THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Lower Kent Ridge Road, Singapore 119260

TRA9/05

Interacting Process Classes

Ankit GOEL, Meng SUN, Abhik ROYCHOUDHURY and P. S. THIAGARAJAN

September 2005

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

Interacting Process Classes

Ankit Goel
ankitgoe@comp.nus.edu.sg

Sun Meng
sunm@comp.nus.edu.sg

Abhik Roychoudhury
abhik@comp.nus.edu.sg

P.S. Thiagarajan
thiagu@comp.nus.edu.sg

School of Computing
National University of Singapore
Singapore

Abstract

Many reactive control systems consist of a large number of similar interacting objects; these objects can be often grouped into classes. Such interacting process classes appear in telecommunication, transportation and avionics domains. In this paper, we propose a modeling and simulation technique for interacting process classes. Our modeling style uses well-known UML notations to capture behavior. In particular, the control flow of a process class is captured by a state diagram, unit interactions between process objects by sequence diagrams and the structural relations are captured via class diagrams. The key feature of our approach is that our simulation is *symbolic*. We dynamically group together objects of the same class based on their past behavior. This leads to a simulation strategy that is both time and memory efficient and we demonstrate this on well-studied non-trivial examples of reactive systems. We also use our simulator for debugging realistic designs such as NASA's CTAS weather monitoring system.

1 Introduction

System level design based on UML notations is a possible route for pushing up the abstraction level when designing reactive embedded systems. For this approach to be viable, such design flows must include simulation and (platform dependent) code generation tools. Here we focus on developing an efficient simulation technique for reactive systems specified as interacting classes of active objects.

Interacting process classes arise naturally in application domains such as telecommunications and avionics. An obvious way to define an execution semantics of a language of interacting process classes is to maintain the local state of each object as the simulation proceeds. This will however lead to an impractical blow-up in realistic system designs; consider for instance, a telephone switch network with millions of phones, an air traffic controller with hundreds of clients etc. To address this issue, we develop a *symbolic* execution methodology. Specifically, we do not maintain/access the name space of objects during simulation. Instead, we group together objects dynamically mainly by keeping track of their past histories.

We use state and class diagrams as a basis for our modeling framework – class diagrams are used to capture the associations between the process classes and state diagrams are used to describe the behavior of process classes. One unconventional feature in our modeling framework is that the unit of interaction is chosen to be not just a synchronization or send-receive event pairs. Instead, we use a sequence diagram as a basic communication unit. We note that at the model level, even primitive interactions between process classes often involve bidirectional information flow and are best depicted as short -acyclic- protocols. Thus from our experience in reactive system modeling (including the examples discussed in this paper), descriptions of process interactions based on sequence diagrams is very natural. Finally, we also introduce static and dynamic associations between objects. This is necessary when classes of active objects interact with each other. Static associations are needed to specify constraints imposed by the structure of the system. For instance, the topology of a network may demand that a node can take part in a "transmit" transaction only with its

neighbors. We use class diagrams in a standard way to specify such associations. On the other hand, *dynamic associations* are needed to instantiate the proper combinations of objects -based on past history- to take part in a transaction. For instance when choosing a send-receive pair of objects to take part in a "disconnect"' transaction we must choose a pair which are currently in the "connected" relation. This relation has presumably arisen by virtue of the fact that they took part last in a "connect" transaction.

All these features of our model demand a good deal of work in terms of defining an execution semantics. Furthermore, developing a *symbolic* execution semantics for process classes where we must also *symbolically maintain static and dynamic class associations* is even harder. In this paper we develop such a symbolic execution mechanism for interacting process classes.

In summary, the highlights of our work are: (a) a symbolic execution semantics which dynamically groups objects of a process class based on behavior, (b) systematic use of sequence diagrams to specify behavioral interactions between interacting process classes, and (c) investigating the efficiency of our symbolic simulation and its use in debugging with the help of realistic examples of reactive controllers. In terms of future work, we are looking into code generation as well as (symbolic) test-bench generation from our models.

Outline We shall develop our material in two phases. As a first step we present the core modeling language and its execution semantics without involving class diagrams and associations (static and dynamic). We shall then introduce these additional features and explain how the simulation mechanism is correspondingly extended before presenting the experimental results.

2 Related Work

In this section, we survey the literature on behavioral simulation of reactive object-based systems.

Our previous work [14] provides a modeling language based on more elaborate inter-object based description of interactions. However it does not consider process *classes*. The work on Live Sequence Charts [3, 7] presents an sequence diagram based inter-object modeling framework for reactive systems. However, this approach does not exploit symbolic execution of process classes. Though process classes are *specified* symbolically, they are instantiated to concrete objects during play-out or simulation. The approach taken in [18] alleviates this problem by maintaining constraints on process identities but falls short of a fully symbolic execution without process identities.

There are a number of design methodologies based on the UML notions of class and state diagrams as exemplified in the tools Rhapsody and RoseRT. These tools also have limited kinds of code generation facilities. Again, no symbolic execution semantics is provided and the interactions between the objects -not classes- have to be specified at a fairly low level of granularity. The new standard UML 2.0 advocates the use of "structured classes" where interaction relationships between the sub-classes can be captured via entities such as ports/interfaces; protocol state machines can be used to specify the allowed access patterns of an interface. Our present framework does not cater for structured classes but it can easily accommodate intra-object based interaction no-

tions such as ports, interfaces and protocol state machines. In this context, we emphasize that our symbolic execution mechanism is *not tied to any particular modeling style for describing process interactions.*

Turning to the works on behavioral subtyping, there exists a rich literature on substitutability of objects of one class by objects of another class. One of the early works in this area is by Liskov and Wing [11]; behavioral subtyping here involves establishing relations between the pre/post conditions of methods of the superclass and subclass. The focus is more on passive objects – objects without a control of their own (i.e. the state change of an object is by invocation of methods by other objects). Subsequently, behavioral subtyping of active objects have been studied in many works, [12, 6, 19] to name a few. These works mostly exploit behavioral inclusion notions from process algebra (trace containment, simulation relations, failure inclusion etc) to define notions of behavioral subtyping. *Unlike the works on behavioral subtyping, our focus is not to detect/establish subclass relationship among classes based on behavior.* Instead we dynamically group together objects of the same class based on behavior exhibited so far.

Our method of grouping together active objects based on behavior bears some relationship to abstraction schemes developed for grouping processes in parameterized systems (e.g. see [13, 4]). In particular, for parameterized systems with many similar processes (whose behavior can be captured by a single FSM or EFSM[1]), it is customary to maintain the count of number of processes in each state of the FSM/EFSM. Thus, the names/identities of the individual processes are not maintained in the state space representation. However, our symbolic simulation of active objects is complicated by the presence of inter-object associations. In other words, *we need to maintain associations between objects without referring to their names.* Such an issue does not arise in parameterized system validation.

The notion of protocols and "roles" played by processes in protocols have appeared in other contexts (*e.g.* [15]). Object orientation based on the actor-paradigm has been studied thoroughly (*e.g.* see [10]). We see this as an orthogonal approach where the computational rather than the control flow -and communication-features are encapsulated using classes and fundamental OO features such as inheritance. It will be interesting and challenging to incorporate this approach into our framework. In a broader perspective, our work has similarities with frameworks such as Ptolemy [9] and Metropolis [1] where the communication and computational aspects are clearly delineated from each other.

3 The Modeling Language

3.1 Model Specification

We model a reactive system as a collection of process classes $\mathcal{P}$, where a class $p \in \mathcal{P}$ is a collection of processes with similar functionalities. Objects belonging to a class will possess a common control flow detailing the pattern of computational and communication activities they can go through and this is described as a state diagram. A communication action will represent a snippet of a protocol,

[1] An EFSM is an extended finite state machine with integer data variables.

namely, the part played by an object of the class in an execution of the protocol. Each such action will have a label of the form γ_r where γ is the name of a transaction (*i.e.*, a sequence diagram together with a set of constraints forming the guard of the transaction), and r is the name of a particular *lifeline* in γ. We define $\Gamma_p = \{\gamma \mid \gamma_r \text{ is an action label in } p \in \mathcal{P}\}$. Further, let $\Gamma = \bigcup_{p\in\mathcal{P}} \Gamma_p$. This induces the function *loc* given by $loc(\gamma) = \{p \mid \gamma \in \Gamma_p\}$. For each $\gamma \in \Gamma$ and process $p \in loc(\gamma)$, we define a set of roles denoted by $R(p,\gamma) = \{r \mid \gamma_r \text{ is an action label in } p\}$. We further make an assumption; *role names are local to a process class.* Thus,

$$\forall\gamma \in \Gamma \bullet \forall p, q \in loc(\gamma) \bullet p \neq q \Rightarrow R(p,\gamma) \cap R(q,\gamma) = \phi$$

Computation actions can also be specified on the local variables of the objects of a process class. Let $V_p = \{v_1, \ldots, v_n\}$ be the set of variables declared in a process class $p \in \mathcal{P}$. Then the state valuation $v \in Val(V_p)$, where

$$Val(V_p) = \{(val(v_1), \ldots, val(v_n)) \mid val(v_i) \in domain(v_i)^2.\}$$

Such computations can be viewed as degenerate sequence diagrams with a single lifeline.

Thus our model consists of a network of interacting process classes where each process class p's behavior is captured by a state diagram TS_p. The set of action labels in TS_p is defined as $Act_p = \{\gamma_r \mid \gamma \in \Gamma_p, r \in R(p,\gamma)\}$. This means that objects of p can participate in transaction γ by playing the role of the lifeline marked by (p, r). Often, different objects of the same process class can play different lifelines in the same execution of a transaction. Hence an action label mentions the transaction name as well as the lifeline/role name. The set of all lifelines in a transaction γ is defined as

$$agents(\gamma) = \{(p, r) \mid p \in loc(\gamma), r \in R(p,\gamma)\}$$

We now give the definition of the specification of our model as,

Definition 1 (Model specification) *Given process classes $\mathcal{P}$, transactions $\Gamma = \bigcup_{p\in\mathcal{P}} \Gamma_p$ and role names $R(_p, _\gamma)$, the model specification* $\mathbf{Spec} = \{TS_p \mid p \in \mathcal{P}\}$ *is the collection of labelled transition systems, where $TS_p = \langle S_p, Act_p, \rightarrow_p, init_p, V_p, vinit_p\rangle$. TS_p is a finite state transition system with: S_p as the finite set of local states of p, $init_p \in S_p$ is the initial state, $\rightarrow \subseteq S_p \times Act_p \times S_p$ is the transition relation, Act_p & V_p are as defined earlier and $vinit_p$ is the initial valuation of V_p*

3.2 Transactions as Sequence Diagrams

A transaction in our model is a guarded sequence diagram. The diagram in Figure 2(b) is an example of a transaction ($NoMoreDest$) consisting of three life lines. We will discuss this example in detail later. Our sequence diagrams will be based on send-receive type of communications. However other features such as synchronizations can be easily added. As usual, the partial order of events in a sequence diagram is the transitive closure of (a) the total order of the events in each process (time flows from top to bottom along each lifeline)

[2] The supported variable types are: *Integer, Real, Boolean* and *Character*.

and (b) the ordering imposed by the send-receive of each message (the send event of a message must happen before its receive event). Internal events can be used to denote update operations associated with the transaction but we will suppress them here for ease of exposition (see Section 7).

3.3 The Guard of a Transaction

The guard of the transaction is a conjunction of the guards of its lifelines. The guard of a lifeline consists of two parts: (a) a constraint on the (finite) execution history of transactions executed (capturing the past behavior) and (b) a propositional formula; both of which must be satisfied by the object currently auditioning for this role.

In this work, we use regular expressions on the alphabet of action labels as guards on past behavior. A propositional formula is constructed using the atomic propositions of a component p which specifies the current state. For the transaction shown in Figure 2(b), no restriction is placed on the histories of the lifelines $Car_{sndStop}$ and $Terminal_{inc}$. However, the *Cruiser* object must have a local history where the last action label executed by this object should be $AlertStop_{rcvDisEg}$. Also, the proposition $dest = 0$, should be *true* for a car object to play $Car_{sndStop}$. The propositional formula for the other two lifelines is trivially *true*. Note that we could have avoided the regular expression guards in our model by blowing up the state diagrams of the process classes. From our experience in modeling some of the examples discussed in this paper, this leads to a substantial blow-up which can be easily avoided by putting in regular expression guards.

To formally define a transaction, we fix an alphabet M of messages and an alphabet A of internal actions which can appear in our transactions. The alphabet of agents is

$$P^R = \{(p, r) \mid p \in \mathcal{P} \wedge \exists \gamma \in \Gamma_p, r \in R(p, \gamma)\}.$$

Definition 2 (Transaction) *A transaction γ is a guarded Message Sequence Chart(MSC) [I:Ch] where Ch is an MSC over (P^R, M, A). I is the guard of the form*

$$\bigwedge_{x \in agents(\gamma)} I_x, \ \textit{where } x = (p, r) \textit{ and } I_x = re_x \wedge g_x.$$

Here re_x is a regular expression built out of the action labels Act_p, capturing the execution history of agent x, and g_x is the propositional formula over the atomic propositions of x.

3.4 Associations

We make use of class-diagrams to model the system structure and associations to model different kinds of communication links that can exist in an interaction among objects. These extensions are crucial for achieving adequate modeling power. The associations are classified as static or dynamic in a manner similar to as presented in [17]. We assume all the associations to be bidirectional.

Static Associations A static association expresses *structural relationship* between the classes. In a class-diagram the static associations are captured using links, annotated with fixed multiplicity $n \geq 1$ at both the association ends. Static associations between the objects remain fixed and do not change at run-time, *i.e.*, they represent permanent or unchanging relations between the classes during the execution. We can refer to static associations in transaction guards to impose the restriction that process classes chosen for a given pair of agents should be statically related.

Dynamic Associations A dynamic association expresses *behavioral relationship* between the classes, which in our case implies that the objects of two dynamically associated classes can become related to each other during simulation for some period of time, exchange messages (by executing transactions together) and then leave that relation. In the class-diagram dynamic associations are captured using links, annotated with multiplicity range $0..n$, $n \geq 1$ at both the association ends. Thus the contents of a dynamic relation can change during simulation (unlike the static relation).

4 Execution Semantics

We now turn to the the execution semantics of our models. At the initial configuration, every object of the class p will be residing at the designated initial state of the state diagram TS_p with null history and some initial values for the local variables. The system will move from a configuration c to a configuration c' by executing a transaction, say γ. For doing so, we must be able to assign to each lifeline γ_r, an object which is at the appropriate control state and has a history and variable valuation which satisfies the guard associated with the role r of transaction γ. Instead of formalizing this idea, we prefer to illustrate it in a concrete setting.

Suppose c is a configuration at which

- 2 *Car* objects oc_1 and oc_2 are residing in state *stopped* and third object, oc_3, in state $s2$ of TS_{Car} (shown in Figure 1(a)), having values for the variable *dest* as 0, 2 and 1 repectively.
- 3 *Cruiser* objects, $or_1 \dots or_3$ are residing in state *started* of $TS_{Cruiser}$ (shown in Figure 1(b)) such that, $or1$ & $or2$ have as their current history-

  $$(Act_{Cruiser})^{\star}.AlertStop_{rcvDisEg}$$

  and $or3$ has as its current history-

  $$(Act_{Cruiser})^{\star}.DepartAckA_{started}.$$

- 6 *Terminal* objects, $ot_1 \dots ot_6$ are residing in state $s1$ of $TS_{Terminal}$, (shown in Figure 2(a)).

Now in order to execute the transaction $NoMoreDest$ at c, we observe that: for *Cruiser*, only or_1 and or_2's history satisfy the guard of the lifeline $Cruiser_{rcvStop}$, though or_3 is also in the appropriate control state. Hence, we

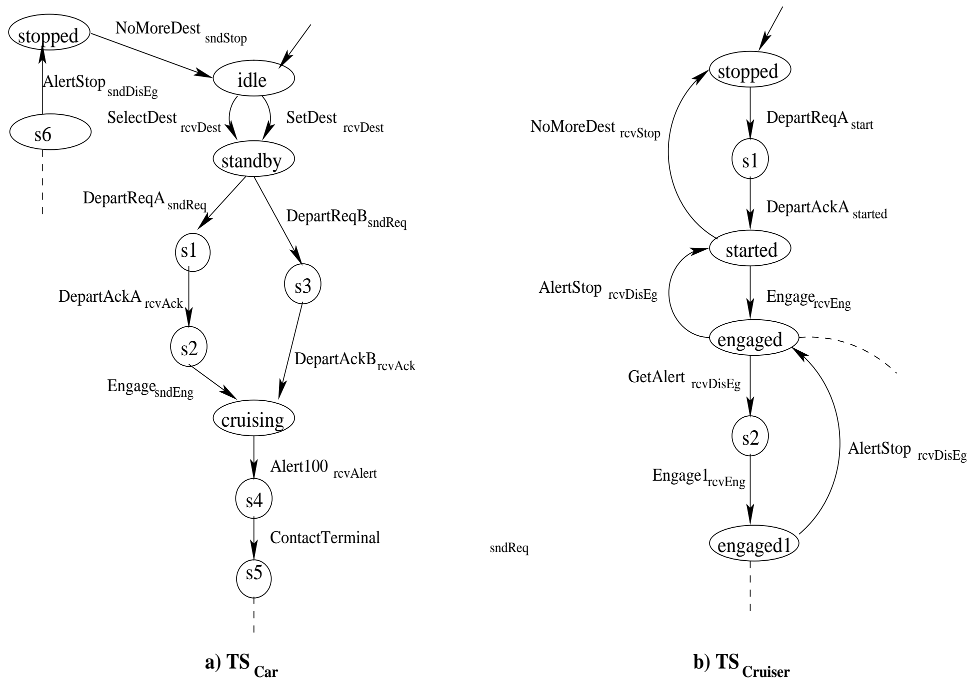

Figure 1: **State Diagrams for *Car* and *Cruiser*.**

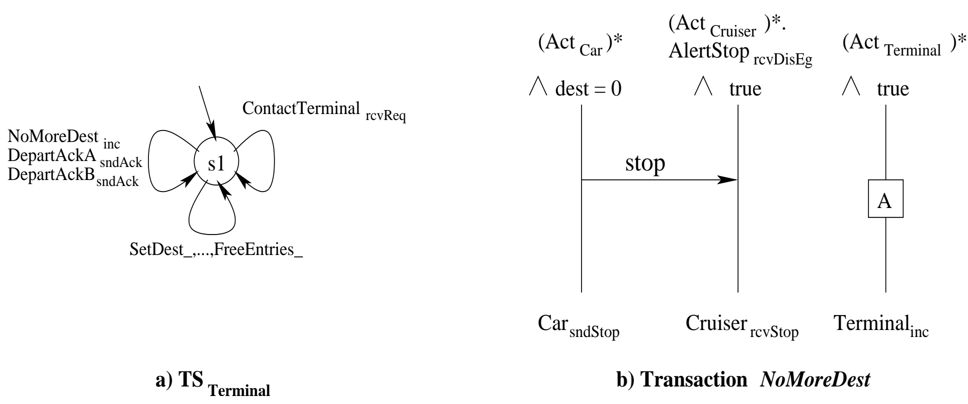

Figure 2: **State Diagram for *Terminal* and transaction *NoMoreDest*.**

Contact Terminal	**inserts**	(Car_{sndreq}, $Terminal_{rcvReq}$)	to	**ItsTerminal**
DepartAck(A/B)	**checks**	(Car_{rcvAck}, $Terminal_{sndAck}$)	belongs to	**ItsTerminal**
DepartAck(A/B)	**deletes**	(Car_{rcvAck}, $Terminal_{sndAck}$)	from	**ItsTerminal**

Figure 3: **Dynamic Relation *ItsTerminal***

can choose either or_1 or $or_2$to play the role in given transaction. For the Car, both oc_1 and oc_2 being in the *stopped* state are eligible for the role $Car_{sndStop}$, however the propositional guard of this lifeline, $dest = 0$, is satisfied only by oc_1 (hence only oc_1 can be chosen for Car). For $Terminal_{inc}$, both the history and propositional guards impose no restriction and hence we can choose any object residing in the appropriate control state (i.e. $s1$).

Assume that besides oc_1 for Car, or_1 and ot_1 are chosen for $Cruiser$ and $Terminal$ respectively to play the roles in $NoMoreDest$ at configuration c. Let the resulting configuration be c'. At c' all objects other than oc_1, or_1 and ot_1 will have their control states and histories unchanged from c. At c', the object oc_1 will reside at *idle* due to transition labeled $NoMoreDest_{sndStop}$ from *stopped* to *idle*. Similarly or_1 and ot_1 will reside at *stopped* and $s1$ in their respective transition systems.

Execution in the presence of *associations* In the case of *static associations*, the only additional restriction to the execution semantics as illustrated above, will be to choose statically related objects for the given lifelines. We now illustrate the use of *dynamic associations* using the rail-car example. Details about this example will be given later in Section 6. During the system run various rail-cars enter and leave the terminals along their path. When a car is approaching a terminal, it sends arrival request to that terminal by executing *ContactTerminal* transaction and while leaving the terminal, its departure is acknowledged by the terminal by executing *DepartReqA* or *DepartReqB* (refer to Figure 1(a)). Hence, the guard of *DepartReq(A/B)* requires that the participating *Car* and *Terminal* objects should have together executed *ContactTerminal* (when the car was entering this terminal). Since this condition involves a relationship between the local histories of multiple objects, we cannot capture it via regular expressions over the individual local histories. Instead, we make use of dynamic relation *ItsTerminal* between the Car and $Terminal$ classes as part of our specification. If car object x and terminal object y play the roles for the lifelines Car_{sndReq} and $Terminal_{rcvReq}$ in *ContactTerminal* (refer to Figure 1(a) for TS_{Car} and Figure 2(a) for $TS_{Terminal}$), then the effect of *ContactTerminal* is to insert the tuple (x, y) into the *ItsTerminal* relation. The *DepartAck(A/B)* transaction's guard now includes the check that the object x', corresponding to lifeline Car_{rcvAck} and object y', corresponding to lifeline $Terminal_{sndAck}$ be related by the dynamic relation *ItsTerminal*, i.e. $(x', y') \in ItsTerminal$ (refer to Figure 3); so if objects x and y have been selected to play the Car_{rcvAck} and $Terminal_{sndAck}$ roles in *DepartAck(A/B)*, the check will succeed. Furthermore, the effect of *DepartAck(A/B)* transaction is to remove the tuple (x, y) from *ItsTerminal* relation.

4.1 Behavioral Partitions

One of our key objectives is to avoid having to keep track of the identities of the objects of a process class during execution. To achieve this, the objects of a process class will be grouped together into "behavioral partitions", based on their potential future behaviors. Note that for an object of a process class p, the transactions it can execute depends on its current state in TS_p, its execution history (which determines the satisfaction of regular expressions occurring in guards of lifelines mentioned in the transition labels in TS_p), and valuation of its local variables (determining the satisfaction of propositional guards of lifelines in TS_p). In the examples we have studied it suffices to deploy very restricted regular expressions (in the guards of the transactions), which can be represented as DFAs and hence, from a pragmatic point of view, complexity is not an issue. For a process class p and any transaction $\gamma \in \Gamma_p$ we now define the following:

$$\begin{aligned} RE_\gamma &= \cup_{x=(p,r)\in agents(\gamma)} re_x, \text{ and} \\ RE_{\Gamma_p} &= \cup_{\gamma\in\Gamma_p} RE_\gamma. \end{aligned}$$

Let the minimized DFAs corresponding to the regular expressions in RE_{Γ_p} be denoted by the set M_{Γ_p}. We now give the definition of a "behavioral partition".

Definition 3 (Behavioral Partition) *For a process class p, let $TS_p = \langle S_p, Act_p, \rightarrow_p, init_p, V_p, vinit_p\rangle$ be the transition system associated with p and $M_{\Gamma_p} = \{m_1, \ldots, m_k\}$ be the set of minimized DFAs corresponding to regular expressions in RE_{Γ_p}, as explained above. Then a* behavioral partition *b_p of class p is a $k+2$ tuple $(s, s_1, \ldots, s_k, v)$, where*

$$s \in S_p, s_1 \in S_{m_1}, \ldots, s_k \in S_{m_k}, v \in Val(V_p).$$

S_{m_i} denotes the set of states of DFA m_i. We use B_p to denote the set of all behavioral partitions of process class p.

Thus, two objects o_1 and o_2 of process class p are in the same *behavioral partition* (at a configuration) if and only if the following conditions hold.

- o_1 and o_2 are currently in the same state of TS_p,
- They have the same valuation of local variables, and
- They are in the same states of all the DFAs corresponding to the regular expressions in RE_{Γ_p}.

The first two criterion are clear. As for the third one, the past histories of two objects need not be identical in order for them to be in the same behavioral partition. However any common future evolution of the two histories must satisfy the same set of guards (of all the lifelines mentioned in the transition labels of the state diagram of the class). This criterion implies that the *potential* computation trees of two objects in the same behavioral partition are the same. This is a strong type of behavioral equivalence to demand and there are many weaker possibilities but we will not explore them here.

Bounding the number of Behavioral Partitions Recall that each action label $a = \gamma_r \in Act_p$ is a lifeline of a transaction and in the present setting has a guard in the form of $re_x \wedge g_x$, where $x = (p, r) \in agents(\gamma)$ and re_x is a regular expression. Let $M(a)$ denote the number of states of the minimal deterministic automaton accepting re_x. The number of behavioral partitions of process class p is then bounded by $|S_p| \times \prod_{a \in Act_p} M(a) \times |Val(V_p)|$, where $|S_p|$ is the number of states of TS_p and $|Val(V_p)|$ is the number of all possible variable valuations of V_p. Note that for reactive systems we are more interested in communication aspects, than in internal computations, hence we make use of local variables essentially for enumerating certain properties, which may otherwise lead to large number of states and transactions in specification, if specified explicitly via different states. Thus, even though the usage of local variables may lead to large number of behavioral partitions, they will essentially remain bounded. From our experience, they remain small for most of the cases.

For example, consider $TS_{Cruiser}$ as shown in Figure 1(b); it contains 7 states, i.e. $|S_{Cruiser}| = 7$[3], and the description of the transaction *NoMoreDest* shown in Figure 2(b). This transaction is executed when a car has no further destinations to visit (as indicated by the propositional guard, $dest = 0$, of lifeline $Car_{sndStop}$), and so it stops its *cruiser* by sending it the message *stop*. For the lifeline corresponding to the cruiser, $Cruiser_{rcvStop}$, we see that it has a non-trivial regular-expression $(Act_{Cruiser})^{\star}.AlertStop_{rcvDisEg}$ as its guard. The DFA for this regular expression contains only 2 states:

- $s1$ which is the *initial state* accepting the regular language

$$((Act_{Cruiser})^{\star}.\neg AlertStop_{rcvDisEg} \mid \epsilon).$$

.

- $s2$ which is the *final state* accepting the regular language

$$((Act_{Cruiser})^{\star}.AlertStop_{rcvDisEg}).$$

Of all the transition labels appearing in $TS_{Cruiser}$, only $NoMoreDest_{rcvStop}$ is guarded using a regular expression (propositional guards of lifelines corresponding to all the labels are trivially *true*). Also, there is no local variable declared for $Cruiser$. Thus, the bound on the number of behavioral partitions for $Cruiser$ is 7*2=14. By carefully examining $TS_{Cruiser}$ and the above regular expression, we can derive the tighter bound of only 8 behavioral partitions.[4] This is an interesting observation, since it indicates that no matter how many objects we choose to have in process class $Cruiser$, they will always be divided into maximum of 8 partitions; and infact in our experiments we have used 24 and 48 $Cruiser$ objects representing two different configurations.

In what follows, for a process class p we denote a behavioral partition $b_p \in B_p$ as $\langle s, History, v \rangle$ where s is a state in TS_p, $History$ is a function which maps each action label $a \in Act_p$ to a state in the minimal DFA accepting the regular expression guard of a, and $v \in Val(V_p)$.

[3]We have not shown one state and its associated transitions to maintain clarity.

[4]We do not compute bounds in this way, but this is just to show that actual bound may in-fact be lower than computed using the formula described earlier.

Simulation example with behavioral partitions Consider $TS_{Cruiser}$ shown in Figure 1(b). Suppose we simulate the specification with 24 *Cruiser* objects (assume that other process-classes are also appropriately populated with objects). As mentioned earlier, in $TS_{Cruiser}$, only $NoMoreDest_{rcvStop}$ is guarded using regular expression (i.e. there is no restriction on the execution histories for other action labels). Let us call DFA corresponding to $NoMoreDest_{rcvStop}$ as *DFA1*, which contains two states as described before. Initially all the objects are in the *stopped* state of $TS_{Cruiser}$. And all of them have null execution history satisfying the trivial regular expression ϵ and hence are in the initial state $s1$ of DFA1. Thus they are in the same behavioral partition $\langle stopped, NoMoreDest_{rcvStop} \rightarrow s1, \phi\rangle$, where ϕ represents an empty variable valuation.

Suppose now one cruiser object, say $o1$, executes the trace "$DepartReqA_{start}$, $DepartAckA_{started}$, $Engage_{rcvEng}$, $AlertStop_{rcvDisEg}$". As a result it now resides in control state *started*, and its execution history satisfies the regular expression corresponding to state $s2$ of DFA1.Another cruiser object, say $o2$, executes the trace "$DepartReqA_{start}$, $DepartAckA_{started}$" and also resides in control state *started*. However, unlike $o1$, the execution history of $o2$ satisfies the regular expression corresponding to state $s1$ of DFA1. Note that, during system simulation objects of other process classes are also involved in various transactions, some of which may not even involve cruiser objects. Here we focus only on the $Cruiser$ class for the purpose of illustration. After above executions, we now have three behavioral partitions for cruiser objects.

1. $\langle stopped, NoMoreDest_{rcvStop} \rightarrow s1, \phi\rangle$ has 22 objects which were idle.
2. $\langle started, NoMoreDest_{rcvStop} \rightarrow s2, \phi\rangle$ has object $o1$, which last played the lifeline $AlertStop_{rcvDisEg}$.
3. $\langle started, NoMoreDest_{rcvStop} \rightarrow s1, \phi\rangle$ has object $o2$, which last played the lifeline $DepartAckA_{started}$.

In the preceding, objects in different behavioral partitions have different sets of actions enabled, thereby obviously leading to different possible future evolutions. Now let object $o1$ execute the transaction $NoMoreDest_{rcvStop}$. Note that we could not have chosen $o2$ to play this role, since it does not satisfies the regular expression guard corresponding to $NoMoreDest_{rcvStop}$ (being in state $s1$ of DFA1, which is *not* the final state). The above execution results in a merger of the first two behavioral partitions shown in the preceding, that is, $o1$ is now indistinguishable (behaviorally) from the 22 objects which never participated in any transaction. For all of these 23 objects, the action $DepartReqA_{start}$ is now enabled. This is the manner in which behavioral partitions will be split and merged during simulation.

4.2 Simulation of Core Model

To explain how symbolic simulation takes place, we first define the notion of "configuration".

Definition 4 (Configuration) *Given a model specification* Spec *consisting of process classes* $\mathcal{P}$*, such that each process class* $p \in \mathcal{P}$ *contains* N_p *objects, the configuration of* Spec *is defined as* $\mathsf{cfg} = \langle \{B_p \mid p \in \mathcal{P}\}, \sigma \rangle$ *where*

- B_p *is the set of all behavioral partitions of process class* p*.*
- $\sigma : \bigcup_{p \in \mathrm{P}} B_p \to \mathbb{N} \cup \{0\}$ *is a mapping defined as* $\sigma(b) = c_p(b)$ *for* $b \in B_p$*, where* $c_p : B_p \to \mathbb{N} \cup \{0\}$ *gives the number of objects in each behavioral partition of* p*, s.t.* $\Sigma_{b \in B_p} c_p(b) = N_p$*.*

The set of all configurations of Spec *is denoted as* $\mathcal{C}_{Spec}$*.*

Our symbolic simulation efficiently keeps track of the objects in various process classes by maintaining the current configuration, s.t. only only the behavioral partitions with non-zero counts are kept track of. The system moves from one configuration to another by executing a transaction. In other words, the transition relation of our operational model is a subset of $\mathcal{C}_{S}pec \times \Gamma \times \mathcal{C}_{S}pec$. How can our simulator check whether a specific transaction γ is enabled at the current configuration c? We say that γ is enabled at c if for every lifeline of γ we can assign a distinct object to take up that lifeline (*i.e.* we do not want the same object to act as several lifelines in the same execution of a transaction γ). Since we do not keep track of object identities, we define the notion of *witness partition* for a lifeline, from which an object can be chosen to play the role in that lifeline.

Definition 5 (Witness partition) *Let* $\gamma \in \Gamma$ *be a transaction and* $\mathsf{cfg} \in \mathcal{C}_{Spec}$ *be a configuration. For an agent* $x = (p, r) \in agents(\gamma)$ *we say that a behavioral partition* $b = (s, s_1, \ldots, s_k, v)$ *is a witness partition, denoted as* $witness(x, \gamma, \mathsf{cfg})$*, for the lifeline* γ_r *at configuration* cfg *if*

1. $b \in B_p$*.*
2. $s \xrightarrow{(\gamma_r)} s'$ *is a transition in* TS_p
3. *If* $m_i \in M_{\Gamma_p}$ *is the DFA corresponding to* re_x*, then* s_i *is an accepting state.*
4. $v \in Val(V_p)$ *satisfies* g_x*.*
5. $\sigma(b) \neq 0$*, that is there is at least one object in this partition.*

Intuitively the third and fourth conditions above require that the objects of a witness partition have: a) an execution history satisfying the regular expression and b) variable valuation satisfying the propositional guard, corresponding to the lifeline for which this partition is chosen for. An "enabled transaction" can now be defined as:

Definition 6 (Enabled Transaction) *Let* γ *be a transaction and* cfg *be a configuration. We say that* γ *is enabled at* cfg *iff for each agent* $x = (p, r) \in agents(\gamma)$*, there exists a witness partition* $witness(x, \gamma, \mathsf{cfg})$ *such that*

- *If* $b \in B_p$ *is assigned as witness partition of* n *agents in* γ*, then* $\sigma(b) \geq n$*. This ensures we do not allow one object to play two different roles in a transaction.*

For an arbitrary configuration cfg, we use $En(\mathsf{cfg})$ to denote the set of enabled transactions at cfg.

Before we continue to describe the *effect* of executing an *enabled* transaction at a configuration cfg, we first explain the term "destination partition", which is the partition to which an object moves from its "witness partition" after executing a transaction.

Definition 7 (Destination Partition) *Let γ be an enabled transaction at configuration cfg and $b = (s, s_1, \ldots, s_k, v)$ be the witness partition for an agent $x = (p, r) \in agents(\gamma)$. Then, the destination partition of b w.r.t. transaction γ and agent x is a behavioral partition $b' = (s', s'_1, \ldots, s'_k, v')$, where*

- $s \xrightarrow{(\gamma_r)} s'$ *is a transition in* TS_p.
- $s_i \xrightarrow{(\gamma_r)} s'_i$ *for* $i = 1, \ldots, k$ *are transitions in* $m_i \in M_{\Gamma_p}$.
- $v' \in Val(V_p)$ *is the effect of executing* γ_r *on* v.

We denote the destination partition of b w.r.t. to transaction γ and agent $x \in agents(\gamma)$ as $b' = dest(b, \gamma, x)$. Thus, any object in behavioral partition b moves to partition $dest(b, \gamma, x)$ by performing role r in transaction γ, where $x = (p, r)$ is an agent in γ. It is important to note that above definition makes explicit the dynamic migration of objects from partition to another during simulation.

We now describe the effect of executing an enabled transaction at a given configuration. Let cfg be a configuration and γ be an enabled transaction at cfg. Computing the new configuration cfg' as a result of executing transaction γ in configuration cfg thus involves computing the destination behavioral partition for each behavioral partition of a process class, and then computing the new count of objects for each behavioral partition.

The operational rule of our model is given below. It specifies that if a transaction is enabled at a configuration cfg then it can execute and the system arrives at a new configuration cfg'.

$$\frac{\begin{array}{c}\mathsf{cfg} = \langle \{B_p\}_{p \in \mathcal{P}}, \sigma \rangle \\ \gamma \in En(\mathsf{cfg}) \\ B'_p = \{dest(b, \gamma, x) \mid b \in B_p \wedge x = (p, r) \in agents(p, \gamma) \wedge b = w\} \cup B_p \\ \forall b \in B_p \,.\, \sigma'(b) = \sigma(b) + |\{x \mid b = dest(w, \gamma, x)\}| \\ -|\{x \mid b = w\}| \\ where\ w = witness(x, \gamma, \mathsf{cfg}) \\ \mathsf{cfg}' = \langle \{B'_p\}_{p \in \mathcal{P}}, \sigma' \rangle \end{array}}{\mathsf{cfg} \xrightarrow{\gamma} \mathsf{cfg}'}$$

4.3 Simulation of Models with Associations

In the specification, for any dynamic relation, we describe the effect of each transaction on the relation (in terms of addition/deletion of tuples of objects into the relation). Furthermore, the guard of any transaction can contain a membership constraint on one or more of the specified dynamic relations. In terms of simulation of concrete objects, it is clear how our extended model should be executed. However, since we do not maintain identities of concrete

objects during simulation, it is not possible to take the obvious approach. We now describe how we can exploit the notion of behavioral partitions for this purpose.

Simulating the extended model Our symbolic execution mechanism needs little change in the presence of static associations between process classes. The guard of a transaction can refer to these static associations. So, the only change in the symbolic execution mechanism is that we need to take these associations into account while assigning the witness behavioral partitions for each lifeline of a transaction. As for dynamic associations, the key question here is how we maintain relationships between objects if we do not keep track of the object identities. We do so by maintaining *dynamic associations between behavioral partitions.* To illustrate the idea, consider a binary relation D which is supposed to capture some dynamic association between two objects of process class p. In our symbolic execution, each element of D will be a pair (b, b') where b and b' are behavioral partitions of class p. To understand what $(b, b') \in D$ means, consider the concrete simulation of the process class p. If after an execution π (a sequence of transactions), two concrete objects o, o' of process class p get related as $(o, o') \in D$ then the symbolic execution along the same sequence of transactions π must produce $(b, b') \in D$ where b (b') is the behavioral partition in which o (o') resides after executing π. The same idea can be used to manage dynamic relations of larger arities.

From the discussion above, we can easily modify the operational semantic rule in Section 4 into the one as follows, with associations being constrained, where $\alpha_a(\mathsf{cfg})$ denotes the current content of the association a at configuration cfg, which consists of a set of behavior partition pairs. And we use $Inserts(x_1, x_2)\ to\ a$ and $Deletes(x_1, x_2)\ from\ a$ to mean that the pair (x_1, x_2) is inserted into and removed from the association a in the transaction respectively.

$$\frac{\begin{array}{c} \mathsf{cfg} = \langle \{B_p\}_{p \in \mathcal{P}}, \sigma \rangle \\ \gamma \in En(\mathsf{cfg}) \\ B'_p = \{dest(b, \gamma, x) \mid b \in B_p \wedge x = (p, r) \in agents(p, \gamma) \wedge b = w\} \cup B_p \\ \forall b \in B_p\ .\ \sigma'(b) = \sigma(b) + |\{x \mid b = dest(w, \gamma, x)\}| \\ -|\{x \mid b = w\}| \\ where\ w = witness(x, \gamma, \mathsf{cfg}) \\ \mathsf{cfg}' = \langle \{B'_p\}_{p \in \mathcal{P}}, \sigma' \rangle \\ \forall a \in A\ .\ \alpha_a(\mathsf{cfg}') = \alpha_a(\mathsf{cfg}) \oplus \\ \bigcup_{Inserts(x_1, x_2)\ to\ a} \{(dest(w_1, \gamma, x_1), dest(w_2, \gamma, x_2))\} \\ \ominus \bigcup_{Deletes(x_1, x_2)\ from\ a} \{(w_1, w_2)\} \\ where\ x_i \in agents(\gamma) \wedge w_i = witness(x_i, \gamma, \mathsf{cfg}) \\ \forall x_1, x_2\ Check((x_1, x_2) \in a) \Rightarrow (w_1, w_2) \in \alpha_a \end{array}}{\mathsf{cfg} \xrightarrow{\gamma} \mathsf{cfg}'}$$

Note that here the operator $\oplus$ and $\ominus$ are used for approximate set union and set minus respectively. If the pair (x_1, x_2) is inserted into association a when executing γ, then the pair of destination partitions of the witness partitions of x_1 and x_2 is added into α_a at the new configuration. On the other hand, if the

pair is deleted from the association a, we need to check the number of objects residing in the witness partitions of x_1 and x_2, if they are greater than 1, then the pair (w_1, w_2) will still be kept in α_a. We will only remove it from the set when the number of remaining objects in either of the witness partitions is 0.

Example As discussed in Section 3.4, dynamic relation *ItsTerminal* is maintained between the objects of class *Car* and *Terminal* (as shown in Figure 3). We do not show the details of the *ContactTerminal* and *DepartAck(A/B)* protocols here for lack of space, but just give their effect on the dynamic relation *ItsTerminal*.

Now, suppose an object from class *Car* and one from class *Terminal* play the roles Car_{sndReq} and $Terminal_{rcvReq}$ in the transaction *ContactTerminal*. Let b_{Car} & (b_{Term}) be the behavioral partitions in to which the objects of *Car* & *Terminal* go by executing $ContactTerminal_{sndReq}$ & $ContactTerminal_{rcvReq}$ repectively. Say, this is followed by another object pair from the same classes, playing the roles in another execution of *ContactTerminal*. Assume that, this time the car object goes to another behavioral partition b_{Car1} (possibly due to different execution history than the previous car object), while the *Terminal* object goes to the same behavioral partition b_{Term}, which now contains atleast two objects. So the contents of *ItsTerminal* now are,

$$\text{ItsTerminal} = \{(b_{Car}, b_{Term}), (b_{Car1}, b_{Term})\}.$$

Now when we execute *DepartAck(A/B)* transaction, we will pick a pair from this relation as witness behavioral partitions for lifelines Car_{rcvAck} and $Terminal_{sndAck}$. We *have not* maintained information about which $Terminal$ object in b_{Term} is related to which Car of b_{Car} or b_{Car1}. But this information is not required for our symbolic simulation to proceed. This is because, the two objects in the behavioral partition b_{Term} are behaviorally identical, so during symbolic simulation we can safely assume that we have chosen the right related object corresponding to the car object chosen from either b_{Car} or b_{Car1}.

4.4 The Soundness and Incompleteness of Symbolic Execution

One important problem about the symbolic execution mechanism is its correctness. In other words, does the symbolic execution of a model preserves all possible concrete execution traces? Ideally, it should be required that any behavior exhibited by concrete execution of a model is also exhibited by symbolic execution. The following theorem states the soundness of the symbolic execution w.r.t. concrete execution, i.e., the symbolic execution of a model preserves all possible concrete execution traces.

Theorem 1 *Consider the symbolic execution of a model specification* **Spec** *with process classes $\mathcal{P}$ with N_p objects of process class $p \in Proc$. Let σ be a sequence of transactions exhibited when we give unique names to the $N = \Sigma_{p \in \mathcal{P}} N_p$ objects and execute* **Spec**. *Then σ must also be exhibited in the symbolic execution of* **Spec**.

Proof: To establish this result, it is sufficient to prove that any sequence of transactions allowed by concrete execution of a model is also allowed by symbolic

execution of the same model. Since the transaction alphabet as well as the number of possible global states of **Spec** is guaranteed to be finite, the result can be proved by induction on the length of the sequence of transactions. When the sequence is empty, then the result is trivially satisfied. Let $\sigma = \sigma_1 \circ \tau$ where σ_1 is a sequence of transactions exhibited in both concrete and symbolic executions, $\circ$ denotes concatenation, τ is a single transaction which can be executed in concrete execution. Then we just need to show that τ can be executed in symbolic execution after the behavior σ_1 is exhibited.

We first consider the case in which no association is involved. From the assumption that σ_1 is exhibited in both concrete and symbolic executions and τ is executable in concrete execution after σ_1, we can find concrete objects to satisfy the guard of τ in the concrete execution. And after the execution of σ_1, for every concrete object o which will take part in the transaction τ, being in control state s and DFA states $s_1, \cdots, s_k$, and state valuation v, we can construct a behavioral partition $b = (s, s_1, \cdots, s_k, v)$, which contains at least the objects that has control state s and satisfies the corresponding guard condition of τ. From Definition 6, we know that $\sigma(b)$ at least equals to the number of objects taking up the same role r as o.

Thus for every agent x involved in transaction τ, there exists a behavioral partition b which satisfies the guard condition of x. From Definition 5 we can easily know that b is the witness partition for the role r of x in transaction τ at the configuration cfg after the symbolic execution of σ_1. Furthermore, b contains at least the objects taking up role r. So τ is enabled at cfg according to Definition 6. Therefore τ is executable in symbolic simulation.

For static associations, if there is a static association between two objects o_1 and o_2 involved in a concrete transaction sequence, then we need to show that there is also an association between their corresponding behavioral subclasses in the symbolic execution. In fact, we can construct the behavior subclasses corresponding to o_1 and o_2 by using their control states and the DFA states, and the association should be kept as a reference in the subclasses. Therefore the static association also exists between the subclasses in the symbolic execution. Moreover, static associations are not affected by execution. So the symbolic execution is still correct w.r.t. the concrete execution.

We now consider the case for dynamic associations. We need to show that the effect of symbolic execution of any transaction sequence σ preserves the concrete execution. We prove the result by induction on the length of σ. For length 0, the association contains no object pairs and the result trivially holds. Suppose $\sigma = \sigma_1 \circ \tau$ and the trace σ_1 is exhibited by both concrete and symbolic executions. If the dynamic association a between o_1 and o_2 is introduced in τ, then in concrete simulation, the pair (o_1, o_2) is added to α_a. In the symbolic simulation, let b_1 and b_2 be the behavioral partitions corresponding to the objects o_1 and o_2 respectively. According to the semantics, we just need to add the pair of their destination partitions (b'_1, b'_2) into α_a. If it already exists in α_a, then nothing is modified in α_a. If a is removed between o_1 and o_2 in τ, then in concrete simulation we simply remove the pair (o_1, o_2) from α_a. For symbolic case, if there remains objects of the behavioral partitions b_1 and b_2, then we just keep (b_1, b_2) in α_a since we do not know if the remaining objects in these subclasses are related or not. And (b'_1, b'_2) will not be added in α_a. If a pair of objects taking part in the transaction is in a, then in concrete simulation nothing is changed and only the entry for the pair will be checked. In

symbolic case, the pair of destination subclasses (b_1', b_2') will be added into α_a if it is not in. Therefore, whenever τ can be executed in the concrete setting, it is executable in the symbolic simulation and the objects involved in dynamic associations may still be associated to some objects in the behavioral subclasses to which its witness subclass is associated with. This completes the proof of the theorem. □

Next we address the following problem: For a given symbolic execution σ, could we always find a concrete execution? In other words, is there any choice of objects such that they can execute the trace concretely?

If we consider the executions where only static associations are permitted, then we can find that the concrete execution can be found. We prove it by induction on the length of σ. For length 0, the result trivially holds. Let $\sigma = \sigma_1 \circ \tau$ and for σ_1 the result holds, then after the execution of σ_1, for every behavioral partition $b = (s, s_1, \cdots, s_k, v)$, we can always find at least one object o of the subclass, having control state s and DFA states $s_1, \cdots, s_k$. Since the transaction τ is enabled in the symbolic execution after σ_1, both the guards and static associations should be satisfied. Because the guards are encoded in the DFA states, the objects corresponding to the behavioral partitions involved in τ also satisfy the guard condition. For static associations, if there exists a static association asc between behavioral partitions b_1 and b_2, then we can always find objects o_1 and o_2 of them respectively, such that they are statically associated by asc. So we just need to choose o_1 and o_2 for the concrete execution, thus both the guard condition on history and static associations are satisfied in the concrete execution, and hence the trace can be executed concretely.

However, for dynamic associations, the situation is not so simple. In fact, it is not always possible that a concrete execution can be found for a behavior in the symbolic execution. We now give an example in which a behavior observed in the symbolic simulation (in the presence of dynamic associations) does not correspond to any actual system run. Consider the system consisting of 3 process classes: A, B and C, as shown in Figure 4, together with the effect of transactions $T1$, $T2$ and $T3$ on the dynamic associations Asc_1 and Asc_2. Assume that all the action labels shown in the example have trivial guards, i.e. they do not impose any restriction on the execution history or state of the object to play that lifeline (of course object should be in appropriate control state). Also, initially process class A contains object $o1$ in the initial partition b_{A1}, B contains objects $o2$ & $o3$ in initial partition b_{B1} and C contains $o4$ in initial partition b_{C1}.

The given system then executes the trace: $T1.T2.T3$ in a symbolic run, such that:

- $o1$, $o2$ execute $T1$, playing role in the lifelines $T1_{p1}$ and $T1_{p2}$ respectively and move to behavioral partitions b_{A2} and b_{B2}. Also, (b_{A2}, b_{B2}) is inserted in Asc_1.

- $o4$, $o3$ execute $T2$, playing role in the lifelines $T2_{q1}$ and $T2_{q2}$ respectively and move to behavioral partitions b_{A2} and b_{C2}. As a result (b_{C2}, b_{B2}) is inserted in Asc_2. Note that b_{B2} now contains two objects: $o2$ and $o3$.

- Now when $T3$ executes, as per the requirements on the dynamic relations as imposed by $T3$ (refer to Figure 4), behavioral partitions b_{A2}, b_{B2} and

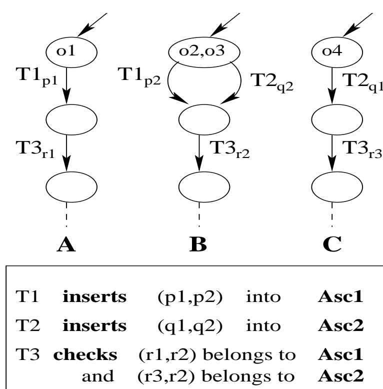

Figure 4: An example

b_{C2} are assigned as witness partitions to the lifelines $T3_{r1}$, $T3_{r2}$ and $T3_{r3}$. Then an object is chosen from each of the partitions (it can be any one of those present in the partition) to execute $T3$. Note that, from b_{A2} and b_{C2} only $o1$ and $o4$ can be chosen, whereas either $o2$ or $o3$ can play the role in $T3_{r2}$.

However, as we can observe, though all the constraints in the symbolic simulation have been met, this trace can never be executed in concrete simulation, since for process class B, neither $o2$ nor $o3$ satisfy both the relational requirements imposed by $T3$. In other words, there is no object $o' \in B$, such that $(o_1, o') \in Asc_1$ and $(o_4, o') \in Asc_2$ when $T3$ is executed.

5 Checking a Symbolic Execution Run

From the discussion given previously, we know that we can not always find a concrete run which corresponds to the behavior presented by a symbolic execution when dynamic associations are involved. In the following we show an approach to construct a check of the existence of the concrete run corresponding to the symbolic one w.r.t. dynamic associations. In the symbolic execution, the simulation only keeps track of the behavioral partitions instead of object identities, and dynamic association a contains the pairs (b, b') of behavioral partitions in which the objects associated by a reside respectively. To find a concrete run for a symbolic one, for the system moving from a configuration to another one by executing a transaction sequence σ, we must be able to assign to each lifeline an object which is at the appropriate control state and satisfies the guard associated with the corresponding role. One possible approach is to check all possible object identities. However, this will be too exhaustive because of the huge number of possible object pairs involved in dynamic associations. Therefore, we use symbolic variables instead of concrete object identities in the checking process. The details of the approach is given as follows[5]:

For a sequence of transactions $\tau_1, \tau_2, \cdots, \tau_n$, suppose each τ_i has k_i separate

[5]For simplicity, here we only consider binary associations.

lifelines, then we construct a set of symbolic variables V_i for each τ_i:

$$V_1 = \{v_1^1, \cdots, v_{k_1}^1\}$$
$$\cdots$$
$$V_n = \{v_1^n, \cdots, v_{k_n}^n\}$$

Each variable v_j^i corresponds to a lifeline j in τ_i. Every variable has a domain of values that it can take. The domains of these variables are the possible identities of the objects involved in the transaction. Then we develop constraints on possible values of variables in V_i based on dynamic associations.

We consider the case for a sequence $\tau_1, \tau_2, \cdots, \tau_n$ of transactions. First, an intuitive constraint is that if a behavioral partition appears in different transactions in the symbolic execution, then the corresponding symbolic variables should also be identified. In general, let τ_j and τ_l be two arbitrary transactions in the sequence, without loss of generality, we suppose $j < l$. Suppose in the symbolic execution transaction τ_j involves lifelines $r_1^j, \cdots, r_{k_j}^j$, and the corresponding witness behavioral subclasses $b_1^j, \cdots, b_{k_j}^j$, we have symbolic variables $v_1^j, \cdots, v_{k_j}^j$ for the lifelines respectively. And similarly, τ_l involves lifelines $r_1^l, \cdots, r_{k_l}^l$, and the corresponding witness behavioral subclasses $b_1^l, \cdots, b_{k_l}^l$, we have symbolic variables $v_1^l, \cdots, v_{k_l}^l$ for the lifelines respectively. If there exists a sequence of behavioral subclasses $b_0, \cdots, b_{l-j}$ such that $b_0 = b_m^j$ for some $1 \leq m \leq k_j$, $b_n = dest(b_{n-1}, \tau_{j+n}, r_{t_n}^{j+n})$, and $b_{l-j} = b_s^l$, then we should derive the equations

$$v_m^j = v_{t_1}^{j+1} = \cdots = v_s^l$$

In general, for a sequence of transactions we can get a family of such equations and use them also as constraints on the symbolic variables.

Additionally, we consider the dynamic associations involved in the sequence of transactions $\tau_1, \tau_2, \cdots, \tau_n$. For an arbitrary association a, we need to use the pair $T_a = (a, X_a)$ to record the information, where a is the name of the association, X_a consists of the pairs of symbolic variables corresponding to the objects contained in the association a. If for a dynamic association a and transaction τ_j, the symbolic execution insert the pair of behavior partitions (b_m^j, b_n^j) into the association, then after the execution of τ_j, the corresponding pair of variables (v_m^j, v_n^j) is put into X_a. Suppose we have another transaction τ_i and the operation $checks(b_l^i, b_{l'}^i) \in a$, and the set X_a contains a family of variable pairs $\{(v_{m_1}^{j_1}, v_{n_1}^{j_1}), \cdots, (v_{m_k}^{j_k}, v_{n_k}^{j_k})\}$ before the execution of τ_i, then we should derive that at least one of the following pairs of equations should be satisfied:

$$v_l^i = v_{m_1}^{j_1}, v_{l'}^i = v_{n_1}^{j_1}$$
$$\cdots\cdots$$
$$v_l^i = v_{m_k}^{j_k}, v_{l'}^i = v_{n_k}^{j_k}$$

On the other hand, if the symbolic execution delete the pair of behavior partitions (b_m^j, b_n^j) from the association a, then we just need to check that the pair of variables should exist in X_a and remove it from X_a. Therefore we should derive a similar family of equation pairs as before and at least one of them should be satisfied.

From the previous discussion, we can find that for each transaction τ_i in the sequence $\tau_1, \tau_2, \cdots, \tau_n$, we have a set of equations E over the variables V_i as the

constraints for dynamic associations involved in the sequence. Furthermore, a family of equations E' is used to link the symbolic variables involved in different transactions. Thus to find a concrete execution of the transaction sequence equals to find a solution to the equations in $E \cup E'$ which satisfies the guard condition of the corresponding lifelines in the transaction sequences.

Theorem 2 *For a sequence of transactions $\tau_1, \cdots, \tau_n$, if $b^i_1, ..., b^i_{k_i}$ are the behavioral partitions for the different agents of transaction τ_i in the symbolic execution, there does not exist any run of $\tau_1, \cdots, \tau_n$ in the concrete system where the object playing the role of the j-th agent in transaction τ_i is in behavioral partition b^i_j for all i, j iff the set-constraints system we set up does not admit a solution.*

Proof: For a given symbolic simulation of the sequence of transactions $\tau_1, \tau_2, \cdots, \tau_n$, if there exists a corresponding concrete simulation, then we have a family of concrete objects which must satisfy the guard condition of the corresponding lifelines in each transaction to make the transaction be executed. First we consider the equations links different transactions. If object o is involved in the concrete execution, suppose in transaction τ_i it is the value of variable v^i_j, and in transaction τ_{i+1} it it the value of variable $v^{i+1}_{j'}$, then the behavioral partition $b^{i+1}_{j'}$ is the destination partition of b^i_j since object o moves from b^i_j to $b^{i+1}_{j'}$. Thus o is the value of both v^i_j and $v^{i+1}_{j'}$, so it satisfies the equation $v^i_j = v^{i+1}_{j'}$. By induction we can easily find that o satisfies a sequence of similar equations. Now suppose object o plays the role of the m-th agent in τ_j, i.e., o is in the behavioral partition b^j_m, then we first investigate the dynamic associations in τ_j in which b^j_m is involved. If the behavioral partition b^j_m is involved in some associations a in the symbolic simulation, i.e., for some behavioral partition b^j_n, we have $insert(b^j_m, b^j_n)$ *to* a in τ_j, where the concrete object o of b^j_m is associated to some object of behavioral partition b^j_n by a, then to make the transactions be enabled, if we have $checks(b^i_l, b^i_{l'}) \in a$ in some transaction τ_i after τ_j, then the object o should provide a value for both v^i_l and v^j_m which satisfies the corresponding equation $v^i_l = v^j_m$ in E. Therefore, the family of concrete object taking part in the concrete execution forms a solution of the set-constraints system.

We now consider another direction. If there exists a solution of the constraint system as given previously, let it be $v^i_{j_i} = o^i_{j_i}$ for $i = 1, 2, \cdots, n$ and $j_i = 1, 2, \cdots, k_i$. First of all, we can easily know that all these objects satisfy the guard conditions of the corresponding lifelines. Furthermore, if there is a family of dynamic associations in the symbolic execution for transaction τ_i and different associations links to the same behavioral partition b^i_j, then since $v^i_{j_i} = o^i_{j_i}$ forms a solution of the constraint system, all the equations as the corresponding constraints for dynamic associations should be satisfied if the variables $v^i_1, \cdots, v^i_{k_i}$ are substituted by $o^i_1, \cdots, o^i_{k_i}$. So all the dynamic associations are satisfied, each object o^i_j is associated to those objects which reside in the corresponding partitions being associated to b^i_j by these associations. Moreover, if two variables v^i_j and $v^{i'}_{j'}$ are equal to each other in the constraint system where $i \neq i'$ (which means they are not in the same transaction), $j \neq j'$, then they must correspond to the same concrete object because they have the same value (the object's identity). So o^i_j plays the role of the j-th agent in τ_i where it is in the behavioral partition b^i_j, and the same object plays the role of the j'-th agent

in transaction $\tau_{i'}$ where it is in the behavioral partition $b^{i'}_{j'}$, and we can easily build the destination partition relation between the different behavioral partitions. Therefore, the solution of the restrictions $E \cup E'$ provides the concrete objects that can execute the transaction sequence. $\square$

6 Benchmarks

We have implemented our symbolic execution method by building a simulator in *Ocaml*, a general purpose programming language supporting functional, imperative and object-oriented programming styles. We briefly discuss here the examples that we have modeled and used for experimenting with the simulator.

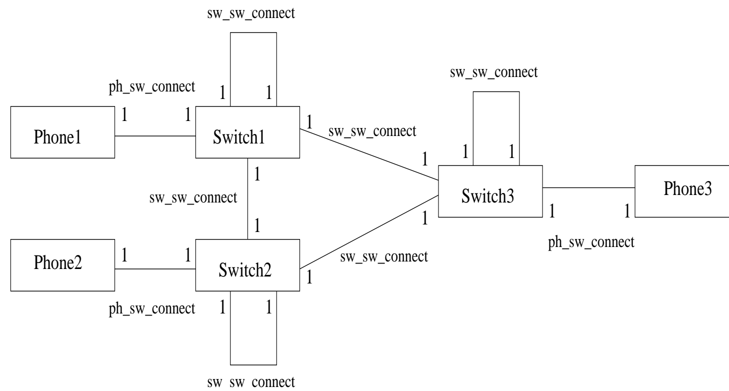

Figure 5: Class diagram for Telephone-switch example.

For initial experiments, we modeled a simple telephone switch drawn from [8]. It consists of a network of switch objects with the network topology showing the connection between different geographical localities. Switch objects in a locality are connected to phones in that locality as well other switches as dictated by the network topology. The network topology is represented using the class diagram as shown in Figure 5. To make a call, a phone connects to a switch in its area, which then establishes connection with another switch in the area being called. This second switch then sends the ring to called phone, and connection is established if the called phone is not busy. Note that a call made can be local (that is within the same area) or remote (from one area to another).

Besides the basic features of local/remote calling, we extended the model with call-waiting and three way calling features. This extension was done on top of the existing model by adding some extra states and transitions. These features allow a phone to be connected to two different phones simultaneously such that, it is in active connection with one phone and other phone is put on hold. This common phone can switch between the two by pressing flash button.

Next we modeled the rail-car system whose behavioral requirements have been specified using Statecharts in [5] and using Live Sequence Charts in [3]. The class diagram containing the main components of the system is shown in Figure 6. The numbers at the top right corner of each class box gives the number of objects of that class in our system. They can be changed easily with minor

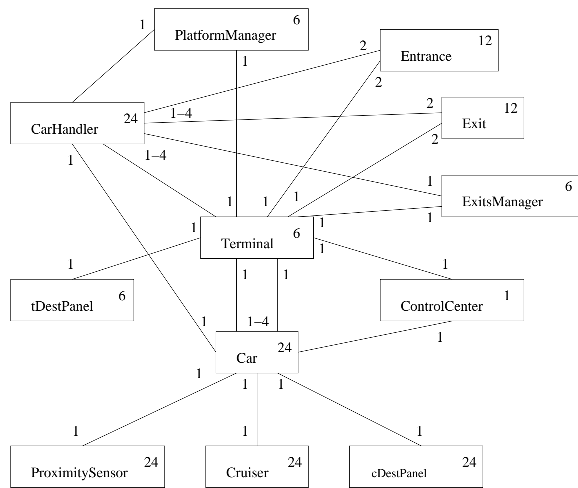

Figure 6: Class diagram for Railcar example.

modifications to the specification. This is an automated rail-car system with several cars operating on two parallel cyclic paths with several terminals, as shown in Figure 7.

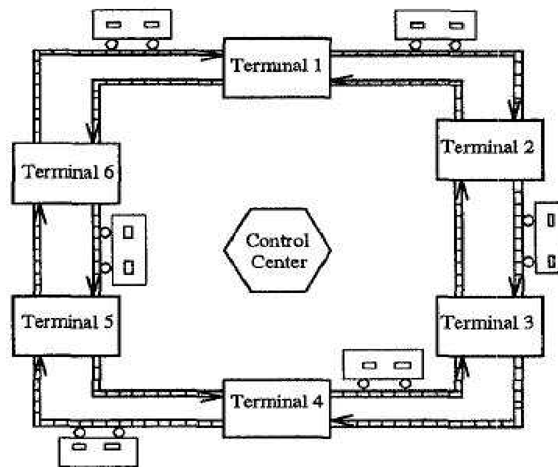

Figure 7: The cyclic path in the Railcar example.

The cars run clockwise on one of the cyclic paths and anti-clockwise direction on the other. This example is a substantial sized system with a number of components in different process classes, for instance a given system configuration could contain: 24 cars, 6 terminals, 24 cruisers (for maintaining speed of a rail-car), 0..24 car-handlers (a temporary interface between a car and a terminal while a car is in that terminal), etc.

We have also modeled the requirement specification of two other systems -

one drawn from the rail transportation domain and another taken from air traffic control. These systems have been proposed in the software engineering community as case studies for trying out reactive system modeling techniques (for example, see `http://scesm04.upb.de/case-studies.html`). We now briefly describe these two systems.

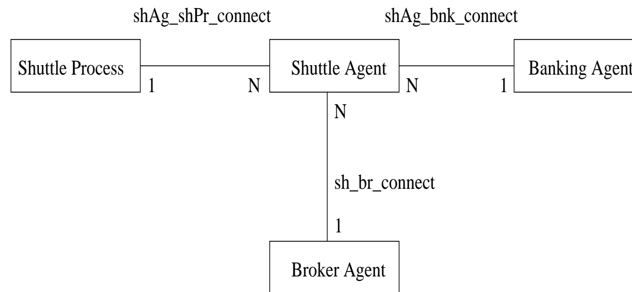

Figure 8: Class diagram for Automated shuttle example.

The automated rail-shuttle system [16] consists of various shuttles which bid for orders to transport passengers between various stations on a railway-interconnection network. The successful bidder needs to complete the order in a given time, for which it gets the payment as specified in the bid; the shuttle needs to pay the toll for the part of network it travels. If an order is delayed or not started in time, a pre-specified penalty is incurred by the responsible shuttle. A part of network may be disabled some times due to repair work, causing shuttles to take longer routes. A shuttle may need maintenance after travelling a specified distance, for which it needs to make payment. Also, in case a shuttle is bankrupt (due to payment of fines), it is retired. The class diagram for this system is shown in Figure 8, showing its main components.

Figure 9: Class diagram for Weather controller example.

The weather update controller [2] is a an important component of the *Center TRACON Automation System (CTAS)*, automation tools developed by NASA to manage high volume of arrival air traffic at large airports. The case study involves three classes of objects: weather-aware clients, weather control panel (WCP) and the controller or communications manager(CM). The class diagram is shown in Figure 9.

All the clients are initially disconnected from the weather control panel and can individually get connected via controller. The latest weather update is then presented by the weather control panel to various connected clients (again via the controller). This update may succeed when all these clients are able to receive and update themselves with the new weather information or, fail in case any of these clients is either unable to receive or unable to update itself using the weather update information. Furthermore, all connected clients get

disconnected in case any one of them fails to update itself as mentioned before.

7 Simulation Results

Our simulator can be used for random as well as guided simulation. We used guided simulation on each of our examples to test out the prominent use cases. We give an example of a use case from each of the examples below:

- Telephone-switch example: With call-waiting feature, a call is made and conversation starts between two phones, then a second call is made to one of the busy phones, invoking the call-waiting feature.
- Rail-car example: Cars move from source to destination terminal, without stopping at intermediate terminals.
- Automated rail-shuttle system: A shuttle successfully bids for an order, but is unable to complete it in given time, leading to payment of penalty.
- Weather-update controller: Some weather-aware clients connect to the weather controller and are successfully updated when new weather information is available.

We now summarize these simulation runs. For each example, we summarize the results of three test cases in Table 1.

We simulate the following test cases for the Rail-car example– (a) cars moving from a busy terminal to another busy terminal (*i.e.* a terminal where all the platforms are occupied, so an incoming car has to wait) while stopping at every terminal, (b) cars moving from a busy terminal to less busy terminals while stopping at every terminal, and (c) cars moving from one terminal to another while not stopping at certain intermediate terminals.

In the rail shuttle-system example, again we report the results for three test runs corresponding to (a) timely completion of order by shuttle leading to payment, (b) late completion of order leading to penalty, and (c) shuttle being unable to carry out order as it gets late in loading the order. Finally, for the weather update controller, we report the results of simulating three test cases corresponding to (a) successful update of latest weather information to all clients, (b) unsuccessful weather update where certain clients revert to older weather settings, and (c) unsuccessful update leading to disconnecting of clients.

The results from simulating all the above-mentioned test cases are reported in Table 1. For each test case of each example, we report the number of concrete objects for each process class as well as the maximum number of behavioral partitions observed during simulation. Of course, we have reported the results for only process classes with more than one concrete object. Since we are simulating reactive systems, we had to stop the simulation at some point; for each test case, we let the simulation run for 100 transactions – long enough to exhibit the test case's behavior.

We observe that the number of behavioral partitions (the groups into which objects are partitioned based on behaviors) is much less than the number of concrete objects. Furthermore, even if the number of concrete objects is increased (say instead of 24 cars in the Rail-car example, we have 48 cars), the number of behavioral partitions in these simulation runs remain the same.

Example	Process Class	# Concrete Objects	# of partitions in Test Case		
			I	II	III
Telephone Switch	Phone	60	9	9	7
	Switch	30	9	9	9
Weather Update	Clients	20	3	3	3
Rail Shuttle	Shuttle Agent	60	6	5	6
Rail-Car Example	Car	24	9	10	12
	CarHandler	24	4	5	5
	Terminal	6	6	6	6
	Platform Mngr.	6	1	3	4
	Exits Mngr.	6	2	2	2
	Entrance	12	1	2	2
	Exit	12	2	2	2
	Cruiser	24	2	4	4
	Proximity Sensor	24	2	2	2
	cDestPanel	24	1	1	1
	tDestPanel	6	1	1	1

Table 1: **Maximum Number of Behavioral partitions observed during symbolic simulation**

As mentioned earlier, at the heart of our symbolic simulation is the idea of a behavioral partition, which groups together objects with identical execution possibilities. And this is done without maintaining object identities or any other state-information related to these objects individually. Since, one of our main aim is to achieve a simulation strategy efficient in both time and memory, a possible concern is whether the management of behavioral partitions introduces unacceptable timing and memory overheads. We measured the timing and memory usage of several randomly generated simulation runs of length 1000(i.e. containing 1000 transactions) in our examples and considered the maximum result for each example. We also compared our results with a concrete simulator (where each concrete object's state is maintained separately). For meaningful comparison, the concrete simulator is also implemented in OCaml and shares as much code as possible with our symbolic simulator. Simulations were run on a Pentium-IV 3 Ghz machine with 1 GB of main memory.

The results for various examples appear in Table 2. For each example, the timing and memory usage is shown for both symbolic and concrete simulations. Also, for a given example, we compared the results for two different configurations, where 2nd configuration is obtained by doubling the number of one (or more) components; such as for *rail-car* example with 24 and 48 cars respectively.

Example	Time (sec)		Memory (MB)	
	Execution Type		Execution Type	
	Symbolic	Concrete	Symbolic	Concrete
RailCar-24cars	2.1	3.9	83	173
RailCar-48cars	2.2	7.0	84	353
Shuttle-30	0.44	0.7	18	33
Shuttle-60	0.44	1.2	18	69
WthrCon 10 Clients	0.5	0.6	18	21
WthrCon 20 Clients	0.5	0.8	18	27
SimpleSwitch 60ph,30sw	1.5	2.0	63	87
SimpleSwitch 120ph,60sw	1.5	4.1	64	189

Table 2: Experimental observations

We observe that for a given example and configuration, the running time and memory usage for the concrete simulator are higher than that for the symbolic simulator. It is in fact more interesting to notice that, for the same example, but with higher number of objects; in case of symbolic execution, the values remain roughly the same for both the configurations, whereas they almost double for the concrete case (except for Weather controller example[6]).

Further, the graphs shown in Figure 10, compare the growth in timing and memory usage respectively for the railcar example, for both concrete and symbolic simulations. Each successive configuration is obtain by increasing the number of cars and its associated components: car-handler, proximity-sensor,

[6]For the weather controller example, there is almost no concurrency during execution, since clients, which are multiple in number, always interact sequentially with the controller.

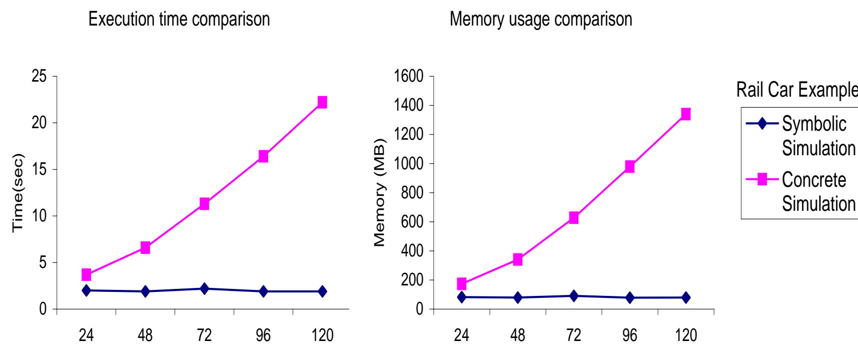

Figure 10: Execution Time and Memory Usage comparison for the Railcar example.

cruiser and dest-panel by 24.

We can easily see that the concrete simulator's timing and memory usage increases appreciably with an increase in number of objects. This is not the case for our symbolic simulator. This indicates one of the primary usefulness of our approach, since users can try out various configurations of the model varying greatly in the number of objects, without worrying about timing or memory overheads.

8 Debugging Experience

Finally, we describe some experiences in debugging the NASA's CTAS weather-update control system [2] using our simulator. As mentioned earlier, the weather-update control system consists of three process classes: the communications manager (call it CM), the weather control panel (call it WCP) and Clients. Both CM and WCP have only one object, while the Client class has many objects. In Figure 11, we show a snippet of the transition system for CM. We have given the transactions names to ease understanding, for example *Snd_Init_Wthr* stands for "send initial weather" and so on. We now discuss two bugs that we detected via simulation. The first one is an under-specification in the informal requirements document for the weather-update controller.

In Figure 11, the controller CM initially connects to one or more clients by executing the transactions *Connect* and *Snd_Init_Wthr*. In the *Connect* transaction CM disables the Weather Control Panel (WCP). If the client subsequently reports that that it did not receive the weather information (*i.e.* transaction *Not_Rcv_Init_Wthr* is executed), CM goes back to *Idle* state without re-enabling the Weather Control Panel (WCP). Hence no more weather-updates are possible at this stage. This results from an important under-specification of the weather-update controller's informal requirements document. This error came up in a natural way during our initial experiments involving random simulation. Simulation runs executing the sequence of transactions

Connect,Snd_Init_Wthr,Not_Rcv_Init_Wthr, Upd_from_WCP

got stuck and aborted as a result of which the simulator complained and provided the above sequence of transactions to us. From this sequence, we could easily fix the bug by finding out why *Upd_from_WCP* cannot be executed (*i.e.* the Weather Control Panel not being enabled). We note that since the above sequence constitutes a meaningful use-case we would have located the bug during guided simulation, even if it did not appear during random simulation. In this context it is worthwhile to mention that for every example, after modeling we ran random simulation followed by guided simulation of prominent test cases.

We found another bug during guided simulation of the test case where connected clients get disconnected from the controller CM since they cannot use the latest weather information. This corresponds to the connected clients executing the *Disconnect* transaction with the CM, and the CM returning from *Done2* to *Idle* by executing *Enable_WCP* (Figure 11). For this simulation run, even after all clients are disconnected, the CM executes *Upd_from_WCP* (update from Weather Control Panel) followed by *Rdy_for_PreUpd* (ready for pre-update). The simulator then gets stuck at the *PreUpd_Wthr* (pre-update weather) transaction since there are no connected clients. From this run, we found a missing

Figure 11: Snippet of Transition System for Weather-Update Controller

corner case in the guard for *Upd_from_WCP* transaction – no weather updates should take place if there are no connected clients. In this case, it was a bug in our modeling which was detected via simulation.

Currently, our simulator supports the following features to help error detection.

- Random simulation for a fixed number of transactions
- Guided simulation for an use-case (the entire sequence of transactions to be executed need not be given)
- Testing whether a given sequence of transactions is an allowed behavior.

9 Discussion and Future Work

In this paper, we have studied a modeling and simulation methodology for interacting process classes; such classes occur in reactive control systems of various application domains such as telecommunications and transportation. Our models are based on standard UML notations and our *symbolic* simulation strategy allows efficient simulation of realistic designs with large number of objects. The efficacy of our method for efficient simulation and debugging has been demonstrated on realistic reactive control systems.

We are currently working on automated test case generation from the given model of the system. The idea is to do explore the state space of the model such that all the transactions are executed at least once. This will serve two purposes:

(a) All the transactions since executed at least once will be verified against any specification inconsistencies (for example, no *receive* specified corresponding to a *send*), and (b) By examining the execution tree obtained so far, we can list out traces as meaningful test sequences for the final system implementation.

Further, to verify various properties of the system at model level itself, we are looking at model-checking the specification. The state-space of a model can be easily obtained by extending the simulator, using which various temporal properties can be verified. The main concern here would be keeping in check the memory consumption due to large state-spaces, which can easily become worse as we increase the number of objects in various process classes. This would require us to look for efficient state representations and techniques such as partial order reduction for reducing the search space itself.

Since we are modeling reactive systems, we would like to be able to specify timing constraints; for example, minimum or maximum time intervals between two events etc., and do various kinds of analysis to check against any timing inconsistencies.

Finally, we want to do automated code generation from our models, which would really strengthen the whole framework: starting from high level modeling and analysis facilities, leading to the final implementation of the system, which can be tested using various test-cases obtained from the model.

References

[1] F. Balarin et al. Metropolis: an integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003.

[2] CTAS. Center TRACON automation system. `http://www.ctas.arc.nasa.gov`.

[3] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 2001.

[4] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *International Conference on Computer Aided Verification (CAV)*, 2000.

[5] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7), 1997.

[6] D. Harel and O. Kupferman. On object systems and behavioral inheritance. *IEEE Transactions on Software Engineering*, 2002.

[7] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[8] G.J. Holzmann. *Modeling a Simple Telephone Switch*, chapter 14. The SPIN Model Checker. Addison-Wesley, 2004.

[9] E.A. Lee. Overview of the Ptolemy project. Technical report, University of California, Berkeley, 2003. Technical Memorandum UCB/ERL M03/25.

[10] E.A. Lee and S. Neuendorffer. Classes and subclasses in actor-oriented design. In *MEMOCODE, ACM Press*, 2004.

[11] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1994.

[12] O. Nierstasz. *Regular Types for Active Objects.* Object-oriented Software Composition. Prentice Hall, 1995.

[13] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In *International Conference on Computer Aided Verification (CAV)*, 2002.

[14] A. Roychoudhury and P.S. Thiagarajan. Communicating transaction processes. In *ACSD, IEEE Press*, 2003.

[15] B. Selic. Using UML for modeling complex real-time systems. In *LCTES, LNCS 1474*, 1998.

[16] Shuttle_Control_System. New rail-technology Paderborn. `http://wwwcs.uni-paderborn.de/cs/ag-schaefer/CaseStudies/ShuttleSystem`.

[17] Perdita Stevens. On the interpretation of binary associations in the Unified Modeling Language. *Journal on Software and Systems Modeling*, 1(1):68–79, 2002.

[18] T. Wang, A. Roychoudhury, R.H.C. Yap, and S.C. Choudhary. Symbolic execution of behavioral requirements. In *Practical Appl. of Declarative Languages (PADL), LNCS 3057*, 2004.

[19] H. Wehrheim. Behavioral subtyping relations for active objects. *Formal Methods in System Design*, 2003.