Requirements Eng (2002) 7:243–263

Requirements Engineering

Requirements-Level Semantics and Model Checking of Object-Oriented Statecharts

Rik Eshuis, David N. Jansen and Roel Wieringa
Department of Computer Science, University of Twente, Enschede, The Netherlands

In this paper we define a requirements-level execution semantics for object-oriented statecharts and show how properties of a system specified by these statecharts can be model checked using tool support for model checkers. Our execution semantics is requirements-level because it uses the perfect technology assumption, which abstracts from limitations imposed by an implementation. Statecharts describe object life cycles. Our semantics includes synchronous and asynchronous communication between objects and creation and deletion of objects. Our tool support presents a graphical front-end to model checkers, making these tools usable to people who are not specialists in model checking. The model-checking approach presented in this paper is embedded in an informal but precise method for software requirements and design. We discuss some of our experiences with model checking.

Keywords: Execution semantics; Model checking; Statecharts

1. Introduction

Statecharts allow the specification of complex event-driven behaviours, containing parallelism, event broadcasting, state hierarchy, interrupts and non-determinism. They have been used in structured analysis since the mid-1980s [1,2] and in object-oriented analysis since the early 1990s [3]. They take a central place in the UML [4], where they can be used, among others, to specify object life cycles.

Correspondence and offprint requests to: Rik Eshuis, Department of Computer Science, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands. Email: eshuis@cs.utwente.nl

Statecharts are powerful notations, and they have proven to be an attractive means to specify behaviour. But this very expressive power brings with it a host of semantic issues that can decrease our ability to understand the behaviour specified by means of statecharts. This is a result of the large number of syntactic combinations that can be made in statecharts, which forces us to make a lot of semantic choices.

The issue is compounded by the fact that different users of statecharts may make these choices differently. Several years ago, von der Beeck [5] listed over 20 variants of statechart semantics in structural analysis approaches, which all make these choices differently. A single statechart in structured analysis can thus have more than 20 different semantics. The number of semantics actually in use has increased with every new object-oriented method that incorporates statecharts. The UML has not resolved this multiplicity. Reports from the standardisation effort suggest that the final UML semantics of statecharts will be one of the most complex imaginable [6]. Moreover, the UML semantics contains many open ends, which have to be filled in before statecharts can be used.

This paper takes a route opposite to the UML, which is aimed at simplicity and precision. We define a lightweight version of statecharts that contains the essential constructs of parallelism, state hierarchy and state reactions and extend this with object-oriented constructs for inter-statechart communication and for object creation and deletion. We then define a STATEMATE-like execution semantics for this, which is formalised by means of a labelled transition system (LTS). We implemented this semantics in our graphical editing tool TCM [7], which allows the analyst to create and edit a collection of formal and informal system specifications. For some of the behaviour descriptions,

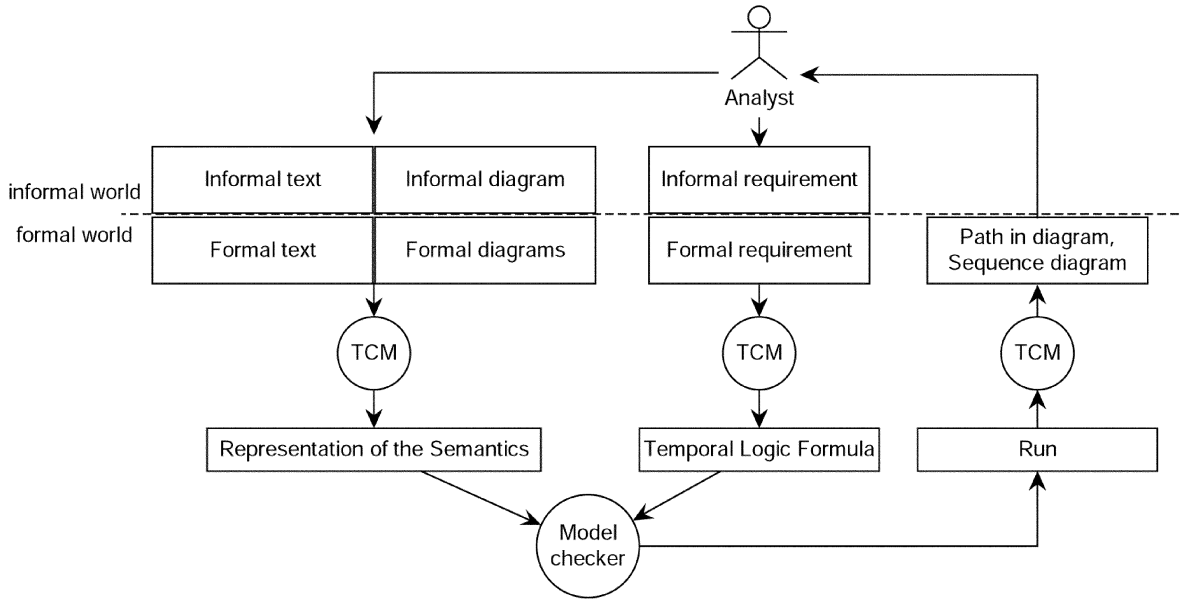

Fig. 1. Using TCM to combine formal and informal specification techniques with model checking.

including UML activity diagrams and simple statecharts, TCM can generate a representation of their semantics (Fig. 1). It can do this in the format required by several model checkers such as SMV [8], NuSMV [9] and Kronos [10]. The semantics can be represented as a labelled transition system, or in other formats understandable by a model checker. The analyst can use the model checker to check properties of the semantic structure. The output of the model checker is fed back to TCM, which presents it in a format understandable to the analyst. The current status of the implementation is discussed in more detail later in this paper.

The goal of this paper is to make model checking available at the requirements level without forcing the analyst to learn unnecessary details of a model checker. We therefore combine our semantics with a tool and a requirements engineering approach, explained in the next section.

1.1. Integrating Formal and Informal Specifications

In our approach, we emphasise the integration of formal with informal specifications. In our terminology, a *notation* is formal if symbol manipulation rules have been defined for it that are sound with respect to an intended meaning. A *symbol manipulation rule*, in turn, is formal if it is defined purely in terms of the physical properties of the symbol occurrences, and not in terms of the meaning of the symbols. An informal notation has no such formal manipulation rules. We assume that the intended meaning with respect to which the manipulation rules must be sound is described in terms of some mathematical structure such as a labelled transition system.

We distinguish formality from *precision*. A description is precise if it is to the point. 'PRECISE denotes the quality of exact limitation, as distinguished from the vague, doubtful, inaccurate … The idea of precision is that of casting aside the useless and superfluous' [11]. A precise description can be informal, and a formal description can be imprecise. Harel and Rumpe [12] distinguish between the precision of a language and the precision of statements in that language, but do not distinguish between formality and precision as we do.

Examples of informal specifications that can nevertheless be precise are a mission statement, a function refinement tree, system function descriptions and an informal system event list. Examples of formal specifications are a state transition table, a Z specification, an entity relationship diagram or a statechart. The formal and informal parts of a specification should supplement each other. In general, only a part of the total specification is formal.

A particular diagram may be interpreted informally first and then reinterpreted formally later. One way of working is that the analyst first describes a statechart informally. TCM then interprets the informal statechart formally, by attaching one particular semantics (namely the requirements-level semantics explained below) to it, and by interpreting the labels as strings that can be tested on equality.

Now the question pops up whether the formalisation done by TCM is in accordance with the intended meaning the analyst attaches to a statechart. We here assume that the analyst is willing to use the semantics TCM uses; by using TCM with model checking, the analyst can learn about the meaning that TCM attaches to the model he specified, and change the model if some unexpected errors occur.

1.2. Not Yet Another Method (NYAM)

Our approach to software requirements is not yet another method, because it consists of elements from structured and object-oriented analysis put together in a particular way [13,14]. There is no recipe to be followed when using these techniques, but there are many guidelines that can help the engineer to find requirements and to design a high-level architecture [14]. For the current paper, two basic ideas in NYAM must be explained.

First, NYAM is goal-oriented. If the composition of the system with its environment is intended to achieve emergent properties E, then the requirements engineer should look for assumptions A about the environment and properties S of the system such that A and S jointly entail E. This approach puts NYAM in the same class of requirements methods as KAOS [15,16], the requirements reference approach of the Gunters, Jackson and Zave [17,18] and, to some extent, SCR [19–21]. We call the argument that the environment and the system jointly assure the emergent properties the *systems engineering argument.* Model checking can help us in a variety of ways in giving this argument. One is to formalise A, S and E and then verify whether $A \wedge S \models E$. Another is to try to check whether $S \models E$ and use the counterexamples to identify any missing assumptions A about the environment needed to produce the argument. We will illustrate this later.

Second, we classify software architecture design decisions into two groups: those based upon software requirements and upon properties of the external environment, and those based upon the software implementation platform. A software architecture motivated in terms of the external environment and the software requirements is a *requirements-level architecture*. When we include in our design motivations arguments based upon a particular implementation platform, we get an *implementation-level architecture.* An example of a requirements-level architecture is the essential data flow diagram of McMenamin and Palmer [22]. It represents the essential structure of the system, stripped of all implementation considerations. By proposing a requirements-level architecture for a system, the requirements engineer says that any implementation of the system must exhibit this architecture. In this paper, our requirements-level architectures will be object-oriented. We represent them by a UML static structure diagram rather than by a data flow diagram.

Our system specification S will take the form of a specification of a collection of communicating objects, to which we give a requirements-level semantics. This is discussed next.

1.3. Requirement-Level Semantics

In a *requirements-level semantics*, the assumption is made that the system under design has infinite resources and can compute infinitely fast. McMenamin and Palmer call this the *perfect technology assumption* [22] and we borrow their terminology. It is implied by the perfect synchrony hypothesis of Esterel, which says that the response to an event occurs at the same time as the event [23].

By making this assumption, the modeller does not have to worry about implementation details, but instead can focus on specifying a solution to the problem at hand. STATEMATE [24] makes this assumption. By contrast, in an *implementation-level semantics* the focus is on implementing a design and the perfect technology assumption is therefore dropped. The goal now is that the implemented design meets the requirements. The OMG UML semantics is an implementation-level semantics [25]. Both a requirements-level and an implementation-level semantics are useful, but at different points in the design cycle. The requirements-level semantics is useful when early

Table 1. Differences between requirements and implementation-level semantics

Requirements-level semantics	Implementation-level semantics
Perfect technology	Imperfect technology
Input is event set	• Input is queue
System reacts to all events in input	• System reacts to one event in input (the first)
Event is responded to in next step	• Event is responded to in some subsequent step
Instantaneous communication	• Non-instantaneous communication
Communication arrives always	Communication may get lost, or arrive at wrong destination
One global clock	Many local clocks
No clock drift	Clocks may drift
Action is instantaneous	• Action takes time
Unlimited concurrency	• Limited concurrency, i.e. threads of control with interleaving per thread (one thread per active object)

requirements need to be specified, whereas the implementation-level semantics is useful when some agreed-upon final design needs to implemented.

Our statechart semantics is intended to be used for requirements specification. This differs from the use of statecharts to describe implementation-level behaviour. The differences are listed in Table 1. The OMG UML semantics makes the assumptions marked by a bullet in Table 1. Our semantics, like STATEMATE, makes all of the assumptions in the left-hand column.

1.4. Structure of the Paper

In Section 2, we summarise the necessary syntax definitions of UML statecharts. Next, in Section 3 we present the requirements-level semantics for statecharts. This is an informal but precise update of a semantics defined formally in an earlier paper [26]. We begin with a discussion of the design choices and define the basic execution of a statechart. Then we define the execution semantics of one statechart in terms of execution algorithms. Next we extend that semantics with creation and deletion action expressions and communication between statecharts. In Section 4 we present a small case study and show how different model checkers can be used to verify different aspects of the design. We end with a discussion and conclusions.

2. Statechart Syntax

We explain the syntax of statecharts informally. We gave formal definitions in an earlier paper [26]. Figure 2 shows an example statechart. It models the behaviour of a dialogue in which electronic railway tickets are sold to travellers. The complete system this statechart is part of is described in Section 4. Rounded rectangles represent state nodes. Directed edges represent state transitions.

A statechart is a collection of state nodes, hierarchically related, and connected by directed edges. A state node is called *active* if it is in the current execution configuration, as explained in detail in Section 3.2. The hierarchy relation is defined as follows: A state node s can have sub-state nodes, called *children* of s. If s' is a child of s, then s is parent of s'. Children are visually

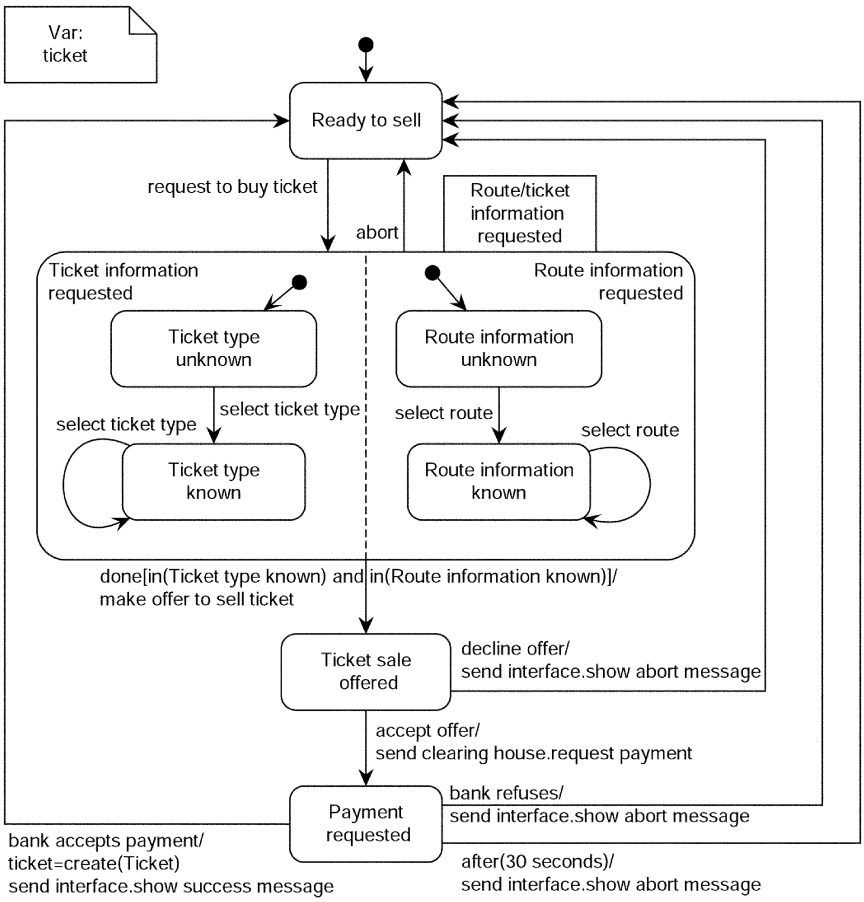

Fig. 2. Statechart for selling railway tickets dialogue.

represented by node containment (child is contained in the super state node). The *descendants* of *s* are all the children, children's children and so on of *s* (the transitive closure of children).

If node *s* has children, we call *s* a *compound state node*. If *s* has no children, it is a *basic state node*. There are two kinds of compound state nodes: AND and OR state nodes. An AND state node denotes parallelism: If *s* is an AND state node and *s* is active, then all its children are active as well. The children of an AND state node are separated by a dashed line. In Fig. 2, Route/Ticket Information Requested is an AND state node. An OR state node denotes exclusive choice: If *s* is an OR state node and *s* is active, then exactly one of its children is active as well. In Fig. 2, Ticket Information Requested is an OR node. For technical reasons, we require that the top-level state node of a statechart be an OR node; we call this state node *root*. Node *root* is not drawn.

The edge leaving a black dot points at an initial state node. It is required that every OR node have an initial, default state node. This initial state node denotes which node is entered by default when the OR node is entered. In Fig. 2, if OR node Ticket information requested is entered, then node Ticket type is unknown is entered by default. A bull's eye (not present in Fig. 2) denotes a final state node. Contrasting to STATEMATE statecharts and in accordance with UML statecharts, in our semantics a final state node denotes local termination of the corresponding OR node, not global termination of the complete statechart.

A directed edge specifies a transition from one state node (source) to another one (target). More than one source and more than one target state node is allowed in our formal semantics [26]; but to simplify the exposition, we assume here every edge has a single source and a single target. Edges can be labelled with an event expression followed by a guard expression between brackets followed by a slash and an action expression. All these three expressions are optional.

Timeout expressions are special event expressions. Following the UML we use two kinds of timeout expressions: absolute and relative. An absolute timeout expression references the current time. It is specified by the UML when(*cond*) construct (e.g. when(time =12:00:00h)). A relative timeout expression references the time relative to some occasion, by default the time the source state node is entered. It is specified by the UML after(*texp*) construct. For example, in Fig. 2, 30 seconds after node Payment requested is entered, timeout after(30 seconds) is generated.

Guard expressions are Boolean expressions that can refer to local variables of the statecharts. The only guard expression we use in Fig. 2, [in(Ticket type known) and in(Route information known)], is true if and only if both Ticket type known and Route information known are active.

An action expression is a sequence of one or more basic actions. A basic action can be, for example, an assignment to a local variable, a create operation, a send operation or a call operation. Send operations are always asynchronous whereas call operations are always synchronous. With both these operations, the receiver must be explicitly denoted. For example, in Fig. 2, if event accept offer occurs, a send operation send clearing house.request payment is executed, meaning that event request payment is sent to object clearing house.

In order for the edge to be taken, its source must be active, the event specified in the label must occur and the guard expression in the label must be true. When the edge is taken, first its source is left, so becomes inactive, then the actions specified in the label are performed, and finally the target state node is entered, so becomes active.

3. Statechart Semantics

In this section we present our statechart semantics in terms of execution algorithms. First we discuss the choices we made in our statechart semantics in Section 3.1. Then, in Section 3.2, we define a step: a set of edges that is concurrently taken. A step is taken in response to changes in the environment. In Sections 3.3 and 3.4 we define two different ways of executing a step. In the *clock-synchronous* semantics, a step is executed when the clock ticks. In the *clock-asynchronous* semantics, a step is executed immediately when new events arrive. Then the system becomes unstable and reacts infinitely fast to become stable again. These two semantics are borrowed from the STATEMATE semantics. The major differences with the STATEMATE semantics are the absence of a separate activity model in our semantics, and the presence of synchronous and asynchronous communication between statecharts, and of object creation and deletion.

In the next sections, we focus on multiple statecharts. In Section 3.5 we extend the semantics of the previous sections with creation and deletion of objects with statecharts, absent from the STATEMATE semantics. In Section 3.6 we add synchronous and asynchronous communication between different statecharts. STATEMATE only uses asynchronous communication. Formal definitions of these semantics, without the execution algorithms and the examples, are presented in a previous paper [26].

3.1. Semantic Choices

Statecharts were introduced by Harel [1] to model the behaviour of activities in the structured analysis approach STATEMATE [24]. They have been adapted in many object-oriented design notations, including the UML [25], but with an informally or undefined semantics that appears to be quite different from the STATEMATE semantics. The actual difference between, for example, the structured-analysis STATEMATE and the object-oriented UML statecharts is blurred because the STATEMATE semantics is defined at the requirements level whereas the OMG semantics of UML is defined at the implementation level.

In a previous paper, we have defined a formal requirements-level semantics for UML-based statecharts [26] that is an object-oriented version of the STATEMATE semantics [24]. By defining this formal semantics, we were able to classify the differences between structured and OO semantics of statecharts in two dimensions: structured versus OO models and requirements level versus implementation level. In the introduction we have discussed the difference between a requirements-level and an implementation-level semantics.

Table 2 summarises the difference between our OO semantics and the structured STATEMATE semantics [24,27].

Firstly, object-oriented models encapsulate data manipulation, control and data state into objects (as operations, statecharts, attributes). Structured Yourdon-style models separate them into data processes, control processes and data stores, respectively. STATEMATE does this too, but in addition it allows for local variables of statecharts. STATEMATE models (as all other structured analysis models) have separate activities where OO models only know of data manipulation local to an object. The absence of separate activities and the use of true local variables considerably simplifies our semantics w.r.t. the STATEMATE semantics. Secondly, to communicate information to a destination, objects in OO models must use the identifier(s) of the destination(s) whereas processes in structured models must use the identifiers of the communication channels. As a consequence, communication in OO models is point-to-point, whereas in structured models it is broadcast. Finally, object-oriented models use the type–instance distinction, which is absent from structured models. This means that in our semantics we deal with dynamic creation and deletion of instances.

Table 2. Differences between STATEMATE and our OO semantics

STATEMATE activity chart + statechart	UML class diagram + statechart
Separation of data state/process and control state/process	Encapsulation of data state/process and control state/process
Channel addressing	Identifier addressing
Broadcast communication	Point-to-point communication
Instance level model	Type–instance distinction

Together, Tables 1 and 2 factorise the differences between the STATEMATE and UML semantics of statecharts into two groups: the differences between structured and OO models and the difference between requirements-level and implementation-level semantics. On the other hand, Harel and Gery [28] state that the main difference is that UML statecharts use run-to-completion (RTC) whereas STATEMATE statecharts do not. RTC, which was introduced in the statechart semantics of ROOM [29], is one possible way to maintain atomicity of transitions at the implementation level. In an RTC semantics, an event can only be processed, if processing of the precious event input has completed (all triggered transitions have been completely taken). Sending an asynchronous message is considered to be completed when the message is sent; calling a synchronous operation is considered completed when the called operation is completed, and this requires maintaining a call stack. STATEMATE has no synchronous communication and hence no call stack. Hence, STATEMATE has RTC semantics in a trivial sense. In our opinion, RTC is but a minor difference compared to the differences identified by us. This difference is caused by the fact that UML has synchronous communication whereas STATEMATE has not. But this is not a particular difference between structured analysis and object orientation!

We now briefly sketch some design choices we have made in defining a semantics. The following choices have to be made for any semantics of statecharts, regardless of semantic level or design paradigm (see also von der Beeck [5]):

- We specify both a clock-synchronous and clock-asynchronous semantics [24]. In the clock-synchronous semantics, the system starts processing its input only at the tick of the clock. In the clock-asynchronous semantics, the system starts processing its input as soon as it receives it. In the OMG semantics of UML, this issue is ignored.
- In synchronous communication, the caller must wait until the callee has finished processing the communication. In asynchronous communication, the caller continues without waiting for the receiver to finish processing the communication. STATEMATE only uses asynchronous communication. We follow the UML in defining a semantics for both. We show that in our

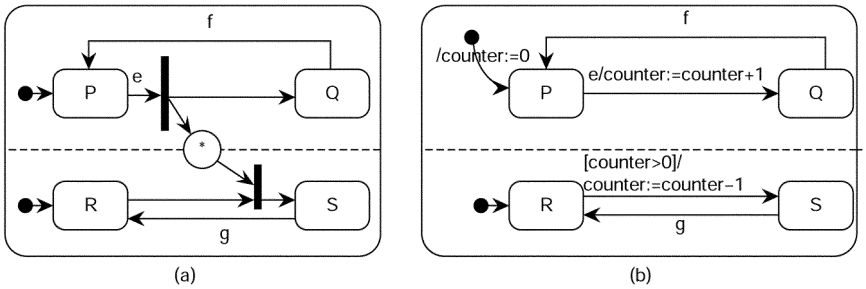

Fig. 3. Example simulation of synchronisation state nodes with counters.

requirements-level semantics synchronous communication is only possible with a clock-asynchronous semantics.

- Updates to variables are made at the end of a step, as in STATEMATE. This way no inconsistent value can be read.
- We assume a given priority order on transitions, and do not commit ourselves to any particular one. Possible priority orders are the STATEMATE one (a higher-level transition has priority over a lower-level one) and the UML one (a lower-level transition has priority over a higher-level one). In Section 3.2 we discuss the definition of priority in the UML and STATEMATE.
- In the UML, a distinction is made between active and passive objects. Active objects have computing resources, passive objects do not. At the requirements level, all objects have unlimited computing resources, so there only are active objects.
- Like the UML [25], we allow no compound triggers, no negated trigger events, and only a single entry and single exit action.
- We allow for the case that the action expressions of a transition contain sequence, interleaving and parallel operators. The example action language that we use only contains the sequence operator, but our semantics can deal with the generalised case. Parallel actions that interfere lead non-deterministically to different possible end states.

The following UML statechart constructs could be added to our requirements-level semantics, but to simplify the exposition we omitted them:

- Deferred events. In the clock-synchronous semantics, these can be simulated by regenerating an event as often as it is to be deferred. They cannot be simulated in the clock-asynchronous semantics, because regenerating an event in the clock-asynchronous semantics would mean that the current state becomes unstable and can never become stable.
- Parametrised events. We abstract away from parameters, since these may make the state space infinite, thus making model checking considerably harder.
- A taxonomy for events. This is an abbreviation mechanism that allows one to reduce the number of transitions in a statechart. It does not add expressive power.
- Activities (actions that take time). These can be simulated by instantaneous start and finish events.
- Dynamic choice points. Adding this UML construct would mean that a step is executed in two parts, separated by the dynamic choice point. Dynamic choice points can be simulated by adding an intermediate state node.
- Synchronisation state nodes. These are pseudo state nodes used to emulate a monitoring or semaphore construct. We regard this to be an implementation-level construct. It can be simulated using a counter (a semaphore). An example elimination is presented in Fig. 3.
- History states. These can be simulated by (re)defining an entry function (see, for example, Damm et al. [27]).

3.2 Step Semantics

The states of a statechart are sets of state nodes. A state of a statechart, called a *configuration*, is a set of state nodes that satisfies the following constraints:

1. *root* is in the configuration.
2. If an OR node is in the configuration, exactly one of its children is in the configuration as well.
3. If node is in the configuration, then its parent is in the configuration as well.
4. If an AND node is in the configuration, all of its children are in the configuration as well.

A set of state nodes N that has no inconsistent nodes can be extended to a configuration in a canonical way. For example, if N contains some OR node n but none of its children, N can be turned into a configuration by adding n's default node to N. We call the resulting configuration

the *default completion* of N. Note that the original set must not contain two nodes that have the same OR node as parent; it is impossible to turn such a set into a configuration.

A node is *active* if and only if it is part of the current configuration.

A statechart can change configuration by taking edges. It only changes configuration when changes in its environment occur. We distinguish three kinds of changes:

- An external event is a discrete change of some condition in the environment. This change can be referred to by giving a name to the change itself or to the condition that changes:
  - A *named external event* is an event that is given a unique name.
  - A *value change event* is an event that represents that a Boolean condition has become true. Value change events are represented by edge labels that have a guard expression but no event expression.
- A *temporal event* is a moment in time to which the system is expected to respond, i.e. some deadline. For example, in Fig. 2 the label after(30 seconds) denotes a deadline, after which the system is supposed to react by moving to node Ready to sell. Temporal events are generated by an internal clock in the system. A formal definition can be found elsewhere [26].

These three kinds of changes are also used in STATEMATE and UML.

An edge is *enabled* and can be taken if and only if its source node is in the configuration, the event specified in the label occurs and the guard expression in the label is true. If some part of the guard expression refers to a local variable, the current value of this variable is substituted for this variable.

Every edge e has a smallest OR node that contains both the source and target state node as descendants. We call this OR node the *scope* of e. For example, in Fig. 2 the edge with event label done has scope *root*, but the edges with event labels select route have scope Route Information Requested.

When the edge e is taken, all the descendants of the scope of e are left (this includes the source), the actions specified in the label are performed and the default completion of the non-left states, extended by the target state node, is entered. For example, in Fig. 2, if the edge with event label done is taken, all the nodes Route/Ticket Information Requested, Ticket Information Requested,Route Information Requested,Ticket type known, Route information known will be left, but not *root*, and node Ticket sale offered will be entered.

Two enabled edges are *inconsistent* if their scopes are equal or one of the scopes is a descendant of the other. In that case, there is a state node that is left by both edges. But since a state node is active at most once, it can be left only once as well. So, inconsistent edges cannot be taken simultaneously.

To choose between inconsistent edges, we assume some priority relation on edges. The priority relation must be a partial order. If one edge has *priority* over another one, then if both are enabled and inconsistent, the one with the higher priority is preferred.

Finally, we require that as many edges as possible are taken. If we would not require this, some event might have no effect. For example, suppose the configuration contains both Ticket type unknown and Route information unknown and both event select ticket type and select route occur. Then both enabled edges with the corresponding event label should be taken, not just one. Therefore, a set of edges, rather than a single edge, is taken.

Summarising, if in a given configuration a certain set of input events occur, a set of edges is taken, called a *step*, that satisfies the following constraints:

1. *Enabledness.* All edges in the step are enabled.
2. *Consistency.* All edges in the step are consistent with each other.
3. *Priority.* If an enabled edge e is not in the step, then there is another edge in the step that is inconsistent with e and that has higher or the same priority.
4. *Maximality.* The step is maximal; i.e., if an enabled edge is not part of the step, adding it either violates constraint 2 or 3.

The above definition of a step is generic and independent from whether we follow a structured or object-oriented design approach. Both the STATEMATE semantics and the OMG UML semantics satisfy it [26]. It is simpler than the STATEMATE definition of a step [24,27] because like UML we have neither negative nor compound events.

The UML and STATEMATE use different priority definitions. We illustrate the differences by means of the example statechart in Fig. 2. This also illustrates the concept of step. If the current configuration is {*root*,Route/Ticket Information requested, Ticket Information Requested,Route Information Requested,Ticket type known, Route information known} and assume events select ticket type and done happen simultaneously. The two edges whose trigger event is select ticket type and done are inconsistent and thus cannot be taken simultaneously. So either the next configuration will be the same as the current one (edge with event select tickettype is chosen), or the next configuration will be

```
Let S:=∅
while S ⊂ addToStep(S) do
do
   pick an edge e with maximal priority from set addToStep(S);
   let S: = S ∪ {e};
endwhile
return S
```

Fig. 4. *nextstep(C,I)*: algorithm to compute steps.

{*root*,Ticket sale offered} (edge with event done is chosen). In the STATEMATE semantics, an edge e has priority over another edge e' if and only if e leaves *more* state nodes than e' [24]. So according to the STATEMATE priority rule, the edge with event done has priority over the edge with event select ticket type and the next configuration will be {*root*,Ticket sale offered}. In the OMG semantics, an edge e has priority over another edge e' if and only if e leaves *less* source state nodes than e' [25]. So under the OMG semantics, the edge with event select ticket type has priority over the edge with event done and the next configuration will be the same as the current configuration.

Finally, we present an algorithm for computing a step in Fig. 4. We construct a step in an incremental way, by adding edges that are consistent. To aid in this construction, we define function *addToStep* as follows. Suppose we have decided to take a set of edges E. The function *addToStep*(E) gives us the edges that are enabled and are consistent with E. Denote the configuration by C and the set of input events by I. In Fig. 4 a non-deterministic algorithm *nextstep*(C,I) is presented that given C and I computes a step that satisfies the above constraints 1–4.

A formal step definition and a proof that the algorithm meets the step constraints can be found elsewhere [26,30].

3.3. Basic Clock-Synchronous Execution

In the clock-synchronous semantics, the system waits and collects changes of the environment in set I of input events until the clock ticks; when the clock ticks the system takes a step and thus reacts to the changes in the environment that occurred since the previous tick of the clock. The system reaction is infinitely fast, since we assume perfect technology. This behaviour is represented by the execution algorithm in Fig. 5.

```
While true do
   • Receive changes in I until clock ticks
   • Execute a step (see figure 6)
```

Fig. 5. Execution algorithm for the clock-synchronous semantics.

1. Compute S=nextstep (C,I) using the algorithm in Fig. 4.
2. For every edge in S execute the actions (only updating primed variables)
3. Compute the next configuration
4. Update C with the next configuration
5. Empty the input set I
6. Update the local variables with the values of their primed counterparts
7. Switch some timers on and some timers off

Fig. 6. Algorithm to execute a step.

The system reacts to changes (collected in input set I) by taking a step. The definition of what happens when a step is taken is the key part of the statechart semantics. The algorithm for executing a step is shown in Fig. 6. This algorithm is independent from the clock-synchronous semantics; we will reuse it in the clock-asynchronous semantics below.

The algorithm consists of seven parts. First, a step is computed, using the *nextstep* algorithm in Fig. 4. Note that for computing the step the value of some local variables needs to be known in order to evaluate guard conditions.

Second, the actions of the step are executed. As in the STATEMATE semantics [24,27], updates are made to primed variables. Each local variable has a primed counterpart. By updating primed variables instead of unprimed ones, we ensure that every action refers to the same value of a variable during a step. Hence, a kind of isolation property, similar to the notion used in database theory, is enforced [31]. As in database theory, however, it might be possible that by executing actions in different orders, while respecting the ordering on the edge labels, the same variable may have different possible end values. (Such executions are not serialisable.) In STATEMATE semantics, such executions are prohibited and checked by the tool. In our semantics, like Damm et al. [27], we just choose one possible ordering of actions that satisfies the ordering on the edge labels.

Third, the next configuration is computed. If an edge has as target an AND or OR node, or if it crosses the boundary of a state node (a so-called inter-level transition), the default completion of the non-left states, extended by the target node, must be taken. For example, if in Fig. 2 the current configuration is {*root*,Ready to sell} and event request to buy ticket occurs, then AND node Route/Ticket Information Requested is entered. The scope of the edge is *root*. The default completion of {*root*,Route/Ticket Information Requested}

is {*root*,Route/Ticket Information Requested,-Ticket Information Requested,Route Information Requested,Ticket type unknown,Route type unknown}, which is the next configuration.

Fourth, the configuration is updated with the next configuration.

Fifth, the input set is emptied. By the perfect technology assumption the system reaction takes zero time, so no changes in the environment can have occurred while the system is reacting to previous changes.

Sixth, the unprimed variables are updated with the values of their primed counterparts.

Seventh, some timers are switched off and some are switched on. We use a dense time model: timers are represented by reals. For each edge *e* with an after(*texp*) constraint, we introduce a timer that produces the desired timeout. The timer is switched on and set to zero if and only if the source of *e* is entered in the current step. The timer can only increase if the system is not reacting to some events. The timer is switched off if and only if the source of *e was* part of the old configuration, but is no longer part of the new configuration. If the timer reaches *texp*, a timeout is generated automatically by the system and added to the input set *I*. For example, in Fig. 2, the timer for the edge with event label after(30 seconds) is started and set to zero when node Payment requested becomes part of the configuration. The timer is switched off when, for example, the edge with event label bank accepts payment is taken. Note that timeouts are not generated *in* system reactions, but *before* system reactions, so as part of the algorithm in Fig. 5, since timeouts are events that must be in the input set before the subsequent reaction occurs.

3.4. Basic Clock-Asynchronous Execution

In the clock-asynchronous semantics, the system reacts immediately to changes in the environment. Note that more than one change can happen in the environment at the same time. By the perfect technology assumption the system reacts infinitely fast to these changes. Consequently, since the system reacts immediately and infinitely fast, the system reaction occurs simultaneously with the events that triggered this reaction! If during the subsequent step some events are generated and put in the input set *I*, the system reacts immediately to these new events in the next step. So, in this semantics, a sequence of steps is taken, called a *superstep*, rather than a single step as in the clock-synchronous semantics. The sequence of steps is stopped when there are no more events in the input set and no edges are enabled. The resulting state we call *stable*. Note that an edge with no label is enabled by default if its source is active. So the source of this edge can never be part of a stable state configuration. The execution algorithm for this semantics is presented in Fig. 7.

While true do
- Receive changes (external event, temporal event)
- Execute a step (see Fig. 6)
- Repeat: if the resulting state is unstable, then execute a step
  Until the state is stable

Fig. 7. Execution algorithm for the clock-asynchronous semantics.

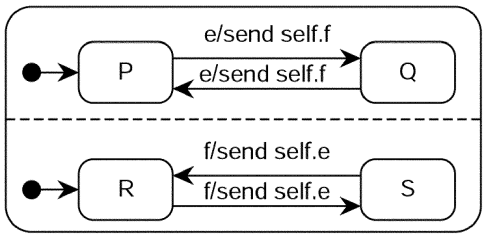

Fig. 8. Divergence in a statechart.

One disadvantage of this definition is that a superstep may be infinite. In that case, we say the superstep *diverges*. For example, if the statechart in Fig. 8 receives either event e or f, a superstep is taken that does not terminate. This behaviour is due to the assumption that generated events cannot be sensed in the current step, but only in the next step. In the fixpoint statechart semantics defined by Pnueli and Shalev [32], the converse assumption is made that generated events are sensed immediately in the current step. However, as pointed out by Leveson et al. [33], this assumption sometimes leads to counterintuitive behaviour. STATEMATE, RSML and UML all make the same assumption as we do, namely that generated events can only be sensed in the next step.

3.5. Creation and Deletion

Above we only defined a semantics for one single statechart. In this (sub)section and the next, we define a semantics for multiple statecharts. We use the following notations and conventions. Assume a set of classes *Class* and a set of object identifiers *OID*. Every class *c* has a statechart definition associated, whose root is identified as *c.root*. Now that we have multiple objects instead of one, we index the configuration *C* and input set *I* with the object id's. So for example *C*[*id*] denotes the configuration *C* of object *id*. We assume that all variables used by objects are unique. Variables can be made unique by simply putting the object id in front. We denote the local

variables of an object *id* with *Var*(*id*). We assume that every object can only update its own local variables in an action, not variables of other objects. As before, every variable *v* in *Var(id)* has a primed version, but now, in addition, every set of input events *I*[*id*] also has a primed counterpart *I*′[*id*], because we define communication between statecharts. In the next paragraph we explain why this is needed. We assume that there is a global clock that represents the current time and that can be referenced by every object.

The meaning of action expression refid:=create (Class) is that an object of class *Class* is created with a new identity that is assigned to variable *refid*. Action expression destroy(id) means that object *id* is destroyed.

In order to give a semantics for creation and deletion we make use of two predicates. Predicate *Exists*(*id*) is true iff an object with identifier *id* exists. Predicate *Used*(*id*) is true iff there exists or has existed an object with identifier *id*. The obvious constraint holds that if *Exists*(*id*) is true, *Used*(*id*) must be true as well. A new object *id* can be created only if *id* has not been used before: ¬*Used(id)*. If object *id* is created, then it is initialised by setting *C*[*id*] to the default completion of {*id.root*}, setting *I*[*id*] to the empty set, and setting *Used*(*id*) and *Exists*(*id*) to true.

An object *id* can be destroyed only if it exists: *Exists*(*id*). If *id* is destroyed, predicate *Exists*(*id*) becomes false.

3.6. Communication

3.6.1. Clock-Synchronous Semantics

In clock-synchronous semantics, we only allow asynchronous communication (the send action). The reason is that in synchronous communication the caller must wait until the callee returns. In the clock-synchronous model, the callee is performing its own step when it receives the call and can respond to the call only in the next step at the next tick of the clock. Since by the perfect technology assumption an edge is taken in zero time, the caller cannot wait for the callee to do its work. We therefore have no synchronous communication in clock-synchronous semantics.

Figure 9 shows the execution algorithm for the multi-object clock-synchronous semantics. All existing objects wait simultaneously for changes until the global clock ticks. Next, all existing objects perform a step in parallel (multistep). The execution of a multistep is described in detail in Fig. 10. It is a straightforward extension of the algorithm for single steps presented above in Fig. 6. All updates to all variables are made in parallel and simultaneously; for example, the configurations of the existing objects are updated simultaneously.

The only thing different is that we now have a primed input set for each object *id* to which updates (events generated) are made. This is done for the following reason. Events are generated while the actions are being executed (in part 2 of Fig. 10). But then the input set is still filled with original input events. Only in part 4 of our algorithm is the input set emptied. If generated events were to be put in the input set immediately, the system would no longer know which input events to remove, and which to keep. We therefore add generated events to primed input sets rather than to the original unprimed ones.

While true do
- Received changes for existing objects until clock ticks
- Execute a step for every existing object (see Fig. 10)

Fig. 9. Execution algorithm for the multi-object clock-synchronous semantics.

1. For every existing object *id*, compute its step *nextstep(C[id],[id])* using the algorithm in Fig. 4
2. For every existing object, execute for every edge in its step the actions (only for updating primed variables)
3. For every existing object
   - Compute its next configuration
   - Update its current configuration *C[id]* with its next configuration
4. For every existing object id
   - Update the variables with the values of their primed counterparts (including the input set)
   - Empty its primed input set *I′[id]* (that contains the events generated in the step itself)
   - Switch some timers on and some timers off

Fig. 10. Algorithm to execute a step in the multi-object clock-synchronous semantics.

3.6.2. Clock-Asynchronous Semantics

In clock-asynchronous semantics both asynchronous and synchronous communication is allowed. Synchronous communication is allowed because each object instantly reacts to the events it receives and hence is always ready to synchronise with another object. We assume that only a single object is active during a single step. We can justify this by the perfect technology assumption: single

While true do
- Receive changes for objects (external event, temporal event)
- Repeat: if there is an object with an unstable stage, then execute a step for this object (see Fig. 12) Until all objects have a stable state

Fig. 11. Execution algorithm for the multi-object clock-asynchronous semantics.

1. For the object do
   - Compute *nextstep(C,I)* using the algorithm in Fig. 4
   - For every edge in the step execute the actions (only updating primed variables). This requires maintaining a call stack for synchronous calls
   - Compute the next configuration
   - Update the current configuration *C* with the next configuration
   - Empty the input set *I*
   - Update the local variables with the values of their primed counterparts
   - Switch some timers on and some timers off
2. For every existing object *id* do
   - Update the input set *I[id]* with the value of the primed input set *I'[id]* (may contain events generated in current step)
   - Empty the primed input set *I'[id]*

Fig. 12. Algorithm to execute a single-object step.

steps and supersteps do not take time to execute. The execution algorithm for the multi-object clock-asynchronous semantics is shown in Fig. 11.

The algorithm to execute a single step for one single object *id* is shown in Fig. 12. The algorithm defines a run-to-completion semantics. The definition is a straightforward adaptation of the step execution algorithms shown before in Figs 6 and 10. Note that the updates to input sets are made to all objects, rather than one.

What remains is to define the meaning of synchronous calls as part of the execution of actions. If an object calls operation *oper* of object *id*, then first *id* should exist, so *Exists*(*id*), and second, *id* should process the call by executing a step itself. Since *id* itself may have a non-empty input set *I*[*id*], this input set should be remembered before the call operation is processed and placed back after *id* has finished executing its step. When *id* finishes taking the step, the call action has finished and the caller proceeds with its own step. So, with call actions nested steps are introduced. This is a run-to-completion semantics, because it says that a call action is executed only when the called action is executed.

We illustrate this by means of a small example. In Fig. 13 we show two objects $O1$ and $O2$. Object $O2$ can call operation f of object $O2$. Suppose each object is in its initial configuration, so $C[O1] = \{O1.root,\mathsf{M}\}$ and $C[O2] = \{O2.root,\mathsf{P},\mathsf{R}\}$, and that simultaneously object $O1$ receives event d and object $O2$ receives event g. Then according to our algorithm in Fig. 11, either $O1$ starts processing first and then $O2$, or the other way around. Let us assume that $O1$ starts processing first and starts executing a step according to the algorithm in Fig. 12. At some point, $O1$ calls operation f of object $O2$. Then, $I[O2]$ still contains g. Thus, this input is remembered and temporarily removed from $O2$'s queue. Next, $O2$ executes the call operation and reaches configuration

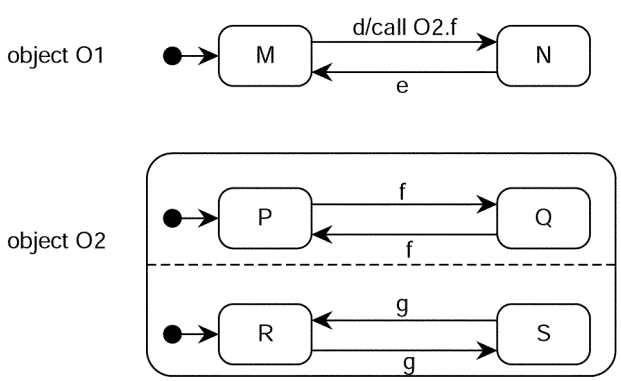

Fig. 13. Example of synchronous communication.

$\{O2.root,\mathsf{Q},\mathsf{R}\}$. Note that $O1$ is still busy taking its step. Then the old input g is copied back to $I[O2]$. Because the call action has finished, $O1$ can finish its step. Next, $O1$ reaches configuration $\{O1.root,\mathsf{N}\}$. Then $O2$ can react to event g and it reaches configuration $\{O2.root,\mathsf{Q},\mathsf{S}\}$.

3.7. Related Work

Our statechart semantics is an object-oriented version of the semantics for STATEMATE statecharts defined by Harel and Naamad [24], and defined more formally by Damm et al. [27]. Our semantics adds creation and deletion and communication to the STATEMATE statecharts semantics.

The RSML notation defined by Leveson et al. [33] is a variant of statecharts, for which a STATEMATE-like semantics is used, as we do. They consider single statecharts without communication and without object creation and deletion.

In a previous paper [26] we have defined our statechart semantics formally but without giving execution algorithms and without providing any details on how the semantics could be applied, nor on how it can be embedded in a method. (We prefer not to call our semantics a UML statechart semantics, since there seems to be a consensus in the UML community that a semantics for UML statecharts must be on the implementation level.) Because of the perfect technology assumption, our statechart semantics is considerably simpler than the informal OMG semantics for UML statecharts. Our semantics is based upon an earlier requirements-level semantics for UML with simple state-transition diagrams given by Wieringa and Broersen [34]. The contribution of this paper lies in the definition of a precise requirements-level statechart semantics with communication and object creation, and in the combination of this with informal techniques and model checking.

Most formalisations of UML statecharts [35–40] are implementation-level semantics. They all deal with one simple statechart only, and often leave out certain features, such as communication or real time, because these features cannot be handled easily by the semantic mechanism proposed by the authors. Examples of semantic mechanisms used are graph transformations [36], inference rules [35] and rewrite rules [37].

Betty Cheng's group has developed a general framework for deriving formal specifications from UML class and state diagrams [41]. They focus on the domain of embedded systems. They translate the UML metamodel, which is the syntax definition of the UML, into the metamodel of a formal language by a homomorphic mapping. The formal languages they consider are VHDL and the input language of Spin. The semantics of the target languages then give a semantics to the UML.

Harel and Kupferman [42] give an executable object model for Mealy diagrams, i.e. statecharts without hierarchy or parallelism. Their semantics is on the implementation level. They focus on communication between different objects and on the possibilities of deadlock in a clock-asynchronous semantics. Our treatment of communication stays at the requirements level, where no deadlock is possible. We consider both a clock-synchronous and clock-asynchronous semantics.

4. Model Checking

4.1. Tool Support

The semantics presented in Section 3 has been implemented in the CASE tool TCM as follows (compare Fig. 1).

- TCM contains a new statechart editor. TCM can generate from a single statechart automatically an input for a symbolic model checker, NuSMV. The semantics NuSMV attaches to the input coincides with our clock-asynchronous semantics. TCM uses the encoding suggested by Chan et al. [43] for (Nu)SMV; their statechart semantics is similar to our clock-asynchronous statechart semantics.

  If a property does not hold, the model checker generates a counterexample in the form of a trace through the model checker input. This trace is fed back to TCM, which shows the counterexample in a way understandable to the analyst, namely as a path through the statechart. However, the feedback may still be difficult to interpret, since the path may not provide sufficient details. Current work includes translating the feedback of the model checker into a UML sequence diagram, which can show more details than a path.
- TCM contains an editor for collections of flat statecharts (communicating Mealy diagrams with parallelism) and can generate the LTS that is the semantics of this collection according to Section 3. In this case, counterexamples generated by the model checker are not yet fed back to TCM but must be interpreted by the analyst. This editor interfaces with Kronos.

The example discussed below has been verified in two ways, namely as a statechart and as a collection of parallel Mealy diagrams. To do this, the statecharts presented below were transformed manually into a singe statechart and a collection of Mealy diagrams, respectively. Both versions of the specification gave the same results.

The logics used most often in model checking are Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). LTL formulas express properties of a single run (i.e., a trace of one possible system behaviour). CTL formulas express properties of states, and of the set of all possible runs starting in the state. In LTL the future of a state in a run is inevitable, whereas in CTL a state usually has many different possible futures. Thus, generally speaking, CTL formulas express possibility properties, whereas LTL formulas express properties that are inevitable [44]. For example, 'the system cannot deadlock' is a CTL property, whereas 'the system will not deadlock' is an LTL property. There are, however, requirements that can be specified both in LTL and CTL. Nevertheless, LTL and CTL are incomparable with respect to expressiveness: there are requirements that can specified in LTL but not in CTL (e.g., P6 in Fig. 17) and vice versa (e.g., P2 in Fig. 17). In formal requirements engineering, usually LTL is used, for example in KAOS [45].

A particular class of LTL formulas are the so-called strong fairness constraints. Strong fairness constraints are used to prevent starvation of some part of a process by forbidding to stay forever in some loops. Below in the example we will show why strong fairness constraints are useful. Strong fairness constraints can be combined with both LTL and CTL [46], but here we only combine them with LTL constraints.

Models and properties may quantify time, i.e., attach a precise duration to time intervals. This is called real time. There are two flavours of real-time models: dense real time and discrete time. In dense real-time models, time is represented by a real variable, whereas in discrete time models time is represented by a integer variable. Dense time models have the property that between any

two points in time there exists another point in time. Discrete time models do not have this property, and they are therefore in general considered to be less realistic than dense time. But discrete time models are easier to analyse than dense time models. Our semantics uses dense time (see Section 3.3).

Dense time properties can be specified with timed CTL (TCTL). TCTL is an extension of CTL with real time [47]. It allows the user to quantify time periods. We have defined an extension of TCTL with actions [48]. This extension can be model checked by Kronos [10] via a reduction to TCTL.

If the property does not refer to real time, then for some class of real-time models we can switch from dense time to discrete time, without invalidating the property to be checked [49,50]. The real-time models used in our semantics of OO statecharts fall into this class. Note that discretisation does not work for properties referring to real time, because the discretised model can have slightly different timing behaviour than the original model.

In our verification approach, we use both dense time models and discrete time models. With discrete time models only properties not referring to real time, i.e. both CTL and LTL, can be verified. With dense time models, CTL and TCTL properties can be verified.

We do not want to restrict ourselves to one particular model checker, because no model checker is usable for each purpose. Which model checker is most suited depends upon the properties of the system being modelled and the requirement being verified. We have implemented an interface to NuSMV [9] and Kronos [10]. Table 3 shows the features of these model checkers. Each of the model checkers has a particular logic to state desired properties; the logic's expressiveness defines which properties can be stated. NuSMV represents states symbolically, i.e. implicitly, by predicates, whereas Kronos represents states in an enumerative way, i.e., explicitly.

We have extended NuSMV with strong fairness to NuSMV [51] by implementing an algorithm defined by Kesten et al. [46]. Although strong fairness properties can be expressed in LTL and thus can be model checked with the standard NuSMV LTL algorithm, it is sometimes more efficient to encode them separately and use a specific model-checking algorithm [51]. Below in the example we will show why strong fairness constraints are useful.

In future work, we intend to use Spin [52]. Existing encodings for statecharts in Spin suggest that using Spin will be less efficient, as these encodings require more variables to encode a statechart [53,54]. In addition, Spin does not represent states symbolically but enumeratively, and it uses an interleaving semantics [52]. We therefore expect that Spin will not perform as well as NuSMV.

Table 3. Model checkers used and their features

Model checker	Time model	Encoding	Logic
NuSMV	Discrete	Symbolic	Computation Tree Logic (CTL) Linear Temporal Logic (LTL)
NuSMV$_{fair}$	Discrete	Symbolic	LTL with strong fairness, LTL
Kronos	Dense	Enumerative	Timed CTL (TCTL), CTL

4.2. Example

The Electronic Ticket System (ETS) allows railway passengers to buy electronic tickets via a personal digital assistant (PDA) and a smart card. The example is adapted from a paper of Reif and Stenzel [55]. A ticket is stored as a virtual, non-physical entity on the smart card. Passengers acquire their tickets through a wireless connection with the computer system of the railway company and pay through a connection with the bank clearing house's computer. In the train, the ticket collector uses a similar PDA to check the ticket's validity and to stamp it. It is possible to get a refund for unused or partly used tickets; however, a ticket collector or a railway clerk has to initiate the refund to prevent fraud.

Figure 14 shows the mission statement of ETS, and Fig. 15 shows the context and a requirements-level architecture of ETS. The architecture is requirements-

Name: Electronic Ticket System (ETS)
Purpose: Sell virtual railway tickets and register their use using a PDA and a smart card.
Responsibilities: Sell tickets; show tickets; stamp tickets (railway personnel only); refund tickets (railway personnel only).
Exclusions: The ETS does not provide travel planning facilities. It only handles tickets for one traveller.

Fig. 14. Mission of ETS.

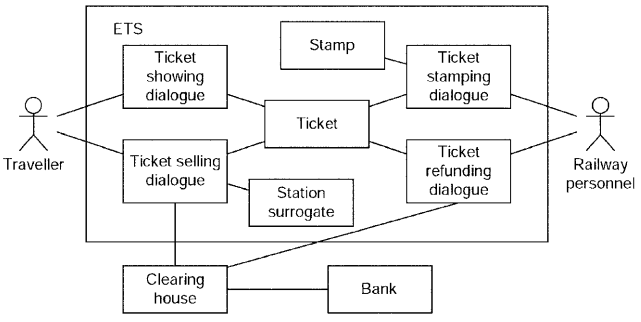

Fig. 15. ETS context and requirements-level decomposition.

level because it is motivated purely in terms of the desired functionality of ETS and of the need to interact with certain external entities in the context of ETS. The requirements-level architecture abstracts from the physically distributed nature of ETS. Note the difference between stations and tickets: stations are not a part of the system; the ETS only contains information about stations in the surrogates. The (electronic) tickets and stamps, in contrast, are part of the system.

The lines in the architecture diagram represent communication channels. The nodes represent object classes. The nodes enclosed in the ETS node represent classes of ETS components. The mission statement and the context and architecture diagram are examples of informal diagrams that are part of a larger ETS specification that contains other parts, both formal and informal. Examples of formal specification parts are the ticket life cycle shown in Fig. 16 and the ticket selling dialogue of Fig. 2.

Figure 17 lists six properties that must be brought about by the ETS. This is the set E of emergent properties mentioned in the systems engineering argument that A and S entail E (Section 1.2). The ETS can bring these properties about only in a joint effort with external entities. Some assumptions about the environment that we note straight away are listed in Fig. 18. Assumptions A1 and A2 help in simplifying the system; they are used in the Kronos model. Assumptions A3 and A4 we discovered when model checking P6; we explain them in the next subsection.

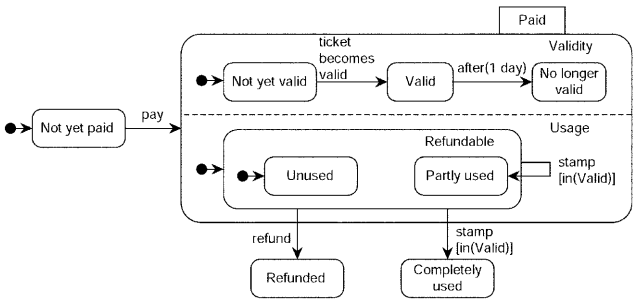

Fig. 16. Ticket life cycle.

P1 For each payment, exactly one ticket is created.
P2 An unused ticket can be used or refunded.
P3 A used ticket cannot be used again nor refunded.
P4 A refunded ticket cannot be used any more.
P5 A ticket cannot be used for more than one day.
P6 A finite time after a ticket selling dialogue is started, ETS is ready to sell another ticket again.

Fig. 17. Desired properties *E.* to be brought about by the ETS.

A1 Railway personnel does not request to refund a ticket while stamping one.
A2 The traveller does not request to buy a ticket while railway personnel is refunding another one.
A3 The traveller only selects a route or ticket type finitely often during one selling dialogue.
A4 During a dialogue, the traveller will provide relevant input in finite time.

Fig. 18. Some assumptions needed for the systems engineering argument.

4.3. Model Checking

We model checked the properties in Fig. 17 with NuSMV 2.0, NuSMV$_{fair}$ 2.0 and Kronos, using both the statechart version (with NuSMV, NuSMV$_{fair}$) and the Mealy diagrams version (with Kronos) of the statechart diagrams. Note that the boxes in the architecture diagram represent classes. To do model checking, we must instantiate each class to a finite number of objects. The specification S_{ETS} used in model checking then consists of the statecharts for each of the individual objects in the system.

In our model-checking experiments, we instantiated the most important classes to a few objects. For NuSMV (statechart version), we have used three different models, in which we always had one ticket-selling dialogue, but one to three different ticket life cycles and corresponding refund dialogues. Other classes we did not instantiate. We did not use Assumptions A1 and A2 here.

For Kronos (Mealy machines version), we have used one ticket, one ticket-selling dialogue, one stamping dialogue and one refund dialogue. The ticket-selling dialogue, stamping dialogue and refund dialogue were interleaved with each other, i.e. only one of them could be active at the same time. This is justified by Assumptions A1 and A2. Other classes we did not instantiate. We used Assumptions A1 and A2 to reduce the state space of the model, otherwise Kronos could not be used. In addition, in the Kronos model we used the observation that in a state only those events need to be considered that trigger some relevant edges. By not considering irrelevant events, the search space is reduced considerably.

Properties P1 to P5 were checked with NuSMV using CTL. Properties P2, P4 and P6 were also checked with NuSMV and NuSMV$_{fair}$ using LTL. We have used discretised timers, using time units of 30 seconds. For example, the timeout of 1 day in Fig. 16 became a timeout of 2880 time units. Because the model includes real time, we additionally checked properties P1 to P5 with Kronos using CTL and TCTL.

We now discuss the most interesting properties in more detail.

P1
For each payment, exactly one ticket is created. This property cannot be formalised in CTL or LTL, as it would require an operator reasoning about creation and deletion of new elements, whereas in standard model checking the set of elements is fixed. Recently, DiStefano et al. [56] have defined an extension of LTL and defined a model-checking algorithm for this extension that makes it possible to model check creation and deletion in object-oriented models. We plan to implement this algorithm in NuSMV. For the Kronos model, we checked a variant of the property.

P2
An unused ticket can be used or refunded. We formalise this property in CTL as:

AG (Unused → ((EF Completely used) ∧ EF Refunded))

This property is not true. The model checker returns a counterexample in which the ticket is in Unused while event refund occurs. Then node Completely used can no longer be reached, even though the ticket is still in Unused. We change the property and try it again.

AG ((Unused ∧ ¬refund) → ((EF Completely used) ∧ EF Refunded))

The changed property P2b is also false. The model checker returns a counterexample in which the ticket is never used and eventually becomes invalid, by entering node No longer valid. Then node Completely used becomes unreachable. We change P2b as follows and check it again.

AG ((Unused ∧ ¬refund) → (EF (Completely used ∨ No longer valid) ∧ EF Refunded))

This property, P2c, is verified to be true.

Even though the property is a possibility property and thus cannot be expressed in LTL, we write the following LTL formula. It states that if a ticket is unused, it inevitably will become, sometime in the future, either completely used, or no longer valid, or refunded.

G (Unused → (F (Completely used ∨ No longer valid ∨ Refunded)))

This property is verified to be true.

P3
A used ticket cannot be used again nor refunded. This is a nice example of an informal specification that can formalised in different ways. One of us formalised this as

AG (Completely used → ¬EF (Partly used ∨ Refunded))

which means that a ticket that is completely used cannot be used again or refunded. This property was verified to be true. But another one of us formalised it as

AG ((Completely used ∨ Partly used) → ¬EF (Partly used ∨ Refunded))

which means that a ticket that has been stamped for at least part of its validity cannot be used again or refunded. This property was verified to be false, as it should be. See Fig. 16 for the difference between Partly used and Completely used. Both properties are possibility properties; therefore they cannot be expressed in LTL. By rewriting the property slightly (as is done in the previous and next items), the property can be formalised in LTL.

P4
A refunded ticket cannot be used any more. We write this property in CTL.

AG (Refunded → ¬EF (Refundable ∨ Completely used))

This property is true. The following LTL formula expresses a similar property: ‘a refunded ticket will not be used any more’.

G (Refunded → ¬F (Refundable ∨ Completely used))

This property is also true. These two properties are not equivalent.

P5
A ticket cannot be used for more than one day.

We have verified the following TCTL formula with Kronos:

AG (¬EF (stamp ∧ $EF^{\geqslant 24h}$ stamp))

The property is a real-time property and cannot be verified using a discretised model. So NuSMV could not be used. With forward reachability analysis, Kronos ran out of memory. But with backward analysis, Kronos reported that ¬EF (stamp ∧ $EF^{\geqslant 24h}$ stamp) is false in all states of the system, which implies that the property is true.

P6
A finite time after a ticket-selling dialogue is started, ETS is ready to sell another ticket again. See Fig. 2 for the ticket-selling dialogue. This is formalised in CTL as follows:

AG (request to buy ticket → AF Ready to sell)

According to NuSMV, this property is false, since NuSMV thinks that it is possible that user will do

select ticket type forever without every doing done or abort. This means that for NuSMV the property

EF EG Ticket type known

is true in the initial state. The formula says that it is possible, after some time, to stay forever in state node Ticket type known(see Fig. 2). But we do not want this property to be true. How can we change the specification of ETS so that it will make this formula false? Note that just adding an assumption that forbids to stay forever in this state node is not enough.

In every state, irrelevant events that have no effect can occur forever while a relevant event may never occur. For example, in node Ticket sale offered irrelevant event request to buy ticket can occur over and over again while relevant events accept offer or decline offer never occur. This would cause the system to remain in node Ticket sale offered forever. So, even if there are no visible loops in the statechart model, it is possible that the system will stay forever in a state! To rule out all these situations, we have added *strong fairness conditions* on edges, which specify that all these loops are finite. For example, a strong fairness condition for edge b in Fig. 19 says that if N is active infinitely often, then edge b is taken infinitely often too. So, a strongly fair run will never reach a state from which N will be active infinitely often, but edge b is never taken. Strong fairness conditions have been introduced by Gabbay et al. [57].

The NuSMV$_{fair}$ model checker only takes into account runs that satisfy these strong fairness constraints. To see why these strong fairness constraints are what we want, consider again node Ticket type known and event select tickettype. In a strongly fair run, it is impossible to iterate infinitely often over event select tickettype, without also performing an abort or done infinitely often. This allows us to prove that after any request to buy a ticket, the system reaches the Ready to sell state in finitely many steps. This is the property we wanted. NuSMV$_{fair}$ requires the desired property to be specified in LTL. In LTL, P6 becomes

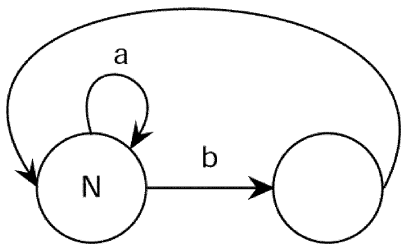

Fig. 19. Example to illustrate strong fairness condition.

G (request to buy ticket → F Ready to sell)

This was verified to be true. Note that the strong fairness conditions added in this case are implied by assumptions A3 and A4 on the environment. If the environment violates these assumptions, it is unknown whether the desired property P6 holds. The assumptions say that under normal usage, P6 holds.

For tickets, there is also a potential infinite loop: one might stamp a partially used ticket over and over again. But here, strong fairness is not necessary to prove termination, since the timeout in node Valid guarantees that stamp cannot occur infinitely often. We have verified this in property P5.

We conclude by showing the performance characteristics of model checking. All the experiments were performed on a Sun Ultra 10 with 256 MB of main memory.

Table 4 shows the performance characteristics of the statechart version, which used NuSMV and NuSMV$_{fair}$. We used a monolithic transition relation. We have used three different statechart models. With option NuSMV$_{fair}$ (LTL), we either do not use strong fairness constraints (P2d, P4b) or encode them as antecedent of the LTL property (P6). With option NuSMV$_{fair}$ we encode strong fairness constraints separately from the LTL property; this option is only used for P6. We were unable to compute the number of reachable states of Model C.

Here are some remarks on the results. Property P2b requires a large number of BDD nodes because a counterexample is returned in which a timeout is generated after 2880 time units. It is quite interesting to see that the performance characteristics of NuSMV with LTL and NuSMV$_{fair}$ with LTL are quite different. Which algorithm performs better apparently depends upon the LTL formula being checked. In this example, NuSMV with LTL performs better than NuSMV$_{fair}$ on larger models, even for strong fairness constraints (P6). We think that this is because there is only a limited number of strong fairness constraints needed in this example (around 10). However, in other examples [51] we had to use much stronger fairness constraints, and there we found that NuSMV$_{fair}$ performed better than NuSMV with LTL. So it cannot be stated beforehand which algorithm is going to perform better, as this heavily depends upon the model and property being checked. Finally, we note that we were unable to generate counterexamples for LTL specifications (with or without strong fairness), even in the simple model. The performance characteristics clearly show that model checking can (yet) only be applied to small-sized

Table 4. Performance characteristics of model checking different statechart versions. The number of BDD nodes is a measure of memory usage

	NuSMV (CTL)		NuSMV (LTL)		NuSMV_{fair} (LTL)		NuSMV_{fair}	
	Time (s)	BDD nodes (no.)	Time (s)	BDD nodes (no.)	Time (s)	BDD nodes (no.)	Time (s)	BDD nodes (no.)
Model A: 1 Ticket-selling dialogue, 1 Ticket, 1 Refund dialogue.2,267,520,000 reachable states.								
P2 a	43.88	88,629						
P2 b	152.98	145,997						
P2 c	44.85	51,231						
P2 d			763.45	50,664	136.63	229,397		
P3 a	43.39	51,463						
P3 b	45.05	77,223						
P4 a	43.52	38,584						
P4 b			43.46	51,345	63.46	111,956		
P6			77.14	361,353	117.27	147,343	71.98	271,841
Model B: 1 Ticket-selling dialogue, 2 Tickets, 2 Refund dialogues. 5,018,360,000,000,000 reachable states								
P2 a	152.31	1,650,749						
P2 b	–	–						
P2 c	87.13	253,783						
P2 d			–	–	1643.83	405,775		
P3 a	84.43	262,090						
P3 b	87.77	324,732						
P4 a	83.81	219,291						
P4 b			84.43	248,358	703.35	241,143		
P6			221.00	280,929	1607.91	278,292	895.82	97,879
Model C: 1 Ticket-selling dialogue, 3 Tickets, 3 Refund dialogues.Unknown number of reachable states								
P2 a	–	–						
P2 b	–	–						
P2 c	133.63	288,828						
P2 d			–	–	–	–		
P3 a	132.37	288,448						
P3 b	134.34	426,255						
P4 a	131.93	234,088						
P4 b			132.05	292,859	–	–		
P6			340.63	224,929	–	–	–	–

'–' denotes a timeout (>half an hour).

Table 5. Performance characteristics of model checking Mealy machines model with Kronos

	Time (s)
P1	1.7
P2	0.7
P3a	0.9
P4	– (0.9)
P5	– (0.8)

Model A': 1 Ticket selling dialogue, 1 Ticket, 1 Refund dialogue, 1 Stamping dialogue.1070 locations.

examples, due to the state space explosion. We analyse the cause of state space explosion in detail elsewhere [51].

Table 5 shows the performance characteristics of the Mealy machine models, which used Kronos. The size of the Mealy machine model we checked with Kronos has 1070 locations and two timers. Due to the timers, the state space is much larger than 1070 states. Since Kronos does not give performance characteristics, we used the operating system to get some performance results. If Kronos did not give a result, sometimes we checked an equivalent property (P4) or used a different search technique (P5, backward instead of forward analysis). P6 was not checked as Kronos does not support strong fairness constraints nor LTL.

4.4. Related Work

The KeY tool [58] extends the UML case tool 'Together' by formal verification. Its goal is to facilitate and promote formal verification in real-world applications. The tool is used in combination with Java and focuses on implementation-level semantics.

The Software Cost Reduction method [19–21] is similar in philosophy to our approach, but is based on a tabular notation rather than statecharts. It is supported by a tool set that interfaces with Spin.

Chan et al. [43] have model checked a single RSML statechart by defining a mapping from statechart syntax to the SMV input syntax, which makes model checking very efficient. We have used this encoding for our example. They only use CTL whereas we use CTL, LTL and strong fairness constraints. We do not know whether they have implemented this mapping in a statechart editing tool. Finally, they do not embed their approach in a method.

Latella et al. [59] model check UML statecharts using Spin. They do not focus on how the model checker is integrated with the UML design tool. Their statechart semantics does not handle multiple statecharts or real time.

Lilius and Porres Paltor [53] have developed a tool to model check UML statecharts using Spin. The tool they developed translates counterexamples returned by Spin back in terms of sequence diagrams. It is unclear whether the tool is integrated with a UML design tool.

5. Conclusions and Further Work

5.1. Summary

In this paper we presented a requirements-level semantics for object-oriented statecharts and showed that it can be used for requirements-level model checking in a goal-oriented approach. Model checking can be used to uncover hidden assumptions in formal requirements specification. It can also be used to identify ambiguities in an informal specification. It is useful in this respect to have different people produce a formalisation of one set of informally specified properties. Our example specification briefly illustrates the way in which formal and informal elements of a system specification can be combined.

5.2. Maturity of Model Checkers

Our experience with model checking is that it is not yet the push-button technology that some of us would like it to be. When we formalise assumptions A, a specification S and desired properties E, we would like a model checker to tell us whether $A \wedge S \models E$ and, if the entailment does not hold, give us a counterexample. However, in actual practice we found that the model checkers that we used had to be used 'smartly' in order to produce the desired result. We defined a general format for the LTS that, with minor adaptations, can be input to various model checkers. However, this input format is very inefficient in memory usage. NuSMV is a symbolic model checker, which can be given a much more efficient input by encoding our LTS semantics symbolically, using the input language of NuSMV optimally. Therefore, in this paper we have used the symbolic encoding for statecharts that has been defined by Chan et al. for RSML [43]. This decreased the amount of time and memory needed significantly, but not enough. NuSMV is sensitive to the order in which the state variables are declared. Two different orderings may differ wildly in their model checking execution time, with differences between seconds or days to verify a property. We checked properties with NuSMV using an efficient ordering, which we found after a few iterations in which we used NuSMV interactively and let NuSMV itself reorder the ordering of variables while it was building an internal model for model checking. These experiences indicate that the ideal of adding a user-friendly front end such as TCM to a set of model checkers is currently not yet attainable. However, we expect the technology to improve in the near future and we think NuSMV is a suitable tool to check properties of statecharts. Kronos is not a symbolic model checker and could only be used for a simplified version of the model.

5.3. Further Work

We plan to extend our tool support for model checking in various ways. First, we intend to interface Kronos directly with the statechart editor. Second we intend to use Spin for model-checking statecharts, as Spin is used quite frequently in other approaches that model check statecharts. Third, we are currently working on tool support for translating the feedback of the model checker into a UML sequence diagram. Fourth, we plan to look at model checking of creation and deletion of objects, by applying some recent work of DiStefano et al. [56].

Another plan for future work is to provide support in the specification of properties in the form of templates of frequently occurring properties. Dwyer et al. [60] have found a list of eight patterns in temporal properties and in the KAOS project [15,45] goal-oriented refinement patterns have been identified. We intend to look at the problem frames identified by Jackson [18] to see if we can build a library of formally verified patterns, plus guidelines to recognise and use these patterns.

Acknowledgements. Rik Eshuis was supported by NWO/SION, grant no. 612-62-02 (DAEMON). David Jansen was partially supported by NWO/SION, grant no. 612-323-419 (DEIRDRE).

References

1. Harel D. Statecharts: a visual formalism for complex systems. Sci Comput Program 1987;8(3):231–274
2. Harel D, Pnueli A. On the development of reactive systems. In: Apt K (ed). Logics and models of concurrent systems. NATO ASI Series. Springer, Berlin, 1985, pp 477–498

3. Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W. Object-oriented modeling and design. Prentice-Hall, Englewood Cliffs, NJ, 1991
4. Rumbaugh J, Jacobson I, Booch G. The Unified Modeling Language reference manual. Addison-Wesley, Reading, MA, 1999
5. von der Beeck M. A comparison of statecharts variants. In: Langmaack H, de Roever W-P, Vytopil J (eds). Formal techniques in real-time and fault-tolerant systems. Lecture Notes in Computer Science 863. Springer, Berlin, 1994, pp 128–148
6. OMG. Action semantics for the UML (OMG ad/2001-03-01), 2001. URL: http://www.umlactionsemantics.org
7. Dehne F, Wieringa R, van de Zandschulp H. Toolkit for conceptual modeling (TCM): user's guide and reference. Technical report, University of Twente, 2000. URL: http://www.cs.utwente.nl/~tcm
8. McMillan KL. Symbolic model checking. Kluwer, Dordrecht, 1993
9. Cimatti A, Clarke E, Giunchiglia F, Roveri M. NuSMV: a new symbolic model checker. Int J Software Tools Technol Transfer 2000;2(4):410–425
10. Yovine S. KRONOS: a verification tool for real-time systems. Int J Software Tools Technol Transfer 1997;1(1/2):123–133
11. Smith CJ. Synonyms discriminated. G. Bell, London, 1926
12. Harel D, Rumpe B. Modeling languages: syntax, semantics and all that stuff. Part I: The basic stuff. Technical report MCS00-16, Weizmann Institute of Science, 2000. URL: http://www.wisdom.weizmann.ac.il/~harel
13. Wieringa RJ. Postmodern software design with NYAM: not yet another method. In Broy M, Rumpe B (eds). Requirements targeting software and systems engineering. Lecture Notes in Computer Science 1526. Springer, Berlin, 1998, pp 69–94
14. Wieringa RJ. Design methods for software systems: Yourdon, Statemate and the UML. Morgan Kaufmann, San Mateo, CA, (to be published)
15. Darimont R, van Lamsweerde A. Formal refinement patterns for goal-driven requirements elaboration. In: Fourth ACM symposium on the foundations of software engineering (FSE4), 1996, pp 179–190
16. van Lamsweerde A, Darimont R, Massonet P. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In: Proceedings of the second IEEE international symposium on requirements engineering, IEEE Computer Society Press, Los Alamitos, CA, 1995
17. Gunter CA, Gunter EL, Jackson MA, Zave P. A reference model for requirements and specifications. IEEE Software 2000; 17(3):37–43
18. Jackson MA. Problem frames: analysing and structuring software development problems. Addison-Wesley, Reading, MA, 2000
19. Heitmeyer C, Kirby J, Labaw B, Bharadwaj R. SCR: a toolset for specifying and analyzing software requirements. In: Hu AJ, Vardi MY (eds). Proceedings of the 10th international computer aided verification conference. Lecture Notes in Computer Science 1427. Springer, Berlin, 1998, pp 526–531
20. Heitmeyer CL, Jeffords RD, Labaw BG. Automated consistency checking of requirements specifications. ACM Trans Software Eng Methodol 1996;5(3):231–261
21. Parnas DL, Madey J. Functional documents for computer systems. Sci Comput program 1995;25:41–61
22. McMenamin SM, Palmer JF. Essential systems analysis. Yourdon Press/Prentice-Hall, Englewood Cliffs, NJ, 1984
23. Berry G, Gonthier G. The ESTEREL synchronous programming language: design, semantics, implementation. Sci Comput Program 1992;19(2):87–152
24. Harel D, Naamad A. The STATEMATE semantics of statecharts. ACM Trans Software Eng Methodol 5(4):293–333
25. OMG. Unified Modeling Language version 1.4, 2001
26. Eshuis R, Wieringa R. Requirements-level semantics for UML statecharts. In: Smith S, Talcott C (eds). Proceedings of FMOODS 2000, IFIP TC6/WG6.1. Kluwer, Dordrecht, 2000, pp 121–140
27. Damm W, Josko B, Hungar H, Pnueli A. A compositional real-time semantics of STATEMATE designs. In: de Roever W-P, Langmaack H, Pnueli A (eds). Proceedings of COMPOS '97. Lecture Notes in Computer Science 1536. Springer, Berlin, 1998, pp 186–238
28. Harel D, Gery E. Executable object modeling with statecharts. IEEE Comput 1997;30(7):31–42
29. Selic B, Gullekson G, Ward P. Real-time object oriented modeling. Wiley, New York, 1994
30. Eshuis R, Wieringa R. Requirements-level semantics for UML statecharts. Technical report TR-CTIT-00-07, University of Twente, 2000
31. Elmasri R, Navathe SB. Fundamentals of database systems, 2nd edn. Benjamin Cummings, Redwood City, CA, 1994
32. Pnueli A, Shalev M. What is in a step: on the semantics of statecharts. In Ito T, Meyer AR (eds). Theoretical aspects of computer software. Lecture Notes in Computer Science 526. Springer, Berlin, 1991, pp 244–265
33. Leveson NL, Heimdahl MPE, Hildreth H, Reese JD. Requirements specification for process-control systems. IEEE Trans Software Eng 1994;20(9):684–707
34. Wieringa R, Broersen J. A minimal transition system semantics for lightweight class- and behavior diagrams. In: Broy M, Coleman D, Maibaum TSE, Rumpe B (eds). Proceedings of PSMT'98, Technische Universität München, TUM-I9803, 1998
35. von der Beeck M. Formalization of UML-statecharts. In: Gogolla M, Kobryn C (eds). Proceedings of 'UML' 2001. Lecture Notes in Computer Science 2185. Springer, Berlin, 2001, pp 406–421
36. Kuske S. A formal semantics for UML state machines based on structured graph transformations. In: Gogolla M, Kobryn C (eds). Proceedings of 'UML' 2001. Lecture Notes in Computer Science 2185. Springer, Berlin, 2001, pp 241–256
37. Kwon G. Rewrite rules and operational semantics for model checking UML statecharts. In: Evans A, Kent S, Selic B (eds). Proceedings 'UML' 2000. Lecture Notes in Computer Science 1939. Springer, Berlin, 2000, pp 528–540
38. Latella D, Majzik I, Massink M. Towards a formal operational semantics of UML statechart diagrams. In: Ciancarini P, Fantechi A, Gorrieri R (eds). Proceedings of FMOODS'99, IFIP TC6/WG6.1. Kluwer, Dordrecht, 1999, pp 331–347
39. Lilius J, Porres Paltor I. Formalising UML state machines for model checking. In: France R, Rumpe B (eds). Proceedings of 'UML' '99. Lecture Notes in Computer Science 1723. Springer, Berlin, 1999., pp 430–445
40. Reggio G, Astesiano E, Choppy C, Hussmann H. Analysing UML active classes and associated state machines: a lightweight formal approach. In: Maibaum TSE (ed). Proceedings of FASE 2000. Lecture Notes in Computer Science 1783. Springer, Berlin, 2000, pp 127–146
41. McUmber WE, Cheng BHC. General framework for formalizing UML with formal languages. In: Proceedings of the 23rd international conference on software engineering (ICSE '01). IEEE Computer Society, 2001, pp 433–442
429. Harel D, Kupferman O. On the behavioral inheritance of state-based objects. Technical report MCS99-12, Weizmann Institute of Science, 1999. URL: http://www.wisdom.weizmann.ac.il/~harel
43. Chan W, Anderson R, Beame P, Burns S, Modugno F, Notkin D, Reese J. Model checking large software specifications. IEEE Trans Software Eng 1998;24(7):498–520
44. Lamport L. What good is temporal logic? In: Mason REA (ed). Proceedings of the IFIP congress on information processing. North-Holland, Amsterdam, 1983, pp 657–667
45. Dardenne A, van Lamsweerde A, Fickas S. Goal-directed requirements acquisition. Sci Comput Program 1993;20(1–2):3–50
46. Kesten Y, Pnueli A, Raviv L. Algorithmic verification of linear temporal logic specifications. In: Larsen KG, Skyum S, Winskel G (eds). Proceedings of the international colloquium on automata, languages and programming (ICALP '98). Lecture Notes in Computer Science 1443. Springer, Berlin, 1998, pp 1–16

47. Alur R, Courcoubetis C, Dill D. Model-checking in dense real-time. Inform Comput 1993;104(1):2–34
48. Jansen DN, Wieringa RJ. Extending CTL with actions and real time. In: Proceedings on international conference on temporal logic 2000, pp 105–114
49. Asarin E, Maler O, Pnueli A. On discretization of delays in timed automata and digital circuits. In: de Simone R, Sangiorgi D (eds). Proceedings of Concur '98. Lecture Notes in Computer Science 1466. Springer, Berlin, 1998
50. Henzinger TA, Manna Z, Pnueli A. What good are digital clocks? In: Kuich W (ed). Proceedings of the international colloquium on automata, languages, and programming (ICALP'92). Lecture Notes in Computer Science 623. Springer, Berlin, 1992, pp 545–558
51. Eshuis R, Wieringa R. Verification support for workflow design with UML activity graphs. In: Proceedings of the 2002 international conference on software engineering (ICSE '02), 2002
52. Holzmann GJ. The model checker SPIN. IEEE Trans Software Eng 1997;23(5):279–295
53. Lilius J, Porres Paltor I. vUML: a tool for verifying UML models. In: 14th IEEE international conference on automated software engineering. IEEE Computer Society Press, Los Alamitos, CA, 1999, pp 255–258
54. Mikk E, Lakhnech Y, Siegel M, Holzmann GJ. Implementing statecharts in promela/spin. In: Proceedings of the second IEEE workshop on industrial-strength formal specification techniques, 1998, pp 90–101
55. Reif W, Stenzel K. Formal methods for the secure application of Java smartcards, December 1999. URL: http://www.informatik.uni-ulm.de/pm/kiv/projects/javacard_presentation.ps.gz
56. Distefano D, Rensink A, Katoen J-P. Model checking dynamic allocation and deallocation. CTIT technical report TR-CTIT-01-04, University of Twente, 2002
57. Gabbay D, Pnueli A, Shelah S, Stavi J. The temporal analysis of fairness. In: Conference record of the seventh annual ACM symposium on principles of programming languages, ACM, 1980, pp 163–173
58. Ahrendt W, Baar T, Beckert B, Giese M, Habermalz E, Hähnle R, Menzel W, Schmitt PH. The KeY approach: integrating object oriented design and formal verification. In: Ojeda-Aciego M, de Guzmán IP, Brewka G, Moniz Pereira L (eds). Proceedings of the eighth European workshop on logics in AI (JELIA). Lecture Notes in Computer Science 1919. Springer, Berlin, 2000, pp 21–36. URL: ftp://ftp.cs.chalmers.se/pub/users/reiner/jelia.ps.gz.
59. Latella D, Majzik I, Massink M. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Formal Aspects Comput 1999;11(6):637–664
60. Dwyer MB, Avrunin GS, Corbett JC. Patterns in property specifications for finite-state verification. In: Proceedings of the 1999 international conference on software engineering (ICSE '99), ACM, 1999, pp 411–421