooit samengaan met verlaat(Maarten,Gevangenis) (aannemende dat er niet betaald is, of een kaart 'verlaat de gevangenis zonder betalen' is ingezet).

Ik heb een notatie bedacht waarin je al dat soort dingen kunt opschrijven. Een proces wordt getekend in een diagram met een rechthoek eromheen en de naam bovenin; afgeronde rechthoeken geven de 'toestanden' van het proces weer, en de pijlen ertussen geven aan hoe een gebeurtenis een proces van de ene toestand in de andere brengt.elkaar Het bijzondere van mijn notatie zit in de notaties bij de pijlen, en hun precieze betekenis.

A.3 De eerste stapjes zijn ook gewoon stapjes

De processen in mijn systemen reageren op gebeurtenissen, en verder is er niets. Maar soms moet een nieuw proces (plus object, of hoopje informatie) worden aangemaakt voor bijvoorbeeld een nieuw huis dat je aanschaft, of als een speler failliet gaat dan moet zijn object worden weggegooid.

Dat is in mijn processen simpel maar ook krachtig opgelost: Het aanschaffen van een nieuw huis is voor de meeste processen een gewone gebeurtenis, maar voor huizen is het speciaal, want het maakt een nieuw huis aan. Alle gebeurtenissen zijn dus gewone gebeurtenissen, maar processen behandelen bepaalde gebeurtenissen speciaal omdat die als eerste of laatste in een proces genoemd staan.

Ook in deze aanpak onderscheid ik me van bestaande praktische notaties. Dat is echter bittere noodzaak. Als ik processen aan elkaar wil knopen en er ook nog dingen over wil bewijzen, dan kan ik maar beter zorgen dat ook het aanmaken van een nieuw proces gewoon meedoet in de processen.

Overigens interessant op te merken is dat mijn processen aan reïncarnatie doen: Als een huis wordt verwijderd (bijvoorbeeld bij inruilen voor een hotel) dan gaat'ie terug in de doos, klaar voor de volgende aankoop, en netzo gebeurt het ook met mijn processen.

A.4 Waarom het zo precies moet

Als ik een weddenschap of een spelletje monopoly met Maarten aanga, dan gaat het niet om onbelangrijke dingen als geld of goud, maar dan gaat het om dingen van Groot Belang, zoals marsrepen. Na een poosje op die manier Maarten's secundaire arbeidsvoorwaarden verzorgd te hebben werd ook mij duidelijk dat het belangrijk was de spelregels heel goed vast te leggen, opdat het allemaal eerlijk zou verlopen. Ook kinderen hebben daar baat bij, ze gaan anders vechten.

Hoe vreemd het ook mag klinken, soortgelijke problemen en ruzietjes komen bij het maken van een groot ontwerp in software evenzeer voor, alleen dan met een stropdas voor. Het probleem blijft dat onduidelijkheid leidt tot misverstanden, en soms kunnen die duur uitpakken. Dus ook in de informatica is het wel zo handig om precies te weten waar je het over hebt. Het is dan ook verrassend te merken dat de meeste informatici schuw zijn van precisie als ze diagrammen tekenen.

Voorbeelden van dingen die door gebrekkige precisie fout kunnen gaan schuilen vaak in de uitzonderingen en de randgevallen. Mag speler Rick bijvoorbeeld de Kalverstraat kopen als speler Maarten die al gekocht heeft, en zo ja, moet Rick dan nog huur betalen aan Maarten? Dat soort dingen worden duidelijk beantwoord in de wiskundige versie (omdat die van nature compleet is) maar staat het ook in de papieren regels? Die gaan er van uit dat je weet wat 'in bezit van' betekent. Allemaal mooi en wel, maar een computer heeft dit soort algemene ontwikkeling niet, en daarvoor moet het dus allemaal expliciet worden opgeschreven.

Natuurlijk zijn er al heel lang wiskundigen die werken aan precisie, maar die verliezen soms de praktische waarde van een theorie uit het oog, omdat de problemen daarzonder ook al moeilijk genoeg zijn. Er waren dus twee werelden die bezig waren met processen, maar ze kwamen maar niet bij elkaar. Ik heb dat met deze promotie voor elkaar gekregen, door op de kennis en ervaring in die twee werelden voort te bouwen. Ik heb een praktisch bruikbare notatie bedacht, en daar heb ik een wiskundig precieze definitie van gegeven, zodat de informaticus diagrammen kan tekenen met een precies vastgelegde betekenis.

A.5 Soms weet je het gewoon niet

We kunnen nu al veel van het monopolyspel in processen en objecten vastleggen, maar we moeten ook weer niet te ver gaan. Bijvoorbeeld, de acties ten gevolge van het trekken van een kanskaart zijn totaal willekeurig, en daardoor kun je gewoon niet zeker zijn wat er gebeurt als gevolg van het trekken van een kanskaart.

Wiskundigen kennen dit probleem al heel lang, dat een keuze valt, maar het is onbekend welke. De term daarvoor is *non-determinisme*, en het is in de informatica nog niet gebruikelijk om zulk bekend gebrek aan kennis op te schrijven.

Omdat ik de computer bewijzen wil laten leveren, moet ik dat hier wel doen. Want een bewijs is niet veel waard als het alleen geldt wanneer 'ga verder naar Kalverstraat' boven op de stapel ligt. Je wilt juist dat een bewijs altijd geldt, ongeacht wat de volgende kanskaart zal zijn!

A.6 Bewijzen hoe het zit

Het ontwerpen van een complex stuk software blijft mensenwerk, en er kunnen fouten in komen. Het is daarom nuttig om checks in te bouwen om te zien of alles wel klopt, vergelijkbaar met de balans die moet kloppen in een boekhouding. Zijn de uitkomsten daarin ongelijk, dan heb je ergens een fout gemaakt.

Ook voor ontwerpen kun je controles inbouwen, maar dan niet met getallen. Processen zijn gelijktijdige gebeurtenissen, en daarvan meerdere in bepaalde volgordes. En elk daarvan kan iets veranderen aan de status en gegevens van een programma.

Het handige aan de wiskunde is niet alleen dat het zo precies is, maar bovendien kun je een wiskundige uitspraak op allerlei manieren anders opschrijven zonder dat het opeens niet meer klopt, en als je geluk hebt kun je daar zelfs zo ver mee gaan dat je een gewenste eigenschap kunt bewijzen. Als dat fout loopt, dan heb je waarschijnlijk een fout in je ontwerp of in je eis zitten, net als bij de balans die niet klopt. Ik doe dit in mijn onderzoek met een bewijsprogramma, zeg maar een wiskundige-in-een-doosje, zodat niemand het moeizame bewijswerk met de hand hoeft te doen maar dat het aan de computer kan worden overgelaten, want anders zou (bijna) niemand er mee willen werken.

A.7 Verfijnde kneepjes

Een extra winstpunt van een wiskundige ondergrond is dat er een heleboel zinnig onderzoek ter beschikking komt voor praktische toepassingen. Zo is er bijvoorbeeld de mogelijkheid om globaal beschreven stappen in een proces in meer detail uit te werken; in de wiskunde heet dat *refinement*.

In het monopolyspel zien we dat ook terug. Bij de actie worp(Rick,3) gaat Rick's pion 3 plaatsen vooruit, maar eigenlijk gebeurt dat in drie losse stapjes achter elkaar, en samen noem je dat dan 3 plaatsen vooruit gaan. De uitwerking is dan iets als stapje(Rick,Spui,Plein), dan stapje(Rick,Plein,Waterleiding) en tenslotte stapje(Rick,Waterleiding,LangePoten). Aan de eerste plak je dan bovendien de gebeurtenis verlaat(Rick,Spui) vast, en aan de laatste aankomst(Rick,LangePoten).

Dit is voor een simpel geval als dit nog wel te overzien, maar ook hier geldt dat mensen fouten kunnen maken als het complexer wordt. Net als je allerlei algemene eigenschappen kunt testen met wiskunde, kun je ook kijken of een bepaalde detailuitwerking wel klopt met de regeltjes van voor de detaillering, tenminste als je de uitwerkingsregeltjes kent. En dat treft: Die regeltjes, daar hebben wiskundigen al veel werk in gestoken en daar kun je dus de vruchten van plukken.

Refinement is voor complexe ontwerpen van groot belang. De meeste ontwerpmethoden passen iets dergelijks toe, maar dat lijkt al erg veel op programmeren, zodat er opeens allerlei extra details nodig zijn, en sommige van die extra details zijn ongewenst omdat ze de aandacht van het ontwerpen afleiden. Dat is niet de bedoeling, omdat je dan het overzicht weer kwijt raakt, en daar was het nou juist allemaal om begonnen!

A.8 Alles of niets

Het zal inmiddels duidelijk zijn, de simpele gebeurtenis worp(Rick,3) leidt tot een hoop andere gebeurtenissen die op allerlei manieren verweven kunnen zijn. Soms kom je er halverwege een worp achter dat je de geplande acties toch niet had kunnen doen. Denk maar aan een speler die een fout heeft gemaakt, of die blijkt niet te kunnen betalen voor een hotel dat al op zijn straat gezet is. De beste oplossing is dan om de gehele worp terug te draaien. Niet alleen een paar, maar *alle* gelijktijdige gebeurtenissen.

Dit kan in een computerprogramma met iets dat als *transactie* bekend staat. Transacties zijn afkomstig uit de wereld van de *databases*, en dat zijn eigenlijk elektronische kaartenbaksysteem, dus daar kan alle informatie in worden opgeslagen; zulke systemen laat ik ook automatisch construeren door de computer; dat kan dankzij de wiskundige precisie van de processen. Transacties zeggen (onder andere) dat een stapel veranderingen van de informatie helemaal wel, of helemaal niet gebeurt; het is alles of niets.

Het is kenmerkend aan mijn werk hoe gemakkelijk transacties er in passen. Alternatieve aanpakken werken niet met gebeurtenissen die tegelijkertijd optreden, en dat bemoeilijkt dit enorm. Door mijn eenvoudiger (maar even krachtige) aanpak voorkom ik een hoop problemen waarvoor niet echt een nette oplossing bestaat.

A.9 Volledigheidshalve. . .

Het is al heel veel werk om bewijzen te kunnen leveren voor processen die links naar elkaar hebben. Ik heb me in mijn onderzoek daartoe beperkt. Bijvoorbeeld het rekenen aan de prijzen die je moet betalen als je op een straat met een hotel aankomt, dat beschrijven mijn processen niet. De bedragen kun je gewoon doorgeven bij een gebeurtenis, maar mijn processen rekenen niet aan die waarden, ze geven ze alleen maar door.

In onderzoek maak je vaak zulke versimpelingen, iemand anders kan er later immers op voortbouwen. In wezen is dat wat ik ook gedaan heb, ik heb heel veel bestaande wiskunde en informatica kunnen gebruiken en voeg daar een stukje van mezelf aan toe.

Appendix B

Notational conventions

This thesis takes a bit of freedom with respect to notation. The following is a list of variations on well-known syntaxes. The non-standard notations were used for æsthetical reasons, to avoid confusion with notation used elsewhere and to aid readability. Only the basic forms are shown below; derived forms receive the same treatment.

Context	**Standard notation**	**Thesis notation**
μ-calculus	$\lambda R\ [\ e\ ]$	$\lambda R \cdot e$
	$\exists x\ [\ e\ ]$	$\exists x \cdot e$
	$[\ x = y\ ]$	$x = y$
	$R(x_1, \ldots, x_n, y_1, \ldots, y_m)$ where R maps $x_1 \ldots x_n$ to $y_1 \ldots y_m$	$R(x_1, \ldots, x_n \mapsto y_1, \ldots, y_m)$
CTL	$E\ [\ e\ U\ f]$	$e\ EU\ f$
Buddhism	Isthmus	Nirvana

Appendix C

Samenvatting

Model Checking uitgebreid naar Object Proces Validatie.

Model checking is een vakgebied dat zich richt op het verifiëren van procesmodellen in onderlinge samenhang. De gebruikelijke werkwijze hierbij is dat alle mogelijk voorkomende momenten in de loop van een 'run' van deze procesmodellen wordt afgelopen. Dit leidt onherroepelijk tot een explosie van toestanden waarin het samengestelde proces zich kan bevinden, en er is dan ook veel onderzoek gedaan naar het optimaliseren van het model check proces.

Model checking is ontstaan uit verificatie van hardware, en hoewel de principes zich lenen voor het verifiëren van een beperkte klasse software (met name regelsystemen), is het ongeschikt voor algemene systemen, omdat daarin het aantal instanties van processen onbeperkt is. Een model dat alle mogelijke executie-toestanden bevat kan daar niet mee omgaan. Om die reden breiden we de gebruikelijke aanpak van model checking hier uit, met als doel het af kunnen handelen van systemen met een onbeperkt aantal proces-instanties.

De aanpak die we hebben gekozen is het uitbreiden van de interne modellen van een checker met de manier waarop theorem provers omgaan met quantoren. We staan daarbij toe te quantificeren over instanties en over alternatieve executies (non-determinisme). Deze minimale uitbreidingen, plus enige triviale constructies om elementaire tests uit te kunnen voeren, bleken afdoende om een krachtige procestaal te definiëren voor gebruik tijdens object modellering: De life cycles die als rode draad door dit proefschrift lopen.

Zoals dit werk laat zien, is het mogelijk met life cycles zinnige, praktische constructies te modelleren. Deze constructies worden alle vertaald naar een eenvoudige taal die gebaseerd is op de μ-calculus. Dit proefschrift laat zien dat over deze constructies vrij goed bewijzen kunnen worden gevoerd. Tenslotte geeft dit proefschrift aan hoe foutlopende bewijzen kunnen worden gebruikt om feedback te leveren aan de ontwerper, zonder daarbij per sé tot wiskundig detail te vervallen.

Voor een uitgebreidere, metaforische inleiding in mijn werk verwijs ik naar Appendix A.

Bibliography

[AB96] W.M.P. van der Aalst and T. Basten. Life-cycle inheritance: A Petri-net-based approach. Computing Science Report 96/06, Eindhoven University of Technology, Department of Mathematics and Computing Science, Mar 1996.

[AB97] W.M.P. van der Aalst and T. Basten. Life-cycle inheritance: A Petri-net-based approach. In P. Azæcutema and G. Balbo, editors, *Application and Theory of Petri Nets*, volume 18, pages 62–81. Springer-Verlag, June 1997.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and computation*, 104:2–34, 1993.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Reading, MA, USA, third edition, 2000.

[BA96a] T. Basten and W.M.P. van der Aalst. Inheritance of dynamic behavior: Development of a groupware editor. Computing science report, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, 1996.

[BA96b] T. Basten and W.M.P. van der Aalst. A process-algebraic approach to life-cycle inheritance: Inheritance = encapsulation + abstraction. Computing Science Report 96/05, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, March 1996.

[BCCZ] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. Out of band communication.

[BCM$^+$90] J.R. Burch, E.M. Clarke, K.K. McMillian, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, pages 428–439, June 1990.

[Bee94] M. von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W.P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148. Springer, 1994. Lecture Notes in Computer Science 863.

[Bla96] J.O. Blanco. *The State Operator in Process Algebra.* PhD thesis, Eindhoven University of Technology, 1996.

[Bro95] Fredrick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition.* Addison Wesley, Reading, Mass., second edition, 1995.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[Cas97] G. Castagna. *Object-Oriented Programming — A Unified Foundation.* Birkhäuser Boston, 1997.

[Cor97] Rational Software Corporation. *UML Semantics.* 1997.

[Dav87] W.V. Davies. *Egyptian Hieroglyphs.* Reading the Past. British museum press, 1987.

[DNFGR93] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems. *Computer networks and ISDN systems*, 25:761–778, 1993.

[DNV90] R. De Nicola and F. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of systems of concurrent processes: ... Proceedings*, volume 469 of *LNCS*, pages 407–419, Berlin, 1990. Springer.

[DW98] D. D'Souza and A. Wills. *Catalysis: Practical Rigor and Refinement.* Addison-Wesley, 1998.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[Esh98] H. Eshuis. Refinement in object-oriented analysis and design. Master's thesis, University of Twente, Faculty of Computer Science, July 1998.

[EVD89] P.H.J. Eijk, C.A. Vissers, and M. Diaz, editors. *The formal description technique LOTOS*. Elsevier Science Publishers B.V., 1989.

[GBHS97] I. Graham, J. Bischof, and B. Henderson-Sellers. Associations considered a bad thing. *Journal of Object-Oriented Programming*, pages 41–48, Feb 1997.

[GHJV86] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison–Wesley, 1986. ISBN: 0–201–63361–2.

[Gla86] R.J. van Glabbeek. Notes on the methodology of CCS and CSP. Technical report, Centre for Mathematics and Computer Science, PO Box 4079, 1009 AB Amsterdam, the Netherlands, Aug 1986.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems — An Approach to the State-Explosion Problem*. LNCS 1032. Springer, 1996.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HG97] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[HO93] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). *OOPSLA*, pages 411–428, 1993.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hol97] G.J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[ISO94] ISO, editor. *Ada 95 Reference Manual, Language and Standard Libraries.* US DoD, 1994. ISO/IEC 8652:1995(E).

[Jan47] J.M.A. Janssen. *Hiëroglyphen: Over Lezen en Schrijven in Oud-Egypte.* Leiden, 1947.

[KM89] R.P. Kurshan and K. McMillian. A structural induction theorem for processes. In *ACM Symposium on Principles of Distributed Computing*, pages 239–247, 1989.

[KØ96] B.B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.

[Kri94] G. Kristen. *Object Orientation: The KISS Method: From Information Architecture to Information System.* Addison-Wesley, 1994.

[LH89] K.J. Lieberherr and I.M. Holland. Assuring good style for object-oriented programs. *IEEE software*, pages 38–48, Sep 1989.

[LM95] D. Lea and J. Marlowe. Interface-based protocol specification of open systems using PSL. *Lecture Notes in Computer Science*, 952:374–398, 1995.

[LMM99] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. *Formal Methods for Open Object-Based Distributed Systems*, pages 331–347, 1999.

[Ltd88] Inmos Ltd. *Occam 2 Reference Manual.* Prentice-Hall, 1988.

[McC97] B. McCarthy. Associaten inheritance and composition. *Journal of Object-Oriented Programming*, pages 69–72,74–77,80–81, Jul/Aug 1997.

[Mil80] R. Milner. *A calculus of communicating systems.* LNCS. Springer-Verlag, 1980.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. In *Proceedings of the International Summer School on Logic and Algebra of Specification*. Springer-Verlag, Aug 1991.

[Nie93] O. Nierstrasz. Regular types for active objects. In *Proceedings OOPSLA'93*, volume 28 of *ACM SIGPLAN Notices*, pages 1–15. ACM Press, October 1993.

[Nie95] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995.

[OMG95] OMG, editor. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 1995.

[PT97] B.C. Pierce and D.N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI Technical Report #476, Indiana University, Mar 1997.

[Rei97a] R. van Rein. Life cycles in quantum. Technical report, University of Twente, Faculty of Computer Science, 1997. Internal report.

[Rei97b] R. van Rein. Refinement of transactional protocols in quantum. Technical report, University of Twente, 1997.

[Rei99] R. van Rein. Protocol-safe workflow support for Santa Claus. In A. Vallecio, J. Hernández, and J.M. Troya, editors, *ECOOP'99 Workshop on Object Interoperability*. Universidad de Málaga, Dpto. Lenguajes y C. de la Computación, 1999.

[Rei00] R. van Rein. On practical verification of processes. In J. Hernández, A. Vallecio, and J.M. Troya, editors, *Proceedings of the ECOOP'2000 Workshop on Object Interoperability*, pages 25–32. Universidad de Extremadura, Dpto. Informática, 2000.

[RF99] R. van Rein and M. Fokkinga. Protocol assuring universal language. *Formal Methods for Open Object-Based Distributed Systems*, pages 241–258, 1999.

[RG99] A. Rensink and R. Gorrieri. Action refinement as an implementation relation. In *LNCS*, volume 1214, pages 772–786. Springer, 1999.

[Ros95] S. Rosmorduc. A LaTeXperiment of hieroglyphic typesetting. Technical report, Ecole Normale Superieure de Cachan, May 1995. CTAN/fonts/hieroglyph.

[SGW94] B. Selic, G. Geullekson, and P.T. Ward. *Real-time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.

[Spi92] J.M. Spivey. *The Z notation: a reference manual (2nd edition)*. Prentice Hall International, UK, 1992.

[Str98] D.E. Strijbos. Global aspects of system protocols. Master's thesis, University of Twente, Faculty of Computer Science, December 1998.

[Szy98] C. Szyperski. *Component software: beyond object-oriented programming*. ACM Press Books. Addison-Wesley, 1998.

[Tan95] C. Tanzer. Remarks on object-oriented modeling of associations. *Journal of Object-Oriented Programming*, pages 43–46, Feb 1995.

[Tel00] P.M. Telford. *Famous Pharaohs*. http://www.skittler.demon.co.uk/pharaohs.htm, Feb 2000.

[X/O96] X/Open, editor. *Distributed Transaction Processing: Reference Model, Version 3*. Feb 1996.

Index