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

Technical Report A9/05
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 standard 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. We observe that during the initial system design phase it may be unnatural to fix the number of objects in each process class of the system. In general, it is also difficult to set a small cutoff number n_p on the number of objects for each process p, such that this restricted system is guaranteed to exhibit all the interesting behaviors of the intended system. This is our motivation for developing a modeling framework, where one can efficiently simulate and validate a system with a large number of active objects, such as a telephone switch network with thousands of phones, an air traffic controller with hundreds of clients etc. If the execution semantics of such systems maintains the local state of each object as simulation proceeds, this will lead to an impractical blow-up. Instead, we dynamically group together objects by maintaining sufficient -but bounded-information to ensure that the grouped objects will exhibit similar future behaviors.

We use labeled transition systems and class diagrams as a basis for our modeling framework – class diagrams are used to capture the associations between the process classes and transition systems 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.

2 Related Work

Simulation of scenario-based specifications as well as synthesis of executable models from such specifications is an important research area. The synthesis task may consist of realizing per-process transition systems from scenario-based specifications (see [4, 19] for example). Alternatively, one may develop executable specifications based on Message Sequence Charts (MSCs). Works in this direction include Live Sequence Charts [2], Triggered Message Sequence Charts [16] as well as our past work [14]. All these approaches deal with *concrete* objects and their interactions.

Live Sequence Charts (LSCs)[2, 6] offer an MSC-based inter-object modeling framework for reactive systems. However, LSCs completely suppress the control flow information for each process class. More importantly, though the objects of a process class can be *specified* symbolically, the LSC execution mechanism (the play-engine as described in [6]) does not support symbolic execution of process classes. The symbolic instances are instantiated to concrete objects during simulation.[1] The work on Triggered Message Sequence Charts [16] allows for a per-process execution semantics (in comparison to the play-engine of LSCs which gives a centralized execution semantics). Again, the execution semantics deals with concrete object interactions.

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.

[1] The approach taken in [20] alleviates this problem of LSCs by maintaining constraints on process identities but falls short of a fully symbolic execution.

These tools also have limited 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; Our present framework does not cater for structured classes but it can easily accommodate notions such as ports/interfaces. Indeed, our execution mechanism is easily applicable to a variety of related modeling styles.

Our technique for grouping together behaviorally similar objects is different from existing works on behavioral subtyping which develop subclass relationships based on behaviors of the objects in those classes. One of the early works in this area is by Liskov and Wing [11] which focuses on passive objects – objects whose state change is only via method invocation by other objects. Subsequently, behavioral subtyping of active objects have been studied in many works (*e.g.* [5, 21]). These works mostly exploit well-known notions of behavioral inclusion (such as trace containment) to define notions of behavioral subtyping. Our aim however is not to detect/establish subclass relationships. Rather, we wish to dynamically group together objects of the same class based on behavior exhibited so far for purposes of efficient system execution or simulation.

Our method of grouping together active objects is related to abstraction schemes developed for grouping processes in parameterized systems (*e.g.*, see [13]). In such systems, there are usually many similar processes whose behavior can be captured by a single finite state machine. It is then customary to maintain the count of number of processes in each state of the finite state machine; the names/identities of the individual processes are not maintained. However, in our setting inter-object associations across classes have to be maintained — an issue that does not arise in parameterized system validation.

The notion of "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 in [10]. We see this work as an orthogonal approach where the computational rather than the control flow features are encapsulated using classes and other object-oriented programming notions (such as inheritance).

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 name a transaction and the role played by an object of the class in the transaction. Message Sequence Charts (sequence diagrams) will be used to represent transactions. We fix a set of **transactions** Γ with γ ranging over Γ. Also, for each transaction $\gamma \in \Gamma$, let R_γ be the set of **roles** (usually called lifelines or instances of the chart) and $R = \cup_{\gamma \in \Gamma} R_\gamma$. Each role r in R will be a pair (p, ρ) where p is the name of a class -from which an object playing this role is to be drawn- and ρ is the chart role to be played by r ("sender", "receiver" etc.). For convenience we assume

that if (p_1, ρ_1) and (p_2, ρ_2) are two distinct members of R_γ (*i.e.*, two distinct roles in transaction γ), then $\rho_1 \neq \rho_2$. We however *do not* demand $p_1 \neq p_2$. Thus *two different roles in a transaction may be played by two objects drawn from the same class*.

More precisely, a transaction γ in our model is a guarded message sequence chart of the form $(I : Ch)$, where Ch is the Message Sequence Chart and I is the transaction *guard*: conjunction of guards, one for each *role* of Ch. We now describe the MSC Ch and its guard I in more detail.

Message Sequence Chart To describe an MSC, we define the following components:

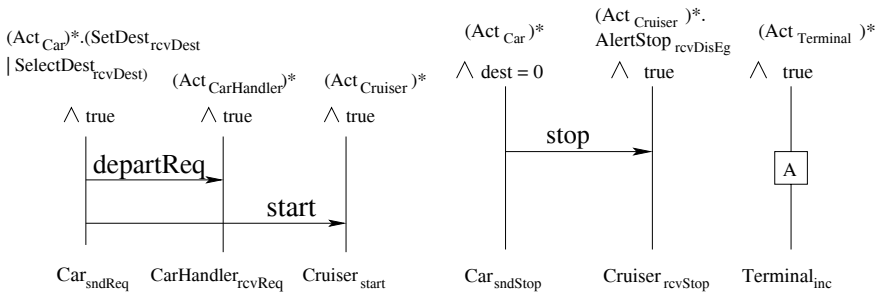
- M is an alphabet of message events.
- Act is an alphabet of internal actions representing computational steps performed by various lifelines.
- Let p, q range over R. Σ_p be the set of actions executed by $p \in R$ and let $\Sigma = \cup_{p \in R} \Sigma_p$. These actions are of the form: <p!q,m>- p sending message m to q, $\langle p?q, m\rangle$- p receiving a message m from q, and $\langle p, a\rangle$ denoting an internal action a of p (where $q \in R$). We assume that the message communication is over point to point reliable FIFOs.
- We define a Σ labeled poset to be a structure $Ch = (R', E, \leq, \lambda)$, where $R' \subseteq R$, $(E, \leq)$ is a poset and $\lambda : E \to \Sigma$ is a labeling function. Also let, $\downarrow(e) = \{e' \mid e' \leq e, \text{ for some } e \in E\}$ and l, r range over R'. We set $E_l = \{e \mid \lambda(e) \in \Sigma_l\}$. These are the events that l takes part in. Further, $E_{l!r} = \{e \mid e \in E_l \text{ and } \lambda(e) = \langle l!r, m\rangle \text{ for some } m \in M\}$. Similarly, $E_{l?r} = \{e \mid e \in E_l \text{ and } \lambda(e) = \langle l?r, m\rangle \text{ for some } m \in M\}$. Also, for any channel $c = (l, r)$, the communication relation R_c is defined as: $(e, e') \in R_c$ iff $|\downarrow(e) \cap E_{l!r}| = |\downarrow(e') \cap E_{r?l}|$ and $\lambda(e) = \langle l!r, m\rangle$ and $\lambda(e') = \langle r?l, m\rangle$ for some message m.

Definition 1 (Message Sequence Chart) *An MSC (over (R, M, Act)) is a Σ labeled poset $Ch = (R', \{E_p\}_{p \in R'}, \leq, \lambda)$, with l,r ranging over R', satisfying the following:*
(1) $\leq_l$ is a linear order for each l where $\leq_l$ is restricted $\leq$ to $E_l \times E_l$.
(2) Suppose $\lambda(e) = \langle l?r, m\rangle$. Then $|\downarrow(e) \cap E_{l?r}| = |\downarrow(e) \cap E_{r!l}|$.
(3) For every l,r with $l \neq r$, $|E_{l?r}| = |E_{r!l}|$.
(4) $\leq = (\leq_{R'} \cup R_{Chan})^\star$ where $\leq_{R'} = \cup_{l \in R'} \leq_l$ and $R_{Chan} = \cup_{c \in Chan} R_c$.

We assume synchronous concatenation of two MSCs. Also message communication is considered to be asynchronous; however other variants such as synchronous communication can be easily supported.

The Guard of a Transaction The guard I of transaction $\gamma = (I : Ch)$, $Ch = (R_\gamma, \{E_p\}_{p \in R_\gamma}, \leq, \lambda)$, is of the form $\{I_r\}_{r \in R_\gamma}$, i.e. a conjunction of the guards, one for each role in R_γ.

In a transaction, the **guard** associated with the role (p, ρ) will specify the conditions that must be satisfied by an object O_r belonging to the class p in order for it to be eligible to play the role $r = (p, \rho)$. These conditions will



(a) DepartReqA (b) NoMoreDest

Figure 1: Transactions *DepartReqA* & *NoMoreDest*; A is an internal computation event in transaction *NoMoreDest*

consist of two components: i) a *history* property of the execution sequence (of communication actions) that O_r has so far gone through ii) a *propositional formula* built from boolean predicates regarding the values of the (instantiated) variables owned by O_r. For instance, in the transaction "DepartReqA" (refer to Figure 1(a)), a Car object wishing to play the role (*Car, sndReq*)[2] must have last played the role (*Car, rcvDest*) in the transaction *SetDest* or in the transaction *SelectDest*. This is captured by the regular expression guard

$$Act^{\star}_{Car}.(SetDest_{rcvDest}|SelectDest_{rcvDest})$$

shown at the top of the *sndReq* lifeline in Figure 1(a). Thus, we will use regular expressions to specify the history component of a guard. Also, note that in the transaction "DepartReqA" (Figure 1(a)), the guard does not restrict the local variable valuation of participating objects in any way. On the other hand, in the transaction of Figure 1(b), the variable "dest" owned by the car-object intending to play the role (*Car,sndStop*) must satisfy "dest = 0".

The transition system describing the common control flow of all the objects belonging to the class p will be denoted as TS_p and it will be a structure of the form

$$TS_p = \langle S_p, Act_p, \rightarrow_p, init_p, V_p, v_{init_p} \rangle.$$

We first explain the nature of the components Act_p, V_p and v_{init_p}. The set of actions Act_p are the set of roles that the p-objects can play in the transactions in Γ. Accordingly, a member of Act_p will be a triple of the form (γ, p, ρ) with $\gamma \in \Gamma$, $\gamma = (I : Ch)$ and $r = (p, \rho) \in R$ where R is the set of roles of Ch. *Since role $r = (p, \rho)$, the action label (γ, p, ρ) will be abbreviated as γ_r; when p is clear from the context it can also be abbreviated as γ_ρ.* The computational steps performed by an object will be described with the help of the set of variables V_p associated with p. Each object O in p of course will have its own copy of the variables in V_p but for convenience of explanation we shall assume that all the objects of class p assign the same initial value to any variable $u \in V_p$. This

[2]As a notational shorthand we have written role (p, ρ) as p_ρ at the bottom of each lifeline in the transactions of Figure 1.

initial assignment is captured by the function v_{init_p} while assuming appropriate value domains for the variables in V_p. Since a computational step can be viewed as a degenerate type of transaction having just one role in its chart, we will not distinguish between computational and communication steps in what follows. Returning to $TS_p = \langle S_p, Act_p, \rightarrow_p, init_p, V_p, v_{init_p} \rangle$, S_p is the set of local states, $init_p \in S_p$ is the initial state and $\rightarrow_p \subseteq S_p \times Act_p \times S_p$ is the transition relation. In summary, our model can be defined as follows.

Definition 2 (The IPC Model) *Given a set $\mathcal{P}$ of process-classes, a set Γ of transactions and a set of action labels Act_p for $p \in \mathcal{P}$ involving transactions from Γ, a system of Interacting Process Classes (IPC) is a collection of $\mathcal{P}$-indexed labeled transition systems $\{TS_p\}_{p \in \mathcal{P}}$ where*

$$TS_p = \langle S_p, Act_p, \rightarrow_p, init_p, V_p, v_{init_p} \rangle$$

is a finite state transition system as explained above.

3.2 Class Diagram and Associations

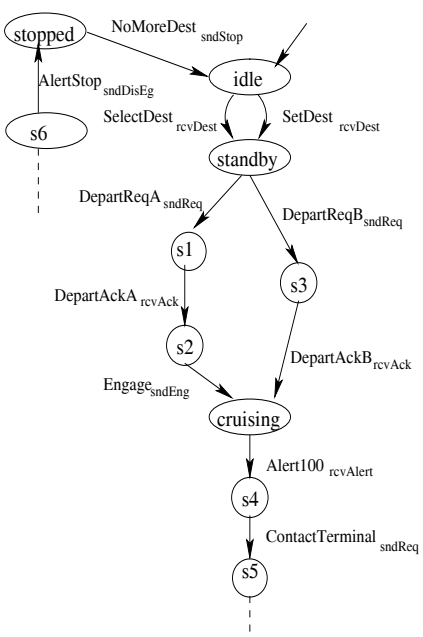
We make use of *class-diagrams* to model the system structure and *associations* to model different kinds of communication links that can exist in an interaction among objects. These extensions are crucial for achieving adequate modeling power. The associations are classified as static or dynamic in a manner similar to as presented in [18].

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 at both the association ends. Static associations between the objects remain fixed and do not change at runtime, *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.

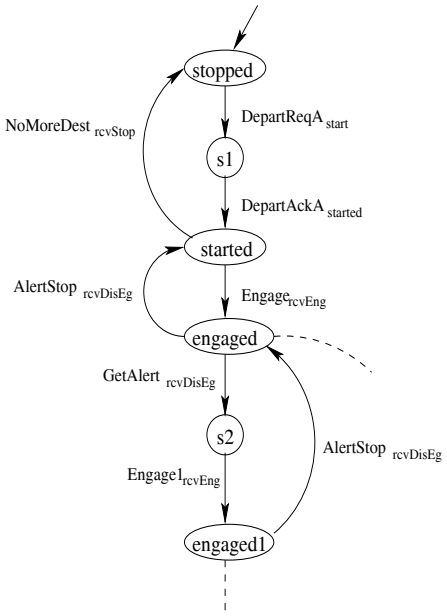
Dynamic Associations A dynamic association expresses *behavioral relationship* between the classes, which in our case implies that the objects of two dynamically associated classes can become related to each other during simulation for some period of time, exchange messages (by executing transactions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

4 Execution Semantics

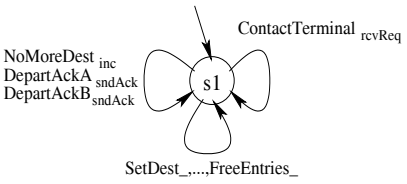
In this section, we describe the execution semantics of our IPC model. At the *initial configuration*, for each class p, every p-object will be residing at the designated initial state of TS_p. The history of each such object will be the null string and for each variable associated with p, each object of p will initialize it to the same value. The system will move from the current configuration by



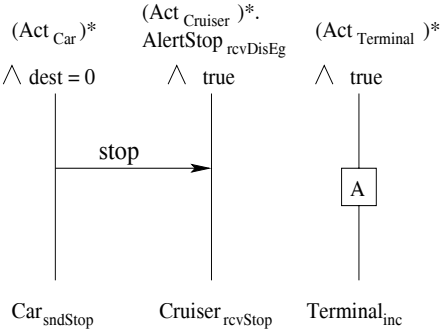
a) TS $_{\text{Car}}$



b) TS $_{\text{Cruiser}}$



c) TS $_{\text{Terminal}}$



d) Transaction *NoMoreDest*

Figure 2: State Diagrams for *Car*, *Cruiser* & *Terminal* and transaction *NoMoreDest*.

executing an enabled transaction and as a result, move to a new configuration. The transaction $\gamma = (I : Ch)$ is *enabled* at the configuration c if we can assign to each role $r = (p, \rho)$ of Ch, a *distinct* object O_r belonging to p such that the following conditions are satisfied (we will state these conditions informally and illustrate them via the example shown in Figure 2):

- First, the object O_r must reside at a state s in TS_p such that there is transition $s \xrightarrow{\gamma_r} s'$ in TS_p. (This object will move to s' when γ executes at c).
- Next suppose the guard I in $\gamma = (I : Ch)$ is of the form $\{I_r\}_{r \in R_\gamma}$.Furthermore, assume that $I_r = (\Lambda, \Psi)$ where Λ is a regular expression over alphabet Act_p and Ψ is a propositional formula constructed from some boolean predicates over the variables associated with p. Then σ, the current history of O_r (*i.e.*, sequence of actions executed by O_r), must be in the language defined by the regular expression Λ. Furthermore, the valuation of the variables of O_r should satisfy the formula Ψ.

If both these conditions are satisfied for an object O_r for each role r, then the transaction γ can occur at c. This will result in a new configuration c' obtained by updating current control locations, current history and the values of the variables of the objects O_r for each role r. In the example shown in Figure 2, suppose c is a configuration at which

- Two *Car* objects Oc_1 and Oc_2 are residing in state *stopped* and a third object, Oc_3, is in state $s2$ of TS_{Car}. Further suppose they have the values 0, 1 and 2 respectively for the variable *dest*.
- Three *Cruiser* objects, $O_1 \ldots O_3$ are residing in state *started* of $TS_{Cruiser}$ such that the history of O_1 and O_2 satisfy the regular expression
$$(Act_{Cruiser})^\star.AlertStop_{rcvDisEg}$$
while the history of O_3 satisfies the regular expression
$$(Act_{Cruiser})^\star.DepartAckA_{started}$$
- Six *Terminal* objects, $Ot_1 \ldots Ot_6$ are residing in state $s1$ of $TS_{Terminal}$.

Suppose we want to execute transaction *NoMoreDest* — shown in Figure 2(d) — at configuration c. As for the role (*Car,sndStop*), though Oc_1 and Oc_2 are in the appropriate control state, only Oc_1 can be chosen since it (and not Oc_2) satisfies the guard $dest = 0$. For the cruisers, we observe that all the three *Cruiser* objects O_1, O_2, O_3 are in the "appropriate" control state at configuration c for the purpose of executing *NoMoreDest*. However, only O_1 and O_2 have histories which satisfies the history part of the guard associated with the role (*Cruiser,rcvStop*). Hence either one of them (but not O_3) can be chosen to play this role. For the role (*Terminal,inc*), both the history and propositional guards are vacuous and hence we can choose any one of the 6 objects residing in the control state $s1$.

Assume that Oc_1, O_1 and Ot_1 are chosen to execute transaction *NoMoreDest* in configuration c. In the resulting configuration c', all objects other than

Oc_1, O_1 and Ot_1 will have their control states and histories unchanged from c. The objects Oc_1, O_1, Ot_1 will reside in states *idle*, *stopped*, $s1$ respectively. The history of Oc_1, O_1, Ot_1 will be obtained by appending $NoMoreDest_{sndStop}$, $NoMoreDest_{rcvStop}$ and $NoMoreDest_{inc}$ to their respective histories at configuration c. Object Ot_1 also updates a local variable via an *internal* event — refer to $\text{Terminal}_{\text{inc}}$ in Figure 2(d).

Execution in the presence of *associations* In the case of *static associations*, the only additional restriction to the execution semantics (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 *DepartAckA* or *DepartAckB* (refer to Figure 2(a)). Hence, the guard of *DepartAck(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. Instead of giving details of the *ContactTerminal* and *DepartAck(A/B)* transactions, we list here relevant roles of these transactions.

- *ContactTerminal* has roles *(Car,sndReq)* and *(Terminal,rcvReq)*,
- *DepartAckA* and *DepartAckB* have roles *(Car,rcvAck)* and *(Terminal,sndAck)*. Transactions *DepartAck(A/B)* also involve other roles which we choose to ignore here for the purpose of our discussion.

If car object O_1 and terminal object O_2 play the roles *(Car,sndReq)* and *(Terminal,rcvReq)* in *ContactTerminal* (refer to Figure 2(a) for TS_{Car} and Figure 2(c) for $TS_{Terminal}$), then the effect of *ContactTerminal* is to insert the tuple (O_1, O_2) into the *ItsTerminal* relation (refer to Figure 3). The *DepartAck(A/B)* transaction's guard now includes the check that the object pair corresponding to roles *(Car,rcvAck)* and *(Terminal,sndAck)*, be related by the dynamic relation *ItsTerminal*; so if objects O_1 and O_2 are 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 (O_1, O_2) 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

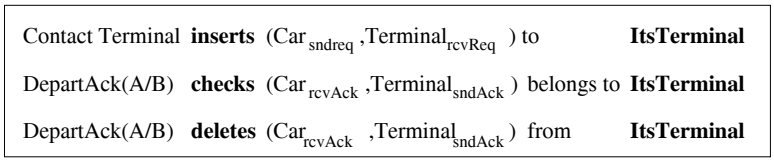


Figure 3: **Dynamic Relation *ItsTerminal***

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.

Given an IPC model as defined in the previous section, for the class p we define H_p to be the least set of DFAs given by: $\mathcal{A}$ is in H_p iff there exists a transaction $\gamma = (I : Ch)$ and a role r of Ch of the form (p, ρ) such that the guard I_r of r is (Λ, Ψ) and $\mathcal{A}$ is the minimal DFA recognizing the language defined by the regular expression Λ, the history part of the guard. The notion of behavioral partitions can now be defined as follows.

Definition 3 (Behavioral Partition) *Let* $\{TS_p = \langle S_p, Act_p, \rightarrow_p, init_p, V_p, v_{init_p}\rangle\}_{p\in\mathcal{P}}$ *be an IPC. Let* $H_p = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ *be the set of minimal DFAs defined for class* p *as described in the preceding. Then a* behavioral partition beh_p *of class* p *is a tuple* $(s, q_1, \ldots, q_k, v)$*, where*

$$s \in S_p, q_1 \in Q_1, \ldots, q_k \in Q_k, v \in Val(V_p).$$

Q_i *is the set of states of automaton* $\mathcal{A}_i$ *and* $Val(V_p)$ *is the set of all possible valuations of variables* V_p*. We use* BEH_p *to denote the set of all behavioral partitions of class* p*.*

We also define *initial* behavioral partitions representing the initial state of an IPC model.

Definition 4 (Initial Behavioral Partition) *Let* $\{TS_p = \langle S_p, Act_p, \rightarrow_p, init_p, V_p, v_{init_p}\rangle\}_{p\in\mathcal{P}}$ *be an IPC. Let* $H_p = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ *be the set of minimal DFAs defined for class* p *as described in the preceding and* q_i^{init} *be the initial state of* $\mathcal{A}_i$*. Then the* initial behavioral partition beh_p^{init} *of class* p *is the behavioral partition* $(init_p, q_1^{init}, \ldots, q_k^{init}, v_{init_p})$*.*

Now suppose c is a configuration and the object O belonging to the class p has the history $\sigma \in Act_p^\star$ at c and the valuation of its variables is given by the function v_O. We will say that at c, the object O *belongs* to the behavioral partition $(s, q_1, \ldots, q_k, v)$ in case O resides in s at c and q_j is the state reached in the DFA $\mathcal{A}_j$ when it runs over σ for each j in $\{1, \ldots, k\}$. Furthermore, the valuation of O's local variables is given by v. Thus, two p-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
- Their current histories lead to the same state for all the DFAs in H_p.

This implies that the computation trees of two objects in the same behavioral partition at a configuration are isomorphic. This is a strong type of behavioral equivalence to demand. There are many weaker possibilities but we will not explore them here.

Bounding the number of Behavioral Partitions We shall assume in what follows that the value domains of all the variables are finite sets. Thus, the number of behavioral partitions of a process class is finite. In fact, the number of partitions of a process class p is bounded by

$$|S_p| \times |Val(V_p)| \times \prod_{\mathcal{A} \in H_p} |\mathcal{A}|$$

where $|S_p|$ is the number of states of TS_p, $|Val(V_p)|$ is the number of all possible valuations of variables V_p, $|\mathcal{A}|$ is the number of states of automaton $\mathcal{A} \in H_p$. As described in the preceding, H_p is the set of minimal DFAs accepting the regular expression guards of the various roles of different transactions played by class p. Note that the maximum number of behavioral partitions does not depend on the number of objects in a class. In practice, many regular expression guards of transactions are vacuous leading to a small number of partitions.

For example, consider $TS_{Cruiser}$ as shown in Figure 2(b); it contains 7 states, i.e. $|S_{Cruiser}| = 7$ (one state and its associated transitions are not shown in the figure to maintain clarity), and the description of the transaction *NoMoreDest* shown in Figure 2(d). 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:

- $q1$ which is the *initial state* accepting the regular language

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

.

- $q2$ 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.[3]

[3]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.

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 in fact in Section 7 we report experiments that the number of behavioral partitions encountered in actual simulation runs is often lower than the upper bound on number of partitions (48 Cruiser objects are divided into less than 6 partitions, see Table 1).

Simulation example with behavioral partitions Consider $TS_{Cruiser}$ shown in Figure 2(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 $q1$ of DFA1. Thus they are in the same behavioral partition $\langle stopped, q1\rangle$, where we have suppressed the valuation component since there are no local variables associated with this class in this example.

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 $q2$ 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 $q1$ 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. $\langle stopped, q1\rangle$ has 22 objects which were idle.
2. $\langle started, q2\rangle$ has object $O1$.
3. $\langle started, q1\rangle$ has object $O2$.

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 action $NoMoreDest_{rcvStop}$. 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 $q1$ 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 an "abstract configuration".

Definition 5 (Abstract Configuration) *Let $\{TS_p\}_{p\in\mathcal{P}}$ be an IPC specification such that each process class p contains N_p objects. An abstract configuration of the IPC is defined as follows.*

$$\mathsf{cfg} = \{(BEH_p, count_p)\}_{p\in\mathcal{P}}$$

- *BEH_p is the set of all behavioral partitions of class p.*
- *$count_p : BEH_p \to \mathbb{N} \cup \{0\}$ is a mapping s.t.*

$$\Sigma_{b\in BEH_p} count_p(b) = N_p$$

$count_p(b)$ is the number of objects in partition b.

The set of all configurations of an IPC $\mathcal{S}$ is denoted as $\mathcal{C}_\mathcal{S}$.

We note that N_p can be a given positive integer constant or it can be ω (standing for unbounded number of objects). If N_p is ω, our operational semantics remains unchanged provided we assume the usual rules of addition/subtraction (i.e. $\omega + 1 = \omega$, $\omega - 1 = \omega$ and so on). Hence for convenience of explanation, we assume that N_p is a given constant in the rest of the article

Our symbolic simulation efficiently keeps track of the objects in various process classes by maintaining the current abstract configuration; only the behavioral partitions with non-zero counts are kept track of. The system moves from one abstract configuration to another by executing a transaction. In what follows, for the sake of convenience we shall often drop the "abstract" when talking about "abstract configurations". How can our simulator check whether a specific transaction γ is enabled at the current configuration cfg? We say that γ is enabled at cfg 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 role, from which an object can be chosen.

Definition 6 (Witness partition) *Let $\gamma \in \Gamma$ be a transaction and $\mathsf{cfg} \in \mathcal{C}_\mathcal{S}$ be a configuration. For a role $r = (p, \rho)$ of γ where r has the guard (Λ, Ψ), we say that a behavioral partition $beh = (s, q_1, \ldots, q_k, v)$ is a witness partition, denoted as $witness(r, \gamma, \mathsf{cfg})$, for r at cfg if*

1. *$s \xrightarrow{(\gamma_r)} s'$ is a transition in TS_p*
2. *For all $1 \le i \le k$, if $\mathcal{A}_i$ is the DFA corresponding to the regular expression of Λ, then q_i is an accepting state of $\mathcal{A}_i$.*
3. *$v \in Val(V_p)$ satisfies the propositional guard Ψ.*
4. *$count_p(b) \neq 0$, that is there is at least one object in this partition in the configuration cfg.*

An "enabled transaction" can now be defined as follows.

Definition 7 (Enabled Transaction) *Let γ be a transaction and $\mathsf{cfg} \in \mathcal{C}_{\mathcal{S}}$ be a configuration. We say that γ is enabled at cfg iff for each role $r = (p, \rho)$ of γ, there exists a witness partition $witness(r, \gamma, \mathsf{cfg})$ such that*

- *If $beh \in BEH_p$ is assigned as witness partition of n roles in γ, then $count_p(b) \geq n$. This ensures that one object does not play multiple roles in a transaction.*

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

The "destination partition" — the partition to which an object moves from its "witness partition" after executing a transaction — can be defined as follows. We denote the destination partition of *beh* w.r.t. to transaction γ and role r as $beh' = dest(beh, \gamma, r)$. Thus, an object in behavioral partition *beh* moves to partition $dest(beh, \gamma, r)$ by performing role r in transaction γ, where $r = (p, \rho)$ is a role in γ.

Definition 8 (Destination Partition) *Let γ be an enabled transaction at configuration $\mathsf{cfg} \in \mathcal{C}_{\mathcal{S}}$ and $beh = (s, q_1, \ldots, q_k, v)$ be the witness partition for the role $r = (p, \rho)$ of γ. Then we define $dest(beh, \gamma, r)$ — the destination partition of beh w.r.t. transaction γ and role r — as a behavioral partition $beh' = (s', q'_1, \ldots, q'_k, v')$, where*

- *$s \xrightarrow{(\gamma_r)} s'$ is a transition in TS_p.*
- *for all $1 \leq i \leq k, q_i \xrightarrow{(\gamma_r)} q'_i$ is a transition in DFA $\mathcal{A}_i$.*
- *$v' \in Val(V_p)$ is the effect of executing γ_r on v.*

Finally, we 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 beh' for each behavioral partition *beh* of a process class at cfg and then computing the new count of objects for each beh'.

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'.

$$\frac{\begin{array}{c}\mathsf{cfg} = \{(BEH_p, count_p)\}_{p \in \mathcal{P}} \\ \gamma \in En(\mathsf{cfg}) \\ \forall b \in BEH_p \,.\, count'_p(b) = count_p(b) + |\{x \mid b = dest(w, \gamma, x)\}| \\ -|\{x \mid b = w\}| \\ where\ w = witness(x, \gamma, \mathsf{cfg}) \\ \mathsf{cfg}' = \{(BEH_p, count'_p)\}_{p \in \mathcal{P}}\end{array}}{\mathsf{cfg} \xrightarrow{\gamma} \mathsf{cfg}'}$$

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 or static 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 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.

In case of static associations we do not by default enter the partitions containing the statically associated objects into respective association relations. Thus, initially there will be *no partition pairs* in any static associations, but they are discovered and inserted on-the-fly during execution. When a membership constraint for an object pair to be in static association is encountered during simulation, the *destination partitions* of the participating objects are inserted in the corresponding relation. Unlike for dynamic associations, while selecting the *witness partitions* for static associations we can choose either a pair already present in the association relation (inserted during an earlier check), or allow for a fresh pair of partitions to play the role.

Note that associations are maintained between behavioral partitions, but associations are not used to define behavioral partitions. Hence there is no blow-up in the number of behavioral partitions due to associations.

We now give the operational semantic rule taking into account various associations. Let $\alpha_a(\mathsf{cfg})$ denote the content of association a at configuration cfg. We use:

(a) "'$Inserts(x_1, x_2)$ to a" to represent insertion of object pair playing the roles x_1, x_2 into a.

(b) "$Deletes(x_1, x_2)$ from a" to represent removal of object pair playing the roles x_1, x_2 from a.

(c) "$Check((x_1, x_2) \in a)$" constrains the choice of object pair for roles x_1, x_2 to be in relation a.

$$\frac{\begin{array}{c}\mathsf{cfg} = \{(BEH_p, count_p\}_{p\in\mathcal{P}} \\ \gamma \in En(\mathsf{cfg}) \\ \forall b \in BEH_p \;.\; count'_p(b) = cnt_p(b) + |\{x \mid b = dest(w,\gamma,x)\}| \\ -|\{x \mid b = w\}| \\ where\ w = witness(x,\gamma,\mathsf{cfg}) \\ \mathsf{cfg}' = \{(BEH_p, count'_p\}_{p\in\mathcal{P}} \\ \forall a \in A \;.\; \alpha_a(\mathsf{cfg}') = \alpha_a(\mathsf{cfg}) \oplus \\ (\bigcup_{Inserts(x_1,x_2)\ to\ a}\{(d1,d2)\} \bigcup_{Check((x_1,x_2)\in a}\{(d1,d2)\}) \\ \ominus \bigcup_{Deletes(x_1,x_2)\ from\ a}\{(w_1,w_2)\} \\ where\ x_i \in agents(\gamma) \wedge w_i = witness(x_i,\gamma,\mathsf{cfg}) \wedge d_i = dest(w_i,\gamma,x_i)\end{array}}{\mathsf{cfg} \xrightarrow{\gamma} \mathsf{cfg}'}$$

Here the operator $\oplus(\ominus)$ describes the action of inserting(deleting) behavioral partitions from association relations as described below. If the pair (x_1, x_2) is inserted into association a when executing γ, then the pair of destination partitions (d_1, d_2) corresponding to the witness partitions (w_1, w_2) of (x_1,x_2) is inserted into α_a at the new configuration (if it is not already present). On the other hand, if the pair is deleted from the association a, we do not add (d_1, d_2) to α_a. When there is a check for the pair of roles (x_1, x_2) to be related by an association a, then (w_1, w_2) must belong to α_a in case of dynamic associations. For static associations, we can either choose witness partitions (w_1, w_2), s.t. $(w_1, w_2) \in \alpha_a$, or select a fresh pair of witness partitions w'_1, w'_2 (i.e. $(w'_1, w'_2) \notin \alpha_a$ and satisfies other requirements to qualify as witness partitions). Moreover, for both static and dynamic associations, the destination partition pair (d_1, d_2) is inserted into α_a. Finally, we remove any (w_1, w_2) from the α_a when the number of remaining objects in either of the witness partitions is 0.

Example As discussed in Section 4, dynamic relation *ItsTerminal* is maintained between the objects of class *Car* and *Terminal* (as shown in Figure 3). This relationship is established between a *Car* and a *Terminal* object while executing *ContactTerminal* and exists till the related pair executes either *DepartAckA* or *DepartAckB*. For illustration, suppose one object each from class *Car* and class *Terminal* plays the role (*Car,sndReq*) and (*Terminal,rcvReq*) respectively 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*) respectively. 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 at-least 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

It turns out, that due to the presence of associations, there can be symbolic executions which do not correspond to concrete executions. However every concrete execution can be realized as a symbolic execution. In this sense, our symbolic execution semantics is sound as stated next.

Theorem 1 *Suppose σ is a sequence of transactions that can be executed by the IPC, $\mathcal{S}$. Then σ can also be exhibited in the symbolic execution of $\mathcal{S}$.*

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 of process class p which will take part in the transaction τ, being in control state s and DFA states $q_1, \cdots, q_k$, and state valuation v, we can construct a behavioral partition $b = (s, q_1, \cdots, q_k, v)$, which contains at least the objects that has control state s and satisfies the corresponding guard condition of τ. From Definition 7, we know that $count_p(b)$ 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 6 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 7. 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 partitions in the symbolic execution. In fact, we can construct the behavioral partitions corresponding to O_1 and O_2 by using their control states, DFA states and variable valuations, and choose these partitions as witness partitions, later entering the destination partitions of the participating objects in static association relation

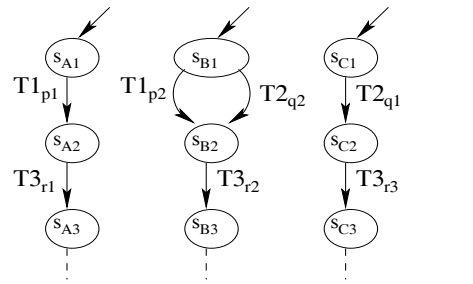
as discussed in Section 4.3. So the symbolic execution is still correct w.r.t. the concrete execution.

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 objects in these partitions 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 partitions (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 partitions to which its witness partition 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?

In the absence of associations we prove that, corresponding to a symbolic execution run we can always construct a concrete execution run. We prove it by induction on the length of symbolic execution run $\sigma = \sigma_1 \circ \tau$. For length $=$ 0, the result trivially holds. Assume that result holds for the execution trace up-to σ_1, i.e. σ_1 is exhibited by both symbolic and concrete runs. Now, since τ can execute in the symbolic run, we have an assignment of witness partitions $b = (s, q_1, \ldots, q_k, v)$ for each lifeline l in τ. This implies that we can always find at-least one concrete object O corresponding to behavioral partition b in control state s, DFA states $q_1, \ldots, q_k$ and having variable valuation v such that O satisfies the guard of l as well. Hence O can be chosen to play the role of lifeline l in the concrete execution. Thus, if τ executes symbolically, we can always find an assignment of objects to various lifelines in τ which can then be executed in concrete setting. This completes our proof.

However, in the presence of associations we show that it is not always possible that a concrete execution can be found for a behavior in the symbolic execution. We now give an example involving static associations in which a behavior observed in the symbolic simulation does not correspond to any actual system run. Consider the system consisting of 3 process classes: *Cruiser*, *Car* and *BrakeControl*, such that each *Cruiser* and *BrakeControl* object is associated with a *Car* object via static associations *Asc*1 and *Asc*2. The partial transition systems for these components are shown in Figure 4, along with the checks on the static associations by various transactions. Assume that there



T1	**checks**	(p1,p2)	is in	**Asc1**
T2	**checks**	(q1,q2)	is in	**Asc2**
T3	**checks**	(r1,r2)	is in	**Asc1**
	and	(r3,r2)	is in	**Asc2**

Figure 4: An example

are no variables declared in these process classes and that all the action labels shown in the example have trivial guards, i.e. they do not impose any restriction on the execution history of the object to play that lifeline (of course object should be in appropriate control state). Suppose now, that we have an initial configuration:

$$c = \{(\langle s_{A1}\rangle, 2), (\langle s_{B1}\rangle, 2), (\langle s_{C1}\rangle, 2)\}.$$

Each process class:*Cruiser*, *Car* and *BrakeControl*, contains 2 objects in their initial states: s_{A1}, s_{B1} and s_{C1} respectively. The given system then executes the trace: $T1.T2.T3$ in a symbolic run. For each transaction executed we give below the witness and destination partitions. We use w_{γ_r} to denote the witness partition for role r in transaction γ.

a) **T1** executes at configuration **c** as given above with:
$w_{T1_{p1}} = \langle s_{A1}\rangle$, $w_{T1_{p2}} = \langle s_{B1}\rangle$
with resulting **destination partitions**-
$\langle s_{A2}\rangle = dest(\langle s_{A1}\rangle, T1, (Cruiser, p1))$, $\langle s_{B2}\rangle = dest(\langle s_{B1}\rangle, T1, (Car, p2))$
and resulting **configuration**-
$c1 = \{(\langle s_{A1}\rangle, 1), (\langle s_{A2}\rangle, 1), (\langle s_{B1}\rangle, 1), (\langle s_{B2}\rangle, 1), (\langle s_{C1}\rangle, 2)\}$

b) Then **T2** executes at configuration **c**1 with:
$w_{T2_{q1}} = \langle s_{C1}\rangle$, $w_{T2_{q2}} = \langle s_{B1}\rangle$
with resulting **destination partitions**-
$\langle s_{C2}\rangle = dest(\langle s_{C1}\rangle, T2, (BrakeControl, q1))$, $\langle s_{B2}\rangle = dest(\langle s_{B1}\rangle, T2, (Car, q2))$
and resulting **configuration**-
$c2 = \{(\langle s_{A1}\rangle, 1), (\langle s_{A2}\rangle, 1), (\langle s_{B2}\rangle, 2), (\langle s_{C1}\rangle, 1), (\langle s_{C2}\rangle, 1)\}$.

At this point we note that the *Car* objects playing roles in $T1$ and $T2$ can never be the same, since after playing role $T1_{p2}$, the control state in the destination partition to which this object moves will be s_{B2}: $T2_{q2}$ is never enabled from this control state for a *Car* object.

c) Finally **T3** executes at configuration **c**2 with:
$w_{T3_{r1}} = \langle s_{A2}\rangle$, $w_{T3_{r2}} = \langle s_{B2}\rangle$, $w_{T3_{r3}} = \langle s_{C2}\rangle$

One can easily verify that $T1$ and $T2$ and $T3$ are enabled at the given configurations (following Definition 7) and also satisfy the static association constraints, i.e. the witness partitions for the roles which need to be statically associated contain at least one such pair of objects.

However, as we can observe, though all the constraints in the symbolic simulation have been met, the Car objects playing the roles $T1_{p2}$ and $T2_{q2}$ are always distinct. In other words there is no object $O' \in Car$, such that $(O', O_{Cruiser}) \in Asc_1$ and $(O', O_{BrakeControl}) \in Asc_2$ when $T3$ is executed. Though in this example we have only considered static associations, we can easily extend our example to involve dynamic associations as well.

5 Checking a Symbolic Execution Run

From the discussion in the previous section, we know that there may exist symbolic execution runs that do not correspond to any concrete execution. Thus, we need a mechanism that can efficiently detect such spurious symbolic executions. This is similar to detecting spurious counter-example traces in abstraction-refinement based software model checking (e.g. see [7]). Fortunately, one can effectively check in our setting if a symbolic execution run σ corresponds to a concrete run as follows. For each process class p, let $num_{p,\sigma}$ be the total number of roles having the *initial partition* (refer to Definition 4) as the witness partition in the transactions appearing in σ. We define $x_{p,\sigma} = min(N_p, num_{p,\sigma})$ if N_p, the number of objects in p is a given constant. Otherwise the number of objects of p is not fixed and we set $x_{p,\sigma} = num_{p,\sigma}$. It is worth noting that $x_{p,\sigma}$ serves as a cutoff on the number of objects of class p only for the purpose of exhibiting the behavior σ and not all the behaviors of the system. Clearly, σ is a concrete run in the given system iff it is a concrete run in the *finite state* system where each process class p has $x_{p,\sigma}$ objects.

We have implemented the above spuriousness check using the Murphi model checker [12]. The reason for using Murphi is that it has in-built support for symmetry reduction [9]; this can speed up model checking of process classes with many similar processes. Such systems often exhibit structural symmetry which can be exploited to avoid constructing/traversing the full state space. This is achieved by using the *scalarset* data type, which makes it easy to detect and exploit symmetry in the system structure. In the case of IPC models we identify various objects of a process class using the scalarset type. We now briefly discuss the our translation from an IPC model into Murphi input format for checking a spurious trace.

IPC translation to Murphi In the Murphi translation we define the following data types for each process class:

- A *scalarset* type to act as an object identifier having the cut-off number as its upper limit. For example, for Car class containing N objects, following type will be declared:

  Car: Scalarset(N); –index for process class Car

- Enumeration variable types which define *sets of states* of its LTS and various DFAs. Assuming that the LTS of process class Car contains M

states and one of its DFAs, say dfa_i has D_i states, the following translation will result:

stCar: Enum {st_car1,...,st_carM}; –states for LTS of Car
dfai_Car: Enum {d_car_i1,...,d_car_iD_i}; –states for dfa_i of Car

Based on the types defined above, following variables are declared for each process class:

- An array of *enumeration type representing the LTS states*, indexed by the scalarset type corresponding to this process class. For example, LTS states for objects of process class Car will be represented using the following array variable:

  Car_lts: Array [Car] of stCar;

- Similarly, array variables are defined to represent the DFA states.

- Arrays corresponding to the variables in the IPC model. Murphi supports only integer/boolean variables and the range for integer type needs to be specified in declaration. For example, variable *mode*[4] for the Car is declared as follows:

  Car_mode: Array [Car] of 0..1;

Associations are represented using two dimensional arrays having the value range 0..1. For an association "Asc" between two process classes A and B, this array is indexed by their scalarset types, for example:

Asc: Array [A] of Array [B] of 0..1;,

assuming A and B have been declared as appropriate scalarset types. Value 1 will indicate existence of an association between the objects of A and B, whose identities are represented by the index values of that particular array element.

For each *transaction* in the trace being checked, a corresponding *rule* is defined in Murphi (representing a guarded command) using the witness and destination partitions' information: *control states, dfa states* and *variable valuations* for the participating agents, obtained from the symbolic execution. For example, the guard in the translation of *ContactTerminal* transaction may look as follows:

Car_lts[i2] = st_car10; –checking for source control state
Car_dfa1[i2] = d_car_11; –checking for source dfa state

where $i2$ is an index variable of type Car (scalarset type to represent *car* objects). This snippet of guard shows the constraints a Car object must satisfy to execute this rule; intuitively its an encoding of its witness partition.

Another important aspect of translation is imposing the association count constraints while entering two objects into an association. For this, additional guard terms are introduced in the rule for a transaction in which a) a dynamic association is 'imposed' between two agents or, b) a static association is

[4]Mode indicated whether a car will stop or pass through its current terminal.

'checked' between two agents. Since both these operations can insert a *fresh* object pair into an association, it is ensured that inserting them into the particular association does not violate the count restriction for either of the object. The association count information is gathered from the class-diagram of the IPC model.

The initial configuration for the Murphi execution is given as the "Start-state" declaration, where the initial control states, dfa states and variable valuations for various objects are defined.

The spurious run corresponds to a deadlocked run in Murphi for which no alternative execution path exists or all result in a deadlock. By slight modification in the Murphi code we are able to detect all possible deadlock states, and precisely identify the transaction it got stuck at and furthermore report the system state at that point. This can then be analyzed by the user to determine the possible cause for deadlock.

For initial experiments we have have used the test cases for the four examples discussed later in the Section 6. These test cases correspond to meaningful use cases and no spurious run is detected in their check except for one case: in the trace for the telephone-switch example with call waiting feature[5].The running times for various cases is within 0.1 second with length of traces varying between 12–50 transactions; which is an encouraging indicator as to the usefulness and scalability of this approach.

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.

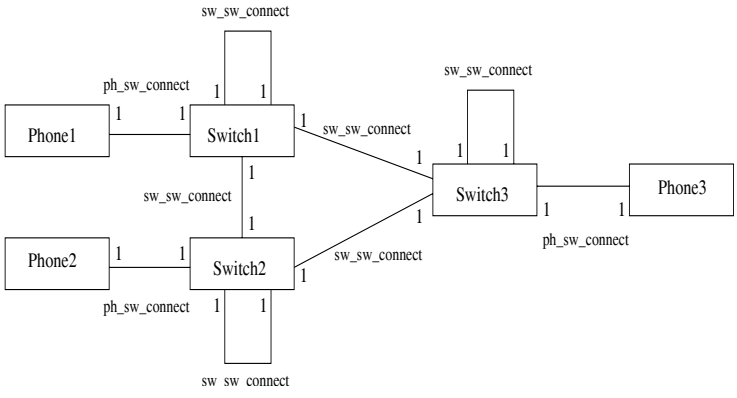


Figure 5: Class diagram for Telephone-switch example.

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

[5]Details are discussed in Appendix B.

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 feature. 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 [3] and using Live Sequence Charts in [2]. This is an automated rail-car system with several cars operating on two parallel cyclic paths with several terminals.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. This example appears in more detail in Appendix A.

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.

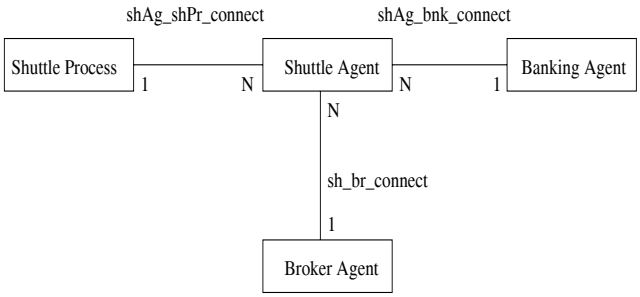


Figure 6: Class diagram for Automated shuttle example.

The automated rail-shuttle system [17] 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 6, showing its main components.

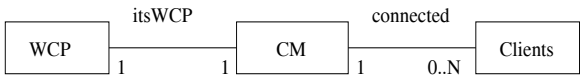


Figure 7: Class diagram for Weather controller example.

The weather update controller [1] 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 7.

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

For the Telephone Switch example with call-waiting feature, we consider three possible test cases. In the first one there were three calls made, each independent of another, and without invoking the call-waiting feature. In the second and third cases, we have two ongoing calls and then a third call is made to one of the busy phones, invoking the call-waiting feature. These two cases differ in how the calls resume and terminate.

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

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	48	12	10	11
	CarHandler	48	3	8	8
	Terminal	6	6	6	6
	Platform Mngr.	6	1	3	3
	Exits Mngr.	6	1	2	2
	Entrance	12	2	1	2
	Exit	12	1	2	2
	Cruiser	48	1	3	5
	Proximity Sensor	48	1	1	2
	cDestPanel	48	1	1	1
	tDestPanel	6	1	1	1

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

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 is much less than the number of concrete objects in various examples.

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.

Example		Time (sec)			Memory (MB)		
		C	S	**C/S**	C	S	**C/S**
RailCar	24cars	3.9	2.1	**1.9**	173	83	**2.1**
	48cars	7.0	2.2	**3.2**	353	84	**4.2**
Shuttle	30 Shuttles	0.7	0.4	**1.6**	33	18	**1.8**
	60 Shuttles	1.2	0.4	**2.7**	69	18	**3.8**
WthrCon	10 Clients	0.6	0.5	**1.2**	21	18	**1.2**
	20 Clients	0.8	0.5	**1.6**	27	18	**1.5**
Simple switch	60ph,30sw	2.0	1.5	**1.3**	87	63	**1.4**
	120ph,60sw	4.1	1.5	**2.7**	189	64	**3.0**

C ≡ Concrete Execution, S ≡ Symbolic Execution

Table 2: Timing and Memory Overheads

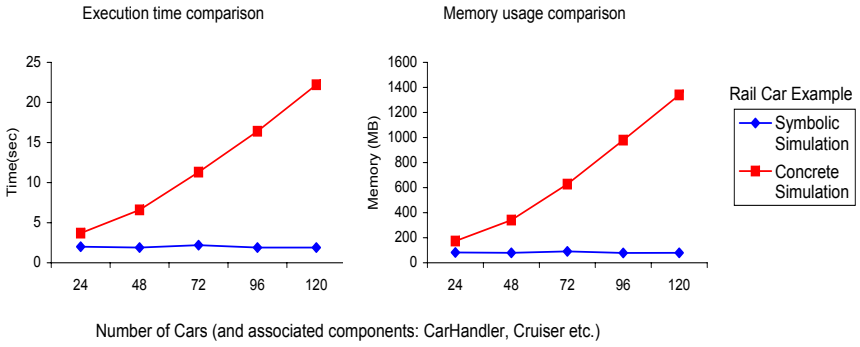


Figure 8: Execution Time and Memory Usage comparison for the Railcar example.

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 2^{nd} configuration is obtained by doubling the number of one (or more) components; such as for *rail-car* example with 24 and 48 cars respectively.

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, as indicated by the ratio C/S in Table 2 (except for Weather

controller example[6]).

Further, the graphs shown in Figure 8, 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, 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 [1] 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 9, 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 9, 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

[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.

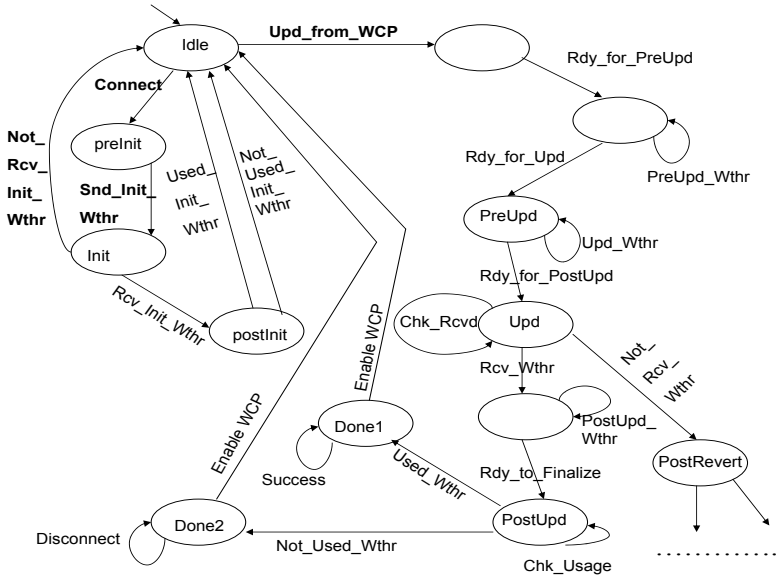


Figure 9: Snippet of Transition System for Weather-Update Controller

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 9). 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 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 (due to increase in the number of possible partitionings of the objects into various behavioral partitions at runtime). 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] CTAS. Center TRACON automation system. `http://www.ctas.arc.nasa.gov`.

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

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

[4] D. Harel, H. Kugler, and A. Pnueli. Synthesis revisited: Generating statechart models from scenarios-based requirements. In *Formal Methods in Software and System Modeling, LNCS 3393*, 2005.

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

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

[7] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

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

[9] C.N. Ip and D.L. Dill. Better verification through symmetry. In *Formal Methods in System Design, 9(2)*, 1996.

[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] Murphi. Murphi description language and verifier, 2005. `http://verify.stanford.edu/dill/murphi.html`.

[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] B. Sengupta and R. Cleaveland. Triggered message sequence charts. In *FSE*, 2002.

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

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

[19] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenario. In *IEEE Transactions on Software Engineering*, volume 29, 2003.

[20] 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.

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

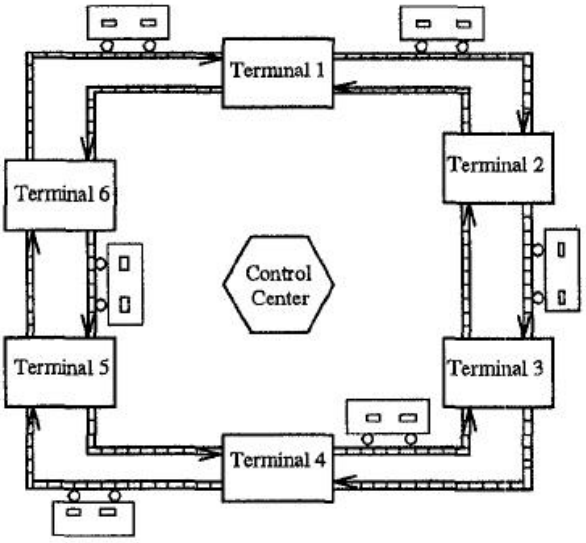


Figure 10: Rail-Car system.

Appendix A RailCar Example

A.1 Introduction

This automated rail car example has been constructed using the descriptions in [3] and [2]. The overall system is shown in Figure 10. As shown in the figure, there are six terminal located along the cyclic path. Each adjacent pair of these terminals is connected by a two rail tracks, one of which is for the clockwise and another for the anti-clockwise travel of the rail cars.

There are several (a fixed number) of rail cars available for transporting passengers between the terminals. There is a control center which receives, processes and communicates data between various terminals and railcars.

As shown in Figure 11, each terminal has got four parallel platforms, on each of which a single railcar can be parked. Further, we have two *entrance* and two *exit* segments which connect the main rail tracks to the terminals platform tracks. These segments can connect the rail tracks to any of the four terminal platforms.

Also, each terminal has a destination board for the use of passengers. It contains a push button and an indicator for each destination terminal. Each rail-car also has a similar destination-panel for the use by passengers. Further a rail-car is equipped with an engine and cruise-controller to maintain the speed. The cruiser can be off, engaged or disengaged.

Figure 12 shows the class diagram for the complete system. 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 modifications in the overall system. This class diagram is the flattened representation of the *object-model diagram* given in [3]. This results in loss of clarity of actual component structures, but in our case we just need to capture relationships

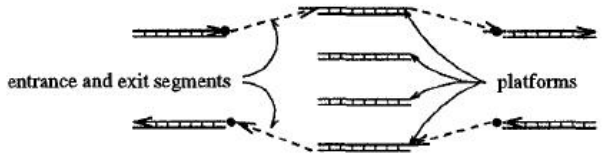


Figure 11: Terminal.

between various components. For example, just from our class diagram we can not say that $PlatformManager$ is part of the $Terminal$ itself. Also, comparing to the *object-model diagram* we have excluded the $Passenger$ class. This does not makes any difference in the model, since we model various user actions non-deterministically and achieve the same effect.

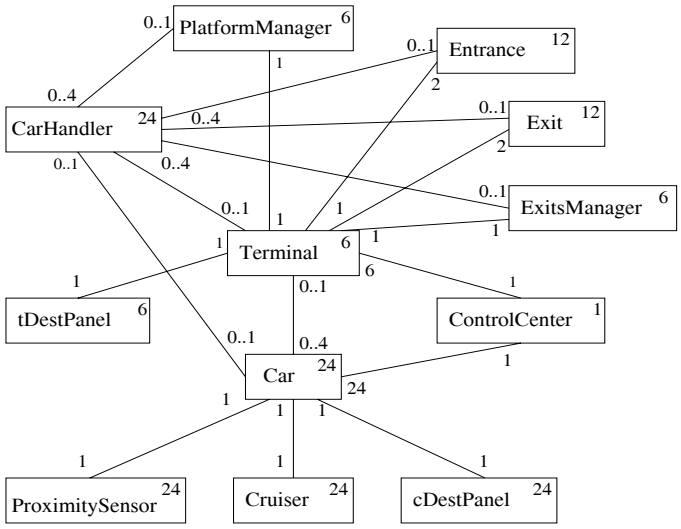


Figure 12: Class diagram for Railcar example.

As mentioned earlier $tDestPanel$ represents the destination panel in the terminal and correspondingly $cDestPanel$ represents the destination panel in the rail-car. We have only one $ControlCenter$ object which is related to all the $Terminal$ and Car objects. Also each Car is related to a $ProximitySensor$, which notifies the car when it arrives within 100 and 80 yards of some terminal, and also related to a $Cruiser$ which maintains the car speed.

When a car is in some terminal or arrives at one, then a unique $CarHandler$ is associated with the Car to handle communication between the car and the terminal. Whenever the car leaves the terminal there is no $CarHandler$ associated with it anymore.

With each terminal we have two $Entrance$ and two $Exit$ objects associated which represent the entrance and exit segments connecting the rail tracks to the terminal's platforms. $PlatformManager$ and $ExitsManager$ respectively allocate platforms and exits to $CarHandler$, which in turn notifies the Car of these events.

A.2 System description

As shown in the class diagram, we have 24 cars in all, and initially we assume that there are 4 cars standing idle in each of the platforms. Then they can get requests to go to some other terminal via the passenger pressing the button in the destination panel of the car itself. Later at runtime, if there is no car in the terminal and a user wishes to go to some place, he will indicate so by pressing the relevant destination button in the terminal's destination panel.

A.2.1 Some use cases

These use cases describe the main execution scenarios during the system execution.

1. *Car approaching terminal* While cruising, a car eventually approaches the next terminal on its path. When it is 100 yards from the terminal, it is alerted by its proximity sensor, as a result of which it contacts sends the arrival request to the next terminal and is allocated a car handler for further communication. Further, the system allocates it a platform and an entrance segment so that it can enter the appropriate platform in the terminal it is heading to. If this allocation can not be made(either due to some technical problem or maybe all platforms are already full) by the time the car is within 80 yards from the terminal, the car is delayed until the allocation is done. Then the car is notified, and it moves in the terminal.

2. *Car departing terminal* If a car has more destination stations to reach then it departs the current terminal after being parked for 90 seconds there, or else it just stays there until any further request comes. If the car needs to depart then the system connects its platform to the appropriate rail-track via the exit segment and engages the car's engine.

3. *Passenger in terminal* In case there is a car in the terminal travelling in the direction in which passenger wishes to travel, he can simply board the train and press the desired destination button in its destination panel. In case, there is no car in the terminal or none travelling in the desired direction, then passenger simple presses the destination button in the terminal's destination panel and waits until the car arrives. The system now will send an idle car from some other terminal.

All these cases have been discussed in [3] and LSCs representing these scenarios have been given in [2].

A.2.2 Component Descriptions

We describe the behavior of various components using state machines. These state machines are based on the statechart descriptions and simple program like descriptions for the main components as given in [3]. The state machines for the rail-car, car-handler, and cruiser are shown in the Figures 13(a), 13(b), and 14 respectively. The state machines of other components are quite simple and are not shown.

Various transitions having the name $Initial\_$ are executed first before the main simulation starts. Their execution initializes the system by making some

initial entries in the global relational table. In this example the initial entries relate the four cars with the terminal they are standing in, with their car-handlers etc.

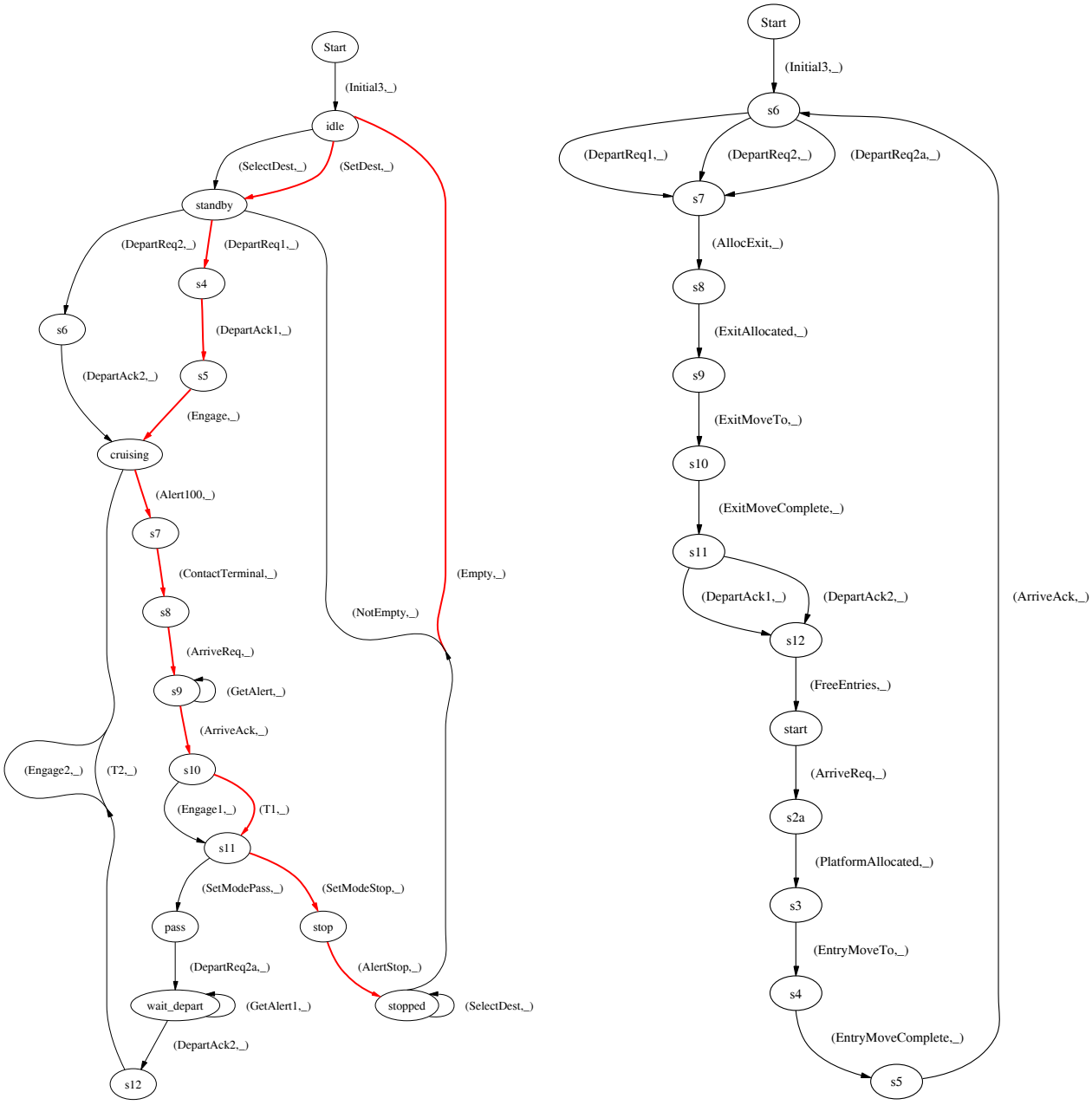
The *SelectDest* transaction indicates the pressing of any one of the six destination buttons at the rail-car's or the terminal's destination panel. *ContactTerminal* transaction represents scenario when a rail-car arrives at the next terminal along it's path.

A.2.3 Sample Execution

We give a sample execution sequence, and describe it in reference with the LSC shown in Figure 16, which is taken from [2]. To produce the similar scenario execution the sequence of transactions that will be executed are shown as the red path in the state machine of rail-car. We give the MSCs corresponding to the main transactions along the path, based on which one can immediately see direct correspondence with the LSC shown in Figure 16. Various relevant MSCs are shown in Figures 15(a) and 15(b).

Transactions $T1$ and $SetModeStop$ are internal transactions of rail-car. $T1$ is an empty transaction, and $SetModeStop$ indicates that the rail-car has to stop at the next terminal it is approaching.

For illustration, if we consider the execution sequence of the transactions $DepartReq1$, $DepartAck1$ and $Engage$, then we get the execution scenario similar to the subchart *Perform Departure* as shown in LSC 16.



(a) State machine for Rail-Car

(b) State machine for Car-Handler

Figure 13: Various state machines

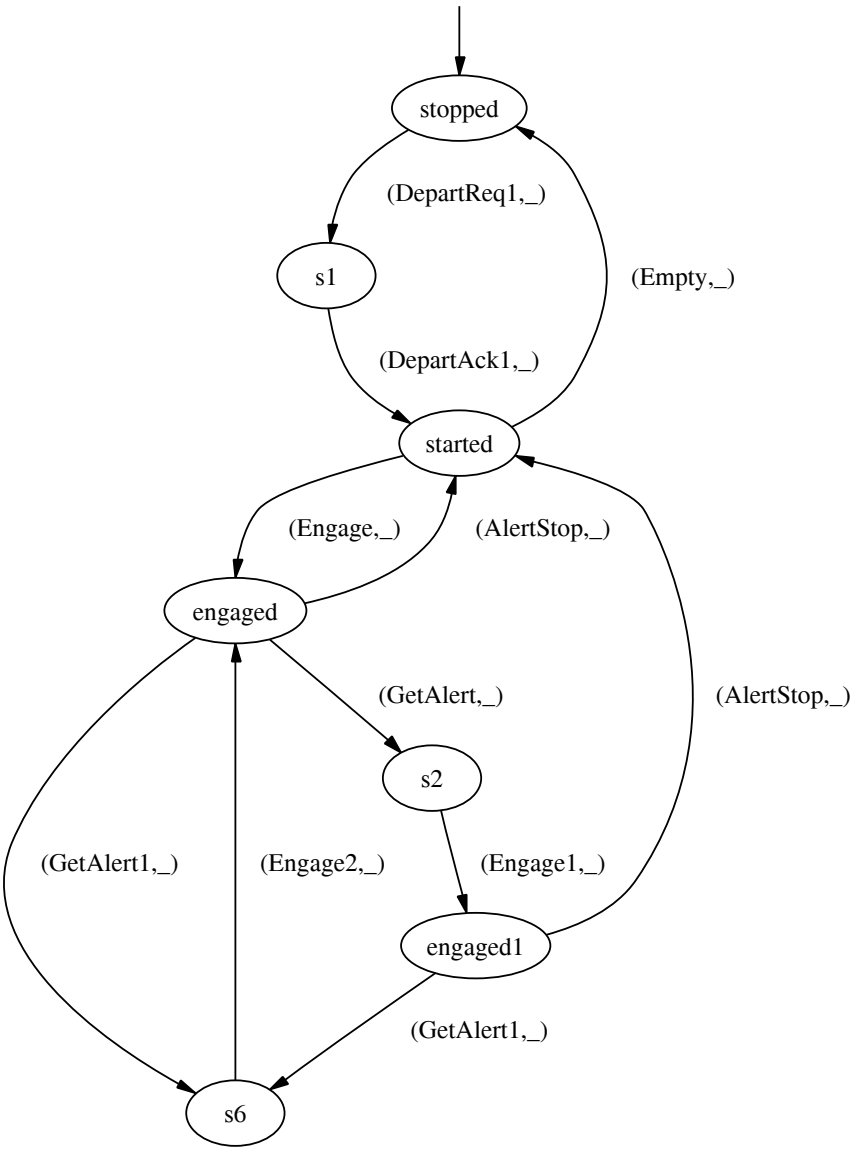
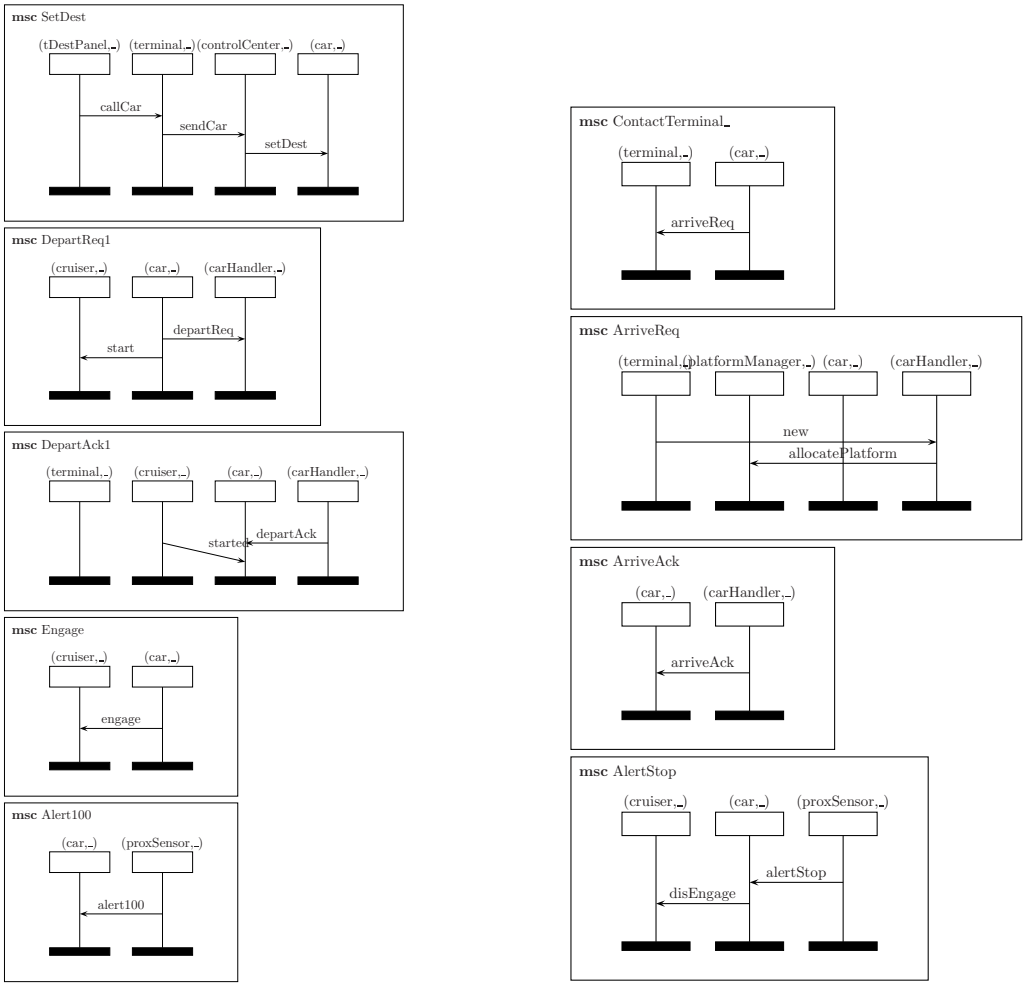


Figure 14: State machine for Cruiser.



(a) MSC1

(b) MSC2

Figure 15: MSCs for rail-car model.

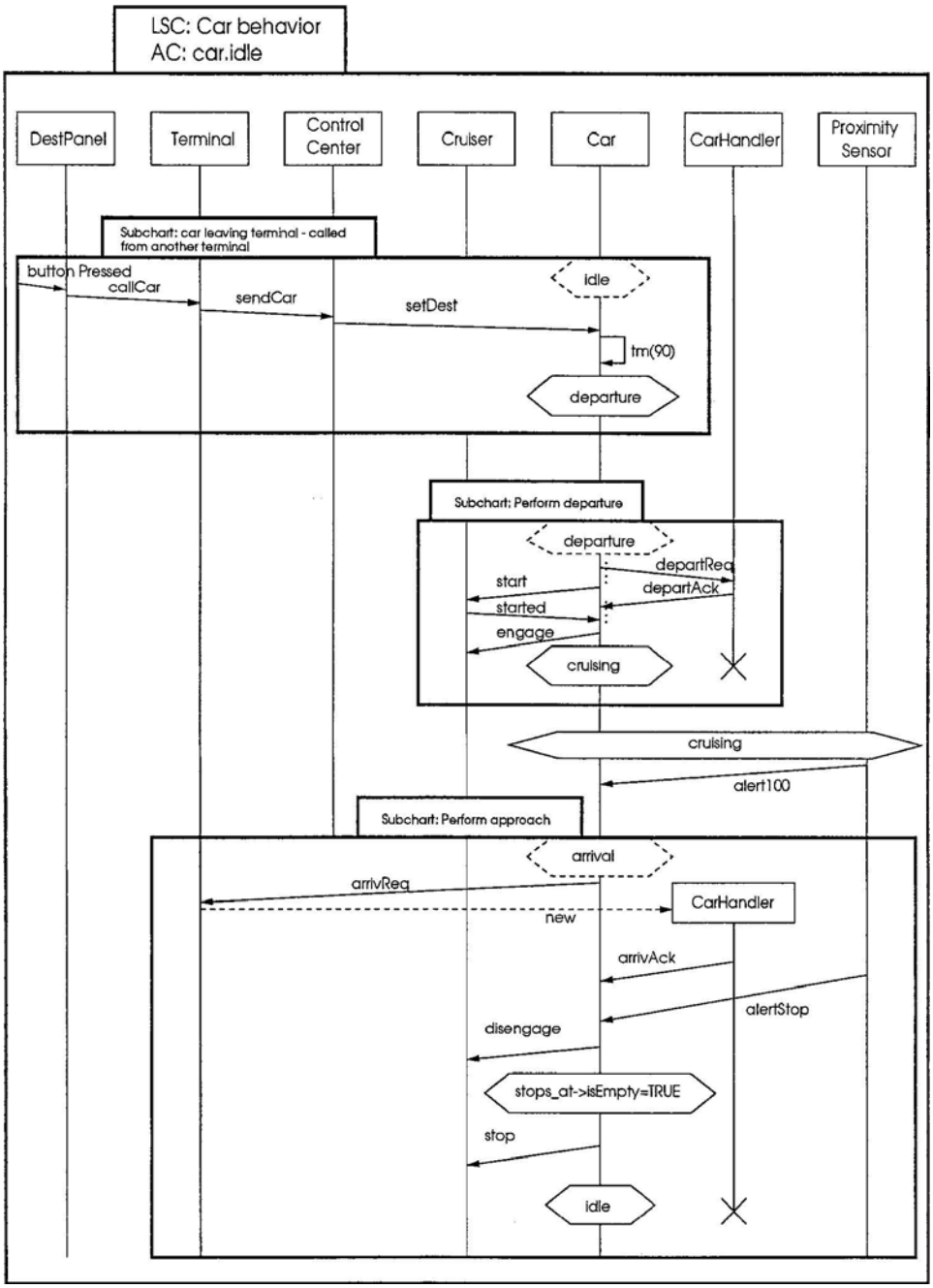


Figure 16: LSC example.

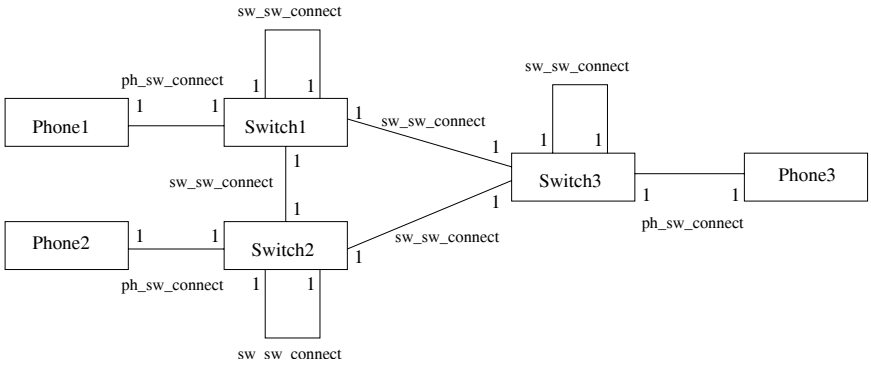


Figure 17: Telephone-Switch Network

Appendix B Telephone Switch Example – Spuriousness Check

B.1 Introduction

This telephone switch model is based on ISUP (ISDN User Part) which is part of SS7 signalling system: a global standard for telecommunications defined by the International Telecommunication Union (ITU) Telecommunication Standardization Sector (ITU-T). ISUP defines the protocol and procedures used to set-up, manage, and release trunk circuits that carry voice and data calls over the public switched telephone network (PSTN).

In this example we assume that various telephones in a particular area are connected to a specific group of switches which are used to handle all the outgoing and incoming calls for this batch of phones. To communicate with another phone, some switch in caller's area (on behalf of the caller) initiates a connection to a remote switch on the callee's side.

In the model we assume enough number of switches in any group to be able to handle calls for all the phones simultaneously in their area. We also assume that a connection between two switches at different locations can always be made if the concerned parties are available.

The above setup needs at least two process classes: one to describe the behavior of switches and another for the phones. To consider a realistic example we consider a network of phones and switches such that it consists of 3 groups of phones, connected to 3 groups of switches as shown in the class diagram in Figure 17.

For example, *Phone1* represents a group of phones, say in *Area1*, whose calls are handled by the switches represented by *Switch1*. To communicate with a phone in another area (or even in the same area), some switch in the caller's area initiates a connection to another switch on the callee's side.

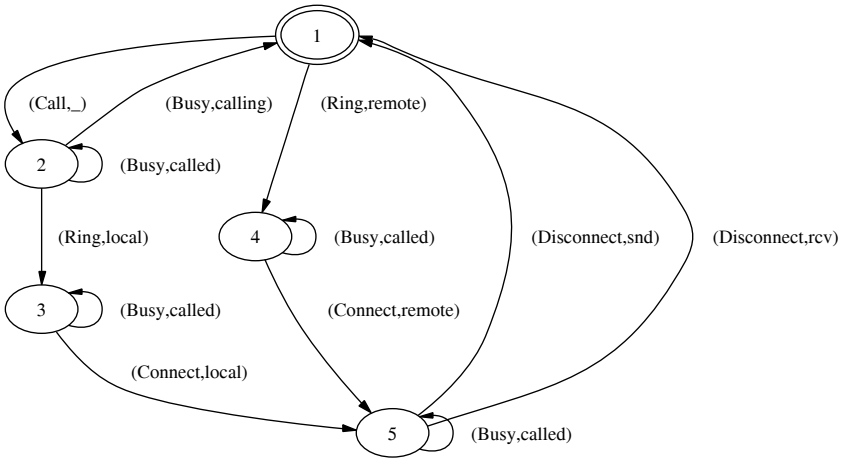


Figure 18: State Machine for Phone

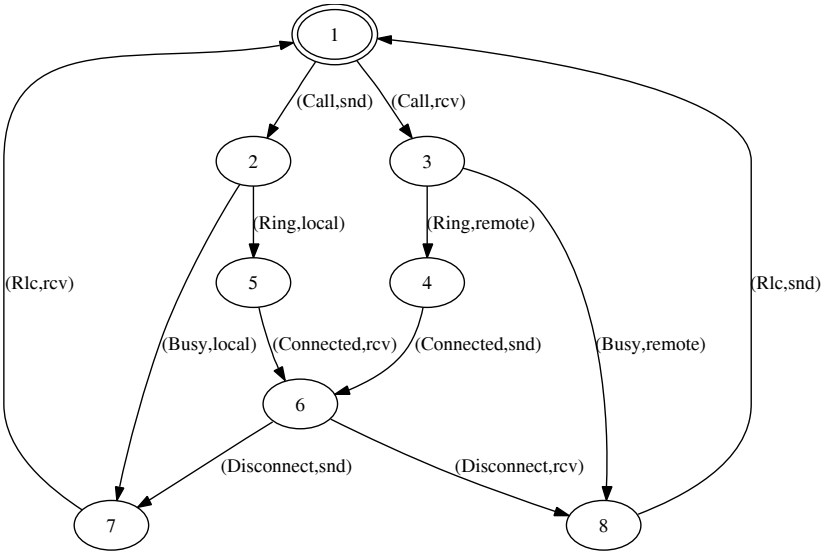


Figure 19: State machine for Switch

B.2 Brief description of the model

B.2.1 Call setup and breakdown

This example models the call setup and breakdown scenario, that may take place in the real telephone network. State machines describing the behaviors of *Phones* and *Switches* are shown in Figures 18 & 19 respectively. To keep the model simple and easy for discussion, out of various ways in which line may be disconnected (since in real world at any given time either of caller or callee may choose to disconnect), we assume that if a call is successful, then it can be disconnected by either caller or callee putting down the phone, and not both of them disconnecting simultaneously.

To establish a call, the transaction *Call* will execute, involving the calling *phone*, *switch* to which it connects and the *remote switch*. Then the remote switch will send the 'ring' to the called phone and local switch will send the 'ringing-tone' to the caller. This is done via the execution of the *Ring* transaction. Consequently, assuming that called phone is always picked up when ringing, the line is established and the two parties talk. This takes place via the execution of the *Connect* transaction.

Now that the two phones are connected, either of them can be put on-hook, sending the busytone to other. This is done via the *Disconnect* transaction. Finally, the two switches disconnect the line by executing the *Rlc* transaction, which stands for *Release Confirm*.

B.2.2 Busy Phone

We also model the situation where the called phone may be busy. A phone is considered busy if it is in either of the states 2–5. Initially the call is attempted as before via the execution of the *Call* transaction. However, since we do not maintain object identities, in case some phone is busy in the locality where the call is made, we may nondeterministically assume that this is the phone being called by the current caller. If that is the case then we execute the *Busy* transaction where the called phone says that it is busy and the caller is sent the busy-tone, after which it puts down the phone. The two switches involved also disconnect.

B.2.3 Abbreviations used

- *iam*: Initial Address message
- *acm*: Address confirm message
- *anm*: Answer message
- *rel*: Release connection
- *rlc*: Release confirm

B.2.4 Call-waiting feature

The model described above is for the basic call-setup-breakdown in telephone networks. However, there are various other features, such as: call forwarding,

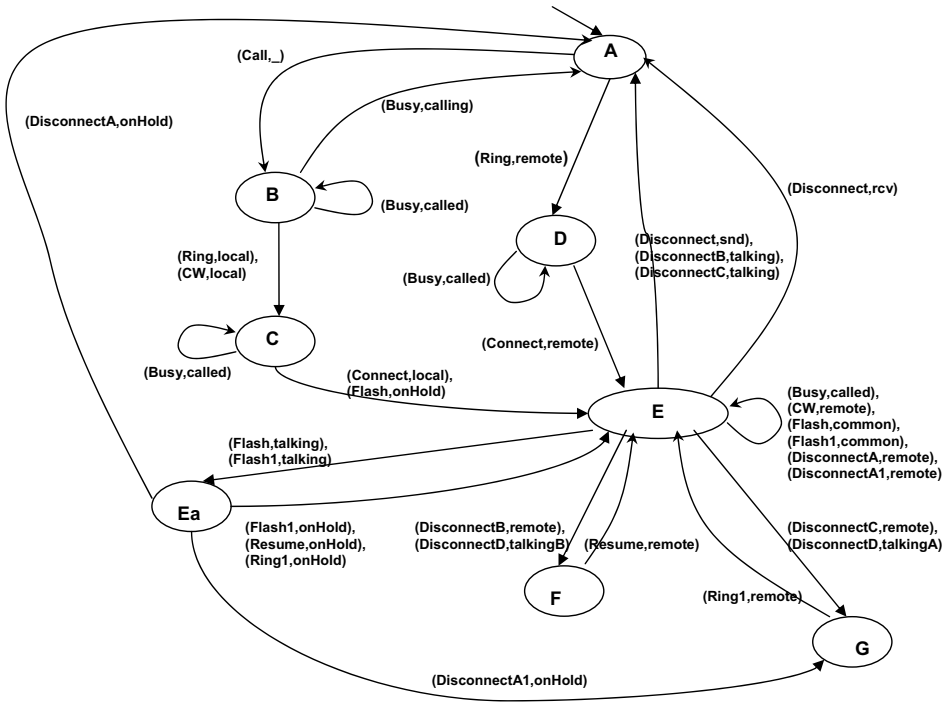


Figure 20: State machine for phone with Call-waiting feature

where the called phone forwards the call to another prespecified phone, or busy call waiting - the called phone if busy puts the calling phone on hold, etc.

We extended the basic model with the call waiting feature. This extension did not require modifying the existing model, but rather was achieved by adding extra transitions and few states. The state-machines for the phone and switch with the call-waiting feature are shown in Figures 20 and 21.

With call waiting feature enabled, when a phone calls a busy phone, then the busy phone may put on hold the phone it is currently talking to and talk to the phone which has called. Then by pressing the *Flash* button it can switch between these phones, putting one on-hold and talking to other. The phone may be put down by any of the three parties and this gives rise to various interesting cases. Let the phone currently on-hold be called *onHold*, one using call-waiting feature be called *common*, and third talking phone be simply called *talking*.

If the *talking* phone in the above scenario diconnects, then call between *onHold* and the *common* is resumed. If the *onHold* phone disconnects, then other two simply continue talking. If *common* phone disconnects, then the *talking* phone is sent the busy tone and it disconnects, and ringing-tone is sent again to the *common* phone on behalf of the *onHold* phone.

As can be observed, there are many possible interacting scenrios that are possible. We experimented with this feature in controlled settings, and guided the simulation by providing sequence of transactions and process-class names for various agents.

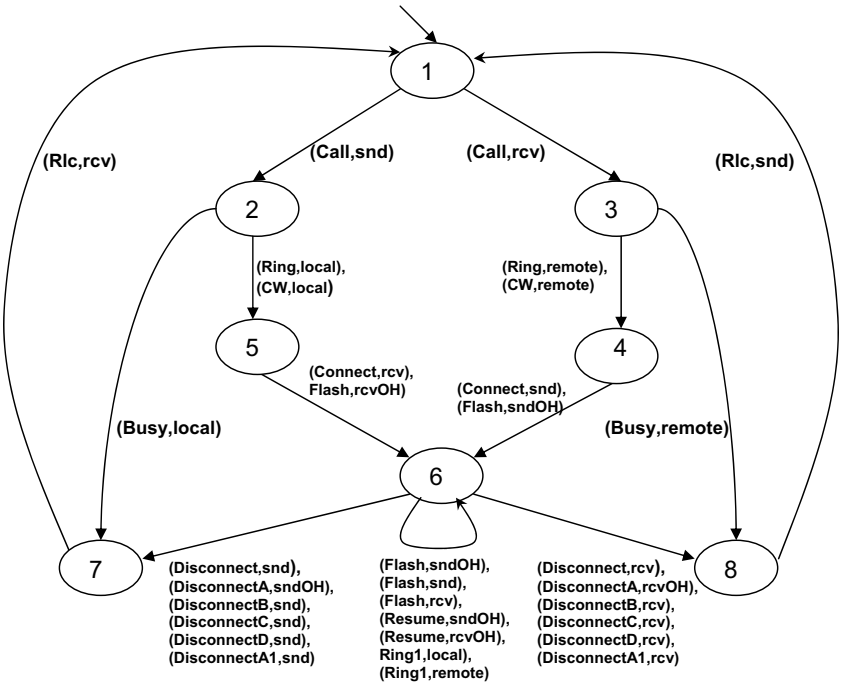


Figure 21: State machine for switch with Call-waiting feature

B.3 Spuriousness Check

As a use-case scenario involving the call-waiting feature we ran the following trace using our simulator: *"Call, Ring, Call, Connect, Ring, Connect, Call, CW, Flash, Flash1, DisconnectB, Rlc, Disconnect, Resume, Rlc, Disconnect, Rlc"*. This use-case involves two independent connections involving four phones, followed by a fifth phone calling one of the busy phones. This leads to three phones being involved the call-waiting feature (two others remain in an independent connection).

Considering the phones involved in the call-waiting feature, while the first two are connected and talking, the third phone calls one of the busy phones, which then puts the other phone on hold and talks to the third phone by executing *Flash* transaction; later again switching between them by executing *Flash1*. As mentioned earlier, we have a *common, talking* and an *onHold* phone involved in the call-waiting feature. Then *DisconnectB* executes, where the *talking* phone disconnects from the *common* phone, later followed by the transaction *Resume* where the talk between *onHold* and *common* phones resume. In the complete trace eventually all the parties disconnect and system returns to initial state. This trace runs to completion in the symbolic simulation. However, during the check for one of the symbolic execution runs of this trace using the Murphi model checker, we find that it is spurious as described below.

The Murphi reports a 'deadlock' after executing first 10 transactions in the trace, i.e. it is unable to execute *DisconnectB* and proceed further. It prints the system state at this point, relevant part of which is shown in Table 3. It gives the

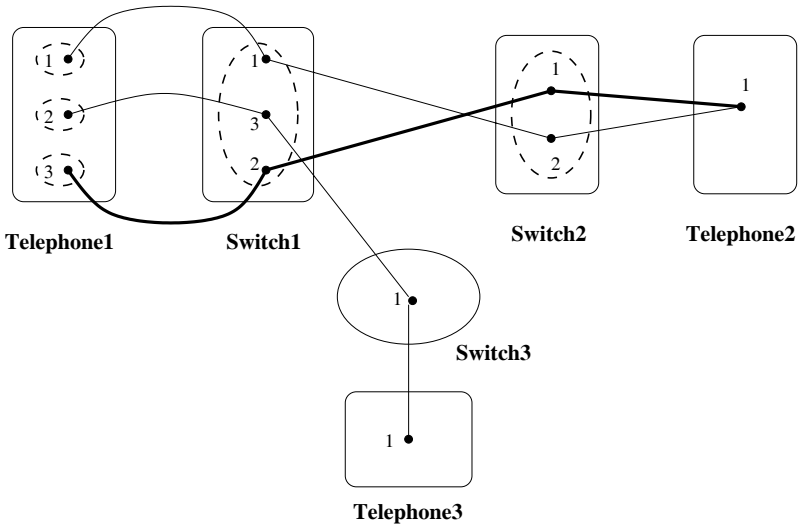


Figure 22: Class Diagram showing Objects with Associations

LTS & DFA states, associations and variable valuations for various objects in the system[7]. This information is graphically presented in Figure 22, which shows various objects in their respective classes along with various associations. We also show the objects in the same behavioral partitions using dashed boundaries.

Since at the point of deadlock, system has executed the first 10 transactions, as described earlier: three phones: *Telephone1_1*[8], *Telephone2_1* and *Telephone1_3* are involved in the call-waiting feature s.t. Telephone1_1 is *onHold*, Telephone1_3 is *talking* and Telephone2_1 is *common* phone (this relationship is shown via associations in Figure 22). Also, another pair *(Telephone1_2,Telephone3_1)* is in an independent connection.

Next we examine the reason for deadlock. From the given configuration, we determine the appropriate set of objects to execute *DisconnectB*, which are shown connected by bold associations in Figure 22. However, from the guard of *DisconnectB* as produced in Murphi translation (shown in Table 4), we find that in symbolic simulation the witness partition corresponding to 'Telephone1_2' instead of 'Telephone1_3' was chosen. This happened because: a) witness partition for 'Telephone1_2' also satisfies the guard requirements for *DisconnectB* and b) 'Switch1_2' & 'Switch1_3' are in same behavioral partition. And since we maintain the association information between partitions and not objects in the symbolic execution semantics, this choice of partitions executes fine in symbolic simulation. However, due to exact association information being maintained in Murphi, this spuriousness gets detected and deadlock is reported.

[7]Note that a trace is simulated using the cut-off number of objects in Murphi.

[8]TelephoneI_N indicates Nth object of TelephoneI process class. Similar notation is used for various Switch classes.

```
connected11[Telephone1_1][Switch1_1]:1
connected11[Telephone1_2][Switch1_3]:1
connected11[Telephone1_3][Switch1_2]:1
connected22[Telephone2_1][Switch2_1]:1
connected22[Telephone2_1][Switch2_2]:1
connected33[Telephone3_1][Switch3_1]:1
connected12[Switch1_1][Switch2_2]:1
connected12[Switch1_2][Switch2_1]:1
connected13[Switch1_3][Switch3_1]:1
Telephone1_lts[Telephone1_1]:st_telephone3      –Correponds to PhoneCW LTS state 'Ea'
Telephone1_lts[Telephone1_2]:st_telephone4      –Correponds to PhoneCW LTS state 'E'
Telephone1_lts[Telephone1_3]:st_telephone4
Telephone1_dfa1[Telephone1_1]:d_telephone_12
Telephone1_dfa1[Telephone1_2]:d_telephone_11
Telephone1_dfa1[Telephone1_3]:d_telephone_11
Telephone1_dfa2[Telephone1_1]:d_telephone_21
Telephone1_dfa2[Telephone1_2]:d_telephone_21
Telephone1_dfa2[Telephone1_3]:d_telephone_22
Telephone1_callWaitMode[Telephone1_1]:false
Telephone1_callWaitMode[Telephone1_2]:false
Telephone1_callWaitMode[Telephone1_3]:false
Telephone2_lts[Telephone2_1]:st_telephone4
Telephone2_dfa1[Telephone2_1]:d_telephone_11
Telephone2_dfa2[Telephone2_1]:d_telephone_21
Telephone2_callWaitMode[Telephone2_1]:true
Telephone3_lts[Telephone3_1]:st_telephone4
Telephone3_dfa1[Telephone3_1]:d_telephone_11
Telephone3_dfa2[Telephone3_1]:d_telephone_21
Telephone3_callWaitMode[Telephone3_1]:false
Switch1_lts[Switch1_1]:st_switch3      –Correponds to SwitchCW LTS state '6'
Switch1_lts[Switch1_2]:st_switch3
Switch1_lts[Switch1_3]:st_switch3
Switch2_lts[Switch2_1]:st_switch3
Switch2_lts[Switch2_2]:st_switch3
Switch3_lts[Switch3_1]:st_switch3
```

Table 3: Deadlocked state from Murphi

```
Ruleset i1:Telephone2; i2:Switch2; i3:Switch1; i4:Telephone1 Do
    Rule "Execute DisconnectB"
        (Count = 11
        & Telephone2_callWaitMode[i1] = true
        & Telephone2_lts[i1] = st_telephone4       –checking for source control state
        & Telephone2_dfa1[i1] = d_telephone_11      –checking for source dfa state
        & Telephone2_dfa2[i1] = d_telephone_21      –checking for source dfa state
        & Switch2_lts[i2] = st_switch3      –checking for source control state
        & Switch1_lts[i3] = st_switch3      –checking for source control state
        & Telephone1_callWaitMode[i4] = false
        & Telephone1_lts[i4] = st_telephone4       –checking for source control state
        & Telephone1_dfa1[i4] = d_telephone_11      –checking for source dfa state
        & Telephone1_dfa2[i4] = d_telephone_21      –checking for source dfa state
        & connected11[i4][i3] = 1
        & connected22[i1][i2] = 1
        & connected12[i3][i2] = 1           )
        .
        .
```

Table 4: Guard of DisconnectB from Murphi