Interacting Process Classes

Ankit Goel Sun Meng Abhik Roychoudhury P. S. Thiagarajan
{ankitgoe,sunm,abhik,thiagu}@comp.nus.edu.sg
School of Computing, National University of Singapore

ABSTRACT

Many reactive control systems consist of classes of interacting objects where the objects belonging to a class exhibit similar behaviors. Such interacting process classes appear in telecommunication, transportation and avionics domains. In this paper, we propose a modeling and simulation technique for interacting process classes. Our modeling style uses standard notations to capture behavior. In particular, the control flow of a process class is captured by a labeled transition system, unit interactions between process objects are described by Message Sequence Charts and the structural relations are captured via class diagrams. The key feature of our approach is that our execution semantics leads to a *symbolic* simulation technique. Our simulation strategy is both time and memory efficient and we demonstrate this on well-studied non-trivial examples of reactive systems.

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

Keywords

Symbolic Execution, Active Objects, Message Sequence Charts, Unified Modeling Language (UML)

1. INTRODUCTION

Model-driven design methods based on the Unified Modeling Language (UML) are being advocated to push up the abstraction level in the design of reactive systems [11, 14]. Such design methods are necessary for supporting component reuse, early detection of errors and the co-design of hardware and software components. A crucial ingredient for the viability of such model-driven design methods is a simulation technique using which initial functional validation can be carried out. Here we propose a modeling language based on UML-compatible notations to describe interacting classes of active objects. The execution semantics of our language leads to a scalable simulation technique.

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

We use labeled transition systems to describe the behavior of process classes. One notable feature of our model is that the unit of interaction is chosen to be not just a synchronization or send-receive event pairs. Instead, we use a Message Sequence Chart (MSC) as the basic interaction unit. This is guided by the observation that even primitive interactions between process classes often involve bidirectional information flow. Further, the *roles* played by the participants is a key facet of an interaction and MSCs are ideally suited to depict this information.

We also specify static and dynamic associations between objects. We use class diagrams in a standard way to specify such associations. Static associations are needed to specify constraints imposed by the structure of the system. For instance, a node may be able to take part in a "transmit" transaction only with its neighbors. *Dynamic associations* on the other hand are needed to instantiate the proper combinations of objects to take part in a transaction. For instance, when choosing a pair of phone objects to take part in a "disconnect" transaction we must 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 a symbolic execution semantics is a challenging task. We present a solution to this problem and describe a simulator based on our so-

ICSE'06, May 20–28, 2006, Shanghai, China.

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

Outline. We develop our material in three steps. First we present the core modeling language and its execution semantics without involving associations (static/dynamic). We then introduce object associations and correspondingly extend the semantics. Finally, we present experimental results demonstrating capabilities of the simulator for our model.

2. RELATED WORK

Simulation of scenario-based specifications as well as synthesis of executable models from such specifications is an important research area. The synthesis task may consist of realizing per-process transition systems from scenario-based specifications (see [5, 23] for example). Alternatively, one may develop executable specifications based on Message Sequence Charts (MSCs). Works in this direction include Live Sequence Charts [2], Triggered Message Sequence Charts [20] as well as our past work [18]. All these approaches deal with *concrete* objects and their interactions.

Live Sequence Charts (LSCs)[2, 7] offer an MSC-based inter-object modeling framework for reactive systems. However, LSCs completely suppress the control flow information for each process class. More importantly, though the objects of a process class can be *specified* symbolically, the LSC execution mechanism (the play-engine as described in [7]) does not support symbolic execution of process classes. The symbolic instances are instantiated to concrete objects during simulation.[1] The work on Triggered Message Sequence Charts [20] allows for a per-process execution semantics (in comparison to the play-engine of LSCs which gives a centralized execution semantics). Again, the execution semantics deals with concrete object interactions.

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

Our technique for grouping together behaviorally similar objects is different from existing works on behavioral subtyping which develop subclass relationships based on behaviors of the objects in those classes. One of the early works in this area is by Liskov and Wing [13] which focuses on passive objects – objects whose state change is only via method invocation by other objects. Subsequently, behavioral subtyping of active objects have been studied in many works (*e.g.* [6, 25]). These works mostly exploit well-known notions of behavioral inclusion (such as trace containment) to define notions of behavioral subtyping. Our aim however is not to detect/establish subclass relationships. Rather, we wish to dynamically group together objects of the same class based on behavior exhibited so far for purposes of efficient system execution or simulation.

Our method of grouping together active objects is related to abstraction schemes developed for grouping processes in parameterized systems (*e.g.*, see [17]). In such systems, there are usually many similar processes whose behavior can be captured by a single finite state machine. It is then customary to maintain the count of number of processes in each state of the finite state machine; the names/identities of the individual processes are not maintained. However, in our setting inter-object associations across classes have to be maintained — an issue that does not arise in parameterized system validation.

The notion of "roles" played by processes in protocols have appeared in other contexts (*e.g.* [19]). Object orientation based on the actor-paradigm has been studied thoroughly in [12]. We see this work as an orthogonal approach where the computational rather than the control flow features are encapsulated using classes and other object-oriented programming notions (such as inheritance).

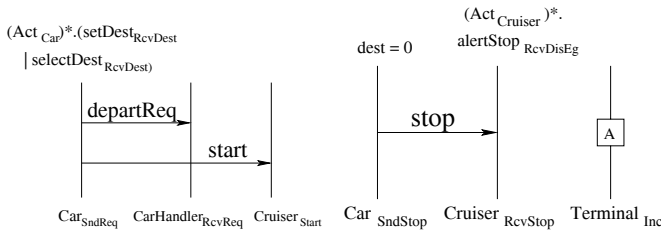
3. THE MODELING LANGUAGE

We model a reactive system as 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.

For each class, a labeled transition system will capture the common sequences of the computational and communication actions that the objects belonging to the class can go through. A communication action will name a transaction and the role played by an object of the class in the transaction. Message Sequence Charts (sequence diagrams) will be used to represent transactions. For our purposes, it will be convenient to view a chart as a labeled poset of the form $Ch = (R, \{E_r\}_{r \in R}, \leq, \lambda)$ where R is a set of **roles** (usually called lifelines or instances of the chart), and E_r is the set of events that the role r takes part in during the execution of Ch. The labeling function λ -with a suitable range of labels- describes (a) the messages exchanged by the instances and (b) the internal computational steps during the execution of the chart Ch. Finally, $\leq$ is the partial ordering relation over the occurrences of the events in $\{E_r\}_{r \in R}$. The formal definition of a Message Sequence Chart (MSC) is the standard one and is given in [3]. As will become clear shortly, the name we assign to a role will reflect its functionality *and* the class of the object that can play this role. Within the MSCs, the communication of messages can be synchronous or asynchronous – this issue is orthogonal to our model. In the operational semantics of our model, we assume that the execution of each transaction is atomic.

We show an example Message Sequence Chart (MSC) in Figure 1(a). It is taken from our modeling of Harel and Gery's Rail-car example presented in [2, 4]. For the moment, let us ignore the regular expression at the top portion of this chart. In Figure 1(a), the roles are "Car_{SndReq}", "$CarHandler_{RcvReq}$" and "$Cruiser_{Start}$". This naming convention is intended to indicate that the chart role "SndReq" is played by an object drawn from the class "Car", the chart role "RcvReq" is played by an object belonging to the class "CarHandler", and so on.

[1] The approach taken in [24] alleviates this problem of LSCs by maintaining constraints on process identities but falls short of a fully symbolic execution.



(a) departReqA (b) noMoreDest

Figure 1: Transactions *departReqA* & *noMoreDest*; A is an internal computation event in *noMoreDest*

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 also fix a set of **transactions** Γ with γ ranging over Γ. A transaction $\gamma = (I : Ch)$ will consist of a *guard* I and a Message Sequence Chart $Ch = (R, \{E_r\}_{r \in R}, \leq, \lambda)$. For a transaction $\gamma = (I : Ch)$, the guard I will consist of a conjunction of guards, one for each *role* of Ch. Each role r in R will be a pair (p, ρ) where p is the name of a class -from which an object playing this role is to be drawn- and ρ is the chart role to be played by r ("sender", "receiver" etc.) in the interaction specified by Ch; as a notational shorthand we have written role (p, ρ) as p_ρ at the bottom of each lifeline in the transactions of Figure 1.

It will be convenient to assume that if (p_1, ρ_1) and (p_2, ρ_2) are two distinct members of R (*i.e.*, two distinct roles), 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.*

In a transaction, the **guard** associated with the role (p, ρ) will specify the conditions that must be satisfied by an object O_r belonging to the class p in order for it to be eligible to play the role $r = (p, \rho)$. These conditions will consist of two components: i) a *history* property of the execution sequence (of communication actions) that O_r has so far gone through ii) a *propositional formula* built from boolean predicates regarding the values of the (instantiated) variables owned by O_r. For instance, in the transaction "departReqA" (refer to Figure 1(a)), a Car object wishing to play the role (*Car, SndReq*) must have last played the role (*Car, RcvDest*) in the transaction *setDest* or in the transaction *selectDest*. This is captured by the regular expression guard

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

shown at the top of the *SndReq* lifeline in Figure 1(a). Thus, we will use regular expressions to specify the history component of a guard. Also, note that in the transaction "departReqA" (Figure 1(a)), the guard does not restrict the local variable valuation of participating objects in any way. On the other hand, in the transaction of Figure 1(b), the variable "dest" owned by the car-object intending to play the role (*Car,SndStop*) must satisfy "dest = 0". *Finally, for some lifeline if no guard is mentioned (e.g. $Cruiser_{Start}$ in Fig.1(a)) then the corresponding guard is always enabled.*

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

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

We first explain the nature of the components Act_p, V_p and v_{init_p}. The set of actions Act_p are the set of roles that the p-objects can play in the transactions in Γ. Accordingly, a member of Act_p will be a triple of the form (γ, p, ρ) with $\gamma \in \Gamma$, $\gamma = (I : Ch)$ and $r = (p, \rho) \in R$ where R is the set of roles of Ch. *Since role $r = (p, \rho)$, the action label (γ, p, ρ) will be abbreviated as γ_r; when p is clear from the context it can also be abbreviated as γ_ρ.* The computational steps performed by an object will be described with the help of the set of variables V_p associated with p. Each object O in p of course will have its own copy of the variables in V_p but for convenience of explanation we shall assume that all the objects of class p assign the same initial value to any variable $u \in V_p$. This initial assignment is captured by the function v_{init_p} while assuming appropriate value domains for the variables in V_p. Since a computational step can be viewed as a degenerate type of transaction having just one role in its chart, we will not distinguish between computational and communication steps in what follows. Returning to $TS_p = \langle S_p, Act_p, \rightarrow_p, init_p, V_p, v_{init_p} \rangle$, S_p is the set of local states, $init_p \in S_p$ is the initial state and $\rightarrow_p \subseteq S_p \times Act_p \times S_p$ is the transition relation. In summary, our model can be defined as follows.

Definition 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.

We show an example of an IPC in Figure 2. It is a fragment of our modeling of the Rail-car example [2, 4] consisting of a cyclic rail network with fixed number of terminals and several moving cars. Controlling the movement of the cars between the terminals requires a complex description. The classes shown in Figure 2 are Car, $Cruiser$, $Terminal$ and $CarHandler$. The Cruiser stands for the cruise control of a car (this can be captured as associations via Class Diagrams as discussed in Section 5), while the CarHandler manages interaction between an approaching/departing car and the corresponding terminal. The interested reader can access more details from the [3].

4. SYMBOLIC EXECUTION MECHANISM

In this section, we describe the execution semantics of our IPC model. At the *initial configuration*, for each class p, every p-object will be residing at the designated initial state of TS_p. The history of each such object will be the null string and for each variable associated with p, each object of p will initialize it to the same value. The system will move from the current configuration by executing an enabled transaction and as a result, move to a new configuration. The transaction $\gamma = (I : Ch)$ is *enabled* at the configuration c if we can assign to each role $r = (p, \rho)$ of Ch, a *distinct* object O_r belonging to p such that the following conditions

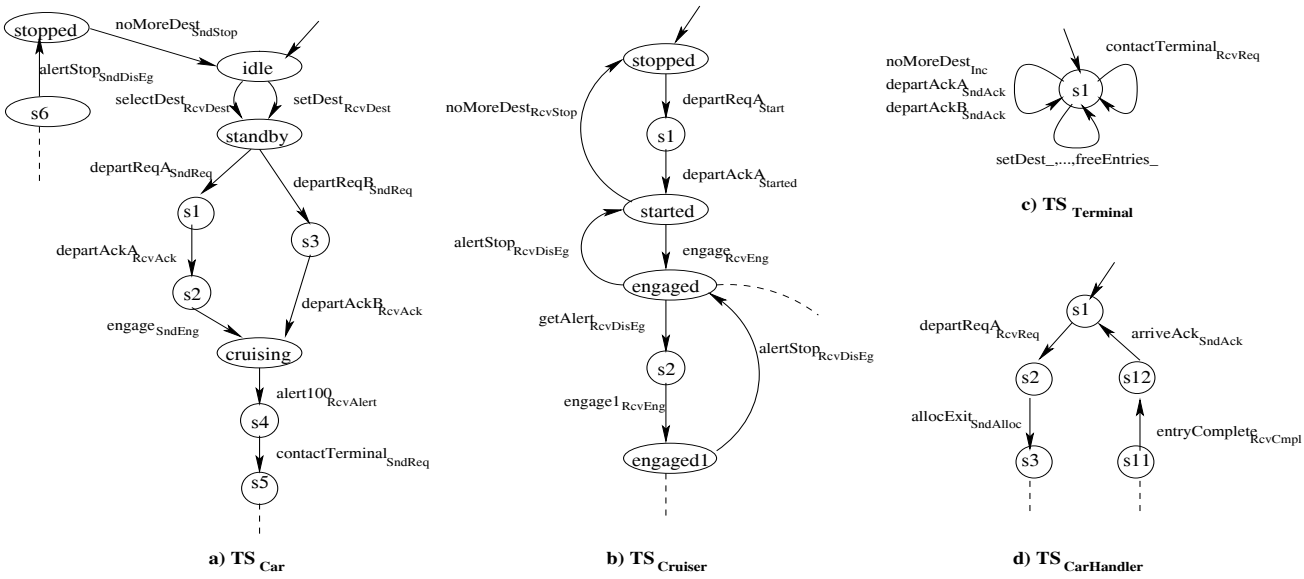


Figure 2: Fragment of Labeled Transition Systems for various process classes of the Rail-car example

are satisfied. We will state these conditions informally and illustrate them via the example shown in Figure 2.

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

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

- Two *Car* objects Oc_1 and Oc_2 are residing in state *stopped* and a third object, Oc_3, is in state $s2$ of TS_{Car}. Further suppose they have the values 0, 1 and 2 respectively for the variable *dest*.
- Three *Cruiser* objects, $O_1 \ldots O_3$ are residing in state *started* of $TS_{Cruiser}$ such that the history of O_1 and O_2 satisfy the regular expression

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

  while the history of O_3 satisfies the regular expression

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

- Six *Terminal* objects, $Ot_1 \ldots Ot_6$ are residing in state $s1$ of $TS_{Terminal}$.

Suppose we want to execute transaction *noMoreDest* — shown in Figure 1(b) — at configuration c. As for the role (*Car,SndStop*), though Oc_1 and Oc_2 are in the appropriate control state, only Oc_1 can be chosen since it (and not Oc_2) satisfies the guard $dest = 0$. For the cruisers, we observe that all the three *Cruiser* objects O_1, O_2, O_3 are in the "appropriate" control state at configuration c for the purpose of executing *noMoreDest*. However, only O_1 and O_2 have histories which satisfies the history part of the guard associated with the role (*Cruiser,RcvStop*). Hence either one of them (but not O_3) can be chosen to play this role. For the role (*Terminal,Inc*), both the history and propositional guards are vacuous and hence we can choose any one of the 6 objects residing in the control state $s1$.

Assume that Oc_1, O_1 and Ot_1 are chosen to execute transaction *noMoreDest* in configuration c. In the resulting configuration c', all objects other than Oc_1, O_1 and Ot_1 will have their control states and histories unchanged from c. The objects Oc_1, O_1, Ot_1 will reside in states *idle*, *stopped*, $s1$ respectively. The history of Oc_1, O_1, Ot_1 will be obtained by appending $noMoreDest_{SndStop}$, $noMoreDest_{RcvStop}$ and $noMoreDest_{Inc}$ to their respective histories at configuration c. Object Ot_1 also updates a local variable via an *internal* event — refer to $\text{Terminal}_{\text{Inc}}$ in Figure 1(b).

4.1 Behavioral Partitions

To achieve the goal of not maintaining the identities of the objects during execution, the objects of a class will be grouped together into *behavioral partitions*. We note that the ability of a p-object to participate in a transaction depends on its current state in TS_p, its execution history and valuation of its local variables. Given an IPC model as defined in the previous section, for the class p we define H_p to be the least set of DFAs given by: $\mathcal{A}$ is in H_p iff there exists a transaction $\gamma = (I : Ch)$ and a role r of Ch of the form (p, ρ) such that the guard I_r of r is (Λ, Ψ) and $\mathcal{A}$ is the minimal DFA recognizing the language defined by the regular expression Λ, the history part of the guard. The notion of behavioral partitions can now be defined as follows.

Definition 2 (Behavioral Partition). *Let* $\{TS_p = \langle S_p, Act_p, \rightarrow_p, init_p, V_p, v_{init_p}\rangle\}_{p\in\mathcal{P}}$ *be an IPC. Let* $H_p = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ *be the set of minimal DFAs defined for class* p *as described in the preceding. Then a* behavioral partition beh_p *of class* p *is a tuple* $(s, q_1, \ldots, q_k, v)$*, where*

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

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

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

- O_1 and O_2 are currently in the same state of TS_p,
- They have the same valuation of local variables, and
- Their current histories lead to the same state for all the DFAs in H_p.

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

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 $\mathcal{A} \in H_p$. As described in the preceding, H_p is the set of minimal DFAs accepting the regular expression guards of the various roles of different transactions played by class p. Note that the maximum number of behavioral partitions does not depend on the number of objects in a class. In practice, many regular expression guards of transactions are vacuous leading to a small number of partitions. For example, the Cruiser class of the Rail-Car Example shown in Figure 2(b) can have at most 14 behavioral partitions since — (i) $TS_{Cruiser}$ has seven (7) states (not all of them are shown in Figure 2(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 7 we report experiments that the number of behavioral partitions encountered in actual simulation runs is often lower than the upper bound on number of partitions (48 Cruiser objects are divided into less than 6 partitions, see Table 1).

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

Example. Consider $TS_{Cruiser}$ shown in Figure 2(b). Suppose we simulate the specification with 24 *Cruiser* objects (assume that other process-classes are also appropriately populated with objects). In $TS_{Cruiser}$, only the transition $noMoreDest_{RcvStop}$ 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:

"$departReqA_{Start}$, $departAckA_{Started}$, $engage_{RcvEng}$, $alertStop_{RcvDisEg}$"

O_1 will now resides in the control state *started*. Also, since $alertStop_{RcvDisEg}$ 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: "$departReqA_{Start}$, $departAckA_{Started}$". 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 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 $noMoreDest_{RcvStop}$. This will result in a merger of the first two behavioral partitions mentioned above. In other words, O_1 will be now indistinguishable from the 22 objects which have remained idle throughout. For all of these 23 objects, the action $departReqA_{Start}$ is now enabled. This is the manner in which behavioral partitions will be split and merged during simulation.

4.2 Simulation of Core Model

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

Definition 3 (Abstract Configuration). *Let* $\{TS_p\}_{p\in\mathcal{P}}$ *be an IPC specification such that each process class* p *contains* N_p *objects. An abstract configuration of the IPC is defined as follows.*

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

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

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

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

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

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

Our symbolic simulation efficiently keeps track of the objects in various process classes by maintaining the current abstract configuration; only the behavioral partitions with non-zero counts are kept track of. The system moves from one abstract configuration to another by executing a transaction. In what follows, for the sake of convenience we shall often drop the "abstract" when talking about "abstract configurations". How can our simulator check whether a specific transaction γ is enabled at the current configuration cfg? We say that γ is enabled at cfg if for every lifeline of γ we can assign a distinct object to take up that lifeline (*i.e.* we do not want the same object to act as several lifelines in the same execution of a transaction γ). Since we do not keep track of object identities, we define the notion of *witness partition* for a role, from which an object can be chosen.

DEFINITION 4 (WITNESS PARTITION). *Let $\gamma \in \Gamma$ be a transaction and $\mathsf{cfg} \in \mathcal{C}_\mathcal{S}$ be a configuration. For a role $r=(p,\rho)$ of γ where r has the guard (Λ,Ψ), we say that a behavioral partition $beh=(s,q_1,\ldots,q_k,v)$ is a witness partition, denoted as $witness(r,\gamma,\mathsf{cfg})$, for r at cfg if*

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

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

DEFINITION 5 (ENABLED TRANSACTION). *Let γ be a transaction and $\mathsf{cfg} \in \mathcal{C}_\mathcal{S}$ be a configuration. We say that γ is enabled at cfg iff for each role $r=(p,\rho)$ of γ , there exists a witness partition $witness(r,\gamma,\mathsf{cfg})$ such that*

- *If $beh \in BEH_p$ is assigned as witness partition of n roles in γ, then $count_p(b) \ge n$. 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 in transaction γ, where $r=(p,\rho)$ is a role in γ.

DEFINITION 6 (DESTINATION PARTITION). *Let γ be an enabled transaction at configuration $\mathsf{cfg} \in \mathcal{C}_\mathcal{S}$ and $beh=(s,q_1,\ldots,q_k,v)$ be the witness partition for the role $r=(p,\rho)$ of γ. Then we define $dest(beh,\gamma,r)$ — the destination partition of beh w.r.t. transaction γ and role r — as a behavioral partition $beh'=(s',q'_1,\ldots,q'_k,v')$, where*

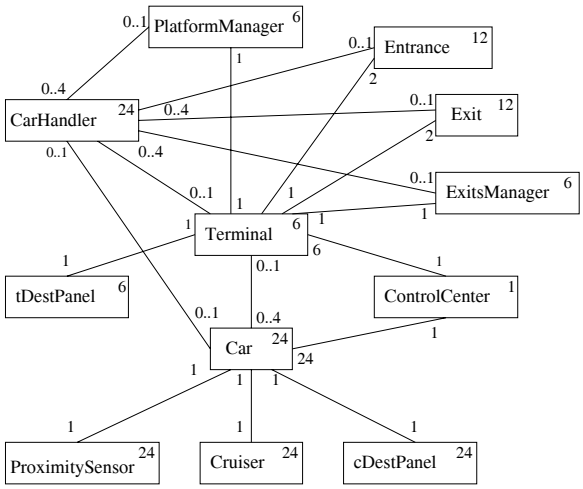


Figure 3: Class diagram for Rail-car example.

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

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

5. 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. Our notion of static and dynamic associations is similar to the classification presented in [22].

Static Associations. A static association expresses a *structural relationship* between the classes. In a class-diagram the static associations are captured using links, annotated with fixed multiplicities at both the association 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 3. For example, the following pairs of classes: (PlatformManager, Terminal), (Terminal, ControlCenter), (Car, ControlCenter) and (Car, Cruiser) are statically associated in Figure 3. In particular, the link between the Car class and the Cruiser class denotes the *itsCruiser* association between a car and its cruiser.

Dynamic Associations. A dynamic association expresses *behavioral relationship* between classes, which in our case

contactTerminal	**inserts**	(O1,O2)	into	**itsTerminal**
where, O1 plays the role	**(Car, SndReq)**, and			
O2 plays the role	**(Terminal, RcvReq)**			
departAck(A/B)	**checks**	(O1,O2)	belongs to	**itsTerminal**
where, O1 plays the role	**(Car, RcvAck)**, and			
O2 plays the role	**(Terminal, SndAck)**			
departAck(A/B)	**deletes**	(O1,O2)	from	**itsTerminal**
where, O1 plays the role	**(Car, RcvAck)**, and			
O2 plays the role	**(Terminal, SndAck)**			

Figure 4: Dynamic Relation *itsTerminal*

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, dynamic associations are captured using links, annotated with varying multiplicities.

We 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 individual local histories. Hence we make use of dynamic relation *itsTerminal* between the *Car* and *Terminal* classes as part of our specification.

Instead of giving details of the *contactTerminal* and *departAck(A/B)* transactions, we list here relevant roles of these transactions.

- *contactTerminal* has roles *(Car,SndReq)* and *(Terminal,RcvReq)*,
- *departAckA* and *departAckB* have roles *(Car,RcvAck)* and *(Terminal,SndAck)*. Note that transactions *departAck(A/B)* also involve other roles which we choose to ignore here for the purpose of our discussion.

If car object O_c and terminal object O_t play the roles (*Car,SndReq*) and (*Terminal,RcvReq*) in *contactTerminal*, then the effect of *contactTerminal* is to insert the pair (O_c, O_t) into the *itsTerminal* relation (refer to Figure 4). The *departAck(A/B)* transaction's guard now includes the check that the object corresponding to the role (*Car,RcvAck*) and object corresponding to lifeline (*Terminal,SndAck*) be related by the dynamic relation *itsTerminal*; so if objects O_c and O_t are selected to play the (*Car,RcvAck*) and (*Terminal,SndAck*) roles in *departAck(A/B)*, the check will succeed. Furthermore, the effect of *departAck(A/B)* transaction is to remove the tuple (O_c, O_t) from *itsTerminal* relation.

For a dynamic relation, we describe the effect of each transaction on the relation in terms of addition/deletion of tuples of objects into the relation. Furthermore, the guard of any transaction can contain a membership constraint on one or more of the specified dynamic relations. For simulation 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 symbolic execution semantics.

Simulating the extended model. For dynamic associations, the key question here is how we maintain relationships between objects if we do not keep track of the object identities. We do so by maintaining *dynamic associations between behavioral partitions.* To illustrate the idea, consider a binary relation D which is supposed to capture some dynamic association between objects of the process class p. In our symbolic execution, each element of D will be a pair (b, b') where b and b' are behavioral partitions of class p. To understand what $(b, b') \in D$ means, consider the concrete simulation of the process class p. If after an execution π (a sequence of transactions), two concrete objects O, O' of p get D-related ($(O, O') \in D$) then the symbolic execution along the same sequence of transactions π must produce $(b, b') \in D$ where b (b') is the behavioral partition in which O (O') resides after executing π. The same idea can be used to manage dynamic relations of larger arities. The handling of static associations is similar to that of dynamic associations. Again the guard of a transaction can refer to these associations; so we need to take these associations into account while assigning the witness behavioral partitions for each lifeline of a transaction. However when we check that the witness partitions of two roles in a transaction are statically associated, we also allow for the possibility that these partitions may consist of fresh objects which have not executed any transaction so far. Clearly, this possibility does not exist for dynamic associations.

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.

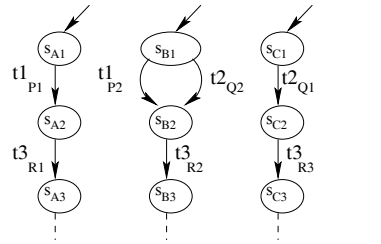
Example. As discussed earlier, the dynamic relation *itsTerminal* is maintained between the objects of class *Car* and *Terminal* (as shown in Figure 4). This relationship is established between a *Car* and a *Terminal* object while executing *contactTerminal* and exists till the related pair executes either *departAckA* or *departAckB*. For illustration, suppose one object each from class *Car* and class *Terminal* plays the role (*Car,SndReq*) and (*Terminal,RcvReq*) respectively in the transaction *contactTerminal*. Let b_{Car} (b_{Term}) be the behavioral partitions in to which the objects of *Car* (*Terminal*) go by executing $contactTerminal_{SndReq}$ ($contactTerminal_{RcvReq}$). So in our symbolic execution,

$$itsTerminal = \{(b_{Car}, b_{Term})\}.$$

Now when we execute *departAck(A/B)* transaction, we will pick a pair from this relation as witness behavioral partitions for the roles (*Car,RcvAck*) and (*Terminal,SndAck*). We *have not* maintained information about which *Terminal* object in b_{Term} is related to which *Car* of b_{Car}. In our symbolic simulation, 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.

6. HOW CONCRETE IS SYMBOLIC?

It turns out, that due to the presence of associations, there can be symbolic executions which do not correspond to concrete executions. However every concrete execution can be realized as a symbolic execution. In this sense, our symbolic execution semantics is sound as stated next. The proof is omitted and can be found in the technical report [3].



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

Figure 5: An example

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

It can also be shown that for IPC specifications not containing associations, corresponding to a symbolic execution run we can construct a concrete execution run. The proof follows from a straightforward induction on the length of symbolic execution run and can be found in [3].

To see that the converse of theorem 1 does not hold in presence of associations, 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 *its-Cruiser* (*its-BrakeController*). Fragments of the transition systems for these components are shown in Figure 5, 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 lifeline (of course the object should be in the appropriate control state). Suppose now, that we have an initial configuration

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

Each process class *Cruiser*, *Car* and *BrakeControl*, contains 2 objects in their initial states s_{A1}, s_{B1} and s_{C1} respectively.

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

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

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 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

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

In our symbolic simulation, 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 executed. But since the distinction between $Car1$ and $Car2$ is not made in symbolic simulation, transaction $t3$ (involving all the classes — Car, Cruiser, BrakeControl) will be executed in the symbolic simulation. In the concrete simulation 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 $t3$. Though in this example we have only considered static associations, similar incompleteness of our symbolic execution can be shown with dynamic associations.

Checking a symbolic run. Since symbolic execution runs may not correspond to concrete runs, we need a mechanism to detect spurious symbolic executions. This is similar to detecting spurious counter-example traces in abstraction-refinement based software model checking (e.g. see [8]). Fortunately, one can effectively check in our setting if a symbolic execution run σ corresponds to a concrete run as follows. For each process class p, let $num_{p,\sigma}$ be the total number of roles played by an object of class p from its initial state in the transactions appearing in σ. This number $num_{p,\sigma}$ can be calculated by counting the number of roles (p, ρ) of a transaction γ appearing in transaction sequence σ s.t. γ_ρ is an outgoing transition from the initial state of TS_p (the transition system for process class p). We define $x_{p,\sigma} = min(N_p, num_{p,\sigma})$ if N_p, the number of objects in p is a given constant. Otherwise the number of objects of p is not fixed and we set $x_{p,\sigma} = num_{p,\sigma}$. It is worth noting that $x_{p,\sigma}$ serves as a cutoff on the number of objects of class p only for the purpose of exhibiting the behavior σ and not all the behaviors of the system. Clearly, σ is a concrete run in the given system iff it is a concrete run in the *finite state* system where each process class p has $x_{p,\sigma}$ objects.

We have implemented the above spuriousness check using the Murphi model checker [15]. The reason for using Murphi is that it has in-built support for symmetry reduction [10]; this can speed up model checking of process classes with many similar processes. Such systems often exhibit structural symmetry which can be exploited to avoid constructing/traversing the full state space. Indeed 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 symbolic simulator was typically not spurious. In fact, there was only one false positive among all the test cases we tried for all the examples.

7. EXPERIMENTS

We have implemented our symbolic execution method by building a simulator in *OCaml* [16], a general purpose programming language supporting functional, imperative and object-oriented programming styles.

Modeled Examples. For our initial experiments, we modeled a simple telephone switch drawn from [9]. 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

Process Class	# Concrete Objects	# of partitions in Test Case		
		I	II	III
Car	48	12	10	11
CarHandler	48	3	8	8
Terminal	6	6	6	6
Platform Mngr.	6	1	3	3
Exits Mngr.	6	1	2	2
Entrance	12	2	1	2
Exit	12	1	2	2
Cruiser	48	1	3	5
Proximity Sensor	48	1	1	2
cDestPanel	48	1	1	1
tDestPanel	6	1	1	1

Table 1: Maximum Number of Behavioral partitions seen during symbolic simulation of RailCar example.

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

C ≡ Concrete Exec., S ≡ Symbolic Exec.

Table 2: Timing/Memory Overheads of Concrete Simulation and Symbolic Simulation

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 [4] and using Live Sequence Charts in [2]. This is an automated rail-car system with several cars operating on two parallel cyclic paths with several terminals. The cars run clockwise on one of the cyclic paths and anti-clockwise direction on the other. This example is a substantial sized system with a number of 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 [21] 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. Also, in case a shuttle is bankrupt (due to payment of fines), it is retired. The weather update controller [1] is a an important component of the *Center TRACON Automation System (CTAS)*, automation tools developed by NASA to manage high volume of arrival air traffic at large airports. The case study involves three classes of objects: weather-aware clients, weather control panel 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.

Behavioral Partitions. We used guided simulation on each of our examples to test out the prominent use cases. The details of these experiments appear in the technical report [3]; due to lack of space we only mention the results for the Rail-car example [2]. We simulate the following test cases for the Rail-car example– (a) cars moving from a busy terminal to another busy terminal (*i.e.* a terminal where all the platforms are occupied, so an incoming car has to wait), while stopping at every terminal, (b) cars moving from a busy terminal to less busy terminals while stopping at every terminal, and (c) cars moving from one terminal to another while not stopping at certain intermediate terminals. We summarize the results of three test cases in Table 1. For each test case, we report the number of concrete objects 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 concrete object. For each test case, we let the simulation run for 100 transactions – long enough to exhibit the test case's behavior. From Table 1, we can see that the number of behavioral partitions is substantially less than the number of concrete objects.

Timing and Memory Overheads. At the heart of our symbolic simulation is the idea of a behavioral partition, which groups together objects. 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 symbolic simulator. Simulations were run on a Pentium-IV 3 GHz machine with 1 GB of main memory. The results are shown in Table 2. For each example we show the time and memory usage for both the symbolic 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

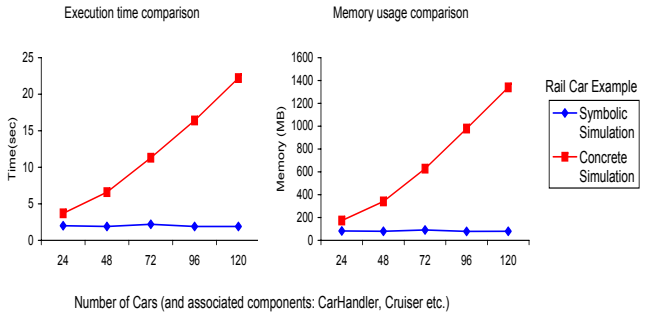


Figure 6: Execution Time and Memory Usage for different settings of the RailCar example.

and a given number of objects, the running time and memory usage for the concrete simulator are higher than that for the symbolic simulator. Also for the same example but with higher number of objects, in case of symbolic execution, the time/memory remain roughly the same, whereas they increase substantially for the concrete case (as indicated by the increase in ratio C/S for higher number of objects in Table 2). Further, in the graphs shown in Figure 6, we compare the growth in timing and memory usage in the railcar example, for both concrete and symbolic 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 symbolic simulation allows the designer to try out different settings of a model by varying the number of objects without worrying about time/space overheads.

Simulator Features. Currently, our simulator supports the following features to help error detection – (a) random simulation, (b) guided simulation for a use-case, and (c) testing whether a given sequence of transactions is an allowed behavior. The simulator source code as well as the modeling of all the examples reported in this paper can be obtained from `http://www.comp.nus.edu.sg/~ankitgoe/simulator`

8. DISCUSSION

In this paper, we have studied a modeling formalism accompanied by a simulation technique for dealing with interacting process classes; such systems arise in a number of application domains such as telecommunications and transportation. Our models are based on standard UML notations and our symbolic simulation strategy allows efficient simulation of realistic designs with large number of objects. The 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. Finally, we plan to develop a verification framework centered on our symbolic execution semantics that will exploit the abstraction-refinement based approach to software model checking.

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.

9. REFERENCES

[1] CTAS. Center TRACON automation system. `http://www.ctas.arc.nasa.gov`.
[2] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 2001.
[3] A. Goel et al. Interacting process classes. Technical report, NUS, 2005. TRA9/05, `http://www.comp.nus.edu.sg/~abhik/pdf/IPC-TR.pdf`.
[4] D. Harel and E. Gery. Executable object modeling with statecharts. *ICSE*, 1996.
[5] D. Harel, H. Kugler, and A. Pnueli. Synthesis revisited: Generating statechart models from scenarios-based requirements. In *Formal Methods in Software and System Modeling, LNCS 3393*, 2005.
[6] D. Harel and O. Kupferman. On object systems and behavioral inheritance. *IEEE Transactions on Software Engineering*, 2002.
[7] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
[8] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
[9] G.J. Holzmann. *Modeling a Simple Telephone Switch*, chapter 14. The SPIN Model Checker. Addison-Wesley, 2004.
[10] C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(2), 1996.
[11] L. Lavagno, G. Martin, and B. Selic. *UML for Real: Design of Embedded Real-time Systems*. Kluwer, 2003.
[12] E.A. Lee and S. Neuendorffer. Classes and subclasses in actor-oriented design. In *MEMOCODE, ACM Press*, 2004.
[13] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1994.
[14] S.J. Mellor and M.J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.
[15] Murphi. Murphi description language and verifier, 2005. `http://verify.stanford.edu/dill/murphi.html`.
[16] OCaml. The OCaml programming language, 2005. `http://caml.inria.fr/ocaml/index.en.html`.
[17] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In *Intl. Conf. on Computer Aided Verification (CAV)*, 2002.
[18] A. Roychoudhury and P.S. Thiagarajan. Communicating transaction processes. In *ACSD, IEEE Press*, 2003.
[19] B. Selic. Using UML for modeling complex real-time systems. In *LCTES, LNCS 1474*, 1998.
[20] B. Sengupta and R. Cleaveland. Triggered message sequence charts. In *FSE*, 2002.
[21] Shuttle_Control_System. New rail-technology Paderborn. `http://wwwcs.uni-paderborn.de/cs/ag-schaefer/CaseStudies/ShuttleSystem`.
[22] Perdita Stevens. On the interpretation of binary associations in the Unified Modeling Language. *Journal on Software and Systems Modeling*, 1(1):68–79, 2002.
[23] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29, 2003.
[24] T. Wang, A. Roychoudhury, R.H.C. Yap, and S.C. Choudhary. Symbolic execution of behavioral requirements. In *PADL, LNCS 3057*, 2004.
[25] H. Wehrheim. Behavioral subtyping relations for active objects. *Formal Methods in Sys. Design*, 2003.