Specifying Processes with Dynamic Life Cycles

Rick van Rein

University of Twente, Dept INF
PO Box 217, NL-7500 AE Enschede, the Netherlands
vanrein@cs.utwente.nl

Abstract. We propose an alternative notation and semantics for process models in object analysis, to resolve problems with current diagram languages. Our dynamic life cycles are communicating state diagrams. Our life cycles support polymorphic creation, and they are straightforwardly composed. We provide an operational semantics, and demonstrate how to interact with a system of life cycles.

Keywords. object orientation, polymorphic creation and deletion, conceptual modelling, communication, state diagrams, life cycles, process algebra, component composition.

Introduction

Modern systems analysis and conceptual modelling is performed with object oriented methods, which offer rich notation to capture the constraints on implementations. An often-used notation is OMT, increasingly replaced with UML. The most-used models in these languages are *class diagrams* (which express classes with associations between them) and *state diagrams* (which express dynamic aspects of classes).

An example of a class diagram in OMT notation is found in figure 1, which shows a library (example taken from the analysis course on Catalysis [5]; Catalysis is an analysis and design method exploiting the notation of UML). This library registers *books*, which may be lent by *members*. A *book* has a *title*, against which *reservations* can be made.

Class diagrams have evolved from entity relationship diagrams [7], which explains why they represent data aspects but no process aspects. A result of the emphasis on data aspects in object oriented analysis is that associations lack a deeper meaning than "two connected classes." Many designs do not even name associations!

In the design in figure 1, the *lentby* association optionally refers to a *member*, but no knowledge is added on when a *member* is found over that association. Furthermore, the meaning of the association can only be found by interpreting the word 'lentby' — a common source of misunderstanding. Similarly, the class diagram does not reveal whether the *lentby* and *heldby* links can lead to an object at the same time.

B. Wangler, L. Bergman (Eds.): CAiSE 2000, LNCS 1789, pp. 192–211, 2000.

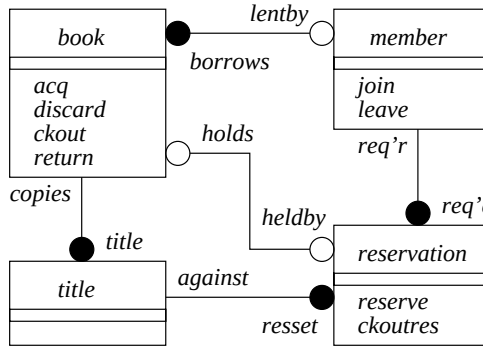

Fig. 1. Class diagram for a library system.

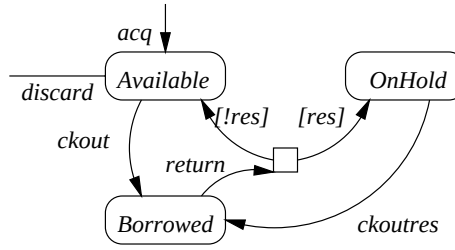

Fig. 2. State chart for a book in the library.

To express such dynamic aspects of the design in an object oriented way, a state diagram is usually drawn, like the one in figure 2, and some methods (like Catalysis) add mutually exclusive predicate expressions to the states, to hold for all *b:book*:

$$\begin{aligned} b.Available &= (b.lentby = nil) \wedge (b.heldby = nil) \\ b.Borrowed &= (b.lentby \neq nil) \wedge (b.heldby = nil) \\ b.Onhold &= (b.lentby = nil) \wedge (b.heldby \neq nil) \end{aligned}$$

This *does* answer our previous questions. A caveat here is the use of the value *nil*, which always introduces danger of loss of referential integrity; luckily, the relation between *nil* values and states makes it possible to check the models for *nil* dereferencing.

The state diagram in figure 2 demonstrates another place where a predicate expression can be of use, namely when the *return* operation is applied to a *book;* in this case, some condition (mentioned as *res* here) specifies whether or not the *book* is put on hold for some *reservation*. The disadvantage of this approach is that the *book* must be aware of the existence of *reservations*. It would be better for extensible designs when *books* are just entities with an *Available* and a *Borrowed* state, processing *ckout* and *return* operations, and not minding about *reservations*. Unfortunately this is not possible because a *ckout* for a reservation (distinguished here by naming it *ckoutres)* poses additional constraints and also terminates the resolved *reservation*.

Another class of process related problems in object oriented models is how to express synchronisation; for example, not enabling a *member* to *leave* the library (membership) before his *lends* set is empty. These problems are solved with preconditions on the *leave* operation in Catalysis.

Predicate logic expressions are powerful tools for object modelling, but not without a downside; free-form quantifications over sets make automatic model checking very hard and in general impossible since all possible runtime occurrences of sets cannot be foreseen at design time. Design specific ('free form') predicate expressions introduce the need for runtime checking, which has a negative impact on runtime performance.

The remainder of this paper is dedicated to introducing an (object) process model that contains more information than current life cycle models; the presentation is structured as follows: section 1 introduces and defines the concepts of life cycles by showing solutions to typical problems; section 2 shows a number of practical applications of the life cycle concept; finally, we draw conclusions and look forward to future research.

This work 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.

Our approach. In this paper we introduce our notion of *(dynamic) life cycles*, as a variation on state diagrams. Life cycles can be used to express the process view on a class or component. They are capable of communicating with each other, and to create and destruct themselves on the fly. Life cycles pass around identities of life cycle instances (called *lifes*, the process view on objects). We choose not to model nesting, direction, or fixate the number of participants for operations; instead, we model an event-like scheme with the possibility to extend the set of event participants in several ways. Our events interact synchronously, so that lifes have the capability to block events.

We define the operational semantics for our life cyclesin the functional programming language Miranda[1] [3]; it is a precise, high-level language that hardly forces us to over-specify, it is executable and relatively easy to read.

1 The Concept of Life Cycles

This section introduces the concept of (dynamic) life cycles, by showing which problems exist, and how they are solved by life cycles. Each problem is introduced with an example, and guides the definition of the graphical syntax and an operational semantics for life cycles.

1.1 Undirected Events

Problem. Events mentioned in state diagrams can be related to 'the real world' in a number of ways. State charts by Harel [9] and in UML [16] usually have a

[1] Miranda is a trademark of Research Software Ltd.

notion of events coming from outside the state charts' domain. To provide state charts with the ability of initiating further actions, they can send out events. This approach is also taken in ROOM [19] and in our own protocol checking work [17].

This approach is not free of problems. Firstly, it is often debatable [12] [2] what the semantics are if a multitude of events arrives at the same time, and whether a nesting of events is allowed. It seems that no reasonable choice covers all cases of practical interest. We believe that the problem is that the event-sends-event structure reveals too many details of implementation in programming languages, notably C++.

Discussion. We believe that this problem is caused by the imperative idea of events calling each other. To model all aspects of these calls well, the models either contain a lot of complicating detail, or fail to be sufficiently precise for verification by automatic tools. In contrast, process-algebras [1] are simpler and express similar information. In these models however, events are just atomic actions that communicate, with no distinction between caller and callee.

Solution. Our choice is to simplify the semantics of events in state charts, by removing knowledge of the direction of event communication. Instead, we observe something happening in the 'real world' (what we will henceforth call an *event occurrence*), and our life cycles define *events*, which are system-perspectives of event occurrences.

We define an *event* as a name plus a list of parameters (introduced in subsection 1.4). We define event occurrences *evocc* as a set[2] of events. This makes it possible to model complicated event occurrences that can be matched by different events in different places:

$$event \equiv (name, [param])$$
$$evocc \equiv [event]$$

Based on these definitions, we define a *lifecycle* as a triple containing a name, a set of states (which we define as simple names, unique within a *lifecycle*), and a set of *transition*s (which we introduce in subsection 1.6):

$$lifecycle \equiv (name, [state], [transition])$$

Where we leave the types *name* and *state* unspecified, and defer the definition of *transition* to later.

Example. In figure 3, we show a simple system comprising of two life cycles. An event occurrence in this system is [*acq*], which matches the event *acq* in *book* but is ignored in *members*. Another event occurrence is [*ckout*], which is matched by an event *ckout* in both *member* and *book* life cycles and which therefore is a

[2] Miranda does not support sets as language primitives; we will therefore use lists without taking order or multiplicity of elements into account.

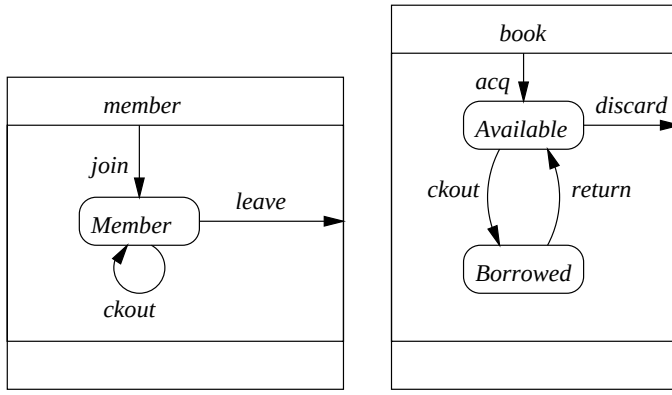

Fig. 3. Library in normal state diagram notation.

step that the two life cycles make at the same time. Though not useful in this application, it would be correct to consider $[acq, leave]$ as an event occurrence, which is matched by *acq* in *book* and by *leave* in *member*. Instead of a nested calling structure between events, the events communicate by observing the same event occurrences at the same time.

1.2 Dynamic Communication Structures

Problem. Modern software development aims for modular (or 'component-based') development, in the hope that modules can be composed after they have been separately built. This makes it necessary to construct modules that can dynamically modify their patterns of interaction with the environment.

A problem encountered in implementing this wish is the strict pairing of caller to callee in most implementation languages. This style of specification makes it hard to add or remove a party without adding or removing its communication partners at the same time.

Discussion. An elegant and general scheme for communication is one where *any number* of parties may participate in an event. When a party wants to participate in an event, it cannot be ignored just because it currently is in a wrong state for participation. So, when a life cycle (like *book* in figure 3) contains the operation *return*, it can block that operation on other life cycles if it is not yet ready to participate in it. However, if a life cycle (like *member*) does not contain the *return* event, it simply ignores occurrences of that event.

The ideas of synchronisation (or 'blocking') described here are well known in concurrency theory. It is found in LOTOS [6] [4] as `||`, and in ACP [1, §4.2], CSP [10] and CCS [14] as ||. Collaborative processing of events is known in programming languages as *rendez-vous synchronisation.* It is found in the languages Ada [11] and in Occam [3] [13]. Furthermore, there seems to be a correspondence between event occurrences and distributed transactions [21].

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

Solution. To be able to talk about acceptance or blocking of event occurrences, we introduce a small algebra in which the interesting values are:

- *Ignore* to denote that an event (occurrence) is ignored;
- *Match xs* to denote that an event (occurrence) is accepted provided that the unifications in *xs* are established;
- *Block* to denote that an event (occurrence) is blocked and therefore may not take place.

The algebra provides a type *acceptance* with these values and a *compose* operation to compose lists of such values:

$$
\begin{aligned}
&acceptance ::= \ Ignore \mid Match\ [(arg, arg)] \mid Block\\
&isIgnore = (=\ Ignore)\\
&isMatch(Match\ b) = True\\
&isMatch\ x = False\\
&isBlock = (=\ Block)\\
&compose :: [acceptance] \to acceptance\\
&compose\ xs \ = Block, && \text{if } \exists x \in xs @ isBlock\ x\\
&\qquad\quad = Match\ [v \mid (Match\ vs) \leftarrow xs; v \leftarrow vs], && \text{if } \exists x \in xs @ isMatch\ x\\
&\qquad\quad = Ignore, && \text{otherwise}
\end{aligned}
$$

Using this *acceptance* algebra, we can define a function that indicates how two events match. This function uses the notion of parameters in events, which are introduced in subsection 1.4; in short, a parameter can either be an *In* or *Out* value with the usual meaning, or it can be a *Force* value which means that an event with a different value for that parameter *Block*s if it matches on the non-*Force*d parameters.

The *eventmatch* function returns *Ignore* if there is a discrepancy between the types of the two events. Note that *Out* parameters are treated as part of the event name. If the events do not *Ignore* each other, they should *Match* or *Block*; the latter case occurs when a *Force* parameter (introduced in subsection 1.4) has a wrong value. When *Match* is returned, variable bindings due to parameters are returned as well. So:

$$
\begin{aligned}
&eventmatch :: event \to event \to acceptance\\
&eventmatch\ (n,p)\ (n',p') \ = Ignore, && \text{if } n \neq n' \vee \#p \neq \#p'\\
&\qquad\qquad\qquad\quad = Ignore, && \text{if } \exists x \in matchers @ mismatch\ x\\
&\qquad\qquad\qquad\quad = Block, && \text{if } \exists x \in unification @ blocking\ x\\
&\qquad\qquad\qquad\quad = Match\ unification, && \text{otherwise}
\end{aligned}
$$

where

$$
\begin{aligned}
&matchers = zip(p, p')\\
&mismatch((Out, Bound\ v), (Out, Bound\ v')) = v \neq v'\\
&mismatch((d, w), (d', w')) = False\\
&unification = [(w, w') \mid ((d, w), (d', w')) \leftarrow matchers]\\
&blocking((Bound\ v), (Bound\ v')) = v \neq v'\\
&blocking(w, w') = False
\end{aligned}
$$

Example. With the above definition of *eventmatch*, it is possible to verify the examples in the previous subsection. Since we have not yet considered parameters, we will set the lists of parameters to [], the empty list. The event *acq* is modelled formally as ("`acq`", []) and we find that *eventmatch* ("`acq`", []) ("`acq`", []) = *Match* [] and *eventmatch* ("`acq`", []) ("`ckout`", []) = *Ignore*. We generalise this notion to the matching with event occurrences in subsection 1.6.

The problem of allowing dynamically changing numbers of collaborators on an event occurrence can be solved with the *eventmatch* definition: Those events that *Ignore* an occurring event will not take part in it; those that *Block* cause the whole event occurrence to *Block* (thanks to the *compose* operator) and if this does not happen, all events that *Match* take part in the occurring events.

1.3 Creation and Destruction of Lifes

Problem. A running life cycle system must support instances, which we call *lifes*: some *books* in our library will be *Borrowed,* while others are *Available*. To keep track of different books, life cycles must be instantiated to *lifes*. Only for a life is it sensible to speak of its current state, and a life is what responds to event occurrences, or blocks them.

Some common problems related to creation and destruction of lifes or objects are polymorphic creation of a new instance (often solved with factories [8]) and when to destruct a life or object (often solved with garbage collection [22]).

Discussion. We prefer to solve creation and deletion of lifes without explicit operations. We prefer creation and destruction to be a response of a life cycle system to a plain event occurrence, because this hardly requires additional life cycle concepts.

Process algebras such as ACP [1, table 35], which model concurrent processes, silently remove a finished process; this is part of the design of the concurrency operators. There is also a process algebra, namely the π-calculus [15], which introduces a similarly elegant creation construct: a process P may be prefixed with a replication operator, with the interesting property that $!P = P||!P$, which is any number of P processes interleaved. This appears like an infinite-sized pool of instances of P being ready to get created at the first communication with one of the Ps.

The π-calculus approach to process creation is generally considered elegant because it eliminates the requirement of explicit creating operations. Creation is just an effect of normal communication in the π-calculus.

Solution. To handle instances of life cycles, which we call *lifes*, we define a *life* to be a quadruple with its *lifecycle*, an identity that is unique for that life cycle, a current *state* taken from the life cycle, and a binding relation for currently known variables (treated in subsection 1.4):

$$life \equiv (lifecycle, lifeID, state, binding)$$

The type *lifeID* is only required to support an equivalence relation; in our examples, we assume that *lifeID* is represented as *num*. This paper works only with

life cycles with one creating transition, and these are mutually exclusive, so that for each life cycle, at most one instance is created during any event occurrence.

We define a global system state *sys* as a set of life cycles and a set of lifes that instantiate those life cycles:

$$sys \equiv ([lifecycle], [life])$$

We define creation and deletion as actions on such a *sys*. Later restrictions will ensure that only one instance at a time can be created for each life cycle. We already exploit this knowledge here and create one *life* for every *lifecycle* in a system, exploiting a special state *nirwana* that we reserve to represent the state of a life before its creating transition(s) or after its destroying transition(s):

$$nirwana :: state$$
$$nirwana = \text{“}\texttt{Nirwana}\text{”}$$
$$inNirwana :: life \to bool$$
$$inNirwana(lc, lid, st, b) = (st = nirwana)$$

The *nirwana* state is useful, because it enables the creation of non-existent lifes in a pre-natal state in the following *birth* function, and destroys outlived and never-used lifes in the *death* function:

$$birth, death :: sys \to sys$$
$$birth(lcs, ls) = (lcs, ls +\!\!+ [(lc, nid\ lc, nirwana, [(this, nid\ lc)]) \mid lc \leftarrow lcs])$$
where
$$nid\ lc = id\_not\_in\ [lid \mid (lc', lid, y, z) \leftarrow ls; lc' = lc]$$
$$death(lcs, ls) = (lcs, ls \mathbin{--} [l \mid l \leftarrow ls; inNirwana\ l])$$

We do not specify *id_not_in*, except that for all *xs* it holds that $(id\_not\_in\ xs) \notin xs$.

Example. The *birth* and *death* functions are applied just before and after a basic step of a life cycle system. This means that for every lifecycle, there is always a life ready to be instantiated in response to an event occurrence. Similarly, a life can engage in an event occurrence which terminates its life, and it would be implicitly cleaned up by the *death* function.

To demonstrate the conceptual simplicity of our approach to creation, imagine adding a (redundant) constraint to our library, to express that every *ckout* of a *book* must be followed by a *return* of that *book*. This can be done by adding the life cycle in figure 4. This creates a new life on the fly, as a side-effect of the occurrence of a plain *ckout* event. The *ckout* event receives no special treatment anywhere else than on the creating arrow in this new life cycle; deletion is similar.

1.4 Parameters and Variables

Problem. If we instantiate numerous lifes from the *book* life cycle, we do not wish them to jointly transit from the *Available* state to the *Borrowed* state, but

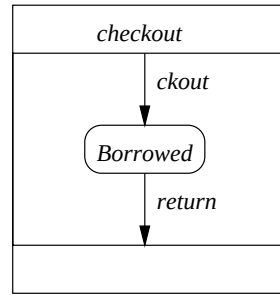

Fig. 4. The checkout life cycle.

rather one at a time. However, it is at the same time useful to let events be globally visible, to enable any life to participate in any event occurrence.

Discussion. We need to give every life its own identity, with a *this* notation, just like the keyword representing the 'current object' in object-oriented programming languages. The *this* keyword makes it possible to distinguish lifes from each other; but the events should also distinguish between different lifes involved, rather than life cycles. Therefore, we introduce parameters on events.

Normal object-oriented programming languages often support *overloading* of operations, where two operation names are only equivalent when the number and types of parameters somehow match. We take this same approach, and go one step further, by not only looking at the types of parameters, but also at the identities of the parameters. This is like multi-methods, defined from a process perspective.

To make it possible to give out parameters other than *this*, a life cycle can define variables to hold references to (other) lifes. These variables can be read or written as event parameters.

Solution. Every life can define a set of variables, which hold references to (other) lifes. These values are passed around in parameters to events and event occurrences. Two important properties of parameters are their direction and whether or not they are (already) bound to a value.

The direction of parameters may be:

- *In*, denoted with *?* graphically, for variables whose value is read from the parameter. The value for these parameters is not constrained by this parameter.
- *Out*, the graphical default, for variables that are exposed to a match along with the name of the event. The third clause of *eventmatch* exists because of this matching on *Out* parameters.
- *Force*, denoted with *!* graphically, to enforce that an event's parameter matches a variable's value. An example of this will be introduced in subsection 1.5. If the rest of an event e matches but this parameter does not, then e is *Blocked*.

The parameter may either be an *Unbound* reference to a variable (mentioning that variable by name) or it may be *Bound* to some *lifeID* value:

$$param \equiv (dir, arg)$$
$$dir ::= In \mid Out \mid Force$$
$$arg ::= Unbound\ var \mid Bound\ lifeID$$
$$var \equiv name$$

Variable names are unique within a life cycle. Furthermore, every life cycle defines a variable named "`this`", that already holds the *lifeID* of a life on creating transitions; all other variables can only get values by obtaining them as *In* parameters from some event occurrence:

$$this :: var$$
$$this = \text{"}\texttt{this}\text{"}$$

A life cycle definition only contains *Unbound* parameters, but after a life is instantiated, variables are filled in. We define a *binding* to be a mapping from *Unbound* to *Bound* parameters; this is the way variable values are represented in a *life*. The *bind* function applies a *binding* by mapping it over a *transition*, whose introduction we defer to subsection 1.6:

$$binding \equiv [(var, lifeID)]$$
$$bind :: binding \to transition \to transition$$
$$bind\ bnd\ (f, t, eves) = (f, t, map\ (eventbind\ bnd)\ eves)$$
where
$$eventbind\ bnd\ (nm, parms) = (nm, map\ (parmbind\ bnd)\ parms)$$
$$parmbind\ ((nm, val) : nms)\ (d, Unbound\ nm) = (d, Bound\ val)$$
$$parmbind\ (nv : nms)\ x = parmbind\ nms\ x$$
$$parmbind\ y\ x = x$$

A variable v may only be used as an *Out* or *Force* parameter in transitions leaving a state s if either v is *this* or v is mentioned at least once as an *In* parameter on each possible path leading to state s. This is a syntactical constraint on life cycles.

Example. Figure 5 presents a redesign of the library of figure 3, where the events have parameters. For example, the event occurrence *ckout(b,m)* communicates with *book* b and *member* m; therefore, a single *book* b is checked out, and other books remain in the state in which they were; also, if some *book* b' is in *Borrowed* state, then it does not block the *ckout(b,m)* event occurrence, provided that b' differs from b.

1.5 Enforcing Synchronisation

Problem. Imagine extending the library in figure 5 with reservations (on *books* rather than *titles* for simplicity). Then, whenever a reserved *book* is returned,

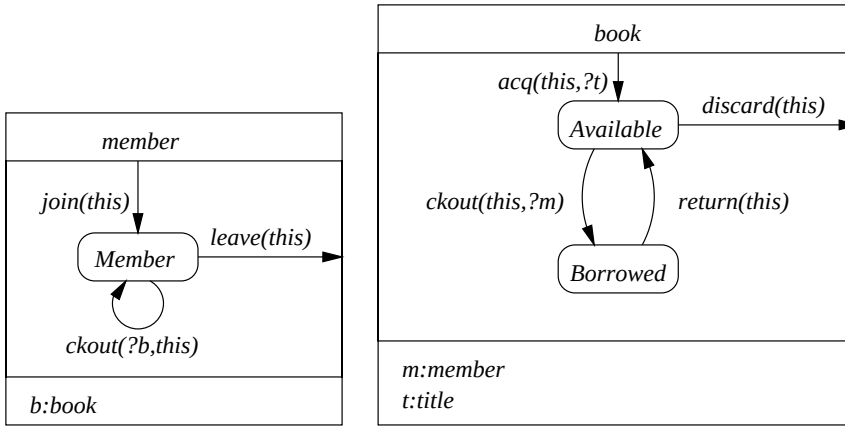

Fig. 5. Two library life cycles with parameters.

it must be put on hold. In many object methods, this is done by splitting the *return* transition like in figure 2. In our life cycle approach, we prefer to exploit communication to add a constraint, stating that the next *member* to check out the *book* b must be the *member* for whom b was put on hold.

Discussion. With only *In* and *Out* parameters, life cycles could not express this kind of 'identity enforcement.' An *In* parameter matches with anything, and an *Out* parameter simply *Ignore*s event occurrences that have different identities for that variable. Therefore, we need to introduce another kind of parameter, which disallows (or *Blocks*) event occurrences with non-matching parameters.

Solution. We will graphically annotate such parameters with a *!* which is pronounced '(en)force.' In our formal models, we use the direction *Force* for these parameters. If an event occurrence *eo* matches an event on name, parameter count and all *Out* parameters, then there still is the question whether *eo* should be *Matched* or *Blocked*; the latter is chosen in case a *Force* parameter has different value from the value in *eo*. This is what has been formalised in clause 5 of the definition of *eventmatch*.

Example. Figure 6 defines the *reservation* life cycle. This life cycle is created when a *reserve(?b,?m)* event occurs, and awaits the *return* of *book* b. Wen this book is returned, the *reservation* life 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. The result is a dynamic extension to the communication structure which deals with the added constraint due to the *reservation*; 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 another *member* than m.

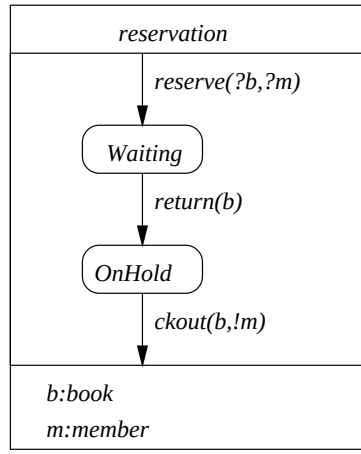

Fig. 6. Enforcing a certain *member* parameter value.

1.6 Multiple Perspectives on Event Occurrences

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 (*books*) of the same *title* are owned by the library. Such a *reservation* life cycle that awaits a *title* cannot be made without requiring the title in the parameter list of the events that make a book available, thus also on *return*. The *title* has thus far not been a parameter to the *return* event, and it would be a demonstration of bad extensibility of life cycles if we would 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.

We want to support observing event occurrences from different perspectives, much like in step failures semantics [20]. We therefore allow *multiple events per transition.* Each of these events can represent a different perspective on the transition. To ensure that all perspectives occur at the same time, the events on a transition must all occur together.

Note how this approach replaces that of nested states, as used in state charts. Nested states have a hierarchical ordering, and that hierarchy is for one perspective only. Another perspective usually shows up as a cross-level transition, which indeed is part of the syntax of state charts. We therefore believe that an extension of life cycles with hierarchical states would *weaken* the concept of life cycles rather than improve it.

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

$$
\begin{aligned}
&\mathit{transition} \equiv (\mathit{state}, \mathit{state}, [\mathit{event}]) \\
&\mathit{transfrom} :: \mathit{lifecycle} \to \mathit{state} \to [\mathit{transition}] \\
&\mathit{transfrom}\ (n, x, ts)\ \mathit{from} = [(f, t, es) \mid (f, t, es) \leftarrow ts; f = \mathit{from}]
\end{aligned}
$$

Now that we have seen that event occurrences as well as transitions are represented with sets of events, it is possible to define when they match; informally, this is true when for every event in the transition, there is a matching event in the event occurrence:

$$
\begin{aligned}
&\mathit{evoccmatch} :: \mathit{evocc} \to \mathit{transition} \to \mathit{acceptance} \\
&\mathit{evoccmatch}\ \mathit{eo}\ (f, t, es) \ = \mathit{Block}, \qquad\qquad \text{if } \mathit{blocked} \\
&\qquad\qquad\qquad\qquad\ = \mathit{compose}\ [\mathit{eventmatch}\ e'\ e \mid e' \leftarrow \mathit{eo}; e \leftarrow es], \\
&\qquad \text{where} \qquad\qquad\ \text{otherwise} \\
&\qquad \mathit{matchdom} = [e \mid e \leftarrow es; \exists x \in \mathit{eo}@(\mathit{isMatch} \circ \mathit{eventmatch}\ e)\ x] \\
&\qquad \mathit{blocked} = [] \neq \mathit{matchdom} \neq es
\end{aligned}
$$

In general, a *match* is found if no inconsistencies exist in the required *Match* unifications:

$$
\begin{aligned}
&\mathit{match} :: \mathit{evocc} \to \mathit{transition} \to \mathit{acceptance} \\
&\mathit{match}\ \mathit{eo}\ \mathit{tr} = \mathit{consistent}(\mathit{evoccmatch}\ \mathit{eo}\ \mathit{tr}) \\
&\qquad \text{where} \\
&\qquad \mathit{consistent}\ \mathit{Ignore} = \mathit{Ignore} \\
&\qquad \mathit{consistent}(\mathit{Match}\ xs) \ = \mathit{Match} xs, \\
&\qquad\qquad\qquad\qquad\qquad \mathit{if and}\ [\text{a=a'} \mid (x, a) \leftarrow xs; (x', a') \leftarrow xs; x = x'] \\
&\qquad \mathit{consistent}\ x = \mathit{Block}
\end{aligned}
$$

Example. To notate multiple events on one transition, we intersect the transitions with a comma, pronounced 'and.' For example, in the life cycle for a *book,* it is sensible to introduce an event *avail(this,t)* along the transition that now holds the event *return(this)* only. The *avail* event can be caught in the *reservation* life cycle. Moreover, from the perspective of books becoming available, we can see the acquisition of a new *book* as an *avail(this,t)* event too. The resulting life cycles for *book* and *reservation* are therefore as given in figure 7. It is sufficient to *mention* the existence of a *title*, we need not define its life cycle.

1.7 Step Semantics

This subsection is dedicated to describing precisely what the semantics of a step of a complete life cycle system are. The general approach is to first find out what transitions a single life can make given some event occurrence, then what happens if these are applied to a single life, and then map this approach over all life cycles in the system.

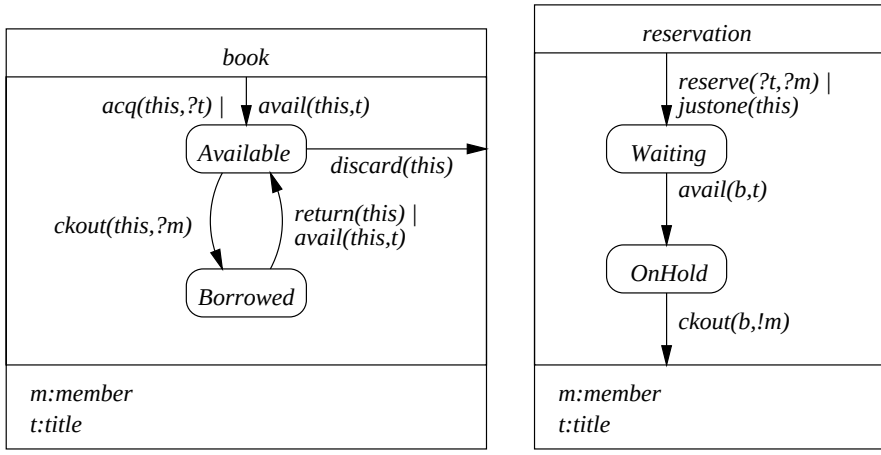

Fig. 7. The use of multiple events per transition.

The *lifesteps* function lists every possible *transition* that a life can engage into, given its current state and an event occurrence. The result lists the *acceptance* for every possible *transition*. The result is a set (implemented here as a list); an empty set means that no non-blocking transitions are possible; one result means that a unique transition can be made; multiple results means that a non-deteministic choice can be made between the outcomes.

$$
\begin{aligned}
&\mathit{lifesteps} :: \mathit{evocc} \to \mathit{life} \to [(\mathit{transition}, \mathit{acceptance})]\\
&\mathit{lifesteps}\ \mathit{eo}\ (\mathit{lc}, \mathit{lid}, \mathit{st}, b) = \mathit{matched}\ \underline{\text{else}}\ \mathit{ignored}\\
&\qquad \text{where}\\
&\qquad \mathit{transitions} = [(\mathit{tr}, \mathit{match}\ \mathit{eo}\ \mathit{tr}) \mid \mathit{tr} \leftarrow \mathit{map}\ (\mathit{bind}\ b)\ (\mathit{transfrom}\ \mathit{lc}\ \mathit{st})]\\
&\qquad \mathit{nonblocked} = [(\mathit{tr}, \mathit{acc}) \mid (\mathit{tr}, \mathit{acc}) \leftarrow \mathit{transitions}; \neg \mathit{isBlock}\ \mathit{acc}]\\
&\qquad \mathit{matched} = [(\mathit{tr}, \mathit{acc}) \mid (\mathit{tr}, \mathit{acc}) \leftarrow \mathit{nonblocked}; \mathit{isMatch}\ \mathit{acc}]\\
&\qquad \mathit{ignored} = [(\mathit{tr}, \mathit{acc}) \mid (\mathit{tr}, \mathit{acc}) \leftarrow \mathit{nonblocked}; \mathit{isIgnore}\ \mathit{acc}]\\
&\qquad [\,]\ \underline{\text{else}}\ \mathit{ys} = \mathit{ys}\\
&\qquad \mathit{xs}\ \underline{\text{else}}\ \mathit{ys} = \mathit{xs}
\end{aligned}
$$

This makes it possible to construct a function *steplife* that constructs a list of possible future *lifes* from a current *life* and an *evocc* happening to it:

$$
\begin{aligned}
&\mathit{steplife} :: \mathit{evocc} \to \mathit{life} \to [\mathit{life}]\\
&\mathit{steplife}\ \mathit{eo}\ l = \mathit{map}\ (\mathit{actualstep}\ l)\ (\mathit{lifesteps}\ \mathit{eo}\ l)
\end{aligned}
$$

With the *actualstep* function, a transition is applied to a life; this influences the state in which the life resides. Furthermore, old bindings *bnd_old* are removed as far as they are read by this step, and new bindings *bnd_new* are added instead for read-in variables:

$$
\begin{aligned}
&\mathit{actualstep} :: \mathit{life} \to (\mathit{transition}, \mathit{acceptance}) \to \mathit{life}\\
&\mathit{actualstep}\ (\mathit{lc}, \mathit{lid}, \mathit{st}, \mathit{bnd})\ (\mathit{tr}, \mathit{Ignore})
\end{aligned}
$$

$$
\begin{aligned}
&= (lc, lid, st, bnd)\\
&actualstep\ (lc, lid, st, bnd)\ ((f, t, es), Match\ ms)\\
&\quad = (lc, lid, t, mkset((bnd \mathbin{---} bnd\_old) +\!\!+ bnd\_new))\\
&\qquad \text{where}\\
&\qquad bnd\_old = [(var, val) \mid (var, val) \leftarrow bnd;\\
&\qquad\qquad\qquad\qquad \exists x \in ms@((Unbound\ var\ =) \circ snd)\ x]\\
&\qquad bnd\_new = [(var, val) \mid (Bound\ val, Unbound\ var) \leftarrow ms]
\end{aligned}
$$

A life can block the execution of an *evocc* that it can accept in some possible future; this is a consequence of the dynamic communication principle explained in subsection 1.2. The function *futureblock* determines if the given *sys* blocks the given *evocc* because of some future transition of which all variables are currently bound:

$$
\begin{aligned}
&futureblock :: evocc \to sys \to bool\\
&futureblock\ eo\ (lcs, ls) = \exists x \in ls@futureblockinglife\ x\\
&\qquad \text{where}\\
&\qquad futureblockinglife\ l = \exists x \in futurematches\ l@((\neg) \circ isIgnore)\ x\\
&\qquad futurematches\ l = [acc \mid (tr, acc) \leftarrow futurelifesteps\ [nirwana]\ eo\ l;\\
&\qquad\qquad\qquad\qquad allbound\ tr]\\
&\qquad allbound(f, t, es) = [var \mid (nm, parms) \leftarrow es;\\
&\qquad\qquad\qquad\qquad (d, Unbound\ var) \leftarrow parms; d \neq In] = []
\end{aligned}
$$

This uses a function *futurelifesteps*, which lists, once again for each possible *transition*, the possible *acceptance* value, only this time taking the future after that step into account in the *acceptance* value. This blocks on events that are *Ignore*d or *Match*ed in the current state, but that *Block* in some future state.

$$
\begin{aligned}
&futurelifesteps :: [state] \to evocc \to life \to [(transition, acceptance)]\\
&futurelifesteps\ states\ eo\ l = [(tr, compose(acc(tr, a) : accs(tr, a))) \mid\\
&\qquad\qquad\qquad\qquad (tr, a) \leftarrow stepstodo]\\
&\qquad \text{where}\\
&\qquad stepstodo = lifesteps\ eo\ l\\
&\qquad stepsdone((f, t, es), a) =\\
&\qquad\qquad\qquad\qquad filter\ otherstate\ [actualstep\ l\ ((f, t, es), Match\ [])]\\
&\qquad otherstate(lc, lid, s, b) = s \notin states\\
&\qquad acc = compose \circ map\ snd \circ concat \circ map(futurelifesteps\ (states'\ l)\ eo)\\
&\qquad\qquad\qquad\qquad \circ stepsdone\\
&\qquad accs(tr, a)\ = [], \quad \text{if } \#states = 1\\
&\qquad\qquad\ \ = [a], \quad \text{otherwise}\\
&\qquad states'(lc, lid, s, b) = s : states
\end{aligned}
$$

When a *sys* makes a step, every *life* decides independently which of its possible futures to select; therefore, the whole *sys* has as its possible futures the cartesian product of the possible futures of all its *lifes*. Note how we apply the functions

birth and *death* to obtain the desired effect of creation and destruction as side-effects of communication:

$$
\begin{aligned}
&step :: evocc \to sys \to [sys] \\
&step\ eo\ sys\ = [], && \text{if } futureblock\ eo\ (birth\ sys) \\
&\qquad\quad = ((map\ death) \circ sysstep \circ birth)\ sys, && \text{otherwise} \\
&\quad \text{where} \\
&\quad sysstep(lcs, ls) = [(lcs, ls') \mid \\
&\qquad\qquad ls' \leftarrow cartesian(map\ (mkset \circ steplife\ eo)\ ls)]
\end{aligned}
$$

The *step* function is the interface through which users of a life cycle system can make the system perform steps.

2 Interacting with a Life Cycle System

The previous section introduced our life cycle concept. In this section we demonstrate how commonly requested information is available in a life cycle system. We describe the demonstrations in this example in our semantics language Miranda, to achieve sufficient precision, also outside the boundaries of our life cycles.

2.1 A Component with a Book LifeCycle

The *book* life cycle in figure 7 is translated into the formal notation by introducing a number of transitions with sets of events on them, and building the *book* life cycle from it:

$$
\begin{aligned}
&book :: lifecycle \\
&book = (\text{“}\texttt{book}\text{”}, [nirwana, \text{“}\texttt{Available}\text{”}, \text{“}\texttt{Borrowed}\text{”}], \\
&\qquad [bookT1, bookT2, bookT3, bookT4]) \\
&\quad \text{where} \\
&\quad bookT1 = (nirwana, \text{“}\texttt{Available}\text{”}, [(\text{“}\texttt{acq}\text{”}, [(Out, Unbound\ this), \\
&\qquad\qquad (In, Unbound\ \text{“}\texttt{t}\text{”})]), (\text{“}\texttt{avail}\text{”}, [(Out, Unbound\ this), \\
&\qquad\qquad (Out, Unbound\ \text{“}\texttt{t}\text{”})])]) \\
&bookT2 = \ \ldots
\end{aligned}
$$

It is possible to apply event occurrences, including things like members joining the library, to this system (or component) of one life cycle; it simply ignores such event occurrences because no (future) transition of the *book* communicates with a *join* event. This is useful, since it means that an additional life cycle (or component) that does take the *join* event occurrence into account can process it. The *book* does not care about *joins*, but is not blocking them either; this is the idea of compositionality behind life cycles.

Assuming a library with a *book* life cycle present (and no *reservation* life cycle), it is possible to acquire a new book with event occurrences like *acq*:

$$acq, ckout, return :: lifeID \to lifeID \to sys \to [sys]$$

$$
\begin{aligned}
&\mathit{discard} :: \mathit{lifeID} \to \mathit{sys} \to [\mathit{sys}]\\
&\mathit{acq}\ b\ t = \mathit{step}\ [(\text{“}\texttt{acq}\text{”}, [(\mathit{Out}, \mathit{Bound}\ b), (\mathit{Out}, \mathit{Bound}\ t)]),\\
&\qquad\qquad (\text{“}\texttt{avail}\text{”}, [(\mathit{Out}, \mathit{Bound}\ b), (\mathit{Out}, \mathit{Bound}\ t)])]\\
&\mathit{ckout}\ b\ m = \ldots\\
&\mathit{return}\ b\ t = \ldots\\
&\mathit{discard}\ b = \ldots
\end{aligned}
$$

The *acq* event occurrence matches the *bookT2* transition in the *book* life cycle when the right values for b and m are filled in. Assume a library system with no instances: $([\mathit{book}], [])$. Applying *acq* 1 3 to this system returns a list with one future library system, in which a book with *lifeID* 1 and title 3 is registered in state *In*. Applying *ckout* 1 7 to this result leads to a library system with that same book in *Borrowed* state.

$$
\begin{aligned}
&\mathit{exlib}_0 = ([\mathit{book}], [])\\
&[\mathit{exlib}_1] = \mathit{acq}\ 1\ 3\ \mathit{exlib}_0\\
&[\mathit{exlib}_2] = \mathit{ckout}\ 1\ 7\ \mathit{exlib}_1\\
&\mathit{nolibs} = \mathit{discard}\ 1\ \mathit{exlib}_2\\
&[\mathit{exlib}_3] = \mathit{return}\ 1\ 3\ \mathit{exlib}_2\\
&[\mathit{exlib}_4] = \mathit{discard}\ 1\ \mathit{exlib}_3
\end{aligned}
$$

It is impossible to *discard* that book from the resulting library exlib_2; it should first be returned. This is the kind of synchronisation constraints that we strive for with life cycles, and which is inspired on the communication primitives in process algebras.

Note how the *step* function for the *ckout* and *acq* functions is used in the same way; the fact that *acq* constructs a new object is not visible here, meaning that we succeeded in hiding the creation (and destruction) of objects entirely.

2.2 Deriving Association Information

In the introduction of this paper, we argued that associations contain too little temporal information, and often require further constraints. In this subsection, we demonstrate how association information can be obtained from a life cycle system.

The vagueness of associations means that several definitions are thinkable; for example, in which states of the ‘from’ object (or life) the association is defined. We introduce the following helper function to define such associations:

$$
\begin{aligned}
&\mathit{assoc} :: \mathit{lifecycle} \to [\mathit{state}] \to \mathit{var} \to \mathit{sys} \to [(\mathit{lifeID}, \mathit{lifeID})]\\
&\mathit{assoc}\ \mathit{lc}\ \mathit{ss}\ v\ (\mathit{lcs}, \mathit{ls}) = [(f, t) \mid (\mathit{lc}', f, s, b) \leftarrow \mathit{ls}; \mathit{lc} = \mathit{lc}';\\
&\qquad\qquad s \in \mathit{ss}; (v', t) \leftarrow b; v = v']
\end{aligned}
$$

Given a life cycle *lc* and the states *ss* in which the association should hold, and a variable *v* in *lc* (which must be defined in all the states *ss*) points to the 'to' object (or life), this function returns a list of links (association instances) between 'from' and 'to.'

An example application of this function is the *lentby* association from the introduction:

$$lentby :: sys \to [(lifeID, lifeID)]$$
$$lentby = assoc\ book\ [\text{“}\texttt{Borrowed}\text{”}]\ \text{“}\texttt{m}\text{”}$$

For example, $lentby\ exlib_1 = lentby\ exlib_3 = []$ but $lentby\ exlib_2 = [(1, 7)]$, denoting that in $exlib_2$ a book 1 is *Borrowed* by member 7.

Note how life cycle models need no *nil* values to model absense of links; instead, process state information is used to say which variables are defined.

2.3 Global Process Analysis

Unlike UML state charts, our life cycles communicate in a precisely defined way. The semantics of life cycles thus expresses sufficient information to infer global system behaviour. Consequently, we see possibilities to perform a number of interesting global process checks on a system of life cycles. We intend to perform research on these process checks [18], in spite of the unbounded number of lifes that can be created.

Deadlock avoidance. One way to avoid deadlock is to ensure the following: *Given that the life cycle system $s \in S$, there must be a trace of event occurrences that leads from s to a system without lifes.* The smallest set S to avoid deadlock contains precisely those states that are reachable from a life cycle system without lifes. If a deadlock avoiding checker accepts a life cycle system, an important class of problems has been avoided; furthermore, in the attempts to come to such a system design, the checker can provide valuable feedback to the designer.

Referential integrity. In systems that deal with pointers (like variables in our life cycles), it is important to ensure referential integrity. In systems that use *nil* pointer values, it is desirable to avoid the possibility of runtime exceptions caused by *nil* dereferences. In general, these problems are hard to solve before runtime. For life cycles, this is simpler, since they specify global process knowledge more directly.

We intend to make a checker that ensures the following: *If a life l is in Nirwana state, it is not referenced anywhere.* In other words, in Nirwana state it is safe to delete a life. This resolves the debate [22] on whether to garbage collect or to explicitly delete, since the moment of deletion is precisely known.

Non-determinism. We explicitly allow non-determinism caused by more than one transition from the same life cycle state matching the same event occurrence. This is because non-determinism has a useful interpretation: something

outside the current scope of reference (e.g. some data-related issue) decides on the transition to follow. Checker tools must treat non-determinism as a choice that they cannot influence or count on. In our previous work on protocol checking [17], we used non-determinism in a similar way, namely to model choices made during the execution of an atomic action.

3 Conclusions and Future Work

This paper introduced a variation on state diagrams called (dynamic) life cycles. We presented a graphical notation and a precise semantics.

Our life cycles present some features that we did not find in comparable work: life cycles support polymorphic creation and deletion of new 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 and non-determinism in models.

Life cycles can express multiple perspectives on events as a means of dynamic binding; we expect this to be a good replacement for hierarchical state charts, whose precise meaning is not trivial.

Life cycles contain more information than traditional models. Associations can be inferred from life cycles. Information on which steps can be performed next, and which instances can act as parameters to those steps, can be inferred from life cycles. Finally, life cycles describe how the process aspects of a total system behave, which makes us believe that several global correctness properties become testable; think of deadlock avoidance and referential integrity.

In our future work, we hope to perform such global correctness checks in spite of the unbounded number of lifes in a life cycle system. We intend to gain better understanding on how processes in workflow and implementation objects cooperate. We also wish to investigate the use of refinement in workflow models. We are also interested in relating this work to models like CSP and LOTOS to see if life cycles introduce new formal aspects (possibly, dynamically evolving communication alphabets). Our eventual goal with this work is to come to a simple yet powerful process notion and accompanying verification tools for object oriented models.

Acknowledgements. I wish to thank Maarten Fokkinga for providing constant feedback and input on the ideas behind life cycles. I wish to thank Compuware, in particular Wim Bast, Tom Brus and Edwin Hautus, for providing the challenging practical environment in which this work evolved.

References

1. J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, 1990.
2. 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.

3. R. Bird and P. Wadler. *Introduction to functional programming.* Prentice Hall, 1988.
4. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, 1988.
5. D. D'Souza and A. Wills. *Catalysis: Practical Rigor and Refinement.* Addison-Wesley, 1998.
6. P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors. *The formal description technique LOTOS.* Elsevier Science Publishers B.V., 1989.
7. S.B. Elmasri, R. Navathe. *Fundamentals of database systems.* Benjamin/Cummings, 1994.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison Wesley, Reading, 1996.
9. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
10. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
11. ISO, editor. *Ada 95 Reference Manual, Language and Standard Libraries.* 1994. ISO/IEC 8652:1995(E).
12. 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.
13. Inmos Ltd. *Occam 2 Reference Manual.* Prentice-Hall, 1988.
14. R. Milner. *A calculus of communicating systems.* LNCS. Springer-Verlag, 1980.
15. R. Milner. The polyadic π-calculus: a tutorial. Springer-Verlag, Aug 1991.
16. Rational Software Corporation. *UML Semantics.* Rational Software Corporation, 1997.
17. R. van Rein and M. Fokkinga. Protocol assuring universal language. *Formal Methods for Open Object-Based Distributed Systems*, pages 241–258, 1999.
18. R. van Rein and M.M. Fokkinga. Static checking of dynamic protocols. 1999. submitted to CONCUR'99.
19. B. Selic, G. Geullekson, and P.T. Ward. *Real-time Object-Oriented Modeling.* John Wiley & Sons, Inc., 1994.
20. D. Taubner and W. Vogler. Step failures semantics and a complete proof system. *Acta Informatica*, 27:125–156, 1989.
21. X/Open, editor. *Distributed Transaction Processing: Reference Model, Version 3.* Feb 1996.
22. B. Zorn. The measured cost of conservative garbage collection. *Software, Practice and Experience*, 23(7):733–756, July 1993.