Eliminating Queues from RT UML Model Representations*

Werner Damm[1] and Bengt Jonsson

Uppsala University, Dept. of Computer Systems
S-751 05 Uppsala,
damm@offis.de, bengt@docs.uu.se

Abstract. This paper concerns analyzing UML based models of distributed real time systems involving multiple active agents. In order to avoid the time-penalties incurred by distributed execution of synchronous operation calls, it is typically recommended to restrict inter-task communication to event-based communication through unbounded FIFO buffers. This means that such systems potentially have an infinite number of states, making them out of reach for analysis techniques intended for finite-state systems. We present a symbolic analysis technique of such systems, which can be tuned to give a finite, possibly inexact representation of the state-space. The central idea is to eliminate FIFO buffers completely, and represent their contents implicitly, by their effect on the receiving agent. We propose a natural class of protocols which we call mode separated, for which this representation is both finite and exact. This result has impact on both responsiveness and predictability of end-to-end latencies, as well for the protocol verification, enabling automatic verification methods to be applied.

Keywords: Real-time distributed systems, RT UML, protocol verification, verification of infinite state systems

1 Introduction

We are interested in analysing UML based models of distributed real time systems involving multiple active agents. A central part of this modelling relates to the specification of protocols regulating the co-operation of such agents. Such protocols define the interface between the (possibly complex) processing internal to the agent and those aspects which must be visible to other agents to achieve the global co-operation. A concrete instance of this modelling paradigm is the European Standard on Wireless Train Control currently under development [1], where "agents" correspond to trains, railroad-crossings, switches, or other control points, and the protocol specifies dialogues between such agents, ensuring e.g. that a train only passes a railroad crossing once it has been secured. A simplified model of such a protocol can be found in e.g. [2]. [3] gives a representative example using an executable object model based on UML state-charts for such classes of applications.

* This research was partially supported by DFG USE and the STINT foundation.

[1] On sabbatical leave from Dept. of Computer Science, University of Oldenburg. Oldenburg, FRG

W. Damm and E.-R. Olderog (Eds.): FTRTFT 2002, LNCS 2469, pp. 375–393,2002.

Protocols in transportation applications often relate to safety critical functions of distributed systems. Missing a signalling message in the train system application could potentially cause accidents, hence system components related to such protocol aspects typically would have high safety integrity levels. This paper provides a contribution in assessing the use of UML for modelling such applications. It focuses on those concepts of UML involved in the specification of inter-agent protocols, and assesses the potential to use automatic model-checking based algorithms in their verification. The key result of this paper is, that indeed such verification methods can be applied, given further support for the use of UML and analysing such application classes.

A salient feature imposed from UML is, that state-machines representing the agents communicate by exchanging events, which end up in unbounded FIFO buffers at the receiver side. The verification problem for such models thus entails verification of infinite state systems. In fact, two dimensions of infinity have to be addressed, since in addition to unbounded communication channels such UML models would typically also involve an unbounded number of state-machines. In this paper we focus on providing exact finite abstractions of the communication protocols for a given bounded number of so called *mode-separated* state machines. Intuitively, such state machines use a particular protocol to enter what we call *modes*, i.e. machine states representing global knowledge about the system state.

The analysis of protocols with unbounded FIFO queues has been considered rather extensively in protocol verification (e.g. [4,5,6,7]), and a number of symbolic representations, most of them being variants of regular expressions, have been proposed [8,9,10,11,12,13]. In the current work, we take a different approach, by avoiding to represent explicitly the buffers.

Central to our approach is the observation that in order to capture the behavior of an agent, it is not necessary to represent the concrete contents of the queue of unconsumed incoming events - all that really matters is its future impact on the receiving protocol machine. Thus, instead of giving a symbolic representation of the queue content, we partially evaluate the effect of consuming the events in the queue on the state of the receiver machine. We represent this effect by designating as *pending* the set of possible transitions that could be triggered by consuming events currently in the input buffer. By making a transition pending, we indicate that the part of its guard which requires event consumption can be neglected, when the computation of the state machines eventually analyses enabledness of this transition.

The advantages of using the receiver state machine for representing queue contents is that redundant and irrelevant messages will not be represented at all. A further advantage is that when the symbolic representation grows in complexity, we can use the structure of the receiving state machine as a guide for suitable over-approximations. In fact, our technique allows to tune the degree of approximation. An exact representation cannot in general be finite (this follows from undecidability of the analysis problem [14]) hence we also turn our attention to characterizing classes of protocols for which an exact and finite symbolic representation exists.

A key attribute of the application classes we consider is that the slackness between agents, roughly meaning how closely the local state of one machine is determined by the local state of the other machines, is bounded. Typically, the cooperation protocols are *mode separated*, meaning that they are split into distinct modes, each of which is intuitively associated with a global state of the system. Alluding to the train system application, modes of a train could follow a pattern of moves originating from a

purely local processing state on a track segment well separated from other trains or control points, to an approach mode requiring the initiation of a protocol ensuring a safe passage, to a pass-mode characterising the actual passage of the control point, to a clean-up mode following the passage of the control point. Each such mode typically is supported by distinct phases of the protocol, often requiring a dialogue involving multiple events to be exchanged. Safety concerns stipulate a separation of such modes: there must be no disagreement between agents concerning the sequence of modes visited during an execution, in particular a mode should be re-entered only after all transitions related to the previous activation of this mode have been completed, and modes should not share transitions. In our context, we will give a formal definition of the concept of mode separation, and show that this property implies that our symbolic representation yields a *finite exact* abstraction of the system of state machines.

We conjecture that the technique we are proposing has independent value in ensuring responsiveness of UML based implementations for real-time applications. It can be realised as a pre-compilation phase, compiling away the FIFO buffer, and yielding a model with instantaneous processing of emitted events. This should substantially ease analysis of end-to-end response time, and eliminate the need of determining appropriate FIFO buffer sizes. Moreover, by considering pending transitions as purely local transitions, run-to-completion steps in the precompiled model effectively process multiple events in one sweep, significantly speeding up response time. This is in particular relevant for applications, where events signal hazardous situations requiring immediate attention.

The paper is structured as follows. In the following section, we introduce as a formal model of multi-agent systems an (infinite state) transition system with explicit representations of channel contents, reducing the verification problem of the application domain to a formal reachability analysis of its state space. The subsequent section formalises the idea of introducing pending transitions into the model of a protocol machine, and shows that this yields a finite abstraction for mode-separated protocols. Section 5 gives a simple syntactic condition called mode separation, guaranteeing the abstraction to be exact.

2 Formal Model of UML State Machines

In this section we give a formal model capturing the mathematical essence of UML based specifications of co-operation protocols. We focus here on the co-operation of two active objects, whose behaviour for simplicity we assume to be defined by UML statecharts. The model we propose is scalable to an arbitrary number of active objects, since it explicitly takes into account the effect of an unrestricted environment, which at any point in time can insert events into the event queue of either active object. This section is based on [15] giving a full semantics of the behavioural model of UML. In this paper, we restrict ourselves to pure event-based asynchronous communication between state-machines. Emitting an event does not block the sender machine, and causes the event to be inserted into the event queue of the receiving machine.

We assume as given a set E of events, with typical elements $e, e1, e2, \ldots$. We denote the emitting of event e to the partner state-machine by $e!$, to the sending machine by $self!e$, and to the environment by $env!e$. An emitted event will then be in-

serted into the receiver´s event queue; events emitted to the environment cause no state change in the observed system. The consumption of an event e is denoted by $e?$; note that UML does not allow to qualify such a reception to some selected sender.

A further simplification taken in this paper is to ignore all aspects dealing with local data and operation calls. Thus the action language is reduced to emitting events or performing some un-interpreted local action, denoted α_j – with j ranging over the natural numbers - , for which we assume as given their semantics $[[\alpha_j]]$. We similarly abstract from conditions on data values, by incorporating a countable set γ_j of symbols for local conditions, with semantics $[[\gamma_j]]$. Such conditions can appear as guards of transitions, as may the consumption of an event, or a conjunction of both. Thus, we assume a set A of *actions* with $A = \{ \alpha_j , e! , self!e , env!e \mid e \in E, j \in \mathbb{N}\}$ and a set G of *guards* with $G = \{ \gamma_j , e? , e?[\gamma_j] \mid e \in E, j \in \mathbb{N} \}$.

A UML *state-machine* M is a tuple

$$M = (Q, T, q0)$$

where

- Q is some finite set of states
- $T \subseteq Q \times (G \times A) \times Q$ is the set of labelled transitions
- $q0 \in Q$ is the initial state

We analyse a system $S = M1 \parallel M2$ built from two asynchronously communicating UML state machines $M1$ and $M2$, working in the context of some unspecified environment.

The dynamic behaviour of S is captured as a transition relation over its configurations. A *configuration* c of S is a tuple

$$c = \langle\langle q1 , \sigma 1, \varepsilon 1, \varepsilon' 1\rangle, \langle q2 , \sigma 2 , \varepsilon 2, \varepsilon' 2 \rangle\rangle$$

where

- $q1, q2$ denote the current state of $M1$ and $M2$, respectively
- $\sigma 1, \sigma 2$ denote the current valuation of local data of $M1, M2$, resp.
- $\varepsilon 1 , \varepsilon 2 \in E^*$ denote the current content of event queues associated with $M1, M2$, resp.
- $\varepsilon' 1, \varepsilon' 2 \in E$ denote the set of events emitted in the current step by M1, M2, to the environment

We denote the set of all configurations of S by C_s, or simply C if the denoted system is clear from the context.

A central concept in the execution semantics of UML statecharts is the notion of *run-to-completion steps* (*RTC steps* for short). We call a state q of a UML state machine *stable*, if consuming an event is the only way the computation can proceed, Formally, we define stability of a given state q and data-valuation σ of state-machine Mj as follows:

$$stable(q,\sigma) \equiv \neg (\exists t=(q,\langle \gamma, \varepsilon\rangle,q') \in Tj \quad [[\gamma]](\sigma) = false)$$

In other words: any transition originating from q with a pure local guard is disabled given the current valuation of data-variables σ. Under the RTC semantics, performing local computations has priority over consuming events. Thus, a new event is only dispatched, once the state machine has become stable.

In summary, the transition relation on configurations will thus cover all of the following:

- *local-j*: *Mj* is taking a local computation step (j=1,2)
- *dispatch-j*: *Mj* is stable and an event is dispatched and accepted (j=1,2)
- *discard-j:* *Mj* is stable and the dispatched is not accepted (j=1,2)
- *env-j:* The environment generates an event for machine *Mj* (j=1,2)

Since UML assumes an interleaving semantics, the transition relation capturing the dynamic semantics of *S* is thus defined as the *disjunction* of these sub-relations. We now proceed to define each of the sub-relations.

Local-1

$$c = \langle\langle q1, \sigma 1, \tau 1, \Omega 1\rangle, \langle q2, \sigma 2, \tau 2, \Omega 2\rangle\rangle \; !_{\text{local-1}}$$
$$c' = \langle\langle q1', \sigma 1', \tau 1', \Omega 1'\rangle, \langle q2', \sigma 2', \tau 2', \Omega 2'\rangle\rangle$$

iff

- $\langle q2, \sigma 2\rangle = \langle q2', \sigma 2'\rangle$
- $\Omega 2' = \{\}$
- $\exists (q1, \langle \alpha, a\rangle, q1') \in T1 \;\; [[\alpha]](\sigma) = true \;\wedge$
  $(\alpha = j \wedge a = j \wedge \sigma 1' = [[\alpha j]](\sigma 1) \wedge \tau 1' = \tau 1 \wedge \Omega 1' = \{\} \wedge \sigma 2' = \sigma 2)$
  $\vee (\alpha = e \in E \wedge a = e! \wedge \sigma 2' = \sigma 2 \oplus e \wedge \tau 1' = \tau 1 \wedge \sigma 1' = \sigma 1 \wedge \Omega 1' = \{\})$
  $\vee (\alpha = e \in E \wedge a = self!e \wedge \sigma 1' = \sigma 1 \oplus e \wedge \tau 1' = \tau 1 \wedge \sigma 2' = \sigma 2 \wedge \Omega 1' = \{\})$
  $\vee (\alpha = e \in E \wedge a = env!e \wedge \sigma 1' = \sigma 1 \wedge \sigma 2' = \sigma 2 \wedge \tau 1' = \tau 1 \wedge \Omega 1' = \{e\})$

There are four cases to consider for actions *a*. For a local action $\alpha = j$, we evaluate its semantics $[[\alpha j]]$ on the current valuation of data variables $\sigma 1$, leading to an updated valuation $\sigma 1'$, while leaving both queues unchanged. The other cases deal with emitting events, which are either placed in the queue of *M2*, or in the queue of *M1*, depending on the appearance of the qualifier *self*, or simply emitted without side-effect on the current configuration if addressed to the environment.

Dispatch-1

$$c = \langle\langle q1, \sigma 1, \tau 1, \Omega 1\rangle, \langle q2, \sigma 2, \tau 2, \Omega 2\rangle\rangle \; !_{\text{dispatch-1}}$$
$$c' = \langle\langle q1', \sigma 1', \tau 1', \Omega 1'\rangle, \langle q2', \sigma 2', \tau 2', \Omega 2'\rangle\rangle$$

iff

- $\langle q2, \sigma 2\rangle = \langle q2', \sigma 2'\rangle$
- $\Omega 2' = \{\}$
- $stable(q1, \sigma 1) \wedge \tau 1' = tail(\tau 1)$
- $\exists (q1, \langle e?[\alpha], a\rangle, q1') \in T1 \;\; e = head(\tau 1) \wedge [[\alpha]](\sigma) = true \;\wedge$
  $((\alpha = j \wedge a = j \wedge \sigma 1' = [[\alpha j]](\sigma 1) \wedge \tau 1' = tail(\tau 1) \wedge \sigma 2' = \sigma 2 \wedge \Omega 1' = \{\})$
  $\vee (\alpha = e \in E \wedge a = e \wedge \sigma 2' = \sigma 2 \oplus e \wedge \sigma 1' = \sigma 1 \wedge \tau 1' = tail(\tau 1) \wedge \Omega 1' = \{\})$
  $\vee (\alpha = e \in E \wedge a = self!e \wedge \tau 1' = tail(\tau 1) \wedge e \in \sigma 1' = \sigma 1 \wedge \sigma 2' = \sigma 2 \wedge \Omega 1' = \{\})$
  $\vee (\alpha = e \in E \wedge a = env!e \wedge \sigma 2' = \sigma 2 \wedge \sigma 1' = \sigma 1 \wedge \tau 1' = tail(\tau 1) \wedge \Omega 1' = \{e\}))$

In a stable configuration, the first event of the queue is dispatched. If there is a transition originating from the current state whose trigger matches the dispatched event, and whose associated local condition evaluates to *true*, then one of these is picked, and its action part is evaluated as above. The case of a trigger without local condition is understood to be subsumed by this clause, by assuming the local condition *true*.

Discard-1

$$c = \langle\langle q1, \Box 1, \Box 1, \Box 1\rangle, \langle q2, \Box 2, \Box 2, \Box 2\rangle\rangle \; !_{discard\text{-}1}$$
$$c' = \langle\langle q1', \Box 1', \Box 1', \Box 1'\rangle, \langle q2', \Box 2', \Box 2', \Box 2'\rangle\rangle$$

iff

- $\langle q2, \Box 2, \Box 2\rangle = \langle q2', \Box 2', \Box 2'\rangle$
- $stable(q1, \Box 1) \; \Box \; \Box 1' = tail(\Box 1)$
- $\Box 1' = \Box 2' = \{\}$
- $\langle q1, \Box 1\rangle = \langle q1', \Box 1'\rangle$
- $\Box(q1, \langle e?[\Box], a\rangle, q1') \Box T1 \;\; e = head(\Box 1) \; \Box \;\; [[\Box]](\Box) = false$

If no transition from the current state matches the dispatched event, then the event is *discarded*: it is deleted from the event-queue, and has no other effect on the current state configuration. UML actually allows the specification of so-called *deferred events* for each state; if in the above situation, *e* would have been in the defer-set of *q1*, then *e* would remain in the queue, and the dispatcher would consider the subsequent event. We do not consider handling of deferred events.

Env-1

$$c = \langle\langle q1, \Box 1, \Box 1, \Box 1\rangle, \langle q2, \Box 2, \Box 2, \Box 2\rangle\rangle \; !_{env\text{-}1}$$
$$c' = \langle\langle q1', \Box 1', \Box 1', \Box 1'\rangle, \langle q2', \Box 2', \Box 2', \Box 2'\rangle\rangle$$

iff

- $\langle q2, \Box 2, \Box 2\rangle = \langle q2', \Box 2', \Box 2'\rangle$
- $\Box 1' = \Box 2' = \{\}$
- $\langle q1, \Box 1\rangle = \langle q1', \Box 1'\rangle$
- $\Box e \Box E \; \Box 1' = \Box 1 \Box e$

At any point in time, the environment may choose to insert some arbitrary event into the event queue associated with machine *M1*.

We finally define the transition relation $!_S$ associated with system *S*, or $!$ for short, as the union of the above transition relations:

$$!_S = !_{local\text{-}1} \Box !_{local\text{-}2} \Box !_{dispatch\text{-}1} \Box !_{dispatch\text{-}2} \Box !_{discard\text{-}1} \Box !_{discard\text{-}2} \Box !_{env\text{-}1} \Box !_{env\text{-}2}$$

Figure 1 shows an artificial example system to illustrate our approach, together with a possible computation sequence of this system. The example shows on the top the architecture of the system, consisting of components *M1* and *M2*, which communicate using the associations shown. The behaviour of the components is defined by the two state-machines depicted below *M1* and *M2*, respectively. If triggered to take a *move*, controller *M2* chooses arbitrarily the direction *d* of the move of the robot, and signals controller *M1* in charge of engine control the selected direction. It then checks its sensors for detecting possible obstacles; if the chosen direction is clear of obstacles, the interlock for moves is removed, by emitting the corresponding *clear* event to *M1*. *M2* initiates the step motor depending on the chosen direction, and awaits clearance of the interlock before emitting a *go*, causing the actual move. The *check* event generated by the environment models a cyclic scheduler, while the *move* command is intended to be issued by some operator, and is thus not synchronized to steps of the system.

Note. that states *1,2,3* of *M2* are *instable*, while all its other states are *stable*. Thus, an example of a *run-to-completion* step of *M2* from state *6* is the one which is trig-

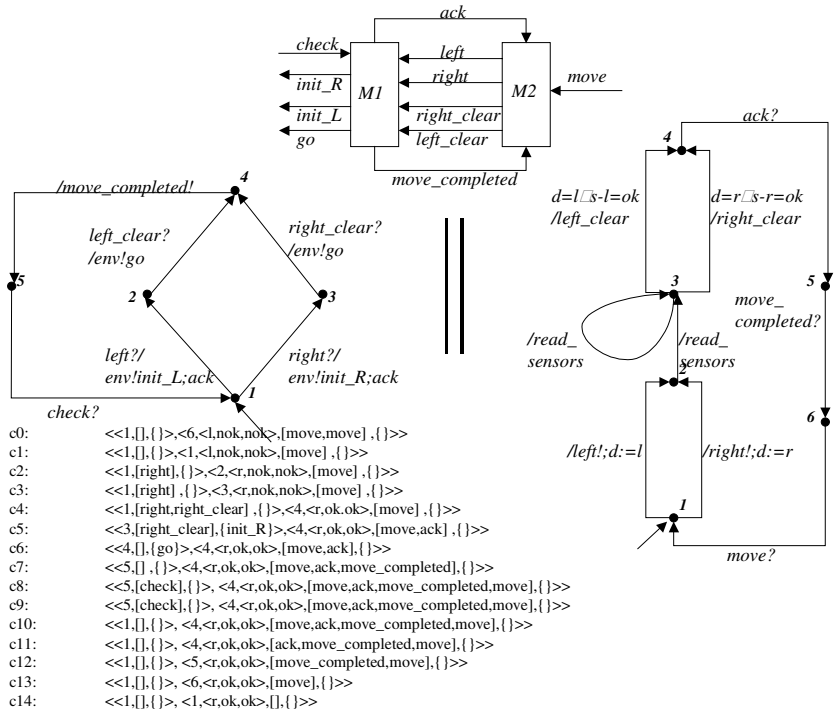
gered by consuming a *move* command, followed by the local processing steps passing through states *1,2,3,* possibly iterating in state *3* waiting for an obstacle to move, and finally passing to state *4*. The only *instable* state of *M1* is state *4*. The sample configuration sequence shows purely local computation steps (such as reading sensors in *c3! c4*), environment steps (such as emitting the spurious *move* command in *c7! c8*), discarding events (such as the older *move* command in *c10! c11*), and dispatching events (such as *ack* in *c11! c12).*

Configurations provide a fine grained view of a systems execution semantics. In the context of system design, we are typically only interested in a *grey box* view of the systems behaviour, where observability is restricted to message exchange between system components themselves, as well as between system components and the environment, such as typically captured by scenarios or message sequence charts. We define this view as follows. For a configuration sequence from an initial configuration

$$\sigma = c0 !_s c1 !_s \ldots !_s cn ,$$

label the *j*th transition $c(j-1) !_s cj$ by <*source, e , dest*> if it involves emission of event e from *source* to *dest,* where *source* and *dest* are either *M1, M2,* or *env.* Let *obs(*σ*)* be the sequence of such labels in σ. We then define the observational semantics *[[S]]* of *S* as the set of seuqences of labels associated with such configuration sequences:

$$[[S]] = \{\, obs(\sigma) \mid \sigma \text{ is a finite configuration sequences of } S \,\}$$




```
c0:   <<1,[],{}>,<6,<l,nok,nok>,[move,move] ,{}>>
c1:   <<1,[],{}>,<1,<l,nok,nok>,[move] ,{}>>
c2:   <<1,[right],{}>,<2,<r,nok,nok>,[move] ,{}>>
c3:   <<1,[right] ,{}>,<3,<r,nok,nok>,[move] ,{}>>
c4:   <<1,[right,right_clear] ,{}>,<4,<r,ok.ok>,[move] ,{}>>
c5:   <<3,[right_clear],{init_R}>,<4,<r,ok,ok>,[move,ack] ,{}>>
c6:   <<4,[],{go}>,<4,<r,ok,ok>,[move,ack],{}>>
c7:   <<5,[] ,{}>,<4,<r,ok,ok>,[move,ack,move_completed],{}>>
c8:   <<5,[check],{}>, <4,<r,ok,ok>,[move,ack,move_completed,move],{}>>
c9:   <<5,[check],{}>, <4,<r,ok,ok>,[move,ack,move_completed,move],{}>>
c10:  <<1,[],{}>, <4,<r,ok,ok>,[move,ack,move_completed,move],{}>>
c11:  <<1,[],{}>, <4,<r,ok,ok>,[ack,move_completed,move],{}>>
c12:  <<1,[],{}>, <5,<r,ok,ok>,[move_completed,move],{}>>
c13:  <<1,[],{}>, <6,<r,ok,ok>,[move],{}>>
c14:  <<1,[],{}>, <1,<r,ok,ok>,[],{}>>
```

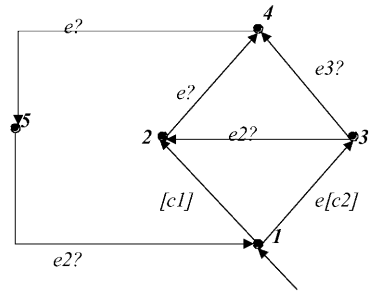
Fig. 1. Example system with robot *M1* and controller *M2* with possible computation sequence at bottom left.

3 Symbolic Evaluation of Events

The aim of this section is to define a finite state representation of a given system *S* which induces the same observational semantics as the standard representation of *S* as defined in the previous section. The central idea has already been elaborated in the introduction: rather than storing an event in the receivers queue, we directly evaluate the effect of emitting the event on the receivers state machine. Concretely, this effect is that transitions which previously required consumption of an event now become only locally guarded or unguarded, and can be performed without event dispatching. We refer to such transitions as *pending*.

We will introduce a simple colouring scheme to classify transitions in a state machine. Let us paint the current state *red*. Transitions that are reachable from the current state by taking only locally guarded or unguarded transitions are also *red*. A pending transition is painted *green* if it can be reached from the initial state, or from another green transition, by taking only locally guarded or unguarded transitions. All other transitions remain uncoloured. We must define rules for recolouring of transitions during computation steps. Intuitively, such rules will either extend the green region to represent reception of events into the event queue, or turn green regions red to represent dispatching of events from the event queue, while red regions turn uncoloured. Our construction thus eliminates unbounded FIFO buffers by the green-set – just another representation of the effect of emitting events.

There are a number of caveats to this intuitive idea, and we take the simple example of Figure 2 to illustrate these. The left box below the state-machine shows the sequence of events received by this state-machine in a possible computation sequence. The right box shows a suggestion for the set of transitions that should be coloured green, a transition being represented by its source and destination state.



<1,[]> ?
<1,[e]> ?
<1,[e,e2]> ?
<1,[e,e2,e]> ?
<1,[e,e2,e,e3]> ?
<1,[e,e2,e,e3,e]> ?

<1,{}>
<1,{<1,3>,<2,4>}>
<1,{<1,3>,<2,4>,<3,2>}>
<1,{<1,3>,<2,4>,<3,2>,<4,5>}>
<1,{<1,3>,<2,4>,<3,2>,<3,4>}>
<1,{< 1,3>,<2,4>,<3,2>,<3,4>,<4,5>}>

Fig. 2. Example system with sequence of received events at left, and possible green-set at right.

Initially, we start with an empty queue, represented symbolically by an empty green set. Now consider the effect of emitting event *e*, with state *1* painted red. Recall, that this colouring also extends to transition *<1,2>*. Hence, adjacent to the red region, we have two transitions willing to consume the emitted event, *<1,3>* and *<2,4>*, and we colour them green. Next, event *e2* becomes emitted – and we find the transition *<3,2>* adjacent to the green region, waiting to consume *e2*, and include it in the green set. The next occurrence of *e* makees *<4,5>* green. Next, consider the arrival of yet another event *e3*. With state *3* being green, we find a transition matching the emitted event, and include it in the green set.

There are two flaws with this construction, when striving for exactness. First, notice that by introducing transition *<3,2>* in the green set, the pragmatics of having transition *<2,4>* in the green set becomes ambiguous: is it because we have received messages corresponding to the "slow" path *<1,3>* *<3,2>* to state *2*, and that actually *<2,4>* represents the second occurrences of *e* being emitted, or is it because we have performed the local step *<1,2>* and are about to swallow the first occurrence of *e* ? This highlights the need to cater for non-determinism by keeping the green sets for the different alternatives *separated*. This separation can come for free if the choices are processed in state disjoint regions of the state-machineö otherwise it can be enforced artificially, by adding primed copies of a state in confusion situations such as above.

The second flaw can be observed when processing event *e3*. By now colouring transition *<3,4>* green, the symbolic representation can actually *choose,* whether to pick this transition, once state *3* becomes the current state, or whether to follow the already green *<3,2>*-transition. Thus our representation has lost ordering information – and consuming the wrong event could easily spoil the capability of imitating a real-computation sequence. We cater for this by never extending the green set with transitions, which are in conflict to an already taken global choice.

We now turn to defining our symbolic representation, in which some main ideas are to use copies of states to maintain separation of alternative computations, and to delete sets of "dead" transitions. Let $Q^{\square}$ denote the set Q together with an unbounded supply of numbered copies of *Q*. *T*he *k-th* copy of state *q* is denoted *q(k)*; we identify *q* with *q(0)*. For a given state *q* of machine *M* its *local post-set q** is defined to be the set of states reachable from *q* by taking only locally guarded or unguarded transitions. We extend this function to *copies* of state *q*. *Symbolic configurations* SC_s of our system *S* take the form

$$sc = << q1 , \square 1, \square 1, \square 1>, < q2 , \square 2 , \square 2, \square 2 >>$$

where $\square j \ \square \ Qj^{\square} \cdot G \cdot A \cdot Qj^{\square}$ denotes the current set of red and green transitions or copies thereof in machine *Mj*. The set $\square j$ satisfies the property that its locally guarded and unguarded transitions generate a graph, whose connected components form the nodes a tree, where the green transitions are arcs. The current state *qj* is in the root component of this tree. In other words, for all reachable states *q*, all paths from *qj* to *q* contain the same sequence of event-consuming transitions. Moreoever, the set $\square j$ has no unreachable states.

We now define a transition relation on symbolic configurations. The transition relation relies on two key operations *extend(e,*$\square j$*, qj)*, and *reduce(*$\square j$*,qj)* on the sets $\square j$ of red and green transitions. Intuitively, *extend* spreads the green colour from the current potential leaves of $\square j$ by evaluating symbolically the effect of consuming *e* in each of

these leaves. The function *reduce* acts like a garbage collector: it removes from □*j* all transitions which are no longer reachable once the actual state has progressed to *qj*. In an actual implementation, *reduce* should incorporate a reclaiming scheme of indices for state-copies.

Define a *potential leaf* of □*j* as a state-copy *q(k)* such that the disjunction of local guards of (local and non-local) transitions from *q(k)* is not *true*. The function *extend(e,□j,qj)*, takes as parameters the emitted event, the current set □*j* of red and green transitions, and the current state of *Mj*. It extends □*j* as follows.

- For each potential leaf *q(k)* in □*j* we add, for each transition <*q*,<*e*[□],*a*>,*q´*> from *q* triggered by *e,* a transition of form <*q(k)*,<*e*[□],*a*>,*q´(l)*> from *q(k)* to a copy of *q´* which is not previously in □*j*.
- If there are transitions from *q* labeled by other events than *e,* we add one transition of form <*q(k)*,<*e*[←□],>,*q(l)*> from *q(k)* to a fresh copy of itself, guarded by the negation ←□ of the disjunction □ of local guards of all *e*-consuming transitions that emanate at *q*. This is an explicit representation of a discard step, which must be added to avoid the order confusion observed in the previous example.

Thereafter □*j* is extended by adding appropriate copies of locally guarded and unguarded transitions that are reachable from the destinations of added green transitions.

The function *reduce(□j,qj´)* takes as parameters the current set □*j* of red and green transitions, and a new current state of *Mj*. It deletes from □*j* all transitions, which are not reachable from *qj´*.

The initial symbolic configuration starts with □*j* being the set of unguarded and locally guarded transitions that are reachable from the initial state of *Mj*.

Slocal-1

$$sc = \langle\langle q1,\Box 1,\Box 1,\Box 1\rangle, \langle q2,\Box 2,\Box 2,\Box 2\rangle\rangle \; ! \; s_{local\text{-}1}$$
$$sc' = \langle\langle q1',\Box 1',\Box 1',\Box 1'\rangle,\langle q2',\Box 2',\Box 2',\Box 2'\rangle\rangle$$

iff

- $\langle q2,\Box 2\rangle = \langle q2',\Box 2'\rangle$
- $\Box 2' = \{\}$
- $\Box(q1,\langle\Box,a\rangle,q1')\Box T1 \; \Box[[\Box]](\Box) = true$
  $(\Box j \Box\Box a \Box\Box j \; \Box\Box 1' = [[\Box j]](\Box 1) \; \Box\Box 1' = \Box\Box\Box 1' = \{\} \Box\Box 2' = \Box 2)$
  $\Box(\Box e \Box E \; a \Box e! \; \Box\Box 2' = extend(e,\Box 2,q2) \Box\Box 1' = \Box 1 \; \Box\Box 1' = \Box\Box\Box 1' = \{\})$
  $\Box(\Box e \Box E \; a \Box self!e \; \Box\Box 1' = extend(e,\Box,q1) \Box\Box 1' = \Box 1 \; \Box\Box 2' = \Box 2 \Box\Box 1' = \{\})$
  $\Box(\Box e \Box E \; a \Box env!e \; \Box\Box 2' = \Box 2 \; \Box\Box 1' = \Box\Box\Box 1' = \Box 1 \; \Box\Box 1' = \{e\})$

  where

  $\Box = reduce(\Box 1,q1')$

There are four sub-cases to consider, corresponding to the four disjuncts in the definition of local computation steps. The first disjunct does not involve event-processing and hence leaves both queues unchanged. However, the green-set □*1* of *M1* may contain alternatve sub+trees that are no longer reachable after the transition to *q1´,* hence the function *reduce(□1,q1´)* should be applied first. In the third case, we note that emitting the event to itself is handled by setting

□1 to *extend(e, □, q1),* and that in the final case of emitting an event to the environment the green set remains unchanged.

Green-1

$$sc = \langle\langle q1,\square 1,\square 1,\square 1\rangle, \langle q2,\square 2,\square 2,\square 2\rangle\rangle \;!_{green\text{-}1}$$
$$sc' = \langle\langle q1',\square 1',\square 1',\square 1'\rangle,\langle q2',\square 2',\square 2',\square 2'\rangle\rangle$$

iff

- $\langle q2,\square 2\rangle = \langle q2',\square 2'\rangle$
- $\square 2' = \{\}$
- *stable(q1,*□*1)*
- $\square(q1,\langle e?[\square],a\rangle,q1'(r)) \square\square 1 \;\; [[\square]](\square) = true \;\square$
  $(\square j \square\square\; a \square\square j \;\square\; \square 1' = [[\square j]](\square 1) \;\square\; \square 1' = \square \;\square\square 1' = \{\} \square\; \square 2' = \square 2)$
  $\square(\square e \square E\; a \square e!\; \square\; \square 2' = extend(e,\square 2,q2) \square\; \square 1' = \square 1\; \square\; \square 1' = \square\square\square 1' = \{\})$
  $\square(\square e \square E\; a \square self!e\; \square\; \square 1' = extend(e,\square,q1) \square\; \square 1' = \square 1\; \square\; \square 2' = \square 2 \square\square 1' = \{\})$
  $\square(\square e \square E\; a \square env!e\; \square\; \square 2' = \square 2\; \square\; \square 1' = \square\square\; \square 1' = \square 1\; \square\; \square 1' = \{e\})$

where

$\square = reduce(\square 1, q1')$

Green-steps intuitively correspond to dispatch or discard steps and thus may only be taken if there is no locally enabled transition leaving the current state *q1*. They move the current state forward along one of the adjacent green transitions, thus possibly also resolving choices. All other concepts have been elaborated before.

Senv-1

$$sc = \langle\langle q1,\square 1,\square 1,\square 1\rangle, \langle q2,\square 2,\square 2,\square 2\rangle\rangle \;!_{senv\text{-}1}$$
$$sc' = \langle\langle q1',\square 1',\square 1',\square 1'\rangle,\langle q2',\square 2',\square 2',\square 2'\rangle\rangle$$

iff

- $\langle q2,\square 2,\square 2\rangle = \langle q2',\square 2',\square 2'\rangle$
- $\square 1' = \square 2' = \{\}$
- $\langle q1,\square 1\rangle = \langle q1',\square 1'\rangle$
- $\square e \square E\; \square 1' = extend(e,\square 1,q1)$

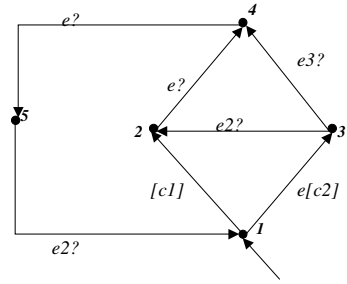
As before, the environment can choose at any time to emit some event *e*, which in the symbolic computation model is then evaluated by updating the green-set.

We finally define the symbolic transition relation $!_{ss}$ associated with system *S*, as the union of the above transition relations:

$$!_{ss} = \; !_{slcal\text{-}1} \square\; !_{slocal\text{-}2} \square\; !_{green\text{-}1} \square\; !_{green\text{-}2} \square\; !_{senv\text{-}1} \square\; !_{senv\text{-}2}$$

Figure 3 shows how the symbolic transition relation operates, and in particular how the problems highlighed for the system in Figure 2 are handled. As before, we represent emitting *e* by including *<1,3>* and *<2,4>* in the green set. The subsequent emission of *e2* would - as noted before – cause a duplication of state *2* in the green set, hence we use a primed copy of this state when painting transition *<3,2>*. This allows us to nicely separate the two alternatives when processing the subsequent emittance of event *e*: the left hand choice leads to the inclusion of *<4,5>* , which has no bearing on the computation path following the other alternative to the copy *2´* of state *2*, which in its turn can now move forward to a fresh copy of state *4*, by including *<2´,4´>* in the

green set. The subsequent emittance of *e3* is ignored: the global choice between the two transitions originating from state *3* has already been decided in favour of *<3,2>*. The subsequent emittance of event *e* causes a copy *<4´,5´>* of transition *<4,5>* to be included in the green set.



<1,[]> !	<1,{}>	
<1,[e]> !	<1,{<1,3>,<2,4>}>	
<1,[e,e2]> !	<1,{<1,3>, <3,2´>, <2,4>}>	made copy of state 2
<1,[e,e2,e]> !	<1,{<1,3>, <3,2´>,<2´,4´>,<2,4>,<4,5>}>	made copy of state 4
<1,[e,e2,e,e3]> !	<1,{<1,3>, <3,2´>,<2´,4´>,<2,4>,<4,5>}>	discarded e3 – no matching front state
<1,[e,e2,e,e3,e]> !	<1,{<1,3>, <3,2´>,<2´,4´>,<4´,5´>,<2,4>,<4,5>}>	made copy of state 5
<2,[e,e2,e,e3,e]> !	<2,{<2,4>,<4,5>}>	deleted unselected subtree
<4,[e2,e,e3,e]> !	<4,{<4,5>}>	
<4,[e2,e,e3,e,e2]> !	<4,{<4,5>,<5,1>}>	
<4,[e2,e,e3,e,e2,e]> !	<4,{<4,5>,<5,1>, <2,4´>,<1,3>}>	created copy of state 4 – **overrun of state 4**
<4,[e,e3,e,e2,e]>!	<4,{<4,5>,<5,1>, <2,4´>,<1,3>}>	discard steps have already been catered for
<5,[e3,e,e2,e]> !	<5,{<5,1>, <2,4>,<1,3>}>	reduce prime count on state 4
<5,[e3,e,e2,e,e2]> !	<5,{<5,1>, <2,4>,<1,3>,<3,2´>}>	created copy of state 2
<5,[e3,e,e2,e,e2,e]> !	<5,{<5,1>, <2,4>,<4,5´>,<1,3>,<3,2´>,<2´,4´>}>	created copies of 5 and 4 – **overrun of state 5**
<5,[e,e2,e,e2,e]> !	<5,{<5,1>, <2,4>,<4,5´>,<1,3>,<3,2´>,<2´,4´>}>	discard steps have already been catered for
<5,[e2,e,e2,e]> !	<5,{<5,1>, <2,4>,<4,5´>,<1,3>,<3,2´>,<2´,4´>}>	discard steps have already been catered for
<1,[e,e2,e]> !	<1,{<2,4>,<4,5>,<1,3>,<3,2´>,<2´,4´>}>	reduce prime count for state 5
<3,[e2,e]> !	<3,{<3,2>,<2,4>}>	eliminate isolated components – reduce prime count for 2 and 4
<2,[e]> !	<2,{<2,4>}>	
<4,[]> !	<4,{}>	

Fig. 3. Symbolic transition sequence of the example system of Figure 2.

We now mimic a state configuration resolving the initial choice of transitions originating from state *1* in favour of transition *<1,2>*. In general, configuration steps advancing the current state are modelled by deleting those components from the green set no longer reachable from the current state, thus the complete sub-tree *{<1,3>,<3,2´>,<2´,4´>,<4´,5´>}* induced by the alternate choice is deleted. By the same policy, just advancing the current state will cause the taken transition to be deleted from the green set, as shown in the subsequent computation step reaching state *4*. This artificial example is too loosely synchronised to avoid *state-overrun* – the partner machine being able to now emit events *e2* and *e* causes a state overrun for state *4*: if including *<2,4,>* in the green-set, we could no longer separate behaviour induced by the current and the subsequent visit of this state. By using a fresh copy of state *4*, we avoid this problem – at the price of loosing boundedness of the green-set.

For a sequence of symbolic configurations

$$s\square = sc0\ !_{ss}\ sc1\ !_{ss}\ \ldots\ !_{ss}\ scn$$

where s*c0* is an initial symbolic configuration, label the *j*th transition $cj\text{-}1\ !_{s}\ cj$ by *<source, e , dest>* if it involves emission of event e from *source* to *dest,* Let *obs(s□)* be the

sequence of such labels in s□. The key property of our construction is the preservation of the observational semantics.

Theorem 1

[[S]] = {obs(s□) | s□ is a finite sequence of symbolic configurations of *S }*

Proof:

□ In this direction, the theorem states that *the symbolic semantics is a safe approximation of the possible computation sequences of each protocol machine.* The theorem follows by composition if for each protocol machine we prove that each computation sequence in the concrete semantics corresponds to a computation sequence in the symbolic semantics, with the same external behaviour. Let us specialize the notation to machine *M1*. We establish a simulation relation between configurations <*q1,□1,□1,□1*> in the concrete semantics, and symbolic configurations <*q1,□1,□1,□1*> in the symbolic semantics. The simulation relation identifies the state *q1*, valuation *□1,* and set of emitted events *□1*. A (concrete) configuration <*q1,□1,□1,□1*> is simulated by a symbolic configuration <*q1,□1,□1,□1*> under the following condition:

> **if** *M1* can perform a sequence □of transitions from *q1* to a state *qf* while consuming the events *□1* in dispatch and discard transitions, **then** the sequence □ (possibly with states replaced by their appropriate copies) can be performed in *□1* from *q1* to a potential leaf which is (possibly a copy of) *qf*.

In order to prove that this simulation relation is preserved by the transitions, we note that the definition of the concrete and symbolic semantics differ only on the following points:

1. Each extension of the input queue (*□1´= □1□e*) correspond to an application of the function *extend(e,□1,q1)*.
2. Each change of control state, from *q1* to *q1´* say, corresponds to deleting the corresponding root of *□1* and resizing it with respect to *q1´* using *reduce(□1,q1´(r))*. If the transition consumes an element of the input queue in a dispatch or discard transition, then that element is removed from the input queue (*□1´ = tail(□1)*) .

We should thus check that

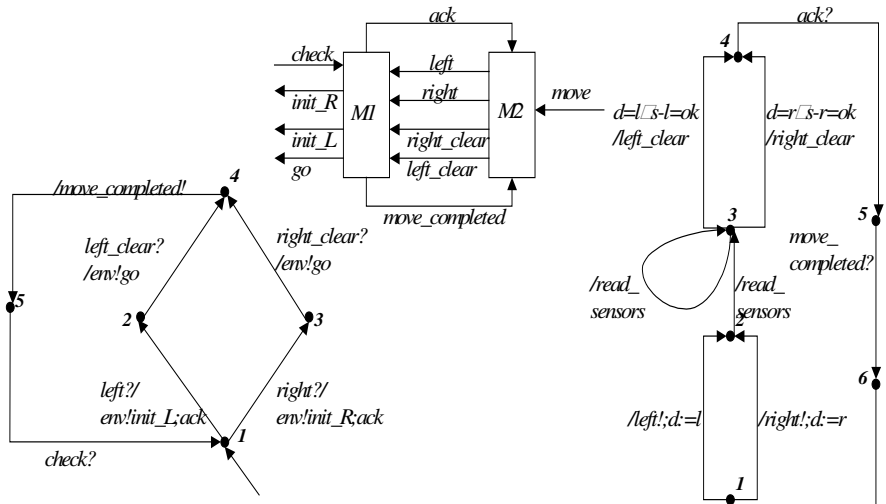
1. the function *extend(e,□1,q1)* extends *□1* by adding paths that correspond to the paths that consume *e* starting from a potential leaf of *□1*;
2. shrinking of *□1* by moving to a child of the previous root corresponds exactly to following the first step of paths that start with this transition.

Both properties should be clear by construction.

□ We prove this direction by establishing, that the "inverse" of the preceding simulation relation,

> Each path in *□1* from *q1* to a potential leaf *qf(k)* corresponds to a sequence of transitions in *M1* from *q1* to *qf,* possibly with discard transitions added, that consume the events *□1* in dispatch and discard transitions

is also a simulation relation in the other direction. This follows form the fact that each state in $\Box 1$ can be reached from the root by a *unique* sequence of consumed events from the input buffer, and that $\Box 1$ contains no "spurious" potential leaves.



c0: <<1,[],{}>,<6,<1,nok,nok>,[move,move] ,{}>>
c1: <<1,[],{}>,<1,<1,nok,nok>,[move] ,{}>>
c2: <<1,[right],{}>,<2,<r,nok,nok>,[move] ,{}>>
c3: <<1,[right] ,{}>,<3,<r,ok,ok>,[move] ,{}>>
c4: <<1,[right,right_clear] ,{}>,<4,<r,ok,ok>,[move] ,{}>>
c5: <<3,[right_clear],{init_R}>,<4,<r,ok,ok>,[move,ack] ,{}>>
c6: <<4,[],{go}>,<4,<r,ok,ok>,[move,ack],{}>>
c7: <<5,[] ,{}>,<4,<r,ok,ok>,[move,ack,move_c],{}>>
c8: <<5,[check],{}>, <4,<r,ok,ok>,[move,ack,move_c],{}>>
c9: <<5,[check],{}>, <4,<r,ok,ok>,[move,ack,move_c,move],{}>>
c10: <<1,[],{}>, <4,<r,ok,ok>,[move,ack,move_c,move],{}>>
c11: <<1,[],{}>, <4,<r,ok,ok>,[ack,move_c,move],{}>>
c12: <<1,[],{}>, <5,<r,ok,ok>,[move_c,move],{}>>
c13: <<1,[],{}>, <6,<r,ok,ok>,[move],{}>>
c14: <<1,[],{}>, <1,<r,ok,ok>,[],{}>>

sc0: <<1,{},{}>,<6,<1,nok,nok>,{<6,1>},{}>>
sc1: <<1,{},{}>,<1,<1,nok,nok>,{},{}>>
sc2: <<1,{<1,3>},{}>,<2,<r,nok,nok>,{},{}>>
sc3: <<1,{<1,3>},{}>,<3,<r,ok,ok>,{},{}>>
sc4: <<1,{<1,3>,<3,4>},{},<4,<r,ok,ok>,{},{}>>
sc5: <<3,{<3,4>},{init_R}>,<4,<r,ok,ok>,{<4,5>},{}>>
sc6: <<4,{},{go}>,<4,<r,ok,ok>,{<4,5>},{}>>
sc7: <<5,{},{}>,<4,<r,ok,ok>,{<4,5>,<5,6>},{}>>
sc8: <<5,{<5,1>},{}>,<4,<r,ok,ok>,{<4,5>,<5,6>},{}>>
sc9: <<5,{<5,1>},{}>,<4,<r,ok,ok>,{<4,5>,<5,6>,<6,7>},{}>>
sc10:<<1,{},{}>,<4,<r,ok,ok>,{<4,5>,<5,6>,<6,7>},{}>>
sc11:<<1,{},{}>,<5,<r,ok,ok>,{<5,6>,<6,7>},{}>>
sc12:<<1,{},{}>,<6,<r,ok,ok>,{<6,7>},{}>>
sc13:<<1,{},{}>,<1,<r,ok,ok>,{},{}>>

Fig. 4. Symbolic representation (right) of corresponding computation sequence (left).

In Figure 4, we illustrate the practical value of the construction using the example of the previous section.The symbolic configuration sequence is paired up with the original computation sequence. The example exhibits a number of properties typical for well structured protocols: separation of alternatives comes for free, and the protocol is free from state overrun. As a consequence, the symbolic representation is finite state.

We conclude this section by proposing an approach to selectively over-approximate the set of observations. The idea is that in the case that our symbolic construction grows unboundedly, by potentially creating an unbounded number of copies of a given state q, we will allow the symbolic representation to blur the distinction between different copies of state q. We may then over-approximate the set of possible behaviours, including behaviours which are not feasible in S, but we can guarantee finiteness of the resulting symbolic transition system. As an example, this

technique can be used to deal with "inessential loops", in which a state machine receives copies of events without being dependent on the number of copies received. In general, by allowing the user to over-approximate the behaviour for a selected set of states, the user can *tune* the trade-off between succinctness and exactness of the symbolic representation.

We now describe how to modify the symbolic representation when states in the subset *Rj* are not copied in the symbolic transition rules. A first effect is that the set of *potential leaves* can no longer be uniquely deduced from the green-set □*j*. In order to increase precision, we therefore extend our symbolic representation by explicitly including a set of *potential leaves*. Formally, r*elaxed symbolic configurations* of our system *S* take the form

$$rsc = << q1 , \square 1, \square 1, F1, \square 1>, < q2 , \square 2 , \square 2, F2, \square 2 >>$$

where *Fj* □ *Qj* denotes the current set of *potential leaves* of machine *Mj.* The effect of an event emission is, as before, computed using the function *extend(e,*□*j,Fj,qj),* which now takes the set of potential leaves as an explicit argument, but otherwise is defined as previously. After following a transition $<q,<e[\square],a>,q'>$, the set *Fj* is updated. As before, it will contain state or state-copies, for which the disjunction of local guards of outgoing transitions is not *true*, but additionally it will contain states (or state copies) which are reachable from *q´* by local transitions and from which the disjunction of guards of local transitions is not *true.* The rationale for including these extra states is that they may correspond to revisits of relaxed states, and therefore outgoing transitions that consume events should be ignored when determining whether they can now be potential leaves. To illustrate the modified construction, in Figure 4 we revisit the artificial example of Figure 2, assuming states *2,4* and *5* to be relaxed. The set of potential leaves is listed as third component of the symbolic state.

We conclude this section by stating that the relaxed symbolic representation yields a safe approximation of the queue-based semantics.

Theorem 2

Let R1, R2 be sets of relaxed sates of machines *M1, M2*, respectively. Then

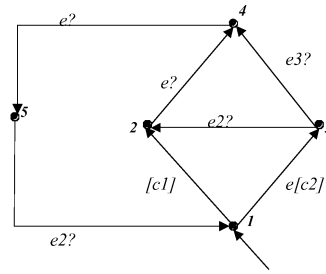
[[S]] □ *{obs(rs*□*)* | *rs*□ is a finite sequence of symbolic configurations of *S }*

Proof

The construction in the first half of the proof of Theorem 1 can still be carried out without modification

4 About Exact Finite Symbolic Representations

In the symbolic analysis method outlined in the previous section, we may need to designate certain states as relaxed in order to keep the symbolic state space finite. At the same time, relaxed states induce an over-approximation of the state-space. In this section, we will give conditions under certain combinations of exact and relaxed states yield a finite and exact analysis of (properties of) a protocol. A central idea is to identify a core *skeleton* of the protocol in each machine, such that the synchronization between the machines is "sufficiently tight" so that the symbolic analysis is faithful, at least with respect to this skeleton.



Relaxed states:
2,4,5

<1,[]> ?	<1,{},{}>	
<1,[e]> ?	<1,{<1,3>,<2,4>},{3,4}>	states 3 and 4 green
<1,[e,e2]> ?	<1,{<1,3>, <3,2>, <2,4>}, {2,4}>	no copy of state 2, frontier passes from 3 to 2
<1,[e,e2,e]> ?	<1,{<1,3>, <3,2>,<2,4>,<4,5>},{4,5} >	no copy of state 4, two possible consumptions for e move frontier from 2 to 4 and 4 to 5
<1,[e,e2,e,e3]> ?	<1,{<1,3>, <3,2>,<2,4>,<4,5>},{4,5}>	e3 discarded at both green states
<1,[e,e2,e,e3,e]> ?	<1,{<1,3>, <3,2>,<2,4>,<4,5>},{5}>	consuming e at green state 4 moves front to 5
<2,[e,e2,e,e3,e]> ?	<2,{<2,4>,<4,5>}, {5}>	deleted unselected subtree
<4,[e2,e,e3,e]> ?	<4,{<4,5>}, {5}>	deleted taken transition
<4,[e2,e,e3,e,e2]> ?	<4,{<4,5>,<5,1>},{1}>	front moves from 5 to 1
<4,[e2,e,e3,e,e2,e]> ?	<4,{<4,5>,<5,1>, <2,4>,<1,3>}, {3,4}>	no copy of state 4 – **overrun of state 4**
<4,[e,e3,e,e2,e]>?	<4,{<4,5>,<5,1>, <2,4>,<1,3>},{3,4}>	discard steps have already been catered for
<5,[e3,e,e2,e]> ?	<5,{<5,1>, <2,4>,<1,3>},{3,4}>	deleted taken transition
<5,[e3,e,e2,e,e2]> ?	<5,{<5,1>, <2,4>,<1,3>,<3,2>},{2,4}>	extended green set at green state 3, no copy of sta
<5,[e3,e,e2,e,e2,e]> ?	<5,{<5,1>, <2,4>,<4,5>,<1,3>,<3,2>},{4,5}>	no copies of 5 and 4 – **overrun of state 5**
<5,[e,e2,e,e2,e]> ?	<5,{<5,1>, <2,4>,<4,5>,<1,3>,<3,2>},{4,5}>	discard steps have already been catered for
<5,[e2,e,e2,e]> ?	<5,{<5,1>, <2,4>,<4,5>,<1,3>,<3,2>},{4,5}>	discard steps have already been catered for
<1,[e,e2,e]> ?	<1,{<2,4>,<4,5>,<1,3>,<3,2>},{4,5}>	delete taken transition
<3,[e2,e]> ?	<3,{<3,2>,<2,4>,<4,5>},{4,5}>	eliminate isolated components
<2,[e]> ?	<2,{<2,4>,<4,5>},{4,5}>	delete taken transition
<4,[]> ?	<4,{<4,5>},{5}>	eliminate isolated green state

Fig. 5. The artificial example of Figure 2 revisited, assuming states *2,4,* and *5* to be relaxed.

As inspiration, we observe that in many protocols, it is possible to identify a certain subset of "important" control states in each protocol machine. These states represent stable situations, associated with certain functionality of the protocol, and typically represent knowledge about the global system state. We use the term *modes* to denote such "important" states of a machine. Furthermore, the occupancy of modes is typically synchronized between the machines, i.e., if at some point in time machine *M1* occupies a certain mode, then *M2* occupies a unique corresponding mode, except possibly during a mode-change. Mode-changes are then synchronized between protocol machines, typically by exchanging a pair of *request* and *grant* messages.

We consider, as previously, a system *S* consisting of protocol machines *M1* and *M2*. We will in addition assume that the message alphabet *E* is finite, and that there are no local actions or guards, i.e., transitions are labeled only by event emissions and consumptions. Moreover, we forbid any reception of events from the environment (i.e., there are no computation steps generated by rule *Env-i*). Let *D* be a subset of the messages exchanged between M1 and M2 (the intuition is that they should include request and grant messages used to synchronize mode changes). Let a *D-emission* be a transition which emits an event in *D*. Let a *D-reception* be a transition which consumes an event in *D*. Let a *D-transition* be a *D*-emission or a *D*-reception. The terms *non-D-emission, non-D-reception,* and *non-D-transition* are defined analogously.

Definition We say that a state machine *Mj* is *mode-separated* by a set *D* of messages if along any sequence of transitions from the initial state, the number of *D*-emissions is at most one more than the number of *D*-receptions.

The intuition is that the potentially extra *D*.emession is a request for a mode change, which will be acknowledged by a grant message in the other direction.

In our first result, Theorem 3, we assume all states to be exact, implying that the symbolic semantics coincides with the actual semantics (by Theorem 1), and define conditions under which the symbolic semantics is finite-state.

Theorem 3

Let the protocol machines M1 and M2 be mode-separated by the set D of messages. If each loop in any of the protocol machines M1 and M2 contains a D-transition, then there is a finite number of distinct reachable symbolic configurations.

Proof

The proof relies on the observation that, due to mode-separation, any green-set in a symbolic configuration may not contain more than two subsequent *D*-receptions. By the condition in the theorem, this restriction gives a uniform bound on the size of any symbolic configuration.

The conditions in Theorem 3 are quite restrictive. They essentially entail that there is a uniform bound on the number of messages in the channels, implying that the entire protocol is finite-state. We will therefore, in Theorem 4, introduce a less restrictive condition, which allows control loops involving emission and reception of messages not in *D*, e.g., for the purpose of retransmission. To obtain a finite symbolic semantics, we must designate the control states in such loops as relaxed, since they otherwise run the risk of being copied unboundedly. This relaxation may induce an over-approximation of the behavior; hence we will state conditions under which such an over-approximation does not affect the core protocol, as defined by its *D*-transitions.

Definition Say that a protocol machine *M* is *D-robust* if whenever *(q, e?,q')* is a non-*D*-reception in *M*, then

1. there is no path from *q* to *q'* which contains a *D*-transition,
2. there is a locally enabled path from *q'* to *q* in *M* without *D*-transitions
3. any path from *q'* reaches *q* without containing any *D*-transitions, using only locally enabled transitions.

Let us turn to the main result of this section. Let us lift the restriction that all states be exact, and assume that some partitioning of states into exact and relaxed states is performed.

Theorem 4

Let D1 and D2 be sets of events with D1 □ D2. Let S be a system of two communicating D2-robust protocol machines M1 and M2., both mode-separated by D1,such that in both M1 and M2:

(a) each loop containing an exact state also contains a D1-transition,
(b) whenever there are two different paths from some state q to a relaxed state q', at least one of which contains a D2-transition, then one of the paths must contain at least 3 D1-receptions.

Then there is a finite number of distinct reachable symbolic configurations. Moreoever, the symbolic semantics generates the same sequences of D2-transitions as the exact semantics.

Proof

By induction over computation sequences of *S*, we prove that the potential over-approximation given by the green-sets is limited to loops that start by a non-*D2*-reception, and contain only non-*D2*-transitions. The proof of that depends on the following observations.

1. Any path in a green-set may contain at most two D1-receptions. Hence, the symbolic representation may not contain two different paths to a relaxed state, except for loops of non-*D2*-transitions.
2. Condition (a) implies that there is a bound on the number of exact states in any path in the induced symbolic representation, hence (as in Theorem 3) the symbolic semantics is finite.

5 Conclusions

We have considered the analysis of protocols, consisting of asynchronously communicating state+mahcines, as arising in UML based models of distributed real time systems. We have proposed a symbolic semantics, which replaces explicit representation of buffer contents by a representation of their effects on the receiving agent. The representation can be tuned to give a balance between succinctness and exactness. For certain classes of protocols, the representation is guaranteed to be both finite-state and exact.. Automatic verification of protocols governing coordination of autonomous systems can therefore be performed by constructing a finite state representation of the system using the techniques of this paper. A potential application of the ideas of the paper could be in code generation for multiple asynchronously communicating tasks allocated on the same processor, replacing event based communication by shared memory communication.

References

1. Euro-Interlocking Requirements, September 2001, see www.eurolock.org
2. W. Damm and J. Klose. Verification of a Radio-based Signaling System Using the Statemate Verification Environment. Formal Methods in System Design, Vol 19(2), 2001.
3. D. Harel and E. Gery. *Executable Object Modelling with Statecharts*. IEEE Computer, July 1997, 31-42
4. D. Brand and P. Zafiropulo. *On communicating finite-state machines.* Journal of the ACM, 2(5):323--342, April 1983.
5. A.P. Sistla and L.D. Zuck. *Automatic temporal verification of buffer systems.* in Larsen and Skou, editors, Proc.3rd Int. Conf. on Computer Aided Verification, volume 575 of Lecture Notes in Computer Science, Springer Verlag, 1991.
6. M.G. Gouda, E.M. Gurari, T.-H. Lai, and L.E. Rosier. *On deadlock detection in systems of communicating finite state machines*. Computers and Artificial Intelligence, 6(3):209-228, 1987.

7. J.K. Pachl. *Protocol description and analysis based on a state transition model with channel expressions.* In Protocol Specification, Testing, and Verification VII, May 1987.
8. P. A. Abdulla, A. Bouajjani, and B. Jonsson. *On-the-fly analysis of systems with unbounded, lossy FIFO channels*. In Proc. 10th Int. Conf. on Computer Aided Verification, volume 1427 of Lecture Notes in Computer Science, pages 305-318, 1998.
9. W. Peng and S. Purushothaman. *Data flow analysis of communicating finite state machines.* ACM Trans. on Progr. Languages and Systems 13(3):399-442, July 1991.
10. 10. B. Boigelot and P. Godefroid. *Symbolic verification of communication protocols with infinite statespaces using QDDs*. In Alur and Henzinger, editors, Proc. 8th Int. Conf. on Computer Aided Verification, volume 1102 of Lecture Notes in Computer Science, pages 1-12. Springer Verlag, 1996.
11. B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. *The power of QDDs*. In Proc. 4th Int. Static Analysis Symposium, Lecture Notes in Computer Science. Springer Verlag, 1997.
12. A. Bouajjani and P. Habermehl. *Symbolic reachability analysis of FIFO channel systems with nonregular sets of configurations*. in Proc. ICALP 97, volume 1256 of Lecture Notes in Computer Science, 1997.
13. 13.P.A. Abdulla and B. Jonsson. *Channel representations in protocol verification*. in Proc. CONCUR 2001, 12th int. Conf. on Concurrency Theory, volume 2154 of Lecture Notes in Computer Science, pages 1-15, 2001.
14. B. Westphal *Exploiting Object Symmetry in Verification of UML-Designs*, Diplomarbeit, Carl von Ossietzky Universität, 2001
15. W. Damm and A. Pnueli. *An Active Object Model*. Technical Report, OFFIS, Oldenburg, FRG, 2002.