Interacting Process Classes

Ankit Goel Abhik Roychoudhury P.S. Thiagarajan
National University of Singapore, Singapore

Many reactive control systems consist of classes of active objects involving both intra-class interactions (i.e. objects belonging to the same class interacting with each other) and inter-class interactions. Such reactive control systems appear in domains such as telecommunication, transportation and avionics. 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 labeled transition system, unit interactions between process objects are described as *transactions* and the structural relations are captured via class diagrams. The key feature of our approach is that our execution semantics leads to an *abstract* simulation technique which involves: (i) grouping together active objects into equivalence classes according their potential futures, and (ii) keeping track of the number of objects in an equivalence class rather than their identities. Our simulation strategy is both time and memory efficient and we demonstrate this on well-studied non-trivial examples of reactive systems. We also present a case study involving a weather-update controller from NASA to demonstrate the use of our simulator for debugging realistic designs.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements / Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*State Diagrams*

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: Abstract Execution, Active Objects, Message Sequence Charts, Unified Modeling Language (UML)

1. INTRODUCTION

Classes of behaviorally similar processes arise naturally in application domains such as telecommunications, transportation systems and avionics. We propose a modeling language based on conventional notations to describe *classes* of interacting processes. In these settings, during the initial design phase it will often be unnatural to fix the number of objects in each process class of the system. One possibility might be to arbitrarily fix a small number n_p of objects for each process p. However it is difficult to ensure that the resulting restricted system exhibits all the interesting behaviors of the intended system.

In our current framework, we allow for a class to have an unbounded number of

An initial version of this paper was published as- Interacting Process Classes. In *ACM International Conference on Software Engineering (ICSE-06)*. `http://www.comp.nus.edu.sg/~abhik/pdf/icse06.pdf`. Authors' e-mails: `{ankit,abhik,thiagu}@comp.nus.edu.sg` Authors' Address: School of Computing, National University of Singapore, Computing 1, Law Link, Singapore 117590.

objects. In other words, we allow n_p, the number of objects to be left unspecified and define a semantics which is valid for *any* value of n_p. Thus, in the present setting, we allow n_p to be specified, either as a *constant*, or an *unbounded* number represented by ω. Moreover, when n_p is specified as a constant, it may still grow unbounded due to the dynamic creation of active objects from a class. In section 6.2, we discuss the support for dynamic object creation and deletion in our framework.

Thus, we develop a modeling framework, where one can efficiently simulate and validate a system with an arbitrary 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, this will lead to an impractical blow-up in memory usage during simulation. Instead, we propose an *abstract* execution semantics, where we dynamically group together objects that will have similar future behaviors. While doing so, we keep track of only the number of objects in each equivalence class and not their identities. This results in considerable time and memory efficiency of our simulator. However, this also leads to a loss of information due to presence of associations between objects as explained below.

We use labeled transition systems to describe the behavior of process classes. The unit of interaction between the objects can involve more than two participants. Further, it can be an abstraction of a protocol that involves bidirectional flow of signals and data between the objects taking part in the interaction. In our operational semantics, this unit of interaction, called a **transaction**, is executed in an atomic fashion. For illustrative purposes however, we often use the notation of Message Sequence Charts (MSCs) to refine the transactions.

We also specify static and dynamic associations between objects. We use class diagrams in a standard way to specify static associations. Structural constraints imposed by the system are naturally captured via static associations. For instance, a node may be able to take part in a "transmit" transaction only with nodes with which it has a "neighbor-of" association. *Dynamic associations* on the other hand are needed to guarantee that proper combinations of objects take part in a transaction. For instance, when choosing a pair of phone objects to take part in a "disconnect" transaction, we may have to choose a pair which is currently in the "connected" relation. This relation has presumably arisen due to the fact that they took part last in a "connect" transaction. The combination of these features together with the imperative to develop an abstract execution semantics is a challenging task.

To avoid the blow-up caused by tracking object identities, one must use an abstraction, which in our case is count-based and more importantly *class-based*. In other words, an abstract state will specify the number of objects currently residing in an equivalence class of objects. Naturally, a necessary -but not sufficient- condition for two objects to be behaviorally equivalent is that they must belong to the same process class. In addition, we must also keep track of -in this abstract setting- the *current* associations that exist between the objects, since they will play a crucial role in determining whether or not a transaction can take place. This is the key challenge in formulating an abstract execution semantics.

We tackle this problem here and present a simulator based on our solution. We

also measure the time/memory efficiency of our simulation mechanism on well-known non-trivial examples of reactive control systems.

The present paper is an expanded and improved version of the conference paper [Goel et al. 2006]. The main additions are as follows.

—To achieve time and memory efficient simulation, our execution semantics is designed to be an "over-approximation"; it allows behaviors which cannot appear in a concrete execution. The inexactness arises from the way we manage associations. We establish the exactness of our abstract execution semantics in the absence of associations. We also prove that our abstract execution semantics is an over-approximation when static/dynamic associations are considered. These results appear in Section 8.1.

—We elaborate on the usage of the model checker Murphi to automatically detect spurious execution runs produced by our abstract execution semantics (see Section 8.2, 8.3 and Appendix B).

—Apart from investigating the space/time efficiency of our simulator for abstract semantics, we have used it for-

a) **Test generation**, where we generate a set of witness traces from a system model, which satisfy a test-purpose given as a sequence of transactions. The experimental results for various examples we have modeled appear in Section 9.4.

b) **Debugging** realistic controller examples. In particular, we have located certain design bugs in the CTAS weather-update controller from NASA. This is a part of a larger system that has been deployed for controlling incoming air-traffic in large airports. These new results appear in Section 9.6.

We develop our modeling language in two steps. First we present the core modeling language without associations and develop its concrete execution semantics. We then formulate our abstract execution semantics. As a second step, we introduce (static and dynamic) object associations and correspondingly extend the semantics. Further, we establish results relating the concrete semantics to the abstract semantics. Finally, we present experimental results demonstrating capabilities of the simulator for our model.

2. RELATED WORK

The use of *visual* and *executable* notations for specifying the requirements of complex reactive systems is an important research area. Use of such models during early phases of system development forms an easy and more sound basis of communication between users and system designers. Moreover, such specifications can be used for requirements simulation and validation. This can provide the user with an early feedback and help detect various design errors in early stages of system design.

Broadly, such modeling notations fall in two categories: a) Intra-object or *state-based* notations, or b) Inter-object or *scenario-based* notations. The state-based notations specify the control flow of various classes of objects using finite state machine representations such as Statecharts [Harel and Gery 1997]. While, the scenario-based notations, such as, Live Sequence Charts [Damm and Harel 2001] and Triggered Message Sequence Charts [Sengupta and Cleaveland 2002], are used

to specify various interaction scenarios between system processes. There has also been work on using a combination of state-based and scenario-based notations, where the control flow for various processes is specified using labeled transition systems and the inter-object interactions are specified using Message Sequence Charts (MSCs) [Roychoudhury and Thiagarajan 2003]. All these approaches deal with *concrete* objects and their interactions.

Executable State-based models. There are a number of design methodologies based on the notions of class and state diagrams as exemplified in the UML-based tools Rhapsody and RoseRT [I-Logix ; Rational Rose]. These tools emphasize the use of state-based notations for system modeling and also have limited code generation facilities. However, no abstract execution semantics is provided and the interactions between the objects -not classes- have to be specified at a fairly low level of granularity, such as a single message send/receive. 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.

Executable Scenario-based Models. In our present work, we use labeled transitions systems to specify the control flow of each process class, while the notion of a transaction is used for describing an interaction involving multiple processes. Moreover, as we illustrate using various examples, we can use Message Sequence Charts for describing transactions, hence leading to an executable scenario based model. In existing literature, Live Sequence Charts (LSCs)[Damm and Harel 2001; Harel and Marelly 2003] offer an MSC-based inter-object modeling and simulation framework for reactive systems. LSCs describe system behavior by prescribing existential and universal temporal properties that the overall system interactions should obey, rather than giving a per-process state machine. As a consequence, LSCs completely suppress the control flow information pertaining to individual process classes. More importantly, we note that in the LSC framework, though the objects of a process class can be *specified* symbolically, the execution mechanism (the play-engine as described in [Harel and Marelly 2003]) does not support abstract execution. The symbolic instances must be instantiated to concrete objects during simulation. The approach taken in [Wang et al. 2004] alleviates this problem by maintaining constraints on process identities but falls short of a fully abstract execution. In the present setting, we do not maintain the process identifiers at all. Also, the work on Triggered Message Sequence Charts [Sengupta and Cleaveland 2002] allows for a non-centralized execution semantics, in comparison to the play-engine of LSCs. However, the TMSC language supports neither symbolic specifications nor abstract execution of process classes.

Parameterized System Verification. Our method of grouping together active objects is related to abstraction schemes developed for grouping processes in parameterized systems (*e.g.*, see [Delzanno 2000; Pnueli et al. 2002]). In such systems, there are usually unboundedly many processes whose behavior can be captured by a single finite state machine or an extended 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. For instance, in [Pnueli et al. 2002] a concrete system consisting of $n > 1$ identical processes is abstracted into a finite state system in which for each local state, the process count can be either 0, 1 or 2. The count of 0(1) indicates currently zero(single) process in the corresponding state, while count of 2 indicates 2 or more processes in a state. However, in our setting inter-object associations across classes have to be maintained — an issue that does not arise in parameterized system verification. This indeed is the key technical challenge in our work: How does one capture the information, say, "an object i is linked via some association asc to an object j" when we do not maintain object identities in our abstract execution semantics?

Model Checking and Abstraction. There are several works which employ abstraction techniques for the purpose of model checking [Clarke et al. 1994; Henzinger et al. 2002; Clarke et al. 2003]. The main aim is to reduce the state space of a given system-model in order to make model checking tractable. In these approaches, various data types are replaced with smaller-sized types, thus obtaining an abstract model of the system in terms of behaviors, which is generally an over-approximation of the original system. Such an abstraction preserves the correctness of properties, in the sense that, a property proven correct for the abstract model, also holds for the original model.

In our present work, we only abstract away the process identities and dynamically group together various processes having identical state during run-time. Thus, currently we do not abstract away any other data types; but it can be easily integrated in to our framework.

Other Works. The notion of "roles" played by processes in protocols have appeared in other contexts (*e.g.* [Selic 1998]). Object orientation based on the actor-paradigm has been studied thoroughly in [Lee and Neuendorffer 2004]. We see this work approach as an orthogonal one where the computational rather than the control flow features are encapsulated using classes and other object-oriented programming notions such as inheritance.

Finally, our model can be easily cast in the setting of Colored Petri nets [Jensen 1995] with our operational semantics translating into an appropriate token game. We feel however our formulation is simpler and better structured in terms of a network of communicating transition systems.

3. THE MODELING LANGUAGE

Our model consists of a network of interacting process classes where processes with similar functionalities are grouped together into a single class. We will often say "objects" instead of processes and speak of "active" objects when we wish to emphasize their behavioral aspects.

In what follows, we fix a set of process classes $\mathcal{P}$ with p, q ranging over $\mathcal{P}$. For each process class p, we let the set of objects in class p to be a finite non-empty set but do not require its cardinality to be specified; this is a fundamental feature of our modeling language. We now describe the notion of a **transaction** for specifying a unit of execution involving one or more processes in the system. A *transaction*

$\gamma = (R, I, Ch)$ consists of three components-

(1) A finite set of roles R. Each role r in R is a pair (p, ρ) where $p \in \mathcal{P}$ is the name of a class from which an object playing this role is to be drawn. On the other hand, ρ indicates the *functionality* assigned to the role r ("Sender", "Receiver" etc.).

(2) I is a guard consisting of conjunction of guards, one for each role r in R. The guard I_r associated with role $r = (p, \rho)$ specifies the conditions that must be satisfied by an object of class p in order for it to be eligible to play the role r.

(3) Ch represents the *behavior* corresponding to roles R in transaction γ.

The set of all transactions is denoted as Γ.

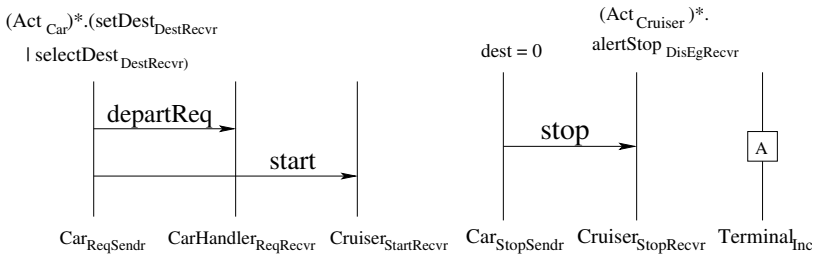
We require that if (p_1, ρ_1) and (p_2, ρ_2) are two distinct members of R , then $\rho_1 \neq \rho_2$. We however *do not* demand $p_1 \neq p_2$. Thus two different roles in a transaction may be played by two objects drawn from the same class. Further, for a transaction $\gamma = (R, I, Ch) \in \Gamma$, the way Ch is defined is not central to our modeling notation and its semantics; any suitable means for specifying the behavior can be used– for example, Message Sequence Charts (MSCs) specifying the communication/computation actions to be executed by objects assigned to various roles in a transaction, or, simply post-conditions specifying change in the variable valuations of objects playing various roles in a transaction can be used.

For the purpose of exposition we will use Message Sequence Charts (MSCs) for describing the behavior (Ch) of a transaction. For a transaction $\gamma = (R, I, Ch) \in \Gamma$, we view Ch as a labeled poset of the form $Ch = (\{E_r\}_{r \in R}, \leq, \lambda)$[1] where E_r is the set of events that the role r takes part in during the execution of γ. The labeling function λ, with a suitable range of labels, describes the messages exchanged by the instances as well as the internal computational steps during the execution of Ch. Finally, $\leq$ is the partial ordering relation over the occurrences of the events in $\{E_r\}_{r \in R}$. Within the MSCs, the communication via sending and receiving of messages can be synchronous or asynchronous. This issue is orthogonal to our model. In the operational semantics of our model, we assume for convenience that the execution of each transaction is atomic. As a consequence, the concatenation of MSCs in our setting, say, to depict the execution of a sequence of transactions, will be synchronous [Alur et al. 1996].

We show an example transaction in Figure 1(a). It occurs in our IPC model of Rail-car example presented in [Damm and Harel 2001; Harel and Gery 1997]. For the moment, let us ignore the regular expression at the top portion of this chart. Note that as a notational shorthand we often write the role (p, ρ) as p_ρ. This convention has been followed in the transactions of Figure 1. In Figure 1(a), the roles are "$Car_{ReqSendr}$", "$CarHandler_{ReqRecvr}$" and "$Cruiser_{StartRecvr}$". This naming convention is intended to indicate that the functionality "ReqSendr" is played by an object drawn from the class "Car", the functionality "ReqRecvr" is played by an object belonging to the class "CarHandler", and so on.

For each class, a labeled transition system will capture the common sequences of actions that the objects belonging to the class can go through. An action label

[1] We often use the expression $\{X_r\}_{r \in R}$ as an abbreviation for $\{X_r | r \in R\}$.



(a) departReqA (b) noMoreDest

Fig. 1. **Transactions *departReqA* & *noMoreDest*; A is an internal computation event in *noMoreDest***

will name a *transaction* and the *role* to be played by an object of the class in the transaction. We use Act_p to denote the set of all actions that process class p can go through. Accordingly, a member of Act_p will be a triple of the form (γ, p, ρ) with $\gamma = (R, I, Ch) \in \Gamma$, $r = (p, \rho) \in R$. The action label (γ, p, ρ) will be abbreviated as γ_r. When p is clear from the context it will be further abbreviated as γ_ρ.

As mentioned earlier, in a transaction $\gamma = (R, I, Ch)$, the **guard** I_r associated with the role $r = (p, \rho)$ 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. These conditions will consist of two components: The first one is a *history* property of the execution sequence of actions that O_r has so far gone through. It will be captured using *regular expressions* over the alphabet Act_p, the set of all the action labels corresponding to process class p. We denote it using Λ. The second component is a *propositional formula* (denoted as Ψ) built from boolean assertions regarding the values of the variables owned by O_r. Thus, guard of a role r is of the form $I_r = (\Lambda, \Psi)$. For instance, consider the transaction "departReqA" shown in Figure 1(a)). A Car object wishing to play the role (*Car, ReqSendr*) must have last played the role (*Car, DestRecvr*) in the transaction *setDest* or in the transaction *selectDest*. This is captured by the guard

$$Act^{\star}_{car}.(setDest_{DestRecvr}|selectDest_{DestRecvr})$$

shown at the top of the lifeline corresponding to role (*Car, ReqSendr*) in Figure 1(a). As this example shows, we use regular expressions to specify the history component of a guard. Also, note that in the transaction "departReqA" , 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,StopSendr*) must satisfy "dest = 0"; there is no execution history based guard for this role. Finally, if for some role no guard is mentioned (e.g. $Cruiser_{StartRecvr}$ in Fig.1(a)) then the corresponding guard is assumed to be vacuously true.

The transition system describing the common control flow of the objects belonging to the class p will be denoted as TS_p. It is 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}. As described

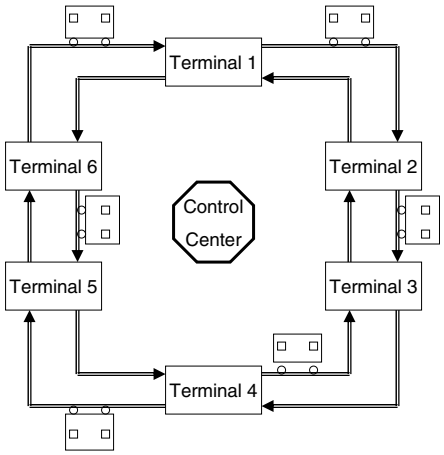


Fig. 2. Rail-car system

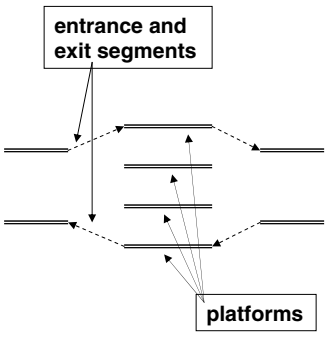


Fig. 3. Terminal

earlier, Act_p is the set of action labels, with each label specifying a transaction and a role in that transaction, that the p-objects can play in the transactions in Γ. The effects of 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. For convenience, we shall assume that for each variable $u \in V_p$, all the objects of class p assign the same initial value to u. This initial assignment is captured by the function v_{init_p}. We assume appropriate value domains for the variables in V_p exist but will not spell them out here. A computational step can be viewed as a degenerate type of transaction having just one role. Hence we will not distinguish between computational steps and transactions 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 finite 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 1. **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.

4. MODELING A RAIL-CAR SYSTEM: THE FIRST-CUT

We discuss here the preliminary modeling of a Rail-car example [Damm and Harel 2001; Harel and Gery 1997] using the Interacting Process Classes (IPC) formalism. This example was initially developed in [Harel and Gery 1997] and used subsequently in [Damm and Harel 2001] to illustrate the modeling capabilities of Live Sequence Charts. It is a non-trivial distributed control system with many process classes such as cars, cruiser, terminal etc. The schematic structure is shown in Figure 2. There are six terminals located along two parallel cyclic tracks, one of

Process Class	# Concrete Objects
Control Center	1
Car	48
CarHandler	48
Cruiser	48
Proximity Sensor	48
Terminal	6
Platform Manager	6
Entrance	12
Exit	12
Exit Manager	6
cDestPanel	48
tDestPanel	6

Table I. Process Classes and Object counts in Rail-car Example with 48 cars

them running clockwise and the other anti-clockwise. Each adjacent pair of these terminals is connected by the two parallel tracks. There is fixed number of rail cars 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 3, each terminal has four parallel platforms. At any time at most one car can be parked at a platform. Further, there are two *entrance* and two *exit* segments which connect the two main rail tracks to the terminal's platform tracks. 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.

The list of process classes and the number of concrete objects in each process class for a rail-car system with 48 cars is shown in Table I. Note that $tDestPanel$ represents the destination panel in the terminal and $cDestPanel$ represents the destination panel in the rail-car. Thus, with 48 cars and 6 terminals, we have 48 cDestPanel and only 6 tDestPanel (refer Table I). We have only one $ControlCenter$ object which is related to all the $Terminal$ and Car objects. Also each Car is associated to a $ProximitySensor$, which notifies the car when it arrives within 100 and 80 yards of some terminal, and also associated to a $Cruiser$ which maintains the car speed.

When a car is at a terminal or arrives in one, a unique $CarHandler$ gets associated with the Car to handle communication between the car and the terminal. Once the car leaves the terminal a $CarHandler$ is no longer associated with it. We have two $Entrance$ and two $Exit$ objects associated with each terminal. They 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.

We show a fragment of the IPC model of the Rail-car example in Figure 4. Controlling the movement of the cars between the terminals involves a complex description. The classes shown in Figure 4 are Car, $Cruiser$, $Terminal$ and $CarHandler$. The Cruiser stands for the cruise control of a car. This will be captured as asso-

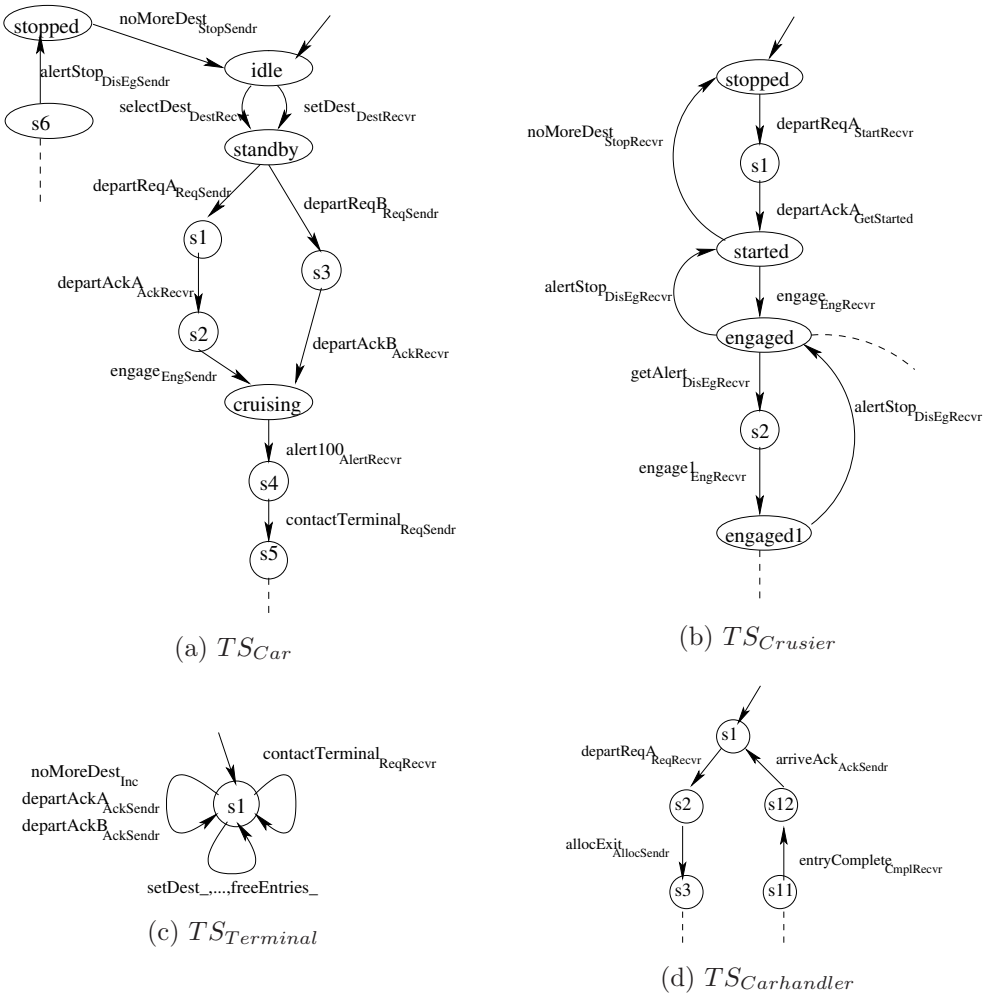


Fig. 4. **Fragment of Labeled Transition Systems for process classes of the Rail-car example — (a)Car (b)Cruiser (c)Terminal (d)Carhandler**

ciations via Class Diagrams as discussed in Section 7. The CarHandler manages interaction between an approaching/departing car and the corresponding terminal.

In Figure 4, for each process class, we have shown a fragment of the transition system corresponding to that process class. As explained in the last section, the action labels of the transition system for a process class specify a transaction and a role in that transaction (which we model using a Message Sequence Chart). We have not shown all the transactions corresponding to the (transaction, role) pairs appearing as action labels in Figure 4; there are too many of them. However, two of these transactions, namely, *departReqA* and *noMoreDest* appear in Figure 1.

In this preliminary model, we do not discuss associations; the class associations for the rail-car example appear in Figure 5. The modeling and simulation of pro-

cess classes with associations will be dealt with in Section 7 after developing the execution semantics of the core model.

5. CONCRETE EXECUTION SEMANTICS

We now formulate a **concrete execution semantics** of the IPC model. Given a set of transactions Γ and an IPC model $\{TS_p\}_{p\in\mathcal{P}}$ as defined in Section 3 (Definition 1), for any class p we define H_p to be the least set of minimal DFAs (Deterministic Finite State Automata) [Hopcroft and Ullman 1979] given by: $\mathcal{A}$ is in H_p iff there exists a transaction $\gamma = (R, I, Ch)$ and a role $r \in R$ 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, i.e.

$$H_p = \{dfa(\Lambda) | \gamma = (R, I, Ch) \in \Gamma \wedge r = (p, \rho) \in R \wedge I_r = (\Lambda, \Psi)\}. \quad (1)$$

Expression $dfa(\Lambda)$ in Eq. (1) above represents the minimal DFA corresponding to regular expression Λ.

To capture the state of an object we now define the notion of a *behavioral partition*

Definition 2. **Behavioral Partition** *Let the following be an IPC description.*

$$\{TS_p = \langle S_p, Act_p, \rightarrow_p, init_p, V_p, v_{init_p}\rangle\}_{p\in\mathcal{P}}$$

Let $H_p = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ be the set of minimal DFAs defined for class p (Eq.(1)). 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.

A *concrete* configuration is used to capture the "local states" of all objects of all process classes and is defined below.

Definition 3. **Concrete Configuration** *Given an IPC specification $\mathcal{S} = \{TS_p\}_{p\in\mathcal{P}}$ such that O_p represents the set of objects of process class p, a* concrete configuration *of $\mathcal{S}$ is defined as follows.*

$$\mathsf{cfg_c} = \{pmap_p : O_p \rightarrow BEH_p\}_{p\in\mathcal{P}},$$

where BEH_p is the set of all behavioral partitions of class p (Definition 2). The set of all concrete configurations of $\mathcal{S}$ is denoted as $\mathcal{C}^c_{\mathcal{S}}$.

The system moves from one concrete configuration to another by executing a transaction. A transaction $\gamma \in \Gamma$ can be executed at a concrete configuration $\mathsf{cfg_c} = \{pmap_p\}_{p\in\mathcal{P}}$ iff for each role $r = (p, \rho)$ of γ where r has the guard (Λ, Ψ), there exists a distinct object $o \in O_p$ such that $(s, q_1, \ldots, q_k, v) = pmap_p(o)$ and following conditions hold-

(1) $s \xrightarrow{(\gamma_r)} s'$ is a transition in TS_p
(2) For all $1 \leq i \leq k$, 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.

This implies that there exists a distinct object for each role r of γ, such that these objects can together execute the transaction γ. We let $objects(\gamma)$ represent the set of objects chosen to execute transaction γ. Computing the new configuration cfg'_c as a result of executing transaction γ in configuration cfg_c involves computing the new state or *destination* behavioral partition for each object $o \in objects(\gamma)$. For an object o playing the role r in transaction γ, such that its current state is given by the behavioral partition $pmap_p(o) = (s, q_1, \ldots, q_k, v)$, we use '$nstate_{\gamma_r}(o)$' to represent the corresponding destination behavioral partition $(s', q'_1, \ldots, q'_k, v')$, 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.

Thus, an object o in the state given by behavioral partition $pmap_p(o)$ moves to a new state given by behavioral partition $nstate_{\gamma_r}(o)$ by performing role r in transaction γ. The new concrete configuration cfg'_c as result of executing transaction γ in configuration cfg_c is obtained as follows-

$$\begin{aligned} \mathsf{cfg}'_\mathsf{c} &= \{pmap'_p | p \in \mathcal{P}\}, \text{ where} \\ \forall p \in \mathcal{P} \;.\; pmap'_p(o) &= \begin{cases} pmap_p(o), & o \in O_p \backslash objects(\gamma) \\ nstate_{\gamma_r}(o), & o \in O_p \cap objects(\gamma) \end{cases} \end{aligned}$$

Thus, for all the objects that do not participate in transaction γ, i.e. they are in the set $(\cup_{p \in \mathcal{P}} O_p) \backslash objects(\gamma)$, their state in the new configuration cfg'_c remains unchanged from cfg_c. While for all the objects which execute γ, their states in cfg'_c are computed as described above.

Example. For illustration, consider the example shown in Figure 4. Suppose c is a concrete 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* and have no regular expression based history guards. Then we have– $pmap_{Car}(Oc_1) = \langle stopped, 0 \rangle$, $pmap_{Car}(Oc_2) = \langle stopped, 1 \rangle$ and $pmap_{Car}(Oc_3) = \langle s2, 2 \rangle$.

—Three *Cruiser* objects, $O_1 \ldots O_3$ are residing in state *started* of $TS_{Cruiser}$ such that the histories of O_1 and O_2 satisfy the regular expression

$$(Act_{Cruiser})^\star.alertStop_{DisEgRecvr}$$

while the history of O_3 satisfies the regular expression

$$(Act_{Cruiser})^\star.departAckA_{GetStarted}.$$

Further, let '$(Act_{Cruiser})^\star.alertStop_{DisEgRecvr}$' be the only regular expression guard appearing in the *Cruiser's* specification. It can be easily verified that the minimal DFA, say $\mathcal{A}_1$, recognizing the language of this regular expression has only two states- let these be q_1 and q_2, where q_1 is the initial state and q_2 is the accepting state. Assuming no *Cruiser* variables, at current configuration c we have– $pmap_{Cruiser}(O_1) = pmap_{Cruiser}(O_2) = \langle started, q_2 \rangle$ and $pmap_{Cruiser}(O_3) = \langle started, q_1 \rangle$.

—Six *Terminal* objects, $Ot_1 \ldots Ot_6$ are residing in state $s1$ of $TS_{Terminal}$. Assuming no history based guards and local variables, we have– $pmap_{Terminal}(Ot_1) = \ldots = pmap_{Terminal}(Ot_6) = \langle s_1 \rangle$.

Suppose we want to execute transaction *noMoreDest* shown in Figure 1(b) at configuration c. As for the role (*Car,StopSendr*), 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 satisfy the history part of the guard associated with the role (*Cruiser,StopRecvr*), i.e. they are in the accepting state q_2 of DFA $\mathcal{A}_1$ representing this history guard. 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, histories and variable valuations unchanged from c, thus remaining in the same behavioral partitions as c. The objects Oc_1, O_1, Ot_1 will move to control states *idle*, *stopped*, $s1$ in their respective transition systems. Value of variable *dest* for Oc_1 remains unchanged, while the state of DFA $\mathcal{A}_1$ for O_1 is updated to q_1. Thus at the resulting configuration c' we have– $pmap'_{Car}(Oc_1) = \langle idle, 0 \rangle$, $pmap'_{Cruiser}(O_1) = \langle stopped, q_1 \rangle$ and $pmap'_{Terminal}(Ot_1) = \langle s1 \rangle$.

6. ABSTRACT EXECUTION SEMANTICS

We observe that various objects of a process class have same local state (i.e. they map to the same behavioral partition) at a given concrete configuration during execution, and are thus *behaviorally indistinguishable*. Further, for classes with unboundedly many instances at run-time, the concrete execution semantics will produce an infinite-state system. Thus, we formulate an **abstract execution semantics** of the IPC model, where we do not maintain states and identities of individual objects. Instead, during execution, the objects of a class are grouped into *behavioral partitions*.

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 specification $\mathcal{S}$. Now suppose c is a concrete configuration and an object O belonging to process class p has an execution history $\sigma \in Act_p^\star$ at c. Then at c, the state of object O is given by the behavioral partition $(s, q_1, \ldots, q_k, v)$ in case- O resides in $s \in S_p$ at c, q_j is the state reached in the DFA $\mathcal{A}_j \in H_p$ when it runs over σ for each j in $\{1, \ldots, k\}$, and the valuation of O's local variables is given by v. Thus, two p-objects O_1 and O_2 of process class p map to the same *behavioral partition* (at a concrete configuration) if and only if the following conditions hold.

—O_1 and O_2 are currently in the same state of S_p,

—Their current histories lead to the same state for all the DFAs in H_p, and

—They have the same valuation of local variables.

This implies that the computation trees of two objects in the same behavioral partition at a concrete configuration are isomorphic. We make use of this property for

dynamically grouping together objects of a process class into behavioral partitions. This is a strong type of behavioral equivalence to demand. There are many weaker possibilities but we will not explore them here.

6.1 Abstract Execution of Core Model

To explain how abstract execution takes place, we first define the notion of an "abstract configuration".

DEFINITION 4. **Abstract Configuration** *Let $\{TS_p\}_{p\in\mathcal{P}}$ be an IPC specification $\mathcal{S}$ such that each process class p contains N_p objects. An abstract configuration of the IPC is defined as follows.*

$$\mathsf{cfg_a} = \{count_p\}_{p\in\mathcal{P}}$$

- *$count_p : BEH_p \rightarrow \mathbf{N} \cup \{\omega\}$ is a mapping s.t.*

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

- *BEH_p is the set of all behavioral partitions of class p,*
- *ω represents an unbounded number.*

So $count_p(b)$ is the number of objects in partition b. The set of all abstract *configurations of an IPC $\mathcal{S}$ is denoted as $\mathcal{C}^a_{\mathcal{S}}$.*

We note that N_p can be a given positive integer constant or it can be ω (standing for an 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). For a process class p, N_p can be ω if, either a) the initial object count for p is explicitly specified as ω, in order to model and validate a system with *any* number of p objects, or b) objects of class p can be dynamically created and we specify a threshold object count, exceeding which the object count of p will become ω. We discuss the exact mechanism for object creation/deletion for a process class p later in Section 6.2. For convenience of explanation, we assume that N_p is a given constant in the rest of the paper.

Our abstract execution 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. How can our simulator check whether a specific transaction γ is *enabled* at an abstract configuration cfg? Since we do not keep track of object identities, we define the notion of *witness partition* for a role r, from which an object can be chosen to play the role r in transaction γ.

DEFINITION 5. **Witness partition** *Let $\gamma \in \Gamma$ be a transaction and $\mathsf{cfg_a} \in \mathcal{C}^a_{\mathcal{S}}$ be an abstract 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 $witness(r, \gamma, \mathsf{cfg_a})$, for r at $\mathsf{cfg_a}$ if*

(1) $s \xrightarrow{(\gamma_r)} s'$ is a transition in TS_p
(2) For all $1 \leq i \leq k$, 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* $\mathsf{cfg_a}$.

An "enabled transaction" at an abstract configuration can now be defined as follows.

DEFINITION 6. **Enabled Transaction** *Let γ be a transaction and $\mathsf{cfg_a} \in \mathcal{C}^a_{\mathcal{S}}$ be an abstract configuration. We say that γ is enabled at $\mathsf{cfg_a}$ iff for each role $r = (p, \rho)$ of γ , there exists a witness partition $witness(r, \gamma, \mathsf{cfg_a})$ 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.*

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 $r = (p, \rho)$ in transaction γ.

DEFINITION 7. **Destination Partition** *Let γ be an enabled transaction at an abstract configuration $\mathsf{cfg_a} \in \mathcal{C}^a_{\mathcal{S}}$ 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.*

We now describe the effect of executing an enabled transaction γ at a given abstract configuration $\mathsf{cfg_a}$. Computing the new abstract configuration $\mathsf{cfg'_a}$ as a result of executing transaction γ in configuration $\mathsf{cfg_a}$ involves computing the destination behavioral partition $beh' = dest(beh, \gamma, r)$ corresponding to witness partition $beh = witness(r, \gamma, \mathsf{cfg_a})$ for each role r of γ, and then computing the new count of objects for each behavioral partition. In other words, we have

$$\begin{aligned}\forall b \in BEH_p \;.\; count'_p(b) = count_p(b) \\ + |\{x \mid b = dest(w, \gamma, x) \wedge w = witness(x, \gamma, \mathsf{cfg})\}| \\ - |\{x \mid b = witness(x, \gamma, \mathsf{cfg})\}|\end{aligned}$$

where $count_p(b)$ and $count'_p(b)$ are the number of objects of class p appearing in the behavioral partition b in the abstract configurations $\mathsf{cfg_a}$ and $\mathsf{cfg'_a}$ respectively. Recall that BEH_p is the set of all behavioral partitions of p. Hence, given the abstract source configuration $\mathsf{cfg_a}$, the above formula determines the abstract destination configuration $\mathsf{cfg'_a}$.

Example. Consider $TS_{Cruiser}$ shown in Figure 4(b). Suppose we simulate the specification with 24 *Cruiser* objects (assume that other process-classes are also appropriately populated with objects) using abstract execution semantics. In the

transition system $TS_{Cruiser}$, only the transition $noMoreDest_{StopRecvr}$ is guarded using a non-trivial regular expression $Act^{\star}_{Cruiser}.alertStop_{RcvDisEng}$; the corresponding DFA, say $\mathcal{A}_1$, will have just two states as can be easily verified. Initially all the 24 objects will be in the *stopped* state of $TS_{Cruiser}$ with null history and this will correspond to the initial state, say $q1$, of $\mathcal{A}_1$. All these objects 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 a cruiser object, say O_1, executes (in cooperation with objects in other classes) the trace:

"$departReqA_{StartRecvr}$, $departAckA_{GetStarted}$, $engage_{EngRecvr}$, $alertStop_{DisEgRecvr}$"

O_1 will now reside in the control state *started*. Also, since $alertStop_{DisEgRecvr}$ is executed at the end, O_1's history will correspond to the non-initial state (call it $q2$) of the DFA $\mathcal{A}_1$. Subsequently suppose another cruiser object, say O_2, executes the trace: "$departReqA_{StartRecvr}$, $departAckA_{GetStarted}$". Then O_2 will also end up in the control state *started*. However, unlike O_1, the execution history of O_2 will correspond to $q1$, the initial state of $\mathcal{A}_1$. After the above executions we have three non-empty behavioral partitions for cruiser objects — (i) $\langle stopped, q1 \rangle$ which has 22 objects which have remained idle, (ii) $\langle started, q2 \rangle$ which has object O_1 and (iii) $\langle started, q1 \rangle$ which has object O_2. Objects in different behavioral partitions have different sets of actions enabled, thereby leading to different possible future evolutions. Now let object O_1 execute the action $noMoreDest_{StopRecvr}$. This will result in O_1 migrating from behavioral partition (ii) to (i) above. Thus, O_1 will be now indistinguishable from the 22 objects which have remained idle throughout. For all of these 23 objects, the action $departReqA_{StartRecvr}$ is now enabled. This is the manner in which objects migrate between different behavioral partitions during abstract execution.

Maximum Number of 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 $\mathcal{A} \in H_p$. Recall that H_p is the set of minimal DFAs accepting the regular expression guards of the various roles of different transactions played by class p (Eq.(1)). 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, the *Cruiser* class of the Rail-Car Example shown in Figure 4(b) can have at most 14 behavioral partitions since — (i) $TS_{Cruiser}$ has seven (7) states (not all of them are shown in Figure 4(b)), (ii) the Cruiser class has no local variables that is $V_{Cruiser} = \emptyset$ and (iii) only one of the regular expression guards involving a Cruiser object results in

a DFA with two states[2]; all other regular expression guards involving the Cruiser class are accepted by a single state DFA. Thus, the number of behavioral partitions of the Cruiser class is at most $7 * 2 = 14$ while the number of objects can be very large. In fact, in Section 9 we report experiments that the number of behavioral partitions encountered in actual abstract execution runs is often lower than the upper bound on number of partitions (48 Cruiser objects are divided into less than 6 partitions, see Table II).

6.2 Dynamic Process Creation/Deletion

We also support dynamic process *creation* and *deletion* by means of special transactions whose names are prefixed with *start_p* or *stop_p* for all $p \in \mathcal{P}$. We let *start_pX*/*stop_pX* denote any such special transaction name. A transaction *start_pX* (*stop_pX*) contains a single role r which does not contain any events, though role r may have a guard $I_r = (\Lambda, \Psi)$ as any other transaction. Transaction *start_pX* starts off a new process instance of class p, while transaction *stop_pX* terminates the process instance executing it. A *start_pX* transaction can appear in the transition system of a process class q, where q may be different from p (i.e. $q \neq p$), thus allowing an object of one process class to create an object of another process class. However, transaction *stop_pX* can only appear in TS_p, the transition system for process class p. We now discuss the effect of executing these transactions for abstract execution semantics.

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 specification and transaction *start_pX* appearing in TS_q (q may be different from p) be enabled at an abstract configuration $\mathsf{cfg_a}$ with $b = (s, q_1, \ldots, q_k, v)$ as a witness partition. Let N_p be the current object count of process class p, and q_i^0 be the initial state of DFA $\mathcal{A}_i \in H_p$, where $i \in \{1, \ldots, k\}$ and $k = |H_p|$. Then executing *start_pX* at $\mathsf{cfg_a}$ with b as the witness partition will result in the new configuration $\mathsf{cfg'_a}$ such that: i) a process instance of q executing *start_pX* will move from witness partition b to destination partition $b' = dest(b, start\_pX, r)$ resulting in $count'_q(b) = count_q(b) - 1$, $count'_q(b') = count_q(b') + 1$, and ii) a new p-process instance will start off in the initial partition given by $b_i = (init_p, q_1^0, \ldots, q_k^0, v_{init_p})$ such that, $count'_p(b_i) = count_p(b_i) + 1$, if the number of objects in process class p (i.e. N_p) is less than a given threshold value (say t_p) which is set by the user, and $count'_p(b_i) = \omega$ otherwise. We also update $N'_p = N_p + 1$, when N_p is less than the threshold value t_p; otherwise $N'_p = \omega$.

Similarly, if *stop_pX* is enabled at an abstract configuration $\mathsf{cfg_a}$ with $b = (s, q_1, \ldots, q_k, v)$ as a witness partition, then executing *stop_pX* at $\mathsf{cfg_a}$ with b as the witness partition will result in the new configuration $\mathsf{cfg'_a}$ such that $count'_p(b) = count_p(b) - 1$, when $count_p(b)$ is a constant number, and $count'_p(b) = \omega$ otherwise. Thus, we just decrement the count of witness partition b by 1 (when it is a constant), without incrementing the count of its corresponding destination partition. This indicates termination of the process instance executing this transaction. Similarly, we also update $N'_p = N_p - 1$, when N_p is constant; otherwise $N'_p = \omega$.

[2]This is the guard for the role $Cruiser_{StopRecvr}$ in transaction *noMoreDest*, see Figure 1(b).

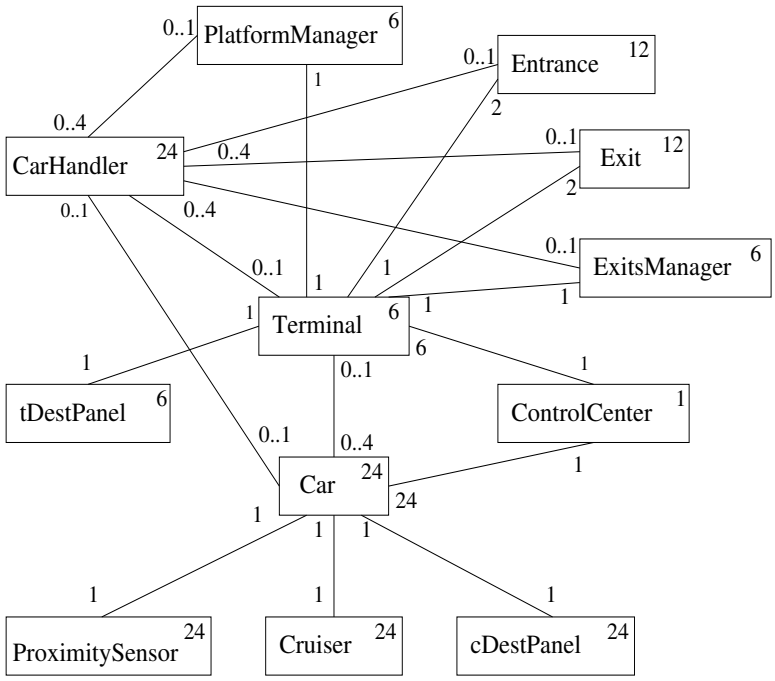


Fig. 5. **Class diagram for Rail-car example.**

7. ASSOCIATIONS

We now turn to extending our language with static and dynamic associations. This will help us to model different kinds of relationships (either structural or established through communications) that can exist between objects. The ability to track such relationships substantially increases the modeling power.

7.1 Modeling Static and Dynamic Associations

Our notion of static and dynamic associations is similar to the classification presented in [Stevens 2002].

Static Associations. A static association expresses a *structural relationship* between the classes. In a class-diagram, a static association is annotated with fixed multiplicities at both its ends. Static associations, as the name suggests, remain fixed and do not change at runtime. We can refer to static associations in transaction guards to impose the restriction that objects chosen for a given pair of agents should be statically related. The full class diagram for the Rail-car example with 24 cars appears in Figure 5. For example, the following pairs of classes: (PlatformManager, Terminal), (Terminal, ControlCenter), (Car, ControlCenter) and (Car, Cruiser) are statically associated in Figure 5. In particular, the association between Car class and the Cruiser class denotes the *itsCruiser* relation of a car with its cruiser. Note that we do not allow dynamic object-creation/deletion of a statically associated class– such as Car, Terminal etc. in the Rail-car example.

Dynamic Associations. A dynamic association expresses *behavioral relationship* between classes, which in our case would imply that the objects of two dynamically associated classes can become related to each other through exchange of messages (by executing transactions together) and then at some stage leave that relation. In the class-diagram, a dynamic association is annotated with varying multiplicities at both its ends.

7.2 Concrete execution of IPC models with associations

In our concrete execution semantics, for a k-ary association *asc* relating objects of process classes $p_1, \ldots, p_k$, the k-tuple(s) of objects $\langle O_1, \ldots, O_k \rangle$ following the association are stored in relation *asc* during execution . We now describe the steps for handling an arbitrary association *asc* of arity k involving process classes $p_1, \ldots, p_k$ during concrete execution.

—**Initialization:** For a k-ary static association *asc*, the k-tuples of objects following the association are inserted in relation *asc*. For example, if $k = 2$ (binary association), and *asc* is a one-to-one binary association between two process classes p_1, p_2 each containing n objects $O_1, \ldots, O_n (O'_1, \ldots, O'_n)$, we populate *asc* with n object pairs, $\langle O_1, O'_1 \rangle, \ldots, \langle O_n, O'_n \rangle$.
If *asc* is a dynamic association, for convenience we assume that initially *asc* does not contain any k-tuple of objects. However, its content can change during execution.

—**Check:** Let γ be a transaction and $r_1, \ldots, r_k$ be the roles in transaction γ with the guard $(r_1, \ldots, r_k) \in asc$. If $O_1, \ldots, O_k$ are the objects chosen to play the roles $r_1, \ldots, r_k$ respectively, we also require that the the k-tuple $\langle O_1, \ldots, O_k \rangle$ be present in the *asc* relation.

—**Insert:** Let γ be a transaction and $r_1, \ldots, r_k$ be the roles in transaction γ with the post-condition *insert* $(r_1, \ldots, r_k)$ *into asc*. Let $O_1, \ldots, O_k$ be the objects chose to play the roles $r_1, \ldots, r_k$ respectively. Upon executing γ, we insert the k-tuple $\langle O_1, \ldots, O_k \rangle$ in the relation *asc*. Note that the insert operation is possible only if *asc* is a dynamic association.

—**Delete:** Let γ be a transaction and $r_1, \ldots, r_k$ be the roles in transaction γ with the post-condition *delete* $(r_1, \ldots, r_k)$ *from asc*. If $O_1, \ldots, O_k$ are the objects chosen to play the roles $r_1, \ldots, r_k$ respectively, we require that the the k-tuple $\langle O_1, \ldots, O_k \rangle$ be present in the relation *asc*. Furthermore, we delete the above k-tuple from *asc* upon executing γ. Note that the delete operation is possible only if *asc* is a dynamic association.

Example. We now illustrate the use of *dynamic associations* using the Rail-car example. During execution, various rail-cars enter and leave the terminals along their paths. 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* transaction. Hence, the guard of *departAck(A/B)* requires that the participating *Car* and *Terminal* objects should have together executed *contactTerminal* in the past. Since this condition involves a relationship between the local histories of multiple objects, we cannot capture it via regular expressions over the

contactTerminal	**inserts**	(O1,O2) into	**itsTerminal**
where, O1 plays the role		**(Car, ReqSendr)**, and	
O2 plays the role		**(Terminal, ReqRecvr)**	
departAck(A/B)	**checks**	(O1,O2) belongs to	**itsTerminal**
where, O1 plays the role		**(Car, AckRecvr)**, and	
O2 plays the role		**(Terminal, AckSendr)**	
departAck(A/B)	**deletes**	(O1,O2) from	**itsTerminal**
where, O1 plays the role		**(Car, AckRecvr)**, and	
O2 plays the role		**(Terminal, AckSendr)**	

Fig. 6. **Dynamic Relation *itsTerminal***

individual local histories. Hence we make use of the 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,ReqSendr)* and *(Terminal,ReqRecvr)*,

—*departAckA* and *departAckB* have roles *(Car,AckRecvr)* and *(Terminal,AckSendr)*. Note that transactions *departAck(A/B)* also involve other roles which we choose to ignore here for the purpose of our discussion.

In the concrete execution, if car object O_c and terminal object O_t play the roles (*Car,ReqSendr*) and (*Terminal,ReqRecvr*) in *contactTerminal*, then the effect of *contactTerminal* is to insert the pair $\langle O_c, O_t \rangle$ into the *itsTerminal* relation (refer to Figure 6). The *departAck(A/B)* transaction's guard now includes the check that the object corresponding to the role (*Car,AckRecvr*) and object corresponding to role (*Terminal,AckSendr*) be related by the dynamic relation *itsTerminal*; so if objects O_c and O_t are selected to play the (*Car,AckRecvr*) and (*Terminal,AckSendr*) roles in *departAck(A/B)*, the check will succeed. Furthermore, the effect of *departAck(A/B)* transaction is to remove the tuple $\langle O_c, O_t \rangle$ from *itsTerminal* relation.

Thus, for a dynamic relation, the specifications will include the effect of each transaction on the relation in terms of insertion/deletion of tuples of objects into the relation. Furthermore, the guard of a transaction can contain a membership constraint ('check') on one or more of the specified static/dynamic relations. For execution of concrete objects, it is clear how our extended model should be executed. The question is how can we keep track of associations in the abstract execution semantics.

7.3 Abstract execution of IPC models with associations

For 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 *associations between behavioral partitions.* To illustrate the idea, consider a binary relation D which is supposed to capture some dynamic association between objects of the process class p. In our abstract execution, each element of D will be a pair (b, b') where b and b' are behavioral partitions of class p; furthermore for all

pairs $(b, b') \in D$ we also maintain a count indicating the number of concrete object pairs in behavioral partitions b, b' which are related via D. To understand what $(b, b') \in D$ means, consider the concrete execution of the process class p. If after an execution σ (a sequence of transactions), two concrete objects O, O' of p get D-related ($\langle O, O' \rangle \in D$) then the abstract execution along the same sequence of transactions σ must produce $\langle b, b' \rangle \in D$ where b (b') is the behavioral partition in which O (O') resides after executing σ. The same idea can be used to manage relations of larger arities. *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.

Formally, the set of abstract configurations (Definition 4) in our operational model remains unchanged. Recall from Definition 4 that an abstract configuration is defined as $\mathsf{cfg} = \{count_p\}_{p \in \mathcal{P}}$ where $count_p(b)$ is the number of objects in partition $b \in BEH_p$; BEH_p is the set of all behavioral partitions of class p. In the presence of a k-ary association asc relating objects of process classes $p_1, \ldots, p_k$ we maintain asc in our abstract execution as

$$BEH_{p_1} \times BEH_{p_2} \times \ldots \times BEH_{p_k} \rightarrow \mathbf{N} \cup \{\omega\}$$

The association is maintained by maintaining counts for k-tuples $\langle beh_1, \ldots, beh_k \rangle$ where $beh_1 \in BEH_{p_1}, \ldots, beh_k \in BEH_{p_k}$. Following are the steps for handling associations in our abstract execution. We describe the steps for an arbitrary association asc of arity k involving process classes $p_1, p_2, \ldots, p_k$.

—**Initialization:** For each process class p we have an initial variable valuation v_p^{init} and an initial state $init_p$ in the high-level LTS of class p. Consequently, we can compute an initial behavioral partition beh_p^{init} for each process class p. Now, if asc is a static association, we initialize the counts of k-tuples of behavioral partitions in the following way. For every k-tuple other than $\langle beh_{p_1}^{init}, \ldots, beh_{p_k}^{init} \rangle$ we set the asc count to be zero. For the k-tuple $\langle beh_{p_1}^{init}, \ldots, beh_{p_k}^{init} \rangle$, the count is non-zero and is obtained from the class diagram annotations. For example, if $k = 2$ (binary association), and asc is a one-to-one binary association between two process classes p_1, p_2 each containing n objects, we initialize the count for $\langle beh_{p_1}^{init}, beh_{p_2}^{init} \rangle$ to be n.
Now suppose asc is a dynamic association. For convenience we assume that initially asc does not contain any k-tuple of objects (which are in their initial state and have null histories). Consequently, the counts for *every* k-tuple of behavioral partitions is set to zero.

—**Check:** Let γ be a transaction and $r_1, \ldots, r_k$ be the roles in transaction γ with the guard $(r_1, \ldots, r_k) \in asc$. If $beh_1, \ldots, beh_k$ are the chosen witness partitions for $r_1, \ldots, r_k$ respectively, we also require that the asc count maintained for the k-tuple $\langle beh_1, \ldots, beh_k \rangle$ be greater than zero. Furthermore, let $beh'_1, \ldots, beh'_k$ be the destination partitions of $beh_1, \ldots, beh_k$ respectively, upon executing γ. We then decrement the asc count for the k-tuple $\langle beh_1, \ldots, beh_k \rangle$ by 1; the asc count for the k-tuple $\langle beh'_1, \ldots, beh'_k \rangle$ is incremented by 1.

—**Insert:** Let γ be a transaction and $r_1, \ldots, r_k$ be the roles in transaction γ with the post-condition *insert* $(r_1, \ldots, r_k)$ *into asc*. Let $beh'_1, \ldots, beh'_k$ be the destination partitions of the roles $r_1, \ldots, r_k$ respectively, upon executing γ. We increment

the *asc* count for the k-tuple $\langle beh'_1, \ldots, beh'_k \rangle$ by 1. Note that the insert operation is possible only if *asc* is a dynamic association.

—**Delete:** Let γ be a transaction and $r_1, \ldots, r_k$ be the roles in transaction γ with the post-condition *delete* $(r_1, \ldots, r_k)$ *from asc*. If $beh_1, \ldots, beh_k$ are the chosen witness partitions for $r_1, \ldots, r_k$ respectively, we require that the *asc* count maintained for the k-tuple $\langle beh_1, \ldots, beh_k \rangle$ be greater than zero. Furthermore, we decrement the *asc* count for the k-tuple $\langle beh_1, \ldots, beh_k \rangle$ by 1, upon executing γ. Note that the delete operation is possible only if *asc* is a dynamic association.

—**Default:** Let γ be a transaction and $r_1, \ldots, r_j$ $(1 \leq j \leq k)$ be the roles in transaction γ without any association guard or post-condition. Let $beh_1, \ldots, beh_j$ be the witness partitions chosen for $r_1, \ldots, r_j$ respectively, and $beh'_1, \ldots, beh'_j$ be the corresponding destination partitions upon executing γ. Then if the *asc* count maintained for the k-tuple $\tau = \langle beh_1, \ldots, beh_j, beh_{j+1}, \ldots, beh_k \rangle$ is greater than zero, the *asc* count for the k-tuple $\tau' = \langle beh'_1, \ldots, beh'_j, beh_{j+1}, \ldots, beh_k \rangle$ is incremented by 1.

The *default* case is required to update the *asc* count for a k-tuple such that–

(1) its *asc*-count is greater than 0, and
(2) a (non-empty) subset of its behavioral partitions participate as witness partitions in execution of a transaction γ, without any *asc* guard or post-condition over the roles they are chosen for.

Note that, the j objects chosen above from behavioral partitions $beh_1, \ldots, beh_j$ to participate in transaction γ may or may-not follow association *asc* with objects in $beh_{j+1}, \ldots, beh_k$. Hence, we over-approximate by not decrementing the *asc*-count for k-tuple τ, but incrementing it for the destination k-tuple τ' by 1.

It might seem that our maintenance of association information will lead to undue blow-up. This is because we maintain counts corresponding to k-tuples of behavioral partitions. However, typically we only have *binary* associations in the class diagrams of the IPC specifications. So, we only need to maintain counts for pairs of behavioral partitions. Furthermore, very few of these pairs have non-zero counts during execution, and we only need to maintain pairs which have non-zero counts.

Example. As discussed earlier, the dynamic relation *itsTerminal* is maintained between the objects of class *Car* and *Terminal* (as shown in Figure 6). 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,ReqSendr*) and (*Terminal,ReqRecvr*) 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 $contactTerminal_{ReqSendr}$ ($contactTerminal_{ReqRecvr}$). So in our abstract execution, corresponding to pairs of behavioral partitions of the Car and $Terminal$ class, we maintain a count indicating the number of pairs in the *itsTerminal* relation. Thus, for the pair $\langle b_{Car}, b_{Term} \rangle$ we increment its count by 1.

Now when we execute *departAck(A/B)* transaction, we will pick a pair from this relation as witness behavioral partitions for the roles (*Car,AckRecvr*) and (*Terminal,AckSendr*). We *have not* maintained information about which *Terminal* object

in b_{Term} is related to which *Car* of b_{Car}. In our abstract execution, when we pick b_{Car} and b_{Term} as witness partitions of two roles in transaction *departAck(A/B)*, we are assuming that the corresponding objects of b_{Car} and b_{Term} which are associated via *itsTerminal* are being picked. Furthermore, after executing *departAck(A/B)* transaction, we decrement the count for pair $\langle b_{Car}, b_{Term} \rangle$ by 1.

8. EXACTNESS OF ABSTRACT SEMANTICS

In this section we first show that our abstract execution semantics is an over-approximation in the sense that every concrete execution can be realized under the abstract execution semantics but the converse, in general, is not true. We then describe a procedure for checking whether an abstract execution run is a spurious one.

8.1 Over-Approximation Results

In what follows, we only consider finite sequence of transactions. After all, traces produced by (concrete or abstract) execution are always finite.

THEOREM 1. *Suppose σ is a finite sequence of transactions that can be exhibited in the concrete execution of an IPC model $\mathcal{S}$. Then σ can also be exhibited in the abstract execution of $\mathcal{S}$.*

The proof of Theorem 1 proceeds by induction on N, the length of the execution sequence σ. The detailed proof appears in Appendix A.

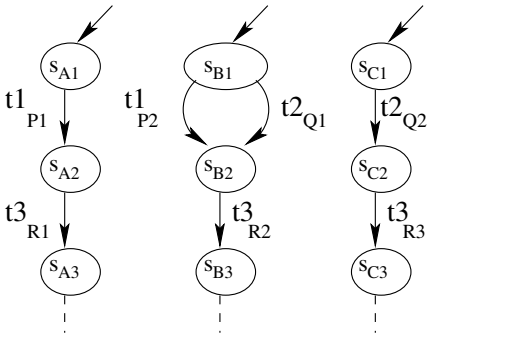
We next note that the converse of the above theorem holds in the *absence* of associations.

THEOREM 2. *Suppose $\mathcal{S}$ is an IPC model which has no association relations appearing in the guards of any of the transactions. Then every finite sequence of transactions under the abstract execution semantics is also an execution sequence under the concrete execution semantics.*

PROOF. The proof follows by a straightforward induction on the length of abstract execution run. The induction hypothesis is:

Let σ be a finite sequence of transactions allowed in the abstract execution semantics of an IPC model $\mathcal{S}$. Let $beh \in BEH_p$ be a behavioral partition of class p with count $= n$ after the abstract execution of σ. Then σ is also a concrete execution. Furthermore, after the concrete execution of σ there exists exactly n concrete objects of class p which reside in partition beh based on their control state, execution history and variable valuation.

Let $\sigma = \sigma^{prev} \circ \gamma$ and the induction hypothesis holds for σ^{prev}. In the induction step, we need to show that the above holds after the execution of $\sigma = \sigma^{prev} \circ \gamma$ as well. Let $r_1, \ldots, r_m$ be the roles of transaction γ and let $beh_1, \ldots, beh_m$ be their witness behavioral partitions in the abstract execution of γ. By the induction hypothesis, we have concrete objects $o_1, \ldots, o_m$, whose states are given by the behavioral partitions $beh_1, \ldots, beh_m$, to play the roles $r_1, \ldots, r_m$ in the concrete execution of γ. Furthermore, if $beh'_1, \ldots, beh'_m$ are the destination partitions of $beh_1, \ldots, beh_m$ after the abstract execution of γ, we are guaranteed that $o_1, \ldots, o_m$ will move to $beh'_1, \ldots, beh'_m$ after the concrete execution of γ. This follows from-



t1	**checks**	(P1,P2)	is in	**Asc1**
t2	**checks**	(Q1,Q2)	is in	**Asc2**
t3	**checks**	(R1,R2)	is in	**Asc1**
	and	(R2,R3)	is in	**Asc2**

Fig. 7. **An example to show spurious runs in our abstract execution semantics**

a) the definition of a destination partition, Def. 7, and b) the method for computing the behavioral partition representing the new state of an object in concrete execution semantics (Section 5). Now, in abstract execution, the object count of each of the witness (destination) partition $beh_i(beh_i')$ will be decremented (incremented) by 1 after executing γ. Similarly, after the concrete execution of γ, the number of objects whose state is given by the behavioral partition $beh_i(beh_i')$, will be decremented (incremented) by 1. Thus, the induction step is established. □

8.2 Spurious abstract executions

We now show that the converse of theorem 1 does not hold, i.e. *a finite sequence of transactions exhibited by the abstract execution of an IPC model $\mathcal{S}$ may not be exhibited by the concrete execution of $\mathcal{S}$.* Consider a fictitious 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. In other words, Asc_1 (Asc_2) captures the relationship between a car and *itsCruiser* (*itsBrakeController*). Fragments of the transition systems for these components are shown in Figure 7, along with the checks on the static associations by various transactions. Assume that there are no variables declared in these process classes and that all the action labels shown in the example have trivial guards, that is they do not impose any restriction on the execution history of the object to play that role (of course the object should be in the appropriate control state). Suppose now, that we have an initial abstract configuration

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

Process classes *Cruiser*, *Car* and *BrakeControl* contain 2 objects each, in their initial states s_{A1}, s_{B1} and s_{C1} respectively.

Furthermore, for the association Asc_1 (representing *itsCruiser* relationship), the count associated with the pair $\langle s_{A1}, s_{B1}\rangle$ is 2, and for the association Asc_2 (representing *itsBrakeController* relationship), the count associated with the pair $\langle s_{B1}, s_{C1}\rangle$ is 2.

It is easy to see that the abstract execution semantics allows the sequence of transactions $t1, t2, t3$. After a car object and its cruiser execute $t1$, abstract configuration reached is

$$c1 = \{(\langle s_{A1}\rangle, 1), (\langle s_{A2}\rangle, 1), (\langle s_{B1}\rangle, 1), (\langle s_{B2}\rangle, 1), (\langle s_{C1}\rangle, 2)\}.$$

Also, for the association Asc_1, the count associated with the pair $\langle s_{A1}, s_{B1}\rangle$ now becomes 1 (it is decremented by 1), and incremented by 1 for the pair $\langle s_{A2}, s_{B2}\rangle$. There is no change in the association content for Asc_2.

Since the car object executing $t1$ (call it Car1 for convenience of explanation) is now in state s_{B2} it cannot execute transaction $t2$ since it is not enabled from s_{B2}. Suppose now $t2$ is executed by another car object (call it Car2 for convenience of explanation). This produces the configuration

$$c2 = \{(\langle s_{A1}\rangle, 1), (\langle s_{A2}\rangle, 1), (\langle s_{B2}\rangle, 2), (\langle s_{C1}\rangle, 1), (\langle s_{C2}\rangle, 1)\}.$$

For association Asc_2, the count associated with the pair $\langle s_{B1}, s_{C1}\rangle$ is decremented by 1, and incremented by 1 for the pair $\langle s_{B2}, s_{C2}\rangle$. There is no change in the association content for Asc_1.

In our abstract execution, the two car objects are *not distinguishable* at this point since they are both in state s_{B2}. One of these cars (actually Car1) has its cruiser in state s_{A2} from where transaction $t3$ is enabled; another car (actually Car2) has its brake controller in state s_{C2} from where $t3$ is enabled. But since the distinction between $Car1$ and $Car2$ is not made in abstract execution, transaction $t3$ (involving all the classes — Car, Cruiser, BrakeControl) will be executed in the abstract execution. In particular note that in the association information for Asc_1 (Asc_2), the count associated with $\langle s_{A2}, s_{B2}\rangle$ ($\langle s_{B2}, s_{C2}\rangle$) is greater than zero. This will allow $t3$ to be executed "as per" our abstract execution semantics.

In the concrete execution, however $t3$ cannot be executed after transactions $t1, t2$ are executed. After executing transactions $t1, t2$ there *cannot be any concrete car object* which has its cruiser (related by association Asc_1) as well its brake controller (related by association Asc_2) in the appropriate control states for executing transaction $t3$. Thus, if trace σ is simulated in the concrete execution, it will get *deadlocked* after executing the transactions $t1, t2$. Though in this example we have only considered static associations, similar incompleteness of our abstract execution can be shown with dynamic associations.

8.3 Detecting spurious abstract executions

Detecting spurious abstract executions is similar in objective to detecting spurious counter-example traces in abstraction-refinement based software model checking (e.g. see [Henzinger et al. 2002]). In our setting, this can be done effectively.

THEOREM 3. *There is an effective procedure which accepts as input an IPC* $S = \{TS_p\}_{p\in\mathcal{P}}$ *and a finite sequence* σ *which is an execution sequence under the*

abstract execution semantics, and determines whether or not σ is a spurious execution sequence; in other words, σ is not an execution sequence under the concrete semantics.

PROOF. Let $\sigma = \gamma^1 \ldots \gamma^n$ be a finite sequence of transactions from an IPC S which is allowed under our abstract execution semantics.

For each process class p, let $num_{p,\sigma}$ denote an upper-bound on the number of p-objects required for exhibiting the execution sequence σ. We define $num_{p,\sigma}$ to be the total number of roles (p, ρ) appearing in transaction sequence σ s.t. for each such role (p, ρ) in a transaction occurrence γ^i in σ, γ^i_ρ is an outgoing transition from the initial state of TS_p (the transition system for process class p). This is because, the number of unique p-objects that can participate in transactions in σ, must have initially executed a role (p, ρ) in a transaction γ^i (occurring in σ) s.t. γ^i_ρ is an outgoing transition from the initial state of TS_p. Note that, a transaction γ can occur more than once in an execution trace σ. For computing $num_{p,\sigma}$, we treat each transaction occurrence as distinct.

We now 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. For the execution trace σ, we can say that 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. To show this we consider the following two cases:

(1) $N_p \leq num_{p,\sigma}$ for each process class $p \in \mathcal{P}$. In this case $x_{p,\sigma} = N_p$ for all p and hence the given system and the finite state system are equivalent.

(2) $N_p > num_{p,\sigma}$ for some process classes $p \in \mathcal{P}$. Then $x_{p,\sigma} = num_{p,\sigma}$ for the process classes having $N_p > num_{p,\sigma}$, and $x_{p,\sigma} = N_p$ for the remaining process classes. From our earlier argument that $num_{p,\sigma}$ gives an upper bound on the number of p-objects for each class p to exhibit the trace σ, if this finite state system exhibits σ, it must be exhibited by the concrete execution of the given system with N_p objects for each class p. The reverse direction follows from the reasoning that no more than $num_{p,\sigma}$ objects of class p can participate in one or more transactions of trace σ, even if the system has more than $num_{p,\sigma}$ objects of class p.

□

Using an illustrative example, we now show how a spurious run is detected for a given IPC system $\mathcal{S}$ and a trace σ, by first deriving a *finite state* system from $\mathcal{S}$ corresponding to σ. Again, we consider the example discussed in Section 8.2 (Figure 7), consisting of three process classes- *Cruiser*, *Car* and *BrakeControl*. Also, consider the trace $\sigma = t_1.t_2.t_3$ which can be exhibited in the given system following our abstract execution semantics, as was demonstrated in Section 8.2.

Suppose we now want to check whether or not σ is spurious. From Figure 7 we obtain the roles for transactions in σ– for t_1, roles are (*Cruiser,P1*) and (*Car,P2*), for t_2, roles are (*Car,Q1*) and (*BrakeControl,Q2*), and roles in transaction t_3 are (*Cruiser,R1*), (*Car,R2*) and (*BrakeControl,R3*). We now compute $num_{p,\sigma}$ for each

process class in the system. First, we consider the *Cruiser* class. It participates in transactions t_1 and t_3 in σ, playing the roles (*Cruiser,P1*) and (*Cruiser,R1*) in these transactions respectively. Only one of these roles, (*Cruiser,P1*) in transaction t_1, is played from the initial state s_{A1} of the transition system describing *Cruiser* (Figure 7). Thus, $num_{Cruiser,\sigma} = 1$. Similarly we can determine that $num_{Car,\sigma} = 2$ and $num_{BrakeControl,\sigma} = 1$. As $N_{Cruiser} = N_{Car} = N_{BrakeController} = 2$ in the given system $\mathcal{S}$, we get $x_{Cruiser,\sigma} = 1$, $x_{Car,\sigma} = 2$ and $x_{BrakeControl,\sigma} = 1$. Now to detect whether or not σ is spurious in the given system $\mathcal{S}$, we only need to check if σ is a valid execution trace of the finite state system obtained above. Note that we have already shown σ to be spurious for the given system $\mathcal{S}$ in Section 8.2. Following the similar reasoning, σ can easily be shown spurious for this finite state system as well, leading to a *deadlock* after the execution of transactions t_1, t_2.

We have implemented the above procedure using the Murphi model checker [Murphi 2005]. This model checker has in-built support for symmetry reduction [Ip and Dill 1996] which can be exploited in the IPC setting. We present the details of the implementation in Appendix B.

9. EXPERIMENTS

We have implemented our abstract execution method by building a simulator in *OCaml* [OCaml 2005], a general purpose programming language supporting functional, imperative and object-oriented programming styles.

9.1 Modeled Examples

For our initial experiments, we modeled a simple telephone switch drawn from [Holzmann 2004]. It consists of a network of switch objects with the network topology showing the connection between different geographical localities. Switch objects in a locality are connected to phones in that locality as well as to other switches as dictated by the network topology. We modeled basic features such as local/remote calling as well as advanced features like call-waiting. Next we modeled the rail-car system whose behavioral requirements have been specified using Statecharts in [Harel and Gery 1997] and using Live Sequence Charts in [Damm and Harel 2001]. As mentioned in Section 4, Rail-Car system is a substantial sized system with a number of process classes: car, terminal, cruiser (for maintaining speed of a rail-car), car-handler (a temporary interface between a car and a terminal while a car is in that terminal), etc.

We have also modeled the requirement specification of two other systems - one drawn from the rail transportation domain and another taken from air traffic control (see `http://scesm04.upb.de/case-studies.html` for more details of these examples). We now briefly describe these two systems. The automated rail-shuttle system [RailShuttle_ System] 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 traveling a specified distance, for which it has

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
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
Automated Shuttle	Shuttle Agent	60	6	5	6
Weather Update	Clients	20	3	3	3

Table II. **Maximum Number of Behavioral partitions observed during abstract simulation**

to pay. Also, in case a shuttle is bankrupt (due to payment of fines), it is retired. The weather update controller [CTAS] is a an important component of the *Center TRACON Automation System*, 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 and the controller or communications manager. The latest weather update is presented by the weather control panel to various connected clients, via the controller. This update may succeed or fail in different ways; furthermore, clients get connected/disconnected to the controller by following an elaborate protocol.

9.2 Use Cases

We used guided abstract simulation on each of our examples to test out the prominent use cases. The details of these experiments appear in Table II. 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.

For the Rail-car example we simulate the following test cases– (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 ter-

Example	Setting	Time (sec)			Memory (MB)		
		C	A	**C/A**	C	A	**C/A**
Telephone switch	60 phones	2.0	1.5	**1.3**	87	63	**1.4**
	120 phones	4.1	1.5	**2.7**	189	64	**3.0**
Rail-Car	24 cars	3.9	2.1	**1.9**	173	83	**2.1**
	48 cars	7.0	2.2	**3.2**	353	84	**4.2**
Automated Shuttle	30 cars	0.7	0.4	**1.6**	33	18	**1.8**
	60 cars	1.2	0.4	**2.7**	69	18	**3.8**
Weather Update	10 clients	0.6	0.5	**1.2**	21	18	**1.2**
	20 clients	0.8	0.5	**1.6**	27	18	**1.5**

C ≡ Concrete Exec., A ≡ Abstract Exec.

Table III. **Timing/Memory Overheads of Concrete Execution and Abstract Execution**

minal, (b) cars moving from a busy terminal to less busy terminals while stopping at every terminal, and (c) cars moving from one terminal to another while not stopping at certain intermediate terminals. In the rail shuttle-system example, again we report the results for three test runs corresponding to (a) timely completion of order by shuttle leading to payment, (b) late completion of order leading to penalty, and (c) shuttle being unable to carry out order as it gets late in loading the order. Finally, for the weather update controller, we report the results of simulating three test cases corresponding to (a) successful update of latest weather information to all clients, (b) unsuccessful weather update where clients revert to older weather settings, and (c) unsuccessful update leading to disconnecting of clients.

For each test case, we report the object count for each process class as well as the maximum number of behavioral partitions observed during simulation. We have reported the results for only process classes with more than one 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. From Table II, we can see that the number of behavioral partitions is substantially less than the number of concrete objects. Furthermore, even if the number of concrete objects is increased (say instead of 48 cars in the Rail-car example, we have 96 cars), the number of behavioral partitions in these simulation runs remain the same.

9.3 Timing and Memory Overheads

Since one of our main aims 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 timing and memory usage of several randomly generated simulation runs of length 1000(i.e. containing 1000 transactions) in our examples and considered the maximum resource usage 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 abstract simulator. Simulations were run on a Pentium-IV 3 GHz machine with 1 GB of main memory. The results are shown in Table III. For each example we show the time and memory usage for both the

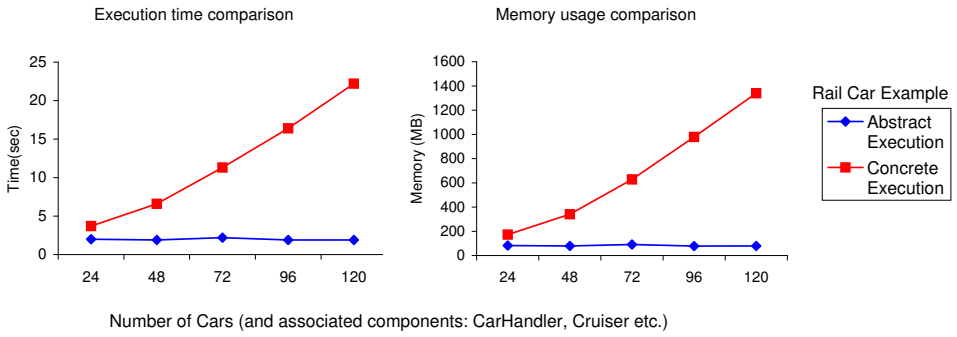


Fig. 8. **Execution Time and Memory Usage for different settings of the RailCar example.**

abstract and concrete simulation. Also, for a given example, we obtained results for two different settings, where the second setting was obtained by doubling the number of objects in one or more of the classes, e.g. in the *rail-car* example with 24 and 48 cars respectively.

We observe that for a given example and a given number of objects, the running time and memory usage for the concrete simulator are higher than that for the abstract simulator. Also for the same example but with higher number of objects, in case of abstract execution, the time/memory remain roughly the same, whereas they increase for the concrete case (as indicated by the increase in ratio C/A for higher number of objects in Table III).

Furthermore, in the graphs shown in Figure 8, we compare the growth in timing and memory usage in the railcar example, for both concrete and abstract simulations. Each successive setting is obtained by increasing the number of cars and its associated components: "car-handler", "proximity-sensor", "cruiser" and "dest-panel" by 24. Clearly our abstract execution allows the designer to try out different settings of a model by varying the number of objects without worrying about time/space overheads.

9.4 Test-case generation from IPC models

We have also used our IPC execution semantics for test generation. In particular, we generate a set of witness traces from an IPC model, which satisfy a high-level test purpose given as a sequence of transactions. This serves the purpose of– a) validating the requirements from which the IPC model has been derived, and b) the generated traces can be used as test cases to test an implementation derived from the same requirements.

The user gives a sequence of transactions $\gamma_1\gamma_2\ldots\gamma_n$ as a test purpose. At this level of abstraction it is much more intuitive and easier for user to design test specifications corresponding to various *use case* scenarios. The test-case generation procedure aims at producing one or more test sequences of the form

$$\gamma_1^1\ldots\gamma_1^{i_1}\gamma_1\gamma_2^1\ldots\gamma_2^{i_2}\gamma_2\ldots\gamma_n^{i_n}\gamma_n$$

This can be viewed as finding a witness trace satisfying the Linear-time Temporal

Logic (LTL) property [Clarke et al. 2000]-

$$\mathbf{F}(\gamma_1 \wedge (\mathbf{F}(\gamma_2 \wedge (\ldots(\mathbf{F}\gamma_n)\ldots)))$$

We always generate only finite witness traces (i.e., a finite sequence of transactions) such that any infinite trace obtained by extending our finite traces will satisfy the above mentioned LTL property.

Example	Test Spec. #	Expl. Depth	# Witnesses			Exec. Times(sec)		
			C	A	**C/A**	C	A	**C/A**
Telephone Switch (5 phones)	1	5	560	70	**8**	28	1	**28**
	2	8	80	12	**6.67**	138	7	**19.71**
	3	9	80	12	**6.67**	212	17	**12.47**
Rail-Car (6 cars, 3 terminals)	1	18	–	6	–	> 1 hr.	380	–
	2	15	–	32	–	> 1 hr.	19	–
	3	15	–	62	–	> 1 hr.	19	–
Automated Shuttle (5 shuttles)	1	15	11	5	**2.2**	15	0.7	**21.43**
	2	15	18	9	**2**	15	0.72	**20.83**
	3	17	5	2	**2.5**	63	2	**31.5**
Weather Update (10 clients)	1	20	129	5	**25.8**	2.5	0.1	**25**
	2	20	3	2	**1.5**	2.5	0.1	**25**
	3	25	7	3	**2.33**	12	0.15	**80**

Table IV. Comparing Abstract and Concrete test generation, A ≡ Abstract, C ≡ Concrete.

Experiments. Again, the experiments were performed for the IPC models of the four controller examples described earlier on a Pentium-IV machine with 3GHz CPU and 1GB of main memory. We compared the test-suite size and generation times obtained via *abstract* execution semantics with corresponding numbers for *concrete* execution semantics (where the state of each object is maintained separately). The results appear in Table IV. *The results serve as a measure of importance of abstract execution for model based test generation.* For each test purpose, the state space of the models were explored (using both abstract and concrete execution semantics) in *breadth- first* manner up to a given depth, and the total number of all possible witness-traces corresponding to the given test purpose were recorded within that exploration tree. The time to generate the test cases is much lower in abstract execution as compared to concrete execution. More importantly, using abstract execution we can group many behaviorally similar tests into a single test case. For example in the first test specification of the Telephone Switch example (see Table IV), 560 concrete test cases are grouped into 70 abstract tests. Note that the number of concrete tests is not always an exact multiple of the number of abstract tests. This is because different abstract tests may be blown up into a different number of concrete tests.

We also tried the notion of transaction coverage based test suite generation for the IPC models described earlier, where the aim is to generate a set of witness traces covering all the transactions in the system model at least once. Again, the transaction coverage was conducted for abstract execution of the models. *If we try to achieve transaction coverage by concrete execution, the test-suite construction does not even terminate in reasonable time for most examples.* Using abstract

execution the coverage was achieved within 20 seconds for all examples and the test-suite sizes thereby produced ranged from 10 to 700.

9.5 Checking for spurious execution runs

Recall that in the presence of associations, our abstract execution semantics is sound but incomplete. Consequently, we may encounter execution runs which are "spurious", that is, do not appear in any concrete system execution. In Section 8.3, we have presented a decision procedure for checking whether a finite sequence of transactions produced by our abstract simulator is spurious. As mentioned, we used the Murphi model checker to implement this spuriousness check.

During our experiments, we found that the spuriousness check for all the test cases of all our examples was completed in less than 0.1 second using Murphi. Also, when simulating an example system against meaningful use cases, the execution run produced by our abstract simulator was typically not spurious. In fact, there was only one false positive among all the test cases we tried for all the examples. This is to be expected, since we use our simulator to try out meaningful/prominent use-cases for a given system specification in the IPC model.

Also, in order to evaluate the overheads involved, we implemented an **exact** version of abstract execution semantics, with the difference being that for any object following an association, we treat it as a concrete object and do not merge it with other objects in the same behavioral partition as this object. This will eliminate the spuriousness, since any loss of information with respect to associations will be avoided. For the random simulation runs, we observed that the execution times and memory usage for the *exact* abstract execution semantics lies between those for the abstract and concrete semantics; thus making it an attractive alternative for our abstract execution semantics for simulation purposes. However, we found that the *exact* abstract semantics are still not suitable for experiments involving state-space exploration of the models- such as model checking, test generation, etc. This was demonstrated in the case of Rail-Car example for witness trace generation experiments, where the experiment did not even terminate for one of the test-specification and took nearly 10 minutes in the other two cases. Note that, for the first test-specification, abstract semantics took 380 seconds, while only 19 sec. were required for remaining two cases (see results for Rail-Car example in Table IV).

9.6 Debugging Experience

In this section, we share some experiences in reactive system debugging gained using our simulator tool. In particular, we describe our experiences in debugging the weather-update control system [CTAS]. 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. Even for the snippet shown in Figure 9, the transition system shown is a slightly simplified version of our actual modeling. 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 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 a 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.

In future, we plan to employ error *localization* techniques (*e.g.* dynamic slicing) on problematic simulation runs. *Full details of the simulator (along with its source code) are available from the web-site* `http://www.comp.nus.edu.sg/~release/simulator`

10. DISCUSSION

In this paper, we have studied a modeling formalism accompanied by an execution technique for dealing with interacting process classes; such systems arise in a number of application domains such as telecommunications and transportation. Our

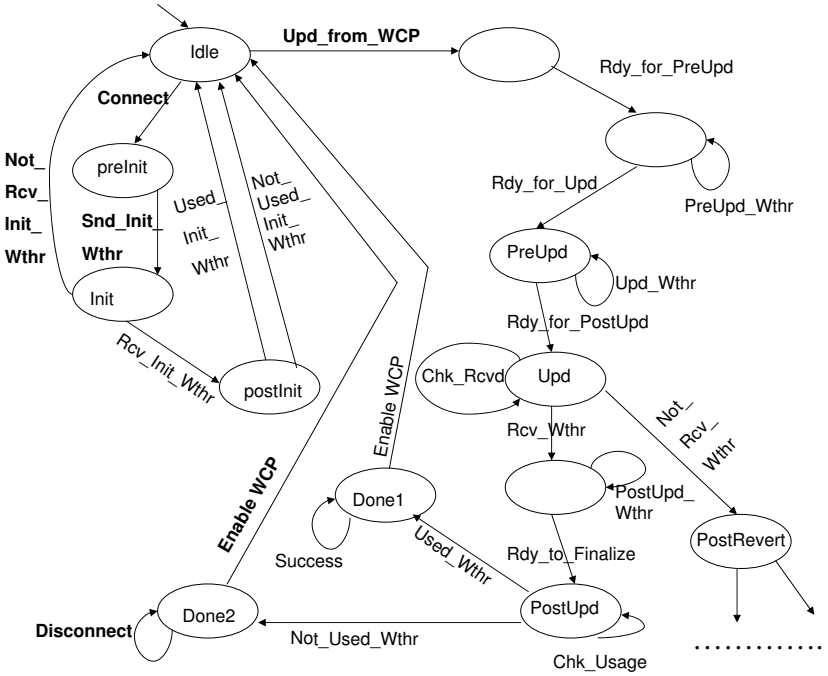


Fig. 9. **Snippet of Transition System for Weather-Update Controller**

models are based on standard notations for capturing behaviors and our abstract execution strategy allows efficient simulation of realistic designs with large number of objects. The feasibility of our method has been demonstrated on realistic examples.

In the present work, our state and class diagrams are "flat"; we plan to extend this in future. We are also integrating timing features in our modeling framework that would enable us to specify timing constraints such as: message delays and upper/lower time bounds on a process to engage in certain events.

Also, currently we do not support class inheritance, either structural or behavioral. The structural inheritance primarily aims at reusing existing class definitions and possibly adding new behaviors, or redefining existing ones. However, this does not guarantee any form of behavioral conformance between the subtype and the supertype. On the other hand, the notion of behavioral subtyping plays a crucial role in object oriented systems by allowing an object of a subtype to replace the object of its parent type, without changing the overall system behavior. One of the early works in this area is by Liskov and Wing [Liskov and Wing 1994] which focuses on passive objects – objects whose state change is only via method invocation by other objects. Subsequently, behavioral subtyping of active objects has been studied in many works (*e.g.* [Basten and van der Aalst 2001; Harel and Kupferman 2002; Wehrheim 2003]). These works mostly exploit well-known notions of behavioral inclusion (such as trace containment or simulation relations) to define notions

of behavioral subtyping. In future, we aim to incorporate similar notion(s) of behavioral inclusion to allow reuse of existing process classes, and define an efficient mechanism for checking substitutability of a subtype for its supertype.

Finally, we plan to develop a verification framework centered on our abstract execution semantics that will exploit the abstraction-refinement based approach to software model checking. The abstraction refinement framework can be used to find which associations need to be tracked in the abstract execution semantics, in order to avoid spurious run(s). This is important since we may have transaction guards of the form $(r_1, r_2) \in asc_1 \wedge (r_2, r_3) \in asc_2$ where r_1, r_2, r_3 are transaction roles. Consequently, it will not be sufficient to track only associations asc_1 and asc_2 appearing in the system specification. Instead, we also need to track "derived" associations during abstract execution; in the above example the relation formed by the join of the asc_1, asc_2 relations is one such association. This is similar in flavor to predicate abstraction based abstraction refinement [Ball et al. 2001; Henzinger et al. 2002] — where tracking the predicates/conditions appearing in the program is not sufficient, and abstraction refinement gradually finds out the additional predicates to track.

Acknowledgments

This work was partially supported by two research grants from Singapore's *Agency for Science, Technology and Research (A*STAR)* — one under Public Sector Funding and another under the Embedded & Hybrid Systems programme.

REFERENCES

Alur, R., Holzmann, G., and Peled, D. 1996. An analyzer for message sequence charts. In *Intl. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS), LNCS 1055*.

Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. 2001. Automatic predicate abstraction of C programs. In *PLDI*.

Basten, T. and van der Aalst, W. 2001. Inheritance of behavior. *Journal of Logic and Algebraic Programming 47,* 2.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM 50,* 5.

Clarke, E. M., Grumberg, O., and Long, D. E. 1994. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems 16,* 5.

Clarke, E. M., Grumberg, O., and Peled, D. A. 2000. *Model Checking*. The MIT Press.

CTAS. Center TRACON automation system. `http://ctas.arc.nasa.gov/`.

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

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

Goel, A., Meng, S., Roychoudhury, A., and Thiagarajan, P. S. 2006. Interacting process classes. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*. ACM Press. `http://www.comp.nus.edu.sg/~abhik/pdf/icse06.pdf`.

Harel, D. and Gery, E. 1997. Executable object modeling with statecharts. *IEEE Computer 30,* 7.

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

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

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

Holzmann, G. 2004. *Modeling a Simple Telephone Switch*. The SPIN Model Checker. Addison-Wesley, Chapter 14.

Hopcroft, J. E. and Ullman, J. D. 1979. *Introduction to automata theory, languages, and computation*. Addison-Wesley.

I-Logix. I-Logix, Inc. website: `http://www.ilogix.com`.

Ip, C. and Dill, D. 1996. Better verification through symmetry. *Formal Methods in System Design 9,* 2.

Jensen, K. 1995. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*. Vol. 1. Springer.

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

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

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

OCaml. 2005. The OCaml programming language. `http://caml.inria.fr/ocaml/index.en.html`.

Pnueli, A., Xu, J., and Zuck, L. 2002. Liveness with (0,1,infinity)-counter abstraction. In *Intl. Conf. on Computer Aided Verification (CAV)*.

RailShuttle_System. New rail-technology Paderborn. `http://nbp-www.upb.de/en/`.

Rational Rose. IBM Rational Rose. website: `http://www-306.ibm.com/software/rational/`.

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

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

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

Stevens, P. 2002. On the interpretation of binary associations in the Unified Modeling Language. *Journal on Software and Systems Modeling 1,* 1, 68–79.

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

Wehrheim, H. 2003. Behavioral subtyping relations for active objects. *Formal Methods in Sys. Design*.

APPENDIX

A. PROOF OF THEOREM 1

PROOF. The proof is by induction on N, the length of the execution sequence σ. It will be convenient to strengthen the induction hypothesis by assuming the following two properties to hold inductively as well:

—**Property (1)** Let n be the number of objects whose local states are given by the behavioral partition $beh \in BEH_p$ after the concrete execution of σ.[3] Then, after the abstract execution of σ, the object count for behavioral partition beh is also n.

—**Property (2)** After the concrete execution of σ, let there be n tuples of the form $\langle o_1, \ldots, o_k \rangle$ following association *asc*, where $o_1, \ldots, o_k$ are concrete objects, and the states of $o_1, \ldots, o_k$ are defined by the behavioral partitions $beh_1, \ldots, beh_k$. Then, after the abstract execution of σ, the association information maintained is such that the *asc* count for the k-tuple $\langle beh_1, \ldots, beh_k \rangle$ is also n.

The result is obvious if $N = 0$. Hence assume that $N > 0$ so that $\sigma = \sigma^{prev} \circ \gamma$ and the induction hypothesis holds for σ^{prev}. Let $o_1, \ldots, o_m$ be the concrete objects used to play the roles $r_1, \ldots, r_m$ in the concrete execution of transaction γ. If the states of $o_1, \ldots, o_m$ are given by behavioral partitions $beh_1, \ldots, beh_m$ ($beh'_1, \ldots, beh'_m$) in the concrete execution before (after) execution of γ, the following holds:

—$beh_1, \ldots, beh_m$ can serve as witness partitions of lifelines $r_1, \ldots, r_m$ of transaction γ under the abstract semantics after the execution of σ^{prev}. This follows from properties (1) *and* (2) above in the induction hypothesis.

—$beh'_1, \ldots, beh'_m$ are the destination partitions of $beh_1, \ldots, beh_m$ in the abstract execution of γ. This follows from the definition of destination partition (Definition 7).

Now, in the concrete execution of γ, corresponding to each participating object o_i, whose state changes from beh_i to beh'_i, the count of objects whose state is given by the behavioral partition beh_i (beh'_i) is decremented (incremented) by 1. Note that it is possible that there are more than one objects participating in γ whose state is given by the same behavioral partition beh (beh') before (after) execution of γ. Suppose, there are n_1 (n_2) concrete objects (participating in γ) whose state is given by the behavioral partition beh (beh') before (after) execution of γ. Then, after the execution of γ, the count of beh (beh') will be decremented (incremented) by $n_1(n_2)$.

Similarly, in the abstract execution of γ, corresponding to each role r_i in transaction γ, the object count of the witness (destination) partition, beh_i (beh'_i) will be decremented (incremented) by 1.Again, it is possible for a behavioral partition beh (beh') to be the witness (destination) partition for more than one roles in γ. Suppose, there are $n_1(n_2)$ roles having same witness (destination) partition, beh (beh'). Then, after the execution of γ, the count of beh (beh') will be decremented (incremented) by $n_1(n_2)$. Thus, we can conclude that Property (1) of the induction

[3]Recall that whether an object resides in beh is determined by its control state, history and valuation at the end of executing σ

hypothesis therefore holds after the execution of $\sigma = \sigma^{prev} \circ \gamma$.

To show that property (2) also holds, we consider the four cases according to the ways in which a k-ary association asc may be altered via the concrete execution of γ. Let $r_1, \ldots, r_m$ be the roles of transaction γ.

—Case A: We have a guard $(r_{i_1}, \ldots, r_{i_k}) \in asc$ as part of γ. Suppose that there are n_1 and n_2 number of k-tuples of concrete objects in association asc, whose local states are given by the behavioral partitions $beh_1, \ldots, beh_k$ and $beh'_1, \ldots, beh'_k$ respectively, before concrete execution of γ. Now, suppose a k-tuple of concrete objects $\langle o_1, \ldots, o_k \rangle$ in association asc is chosen for concrete execution of γ, s.t. the local states of objects $o_1, \ldots, o_k$ are given by the behavioral partitions $beh_1, \ldots, beh_k$ ($beh'_1, \ldots, beh'_k$) before (after) execution of γ. Thus, after the concrete execution of γ, the count of k-tuples in asc whose objects' states are given by $beh_1, \ldots, beh_k$ ($beh'_1, \ldots, beh'_k$) is decremented (incremented) by 1, resulting in association counts of '$n_1 - 1$' ($n_2 + 1$).
By the induction hypothesis, the asc count for the k-tuple $\langle beh_1, \ldots, beh_k \rangle$ ($\langle beh'_1, \ldots, beh'_k \rangle$) in the abstract execution is $n_1(n_2)$ after σ^{prev}. Further, we can choose the behavioral partitions $beh_1, \ldots, beh_k$ as the witness partitions for roles $r_{i_1}, \ldots, r_{i_k}$ to execute γ. Then, from Property (1), the corresponding destination partitions after executing γ are $beh'_1, \ldots, beh'_k$. Thus, after the abstract execution of γ, the asc count for the k-tuple $\langle beh_1, \ldots, beh_k \rangle$ ($\langle beh'_1, \ldots, beh'_k \rangle$) is updated to '$n_1 - 1$' ($n_2 + 1$) (see the case for "Check" in the handling of associations, Section 7).

—Case B: We have *insert* $(r_{i_1}, \ldots, r_{i_k})$ *into* asc as the post-condition of γ. Suppose $o_1, \ldots, o_k$ are the objects chosen to play the roles $r_{i_1}, \ldots, r_{i_k}$ in the concrete execution of γ and their local states are given by the behavioral partitions $beh_1, \ldots, beh_k$ ($beh'_1, \ldots, beh'_k$) before (after) execution of γ. Then a new k-tuple $\langle o_1, \ldots, o_k \rangle$ will be inserted in asc in concrete execution, thereby incrementing the count of k-tuples in asc, whose objects' states are given by $beh'_1, \ldots, beh'_k$, by 1. Due to induction hypothesis, in abstract execution we can choose the behavioral partitions $beh_1, \ldots, beh_k$ as the witness partitions for the roles $r_{i_1}, \ldots, r_{i_k}$ for executing γ. This means that $beh'_1, \ldots, beh'_k$ will be the destination partitions of roles $r_{i_1}, \ldots, r_{i_k}$ in the abstract execution; the asc count of the corresponding k-tuple is incremented by 1 in the abstract execution as well.

—Case C: We have *delete* $(r_{i_1}, \ldots, r_{i_k})$ *from* asc as the post-condition of γ. Suppose $o_1, \ldots, o_k$ are the objects chosen to play the roles $r_{i_1}, \ldots, r_{i_k}$ in the concrete execution of γ and their local states are given by the behavioral partitions $beh_1, \ldots, beh_k$ before execution of γ. Then the k-tuple $\langle o_1, \ldots, o_k \rangle$ is in asc before executing γ and is removed from asc after executing γ, thus decrementing the count of k-tuples in asc, whose objects' states are given by $beh_1, \ldots, beh_k$, by 1. Again, in abstract execution we choose the behavioral partitions $beh_1, \ldots, beh_k$ as the witness partitions for the roles $r_{i_1}, \ldots, r_{i_k}$ for executing γ, and decrement the count of k-tuple $\langle beh_1, \ldots, beh_k \rangle$ by 1 after executing γ.

—Case D: This is the *default* case where no association related guard or post-condition is specified for the roles $(r_{i_1}, \ldots, r_{i_k})$ of γ. Suppose $o_1, \ldots, o_k$ are the objects chosen to play the roles $r_{i_1}, \ldots, r_{i_k}$ in the concrete execution of γ and their

local states are given by the behavioral partitions $beh_1, \ldots, beh_k$ ($beh'_1, \ldots, beh'_k$) before (after) execution of γ. Then, if the *asc* count for the k-tuple $\langle o_1, \ldots, o_k \rangle$ is greater than zero before the concrete execution of γ, the handling of association counts in concrete and abstract executions will follow the 'Case A' above. Otherwise, there will be no update in *asc* association counts corresponding to the k-tuple of objects and behavioral-partitions for the roles $(r_{i_1}, \ldots, r_{i_k})$ in the concrete and abstract executions of γ.

This concludes the induction step for Property (2).

□

B. CHECKING SPURIOUSNESS OF EXECUTION RUNS IN MURPHI

Here we elaborate how we can check whether an execution run produced by our abstract simulator is spurious (*i.e.* cannot be realized in concrete executions). We have implemented the spuriousness check using the Murphi model checker [Murphi 2005]. The reason for using Murphi is its inherent support for symmetry reduction via the *scalarset* data type. We now discuss how Murphi's support of symmetry reduction is exploited to perform our spuriousness check efficiently.

We define the following data types for each process class.

—A *scalarset* type to act as an object identifier having the cut-off number[4] 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_iDi};    –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*[5] for the Car is declared as follows:

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

[4]The number $x_{p,\sigma}$ for process class p and execution run σ, defined in Section 8.3

[5]A car's *mode* indicates whether the car will stop or pass through its current terminal.

Associations are represented using two dimensional arrays having the value range 0..1. For an association "Asc" between two process classes A and B, assuming that A and B have been declared as appropriate scalarset types, this array is declared as follows:

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

An array entry of 1 will indicate the 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-occurrence* in the trace σ 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 abstract execution. The initial configuration for the Murphi execution is given as the "Startstate" declaration, where the initial control states, dfa states and variable valuations for various objects are defined. If an execution run σ produced by our abstract simulation is suspected to be spurious by the user, (s)he can submit it to Murphi for spuriousness check. Given our encoding of the reduced concrete IPC model to Murphi, if σ is indeed spurious it will correspond to a deadlocked run in Murphi, with Murphi getting stuck at a spurious transaction. By slight modification in the Murphi code we are able to precisely identify the transaction that σ got stuck at and furthermore report the system state at that point. This can then be analyzed by the user to determine the cause of spuriousness.

For illustration, in Appendix B.1 we give complete Murphi code listing corresponding to the example discussed in Section 8.3, for which we found the trace $\sigma = t_1.t_2.t_3$ to be spurious. The declarations corresponding to the $Cruiser$, Car and $BrakeControl$ process classes and static associations between them appear at the beginning (lines 1–21). Note that the *scalarset* types representing the object identifiers for process classes $Cruiser$, Car and $BrakeControl$ have size one, two and one respectively (lines 3–5). As discussed in Section 8.3, these are the cut-off number of objects for these classes to exhibit σ. An auxiliary variable 'Count' (line 27) is used to enforce the ordering, such that the rules corresponding to various transactions can only be triggered in the sequence in which they appear in σ. Murphi rules corresponding to transactions t_1, t_2 and t_3 appear at lines 87–103, 105–120 and 122–141 respectively. The guard of rule for t_1 (91–94) checks that– a) t_1 is the first transaction to execute (Count=1), b) participating $Cruiser$ and Car components are in the appropriate control states, and c) they satisfy the static association Asc_1. The body of the rule for t_1 (97–100) updates the control states of participating objects and the Count variable. Code corresponding to transactions t_2 and t_3 is similar to that of t_1. The successful completion of the trace being checked is indicated by an 'Error' state. It is reached in the last rule which is executed only after the trace has been completely executed (lines 143–149). The "Startstate" declaration, which sets the initial configuration for checking σ, appears at the end (lines 151–206).

The output of the verifier generated from the Murphi code for this example is presented in Appendix B.2. The trace σ can not be completely executed. This is indicated in the 'Status' message 'No error found' (lines 51–53); while, there should

be a 'Trace Complete' error message if the trace successfully executes, as discussed above. Further, σ is executed only upto t_2, after which Murphi gets deadlocked. This is indicated in the output at lines 33–47, which also give system state at the point of deadlock (35–47). The value of variable 'Count' is 3 (line 47), indicating that transaction t_3 should have executed next. Remaining variables give the control states for various objects and the associations among them. By examining this state and the guard of transaction t_3 (Appendix B.1, lines 126–131), we find that *Cruiser* object 'Cruiser1_1', *Car* objects 'Car1_1' and 'Car1_2', and *BrakeControl* object 'BrakeControl1_1' are in appropriate control states for executing t_3. However, none of the *Car* objects is associated both with a *Cruiser* object (via association Asc_1) and with a *BrakeControl* object (via association Asc_2). Therefore, t_3 can not be executed.

B.1 Murphi Code for example demonstrating spuriousness in Section 8.3

```
1   Type
2       -- index for process classes Cruiser1, Car1 and BrakeControl1 --
3       Cruiser1: Scalarset(1);
4       Car1: Scalarset(2);
5       BrakeControl1: Scalarset(1);
6
7       -- enumeration types for LTS states for the three process classes --
8       stCruiser: Enum {st_Cruiser1,st_Cruiser2,st_Cruiser3};
9       stCar: Enum {st_Car1,st_Car2,st_Car3};
10      stBrakeControl: Enum {st_BrakeControl1,st_BrakeControl2,st_BrakeControl3};
11
12  Var
13      -- static associations Asc1 (Cruiser1, Car1) and Asc2 (Car1 and BrakeControl1)
14      s_Asc111:
15      Array [Cruiser1] of Array [Car1] of 0..1;
16      s_Asc211: Array [Car1] of Array [BrakeControl1] of 0..1;
17
18      -- variables representing current control state for objects of each process class --
19      Cruiser1_lts: Array [Cruiser1] of stCruiser;
20      Car1_lts: Array [Car1] of stCar;
21      BrakeControl1_lts: Array [BrakeControl1] of stBrakeControl;
22
23      -- auxiliary variables --
24      Cruiser1_mark: Array [Cruiser1] of boolean;
25      Car1_mark: Array [Car1] of boolean;
26      BrakeControl1_mark: Array [BrakeControl1] of boolean;
27      Count: 0..4;
28
29  -- function to check for static association Asc1 at runtime --
30  function s_Asc111_check (i1:Cruiser1; i2:Car1): boolean;
31  var
32      count1: 0..1;
33      count2: 0..1;
34  begin
35      if (s_Asc111[i1][i2] = 1) then return true; endif;
36      count1 := 0;
37      count2 := 0;
38
39      for j:Car1 Do
40          if (s_Asc111[i1][j] = 1)
41          then count1 := count1 + 1;
42          end;
43      end;
44
45      if (count1 = 1) then return false end;
46
47      for j:Cruiser1 Do
48          if (s_Asc111[j][i2] = 1)
49          then count2 := count2 + 1;
```

```
50          end;
51      end;
52
53      if (count2 = 1) then return false end;
54
55      return true;
56  end;
57
58  -- function to check for static association Asc2 at runtime --
59  function s_Asc211_check (i1:Car1; i2:BrakeControl1): boolean;
60  var
61      count1: 0..1;
62      count2: 0..1;
63  begin
64      if (s_Asc211[i1][i2] = 1) then return true; endif;
65      count1 := 0;
66      count2 := 0;
67
68      for j:BrakeControl1 Do
69          if (s_Asc211[i1][j] = 1)
70          then count1 := count1 + 1;
71          end;
72      end;
73
74      if (count1 = 1) then return false end;
75
76      for j:Car1 Do
77          if (s_Asc211[j][i2] = 1)
78          then count2 := count2 + 1;
79          end;
80      end;
81
82      if (count2 = 1) then return false end;
83
84      return true;
85  end;
86
87  -- rule corresponding to transaction t1 --
88  Ruleset i1:Cruiser1; i2:Car1 Do
89      Rule "Execute T1"
90                                              -- rule guard
91          (Count = 1                          -- checking sequence
92           & Cruiser1_lts[i1] = st_Cruiser3   -- checking source control state
93           & Car1_lts[i2] = st_Car3           -- checking source control state
94           & s_Asc111_check(i1,i2)            -- checking association Asc1
95          )
96      ==>                                     -- rule body
97          s_Asc111[i1][i2] := 1;
98          Cruiser1_lts[i1] := st_Cruiser2;    -- updating control state
99          Car1_lts[i2] := st_Car2;            -- updating control state
100         Count := Count + 1;
101
102     End;  --Rule
103 End;  --Ruleset
104
105 -- rule corresponding to transaction t2 --
106 Ruleset i1:BrakeControl1; i2:Car1 Do
107     Rule "Execute T2"
108                                                     -- rule guard
109         (Count = 2
110          & BrakeControl1_lts[i1] = st_BrakeControl3     -- checking source control state
111          & Car1_lts[i2] = st_Car3                       -- checking source control state
112          & s_Asc211_check(i2,i1)                        -- checking association Asc2
113         )
114     ==>                                             -- rule body
115         s_Asc211[i2][i1] := 1;
116         BrakeControl1_lts[i1] := st_BrakeControl2;      -- updating control state
117         Car1_lts[i2] := st_Car2;                        -- updating control state
118         Count := Count + 1;
```

```
119     End;  --Rule
120 End;  --Ruleset
121
122 -- rule corresponding to transaction t3 --
123 Ruleset i1:Cruiser1; i2:BrakeControl1; i3:Car1 Do
124     Rule "Execute T3"
125                                                     -- rule guard
126         (Count = 3
127          & Cruiser1_lts[i1] = st_Cruiser2           -- checking source control state
128          & BrakeControl1_lts[i2] = st_BrakeControl2 -- checking source control state
129          & Car1_lts[i3] = st_Car2                   -- checking source control state
130          & s_Asc111_check(i1,i3)                    -- checking association Asc1
131          & s_Asc211_check(i3,i2)                    -- checking association Asc2
132         )
133     ==>                                             -- rule body
134         s_Asc111[i1][i3] := 1;
135         s_Asc211[i3][i2] := 1;
136         Cruiser1_lts[i1] := st_Cruiser1;            -- updating control state
137         BrakeControl1_lts[i2] := st_BrakeControl1;  -- updating control state
138         Car1_lts[i3] := st_Car1;                    -- updating control state
139         Count := Count + 1;
140     End;  --Rule
141 End;  --Ruleset
142
143 -- this rule executes after trace being checked is executed successfully --
144 Rule "Trace Completely Executed"
145
146     Count = 4
147     ==>
148     Error "Trace Complete."
149 End;
150
151 -- startstate declaration for setting the initial configuration --
152 Startstate
153 var
154     count: 0..2;
155 begin
156     For i1:Cruiser1 Do
157         For i2:Car1 Do s_Asc111[i1][i2] := 0; End;
158     End;
159     For i1:Car1 Do
160         For i2:BrakeControl1 Do s_Asc211[i1][i2] := 0; End;
161     End;
162
163     For i:Cruiser1 Do
164         Cruiser1_lts[i] := st_Cruiser3;
165         Cruiser1_mark[i] := false;
166     End;  --For
167
168     count := 0;
169     For i:Cruiser1 Do
170         if (Cruiser1_mark[i] = false & count < 1)
171         then
172             count := count + 1;
173             Cruiser1_mark[i] := true;
174         End;  --If
175     End;  --For
176
177     For i:BrakeControl1 Do
178         BrakeControl1_lts[i] := st_BrakeControl3;
179         BrakeControl1_mark[i] := false;
180     End;  --For
181
182     count := 0;
183     For i:BrakeControl1 Do
184         if (BrakeControl1_mark[i] = false & count < 1)
185         then
186             count := count + 1;
187             BrakeControl1_mark[i] := true;
```

```
188         End;  --If
189     End;  --For
190
191     For i:Car1 Do
192         Car1_lts[i] := st_Car3;
193         Car1_mark[i] := false;
194     End;  --For
195
196     count := 0;
197     For i:Car1 Do
198         if (Car1_mark[i] = false & count < 2)
199         then
200             count := count + 1;
201             Car1_mark[i] := true;
202         End;  --If
203     End;  --For
204
205     Count := 1;
206 End;  --End Startstate
```

B.2 Murphi-verifier output for example in Section 8.3

```
1   ============================================================================
2   Murphi Release 3.1
3   Finite-state Concurrent System Verifier.
4
5   Copyright (C) 1992 - 1999 by the Board of Trustees of
6   Leland Stanford Junior University.
7
8   ============================================================================
9
10  Protocol: spur
11
12  Algorithm:
13          Verification by breadth first search.
14          with symmetry algorithm 3 -- Heuristic Small Memory Normalization
15          with permutation trial limit 10.
16
17  Memory usage:
18
19          * The size of each state is 104 bits (rounded up to 16 bytes).
20          * The memory allocated for the hash table and state queue is
21            8 Mbytes.
22            With two words of overhead per state, the maximum size of
23            the state space is 392159 states.
24             * Use option "-k" or "-m" to increase this, if necessary.
25          * Capacity in queue for breadth-first search: 39215 states.
26             * Change the constant gPercentActiveStates in mu_prolog.inc
27               to increase this, if necessary.
28
29  Warning: No trace will not be printed in the case of protocol errors!
30           Check the options if you want to have error traces.
31
32
33  ** Deadlocked state found **
34
35  s_Asc111[Cruiser1_1][Car1_1]:0
36  s_Asc111[Cruiser1_1][Car1_2]:1
37  s_Asc211[Car1_1][BrakeControl1_1]:1
38  s_Asc211[Car1_2][BrakeControl1_1]:0
39  Cruiser1_lts[Cruiser1_1]:st_Cruiser2
40  Cruiser1_mark[Cruiser1_1]:true
41  Car1_lts[Car1_1]:st_Car2
42  Car1_lts[Car1_2]:st_Car2
43  Car1_mark[Car1_1]:true
44  Car1_mark[Car1_2]:true
45  BrakeControl1_lts[BrakeControl1_1]:st_BrakeControl2
46  BrakeControl1_mark[BrakeControl1_1]:true
47  Count:3
48
```

```
49 ================================================================================
50
51 Status:
52
53         No error found.
54
55 State Space Explored:
56
57         3 states, 3 rules fired in 0.10s.
58
59 Analysis of State Space:
60
61         There are rules that are never fired.
62         If you are running with symmetry, this may be why.  Otherwise,
63         please run this program with "-pr" for the rules information.
```