Open access • Proceedings Article • DOI:10.1145/581339.581362

Verification support for workflow design with UML activity graphs — Source link

Rik Eshuis, Roel Wieringa

Published on: 19 May 2002 - International Conference on Software Engineering

Topics: Model checking, Petri net, Graph theory, Workflow and Formal specification

Related papers:

- The application of Petri-nets to workflow management
- UML Activity Diagrams as a Workflow Specification Language
- Workflow Patterns
- Analyzing process models using graph reduction techniques
- Verification of workflow task structures: A petri-net-based approach

Share this paper:

View more about this paper here: https://typeset.io/papers/verification-support-for-workflow-design-with-uml-activity-4zpam8h1ln

Verification Support for Workflow Design with UML Activity Graphs

Rik Eshuis*
eshuis@cs.uwente.nl

Roel Wieringa
roelw@cs.utwente.nl

Department of Computer Science, University of Twente
P.O.Box 217, 7500 AE Enschede, The Netherlands

ABSTRACT

We describe a tool that supports verification of workflow models specified in UML activity graphs. The tool translates an activity graph into an input format for a model checker according to a semantics we published earlier. 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. The tool automatically translates such an error trace into an activity graph trace by high-lighting a corresponding path in the activity graph. One of the problems that is dealt with is that model checkers require a finite state space whereas workflow models in general have an infinite state space. Another problem is that strong fairness is necessary to obtain realistic results. Only model checkers that use a special model checking algorithm for strong fairness are suitable for verifying workflow models. We analyse the structure of the state space. We illustrate our approach with some example verifications.

1. INTRODUCTION

Workflow management is used on a wide scale nowadays, and is supported by tools for editing workflow models as well as tools for the verification of performance properties of workflow models, such as throughput and workload analysis tools [1, 25]. However, the trend is that modern workflow applications are being integrated with Enterprise Resource Planning, e-commerce applications, cross-organisational workflow and flexible case management (see e.g. [4, 18]). These new applications create drastically more complex workflows whose functional properties are not easily checked by visually inspecting a workflow model. The presence of event-driven behaviour, real-time, unrestricted loops, parallelism and distribution make it easy to specify workflow models that have undesirable properties. Since it is very expensive to change a workflow model that has active instances, it is desirable to have tools available for the verification of functional workflow model properties. We propose an approach to property verification based on model checking UML activity graphs [33]. (See Clarke et al. [7] for an introduction to model checking.)

To be able to model check functional properties at build-time, we need a formal semantics of workflow models. This may make the choice for UML activity graphs less obvious, because at the present time, there is no generally accepted formal semantics of UML activity graphs and the informal OMG semantics is not yet suitable for workflow modelling [13]. At the same time, Petri nets are widely used for workflow modelling and they have a formal semantics for which model checkers exist [17]. However, we think that UML activity graphs are better suited for specifying data and event-driven behaviour than Petri nets. Formalisations of Petri nets have problems with modelling these aspects of a workflow [12]. As a consequence, *formal* Petri net models often do not very well reflect the behaviour of actual workflows, whereas on the other hand Petri net models that *do* reflect the behaviour of actual workflows, do not have a formal semantics. But for model checking workflow models we need both a formal semantics and an accurate representation of workflow behaviour.

In a previous paper [11], we proposed a formal execution semantics for activity graphs, intended for workflow modelling, that can be used for model checking. The semantics deals with data, real-time, and event-driven processing. In Section 2, we briefly summarise this semantics. We have implemented this semantics in the Toolkit for Conceptual Modeling (TCM) [9], a set of diagram editing tools, one of which is a tool for drawing activity graphs. Figure 1 shows the outline of the verification support we have developed for workflow modellers. The mapping of the activity graph into the transition system implements the execution semantics. The implementation is available at `http://www.cs.utwente.nl/~eshuis/tatd.html`.

The intended way of working with the verification tool is as follows. The workflow modeller specifies an activity graph with TCM. This activity graph is a workflow model. Using a formal property language, the workflow modeller can define requirements that the intended workflow model must satisfy. Both the activity graph and the requirements must be formal, since they are interpreted by a software tool according to a formal semantics. TCM generates a transition system from the activity graph and translates the requirement into

*Supported by NWO/SION, grant nr. 612-62-02 (DAEMON).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'02, May 19-25, 2002, Orlando, Florida, USA.
Copyright 2002 ACM 1-58113-472-X/02/0005...$5.00.

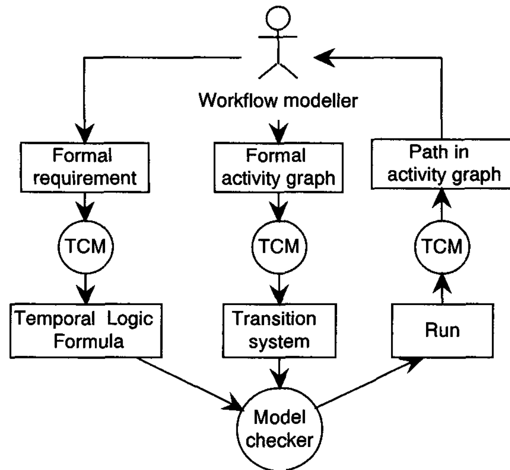

Figure 1: Tool architecture

a temporal logic formula. Currently, the requirements language is a syntactic sugarring of the temporal logic language, but in future work we intend to specify a more abstract requirements language that is closer to the business level. The most commonly used temporal logics are Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [7]. Both the transition system and the temporal logic formula are input for a model checker that checks whether the transition system satisfies the temporal logic formula. If the temporal logic formula fails to hold, the model checker generates an example run (also known as scenario or trace) which shows the sequence of states that lead to violation of the requirement. The corresponding path is then visualised in the activity graph. The workflow modeller can then either change the requirement or the activity graph and do the verification again.

Initially, we included support for as many model checkers as possible and have used for example NuSMV [6] and Spin [21]. However, not every model checker turned out to be useful. As we will argue in Section 5, only model checkers supporting *strong fairness* constraints are useful for workflow models. Since strong fairness is an LTL property and most model checkers support CTL properties only, only a few model checkers are useful for our purpose. Even worse, most LTL model checkers do not use a special model checking algorithm for strong fairness, i.e., they do not support verification of strong fairness at the algorithmic level. Consequently, their performance is so bad that they cannot be used either. We therefore decided to implement in the NuSMV [6] model checker an existing LTL model checking algorithm of Kesten et al. [26] that is intended for strong fairness. The results that are obtained with our implementation are encouraging (see Sections 5 and 6).

The contribution of our work is on three different domains, namely (1) workflow modelling, (2) UML and (3) model checking. (1) Our tool offers flexible analysis support of workflow models that have event driven behaviour, data, loops, and real-time. Previous approaches either focus on fixed analysis properties for simple workflow models without data or real-time (Woflan), whereas others support only flexible analysis for models without loops and data (Testbed Studio) or do not support strong fairness constraints (Mentor) (see Section 8). Besides, the model checker we use is hidden from the user: TCM generates the model checker's input transition system directly from the activity graph and it translates the model checker's feedback directly into a path of the activity graph. TCM also generates the strong fairness constraints for the input model automatically. The user merely has to specify the property she wants to verify.

(2) As far as we know, our tool is the first verification tool for UML activity graphs. Existing verification tools for the related UML statecharts do not present feedback in terms of the original statechart and ignore the issue of strong fairness.

(3) Our work shows that only model checkers supporting strong fairness constraints are useful for verification of workflow models. Usually, strong fairness constraints are used for verifying properties of concurrent processes only (e.g. to show the absence of starvation). Workflow models are a new application domain for strong fairness constraints. Only model checkers supporting strong fairness at the algorithmic level (see Section 5) turned out to be useful.

The structure of this paper is as follows. We start with an explanation in Section 2 of the syntax and semantics of UML activity graphs. In Section 3 we explain how we transform an infinite state space to a finite one. In Section 4 we sketch the structure of our implementation. Section 5 discusses what strong fairness is and why it is needed. Section 6 gives some example verifications of requirements of workflow models. In Section 7 we analyse the factors of the activity graph that impact the size of the state space and we show how the state space can be reduced while retaining properties. In Section 8 we discuss related work. We end with discussion and conclusions.

2. SYNTAX AND SEMANTICS OF UML ACTIVITY GRAPHS

Syntax. Figure 2 shows an example activity graph specification of a workflow model. Ovals represent activity states and rounded rectangles represent wait states. The example models the workflow of a small production company (adapted from [36]). First, an order is received. Next, the departments Production and Finance are put to work. Finance checks whether the customer's account limit is not exceeded by accepting the order. If Finance rejects the order, the whole workflow stops. Otherwise, Finance sends a bill to the customer and waits until the customer pays. Production checks whether the desired product is still in stock. If not, a production plan must be made to produce the product. If according to Finance the order may be accepted, the product is either produced or taken from stock. If both Production and Finance have finished, the product is shipped to the customer.

We follow the UML terminology [33] and call an activity state node an action state node. Crucial in our semantics is that an activity graph represents the behaviour of a workflow management system (WFMS), not of its environment. The environment of a WFMS consists of actors (people or software) that perform activities. The WFMS monitors these activities but does not perform them. An action state node represents a WFMS state in which the WFMS waits for

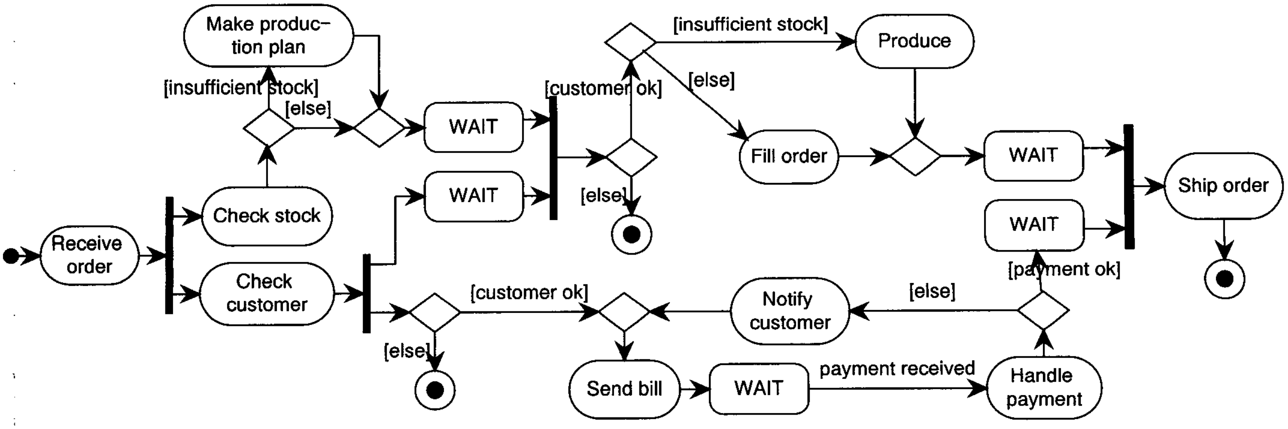

Figure 2: Workflow of production company

an actor to complete its work. During an activity, the actor may change attributes of the workflow. An activity is atomic. A wait state node represents a WFMS state in which the WFMS waits for some external events to occur, e.g. the WFMS waits for a deadline to occur, or for a third party to send some information. The workflow starts in the black dot (the initial state) and ends at a bull's eye (an end state). A bar represents an AND-node, which is either a fork or a join but not both. A diamond represents an XOR-node, which is either a split or a merge but not both. The AND and XOR-nodes in a UML activity graph are pseudo state nodes, i.e. they are transient state nodes that are not part of any legal state of the activity graph. As the UML [33] does, we regard them as syntactic sugar to denote complex hyperedges (called compound transitions in UML).

State nodes (including pseudo state nodes) are linked by directed edges, expressing sequence. An edge can be labelled by $e[g]$ where e is an event expression and g a guard expression. Each of these two components is optional. An edge that leaves an action state node cannot have an event expression in its label, since that would denote an interrupt, whereas an activity cannot be interrupted, since it is atomic. A guard expression can refer to variables of the activity graph. The variables of an activity graph are booleans, integers and strings. Guard expressions can be combined using $\wedge, \vee$, and $\neg$. We do not allow action expressions on the label since these would express changes of the workflow attributes performed by the WFMS and as we just explained, a WFMS does not change the workflow attributes.

A special kind of event expressions are the temporal ones. A when event denotes an absolute time event, for example when(12:00hs), whereas 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.

In our formal semantics [11], we have extended the UML definition of activity graphs by specifying pre and post conditions of activities, that constrain the in- and output of activities. There, we specify for each activity the variables it reads and updates. This latter information is needed to ensure transactional properties of activities.

As stated above, the pseudo state nodes of an activity graph do not represent any legal state of the activity graph. In order to give a semantics to an activity graph, we eliminate pseudo state nodes, by mapping an activity graph into an activity hypergraph.

An *activity hypergraph* consists of a set of labelled state nodes that are connected by labelled directed hyperedges (a hyperedge is an edge with one or more source state nodes and one or more target state nodes). State nodes of an activity hypergraph are action state nodes, wait state nodes, initial state nodes and final state nodes. An activity hypergraph can have variables. A hyperedge can be labelled with an optional event expression and an optional guard expression, where the latter can refer to variables of the activity hypergraph. Figure 3 shows the activity hypergraph of Fig. 2.

An activity graph is mapped into an activity hypergraph as follows.

- The ordinary, i.e. non-pseudo, state nodes of the activity graph become the state nodes of the activity hypergraph.
- The variables of the activity graph become the variables of the activity hypergraph.
- Hyperedges are computed using the notion of a compound transition. A compound transition [19, 33] is an acyclical, maximal chain of edges, linked by pseudo state nodes. Every compound transition maps into a hyperedge by eliminating the pseudo state nodes.

Figure 4 shows some of the most common eliminations. Roughly speaking, for every XOR-node every pair of entering and exiting edges maps into one compound transition. And for every AND-node, all its entering and exiting edges map into the same compound transition. If AND-nodes are connected to OR-nodes, the mapping becomes slightly more complicated. Our full report [10] gives all details. The mapping is implemented in TCM and is invisible to the user (except if the mapping fails, e.g. because there is a cycle between pseudo state nodes; then an error is raised).

Semantics. Our semantics maps an activity hypergraph into a transition system. The transition system we use is a

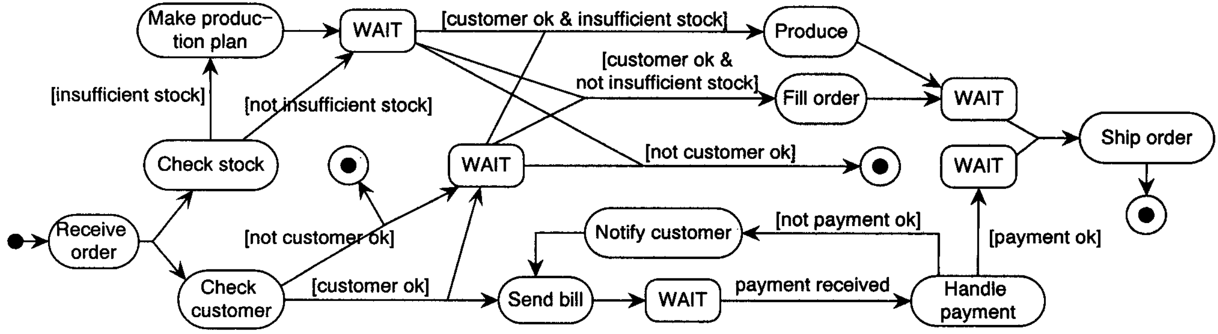

Figure 3: Activity hypergraph of Fig. 2

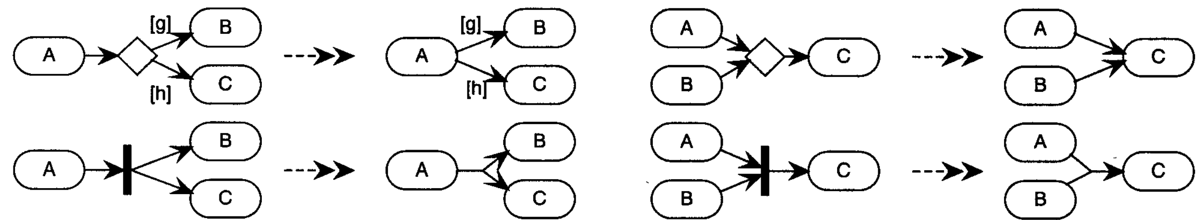

Figure 4: Example eliminations of pseudo state nodes

Kripke structure. Let *Var* be a set of variables. A valuation σ of *Var* assigns to every variable $v \in Var$ a value $\sigma(v)$ in that variable's domain. A valuation represents a possible state of the world. A *Kripke structure* on *Var* is a transition system $(Var, \longrightarrow, \sigma_0)$. Its states are valuations of *Var*. A transition $\sigma \longrightarrow \sigma'$ in the Kripke structure represents that some variables change value, i.e. the world changes. Model checking requires that relation $\longrightarrow$ is total. Valuation σ_0 is the initial valuation. A run of the Kripke structure is an infinite sequence of states connected by transitions.

Since an activity hypergraph represents the behaviour of a WFMS, a state of the Kripke structure in this case is a state of the WFMS. This state is represented by five variables: (1) the bag of nodes that are currently active in the workflow model (called *configuration*), (2) the set of current event occurrences, (3) the valuation of the local variables, (4) the bag of terminated action state nodes, and (5) the current value of the running timers. An action state node terminates iff the activity it controls terminates. Timers are special clock variables that measure the progress of time. They are necessary to generate timeouts.

We view a WFMS as a reactive system [20, 35]. As any reactive system, the WFMS reacts to events it receives from its environment (e.g. activity completion events), based upon its current state, by performing certain desired actions in its environment (in the case of completion events the enabling, but not the execution, of new activities), and updating its current state. For example, if in Fig. 3 the configuration contains node Check stock, so activity *Check stock* is active, and the WFMS receives the completion event of that activity, then the WFMS reacts by enabling activity *Make production plan* if insufficient stock is *true*, and by updating the configuration: node Check stock is removed and either Make production plan or WAIT is inserted, and the set of input events is reset.

In previous work, we have defined a requirements-level semantics [11] and an implementation-level semantics [13]. Key property of the requirements-level semantics is that the perfect synchrony hypothesis [3] is adopted: the system reacts immediately and infinitely fast to events it receives. Since the system is infinitely fast, while a system is reacting to an event, another event can never occur. Also, taking a hyperedge does not cost time. In the implementation-level semantics, we have dropped the perfect-synchrony hypothesis. Hence, 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. Advantage of the requirements-level semantics is that models in this semantics are easy to check and simple to understand, whereas the main disadvantage is that the perfect synchrony assumption may make some of the behaviour of models in this semantics not very realistic [12]. By contrast, the advantage of the implementation-level semantics is that models in this semantics are realistic, but the disadvantages are that models in this semantics are difficult to check and difficult to understand [12]. In this paper, we focus on the requirements-level semantics. We postpone model checking of the implementation-level semantics to future work.

In the requirements-level semantics, during a reaction of the system, the state of the system changes as follows. The configuration is updated (i.e., the state of the workflow model changes and the WFMS enables some new activities to be executed), the bag of terminated action state nodes is updated, the set of current event occurrences is reset, but the local variables do not change. Local variables are updated by actors in activities, not by the WFMS in a reaction. Timers are not increased, since a reaction is instantaneous. Some timers are started, however, since they become relevant, and

some are stopped, since they become irrelevant.

In a reaction, one configuration is updated into another one by taking a bag of hyperedges, called a *step.* Not every bag of hyperedges is a step. First, a hyperedge can only be part of a step if it is enabled. A hyperedge is *enabled* if and only if it is relevant, its trigger event occurs and its guard condition is true. A hyperedge is *relevant* in a configuration iff all its source state nodes are currently active, i.e. they are contained in the configuration, and all source action state nodes have terminated, i.e., they are contained in the bag of terminated action state nodes. Second, some enabled hyperedges can be in conflict with each other. A bag of enabled hyperedges is *consistent* iff the bag union of their source state nodes is contained in the current configuration (i.e. they can be taken simultaneously). For example, in Fig. 3 the two hyperedges leaving Check stock are consistent if and only if the current configuration contains at least two copies of Check stock. Since at most one copy of Check stock will be active at the same time, the two hyperedges will always be inconsistent. Also, a bag of enabled hyperedges must satisfy the transactional isolation property that in the next configuration no two activities update the same variable. If a bag of hyperedges satisfies this constraint, we call this bag *non-interfering.* Now, a *step* is defined as a consistent, non-interfering bag of enabled hyperedges that is *maximal*, i.e. adding an enabled hyperedge would make the step either inconsistent or interfering.

In the requirements-level semantics, we have adopted the STATEMATE [19] semantics of a system reaction. Informally, the semantics is as follows. Initially, the system is in a stable state, i.e., there are no events and no hyperedges are enabled, so no step can be taken. When events occur, the system state becomes unstable. To reach a stable state again, the system reacts by taking a step and entering a new state. If this new state is unstable as well, again a step is taken, otherwise the system stops taking steps. This sequence of taking a step and entering a new state and testing whether the new state is stable, is repeated until a stable state is reached, in which no hyperedges are enabled. Thus, the system reaction is a sequence of steps, called a *superstep* [19]. Note that the perfect synchrony hypothesis implies that (1) while the system is taking a superstep, no other events can occur in the environment, and (2) time does not elapse in unstable states.

The environment behaves nondeterministically. It can generate events to which the system reacts at any time. Since it can generate events at any time, we use a dense time model.

3. FROM INFINITE TO FINITE STATE SPACE

The semantics sketched in the previous section is not yet suitable for model checking, since the transition system of the activity graph can have an infinite state space whereas model checking requires that the state space be finite. In this section we describe how our implementation deals with infinite state spaces.

Unbounded State Nodes. Combining fork and merge, we can specify workflow models and patterns in which multiple instances of the same state node are active at the same time [2]. Figure 5 shows two example activity graphs in which a state node can occur more than once in the same configuration. In the top activity graph, arbitrarily many

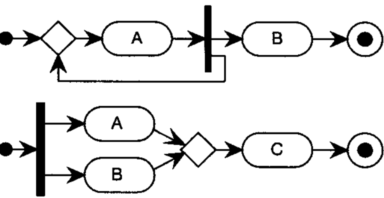

Figure 5: Examples of multiple state instances.

instances of B can be active at the same time. In the lower activity graph, C is executed twice; two instances of C can be active at the same time. These activity graphs might have an infinite state space, if one of the state nodes in the activity graph is unbounded. A state node is *unbounded* if there is no bound on the maximum number of its active instances. For example, the top activity graph in Fig. 5 has an infinite state space since node B is unbounded, but the lower activity graph has a finite state space. Model checking is decidable for finite models [7], but for infinite models it can become easily undecidable [14]. We therefore restrict ourselves to bounded models, which have a finite state space. In our implementation, the computation of the transition system is stopped if one of the state 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 and $C_{s'}$ does not contain n. For example, in the top activity graph in Fig. 5, node B is unbounded since a state with configuration [A,B] is reachable from a state with configuration [A]. (This criterion is derived from the Karp-Miller algorithm that computes the coverability graph of a possibly infinite Petri net [24]; the notion of unboundedness stems from Petri net theory.)

Abstracting from Data. Since an activity hypergraph can have integer and string variables, the state space of the transition system can be infinite. We reduce this infinite transition system to a finite one as follows.

The key observation is 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. For example, suppose a guard tests whether variable $x < 10$. Then we only need to know the truth value of the guard, if we want to know whether the associated hyperedge is enabled.

A naive model checking strategy would therefore be to drop all data and to introduce for every guard expression a boolean representative. The guard is true iff its boolean representative is true. This strategy is naive in the sense that it ignores that guard expressions can be dependent upon each other. For example, if guard expression $[p \wedge q]$ is true then $[p]$ must also be true. And if [s="*red*"] is true then [$s\neq$"*red*"] must be false, and vice versa. But in the naive model checking strategy, $[p \wedge q]$ and $[p]$ might be assigned conflicting truth values, for example $[p \wedge q]$ = *true* and $[p]$ = *false*. Such valuations are infeasible, and therefore 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 $[x = 10]$ and $[x \geq 10]$ are not independent, since $x = 10 \Rightarrow x \geq 10$.

We solve this problem by requiring that a basic guard expression can at most refer to one variable, and that if two basic guard expressions refer to the same variable, then they must be syntactically the same. This may seem a limiting constraint, but we have not yet seen a workflow model in practice that did not satisfy this constraint. We postpone relaxing this constraint to future work.

The approach above is based on existing approaches from modal logic theory, e.g. filtration [15]. Similar techniques are also applied in model checking under the name partition refinement [8]. Partition refinement can only be applied to a finite state space. Therefore, as far as we know, partition refinement is never applied to data abstraction, since data may induce an infinite state space.

Real time. Activity graphs can contain simple real-time constructs of the form when and after (see Section 2). In our prototype, we have only implemented after constraints; when constraints can be dealt with similarly. In computing a transition system, we need to interpret after constraints in order to generate timeouts. One obvious solution is to use discrete timers. But in our semantics we have dense time rather than discrete time: an event can occur at any time, not just at clock ticks. A dense time model cannot be discretised straightforwardly, since the discretisation may introduce some (undesired) properties that are not present in the original dense time model. However, in our case, we can use the result of Göllü et al. [16] that dense time models with n timers can be discretised using clock ticks of $1/(n+1)$. This discretisation preserves the untimed (reachability) properties of the original dense time model, but it may introduce some different timing behaviour [16]. So it is not possible to use a real-time logic as property language. But since there is no real-time model checker supporting strong fairness constraints, we are subject to this limitation anyway.

4. IMPLEMENTATION

We first discuss some simplifying assumptions we made in our implementation. Next, the implementation itself is discussed.

Assumptions. First, our semantics [11] requires that for every activity A the variables that A reads and updates be specified. To deal with this, we have adopted the following assumptions:

1. If an activity reads a variable, we assume that it update that variable too.

2. We assume that a variable is updated by an activity A iff there exists an hyperedge h such that one of $h's$ sources is labelled A and the variable is tested in h's guard expression. So in Fig. 2 we assume that Check stock updates boolean variable insufficient stock.

These two assumptions make it possible to deduce for every activity automatically what variables it updates. So the user does not have to provide this information.

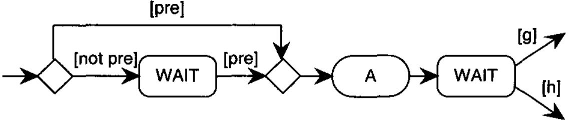

Figure 6: Modelling pre and post conditions

Next, in our formal semantics [11] we require that every activity have a pre and post condition that constrains what actors can do with the activities. To deal with pre and post conditions, in our prototype implementation we have adopted the following assumptions (Fig. 6):

3. For every action state node a that controls an activity with a pre condition *pre*, there should be a preceding wait state node. This wait state node is connected to a by an edge with a guard label [pre]. Our semantics will take care that the system stays in the wait state node as long as the pre condition is *false*. If the pre condition becomes *true*, our semantics will ensure that the wait state node is left, the action state node is immediately entered and the activity is started. Our semantics ensures that the pre condition can only be evaluated iff all updated and observed variables it refers too, are not being updated anymore. If the pre condition is true it can be omitted. (In Fig. 2 all the activities have pre conditions true.)

4. Every post condition *post* of an activity implies the disjunction of the guards of all the (hyper)edges that leave the corresponding action state node. For example, in Fig. 6 we have that $post \Rightarrow [g] \vee [h]$. Also, we assume that the system cannot get stuck due to a false post condition, i.e., every activity will satisfy its post condition when it terminates.

Under these two assumptions we can abstract away from pre and post conditions, since these are already implicitly modelled by the guard conditions.

Next, we use the following data abstraction rule:

5. The effect of an activity is the possible change in valuation of the variables that the activity updates. Since the only relevant changes are changes in 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.

An activity updates those basic guard expressions that contain a variable that is updated by that activity. As explained in the previous Section, we do not allow basic guard expressions that contain more than one variable.

6. The data that is updated in an activity is not updated by the environment.

Not making this assumption would make some workflow models counter intuitive. For example, in Fig. 2 the two choices based upon insufficient stock should have the same outcome. If above assumption is not made, the two choices might have different outcomes, which is undesirable. A nice effect of the assumption is that it reduces the state space explosion.

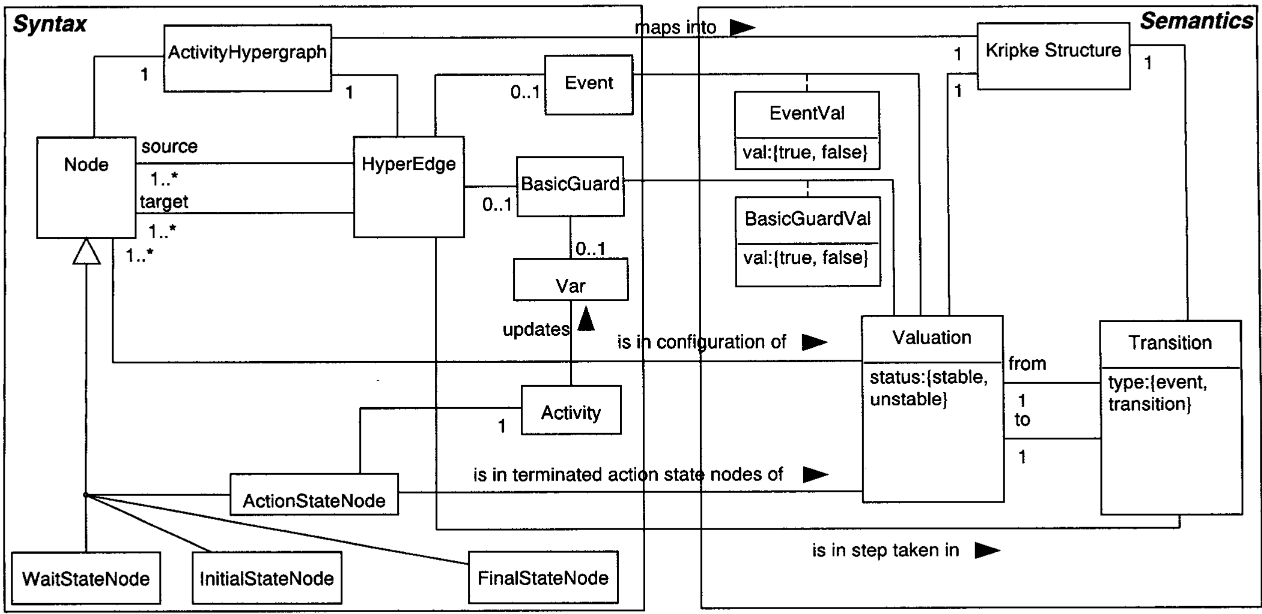

Figure 7: Meta model of implementation

Implementation. Figure 7 shows a meta model in UML notation of our implementation. The model does not show how an activity graph is translated into an activity hypergraph. Although we have implemented this, this is not the interesting part of the implementation.

The main part of our implementation consists of an iterative algorithm that processes states (valuations) of the Kripke structure. A state is processed in one of the following two ways.

- If a processed state is stable, either the timers tick and a new stable state is reached, or some events occur and the next state becomes unstable. The algorithm computes all possible unstable states that can occur next. In an unstable next state, either the set of input events is filled, or some basic guards change value, or some action state nodes terminate, or some timeouts occur.
- If a processed state is unstable, the algorithm computes all possible steps and all the resulting target states that are reached when those steps are taken. A resulting target state is stable if there are no enabled hyperedges in this state; it is unstable otherwise.

The new states that are generated while a state is processed, are processed later. The algorithm stops if all states have been processed. The resulting transition system can be straightforwardly encoded as input for a model checker.

In Section 7 we analyse the structure and size of the state space and analyse different ways of reducing it.

5. STRONG FAIRNESS

Workflow models can contain loops. Consider for example the activity graph in Fig. 2. There is a loop Send bill-WAIT-Handle payment-Notify Customer-Send Bill-.... Without any further hint, every model checker will think the system can stay forever in such a loop. This is not what is intended. A workflow will eventually terminate and will not stay forever in a loop. So in this example, the payment will eventually be ok.

At first sight, it may seem as if loops are only introduced directly in the control flow, as in Fig. 2. But even a workflow model that has no loops in the control flow may have loops in its Kripke structure. This is due to event occurrences that can occur in a certain state but that are irrelevant and therefore ignored. For example, in Fig. 8 event e can occur while A is active, but then it is simply ignored. Nothing in our semantics prevents e from happening over and over again while A is active. The run in Fig. 8 would therefore be a valid run. But we want to exclude such a run because in it, A never terminates while e occurs infinitely often.

To exclude these loops, we have to find a way to instruct the model checker that the loops will be exited eventually. A useful way to specify this is to use strong fairness (also known as compassion) constraints. A strong fairness constraint (p, q), where p and q are properties, states that if p is true infinitely often, then q must be true infinitely often as well. Intuitively, a property p can only be true infinitely often if there is some kind of loop in the model in which p is made true. So the strong fairness constraint (p, q) says that if there is some loop which makes p true infinitely often, then q must be made true infinitely often by the loop as well. If this is not the case, the loop is not strongly fair and the loop must be exited after a finite number of iterations. Using a strong fairness constraint, therefore, we can specify that some loop must be exited eventually. For example, in Fig. 8 the run does not satisfy the strong fairness constraint $([\mathsf{A}] \sqsubseteq C, [\mathsf{WAIT}] \sqsubseteq C)$, where C is the configuration, because node A is infinitely often contained in the configuration, but node WAIT is not.

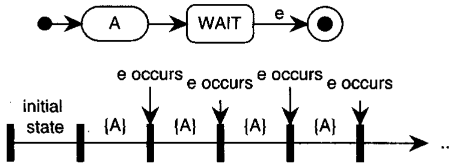

Figure 8: Example of hidden loops

There are several ways to encode strong fairness constraints in the workflow models. We have chosen to encode for every hyperedge h a fairness constraint that states that if h is relevant infinitely often, it must be taken infinitely often[1]: $(source(h) \sqsubseteq C, target(h) \sqsubseteq C)$. An alternative encoding is to specify a strong fairness constraint for each cycle in the generated Kripke structure. But this results in a far greater number of strong fairness constraints, since a workflow model with external events will have cycles in almost every state (cf. Fig. 8).

Each strong fairness constraint (p, q) is equivalent to LTL constraint $\mathsf{G}\ \mathsf{F}\ p \Rightarrow \mathsf{G}\ \mathsf{F}\ q$, where $\mathsf{G}\ \phi$ means that ϕ is globally true in every state of the run and $\mathsf{F}\ \phi$ means that ϕ is true in some future state of the run. At first, we tried to encode the strong fairness constraints as antecedent of the LTL property that has to be verified and then use an ordinary LTL model checker like NuSMV or Spin. Since we have a lot of strong fairness constraints, however, verification of these models was undoable in practice. To illustrate this, our example has 19 hyperedges, so 19 strong fairness constraints. This already is too much for both NuSMV and Spin: we where not able to verify the simple property *fairness* $\Rightarrow$ *false* (true iff the model has no run), where *fairness* is the conjunction of the strong fairness constraints for every hyperedge, as explained above.

We therefore decided to take TLV [32], a model checker that has a special model checking algorithm for strong fairness constraints. The strong fairness constraints are given separately from the property that has to be verified. The algorithm restricts the evaluation of a property to strongly fair runs only. The algorithm that TLV uses, is described in Kesten et al. [26]. TLV performed significantly better than NuSMV and Spin: TLV only took 20 seconds to verify *false* under the strong fairness constraints. But unfortunately, TLV does not support batch processing, so we could not integrate it into TCM. We therefore implemented the mentioned algorithm of Kesten et al. [26] in the open source model checker NuSMV, which does support batch processing. The resulting strong fairness model checker, called NuSMV_{fair}, can be downloaded from `http://www.cs.utwente.nl/~eshuis/nusmvfair.html`.

6. EXAMPLE VERIFICATIONS

We discuss some example verifications of requirements for the workflow model of Fig. 2. We distinguish general and ad-hoc requirements. A general requirement must hold for every possible activity graph, while an ad-hoc requirement is specified for a specific activity graph.

General requirements. There are several general requirements possible. We focus on the following basic requirement: for each strongly fair run, from the initial state a final state should be reachable. We formulate this requirement as the following LTL formula:

(R1): $\mathsf{F}\ \mathsf{G}\ \mathit{final}$

where *final* is a predicate that is true iff the current configuration only contains final state nodes. (TCM translates *final* into an equivalent predicate on state nodes.) TCM automatically generates the appropriate strong fairness constraints for the workflow model. NuSMV_{fair} reports that the property is true.

Other useful general requirements are that there are no dead nodes, i.e., for every node there is a run in which that node becomes active; and that there are no dead hyperedges.

Ad-hoc requirements. Since every workflow model might have its own ad-hoc requirements, we just give an example for the workflow model shown in Fig. 2.

Ad-hoc requirement R2 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. We begin with formalising this property as:

(R2): $\mathsf{F}\ \mathsf{in}(\mathit{Make\ production\ plan}) \Leftrightarrow \mathsf{F}\ \mathsf{in}(\mathit{Produce})$

where $\mathsf{in}(x)$ is true in a valuation σ iff node x is contained in the configuration of σ. (TCM translates in into an equivalent predicate on state nodes.) This property fails to hold: The path that TCM highlights is shown in Fig. 9. We see that if the customer check fails, the workflow stops while activity Make production plan may already have been executed.

There are two ways to repair this error: either adapt the requirement or the activity graph. We decide to adapt the requirement. Apparently, only if the customer check does not fail, the requirement holds:

(R2'): $(\mathsf{F}\ \mathit{customer\ ok}) \Rightarrow$
$(\mathsf{F}\ \mathsf{in}(\mathit{Make\ production\ plan}) \Leftrightarrow \mathsf{F}\ \mathsf{in}(\mathit{Produce}))$

NuSMV_{fair} reports that this property is true. This property is true, because of our assumption 6 in Section 3 which implies for this workflow model that only Check stock can change variable *insuffient stock*. If we had allowed the environment to change *insufficient stock*, the property would not have been true.

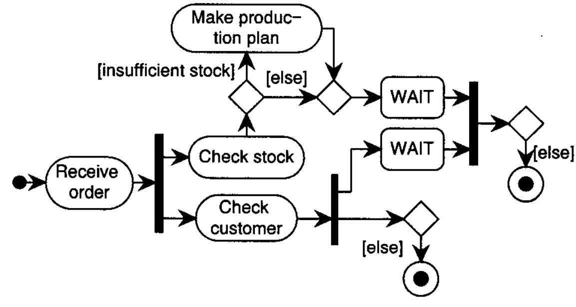

Figure 9: Path illustrating counter example for R2

[1] Strictly speaking, the formalisation above does not express this since it does not state that h should be taken. But for models in which no source and target of a hyperedge is contained in the sources and target of another one, the formalisation is correct.

Requirement	Result	Time (sec)	No.of BDD nodes	Memory allocated (KB)
R1	True	8.0	61448	4855
R2	False	7.8	45835	4755
R2'	True	7.8	49079	4827
R3	True	7.4	48111	4819

Table 1: Resources used by NuSMV$_{fair}$

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:

(R3): F (in(*Produce*) ∨ in(*Fill order*)) ⇔ F in(*Send bill*)

NUSMV$_{fair}$ reports that this property is true.

The resources used by NUSMV$_{fair}$ during this analysis are shown in Table 1. The analysis was performed on a PC with a Pentium III 450 MHz processor with 128Mb of RAM under Red Hat Linux 6.0. We verified the same properties by hand with TLV; the outcomes were the same. NUSMV$_{fair}$ is slightly faster, probably since a more efficient BDD library is used. Most time during analysis was spent by the execution algorithm in TCM that computes the input transition system for the model checker; this time is not shown.

7. ANALYSIS OF THE STATE SPACE

The greatest problem of verification with model checking is the state explosion. For example, even the small example in Fig. 2 has already over 250 states (see Table 2 below), although the number of nodes in the workflow models is only 19. There are several causes for state explosion.

- Parallel branches. These are introduced by hyperedges that have more than one target, and stopped by hyperedges having more than one source. The product of two parallel branches that have x and y states respectively, will have $x \times y$ states. In the example of Fig. 2, due to parallelism there are 41 different configurations, although there are 19 nodes.

- Events. There are external and activity termination events. External events can occur in any stable state, whereas activity termination events can only occur if the corresponding action state node is in the current configuration. Combining this, if there are k external events and in a given stable state the current configuration contains l action state nodes, then there are $2^{k+l} - 1$ (the non-occurrence of events is excluded) possible combinations of events. This means that from this stable state, $2^{k+l}-1$ unstable state can be reached. This accounts for the fact that, although the example in Fig. 2 has only 41 configurations, yet it has over 250 different states (even though there is only 1 named external event!).

  Temporal events (timeouts) can only occur if some timers have reached their limit, see below. Their occurrence is therefore limited.

- Data (basic guard conditions). Due to assumption 6, most data only changes value after their source activity terminates, not by the environment. So data only changes value at some specific time (when some activity terminates). Moreover, because every data is tested in a choice after an activity terminates, there are specific combinations of data and state nodes. For example, in Fig. 2 if node Notify customer is active then payment ok is *false*. Hence, basic guard conditions that are updated by activities do not blow up the state space with respect to stable states.

  But basic guard conditions *do* blow up the state space with respect to unstable states. If after an action state terminates a choice is made out of n alternatives, then there are n different terminations. So n different unstable states are possible. In Fig. 2, for example, Notify customer can terminate in 2 possible ways. In combination with named external event occurrences and activity termination events in parallel branches (see the previous item), this can result in a blowup in the number of unstable states that are possible. For example, Table 2 shows that including payment ok leads to around 20 additional states.

- Real time. Due to clock ticks, a lot of extra states are introduced. For example, if a timer ticks with $1/n$ and the timeout is m, then $m \times n$ ticks are needed before the timeout is generated. This means that $m \times n$ extra states are introduced in this branch. In combination with parallel branches and events, mentioned above, this can result in a large state space explosion.

To illustrate the effect of events and data upon the size of a model, we have computed several variants of the activity graph in Fig. 2. Table 2 shows the results of removing event and guard conditions upon the number of states of the activity graph. A 1 denotes the presence of an item, a 0 denotes its absence.

There are several things worth noticing. First, in this case, including one event doubles the state space. (As a test, we included another dummy event in the model of Fig. 2 on a new edge between WAIT and Handle payment; the resulting model had 535 states.) Second, abstracting from data that is used in one choice only, for example payment ok, does not have a big impact on the state space: the model is reduced by around 20 states. This effect we already explained above in the third item.

Third, perhaps a bit surprisingly, the table shows that removing guards may increase the state space, rather than decreasing it: if guard customer ok is removed, the state space becomes larger. The reason for this is that some choices in parallel branches can be dependent upon each other (in this case the two choices based upon customer ok). This dependency is lost if these choices are made nondeterministic (in this case if customer ok is abstracted away from). Then, some configurations that do not exist when the guard is included, *do* exist if the guard is not included (in this case, when customer ok is not modelled, the branch starting with node Send bill can be active whereas the other parallel branch immediately stops and does not do Produce or Fill order). In the example, removing guard customer ok introduces 8 extra configurations. By the way, removing guard customer ok does have the effect as described under the previous point, but this effect apparently does not weigh up against the extra configurations and states that are introduced.

payment received	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
customer ok	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
insufficient stock	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
payment ok	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
nr of states	277	255	216	198	311	285	234	214	143	132	109	100	160	147	118	108

Table 2: Effects of presence of events and data upon size of the model.

It may seem, therefore, that abstracting away from such guard conditions makes the resulting model useless for analysis. For example, F G *final* is no longer true if guard customer ok is removed from our example. But such abstract models can still be useful, since the dependency between the different choices can be stated as an antecedent to the property to be verified. For example, the property (F in(*Send bill*) ⇔ F (in(*Produce*) ∨ in(*Fill order*))) ⇒ F G *final* is true if guard customer ok is removed. The antecedent states that only those runs should be checked in which a bill is sent if and only if something is produced or an order is filled. Of course, verification of such properties only works if the property does not refer to an item that is abstracted away from.

8. RELATED WORK

There are several workflow verification tools. Woflan [34] is a tool for verification of textual workflow models without data and real-time. It is based on low-level Petri nets. In Woflan the properties that are verified are fixed and cannot be changed by the user. The issue of strong fairness is therefore not relevant. The tool developed in the Mentor project [31] uses a CTL model checker for statecharts [22]. The tool is not integrated with the model checker. The authors do not address strong fairness. They do not provide any details on how the feedback is presented to the user. The Testbed Studio tool [23] supports model checking of business process models with Spin [21]. Only process models that have no loops are supported. The process modelling language neither has external events nor temporal events. The authors do not address strong fairness.

Our work is also closely related to the work done on model checking STATEMATE and UML statecharts. Chan et al. [5] and Mikk [30] have defined model checking for STATEMATE statecharts or variants thereof, using SMV [29] and Spin [21]. Latella et al. [27] present a translation for a subset of UML statecharts to Spin [21]. All these authors encode the syntax of the statechart explicitly in the input language and let the model checking tool derive the step semantics implicitly. We, on the other hand, have programmed our execution algorithm [11] in TCM, so TCM generates the semantic structure directly. In our view, these syntactic encodings are error prone (see for example a discussion by Mikk on errors he found in such translations [30]) and only work for simple models with a restricted syntax. Amongst others, every state node must have a bound of one. This is true for a statechart but not for an activity graph. None of these implementations provide a graphical representation of the feedback of the model checker.

Lilius and Paltor [28] present vUML, a tool for model checking a communicating set of objects whose behaviour is modelled by UML statecharts. They use Spin [21] as their underlying 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. They neither address strong fairness, nor real-time.

It is difficult to compare our work with this work on statecharts, since our semantics differs somewhat from the statechart semantics, both UML and STATEMATE, in particular since we have atomic activity states whose effect is declaratively specified (these are not present in statecharts). Next, we have configurations and steps that are bags rather than sets, since our models can have a bound of more than one. None of the references above use strong fairness constraints, since these are apparently not required in the domain they model (usually embedded real-time systems). Neither do they analyse the state space nor discuss possible reductions.

9. CONCLUSION AND FUTURE WORK

We presented a prototype implementation that supports workflow modellers in verifying workflow models specified in UML activity graphs. Our tool differs from other workflow verification tools because it supports the specification of events, data, real-time and loops in workflow models. Also, the properties that are checked can be specified by the user himself and are not fixed. The appropriate strong fairness constraints are generated automatically by the tool. The used model checker is under the hood of our tool; the user merely has to know an LTL based input language. If the model checker return a counter example, the tool translates this counter examples back into the activity diagram.

The most interesting result is that workflow models require the strong fairness assumption. Although there are some model checkers that support verification of strong fairness constraints, only model checkers that use a special model checking algorithm for strong fairness perform well enough to be useful. Since no existing model checker was suitable for our purposes, we have extended NuSMV with an existing strong fairness model checking algorithm.

Future work includes finding a more abstract requirement specification language, since temporal logic might be difficult to understand for users. To further improve the scalability of our tool, we intend to slice activity graphs in such a way that the proposition is true in the model of the sliced activity graph iff it is true in the original model. Next, we plan to implement model checking for our implementation-level semantics as well. We expect that models in this semantics will at least be twice as big than models in the requirements-level semantics. We therefore plan to study under what restrictions we can use the requirements-level semantics and yet obtain results that are also valid in the implementation-level semantics. In that case, we can avoid using the implementation-level semantics for model checking, and we are then able to tackle larger workflow models.

10. 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. In M. Nielsen and D. Simpson, editors, *Proc. 21st Int.Conf. on Applications and Theory of Petri Nets*, LNCS 1825, 2000.

[2] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced workflow patterns. In O. Etzion and P. Scheuermann, editors, *Proc. CoopIS 2000*, LNCS 1901. Springer, 2000.

[3] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] C. Bussler. Enterprise-wide workflow management. *IEEE Concurrency*, 7(3):32–43, July/September 1999.

[5] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model Checking Large Software Specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, 1998.

[6] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *Int. Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[7] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[8] D. R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, 1996.

[9] F. Dehne, R. Wieringa, and H. van de Zandschulp. Toolkit for conceptual modeling (TCM) — user's guide and reference. Technical report, University of Twente, 2000. `http://www.cs.utwente.nl/~tcm`.

[10] R. Eshuis and R. Wieringa. A formal semantics for UML activity diagrams. Technical Report TR-CTIT-01-04, University of Twente, 2001.

[11] R. Eshuis and R. Wieringa. A real-time execution semantics for UML activity diagrams. In H. Hussmann, editor, *Proc. Fundamental Aspects of Software Engineering (FASE)*, LNCS 2029, pages 76–90. Springer, 2001.

[12] R. Eshuis and R. Wieringa. A comparison of Petri net and activity diagram variants. In *Proc. 2nd Int. Coll. on Petri Net Technologies for Modelling Communication Based Systems*, 2001.

[13] R. Eshuis and R. Wieringa. An execution algorithm for UML activity graphs. In M. Gogolla and C. Kobryn, editors, *Proc. <<UML>> 2001*, LNCS 2185, pages 47–61. Springer, 2001.

[14] J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.

[15] R. Goldblatt. *Logics of Time and Computation*. CSLI Lecture notes 7, Stanford University, 2nd edition, 1992.

[16] A. Göllü, A. Puri, and P. Varaiya. Discretization of timed automata. In *Proc. of the 33rd IEEE conferene on decision and control*, pages 957–958, 1994.

[17] B. Grahlmann. The PEP tool. In O. Grumberg, editor, *Proc. 9th Int. Conf. on Computer Aided Verification*, LNCS 1254. Springer, 1997.

[18] P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig. Crossflow: Cross-organizational workflow management in dynamic virtual enterprises. *Int. Journal of Comp. Systems Science & Engineering*, 15(5):277–290, 2000.

[19] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. on Software Engineering and Methodology*, 5(4):293–333, 1996.

[20] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, NATO/ASI Series nr. 13. Springer, 1985.

[21] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[22] J. Helbig and P. Kelb. An OBDD-Representation of Statecharts. In *Proc. of the European Design and Test Conf.*, pages 142–151. IEEE Comp. Soc. Press, 1994.

[23] W. Janssen, R. Mateescu, S. Mauw, and J. Springintveld. Verifying business processes using Spin. In E. Najm, A. Serhrouchni, and G. Holzmann, editors, *Proc. 4th Int. Spin workshop*, 1998.

[24] R.M. Karp and R.E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.

[25] W.D. Kelton, R.P. Sadowski, and D.A. Sadowski. *Simulation with Arena*. McGraw-Hill, 1998.

[26] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *Int. Coll. on Automata, Languages and Programming (ICALP)*, LNCS 1443. Springer Verlag, 1998.

[27] 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*, 11(6):637–664, 1999.

[28] J. Lilius and I. P. Paltor. vUML: A tool for verifying UML models. In *14th IEEE Int. Conf. on Automated Software Engineering*, pages 255–258, 1999.

[29] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[30] E. Mikk. *Semantics and Verification of Statecharts*. PhD thesis, University of Kiel, 2000.

[31] P. Muth, D. Wodtke, J. Weissenfels, G. Weikum, and A. Kotz-Dittrich. Enterprise-wide workflow management based on state and activity charts. In A. Dogac, L. Kalinichenko, T. Özsu, and A. Sheth, editors, *NATO/ASI Workflow Management Systems and Interoperability*. Springer, 1998.

[32] A. Pnueli and E. Shahar. A platform combining deductive with algorithmic verification. In R. Alur and T.A. Henzinger, editors, *Proc. Int. Conf. on Computer Aided Verification*, LNCS 1102. Springer Verlag, 1996.

[33] UML Revision Taskforce. *OMG UML Specification v. 1.4*. Object Management Group, 2001.

[34] H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing workflow processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.

[35] R. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate and the UML*. Morgan Kaufmann. To be published.

[36] Workflow Management Coalition. Workflow management coalition specification — terminology & glossary (WFMC-TC-1011), 1999. `www.wfmc.org`.