Tool Support for Verifying UML Activity Diagrams

Rik Eshuis and Roel Wieringa

Abstract—We describe a tool that supports verification of workflow models specified in UML activity diagrams. The tool translates an activity diagram into an input format for a model checker according to a mathematical semantics. With the model checker, arbitrary propositional requirements can be checked against the input model. If a requirement fails to hold, an error trace is returned by the model checker, which our tool presents by highlighting a corresponding path in the activity diagram. We summarize our formal semantics, discuss the techniques used to reduce an infinite state space to a finite one, and motivate the need for strong fairness constraints to obtain realistic results. We define requirement-preserving rules for state space reduction. Finally, we illustrate the whole approach with a few example verifications.

Index Terms—Analysis, tools, software/program verification, model checking, state diagrams, workflow management.

1 Introduction

Activity diagrams are flowchart-like notations with constructs to express sequence, choice and parallel execution of activities [39]. Activity diagrams are part of UML, the de facto industry standard for modeling software designs. In this paper, we use activity diagrams for workflow modeling, that is, for the description of the flow of work orders through an organization. Workflow management systems (WFMSs) are software systems that route work orders through a collection of organizational actors (people or software) according to a workflow description in a given notation. Tools exist for the verification of performance properties of workflow models, such as throughput and workload [1], [29]. There is nowadays also a need to verify functional properties of workflow models because current WFMSs are being integrated with Enterprise Resource Planning, e-commerce applications, cross-organizational workflow, and flexible case management [4], [19]. Workflows in these applications may contain event-driven behavior, real-time events, unrestricted loops, parallelism, and distribution, and this makes it easy to specify workflow models with undesirable functional properties. These errors are very hard to spot using simple visual inspection of a workflow model. We therefore propose a tool for verifying functional properties of complex workflow models specified in UML activity diagrams.

Our tool is an extension of the graph editing tool TCM [11] that turns it into a graphical front end for a model checker. The analyst specifies a functional requirement and an activity diagram. The diagram is translated to the input of a model checker, a transition system (TS), according to our formal semantics [13]. The requirement is fed into the model checker. If the requirement is true of the activity diagram, the model checker answers "True;" otherwise, it gives a counterexample in the form of a trace through the underlying mathematical semantics of the activity diagram. TCM presents this trace in an understandable way as a path through the activity diagram.

The most commonly used temporal logics to specify functional requirements on transition systems are Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [9]. As we will argue in Section 5, the transition systems we are interested in must satisfy the property of *strong fairness*. This property can be expressed in LTL but not in CTL, so we must restrict ourselves to model checkers that support LTL as a requirement language. Furthermore, the model checker must use a special model checking algorithm for strong fairness. We have picked the NuSMV model checker [7], which we extended with an LTL model checking algorithm defined by Kesten et al. [31] that is specifically intended for strong fairness.

The structure of this paper is as follows: Section 2 starts with an explanation of the syntax of activity diagrams. Section 3 defines a formal semantics for safe activity diagrams, adapted from a formal semantics for unsafe activity diagrams that we published earlier [13]. (Readers not interested in the formal definition can skip the formulas and read the surrounding text.) Section 4 explains how we transform an infinite TS to a finite one while preserving relevant requirements. Section 5 explains why a TS needs to be extended with strong fairness and how this can be done. Section 6 then discusses the implementation of the generation of a TS with strong fairness constraints from an activity diagram in TCM. Section 7 gives some example verifications of requirements on an example workflow model. Section 8 defines reduction rules that reduce the size of the generated TS while retaining requirements, making model checking more efficient. Section 9 discusses related work and Section 10 summarizes our contribution and discusses some further work.

• *R. Eshuis is with the Department of Technology Management, Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands. E-mail: h.eshuis@tm.tue.nl.*
• *R. Wieringa is with the Department of Computer Science, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands. E-mail: roelw@cs.utwente.nl.*

Manuscript received 19 Mar. 2003; revised 20 Jan. 2004; accepted 8 Apr. 2004.
Recommended for acceptance by M. Broy.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0011-0303.

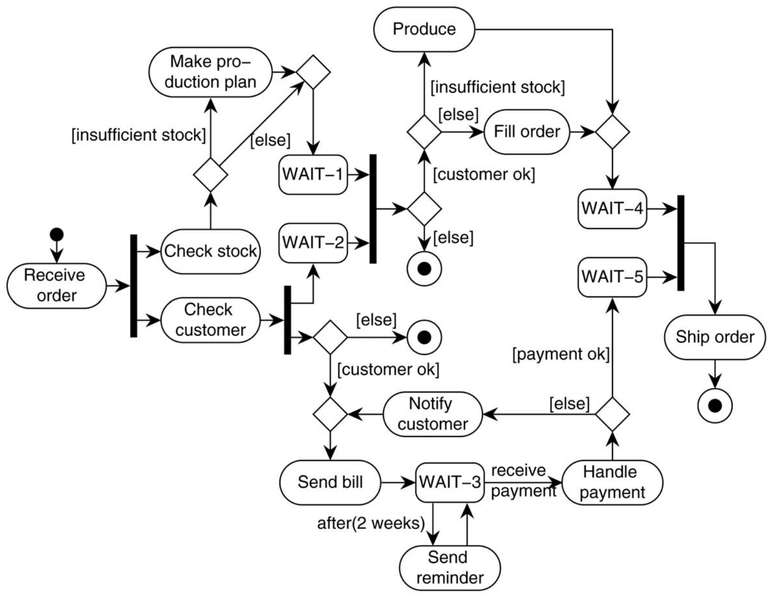


Fig. 1. Activity diagram of a production company.

2 Syntax of UML Activity Diagrams

Fig. 1 shows an example activity diagram specification of a workflow model. Ovals represent activity states (action states in UML terminology [39]) and rounded rectangles represent wait states. A bar represents an AND-node, which is either a fork or a join. A diamond represents an XOR-node, which is either a split or a merge. The workflow starts in the black dot and ends at a bull's eye. The example models the workflow of a small production company (adapted from [43]). After reception of an order, the finance department handles financial checks, billing, and payment, shown in the lower half of the diagram, and the production department handles producing and filling the order, shown in the upper half of the diagram. There are some interdependencies between these two processes, such as that an order will only be filled when the customer check is OK.

The intended informal meaning of the diagram is this. The diagram describes the behavior of a workflow management system (WFMS) and some actors (people or software) that interact with it. These actors handle a case, which is a work order that flows through an organization. A case has attributes, holding data relevant to the case. These may be documents or structured data. Case attributes are stored in a database accessible but external to the WFMS. Activity nodes represent activities performed by actors, such as updating a case attribute or searching a database. During an activity state, the WFMS is waiting for some actor to finish its activity. In a wait state node, the WFMS and actors are waiting for some event, such as the arrival of a message from some third party or the arrival of a deadline. Activity states and wait states take time.

Edges represent state transitions of the WFMS itself. A transition from a wait state is triggered by an external event and a transition from an activity state is triggered by the termination of the activity performed in the state. Each edge can be labeled by $e[g]/a$, where e is an event expression, g an optional guard expression, and a an optional action expression. An edge leaving an activity state node must not have an event label; such an edge is implicitly triggered by an activity termination event. An event can be temporal. A `when` event denotes an absolute time event, for example, `when(12:00hs)`, and an `after` event denotes a relative time event, for example, `after(5s)`, which means that 5 seconds after the corresponding edge became relevant, a timeout occurs. A guard expression can refer to variables of the activity diagram, which are Booleans, integers, and strings. Guard expressions can be combined using $\wedge$, $\vee$, and $\neg$. We only allow send action expressions on the label; other action expressions would express changes of the case attributes performed by the WFMS and as we just explained, a WFMS does not change the case attributes —actors do.

In order to give a formal semantics to an activity diagram, we eliminate bars and diamonds by mapping the activity diagram into a hypergraph called an *activity hypergraph*. The details of this are straightforward and provided elsewhere [12]. Fig. 2 shows the activity hypergraph of Fig. 1. We can view a UML activity diagram as syntactic sugar for an activity hypergraph.

An *activity hypergraph* is a tuple $(Nodes, Events, Guards, HyperEdges, LVar)$ where:

- $Nodes$ is the set of nodes. Set $Nodes$ is partitioned into set AN of activity nodes, set WN of wait nodes, set FN of final nodes, and one initial node, $initial$. Each activity node $a \in AN$ may observe (read) and update some variables, denoted by sets $Obs(a) \subseteq LVar$ and $Upd(a) \subseteq LVar$.
- $Events$ is the set of event expressions. There are external and internal events. Internal events are named. External events are named events (e.g., `receive payment` in Fig. 2), condition change events, temporal events, and termination events [42]. Function $term : AN \rightarrow Events$ defines, for each

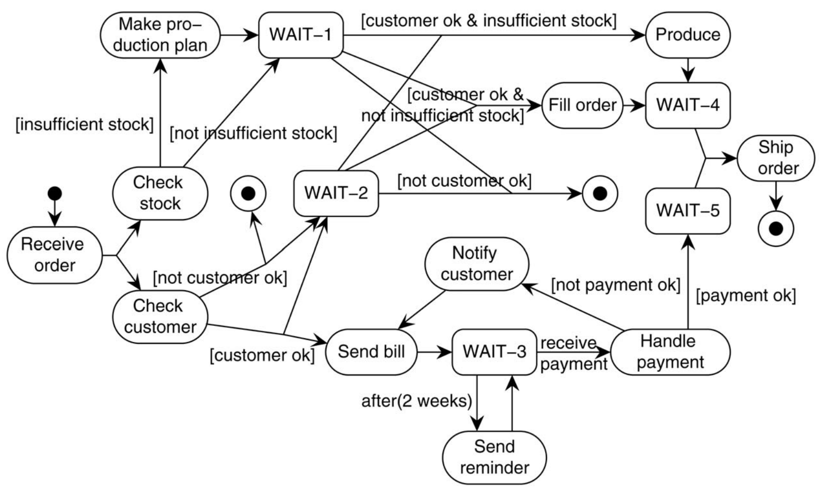


Fig. 2. Activity hypergraph of Fig. 1.

activity node, its termination event. Each temporal `after` event must belong to a unique hyperedge, to ensure that each timeout event does not accidentally trigger other hyperedges with the same label.

- $Guards$ is the set of guard expressions,
- $HyperEdges$ is the transition relation between the nodes of the activity diagram. Hyperedges have several parameters. For every hyperedge h, $source(h)$ denotes the nonempty set of source nodes left and $target(h)$ the nonempty set of target nodes entered if h is taken. The trigger event of h is $event(h) \in Events \cup \{\perp\}$, where $\perp$ denotes the absence of a trigger event. The guard of h is denoted $guard(h) \in Guards$. The set of send events generated by h is denoted $sendactions(h)$. Each send event must be a named event.
- $LVar$ is the set of variables local to the workflow. Every variable in a guard expression is a local variable.

3 Semantics of UML Activity Diagrams

At the present time, there is no generally accepted formal semantics of UML activity diagrams and the informal OMG semantics of UML 1.5 is not entirely suitable for workflow modeling [14], [12]. (UML 2.0, endorsed after this paper was written, does not have a formal workflow semantics either.) We therefore defined a formal semantics for activity diagrams, intended for workflow modeling, that can be used for model checking [13]. Since a workflow model prescribes the behavior of a WFMS, the semantics is defined in terms of WFMSs.

We view a WFMS as a reactive system [21], [42] which reacts to events it receives from its environment (e.g., activity completion events), based upon its current state. It can react by changing its state and by causing certain desired effects in its environment, such as enabling new activities or sending messages to its environment. For example, if, in Fig. 2, node `Check stock` is active and the WFMS receives the completion event of the activity *Check stock*, then the WFMS reacts by leaving `Check stock`, resetting the set of input events, and either entering node `Make production plan` and enabling activity *Make production plan* if `insufficient stock` is true or entering node `WAIT-1` otherwise.

3.1 Clocked Transition Systems

Our formal semantics of activity diagrams maps the activity hypergraph of the diagram to a clocked transition system (CTS) [30], which is an extension of transition systems with real variables to model real-time. Given a set Var of variables, a state of a CTS is a function σ that assigns to each $v \in Var$ a value $\sigma(v)$. The set of all states (i.e., functions) is denoted $\Sigma(Var)$.

Formally, a *Clocked Transition System* (CTS) is a tuple $(Var, \rightarrow, \sigma_{init})$ where:

- $Var = Disc \cup \{MC\} \cup Timers$ is a finite set of variables. Set $Disc$ contains discrete variables. Variable MC represents the master clock that measures the global time. The master clock MC can never be reset. The other clocks are represented by set $Timers$.
- $\rightarrow \subseteq \Sigma(Var) \times \Sigma(Var)$ is the transition relation.
- $\sigma_{init} \in \Sigma(Var)$ is the initial valuation.

Instead of writing $(\sigma, \sigma') \in \rightarrow$, we write $\sigma \rightarrow \sigma'$.

The transition relation $\rightarrow$ is partitioned in two sets, *data transitions*, in which the clocks do not increase but can be reset and in which discrete variables can change arbitrarily, and *time transitions*, in which clocks increase but discrete variables do not change. A *path* of a CTS is an infinite sequence π of valuations, $\pi = \sigma_0\sigma_1\ldots$ satisfying:

- Initiation: $\sigma_0 = \sigma_{init}$.
- Consecution: For every $i = 0, 1, \ldots,$ the valuation σ_{i+1} is a $\rightarrow$ successor of σ_i, i.e., $\sigma_i \rightarrow \sigma_{i+1}$.

A *run* is a path satisfying

- Time divergence: The sequence $\sigma_0(MC)\sigma_1(MC)\ldots$ grows beyond any bound, i.e., the value of MC increases beyond any bound. Thus, a run cannot have Zeno behavior.

3.2 Informal Semantics

We have defined two reactive semantics of activity diagrams, called the requirements-level semantics and the implementation-level semantics [13], [14], [12]. In the *requirements-level semantics*, the perfect synchrony hypothesis [3] is adopted: The system reacts immediately and infinitely fast to the reception of events. Since the system is infinitely fast, no other event can occur while a system is reacting to an event. Also, in this semantics, a transition does not take time. In the *implementation-level semantics*, on the other hand, the perfect synchrony hypothesis does not hold. In this semantics, a queue is needed to store events that occur while the system is busy reacting to an event. Also, taking a hyperedge in this semantics does take time. The requirements-level semantics is based on the STATEMATE semantics of statecharts [20], [22], whereas the implementation-level semantics is based on the OMG semantics of UML statecharts [39].

We use the requirements-level semantics of an activity diagram if we want to specify in an implementation-independent way what any WFMS must do. We use the implementation-level semantics of an activity diagram if we want to describe how a general WFMS implementation behaves. The requirements-level semantics is easy to model check and simple to understand, and the implementation-level semantics is difficult to model check and difficult to understand. We have identified a set of constraints that characterize a large set of requirements that remain invariant when we move from a requirements-level to an implementation-level semantics [12]. If we check these requirements in the requirements-level semantics of an activity diagram, they are guaranteed to hold in the implementation-level semantics of that diagram, too.

In our tool, we have implemented the requirements-level semantics. In the requirements-level semantics, we have adopted the STATEMATE semantics [20], [22] of a system reaction. Informally, the semantics is as follows: When events occur, the system state becomes unstable, and the system reacts by taking a set of hyperedges, called a step. (Steps are defined below.) If the state reached by a step is unstable as well, then again a step is taken. A reached state is either unstable because some events have been generated in the previous step or because another step can be taken. If the reached state is stable, the system stops taking steps. Thus, the WFMS reaction to an event is a sequence of steps, a *superstep* [20].

Below, we present a formal semantics for safe activity hypergraphs, i.e., activity hypergraphs in which nodes are active at most once at the same time. Space restrictions prevent us from presenting a formal semantics for unsafe activity hypergraphs, which is considerably more complex and intricate [13], [12]. Our formalization has been inspired by an existing formalization of STATEMATE statecharts [10].

3.3 Step Semantics

To define a step, we need some auxiliary notions first. A hyperedge is *enabled* iff its sources are active, it is triggered by an event in the input E, and its guard evaluates to true. A guard is evaluated in a valuation σ by substituting, for every variable v, its value $\sigma(v)$. If g is true in valuation σ, we write $\sigma \models g$. Let C denote the set of active nodes, the configuration, and let E the set of input events. Set $enabled_\sigma(C, E)$ of hyperedges enabled in C on input E is defined as follows:

$$\begin{aligned} enabled_\sigma(C,E) \overset{\text{df}}{=} \{h \in HyperEdges \mid source(h) \subseteq C \\ \wedge (event(h) \in E \vee event(h) = \bot) \\ \wedge \sigma \models guard(h)\}, \end{aligned}$$

where, as before, $\bot$ denotes the absence of a trigger event (see Section 2).

A set of hyperedges H is defined to be consistent, written $consistent(H)$, iff all hyperedges can be taken at the same time, i.e., they do not have overlapping sources.

$$\begin{aligned} &consistent(H) \overset{\text{df}}{\Leftrightarrow} \\ &\forall h, h' \in H \bullet h \neq h' \Rightarrow source(h) \cap source(h') = \emptyset. \end{aligned}$$

Two activity nodes, A, B, interfere if one of them observes or reads a variable the other one updates. Configuration C is *interfering* iff two activity nodes in C interfere.

$$\begin{aligned} interfering(C) \overset{\text{df}}{\Leftrightarrow} \forall a, b \in C \bullet a \neq b \Rightarrow \\ ((Obs(A) \cup Upd(A)) \cap Upd(B) \neq \emptyset) \\ \vee ((Obs(B) \cup Upd(B)) \cap Upd(A) \neq \emptyset). \end{aligned}$$

A set of hyperedges H is defined to be maximal iff, for every enabled hyperedge h, set $H \cup \{h\}$ is inconsistent or the configuration reached next is interfering. The function $nextconfig(C, H)$ returns the configuration reached from C by taking H.

$$\begin{aligned} maximal_\sigma(C,E,H) \overset{\text{df}}{\Leftrightarrow} \forall h \in enabled_\sigma(C,E) \bullet \\ h \notin H \Rightarrow (\neg consistent(H \cup \{h\}) \\ \vee interfering(nextconfig(C, H \cup \{h\}))) \end{aligned}$$

$$nextconfig(C,H) \overset{\text{df}}{=} \left(C \setminus \bigcup_{h \in H} source(h)\right) \cup \bigcup_{h \in H} target(h).$$

Finally, predicate $isStep$ defines a set of hyperedges S to be a step iff every hyperedge in S is enabled, S is maximal and consistent, and the next configuration is noninterfering.

$$\begin{aligned} isStep_\sigma(C,E,S) \overset{\text{df}}{\Leftrightarrow} S \subseteq enabled_\sigma(C,E) \wedge consistent(C) \\ \wedge \neg interfering(nextconfig(C,S)) \wedge maximal_\sigma(C,E,S). \end{aligned}$$

Steps can be computed, given some set H of enabled hyperedges, by splitting H into maximal, consistent sets of hyperedges that do not lead to interfering next configurations.

3.4 Formal Semantics

In the requirements-level semantics, a step is taken immediately when an event occurs and, during the step, clocks dot not advance. To define this, we use the variables C, I, and $LVar$ as discrete variables for the Clocked Transition System, so $Disc \overset{\text{df}}{=} \{C, I\} \cup LVar$, where C is the configuration, I is the current set of input events, and $LVar$ is the set of local variables of the activity diagram.

The transition relation $\rightarrow$ for the CTS consists of seven subtransition relations. In a requirements-level run, these transitions must occur in the order specified in Fig. 3. In the figure, a node represents a valuation. The initial valuation of an activity diagram is unstable by definition: A step is taken in order to leave this initial state and enter a stable state. Note that the cycle of transitions $\rightarrow_{step}$ and $\rightarrow_{unstable}$ in Fig. 3 models the superstep.

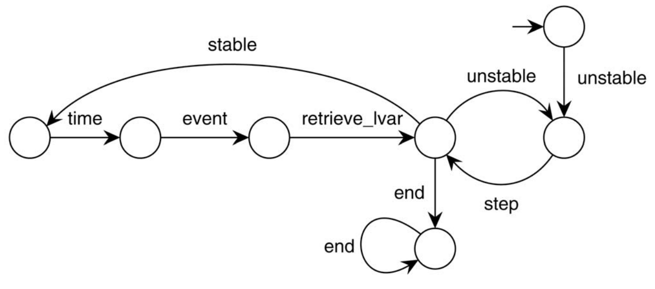


Fig. 3. Execution cycle in requirements-level semantics.

Relation $\rightarrow_{time}$ defines the passage of time. The master clock and the running timers are increased by some real number Δ. The master clock and all timers cannot be advanced beyond their next deadline unless they already are beyond their deadline. If the master clock or some timer reaches its next deadline, a corresponding timeout event is generated. A timer is running iff the sources of its corresponding hyperedge are active. Given a configuration C, set $RT(C)$ of running timers is defined as

$$RT(C) \overset{\text{df}}{=} \{t \in Timers \mid source(hedge(t)) \subseteq C\},$$

where $hedge(t)$ denotes the unique hyperedge belonging to timer t. The deadline of timer t is denoted $deadline(t)$. Let $WhenEvents$ denote the set of all when events. Each when event w occurs modulo a certain period. Thus, there is a set, say $deadlines(w)$, of points in time at which w occurs. The set of all when events specifies a set $deadlines(WhenEvents)$.

$$\begin{aligned}
&\sigma \rightarrow_{time} \sigma' \overset{\text{df}}{\Leftrightarrow} \exists \Delta \in \mathbb{R} \bullet \\
&\Delta > 0 \wedge \sigma' = \sigma[\&_{t \in RT(\sigma(C))} t/\sigma(t)+\Delta, MC/\sigma(MC)+\Delta] \\
&\wedge \forall l \in deadlines(WhenEvents) \bullet \sigma(MC) < l \Rightarrow \sigma(MC)+\Delta \leq l \\
&\wedge \forall t \in RT(\sigma(C)) \bullet \sigma(t) < deadline(t) \Rightarrow \sigma(t)+\Delta \leq deadline(t).
\end{aligned}$$

Valuation $\sigma[x/val]$ assigns to variable x value val and to every other variable y, $y \neq x$, the value $\sigma(y)$. Symbol $\&$ denotes a bulk update: $\sigma[\&_{x \in X} x/val_x] \overset{\text{df}}{=} \sigma[x_1/val_1, \ldots, x_n/val_n]$, where $n = \#X$ [38].

Relation $\rightarrow_{event}$ defines that events occur. The only component that changes is I, the set of input events. The nonoccurrence of events is excluded: No change is not a change. The occurrence of events must satisfy some additional constraints. Line by line, the definition says that a set E of event occurrences is allowed if and only if

1. the set is not empty,
2. activity nodes that terminate are in the current configuration,
3. when the master clock reaches a deadline, the corresponding when events are generated, and
4. an after event is generated when the corresponding timer has reached its deadline.

$$\begin{aligned}
&\sigma \rightarrow_{event} \sigma' \overset{\text{df}}{\Leftrightarrow} \exists E \subseteq Events \bullet \\
&\sigma' = \sigma[I/E] \wedge E \neq \emptyset \\
&\wedge \forall a \in AN \bullet term(a) \in \sigma'(I) \Rightarrow a \in \sigma(C) \\
&\wedge \forall we \in WhenEvents \bullet \sigma(MC) \in deadlines(we) \Leftrightarrow we \in E \\
&\wedge \forall t \in RT(\sigma(C)) \bullet \sigma(t) = deadline(t) \Rightarrow event(t) \in E.
\end{aligned}$$

Before the step can be computed, the valuation of the local variables in the database must be known. The valuation of these variables may have changed because some activities have terminated (recorded in I) or because the environment has updated some variables. Relation $\rightarrow_{retrieve\_lvar}$ specifies that the new values of the local variables are retrieved. The valuation of variables that are observed or updated in some running activity does not change.

$$\begin{aligned}
\sigma \rightarrow_{retrieve\_lvar} \sigma' \overset{\text{df}}{\Leftrightarrow}\ & \sigma(C) = \sigma'(C) \wedge \sigma(I) = \sigma'(I) \\
& \wedge \sigma(MC) = \sigma'(MC) \\
& \wedge \forall t \in RT(\sigma(C)) \bullet \sigma(t) = \sigma'(t) \\
& \wedge \forall a \in AN \bullet a \in \sigma(C) \wedge term(a) \notin \sigma(I) \Rightarrow \\
& \forall v \in LVar \bullet v \in Obs(a) \cup Upd(a) \Rightarrow \sigma(v) = \sigma'(v).
\end{aligned}$$

Transitions $\rightarrow_{unstable}$ and $\rightarrow_{stable}$ test whether a valuation is unstable or stable. Transition $\rightarrow_{end}$ tests whether an activity diagram has ended. Valuation σ is stable iff there are no enabled hyperedges and the set of input events is empty: $stable \overset{\text{df}}{=} enabled(C, I) = \emptyset \wedge I = \emptyset$. Both transitions have lower priority than transition $\rightarrow_{end}$.

$$\begin{aligned}
&\sigma \rightarrow_{unstable} \sigma' \overset{\text{df}}{\Leftrightarrow} \sigma = \sigma' \wedge \sigma \not\models stable \wedge \sigma \not\rightarrow_{end} \sigma' \\
&\sigma \rightarrow_{stable} \sigma' \overset{\text{df}}{\Leftrightarrow} \sigma = \sigma' \wedge \sigma \models stable \wedge \sigma \not\rightarrow_{end} \sigma' \\
&\sigma \rightarrow_{end} \sigma' \overset{\text{df}}{\Leftrightarrow} \sigma = \sigma' \wedge \sigma(I) = \emptyset \wedge \forall n \in Nodes \bullet \\
&\qquad n \in \sigma(C) \Rightarrow n \in FN.
\end{aligned}$$

We next define the step transition relation $\rightarrow_{step}$. Line by line, the $\rightarrow_{step}$ definition below says that a step is done between σ and σ' iff

1. there is a step S (using the predicate $isStep$ defined above),
2. the variables that are contained in the guards of the hyperedges in S are not being updated in some nonterminated activity (otherwise, an inconsistent value could be read),
3. there is a set T of timers that can be turned on,
4. σ is then updated into σ' by computing the next configuration if step S is taken (using the function $nextconfig$), putting the generated events in I, and initializing the new timers.

$$\begin{aligned}
\sigma \rightarrow_{step} \sigma' \overset{\text{df}}{\Leftrightarrow}\ & \exists S \subseteq HyperEdges \bullet isStep_{\sigma}(\sigma(C), \sigma(I), S) \\
& \wedge \forall a \in AN \bullet a \in \sigma(C) \wedge term(a) \notin \sigma(I) \\
& \qquad Upd(a) \cap (\bigcup_{h \in S} var(guard(h))) = \emptyset \\
& \wedge \sigma' = \sigma[C/nextconfig(\sigma(C), S), \\
& \qquad I/ \cup_{h \in H} sendactions(h), \\
& \qquad \&_{t \in RT(\sigma(C)) \setminus RT(\sigma(C))} t/0],
\end{aligned}$$

where $var(g)$ denotes the variables guard g tests.

In the initial valuation σ_0, the configuration only contains one copy of $initial$, $C = \{initial\}$, and the input is empty, $I = \emptyset$. The other variables, including master clock MC, must be initialized with an appropriate value.

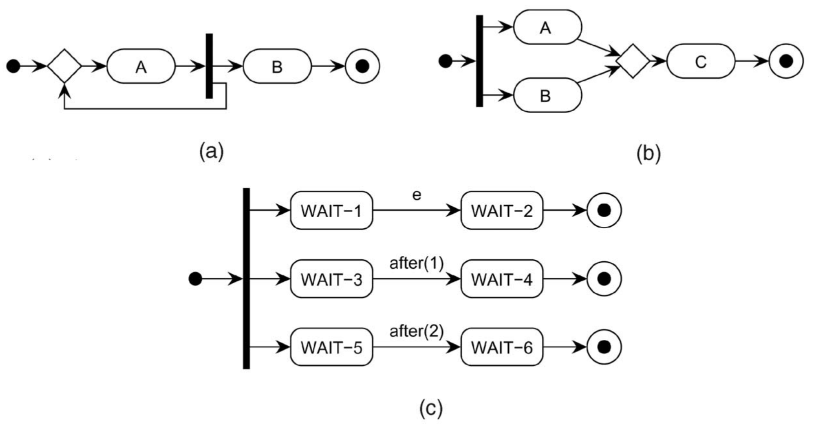


Fig. 4. Examples. (a) Activity diagram with infinite state space. (b) Activity diagram with finite state space. (c) Illustration of discretization of clocks.

4 From Infinite to Finite State Space

To be able to model check an activity diagram, its transition system (TS) must be finite. In this section, we show how we transform an infinite state CTS into a finite state TS using solutions taken from literature.

4.1 Unbounded Nodes

In Fig. 4, arbitrarily many instances of `B` can be active at the same time. This diagram therefore has an infinite state space. In Fig. 4b, `C` is executed twice; two instances of `C` can be active at the same time. This diagram has a finite state space: There are no unbounded nodes. A node is *unbounded* if there is no bound on the maximum number of its active instances [35].

Model checking is decidable for bounded models [9], but, for unbounded models, it can easily become undecidable [17]. We therefore restrict ourselves to bounded models. In our implementation, the computation of the transition system is stopped if one of the nodes becomes unbounded. A node n is unbounded iff there is a state s that has n in its configuration C_s and s has a predecessor state s' such that its configuration $C_{s'}$ is strictly contained in C_s [28]. For example, in Fig. 4, node `B` is unbounded since a state with configuration [`A`,`B`] is reachable from a state with configuration [`A`].

4.2 Abstracting from Data

Since an activity hypergraph can have integer and string variables, the state space of the transition system can be infinite. To reduce this to a finite one, observe that the only data that influences the execution of the activity hypergraph are the event and guard labels. The only relevant data, therefore, is the Boolean valuation of the event and guard expressions. A naive model checking strategy would therefore be to drop all data and to replace every guard expression by a Boolean representative. However, guard expressions can be dependent upon each other. For example, if $[s = "red"]$ is true, then $[s \neq "red"]$ must be false, but, in the naive model checking strategy, $[s = "red"]$ and $[s \neq "red"]$ might both be assigned the value true. Such valuations should not occur in the model.

We therefore consider *basic guard expressions*: those parts of the guard expressions not containing $\wedge$, $\vee$, and $\neg$. This partly solves the problem sketched above (for example, $[p \wedge q]$ and $[q]$ are dependent now), but not fully since basic guard expressions, too, can be dependent upon each other. For example, basic guard expressions $s = "red"$ and $s = "blue"$ are not independent since s cannot be both red and blue. We have solved this problem in our implementation by enforcing that if two basic guard expressions refer to the same variable, then at most one of them can be true at the same time. Furthermore, to avoid that, say, $x < 10$ and $x > 12$ are true at the same time, the only Boolean expressions referring to integers that we allow are equality tests, for example, $[x = 10]$.

The approach above is inspired by existing approaches from modal logic theory, e.g., filtration [18]. Most approaches in model checking to data abstraction use an overapproximation based upon the theory of Abstract Interpretation (see, e.g., [9]).

4.3 Real Time

In our tool, we have only implemented `after` events, `when` events can be dealt with similarly. The problem in interpreting `after` events, is that our semantics uses a dense time model: Between two points in time, there always exists another point in time. In a dense time model, clocks can have infinitely many values in a finite interval of time. Clearly, we cannot compute all these different values.

One obvious solution is to use discrete clocks. The problem then is to find the right discretization such that at least the qualitative behavior of the dense-time model is preserved. For example, discretizing the example in Fig. 4 with clock ticks of 2 makes configuration [`WAIT-1`,`WAIT-4`,`WAIT-5`] unreachable, whereas this configuration is reachable in the dense time model. The dense time model we use can be safely discretized with clock ticks of 1 time unit [2], [24]. The discretization preserves the untimed reachability properties of the original dense time model, but it may introduce some different timing behavior [2], [24]. So, it is not possible to use a real-time logic as requirement language. Since there is no real-time model checker supporting strong fairness constraints, this is not much of a restriction anyway. This discretization transforms a CTS into a TS.

5 Strong Fairness

Workflow specifications can contain loops. For example in Fig. 1, there is a loop `Send bill,WAIT-3,Handle payment,Notify customer,Send Bill,...`. So far, our semantics contains a run in which this loop is never exited. Along this run, the payment is never ok. This is not what is intended. The activity diagram is intended to describe the situation in which, eventually, the payment is ok. If we were to account for the possibility of nonpaying customers, we should describe another workflow that includes the work of the bailiff. In general, loops in a workflow are eventually exited because every workflow eventually terminates.

Note that even a workflow specification that has no loops in the activity diagram may have loops in its underlying transition system. This is due to event occurrences that can occur in a certain state but that are irrelevant and therefore ignored. For example, in Fig. 1, event `receive payment` can occur while node `Receive order` is active. Nothing in our semantics so far prevents `receive payment` from happening over and over again while `Receive order` is active. Thus, there is a run in which node `Receive order` is never left. But we would like to exclude such a run because, in it, activity *Receive order* never terminates while `receive payment` occurs infinitely often. (This behavior resembles Zeno behavior in timed systems.)

A useful way to avoid these loops is to use strong fairness constraints. A strong fairness constraint (p, q), where p and q are properties, states that, if p is true infinitely often in a run, then q must be true infinitely often in the run as well [34]. Intuitively, a requirement p can only be true infinitely often if there is a loop in the transition system in which p is made true. Associating p with a loop and q with its exit, (p, q) effectively blocks a path that would loop infinitely often without exiting infinitely often. Using a strong fairness constraint, therefore, we can specify that some loop must be exited eventually.

We specify a strong fairness constraint for every hyperedge h that is triggered by an external event. The constraint states that if h is relevant infinitely often, it must be taken infinitely often. The strong fairness condition for the complete activity hypergraph is the conjunction of all individual strong fairness constraints:

$$sf \overset{\text{df}}{=} \bigwedge_{h \in HyperEdges \mid \neg internal(h)} [(stable \wedge source(h) \sqsubseteq C, stable \wedge target(h) \sqsubseteq C).]$$

Predicate $internal$ is true if and only if the hyperedge is triggered by an internal event. Predicate $stable$ is true if and only if the current state is stable (cf. Section 2). See [12] for formal definitions of these predicates. This strong fairness constraint states that the environment must behave in a fair way: If a hyperedge is infinitely often relevant in a stable state, the environment must generate the trigger event of this hyperedge some time and must make the guard true some time. We assume that the guard is satisfiable. For the example activity hypergraph in Fig. 2, strong fairness constraint $([\texttt{Receive order}] \sqsubseteq C, [\texttt{Check stock}, \texttt{Check customer}] \sqsubseteq C)$ states that activity node `Receive order` must terminate some time. A run in which node `Receive order` is never left is not strongly fair because node `Receive order` is infinitely often contained in the configuration, but nodes `Check stock` and `Check customer` are not.

Each strong fairness constraint (p, q) is equivalent to LTL constraint $\texttt{G}\,\texttt{F}\,p \Rightarrow \texttt{G}\,\texttt{F}\,q$, where $\texttt{G}\,\varphi$ means that φ is globally true in every state of the run and $\texttt{F}\,\varphi$ means that φ is true in some future state of the run. Encoding of strong fairness constraints as antecedent of the LTL requirement that has to be verified leads to a state explosion, so we decided to use an existing special algorithm [31] for model checking LTL formulas with strong fairness constraints. The algorithm restricts the evaluation of an LTL formula to strongly fair runs only. It has been implemented in a tool called Temporal Logic Verifier (TLV) [37], but, since TLV does not support batch processing, it could not be integrated with TCM. We therefore implemented the algorithm in the open source model checker NuSMV, which does support batch processing. The resulting strong fairness model checker is called NuSMV_{fair} and is now part of NuSMV 2.1 (http://nusmv.irst.itc.it).

6 Implementation

6.1 Assumptions about Variables

For our tool implementation, we have made four assumptions that we motivate below.

1. If an activity reads a variable, we assume that it updates that variable, too.
2. We assume that a variable is updated by an activity A if a guard of a hyperedge leaving A tests the variable. So, in Fig. 1, we assume that `Check stock` updates Boolean variable `insufficient stock`.
3. The effect of an activity is a possible change in valuation of the variables that the activity updates. Since the only relevant changes are changes in the truth value of a guard, the effect of an activity is expressed in terms of the basic guards that are made true or false by that activity.
4. The data that is updated in an activity (by an actor) is not updated by the environment.

Using assumptions 1 and 2, TCM can automatically derive which activity updates which variable. Thus, the user does not have to specify which variables are read or updated by an activity. Assumption 3 states that an activity only changes the truth value of basic guard expressions that contain a variable that is updated by that activity. We do not allow basic guard expressions that contain more than one variable (see Section 4). The final assumption ensures, for example, that, in Fig. 1, the two choices based upon `insufficient stock` have the same outcome. A beneficial side-effect of this assumption is that it reduces the state explosion.

6.2 Two Implementations

We have made two different implementations. The first implementation is an extension of TCM, written in C++, with an execution algorithm that maps an activity hypergraph into a TS with strong fairness constraints. The TS is

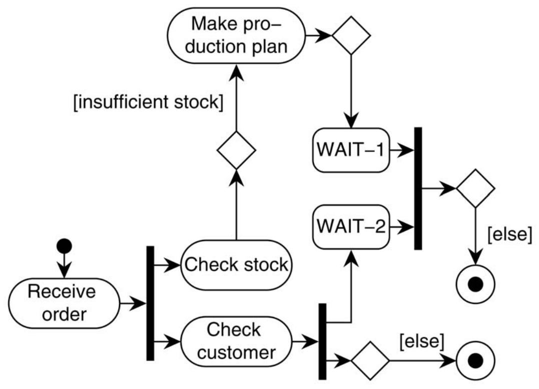


Fig. 5. Path illustrating counterexample for R1.

straightforwardly encoded as input for a model checker by enumerating every state. The second implementation is based on existing approaches (e.g., [5]) to verify statecharts with symbolic model checkers like NuSMV. In this implementation, the syntax of an activity hypergraph is encoded directly as input for a symbolic model checker in such a way that the semantics that the symbolic model checker attaches to the input coincides with the requirements-level semantics. The second implementation can only deal with safe activity diagrams. But, to check safeness, we still need to use the first implementation. Moreover, if some hyperedges share sources and targets, the second implementation does not work any more because then some constraints in the input will conflict with each other. However, if the second implementation can be applied, it is more efficient than the first implementation.

7 Example Verifications

Requirement R1 states that, for each possible strongly fair run, either both Make production plan and Produce occur sometime in the future or both of them do not occur:

$$\text{(R1)}\quad \mathtt{F\ in\ (Make\ production\ plan)} \Leftrightarrow \mathtt{F\ in\ (Produce)},$$

where $\mathtt{in}(x)$ is true in a valuation σ iff node x is contained in the configuration of σ. (TCM translates in into an equivalent predicate on nodes.) This requirement fails to hold: The path that TCM highlights is shown in Fig. 5. We see that, if the customer check fails, the workflow stops, while Make production plan may already have been executed. We decide to improve the requirement by making it conditional on the customer check:

$$\text{(R1}'\text{)}\quad (\mathtt{F\ customer\ ok}) \Rightarrow (\mathtt{F\ in\ (Make\ production\ plan)} \Leftrightarrow \mathtt{F\ in\ (Produce)}).$$

NuSMV$_{fair}$ reports that this requirement is true. It is true because of our Assumption 4 in Section 6 which implies, for this workflow specification, that only Check stock can change variable insufficient stock.

Finally, we verify that, in each strongly fair run, a bill is sent if and only if either something is produced or taken from stock:

$$\text{(R2)}\quad \mathtt{F(in(Produce)} \vee \mathtt{in(Fill\ order))} \Leftrightarrow \mathtt{Fin(Send\ bill)}.$$

NuSMV$_{fair}$ reports that this requirement is true.

A general requirement for workflow models is that the model does not deadlock, i.e., for every strongly fair run, from the initial state a final state should be reachable:

$$\text{(R3)}\quad \mathtt{F\ G}\ final,$$

where $final$ is true in a state iff the configuration only contains final nodes (bull's eyes). TCM translates $final$ into an equivalent predicate on nodes. NuSMV$_{fair}$ reports that the requirement is true. Without using strong fairness, the requirement would have been false because of the loop Send bill, WAIT-3, Handle payment, Notify customer, Send Bill,....

Another general requirement is that activity diagrams do not diverge; that is, there is always a future in which an activity diagram is stable:

$$\text{(R4)}\quad \mathtt{G\ F}\ stable.$$

NuSMV$_{fair}$ reports that this requirement is true.

8 Fighting State Explosion

The main problem with model checking is the state space explosion [9]. We defined and implemented four rules to reduce the state space of an activity hypergraph given a requirement φ where $\varphi \in CTL^*{-}X$, i.e, φ does not contain the next operator. It is easy to show that, for every reduction rule r, φ holds for the transition system of the original activity hypergraph AH iff it holds for the transition system of $r(AH)$ [12]. In particular, for rules 1 and 2, the original activity diagram and the reduced one have the same reachable configurations.

Rule 1: No irrelevant external event occurrences. Only allow an external event e to occur if it triggers a relevant hyperedge. The rule rules out hidden loops in an activity diagram (cf. Section 5).

This reduction rule is allowed iff φ does not refer to external events.

Rule 2: Interleaved named external event occurrences. Only allow interleaved named external event occurrences; that is, no two named external events can occur at the same time. Note that this rule does not apply to temporal events and condition change events.

This reduction rule is allowed iff φ does not refer to named external events. Moreover, the activity hypergraph must satisfy some constraints; otherwise, the reduced activity diagram might behave differently from the original activity diagram; see [12] for details.

Rule 3: Remove local variables. Remove local variable v from the activity hypergraph and remove every basic guard condition that refers to v.

This reduction rule is allowed if:

- v is not updated by two concurrent activities,
- The requirement φ does not refer to v.
- The only hyperedges referring to v are the ones leaving the activity node A in which v is updated.

- In a decision, the disjunction of basic guard expressions referring to v is true. This can be easily ensured by including an `else` branch in every decision.

The first constraint is needed because, otherwise, two interfering activity nodes can become active at the same time in the reduced activity hypergraph, which is impossible in the original activity hypergraph due to our step semantics. The second constraint ensures that the requirement φ can still be evaluated on the reduced activity hypergraph. The third constraint ensures that the reduced activity hypergraph does not have more configurations than the original one. For example, in Fig. 1, variables `insufficient stock` and `customer ok` cannot be removed because they are contained in the guards of the hyperedges leaving nodes `WAIT-1` and `WAIT-2`. The fourth constraint ensures that, if the original activity hypergraph contains a deadlock, the reduced one contains a deadlock as well.

Applying this reduction rule to the activity diagram in Fig. 1, if the requirement to be verified is `F G` $final$, then variable `payment ok` can be removed and the corresponding guard conditions can be removed as well.

Rule 4: Remove nodes. If there is an activity or wait node n with only one outgoing external hyperedge h such that $source(h) = \{n\}$ so h does not conflict with any other hyperedge, then both n and h can be removed from the activity diagram by replacing every occurrence of n in the target of some hyperedge with the targets of h. If h was the last hyperedge referring to some trigger event and/or local variable, these can be removed from the set of events and local variables, respectively.

This reduction rule is allowed if:

- The requirement φ neither refers to n, nor to the label of h, nor to the target nodes of h.
- Neither n nor the target nodes of h are referred to by some `in` predicate in the activity diagram.
- The trigger event of h is a named external event or a termination event.
- The trigger event e of h can only occur in this state, either because 1) n is an activity node and e denotes termination of n or because 2) n is a wait node and e is a named external event that does not trigger any other hyperedge and reduction rule 1 (no irrelevant events) is used.

The first constraint ensures that the requirement can still be evaluated on the reduced activity hypergraph. The second constraint prevents an `in` predicate having an undefined value. The third and fourth constraint ensure that the trigger event of h only triggers h and, moreover, can only occur in n. Thus, removal of both n and h will not affect other parts of the activity hypergraph.

8.1 Practical Experience

To estimate the effect of the reduction rules on real-life workflow models, we have used the reduction approach on two existing workflow models that are used in organizations. Table 1 shows the performance results for model checking our running example and these two additional models with NuSMV$_{fair}$, where all four reduction rules have been applied. Since the last two original models were too big for TCM to construct, we used our second, symbolic implementation to estimate the number of reachable states in the original model. The analysis was performed on a PC with a Pentium III 450 MHz processor with 128Mb of RAM under Red Hat Linux 6.0.

9 Related Work

9.1 Semantics

Though the syntax of activity diagrams resembles the syntax Petri nets, our formal semantics differs from a token-game semantics: Our semantics is reactive, whereas the token-game semantics is not. Our semantics can be easily adapted to give a reactive semantics to Petri nets. For an extensive comparison of our semantics with different Petri net variants, see [16]. Other formalizations of activity diagrams have been presented, but these are neither intended for workflow modeling nor used for model checking; see [12].

9.2 Verification Tools

Woflan [41] is a tool for verification of textual workflow specifications without data and real time. Feedback is also textual. The workflow specifications are based on low-level Petri nets. Woflan allows verification of a fixed set of requirements that cannot be specified by the user. Woflan does not address strong fairness.

The tool developed in the Mentor project [36] uses a CTL model checker for statecharts. The tool is not integrated with the model checker. Strong fairness is not used. No details are given on how the feedback is presented to the user.

The Testbed Studio tool [26] supports model checking of business process models with Spin [25]. The process modeling language neither has external events nor temporal events. Models can have loops, but the analysis results on such models may be counterintuitive since it cannot be specified that loops are exited. The tool does not support strong fairness.

Karamanolis et al. [27] use the existing LTSA toolkit for model checking workflow schemas. Workflow schemas are translated manually into input for LTSA; the output of verification is shown graphically in the LTSA input, not in the original workflow schema. LTSA is based on process algebra; data and real-time cannot be explicitly modeled. LTSA can handle strong fairness constraints, but the authors do not focus on loops in workflow schemas.

Our work is also closely related to the work done on model checking STATEMATE and UML statecharts. Chan et al. [5] have defined model checking for statecharts using SMV [9]. Latella et al. [32] present a translation for a subset of UML statecharts to Spin [25]. None of the implementations discussed in these papers provide a graphical representation of the feedback of the model checker. Both papers encode the syntax of the statechart explicitly in the input language and let the model checking tool derive the step semantics implicitly. Our tool encodes the semantic structure directly in the input language. To compare our approach with these approaches, we have also implemented a syntactic encoding in our tool; see Section 6 for the results.

vUML [33] is a tool for model checking a communicating set of objects whose behavior is modeled by UML

TABLE 1
Resources Used by NuSMV_{fair} During Analysis

Requirement	Result	Reachable states (nr)	Time TCM (sec)	Time NuSMV_{fair} (sec)	BDD nodes (nr)	Memory (kB)
Workflow of production company: 369 reachable states						
R1	False	107	0.16	1.76	40454	3715
R1'	True	107	0.16	1.75	41055	3767
R2	True	133	0.20	2.81	47729	4367
R3	True	75	0.08	1.03	26221	3067
R4	True	71	0.08	0.84	22252	2875
Workflow of public prosecution service: ± 500,000 reachable states						
R3	True	656	15.46	66.55	185656	16547
R4	True	121	0.71	2.66	36405	4279
–	True	502	0.78	36.56	80137	11619
–	True	502	0.77	36.91	109992	12179
–	False	136	0.83	3.82	65708	5271
–'	True	305	16.01	15.43	94417	8963
Workflow of order processing within IT department: ± 100,000 reachable states						
R3	True	458	3.83	34.59	230912	13863
R4	True	162	0.78	4.33	61288	5295
–	True	324	3.73	16.02	69577	8627
–	True	323	3.46	17.87	149256	10247

statecharts. vUML uses the Spin [25] model checker. No details are given on how the statechart is encoded. The feedback of the tool is graphically represented by a UML sequence diagram. vUML does not support strong fairness nor real time.

9.3 Reduction Rules

Our reduction rules are similar to slicing rules in program analysis [40]. Recently, slicing has been used in combination with model checking [6], [8], [23]. Chan et al. [6] slice an RSML statechart, given some requirement φ, by removing parallel nodes in the statechart that are not (in)directly referred to by φ. In particular, if a certain node is relevant, all its predecessors are relevant as well; these are not removed. We sometimes cut away predecessor nodes (rule 4) and do not remove parallel nodes. Other approaches [8], [23] apply slicing to low-level programs, rather than graphical models.

10 Conclusion and Future Work

This paper has three contributions. First, our tool is the first verification tool for UML activity diagrams. Existing verification tools for the related UML statecharts do not address strong fairness constraints, nor present feedback in terms of the original statechart. Second, to our knowledge, our tool is the first to act as a usable front end to model checkers, allowing the use of higher-level behavior notations to define the transition system and presenting the trace in a readable format in this behavior notation. This approach can be generalized to other notations, such as statecharts, and to other model checkers. Third, our tool offers flexible analysis support of workflow models that have event driven behavior, data, loops, and real time. As described in Section 9, previous approaches either focus on fixed requirements for simple workflow models without data or real time or do not support strong fairness constraints.

Future work includes the generalization of TCM to a front end for other model checkers and other behavior notations. We also plan to define and implement a requirement language that is defined in terms of high-level behavior notations, thus further increasing the usability of TCM as a model checker front end.

Acknowledgments

The research reported in this paper was done while R. Eshuis was with the University of Twente, supported by NWO/SION, grant 612-62-02 (DAEMON). An earlier version of this paper has appeared in the *Proceedings of the International Conference on Software Engineering* (ICSE '02) [15].

References

[1] W.M.P. van der Aalst, P.J.N. de Crom, R.R.H.M.J. Goverde, K.M. van Hee, W.J. Hofman, H.A. Reijers, and R.A. van der Toorn, "Exspect 6.4: An Executable Specification Tool for Hierarchical Colored Petri Nets," *Proc. 21st Int'l Conf. Applications and Theory of Petri Nets,* M. Nielsen and D. Simpson, eds., 2000.

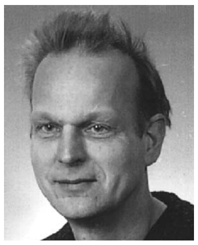
[2] E. Asarin, O. Maler, and A. Pnueli, "On Discretization of Delays in Timed Automata and Digital Circuits," *Proc. Int'l Conf. Concurrency Theory (CONCUR '98),* R. de Simone and D. Sangiorgi, eds., 1998.

[3] G. Berry and G. Gonthier, "The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming,* vol. 19, no. 2, pp. 87-152, 1992

[4] C. Bussler, "Enterprise-Wide Workflow Management," *IEEE Concurrency,* vol. 7, no. 3, pp. 32-43, 1999.

[5] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese, "Model Checking Large Software Specifications," *IEEE Trans. Software Eng.*, vol. 24, no. 7, pp. 498-520, July 1998.
[6] W. Chan, R.J. Anderson, P. Beame, D.H. Jones, D. Notkin, and W.E. Warner, "Optimizing Symbolic Model Checking for Statecharts," *IEEE Trans. Software Eng.*, vol. 27, no. 2, pp. 170-190, Feb. 2001.
[7] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A New Symbolic Model Checker," *Int'l J. Software Tools for Technology Transfer,* vol. 2, no. 4, pp. 410-425, 2000.
[8] E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum, "Program Slicing for VHDL," *Int'l J. Software Tools for Technology Transfer,* vol. 4, no. 1, pp. 125-137, 2002.
[9] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking.* MIT Press, 1999.
[10] W. Damm, B. Josko, H. Hungar, and A. Pnueli, "A Compositional Real-Time Semantics of STATEMATE Designs," *Proc. Int'l Symp. Compositionality,* W.-P. de Roever, H. Langmaack, and A. Pnueli, eds., 1998.
[11] F. Dehne, R. Wieringa, and H. van de Zandschulp, "Toolkit for Conceptual Modeling (TCM)—User's Guide and Reference," technical report, Univ. of Twente, 2000, http://www.cs.utwente.nl/tcm.
[12] R. Eshuis, "Semantics and Verification of UML Activity Diagrams for Workflow Modelling," PhD thesis, Centre for Telematics and Information Technology, Univ. of Twente, 2002, http://www.ctit.utwente.nl/library/phd/eshuis.pdf.
[13] R. Eshuis and R. Wieringa, "A Real-Time Execution Semantics for UML Activity Diagrams," *Proc. Conf. Fundamental Approaches to Software Eng. (FASE 2001),* H. Hussmann, ed., 2001.
[14] R. Eshuis and R. Wieringa, "An Execution Algorithm for UML Activity Graphs," *Proc. Fourth Int'l Conf. Unified Modeling Language (<<UML>>),* M. Gogolla and C. Kobryn, eds., 2001.
[15] R. Eshuis and R. Wieringa, "Verification Support for Workflow Design with UML Activity Graphs," *Proc. Int'l Conf. Software Eng. (ICSE 2002),* pp. 166-176, 2002.
[16] R. Eshuis and R. Wieringa, "Comparing Petri Net and Activity Diagram Variants for Workflow Modelling—A Quest for Reactive Petri Nets," *Petri Net Technology for Comm. Based Systems,* H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, eds., 2003.
[17] J. Esparza, "Decidability of Model Checking for Infinite-State Concurrent Systems," *Acta Informatica,* vol. 34, no. 2, pp. 85-107, 1997.
[18] R. Goldblatt, *Logics of Time and Computation,* second ed. Stanford Univ., 1992.
[19] P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig, "Crossflow: Cross-Organizational Workflow Management in Dynamic Virtual Enterprises," *Int'l J. Computer Systems Science & Eng.*, vol. 15, no. 5, pp. 277-290, 2000.
[20] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Trans. Software Eng. and Methodology,* vol. 5, no. 4, pp. 293-333, 1996.
[21] D. Harel and A. Pnueli, "On the Development of Reactive Systems," *Logics and Models of Concurrent Systems,* K.R. Apt, ed., 1985.
[22] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach.* McGraw-Hill, 1998.
[23] J. Hatcliff, M.B. Dwyer, and H. Zheng, "Slicing Software for Model Construction," *Higher-Order and Symbolic Computation,* vol. 13, no. 4, pp. 315-353, 2000.
[24] T.A. Henzinger, Z. Manna, and A. Pnueli, "What Good Are Digital Clocks?" *Proc. Int'l Colloquium Automata, Languages, and Programming (ICALP '92),* W. Kuich, ed., 1992.
[25] G.J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279-295, May 1997.
[26] W. Janssen, R. Mateescu, S. Mauw, and J. Springintveld, "Verifying Business Processes Using Spin," *Proc. Int'l Spin Workshop,* E. Najm, A. Serhrouchni, and G. Holzmann, eds., 1998.
[27] C. Karamanolis, D. Giannakopoulou, J. Magee, and S. Wheater, "Model Checking of Workflow Schemas," *Proc. Int'l Conf. Enterprise Distributed Object Computing (EDOC 2000),* pp. 170-181, 2000.
[28] R.M. Karp and R.E. Miller, "Parallel Program Schemata," *J. Computer and System Sciences,* vol. 3, pp. 147-195, 1969.
[29] W.D. Kelton, R.P. Sadowski, and D.A. Sadowski, *Simulation with Arena.* McGraw-Hill, 1998.
[30] Y. Kesten, Z. Manna, and A. Pnueli, "Verification of Clocked and Hybrid Systems," *Acta Informatica,* vol. 36, no. 11, pp. 837-912, 2000.
[31] Y. Kesten, A. Pnueli, and L. Raviv, "Algorithmic Verification of Linear Temporal Logic Specifications," *Proc. Int'l Colloquium Automata, Languages, and Programming (ICALP '98),* K.G. Larsen, S. Skyum, and G. Winskel, eds., 1998.
[32] D. Latella, I. Majzik, and M. Massink, "Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-Checker," *Formal Aspects of Computing,* vol. 11, no. 6, pp. 637-664, 1999.
[33] J. Lilius and I. PorresPaltor, "vUML: A Tool for Verifying UML Models," *Proc. Int'l Conf. Automated Software Eng. (ASE '99),* pp. 255-258, 1999.
[34] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems.* Springer, 1992.
[35] T. Murata, "Petri Nets: Properties, Analysis, and Applications," *Proc. IEEE,* vol. 77, no. 4, pp. 541-580, 1989.
[36] P. Muth, D. Wodtke, J. Weissenfels, G. Weikum, and A. Kotz-Dittrich, "Enterprise-Wide Workflow Management Based on State and Activity Charts," *Workflow Management Systems and Interoperability,* A. Dogac, L. Kalinichenko, T. Özsu, and A. Sheth, eds., 1998.
[37] A. Pnueli and E. Shahar, "A Platform Combining Deductive with Algorithmic Verification," *Proc. Int'l Conf. Computer Aided Verification (CAV '96),* R. Alur and T.A. Henzinger, eds., 1996.
[38] P. Spruit, R. Wieringa, and J.J. Meyer, "Regular Database Update Logics," *Theoretical Computer Science,* vol. 254, nos. 1-2, pp. 591-661, 2001.
[39] UML Revision Taskforce, "OMG UML Specification v.1.5," OMG Document Number formal/2003-03-01, Object Management Group, 2003, http://www.omg.org.
[40] F. Tip, "A Survey of Program Slicing Techniques," *J. Programming Languages,* vol. 3, pp. 121-189, 1995.
[41] H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst, "Diagnosing Workflow Processes Using Woflan," *The Computer J.,* vol. 44, no. 4, pp. 246-279, 2001.
[42] R.J. Wieringa, *Design Methods for Reactive Systems.* Morgan Kaufmann, 2003.
[43] Workflow Management Coalition, "Workflow Management Coalition Specification—Terminology & Glossary," WFMC document WFMC-TC-1011, 1999, available at http://www.wfmc.org.

Rik Eshuis received the MSc and PhD degrees in computer science from the University of Twente in 1998 and 2002, respectively. Afterward, he was a visiting postdoctoral researcher at the CRP Henri Tudor and LIASIT in Luxembourg. Currently, he is an assistant professor at Eindhoven University of Technology, The Netherlands. His research interests are in semantics and verification of UML activity diagrams and statecharts, process modeling languages, transformation algorithms between process modeling languages, cross-organizational workflow management, and service composition.



Roel Wieringa holds the chair of information systems at the University of Twente, The Netherlands. His research interests include methods for requirements specification and architecture design of cooperative information systems, the analysis and design of coordination support within and across organizations, methods for aligning information systems to business architecture, and the integration of formal and informal specification techniques. His book on requirements engineering was published in 1996 by Wiley and a book on design methods for reactive systems was published in 2003 by Morgan Kaufmann. He is chair of the steering committee of the IEEE International Requirements Engineering Conference.