Symbolic Message Sequence Charts

Abhik Roychoudhury
School of Computing
National Univ. of Singapore
abhik@comp.nus.edu.sg

Ankit Goel
School of Computing
National Univ. of Singapore
ankit@comp.nus.edu.sg

Bikram Sengupta
IBM India Research Lab
New Delhi, India
bsengupt@in.ibm.com

ABSTRACT

Message Sequence Charts (MSCs) are a widely used visual formalism for scenario-based specifications of distributed reactive systems. In its conventional usage, an MSC captures an interaction snippet between *concrete* objects in the system. This leads to voluminous specifications when the system contains several objects that are behaviorally similar. In this paper, we propose a lightweight syntactic and semantic extension of MSCs, called Symbolic MSCs or SMSCs, where an MSC lifeline can denote some/all objects from a collection. Our extensions give us substantially more modeling power. Moreover, we present a symbolic execution semantics for (structured collections of) our extended MSCs. This allows us to validate MSC-based system models capturing interactions between large, or even unbounded, number of objects. Since our extensions are only concerned with MSC lifelines, we believe that they can be integrated into existing standards such as UML 2.0.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

Design, Languages, Verification

Keywords

Message Sequence Charts, Unified Modeling Language (UML)

1. INTRODUCTION

Message Sequence Charts (MSCs) are widely used by requirements engineers in the early stages of reactive system design. Conventionally, MSCs are used in the system requirements document to describe *scenarios* — possible ways in which the objects constituting a distributed reactive system may communicate among each other as well as with the environment. Due to their widespread usage in requirements engineering, MSCs have been integrated into the Unified Modeling Language (UML). MSCs may be composed to yield complete behavioral descriptions. Such compositions may be captured as a High Level Message Sequence Chart or HMSC. The complete MSC language appears in a recommendation of the ITU [1]. The syntax and process theory based operational semantics of MSCs appear in [2].

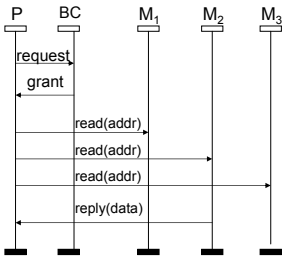

Figure 1: An MSC showing read request from a processor to various memory devices via a bus. The bus controller (BC) controls access to the bus.

The benefits of MSCs notwithstanding, it has been observed that while describing requirements of systems containing many objects, MSC specifications tend to grow too large for human comprehension [2, 3]. We find this problem to be particularly acute when the system contains several objects which conform to a common behavioral protocol when interacting with other objects. Such objects may be considered as instances of a common process *class*. In the absence of suitable abstraction mechanisms, similar interactions involving different objects from the same class have to be repeated to convey all possible scenarios that may occur, thereby leading to voluminous MSC specifications.

Let us consider an example to illustrate this point. Consider a master-slave protocol interaction, where several master processes are competing to get service from the slave processes. An arbiter controls access to the slaves. Furthermore, whenever any master needs service and the arbiter grants access to the slaves, a specific slave will be allocated depending on the kind of service needed. A concrete realization of such an interaction can be observed in bus access protocols. The master processes are the processors hooked to the bus. The slave processes are memory devices from which the processors are trying to read or write. The arbiter is the bus controller which decides, according to some scheduling policy, the processor to be granted bus access. Usually when a processor needs to access a memory address for reading/writing data, it broadcasts the appropriate address and control signals over the bus to various memory

ESEC/FSE'07, September 3–7, 2007, Cavat near Dubrovnik, Croatia.

devices. Then after decoding the address, one of the devices will respond to the processor's read/write request.

Figure 1 shows an MSC capturing the above-mentioned interaction between a processor and memory units. Clearly, if there are n processors hooked to the bus, we will have n such MSCs — all structurally similar! Furthermore, even within each MSC, there is lot of structural similarity. In fact, since MSCs capture point-to-point communication, the broadcast of address by a processor to all memory units is not captured exactly in Figure 1. Instead, the processor sends a read request separately to each of the memory units. In the MSC shown in Figure 1, the read request is for an address which is from the address space of memory unit M_2. Since there are two other memory devices, we would then need to repeat the same interaction to convey the cases when the read request is in the address space of M_1 or M_3.

Clearly, as we increase the number of memory devices and processors in the system, such an approach will not scale up. Individual MSCs will become large, containing many lifelines, and similar MSCs representing essentially the same interaction will have to be repeated. Moreover, since a lifeline may appear in several MSCs, modifying the specification (for example by adding or removing memory devices) may be error-prone and will involve considerable effort. Finally, validation of such specifications will become inefficient as the number of memory devices and processors is increased.

Technical Contributions. The goal of this paper is to propose simple yet powerful extensions to the MSC language to support efficient specification and validation of systems involving classes of behaviorally similar objects. Our extensions are based on the meaning of a lifeline in an MSC. The MSC standard (now integrated into UML 2.0) suggests that a lifeline denotes a concrete object. As the above example indicates, this may lead to voluminous behavioral descriptions that scale poorly. In our extension, we first relax the meaning of a lifeline to consider three possibilities (i) a concrete object (ii) any k objects from a class for a given positive constant k (*existential abstraction*), or (iii) all objects from a class (*universal abstraction*). Moreover, *guards* may be used to further restrict the object(s) that may engage in the events depicted on the lifeline. Thus, if we have a universally abstracted MSC lifeline drawn from process class p with guard g_p, it will be played by all object(s) of class p which satisfy g_p. These extensions yield a concise MSC notation called Symbolic Message Sequence Chart (SMSC).

In this paper, we present the syntax, concrete semantics, and abstract semantics of SMSCs. The abstract semantics allows for efficient symbolic execution of SMSCs showing interactions among process classes with large numbers of objects. It also serves as a formal execution semantics which can be used to reason about interactions between an *unbounded* number of objects. We have employed SMSCs to model a real-life controller- the CTAS weather controller [4] from NASA, which is part of a control system for managing high volume air traffic. Experimental results obtained from the CTAS controller show the utility of our approach in modeling/analyzing real-life control systems with many (behaviorally similar) objects.

2. SYNTAX

The basic building block of our system model is a *Symbolic Message Sequence Chart* or *SMSC*. Like an MSC, a SMSC depicts one possible exchange of messages between a set of objects. However, while a lifeline in an MSC corresponds to one concrete object (henceforth called a *concrete lifeline*), a lifeline in a SMSC may be either concrete or *symbolic*.

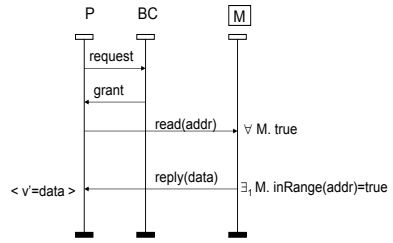

Figure 2: A Symbolic MSC

2.1 Visual Syntax

Graphically, we represent SMSCs as in Figure 2. This SMSC depicts processor-memory interaction, corresponding to the MSC discussed earlier in Figure 1. One important difference between these representations is that the three concrete memory lifelines M_1, M_2 and M_3 appearing in Figure 1, have been merged into a single symbolic lifeline in Figure 2, representing the class of memory devices in the system. Visually, this is depicted in a SMSC by enclosing the symbolic lifeline name by a rectangular box e.g. M in Figure 2. During simulation, a symbolic lifeline may be bound to an arbitrary number of objects. Otherwise, if a lifeline is concrete (*i.e.*, process class contains a single object) its name will appear as it is, e.g. the lifeline corresponding to the bus controller BC in Figure 2.

Within a SMSC lifeline representing a class of objects, a selected subset of objects may engage in events appearing along the lifeline. This selection is performed based on the following criteria associated with each event — (i) valuation of variables of the object, (ii) execution history of the object, and (iii) an abstraction mode that specifies whether all objects (*universal* mode $\forall$) or, any k objects (*existential* mode $\exists_k$) satisfying criteria (i), (ii) may perform the event.

We use the shorthand $\exists_k$ to denote the existential abstraction of lifelines. This is to emphasize that exactly k objects will play such a lifeline. Clearly $k \geq 1$. *For the examples in this paper, we have only used $\exists_1$ (i.e. selecting one object) whenever we used existential abstraction. Henceforth we always assume $\exists_1$ since the extension of our semantics to the general case is trivial. We mention these extensions via footnotes when we present our semantics.*

In Figure 2, initially, the concrete processor lifeline P sends 'read(addr)' message to *all* the memory devices in the system. This is indicated by the universal abstraction with guard *true* for the receive event of message 'read(addr)' by symbolic lifeline M. Subsequently, the memory device in whose address space the read address lies, replies to processor with the required data. Thus, only one memory device replies. This is shown by the existential abstraction $\exists_1$ with guard *inRange(addr) = true* for the send event of message 'reply(data)' by symbolic lifeline M. Note that this single scenario succinctly represents the possibility of *any* device M_1, M_2 or M_3 responding to a processor (refer to Figure 1 to see the interactions between concrete objects), thereby avoiding the need for separate scenarios for each case. Thus SMSCs go far beyond notational shorthands for message

broadcast between scenario lifelines [5]. Furthermore, interaction of the memory devices with *different* processors can also be represented in the same SMSC simply by making the processor lifeline symbolic as well (*i.e.*, the lifeline marked P in Figure 2 also becomes symbolic and denotes *any* processor object). While using SMSCs, it is often the case that every event which appears on a lifeline has the same guard and abstraction mode. In such cases, for ease of specification, the object selection criteria may be written only once, immediately above the lifeline name, with the intended interpretation that the criteria applies to every event shown on the lifeline.

Finally, since variable valuation plays an important role in selecting objects in a symbolic lifeline, SMSCs allow changes in variable valuation to be specified as event postconditions on the lifeline. For example, in the SMSC shown in Figure 2, when P receives the requested data from one of the memory devices, it sets its variable $v = data$. This is shown as $v' = data$ in Figure 2 since for any variable x we show its updated value by its primed version x'.

2.2 Abstract Syntax

The complete MSC language includes several types of events: message sends and receives, local actions, lost and found messages, instance creation and termination and timer events. For SMSCs in this paper, we will only consider message exchange (sends and receives) between lifelines and local actions on individual lifelines. A message m exchanged between two lifelines (representing two concrete objects) p and q in a conventional MSC gives rise to two events: an $\mathsf{out}(p, q, m)$ event denoting the message send event performed by p, and an $\mathsf{in}(p, q, m)$ event that denotes the corresponding receive performed by q. A local action l performed by p is represented by the event $\mathsf{action}(p, l)$. We use A^{MSC} to denote the MSC alphabet consisting of message sends, receives and local actions, for a given set of MSCs. A_p denotes the set of events in A^{MSC} that may be performed by objects in class p.

The notions of lifeline abstraction and event guards in SMSCs necessitate changes in the event syntax as defined above. To explain these, we introduce some auxiliary notation. Let $\mathcal{P}$ denote the set of all process classes[1] with p, q ranging over $\mathcal{P}$. Let G_p^V represent the set of all possible propositional formulae built from boolean predicates regarding the values of the variables owned by p. For example, if p has an integer variable v, then the element $g_p^V \in G_p^V$, where $g_p^V = (v > 5)$, represents those objects of p which have a value greater than 5 for v. Similarly, let G_p^H represent the set of all possible regular expression based execution histories of objects belonging to p. For example, if $g_p^H = (h = A_p^\star.(\mathsf{out}(p, q, m) \mid \mathsf{out}(p, q, n)).A_p^\star)$, where h denotes a variable representing the execution history of an object, then $g_p^H \in G_p^H$, and it denotes those objects of p whose execution history includes the sending of message m or n to object(s) of q. Here A_p represents the set of events that process class p can participate in.

Next, we define an **object selector** of process class p to be an expression of the form $[mode]p.[g_p]$. The square brackets denote optional parts, where *mode* is required only if process class p contains multiple objects (*i.e.* corresponds to a symbolic lifeline); otherwise, p denotes a single concrete object. Further, *mode* represents the abstraction mode and may be either $\exists_1$ for existential ($\exists_k$ in the general case) or $\forall$ for universal interpretations. Also, g_p represents a **guard** which may either be true or consist of a variable valuation constraint $g_p^V \in G_p^V$ and/or an execution history constraint $g_p^H \in G_p^H$. For example, $os_1 = \forall p.(v_1 = 0 \wedge h = A_p^\star.\mathsf{out}(p, q, m).A_p^\star)$ is an object selector for class p that may be used to specify those objects of p whose v_1 variable is set to 0, and whose execution history (denoted by h) involves sending of message m to q. In case $g_p = true$, it indicates that there is no restriction on the variable valuation or the execution history of object(s) to be chosen to play the symbolic lifeline. We use OS^p to denote the set of all object selectors for $p \in \mathcal{P}$, and $OS^{\mathcal{P}}$ denotes the set of all object selectors. We now define the sets of message send events A_{out}, receive events A_{in}, and local action events A_{act}, that are needed for defining the set A^{SMSC} of atomic actions in SMSCs.

Definition 1. *Let $\mathcal{P}$ denote the set of all process classes with $p, q \in \mathcal{P}$. Let $\mathcal{M}$ and $\mathcal{L}$ denote the set of all message names and local action names, respectively. Then the sets A_{out}, A_{in}, A_{act} and A^{SMSC} are defined as follows:*

$$\begin{aligned} A_{out} &= \{\mathsf{out}(os_i, os_j, m) \mid os_i, os_j \in OS^{\mathcal{P}}, m \in \mathcal{M}\} \\ A_{in} &= \{\mathsf{in}(os_i, os_j, m) \mid os_i, os_j \in OS^{\mathcal{P}}, m \in \mathcal{M}\} \\ A_{act} &= \{\mathsf{action}(os_i, l) \mid os_i \in OS^{\mathcal{P}}, l \in \mathcal{L}\} \\ A^{SMSC} &= A_{out} \cup A_{in} \cup A_{act} \end{aligned}$$

We use A_p^{SMSC} to denote the set of all SMSC events that may be performed by objects of class p.

The main difference between MSC events and SMSC events is that in the latter, we use object selectors instead of lifeline names. However, for defining the history based guard of an event, we can simply use the normal MSC events e.g. $h = A_p^\star.\mathsf{out}(p, q, m).A_p^\star$ may be used inside a SMSC event to select object(s) of class p which have previously sent m to some object(s) of class q. Also, for simplicity, we have not included message parameters in the above definition, but our approach may be easily extended to richer message structures. We now formally define a SMSC.

Definition 2 (SMSC). *A SMSC sm can be viewed as a partially ordered set $sm = (L, E_L, \leq)$, where L is the set of lifelines in sm, $E_L = \bigcup_{l \in L} E_l$ where $E_l \subseteq A^{SMSC}$ is the set of events lifeline l takes part in sm; $\leq$ is the partial ordering relation over the occurrences of events in E_L, such that $\leq = (\leq_L \cup \leq_{sm})^\star$ is the transitive closure of $\leq_L$ and $\leq_{sm}$, where*

(a) $\leq_L = \bigcup_{l \in L} \leq_l$, $\leq_l$ is the linear ordering of events in E_l, which are ordered top-down along the lifeline l, and

(b) $\leq_{sm}$ is an ordering on message send/receive events in E_L. If $e_s = \mathsf{out}(os_i, os_j, m)$ and the corresponding receive event is $e_r = \mathsf{in}(os_i, os_j, m)$, we have $e_s \leq_{sm} e_r$.

In our system model, SMSCs may be composed together to yield High-level SMSCs (or HSMSCs) in the same way that MSCs may be arranged in High-level MSCs (HMSCs). A HSMSC is essentially a directed graph whose each node is either a SMSC or (hierarchically) a HSMSC. Formally, a HSMSC is a tuple $H = (N, B, v^I, v^T, l, E)$ where (i) N is a finite set of nodes (ii) B is a finite set of boxes (or supernodes)

[1] A *process class* represents a set of objects following the same behavioral protocol.

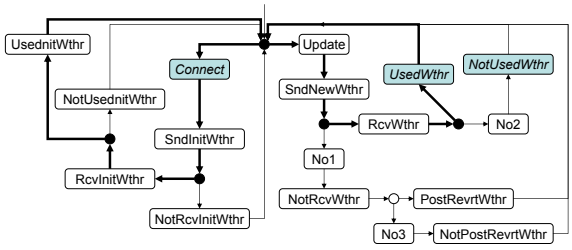

Figure 3: HSMSC for the CTAS case study. The *Connect*, *UsedWthr* and *NotUsedWthr* SMSCs are shown in Figure 4, 5.

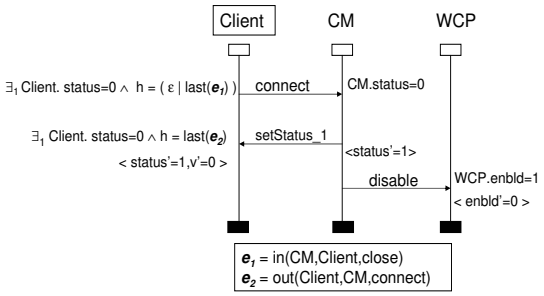

Figure 4: Connect SMSC from CTAS.

representing (already defined) HSMSCs (iii) $v^I \in N \cup B$ is the initial node or box (iv) $v^T \in N \cup B$ is the terminal node or box (v) l is a labeling function that maps each node in N to a SMSC and each box in B to another HSMSC (vi) $E \subseteq (N \cup B) \times (N \cup B)$ is the set of edges that describe control flow. For each process class $p \in \mathcal{P}$, let $\mathcal{V}_p$ be its associated set of variables with function v_p^{init} giving the initial assignment of values to the objects of p. For convenience we assume that all the objects of class p assign the same initial value to any variable $u \in \mathcal{V}_p$.[2] We use $S = \langle H, \bigcup_{p \in \mathcal{P}} \{\mathcal{V}_p, v_p^{init}\}\rangle$ to represent the complete **system specification**, where $H = (N, B, v^I, v^T, l, E)$ is a HSMSC describing the interactions among objects from process classes in $\mathcal{P}$.

Expression Template. In order to make specifications more readable, we identify a commonly occurring regular expression template that we have encountered in our modeling. Consider a process class $p \in \mathcal{P}$ and $e_1 \in A_p$. For regular expressions of the form $h_1 = A_p^\star.e_1$ we write it as $h_1 = last(e_1)$.

3. CTAS CASE STUDY

We now discuss a well known example to illustrate system modeling using SMSCs. The weather update controller [4] is 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. It consists of a central communications manager (CM), a weather control panel (WCP), and several weather-aware clients. The weather-aware clients consist of components such as aircraft trajectory synthesizer, route analyzer etc. which require latest weather information for their functioning. Since the number of clients in the system can be large, the power of lifeline abstraction in SMSCs becomes useful.

Complete behavioral description of the CTAS example as a HSMSC (sans hierarchy) is shown in Figure 3. Various nodes of this HSMSC are labeled with the SMSC names. In the CTAS requirements document [6], the requirements are given from the viewpoint of the CM. All the clients are initially *disconnected* from the controller (CM) and the execution sequence -*2.6.2, 2.8.3, 2.8.5, 2.8.8*- taken from the requirements document, forms the scenario in which a client successfully gets connected to CM. This behavior corresponds to the execution of the left loop (marked using bold lines) in Figure 3, such that the four SMSCs along this marked path correspond to the above four requirements. For example, SMSC *Connect* (shown in Figure 4) represents the requirement *2.6.2* shown below.

Requirement 2.6.2: *The CM should perform the following actions when a weather-aware client attempts to establish a socket connection to CM–* (a) set the weather-aware client's weather status to 'pre-initializing'[3], (b) set the weather-cycle status to 'pre-initializing', (c)disable the weather control panel...

The Client lifeline in *Connect* appears as a symbolic lifeline with its name appearing in a rectangular box. The two events along the Client lifeline: sending message *connect* to CM and receiving message *setStatus_1* from CM, both have existential abstraction. This is because only one client can get connected at a time to CM. Also, they both have propositional guard $status = 0$, showing client status which gives its current interaction stage with CM. However, the history based guards for the two events are different. For sending message *connect* to CM, the regular expression guard for Client is $h = (\epsilon \mid last(e_1))$, where $e_1 = \mathsf{in}(CM, Client, close)$ as shown below in Figure 4. This guard imposes the constraint that either a fresh Client object (having no execution history, and is therefore disconnected), or a Client object which has last been disconnected from CM (due to the receipt of *close* message) can send the 'connect' request to the CM. For the subsequent event of receiving *setStatus_1* message from the CM, the history based guard is $h = last(e_2)$, where $e_2 = \mathsf{out}(Client, CM, connect)$, which allows only the object(s) which have recently sent 'connect' message to CM to execute this event. Note that in specifying the regular expression guards, we have used the template expression $last(e)$ described earlier in Section 2.2.

Further, all the *connected* clients can be updated with the latest weather information by WCP via CM. This behavior corresponds to the execution of the right loop in Figure 3 (marked using bold lines) [4]. In case any client either fails to use the original data (in case it has failed to receive the new data), or update itself (having received the new weather information), all the connected clients get disconnected. These latter scenarios correspond to other execution paths in the CTAS HSMSC.

The two SMSCs shown in Figure 5 show the successful and unsuccessful completion respectively of the weather update cycle for the connected clients. Again in both these SMSCs,

[2]If the initial states of objects in a class are different, we can simply execute additional actions from *our* initial state.

[3]We use integers to represent different status values. For example, in SMSC *Connect* status=1 represents the 'pre-initializing' status.

[4]We do not give the corresponding requirements from the requirements document due to lack of space.

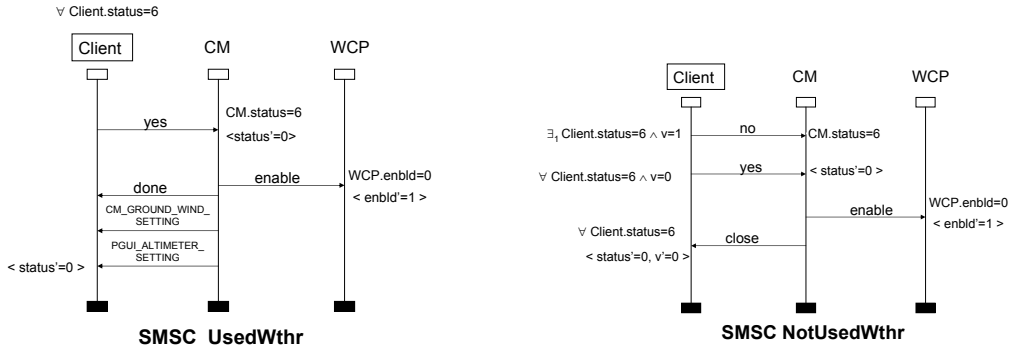

Figure 5: CTAS SMSCs showing successful and unsuccessful completion of weather update.

the *Client* lifeline is symbolic. Further in SMSC *UsedWthr*, since all events along the *Client* lifeline have same guarding expression $\forall Client.status = 6$, it appears only once at the top of the lifeline, which is equivalent to specifying it for each event. The execution sequence in *UsedWthr* takes place when *all* connected clients have responded *yes* to using the new weather update information, and hence the universal abstraction for the *Client* lifeline.

The scenario shown in *NotUsedWthr* SMSC in Figure 5 takes place when one of the clients is unable to use the new weather information, and hence responds with message *no* to CM. This causes CM to send the message *close* to all the connected clients, thereby all of them getting disconnected from CM. Note that in *NotUsedWthr* SMSC, the first event (sending of message *no*) of *Client* has existential abstraction, whereas the subsequent events have *universal* abstraction. Thus, the Client lifeline in the *NotUsedWthr* SMSC shows the use of *mixed abstraction modes within a SMSC lifeline.*

4. PROCESS THEORY

The semantics of the ITU MSC language is defined using a process theory [2]. This theory has a signature Σ that consists of a set of constants and a set of operators. Constants consist of (i) the empty process ϵ (ii) deadlock δ, and (iii) atomic actions from a set Act. Operators comprise the unary operators for iteration $\star$ and unbounded repetition ∞, and the binary operators for delayed choice $\mp$, delayed parallel composition $\|$, weak sequential composition $\circ$, as well as the generalized versions ($\|^S$ and $\circ^S$) of $\|$ and $\circ$, which account for ordering of actions coming from different lifelines e.g. message sends and receives in MSCs. The set of *closed* Σ *terms* is denoted by $\mathcal{CT}(\Sigma)$ (see [2] for details).

4.1 Configurations and Concrete Semantics

To seamlessly integrate our proposed extensions with the ITU framework, we adopt the above theory, but constrain the execution of process terms in this theory by associating a *configuration* element C. As we will see later, we need a notion of configuration to determine at any given point during execution, which object(s) from a given process class can perform a particular action.

In a **concrete execution semantics**, a configuration captures the "local states" of all objects of all process classes. The question is, in a scenario-based modeling language like SMSC, how do we capture the local state of an object. Clearly, we need (a) the object's control flow (which SMSC it is currently executing and which events in the current SMSC have been executed), (b) the variable valuations of the object and (c) a bounded representation of the (unbounded) history of events which allows us to test the satisfaction of history-based regular expression guards in the specification. Here we note that the control flow of objects will be captured by terms in our process theory. The variable valuations will be explicitly captured. Finally, the history information for an object o can be captured by representing the regular expression guards as minimal DFAs and then maintaining the states in these DFAs in which object o lies. Maintaining such information for each individual object for each process class will give a notion of concrete configurations, and the transition between these concrete configurations (using the semantic rules presented below) provides a straightforward concrete execution semantics. However, we will later develop *abstract configurations* to enable (i) efficient simulation of systems with finite number of objects, and (ii) reasoning about systems with infinitely many objects. For this reason, at this stage, we do not impose any concrete structure on configuration C.

We develop the process theory independent of whether configuration C (which appears in the rules of the process theory) is concrete or abstract. We only require C to support two methods (i) a *supports* method, *supports(a)*, which is true iff C permits an action a and (ii) a *migration* method, $migrates\_to(a)$, which returns the set of possible configurations that C migrates to through the execution of a. We use $\mathcal{C}$ to denote the set of all configurations.

4.2 Semantic Rules and Bisimulation

The MSC process theory has an operational semantics defined via deduction rules of the form $\frac{H}{Concl}$, where H is a set of premises and $Concl$ is the conclusion. For ITU MSCs, the semantics for constants consists of two rules (i) $\frac{}{\epsilon\downarrow}$, which implies that the empty process may terminate successfully and immediately; there are no other rules for ϵ as it is unable to perform any action, (ii) $\frac{}{a \xrightarrow{a} \epsilon}$, which expresses that a process represented by the atomic action a can perform a and thereby evolve into the empty process ϵ.

The rules for constants in our theory appear in Table 1. While the empty process may terminate immediately, the execution of an atomic action a is supported by the associated configuration C, leading to a new configuration C'.

Beyond this simple extension, the rest of our technical development is along the lines of the formal MSC theory. For example, our deduction rules for the delayed choice opera-

Table 1: Operational Semantics for Constants

$$\textsf{Const1.}\ \frac{}{\epsilon\downarrow} \qquad \textsf{Const2.}\ \frac{C.supports(a) == true, C' \in C.migrates\_to(a)}{C : a \xrightarrow{a} C' : \epsilon}$$

Table 2: Operational Semantics for Delayed Choice $\mp$

$$\textsf{DC1.}\ \frac{x\downarrow}{x \mp y\downarrow} \qquad \textsf{DC2.}\ \frac{y\downarrow}{x \mp y\downarrow} \qquad \textsf{DC3.}\ \frac{C : x \xrightarrow{a} C' : x', C : y \not\xrightarrow{a}}{C : x \mp y \xrightarrow{a} C' : x'}$$

$$\textsf{DC4.}\ \frac{C : y \xrightarrow{a} C' : y', C : x \not\xrightarrow{a}}{C : x \mp y \xrightarrow{a} C' : y'} \qquad \textsf{DC5.}\ \frac{C : x \xrightarrow{a} C' : x', C : y \xrightarrow{a} C' : y'}{C : x \mp y \xrightarrow{a} C' : x' \mp y'}$$

tor $\mp$ are shown in Table 2. Thus $x \mp y$ can terminate if either x (DC1) or y (DC2) is able to terminate. Except for the addition of the configuration element for the execution semantics rules, the rules are similar in spirit to the MSC rules for $\mp$. If $C : x$ can execute a and $C : y$ cannot (DC3), then on execution of a, the choice is resolved in favor of x. DC4 shows the complementary case when $C : y$ executes a, and $C : x$ cannot. Finally, if both $C : x$ and $C : y$ are able to execute a, then the execution of a does not resolve the choice, but rather, delays it (DC5). Semantic rules for the other operators may be defined similarly (see [2]).

To account for configurations, we also extend the *bisimilarity* based term equality definition in the MSC standard [2]. This definition uses a *permission* relation $\overset{a}{\hookrightarrow}$ on terms; intuitively, $x \overset{a}{\hookrightarrow} x'$ states that an action a is allowed to precede the execution of actions of x even if this action is composed after x by means of sequential composition. The reason for allowing this bypass is that the notion of sequential composition is *weak* and allows instances to proceed asynchronously. The execution of a, however, disables all alternatives in x that do not allow the bypass, and as a result, x evolves to term x'. The deduction rules for $\hookrightarrow$ are presented in [2]. We do not reproduce the details here, but use $\hookrightarrow$ for completeness of our modified term equality definition below.

DEFINITION 3 (BISIMULATION). *Let $\mathcal{C}$ be the set of all possible configurations. A binary relation $B^{\mathcal{C}} \subseteq \mathcal{CT}(\Sigma) \times \mathcal{CT}(\Sigma)$ is called a bisimulation relation iff for all $a \in Act$ and $s, t \in \mathcal{CT}(\Sigma)$ with $sB^{\mathcal{C}}t$, the following conditions hold*

$$\begin{array}{rcl}
\forall_{s' \in \mathcal{CT}(\Sigma), C, C' \in \mathcal{C}} & & \\
(C : s \xrightarrow{a} C' : s' & \Rightarrow & \exists_{t' \in \mathcal{CT}(\Sigma)}(C : t \xrightarrow{a} C' : t' \wedge s'B^{\mathcal{C}}t')) \\
\forall_{t' \in \mathcal{CT}(\Sigma), C, C' \in \mathcal{C}} & & \\
(C : t \xrightarrow{a} C' : t' & \Rightarrow & \exists_{s' \in \mathcal{CT}(\Sigma)}(C : s \xrightarrow{a} C' : s' \wedge s'B^{\mathcal{C}}t')) \\
\forall_{s' \in \mathcal{CT}(\Sigma)}(s \overset{a}{\hookrightarrow} s' & \Rightarrow & \exists_{t' \in \mathcal{CT}(\Sigma)}(t \overset{a}{\hookrightarrow} t' \wedge s'B^{\mathcal{C}}t')) \\
\forall_{t' \in \mathcal{CT}(\Sigma)}(t \overset{a}{\hookrightarrow} t' & \Rightarrow & \exists_{s' \in \mathcal{CT}(\Sigma)}(s \overset{a}{\hookrightarrow} s' \wedge s'B^{\mathcal{C}}t')) \\
s\downarrow & \iff & t\downarrow
\end{array}$$

Two closed terms $p, q \in \mathcal{CT}(\Sigma)$ are bisimilar in $\mathcal{C}$, denoted by $p \leftrightarrow_{\mathcal{C}} q$, iff there exists a bisimulation relation $B^{\mathcal{C}}$ such that $pB^{\mathcal{C}}q$. Note that even though we carry state information in the form of configuration C in above definition, our notion of bisimilarity is *stateless*, which is the most robust notion of equality for state-bearing processes [7]. We can establish the following for our bisimulation relation.

THEOREM 1. *$\leftrightarrow_{\mathcal{C}}$ has the following properties.*

(i) $\leftrightarrow_{\mathcal{C}}$ is an equivalence relation.

(ii) $\leftrightarrow_{\mathcal{C}}$ is a congruence with respect to the function symbols from the signature Σ.

5. ABSTRACT EXECUTION SEMANTICS

We develop an operational semantics for SMSCs using the process theory outlined so far. First, the set of atomic actions *Act* in the process theory is defined as A^{SMSC}, the set of SMSC events as defined in Section 2. In Section 5.1 we explain the translation of SMSC specifications to terms in our process theory. Then, Section 5.2 presents our notion of configuration and a symbolic execution semantics based on these configurations.

5.1 Translating SMSCs to process terms

In general, a SMSC may consist of an arbitrary (but finite) number of events. Semantics is provided for a SMSC body by sequentially composing the semantics of the first event definition with the semantics of the remaining part of the SMSC body. The generalized weak sequential composition operator is used to impose necessary ordering requirements across lifelines. For example, let us consider the *UsedWthr* SMSC in Figure 5. The first event in the SMSC is the sending of message *yes* to CM by all clients with *status*=6; this is represented by the event e_1=$\mathsf{out}(\forall Client.(status = 6), CM.(status = 6), yes)$. The corresponding receive by CM is represented by

$$e_2 = \mathsf{in}(\forall Client.(status = 6), CM.(status = 6), yes)$$

The constraint that e_2 has to follow e_1 is represented by the ordering requirement $e_1 \mapsto e_2$, as in the MSC language. The composition of these two events may be given by $e_1 \circ^{\{e_1 \mapsto e_2\}} e_2$, using the generalized weak sequential operator $\circ^S$, where S is a set of ordering requirements that constrain

execution. Subsequently, CM sends an *enable* message to WCP, represented by the event

$$e_3 = \mathsf{out}(CM, WCP.(enbld = 0), enable)$$

Composing this after e_1 and e_2, we get $e_1 \circ^{\{e_1 \mapsto e_2\}} e_2 \circ e_3$. Since e_2 and e_3 are both performed by CM, the sequential operator ensures their correct ordering, and additional ordering requirements are not needed. Similarly, the subsequent events in the *UsedWthr* SMSC may be composed to obtain the complete event-based behavioral description.

To map HSMSC graphs into event-based descriptions, we follow an approach that is similar to the way a regular expression is obtained from an automaton. Successive applications of a rewrite rule [2] are used to convert the graph into a normal form, ultimately yielding an expression consisting of SMSCs composed via operators like $\mp$, $\circ$ etc. Replacing each SMSC by its corresponding event-based description will then give us the desired event-based representation of the HSMSC graph.

5.2 Representing/Updating Configurations

As mentioned in Section 4, the execution of terms in our theory is constrained by a *configuration*. When we developed our process theory in Section 4, we did not assume a specific definition of configurations, but required our configurations to provide these two functions *supports* and *migrates_to*. In particular, these functions capture the transition between system configurations thereby forming the core of our execution semantics. We develop a notion of **abstract configurations** (Def. 6) and elaborate an abstract execution semantics over these abstract configurations.

Since SMSC events carry guards, we maintain a configuration to keep track of the state of objects during execution, and verify that for each event there are sufficient objects which satisfy its guard. However, maintaining the state of each individual object during simulation can be computationally expensive, and lead to state explosion. To avoid this, the objects of a class are grouped together into *behavioral partitions*. We note that the ability of a p object to perform a SMSC event depends on (i) its execution history and (ii) valuation of its local variables. A behavioral partition for class p represents one possible state of a p object in terms of its execution history and variable valuation.

Definition 4 (Behavioral Partition). *Let V_p be a set of variables belonging to class p. Let R_p denote a set of regular expressions over events A_p (the set of events that may be performed by objects of class p), with $|R_p| = k$. For each $R_i \in R_p$, let D_i be the minimal DFA recognizing R_i. Then a behavioral partition $beh_p(V_p, R_p)$ of class p defined over V_p and R_p is a tuple $\langle q_1, q_2, \cdots, q_k, v\rangle$, where*

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

Q_i is the set of states of automaton D_i and $Val(V_p)$ is the set of all possible valuations of variables V_p. We use $BEH_p(V_p, R_p)$ to denote the set of all behavioral partitions of class p defined over V_p and R_p.

Let us consider any object O of class p with execution history σ. We say that the object O belongs to behavioral partition $\langle q_1, q_2, \cdots, q_k, v\rangle \in BEH_p(V_p, R_p)$ iff (i) q_j is the state reached in the DFA D_j when it runs over σ for each $j \in \{1, 2, \cdots, k\}$ and (ii) the valuation of O's local variables in V_p is given by v. The total number of behavioral partitions of a process class is bounded, provided the value domains of all variables in V_p are also bounded. Also, this number is *independent* of the number of objects in a class.

Next we introduce the notion of a **signature**; for each process class p, a signature contains a set of variables belonging to p and a set of regular expressions over A_p.

Definition 5 (Signature). *For any class p, let V_p be a set of variables belonging to p and R_p be a set of regular expressions over A_p. Then the set of tuples T=$\{(V_p, R_p)\}_{p\in\mathcal{P}}$ is called a signature.*

Given a signature T=$\{(V_p, R_p)\}_{p\in\mathcal{P}}$, the set of all variables in T is then given by $T_v = \bigcup_{p\in\mathcal{P}} V_p$. Similarly, the set of all regular expressions in T is given by T_r=$\bigcup_{p\in\mathcal{P}} R_p$. For any two signatures $T = \{(V_p, R_p)\}_{p\in\mathcal{P}}$ and $T' = \{(V'_p, R'_p)\}_{p\in\mathcal{P}}$, we define their union as a signature $T \cup T' = \{(V_p \cup V'_p, R_p \cup R'_p)\}_{p\in\mathcal{P}}$. Also, we say $T \supseteq T'$ if for all p, $V_p \supseteq V'_p$ and $L(R_p) \supseteq L(R'_p)$ where for a regular expression R we use $L(R)$ to denote the language accepted (set of strings recognized) by the regular expression and $L(R_p) = \bigcup_{R\in R_p} L(R)$. Intuitively, this means that T is defined over a larger state space (variables and execution history) than T'. We now define the notion of an abstract configuration.

Definition 6 (Abstract Configuration). *Let each process class p contain N_p objects, and T=$\{(V_p, R_p)\}_{p\in\mathcal{P}}$ be a signature. An abstract configuration over T is defined as $cfg = \{count_p\}_{p\in\mathcal{P}}$ where $count_p : BEH_p(V_p, R_p) \rightarrow \mathbb{N} \cup \{0\}$ is a mapping s.t. $\Sigma_{b\in BEH_p(V_p,R_p)} count_p(b) = N_p$. The set of all configurations over signature T is $\mathcal{C}^T$.*

Thus, a configuration records the the count of objects in each behavioral partition of each process class. The idea is to dynamically group together objects during execution based on their variable valuation and execution history. This is similar to abstraction schemes developed for grouping processes in parameterized systems [8]. *We note that our notion of configurations and execution semantics permits unboundedly many objects in a system specification.* Thus, in the above definition we could have $N_p = \omega$ and apply our execution semantics provided we assume the usual rules of arithmetic $\omega + 1 = \omega$, $\omega - 1 = \omega$ and so on. Indeed, in Section 6 we present experiments detailing validation of SMSC systems with unbounded number of objects.

We now explain when a SMSC configuration *supports* an event a and the new configuration it *migrates to* on execution of a. These functions appear in rule Const2, Table 1, and are required for defining the transition between system configurations. We begin by defining a mapping $active : A^{SMSC} \rightarrow OS^{\mathcal{P}}$ that indicates which object selector in an event descriptor "causes" the event to occur:

$$active(e) = \begin{cases} os_i & \text{if } e = \mathsf{out}(os_i, os_j, m), \mathsf{action}(os_i, \ell) \\ os_j & \text{if } e = \mathsf{in}(os_i, os_j, m) \end{cases}$$

For any object selector $os = [m]p.[g_p]$, we use the following function.

$$mode(os) \quad = \quad \begin{cases} m \in \{\exists_k, \forall\} & \text{if } p \text{ is symbolic} \\ concrete & otherwise, \end{cases}$$

We also use a function $simple : A^{SMSC} \rightarrow A^{MSC}$ to convert a SMSC event to an MSC event; $simple(e)$ replaces each object selector occurring in e by the corresponding process class. For example, $simple(\mathsf{out}(\forall p.v_1 = 0, q, m))$ $= \mathsf{out}(p, q, m)$.

supports function:. Intuitively, a configuration supports an event defined on process class p if there is at least one p object[5] which satisfies the variable valuation and execution history criteria on the event guard. Since we do not maintain the states of individual objects, this is determined by verifying there is at least one behavioral partition of class p which satisfies the event guard and has a non-zero count of objects. We call such a behavioral partition a *witness partition*, which we formally define below.

DEFINITION 7. *Let $e \in A_p^{SMSC}$ (i.e., active process class in e is p) be a SMSC event and $cfg \in \mathcal{C}^T$ be a configuration defined on signature T=$\{(V_q, R_q)\}_{q \in \mathcal{P}}$. Let ϑ and Λ be the propositional and history based guards in event e. We say that behavioral partition $beh_p = \langle q_1, q_2, \cdots, q_k, v\rangle$ is a* **witness partition** *for event e at configuration cfg if*

(a) $\exists R_i \in R_p$ s.t. $L(R_i) = L(\Lambda)$ (the set of strings accepted by the two expressions are same), Q_i is the set of states in the minimal DFA accepting Λ and $q_i \in Q_i$ is an accepting state of the minimal DFA.

(b) $v \in Val(V_p)$ satisfies the propositional guard ϑ.

(c) $count_p(beh_p) \geq 1$, that is, there is at least one object in the partition beh_p at the configuration cfg.

We use $Witness(e, cfg)$ to represent the set of all behavioral partitions that can act as a witness partition for e at cfg. We are now in a position to define the *supports* function. Let e be a SMSC event and $cfg \in \mathcal{C}^T$ be a configuration. Then, $cfg.supports(e)$=*true* iff there exists a behavioral partition beh, such that beh is a witness partition for e at cfg.[6]

migrates_to function:. We next describe the function $cfg.migrates\_to(e)$, which returns the set of possible destination configurations that result when e is executed at configuration cfg. We first introduce the notion of a *destination partition* — the partition to which an object moves from its "witness partition" after executing an event. We denote the destination partition of beh w.r.t. to e as $dest(beh, e)$.

DEFINITION 8. *Let $beh = \langle q_1, \cdots, q_k, v\rangle \in Witness(e, cfg)$ for a configuration cfg and event $e \in A^{SMSC}$. Let $simple(e)$[7] $= e'$. We then define $dest(beh, e)$ (the* **destination partition** *of beh w.r.t to e) as a behavioral partition $beh' = \langle q_1', \cdots, q_k', v'\rangle$, where*

(a) for all $1 \leq i \leq k$, $q_i \xrightarrow{e'} q_i'$ is a transition in DFA D_i.

(b) v' is the effect of executing e' on v.

Let configuration $cfg \in \mathcal{C}^T$, $e \in A_p^{SMSC}$, $active(e) = os$ and $Witness(e, cfg)$= $\mathcal{B}$. Then $cfg.migrates\_to$ function is defined as follows. In the following $count_p'(b)$ denotes the count of objects in behavioral partition b of process class p after executing e at configuration cfg.

Case 1: $mode(os) = \forall$. Then $cfg.migrates\_to(e)$ returns a unique new configuration that is computed as follows. Let $DP = \{d \mid \exists b \in \mathcal{B}.\ d = dest(b, e)\}$ the set of all destination partitions. Then $\forall d \in DP$, we first set $count_p'(d) = count_p(d) + \Sigma_{b\ s.t.\ dest(b,e)=d}\ count_p(b)$. Then, for all $b \in \mathcal{B}$ we deduct $count_p(b)$ from $count_p'(b)$ to reflect the migration of these objects from b to $dest(b, e)$.

Case 2: $mode(os) = \exists_k$. For $k = 1$, [8] $cfg.migrates\_to(e)$ returns a set of possible new configurations as follows. Let us choose any $b \in \mathcal{B}$, and let $dest(b, e) = d$. Then we set $count_p'(d) = count_p(d) + 1$ and $count_p'(b) = count_p(b) - 1$ to obtain a new configuration (where counts associated with all other partitions remain unchanged). Repeating this for each $b \in \mathcal{B}$ gives us the set of all possible new configurations.

Case 3: $mode(os) = concrete$. Since we are dealing with only one object, we can employ Case 2 for $\exists_1$.

Thus if $mode(e) = \forall$, all objects belonging to all witness partitions for e at cfg, migrate to corresponding new destination partitions.

5.3 Example

We revisit the CTAS example described in Section 3 and show the working of our abstract execution semantics.

In particular, we illustrate the execution of a message send event $e = out(\exists_1 Client.g_1^V \wedge g_1^H, CM, connect)$ in SMSC *Connect* shown in Figure 4, at a given system configuration, where $g_1^V : status = 0$ and $g_1^H : h_1 = (\epsilon \mid last(e_1))$. Assume that only the history based guards -$h_1 = (\epsilon \mid last(e_1))$ and $h_2 = last(e_2)$[9]- shown in SMSC *Connect* appear in the system description for Client process class. Process class CM has got no history based guards. Further, Client class has variables *status* and v, and CM has a single variable *status*. It can be easily seen that the DFAs corresponding to regular expressions h_1 and h_2 contain only two states. Let these be $\{q_0, q_1\}$ and $\{q_0', q_1'\}$ respectively, where q_0 and q_1' are accepting states. q_0 is the state of a Client object ready to connect to the CM. This may either be a fresh Client object (with no execution history), or a Client object that has last received a *close* message from CM. q_1' is the state reached by a client object that has sent a *connect* message to CM in the immediate past.

Assuming 20 *client* objects in a given system specification s.t. 15 of them are ready to connect to CM (i.e. are in state q_0), we consider the following configuration for *Client* class.

$cfg_{Client}(b_1) = 15$, $cfg_{Client}(b_2) = 5$
where $b_1 = \langle q_0, q_0', 0, 0\rangle$ $b_2 = \langle q_1, q_0', 0, 0\rangle$

The first two elements in a behavioral partition descriptor above correspond to the respective states in the two DFAs, while the next two numeric elements represent the values (0 in each case) of variables *status* and v respectively. By executing event e above, a disconnected client sends a connection request to CM. For event e, $active(e) = \exists_1 Client.g_1^V \wedge g_1^H$, i.e., e can be executed by any Client object satisfying the guard $g_1^V \wedge g_1^H$. Following Definition 7, we can determine $Witness(e, cfg) = \{b_1\}$. Thus, there is only one behavioral partition b_1 that can serve as witness partition for e. Any client object from b_1 can now be chosen

[5]To be precise, we need at least one object for events with modes $\exists_1$ or $\forall$. If the mode is $\exists_k$ with $k > 1$, we need at least k objects.

[6]This ensures that there is one witness partition with at least one object satisfying the guards in cfg. If event e's mode is existential abstraction $\exists_k$ with $k > 1$, we need to consider all possible witness partitions in cfg and choose a total of k objects from them.

[7]Recall that $simple(e)$ replaces the object selectors in event e by the corresponding process class names.

[8]The case for $\exists_k$ with $k > 1$ is handled in a similar fashion— except that there may be several witness partitions, and more than one object may be chosen from each witness partition provided the total number of objects is k.

[9]e_1 and e_2 appear at the bottom of MSC *Connect* in Figure 4. Expression $last(e)$ was described at the end of Section 2.2 under 'Expression Template'.

to play event e. After the execution of e, the selected client object will move to states q_1 and q_1' in the two DFAs, and the destination partition (following Definition 8) is given by $dest(b_1, e) = \langle q_1, q_1', 0, 0\rangle$. The new configuration cfg' for *Client* class (following 'Case 2' of **migrates_to** function described earlier) is as follows.

$cfg'_{Client}(b_1) = 14$, $cfg'_{Client}(b_2) = 5, cfg'_{Client}(b_3) = 1$
where $b_1 = \langle q_0, q_0', 0, 0\rangle$, $b_2 = \langle q_1, q_0', 0, 0\rangle$, $b_3 = \langle q_1, q_1', 0, 0\rangle$

Note that one Client object from behavioral partition b_1 has migrated to a new partition b_3.

5.4 Properties of SMSC Semantics

A pertinent question that arises from the above discussion is that given a SMSC specification S, what is the signature $\mathcal{T}$ that we should use to define the configuration space $\mathcal{C}^{\mathcal{T}}$ in which S may be simulated? Let us assume that for any class p, $V_p(S)$ represents the set of all variables that appear on event variable guards or post-conditions on lifeline p in S. Similarly, let $R_p(S)$ denote the set of regular expressions used on event history guards of lifeline p in S. We define signature $T(S)$, the signature derived from S, as $T(S) = \{(V_p(S), R_p(S))\}_{p\in\mathcal{P}}$. Then $T(S)$ represents *a necessary and sufficient signature* to simulate S.

Now, given two SMSC specifications S_1 and S_2, under what signatures $\mathcal{T}$ - or, configuration spaces $\mathcal{C}^{\mathcal{T}}$ - should the bisimulation equivalence of S_1 and S_2 be tested? The following theorems try to address this question.

THEOREM 2. *Let S_1 and S_2 be two SMSC systems and let $T(S_1) \neq T(S_2)$. Then $\forall \mathcal{T} \supseteq T(S_1) \cup T(S_2)$ $S_1 \not\leftrightarrow_{\mathcal{C}^{\mathcal{T}}} S_2$.*

The result follows from the stateless nature of our bisimilarity and the fact that variable/history guards are used to define object selectors, and hence are part of an event.

THEOREM 3. *Let S_1 and S_2 be two SMSC systems such that $T(S_1) = T(S_2) = \mathcal{T}$ and $S_1 \leftrightarrow_{\mathcal{C}^{\mathcal{T}}} S_2$. Then for any signature $\mathcal{T}'$, $S_1 \leftrightarrow_{\mathcal{C}^{\mathcal{T}'}} S_2$.*

Thus, if S_1 and S_2 have the same signatures under which they are bisimilar, they are bisimilar under any signature.

6. EXPERIMENTS

The operational semantics for SMSCs has been implemented as Prolog rules in the XSB logic programming system [9]. XSB supports *tabled* resolution for query evaluation. This speeds up execution by avoiding redundant computation. The operational semantics of SMSCs lend themselves naturally to Prolog rules leading to a straightforward implementation. On the other hand, the underlying well-engineered fixed point engine in XSB ensures that the evaluation of the rules is done efficiently as well. Both concrete and abstract execution semantics of SMSCs are implemented in XSB and they share as much code as possible. We now present the experimental results obtained for the 'CTAS weather controller' example which was described earlier in Section 3. All experiments were conducted on a Pentium-IV 3GHz machine with 1GB of main memory.

Modeling effort. First we briefly discuss the modeling effort required for the CTAS example from given requirements document. The requirements [6] are given as a set of scenarios from the perspective of the controller (or CM - communications manager), with precondition(s) given for the occurrence of each scenario. We were able to directly translate each scenario into corresponding SMSC. The high level control flow (HSMSC) was easily obtained by following the preconditions given for various scenarios. Some characteristic features of the modeled example are shown below. Note that CTAS HSMSC is flat, i.e. each of its node corresponds to a SMSC.

# HSMSC nodes = # SMSCs	17
Total # Events	103
# Events with non-trivial reg. expr. guards	3
# Events with non-trivial propositional guards	65

Simulation. The graphs in Figure 6(a) & (b) compare the simulation time/memory for abstract vs concrete semantics for the CTAS example. The use case used in simulation first connects all the Client objects to the controller (CM) and then performs the weather updates on all the connected clients (refer to Section 3). Different runs correspond to different settings of the CTAS model with number of client objects being varied (shown on the x-axis of the graph). As we can easily observe, for abstract simulation the time/memory remains almost constant ($\approx$ 1.9 sec/40 MB) even as the number of objects is increased from 20 to 100, while it increases at least linearly (3.5 $\rightarrow$ 48.7 sec/100 $\rightarrow$ 900 MB) for concrete simulation. We recall that in abstract simulation, various objects are grouped together into behavioral partitions, unlike in concrete simulation where the states of all objects have to be maintained / manipulated individually.

State Space Exploration. We explored all possible traces up to a certain length, where each step in a trace is the execution of a SMSC event. The results appear in Figure 6(c) & (d), where the bound on the length of traces explored is set to 20. We find that abstract exploration time/memory is constant ($\approx$ 4.8 sec/41 MB) across different settings of CTAS, and increases linearly (19 $\rightarrow$ 139 sec/ 81 $\rightarrow$ 590 MB) for the concrete exploration. Using the abstract execution semantics, we found that the exploration time/memory required for the CTAS model with an *unbounded* number of client objects is same as that for the CTAS model with only 20 client objects (4.8 sec/41 MB). Thus, using our abstract execution semantics, the system designer can try out various system settings (having sufficiently large or even *unbounded* number of objects) without worrying about computation costs. Furthermore, the designer can perform reachability analysis for a system setting with unbounded number of objects to look for falsification of invariant properties in all possible system settings with finitely many objects.

7. RELATED WORK

The MSC language [1] offers two constructs for dealing with the problem of voluminous scenarios involving several instances and events: *gates* and *instance decomposition*. The first option allows a message to be split into two parts, with the message send in one scenario, and the corresponding receive in another scenario, implicitly joined by a gate. The second option allows an instance in one MSC to be decomposed into a collection of instances, whose behavior is depicted in another MSC. These are useful approaches for de-

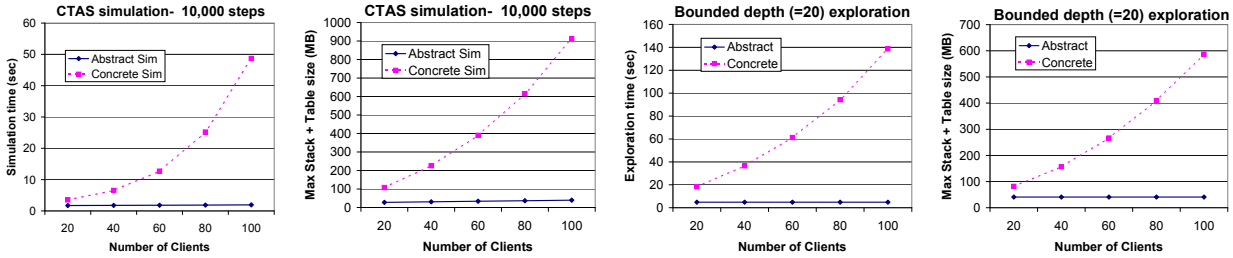

Figure 6: (a) Abstract vs Concrete Simulation Times for Different Settings of CTAS (b) Peak Memory Requirement for Abstract and Concrete Simulation, (c) Abstract vs Concrete State Space Exploration Times, (d) Peak Memory Requirement for Abstract and Concrete State Space Exploration.

composing a large specification into tractable pieces; however, their focus is on structural changes to scenarios rather than behavioral abstractions as in SMSCs, and thus such approaches only partially address the MSC scalability problems discussed in Section 1. Note that in the conventional usage of MSCs, conditions can appear in the MSC syntax. However, there is no attempt to integrate the conditions into the execution semantics of MSCs [2]. On the other hand SMSC event guards not only refer to conditions on variables of concrete objects, but also serve as an object selector from a collection of objects during execution.

The work of [10] presents Interacting Process Classes (IPC), where the behavior of a process class is specified by a labeled transition system and unit interactions between processes are described by MSCs. However, the IPC model does not support universal abstraction of lifelines — only existential abstraction is supported. Furthermore, the MSCs are executed *atomically* in the IPC model, and there is no explicit structuring of MSCs (it is inherent in the control flow of the classes). This makes the IPC model somewhat state-based.

In recent years, a number of MSC variants have been proposed (*e.g.* [11, 12, 13, 14]). Of these, only Live Sequence Charts support symbolic specification, that is, a class of processes can be grouped together into a single symbolic lifeline. However, even LSCs do *not* support symbolic execution [15]. The symbolic lifelines are instantiated to concrete objects during execution in the LSC Play Engine; this increases the number of active copies of LSC universal charts to be maintained. If the chart contains only one symbolic lifeline, the LSC execution semantics will maintain one active chart copy per concrete object, where the number of concrete objects may be huge. If the chart contains several symbolic lifelines (*e.g.* a chart involving caller and callee phone objects in a telecom system), the blow-up is worse.

Overall, the ideas of symbolic execution have not been well supported by existing MSC-based system models. This paper tries to cover the gap by directly integrating symbolic specification/execution into MSCs/HMSCs, a popular design aid for many years.

8. DISCUSSION

In this paper, we have proposed Symbolic Message Sequence Charts (SMSCs) as a lightweight syntactic and semantic extension to conventional MSCs. SMSCs are particularly suitable for behavioral description of systems with many behaviorally similar objects. We presented an abstract execution semantics for SMSCs, in which objects are dynamically grouped together and executed following a process theory based operational semantics. Our approach was validated through a detailed case study and experiments involving a non-trivial weather controller specification.

There are several extensions of SMSCs which add substantial modeling power but involve minimal changes in the execution semantics. One such extension will be to handle requirements of the form "at least (or at most) 10 objects will play a certain role". We can handle it by exploiting the delayed choice operator in our process algebra. In future, we will investigate other requirement templates involving similar processes and integrate them systematically into our framework.

Acknowledgments

This work was partially supported by a Public Sector research grant from A*STAR Singapore.

9. REFERENCES

[1] ITU-TS Recommendation Z.120, 1996. Message Sequence Charts (MSC).
[2] M. A. Reniers. *Message Sequence Chart: Syntax and Semantics*. PhD thesis, Technical University Eindhoven, 1999.
[3] H. Storrle. Semantics of interactions in UML 2.0. In *Visual Languages and Formal Methods (VLFM)*, 2003.
[4] Center-TRACON Automation System (CTAS) for air traffic control. `http://ctas.arc.nasa.gov/`.
[5] I. Kruger, W. Prenninger, and R. Sander. Broadcast MSCs. *Formal Aspects of Computing*, 16(3), 2004.
[6] CTAS requirements document. `http://scesm04.upb.de/case-study-2/requirements.pdf/`.
[7] M. R. Mousavi, M. A. Reniers, and J. F. Groote. Congruence for SOS with data. In *LICS*, 2004.
[8] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In *CAV*, 2002.
[9] XSB logic programming system. `http://xsb.sourceforge.net/`.
[10] A. Goel, S. Meng, A. Roychoudhury, and P. S. Thiagarajan. Interacting process classes. In *ICSE*, 2006.
[11] B. Genest, M. Minea, A. Muscholl, and D. Peled. Specifying and verifying partial order properties using template MSCs. In *FOSSACS*, 2004.
[12] B. Sengupta and R. Cleaveland. Triggered message sequence charts. In *FSE*, 2002.
[13] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 2001.
[14] S. Uchitel, J. Kramer, and J. Magee. Negative scenarios for implied scenario elicitation. In *FSE*, 2002.
[15] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.