Interacting Process Classes: Modeling and Simulation

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

Abstract

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

1. Introduction

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

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

We use state and class diagrams as a basis for our modeling framework – class diagrams are used to capture the associations between the process classes and state diagrams are used to describe the behavior of process classes. One unconventional feature in our modeling framework is that the unit of interaction is chosen to be not just a synchronization or send-receive event pairs. Instead, we use a sequence diagram as a basic communication unit. We note that at the model level, even primitive interactions between process classes often involve bidirectional information flow and are best depicted as short -acyclic- protocols. Thus from our experience in reactive system modeling (including the examples discussed in this paper), descriptions of process interactions based on sequence diagrams is very natural. Finally, we also introduce static and dynamic associations between objects. This is necessary when classes of active objects interact with each other. Static associations are needed to specify constraints imposed by the structure of the system. For instance, the topology of a network may demand that a node can take part in a "transmit" transaction only with its neighbors. We use class diagrams in a standard way to specify such associations. On the other hand, *dynamic associations* are needed to instantiate the proper combinations of objects -based on past history- to take part in a transaction. For instance when choosing a send-receive pair of objects to take part in a "disconnect"' transaction we must choose a pair which are currently in the "connected" relation. This relation has presumably arisen by virtue of the fact that they took part last in a "connect" transaction.

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

In summary, the highlights of our work are: (a) a symbolic execution semantics which dynamically groups objects of a process class based on behavior, (b) systematic

use of sequence diagrams to specify behavioral interactions between interacting process classes, and (c) investigating the efficiency of our symbolic simulation and its use in debugging with the help of realistic examples of reactive controllers. In terms of future work, we are looking into code generation as well as (symbolic) test-bench generation from our models.

Related Work Our previous work [9] provides a modeling language based on inter-object based description of interactions. However it does not consider process *classes*. The work on Live Sequence Charts [3, 5] presents an sequence diagram based inter-object modeling framework for reactive systems. However, this approach does not exploit symbolic execution of process classes. Though process classes are *specified* symbolically, they are instantiated to concrete objects during play-out or simulation. The approach taken in [12] alleviates this problem by maintaining constraints on process identities but falls short of a fully symbolic execution without process identities.

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

The notion of protocols and roles played by processes in protocols have appeared in other contexts (*e.g.* [10]). Object orientation based on the actor-paradigm has been studied thoroughly (*e.g.* see [8]). We see this as an orthogonal approach where the computational rather than the control flow -and communication-features are encapsulated using classes and fundamental OO features such as inheritance. It will be interesting and challenging to incorporate this approach into our framework. In a broader perspective, our work has similarities with frameworks such as Ptolemy [7] and Metropolis [1] where the communication and computational aspects are clearly delineated from each other.

2. The Modeling Language

We model a reactive system as a collection of process classes where a class is a collection of processes with similar functionalities. Objects belonging to a class will possess a common control flow detailing the pattern of computational and communication activities they can go through and this is described as a state diagram. A communication action will represent a snippet of a protocol, namely, the part played by an object of the class in an execution of the protocol. Each such action will have a label of the form γ_r where γ is the name of a transaction (*i.e.*, a sequence diagram together with a set of constraints forming the guard of the transaction), and r is the name of a particular *lifeline* in γ. Computation actions can be viewed as degenerate sequence diagrams with a single lifeline.

Thus our model consist of a network of interacting process classes where each process class p's behavior is captured by a state diagram TS_p. The set of action labels in TS_p is called Act_p with each label in Act_p being of the form γ_r as discussed above. This means that objects of p can participate in transaction γ by playing the role of the lifeline marked by r. Often, different objects of the same process class can play different lifelines in the same execution of a transaction. Hence an action label mentions the transaction name as well as the lifeline/role name.

Transactions as Sequence Diagrams A transaction in our model is a guarded sequence diagram. The diagram in figure 1(b) is an example of a transaction (called Tx) consisting of two life lines. We will discuss this example in details later. Our sequence diagrams will be based on send-receive type of communications. However other features such as synchronizations can be easily added. As usual, the partial order of events in a sequence diagram is the transitive closure of (a) the total order of the events in each process (time flows from top to bottom along each lifeline) and (b) the ordering imposed by the send-receive of each message (the send event of a message must happen before its receive event). Internal events can be used to denote update operations associated with the transaction but we will suppress them here for ease of exposition (see Section 6).

The Guard of a Transaction The guard of the transaction is a conjunction of the guards of its lifelines. The guard of a lifeline is evaluated on the finite history of transactions executed by the object currently auditioning for this "role". The language of these guards (which are evaluated on finite histories) is flexible. In this work, we use regular expressions on the alphabet of action labels as guards on past behavior. For the transaction shown in figure 1(b), no restriction is placed on the history of the object wishing to play the "snd" role. However, the object wishing to play the "rcv" must have a local history where the last transaction should not be Tx with a snd role. Note that we could have avoided the regular expression guards in our model by blowing up the state diagrams of the process classes. From our experience in modeling some of the examples discussed in this

paper, this leads to a substantial blow-up which can be easily avoided by putting in regular expression guards.

Example Consider a single process class *Node* denoting the behavior of nodes in a network. TS_{Node} is given in Figure 1(a) with $Act_{Node} = \{Tx_{snd}, Tx_{rcv}, Erase, Store\}$. The transactions $Erase$ and $Store$ are trivial unconstrained transactions having just one lifeline (basically denoting computational steps). The transaction Tx captures data exchange between two arbitrary nodes with *snd* and *rcv* denoting sender and receiver roles in the data exchange. As explained above, the *snd* role has the trivial guard $(Act_{Node})^\star$ (which accepts all histories), but the *rcv* role has a guard $\neg((Act_{Node})^* Tx_{snd})$. This restricts the last symbol in the history to be different from Tx_{snd}, thereby preventing a node from getting back the data it has just transmitted.

3. Symbolic Execution Mechanism

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

Suppose c is a configuration at which two objects o_1 and o_2 are both residing at the state $s1$ in the single process class example shown in figure 1(a). Suppose further that o_1 has as its current history $Tx_{snd}Erase$ and o_2 has as its current history $Tx_{rcv}Store$. Then the transaction Tx can be executed at c with o_2 playing the role Tx_{snd} and o_1 playing the role Tx_{rcv}. This is because the respective histories satisfy the respective guards and the control states these objects are currently residing in have the required outgoing transitions. As a result of executing Tx at c the system will move to the configuration c'. At c' all objects other than o_1 and o_2 will have their control state and histories unchanged from c. At c', the object o_2 will reside at $s1$ due to the transition labeled Tx_{snd} from $s1$ to $s1$. Its history at c' will be $Tx_{rcv}StoreTx_{snd}$. On the other hand, o_1 will reside at $s2$ in c' and its history will be $Tx_{snd}EraseTx_{rcv}$. As this example illustrates, a process class can contribute more than one object in the execution of transaction.

3.1. Behavioral Subclasses

One of our key objectives is to avoid having to keep track of the identities of the objects of a process class during execution. To achieve this , the objects of a process class will be grouped into "behavioral subclasses" based on their past interactions. There are a variety of ways of keeping track of histories including automata, formal languages and temporal logics. Here, we choose the language theoretic mechanism of regular expressions mainly because of its familiarity. In the examples we have studied, it suffices to deploy very restricted regular expressions. These restricted regular expressions can be represented as small DFAs and hence, from a pragmatic point of view, complexity is not an issue.

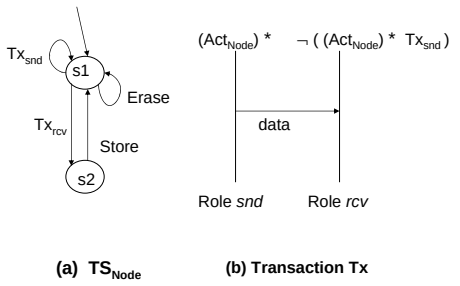

Figure 1: **(a) State diagram of a process class, and (b) Transaction** Tx

We say that two objects o_1 and o_2 of process class p are in the same *behavioral subclass* (at a configuration) if and only if both the following conditions hold.

1. o_1 and o_2 are currently in the same state of TS_p.
2. Let the past history of actions in TS_p executed by o_1 and o_2 be the strings σ_1 and σ_2, that is $\sigma_1 \in (Act_p)^*$ and $\sigma_2 \in (Act_p)^*$. Then, for any extension $\sigma \in (Act_p)^*$, the strings $\sigma_1 \circ \sigma$ and $\sigma_2 \circ \sigma$ are indistinguishable w.r.t. satisfaction of the history-based guard of any action label drawn from Act_p.

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

Bounding the number of Behavioral Subclasses Recall that each action label $a \in Act_p$ is a lifeline of a transaction and in the present setting has a guard in the form of a regular expression. Let $M(a)$ denote the number of states of the minimal deterministic automaton accepting the regular expression denoting the guard of a. The number of behavioral subclasses of process class p is then bounded by $|S_p| \times \prod_{a \in Act_p} M(a)$ where $|S_p|$ is the number of states of TS_p. In Figure 1, the DFA accepting the only non-trivial regular expression $\neg((Act_{Node})^* Tx_{snd})$ has 2 states (see

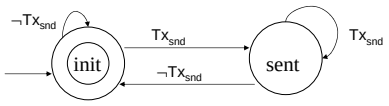

Figure 2: **Minimal DFA accepting guard of action Tx_{rcv} shown in Figure 1**

Figure 2) and TS_{Node} has 2 states. Any action label whose regular expression guard is not shown has a guard represented by a single state DFA (which accepts all histories). Therefore, the maximum number of behavioral subclasses is $2 * 2 = 4$, whereas the number of node objects being simulated might be rather large.

In what follows, the set of all behavioral subclasses of a process class p is denoted as B_p with a typical member denoted as $\langle s, History \rangle$ where s is a state in TS_p and $History$ is a function which maps each action label $a \in Act_p$ to a state in the minimal DFA accepting the regular expression guard of a.

Example Consider the example in Figure 1 discussed in the previous section. Suppose that we simulate the specification for 20 nodes. Initially all the node objects are in state $s1$ with null history. Thus they are in the same behavioral subclass $\langle s1, Tx_{rcv} \rightarrow init \rangle$ where $init$ is the initial state of the automata for the guard of *rcv*. In this example, the guard of all action labels other than Tx_{rcv} accepts all histories. The minimal DFA accepting the regular expression guard for Tx_{rcv} is shown in Figure 2.

If a data transfer Tx is now executed, two node objects participate in it playing the role of *snd* and *rcv* in the transaction *Tx*. The names of objects will not be tracked in our symbolic execution mechanism. For convenience of our discussion here, let us just call them "sender" and "receiver". We now have three behavioral subclasses.

1. $\langle s1, Tx_{rcv} \rightarrow init \rangle$ has 18 objects which were idle
2. $\langle s1, Tx_{rcv} \rightarrow sent \rangle$ has 1 object ("sender" object)
3. $\langle s2, Tx_{rcv} \rightarrow init \rangle$ has 1 object ("receiver" object)

In the preceding, objects in different behavioral subclasses have different sets of actions enabled, thereby obviously leading to different future evolutions. Now suppose the "sender" object executes the internal computation in *Erase*. This results in a merger of the first two behavioral subclasses shown in the preceding, that is, the "sender" object is now indistinguishable (behaviorally) from the 18 objects which never participated in any transaction. For all of these 19 objects, the actions Tx_{snd}, Tx_{rcv} and $Erase$ are now enabled. This is the manner in which behavioral subclasses will be split and merged during simulation.

3.2. Simulation of Core Model

Our symbolic simulation keeps track of the objects in various process classes by maintaining the current global configuration. A global configuration consists of one configuration for each of the constituent process classes. To define the configurations of a process class, consider a process class p whose set of behavioral subclasses is B_p. Furthermore, assume that we are running a simulation with N_p objects of p. A configuration of process class p is then a partition of the N_p objects of p into the behavioral subclasses B_p. This is formalized as a mapping $cfg_p : B_p \rightarrow \{0, 1, \ldots, N_p\}$ s.t. $\sum_{b \in B_p} cfg_p(b) = N_p$. Our simulator efficiently keeps track of only the behavioral subclasses with non-zero counts during simulation.

A global configuration is transformed into another by executing a transaction. How can our simulator check whether a specific transaction γ is enabled at the current global configuration c ? Note that the configuration c captures for each process class p, just the number of objects currently in each behavioral subclass of p. Thus, γ is enabled at c if for every lifeline of γ we can assign a distinct object to take up that lifeline (*i.e.* we do not want the same object to act as several lifelines in the same execution of a transaction γ). Since we do not keep track of object identities, how do we assign objects to lifelines ? This is done by assigning a "*witness behavioral subclass*" in c to each lifeline of transaction γ such that if a behavioral subclass b is assigned as the witness of n lifelines in γ, then $c(b) \geq n$, that is, b has at least n objects in configuration c. *This ensures that we do not allow one object to play two different roles in a transaction.* Note that a behavioral subclass $b = \langle s_b, History_b \rangle$ of process class p can serve as a witness subclass for action label γ_r (denoting lifeline r of transaction γ) if and only if the following hold.

- Lifeline r in γ involves objects from class p.
- s_b is a state in TS_p and γ_r is an outgoing transition from s_b in TS_p.
- $History_b$ satisfies the history-based regular expression guard of action label γ_r. Thus $History_b(\gamma_r)$ is an accepting state of the minimal DFA capturing the regular expression guard of γ_r.

In the preceding we described how to check whether a transaction γ is enabled at the current configuration c. If it is, our simulator can execute γ in c leading to a new configuration c'. The new configuration c' is computed by finding how each behavioral subclass of the process classes are affected by the transaction γ. By playing the lifeline (or role) r of a transaction γ, one of the objects of behavioral subclass $b = \langle s_b, History_b \rangle$ of process class p moves to the "destination behavioral subclass" $b' = \langle s_{b'}, History_{b'} \rangle$ where $s_b \xrightarrow{\gamma_r} s_{b'}$ in TS_p, and $History_{b'}$ is obtained by

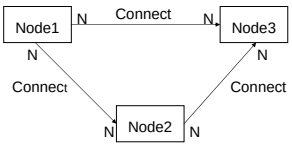

Figure 3: **Example Class Diagram**

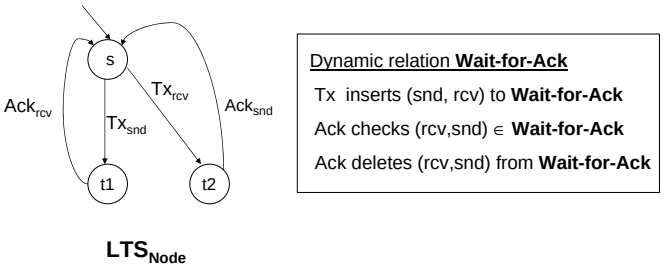

Figure 4: **A process class with dynamic associations**

taking the current states of the automata for the guards according to $History_b$ and then executing the transition γ_r in each of those automata. Computing the new configuration c' as a result of executing transaction γ in configuration c thus involves computing the destination behavioral subclass for each behavioral subclass of a process class, and then computing the new count of objects for each behavioral subclass. Details appear in the technical report [4].

4. Additional Model Features

We discuss the extensions to our core model, namely, static associations and dynamic associations. These extensions are crucial for achieving adequate modeling power.

Static Associations Static associations between process classes are captured by a class diagram. In the example of Figure 1, instead of having the class $Node$ we can have classes *Node1*, *Node2*, *Node3* denoting node objects in three different geographic areas. There can be a *Connect* relation between classes *Node1*/*Node2*/*Node3* as shown by the class diagram in Figure 3. These relations can be used in the guard of a transaction. In our running example (Figure 1) we might want the objects playing the *snd* and *rcv* roles in an execution of transaction *Tx* to be related by the *Connect* relation. This restricts us from choosing two objects from the same geographical area (say two objects of *Node1*) for the roles of *snd* and *rcv* in an execution of transaction *Tx*.

Dynamic Associations Let us consider the *Node* class in Figure 4 where *Tx* and *Ack* are two transactions (not shown) denoting a data transfer protocol and an acknowledgment protocol respectively. The guard of the *Ack* transaction is that the participating objects have *executed together* a data transfer *Tx*. Since this condition involves a relationship between the local histories of multiple objects, we cannot capture it via regular expressions over the individual local histories. Instead, we maintain a dynamic relation called *Wait-for-ack* as part of our specification. We call this relation dynamic, because its contents change during simulation. If nodes x and y play the role of *snd* and *rcv* in *Tx* (refer Figure 4) then the effect of the *Tx* transaction is to insert the tuple (x, y) into the *Wait-for-ack* relation. The *Ack* transaction's guard now includes the check $(rcv, snd) \in$ *Wait-for-ack*; so if objects y and x have been selected to play the *snd* and *rcv* roles in *Ack*, the check will succeed. Furthermore, the effect of *Ack* transaction is to remove the tuple (rcv, snd) from the *Wait-for-ack* relation.

In our model we allow a fixed number of dynamic relations between objects. For any dynamic relation, we describe the effect of each transaction on the relation (in terms of addition/deletion of tuples of objects into the relation). Furthermore, the guard of any transaction can contain a membership constraint on one or more of the specified dynamic relations. In terms of simulation of concrete objects, it is clear how our extended model should be executed. However, since we do not maintain identities of concrete objects during simulation, it is not possible to take the obvious approach. We now describe how we can exploit the notion of behavioral subclasses for this purpose.

5. Simulating the extended model

Our symbolic execution mechanism needs little change in the presence of static associations between process classes. The guard of a transaction can refer to these static associations. So, the only change in the symbolic execution mechanism is that we need to take these associations into account while assigning the witness behavioral subclasses for each lifeline of a transaction.

As for dynamic associations, the key question here is how we maintain relationships between objects if we do not keep track of the object identities. We do so by maintaining *dynamic associations between behavioral subclasses.* To illustrate the idea, consider a binary relation D which is supposed to capture some dynamic association between two objects of process class p. In our symbolic execution, each element of D will be a pair (b, b') where b and b' are behavioral subclasses of class p. To understand what $(b, b') \in D$ means, consider the concrete simulation of the process class p. If after an execution π (a sequence of transactions), two concrete objects o, o' of process class p get related as $(o, o') \in D$ then the symbolic execution along the same sequence of transactions π must produce $(b, b') \in D$ where b (b') is the behavioral subclass in which o (o') resides after executing π. The same idea can be used to manage dynamic relations of larger arities.

Example Let us consider the example in Figure 4. The objects of class *Node* maintain a dynamic relation called *Wait-for-ack*, as shown in Figure 4. Furthermore, assume that the

objects of the *Node* are represented as objects of classes *Node1, Node2, Node3* standing for node objects in different geographical areas. The connectivity among these geographical areas is given by a static relation called *Connect* shown by the class diagram of Figure 3. The guard of Tx requires the pair (snd, rcv) to be in the static relation *Connect*. The guard of Ack demands that the pair (rcv, snd) be in the dynamic relation *Wait-for-Ack*. The history based guard of both these transactions is the vacuous one; all histories are allowed. We do not show the details of the *Tx* and *Ack* protocols here for lack of space.

Now suppose an object of $Node1$ and an object of $Node2$ play *snd* and *rcv* roles in Tx. This is followed by another object of $Node1$ and an object of $Node3$ playing *snd* and *rcv* roles in another execution of Tx. Let b (b')be the behavioral subclass in which an object of *Node* goes to by executing Tx_{snd} (Tx_{rcv}). We now have two objects of $Node1$ in the behavioral subclass b and one object each of $Node2/Node3$ in the behavioral subclass b'. We have

Wait-for-ack = $\{(Node1_b, Node2_{b'}), (Node1_b, Node3_{b'})\}$

where $Node1_b$ denotes the behavioral subclass b of class $Node1$. Now, when we execute an Ack transaction, we will pick a pair from this relation as witness behavioral subclasses for *rcv* and *snd* roles. We *have not* maintained information about which object of $Node1$ transfered data to which object of $Node2$ or $Node3$. But this information is not required for our symbolic simulation to proceed.

6. Experiments

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

In order to be able to work with non-trivial examples, we have made some pragmatic extensions to our modeling language. We allow each object of a process class to have local variables of pre-defined finite types. The names and types of these local variables are part of the process class specification. Bounded arrays of these finite types are also allowed. The transactions are allowed to contain computations on these local variables which are seen as *internal actions* of a transaction. Any internal action can be viewed as a small program in a standard imperative programming language like C, where all data-types used are finite. We also allow the guards of the transactions to contain a propositional logic formula where an atomic proposition is a boolean expression containing local variables, constants and standard arithmetic/relational operators. The valuation of the local variables is taken into account while constructing our behavioral subclasses during simulation.

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

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

6.1. Examples Modeled

For initial experiments, we modeled a simple telephone switch drawn from [6]. It consists of a network of switch objects with the network topology showing the connection between different geographical localities. Switch objects in a locality are connected to phones in that locality as well other switches as dictated by the network topology. We modeled basic features such as local/remote calling as well as call-waiting and three way calling.

Next we modeled the rail-car system whose behavioral requirements have been specified using Live Sequence Charts in [3]. 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 monitoring speed of the cars), proximity-sensor (for checking whether cars are approaching a terminal) etc.

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

The automated rail-shuttle system [11] consists of var-

ious 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 [2] is a an important component of the *Center TRACON Automation System (CTAS)*, automation tools developed by NASA to manage high volume of arrival air traffic at large airports. The case study involves three classes of objects: weather-aware clients, weather control panel 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.

6.2. Simulation Results

Our simulator can be used for random as well as guided simulation. We used guided simulation on each of our examples to test out the prominent use cases; we now summarize these simulation runs. For each example, we summarize the results of three test cases in Table 1.

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

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

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

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

We observe that the number of behavioral subclasses (the groups into which objects are partitioned based on behaviors) is much less than the number of concrete objects. Furthermore, even if the number of concrete objects is increased (say instead of 24 cars in the Rail-car example, we have 48 cars), the number of behavioral subclasses in these simulation runs remain the same. Clearly, our symbolic simulation method is more memory efficient compared to concrete simulation where the states of every object has to be tracked. A possible concern is whether the management of behavioral subclasses in our symbolic simulation method introduces unacceptable timing overheads. To check this, we measured the timings of several randomly generated simulation runs of length 1000 (*i.e.* containing 1000 transactions) in our examples. The maximum time taken by any simulation run was 3.3 seconds on a Pentium-4 3 Ghz machine with 1 GB of main memory (incidentally, this run was an execution trace of the Rail-car example). This time overhead is not high, particularly considering that our simulator uses the OCaml programming language and not C.

We also compared our simulation timings 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. For example, we show the simulation times of the concrete and symbolic simulators on the Rail-car example with two different settings: 24 car objects and 48 car objects. The numbers reported are the maximum time taken by any simulation run drawn from a large set of representative runs each of which contains 1000 transactions. Overall, we observed that the concrete simulator's timing increases appreciably with an increase in number of objects. This is not the case for our symbolic simulator.

	Rail-car (24 cars)	Rail-car (48 cars)
Concrete	4.6 secs	7.8 secs
Symbolic	3.3 secs	3.5 secs

Currently, our simulator supports the following features to help error detection– (a) random simulation for a fixed number of transactions, (b) guided simulation for an use-case, and (c) testing whether a given sequence of transactions is an allowed behavior. Note that for guided simulation in our tool, the entire sequence of transactions

need not be given by the user. *The simulator can be downloaded from* `http://www.comp.nus.edu.sg/~ankitgoe/simulator`

Finally, we describe some experiences in debugging the NASA's CTAS weather-update control system [2] using our simulator. 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 5, we show a snippet of the transition system for CM. We have given the transactions names to ease understanding, for example *Snd_Init_Wthr* stands for "send initial weather" and so on.

In Figure 5, 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 error in the requirements of the controller 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.

7. Discussion

In this paper, we have studied a modeling and simulation methodology for interacting process classes; such classes occur in reactive control systems of various application domains such as telecommunications and transportation. Our models are based on standard UML notations and our *symbolic* simulation strategy allows efficient simulation of realistic designs with large number of objects. The efficacy of our method for efficient simulation and debugging has been demonstrated on realistic reactive control systems. We are currently looking into automated code generation from our models. In the present work, our state and class diagrams are "flat". We plan to extend these state diagrams to

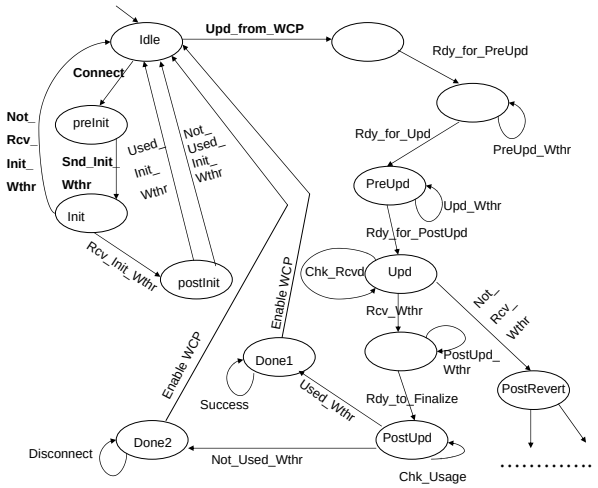

Figure 5: **Snippet of Transition System for Weather-Update Controller**

allow for concurrency and structured classes (of UML 2.0) to specify more sophisticated associations between process classes.

References

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

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

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

[4] A. Goel, A. Roychoudhury, and P. Thiagarajan. Dynamically grouping active objects based on behavior. Technical report, National University of Singapore, 2005. `http://www.comp.nus.edu.sg/~abhik/pdf/TR-05.pdf`.

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

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

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

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

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

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

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

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