Symbolic Message Sequence Charts

ABHIK ROYCHOUDHURY, ANKIT GOEL
National University of Singapore
and
BIKRAM SENGUPTA
IBM India Research Lab

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. MSCs also play an important role in the model-based testing of reactive systems, where they may be used for specifying (partial) system behaviors, describing test generation criteria, or representing test cases. However, since the number of processes in a MSC specification are fixed, model-based testing of systems consisting of process classes may involve a significant amount of rework. For example, reconstructing system models, or regenerating test cases for systems differing only in the number of processes of various types.

In this article, we propose a scenario based notation, called Symbolic Message Sequence Charts (SMSCs), for modeling, simulation, and testing of process classes. SMSCs are a lightweight syntactic and semantic extension of MSCs where, unlike MSCs, a SMSC lifeline can denote some/all objects from a collection. Our extensions give us substantially more modeling power. Moreover, we present an abstract execution semantics for (structured collections of) SMSCs. This allows us to validate MSC-based system models capturing interactions between large, or even unbounded, number of objects. Finally, we describe a SMSC-based testing methodology for process classes, which allows generation of test cases for new object configurations with minimal rework.

Since our SMSC extensions are only concerned with MSC lifelines, we believe that they can be integrated into existing standards such as UML 2.0. We illustrate our SMSC-based framework for modeling, simulation, and testing of process classes using a weather-update controller case-study from NASA.

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

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: Message Sequence Charts, Unified Modeling Language (UML), Symbolic Execution, Test Generation.

An initial version of this paper was published as *Symbolic Message Sequence Charts. Abhik Roychoudhury, Ankit Goel and Bikram Sengupta. In* ESEC/FSE-07. `http://www.comp.nus.edu.sg/~abhik/pdf/fse07.pdf`. Authors' Addresses: A. Roychoudhury and A. Goel, School of Computing, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417; emails: `abhik@comp.nus.edu.sg, ankit.capri@gmail.com`; B. Sengupta, IBM India Research Lab, Embassy Golf Links, Block D, Domlur Ring Road, Bangalore 560071, India; email `bsengupt@in.ibm.com`.

1. INTRODUCTION

In recent years, the use of model-based techniques for system design and development has gained wide acceptance and seen increased usage. The key idea behind the model driven system development is a clear separation of business and application logic from the underlying platform technologies. This separation of concerns immediately offers several advantages, for example, design abstraction, reusability, and portability. Other important benefits of using the model based techniques include — (semi-) automated code generation for obtaining a system implementation (e.g. [Rhap]), validation of functional and non-functional requirements through simulation, testing, model-checking etc. (e.g. [Knapp et al. 2002]), and automated model-based test generation for testing system implementations derived separately from the same requirements (e.g. [Broy et al. 2005]).

In this work, we focus on distributed reactive systems consisting of classes of behaviorally similar interacting processes. Such systems arise naturally in various application domains such as telecommunication, avionics and automotive control. For instance, a telecommunication network with thousands of interacting phones (constituting a phone class), or a controller managing hundreds of clients requiring latest weather information in an air-traffic control system. The initial requirements for such systems generally focus on specifying various inter-process interaction scenarios among system processes, and abstract away from the local computations. Further, at the time of laying out the initial requirements, it is often unnatural to fix, or specify an upper bound on the number of processes of various types (e.g. number of phones in a telecommunication network) in the system. Such systems are also popularly known as *parameterized systems*, the parameter being the number of processes in various classes, and have been well studied in the domain of automated verification (e.g. [Delzanno 2000; Pnueli et al. 2002]).

We find that various existing modeling notations, such as those included in the UML standard (e.g. State-machines and Sequence diagrams), lack certain features required for requirements modeling and validation of such systems. For instance, only a fixed number of lifelines from a class can appear in a Sequence diagram, and this number has to be determined at the time of modeling. While in the case of State-machines, no abstract execution semantics is provided for efficient validation of systems with a large (or even unbounded) number of processes. This lack of features leads to several drawbacks–

(i) *Remodeling:* For different object configurations (differing in the number of objects of various types), separate requirements models need to be obtained in most of the cases. Besides the remodeling effort, various analyses, test generation etc. done over the existing models, may have to be repeated for any newly constructed models. Clearly, this leads to a lot of wasted effort.

(ii) *Problems with validation:* For a requirements model obtained by artificially fixing the number of processes in various classes ,in general, it cannot be guaranteed to exhibit *all* possible system behaviors (when a sufficiently large number of objects are present). Hence, any validation or verification results obtained for the restricted system cannot be guaranteed to hold for all implementations of the given system in general.

(iii) *Scalability:* As the number of objects of various process types is increased

in the system, requirements models may become large and complex, and hence, difficult to maintain and update. For example, in case of Message Sequence Charts [ITU-T Z.120 1999], each process in the system is represented individually. Further, though various notations provide modeling at the level of classes instead of individual processes (e.g. Statecharts [Harel 1987], Live Sequence Charts [Damm and Harel 2001]), their execution semantics is still concrete. This means, that during the execution of requirements models obtained using these notations, various objects and their states in the system are represented individually. Thus, modeling/execution of system requirements with a very large number of objects may face efficiency concerns.

Message Sequence Charts (MSCs) [ITU-T Z.120 1999] is a popular notation for modeling the scenario based requirements, capturing interactions among various processes in the system. In this article, we present our modeling notation of *Symbolic Message Sequence Charts* (SMSCs), which is a light-weight extension of the MSC notation, well suited for requirements modeling, simulation and testing of distributed reactive systems consisting of classes of interacting processes. In the modeling framework that we develop, we impose no restrictions on the number of objects a process class may have. In case, the requirements document does not specify the number of objects for a class, say p, we allow p to have an unbounded number of objects (represented as ω). While modeling the requirements themselves, we do not refer to individual objects of various classes. Instead, we specify the class and constraints for selecting a subset of objects from that class, to participate in a given event. In our setting, the constraints for selecting objects to participate in an event may consist of the following– (i) a boolean guard regarding the valuation of an object's variables, (ii) a history-based guard over the past event-execution history of an object. Further, we propose an *abstract* execution semantics for SMSCs, where behaviorally similar objects are dynamically grouped together into equivalence classes at runtime — only the count of objects in each equivalence class, and not their identities, is maintained. This results in considerable savings in the execution time and memory. Finally, we note that the distributed system requirements highlighting inter-process interactions, are more closely reflected in the scenario-based models such as MSCs. Given the effort involved in deriving a system implementation from such requirements, it is likely to contain errors. Thus, test-cases generated from scenario-based models can provide a crucial link by enabling testing of final implementation with respect to the original requirements [Goel et al. 2009]. We also study the test-generation for process classes based on our notation of SMSCs.

The present article is an improved and expanded version of our conference paper [Roychoudhury et al. 2007], which introduced the notation of SMSCs. In addition to our previous work, which focused on requirements modeling and validation of process classes using SMSCs, in this article we present a model-based testing methodology for process classes based on SMSCs.

In the following, we first introduce syntax of SMSCs in Section 2, and illustrate requirements modeling using SMSCs with the help of a case-study in the subsequent section. We then discuss a process-theory based concrete execution semantics of SMSCs, followed by an abstract execution semantics of SMSCs in Sections 4 and

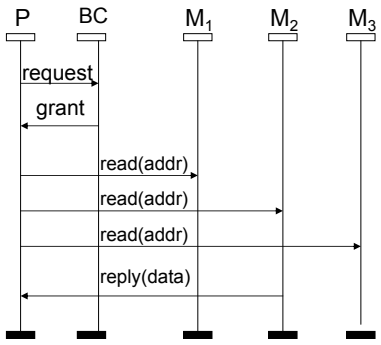


Fig. 1. An MSC showing read request from a processor to various memory devices via a bus. The bus controller (BC) controls access to the bus.

5, respectively. In Section 6, we discuss our SMSC based test generation approach, providing an end-to-end view of our test generation process using CTAS example. Experimental results obtained from the CTAS controller and one other case study are presented in Section 7. The results, (a) show the utility of our approach in modeling/analyzing real-life control systems with many (behaviorally similar) processes, and (b) provide an empirical validation of our methodology of testing process classes. Finally, Section 8 discusses the related work and Section 9 concludes the paper.

2. SYNTAX

Message Sequence Charts (MSCs) [ITU-T Z.120 1999] 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. 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 [Reniers 1999; Storrle 2003]. 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*. 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

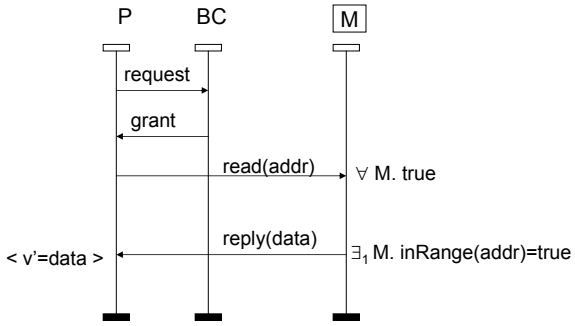


Fig. 2. A Symbolic MSC

address and control signals over the bus to various memory 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.

Our notation of *Symbolic Message Sequence Charts* (SMSCs) introduces simple yet powerful extensions to the MSC language to support efficient specification and validation of distributed reactive systems consisting of process classes. 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. We note that the notion of lifeline abstraction was first introduced in Live Sequence Charts [Marelly et al. 2002] as symbolic instances. Similar to our lifeline abstraction, a symbolic instance in a LSC can be bound to either one or all objects satisfying an associated guard. However, in LSCs the execution remains concrete with a new copy of chart being created for each object binding of a symbolic instance. Thus, charts with infinitely many objects in a class cannot be handled. In this work, we address this issue by introducing an abstract execution semantics for SMSCs.

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*.

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. $\boxed{\text{M}}$ 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. Thus, in case of a symbolic lifeline, different events along the lifeline may be executed by different subsets of objects from the class representing the lifeline. Another possible approach would be to allow multiple lifelines from a class in a SMSC and restrict all events along a lifeline to be executed by a fixed set of objects. However, this would require identifying all possible object combinations (modulo object identities) for executing various events along a symbolic lifeline, hence requiring more effort and resulting in larger specifications.

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 straightforward. We mention these extensions via footnotes when we present our semantics.* Moreover, in Section 9, we mention other variations of existential abstraction which can be incorporated with minimal modifications to our modeling language and its 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 [Kruger et al. 2004]. 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) 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 may denote the single concrete object in the class p.

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

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.

A **postcondition** updating the state of objects executing a given event is specified as a sequence of comma separated assignment statements. For each process class $p \in \mathcal{P}$, let 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 $u \in V_p$.[2] Then, an assignment statement appearing in a postcondition corresponding to an event from class p is of the form $x' = aexpr$, where (i) $x \in V_p$, and (ii) $aexpr$ represents an arithmetic expression involving variables from V_p, integer constants, and the following arithmetic operators '$+, -, /, \times$'. Let $Post^p$ represent the set of postconditions for class $p \in \mathcal{P}$. Further, we use ϵ to represent an empty postcondition.

We now define A^{SMSC}, the set of atomic actions in SMSCs. Let $\mathcal{M}$ and $\mathcal{A}$ denote the sets of message names and local action names, respectively.

Definition 1. *Given, the set of process classes $\mathcal{P}$ with $p \in \mathcal{P}$, message names $\mathcal{M}$ and local action names $\mathcal{A}$, we define the following:*

$$
\begin{aligned}
A^p_{out} &= \{\mathit{out}(os_i, os_j, m, pc) \mid os_i \in OS^p, os_j \in OS^{\mathcal{P}}, m \in \mathcal{M}, pc \in Post^p \cup \{\epsilon\}\},\\
A^p_{in} &= \{\mathit{in}(os_i, os_j, m, pc) \mid os_i \in OS^p, os_j \in OS^{\mathcal{P}}, m \in \mathcal{M}, pc \in Post^p \cup \{\epsilon\}\},\\
A^p_{act} &= \{\mathit{action}(os, a, pc) \mid os \in OS^p, a \in \mathcal{A}, pc \in Post^p \cup \{\epsilon\}\},\\
A_p^{SMSC} &= A^p_{out} \cup A^p_{in} \cup A^p_{act}\\
A^{SMSC} &= \bigcup_{p \in \mathcal{P}} A_p^{SMSC}
\end{aligned}
$$

Here, A_p^{SMSC} denotes the set of SMSC actions that may be performed by objects of class p, while A^{SMSC} denotes the set of all SMSC actions.

The main difference between the MSC and SMSC actions is that in the latter, we use object selectors instead of lifeline names, and include postcondition as part of an action. However, for defining the history based guard of an action, we can simply use the normal MSC actions, e.g. $h = A_p^\star.\mathsf{out}(p,q,m).A_p^\star$ may be used inside a SMSC action 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.

First we define a partially ordered set labeled with actions from A^{SMSC} as a structure $LP = (E, \leq, \lambda)$, where $(E, \leq)$ is a partially ordered set and $\lambda : E \rightarrow$

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

A^{SMSC} is a labeling function. Further, for $p \in \mathcal{P}$, we set $E_p = \{e \in E | \lambda(e) \in A_p^{SMSC}\}$, $E_! = \{e \in E | \lambda(e) \in A_{out}^p, p \in \mathcal{P}\}$, and $E_? = \{e \in E | \lambda(e) \in A_{in}^p, p \in \mathcal{P}\}$.

DEFINITION 2 SMSC. *A SMSC over $(\mathcal{P}, A^{SMSC})$ is a A^{SMSC} labeled partially ordered set $LP = (E, \leq, \lambda)$ satisfying:*

(a) E is a finite set of events.

(b) for each $p \in \mathcal{P}$, there is a total order relation $\leq_p$ on E_p, the events of class p.

(c) there is a relation $\leq_m \subseteq E_! \times E_?$ on message send/receive events in E, satisfying
- *for each $e \in E_!$ there is exactly one $e' \in E_?$ such that $(e, e') \in \leq_m$.*
- *for each $e \in E_?$ there is exactly one $e' \in E_!$ such that $(e', e) \in \leq_m$.*
- *for each $(e, e') \in \leq_m$, if $\lambda(e) = \mathsf{out}(os_p, os_q, m, pc)$, then $\lambda(e') = \mathsf{in}(os_p, os_q, m, pc')$.*

(d) $\leq = ((\bigcup_{p \in \mathcal{P}} \leq_p) \cup \leq_m)^\star$ is the transitive closure of $(\bigcup_{p \in \mathcal{P}} \leq_p)$ and $\leq_m$.

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). An HMSC is essentially a directed graph whose each node is either a MSC or (hierarchically) a HMSC. A HSMSC is defined in a similar manner. For the purpose of our discussion, we assume HSMSCs to be flat, i.e. a node in a HSMSC graph represents a SMSC and not another HSMSC. Formally, a HSMSC is a tuple $H = (N, v^I, v^T, l, E)$ where (i) N is a finite set of nodes (iii) $v^I \in N$ is the initial node (iv) $v^T \in N$ is the terminal node (v) l is a labeling function that maps each node in N to a SMSC (vi) $E \subseteq N \times N$ is the set of edges that describe control flow.

We now define the system specification S.

DEFINITION 3 SYSTEM. *Given, process classes $\mathcal{P}$, a system specification is defined as $S = \langle H, \bigcup_{p \in \mathcal{P}} \{V_p, v_p^{init}\} \rangle$ where $H = (N, v^I, v^T, l, E)$ 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 $e_1, e_2 \in A_p$. 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 [CTAS] 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 [CTRD], the requirements

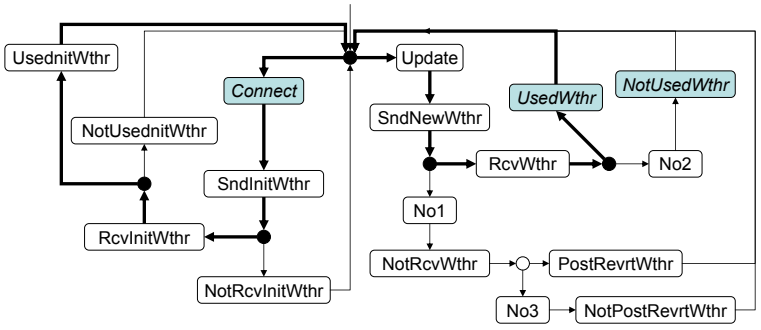


Fig. 3. HSMSC for the CTAS case study. The *Connect*, *UsedWthr* and *NotUsedWthr* SMSCs are shown in Figure 4, 5.

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 SMSC *Connect* 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[4].

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

[4]We assume that the 'connect' message from Client to CM does not appear elsewhere in the

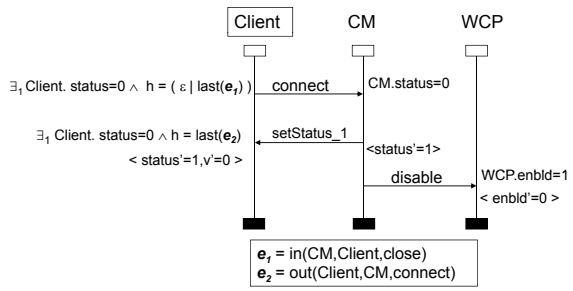


Fig. 4. Connect SMSC from CTAS.

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). In case any client either fails to update itself (having received the new weather information), or use the original data (in case it has failed to receive the new data), all the connected clients get disconnected. These latter scenarios correspond to other execution paths in the CTAS HSMSC.

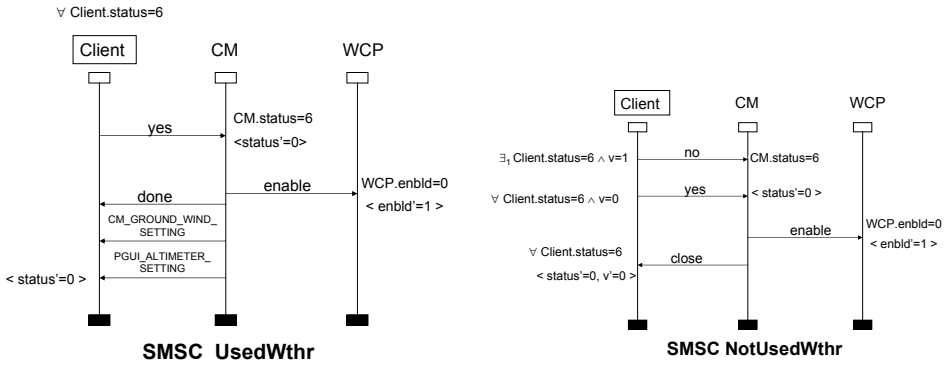


Fig. 5. CTAS SMSCs showing successful and unsuccessful completion of weather update.

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, 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.

model.

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 [Reniers 1999]. 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 iteration $\star$ and unbounded repetition ∞, and the binary operators 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 [Reniers 1999; Mauw and Reniers 1999] for details).

4.1 Configurations

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.

During execution, a configuration is used to capture 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

Table I. Operational Semantics for Constants

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

Table II. Operational Semantics for Delayed Choice $\mp$

DC1. $\dfrac{x \downarrow}{x \mp y \downarrow}$	DC2. $\dfrac{y \downarrow}{x \mp y \downarrow}$	DC3. $\dfrac{C : x \xrightarrow{a} C' : x', C : y \not\xrightarrow{a}}{C : x \mp y \xrightarrow{a} C' : x'}$
DC4. $\dfrac{C : y \xrightarrow{a} C' : y', C : x \not\xrightarrow{a}}{C : x \mp y \xrightarrow{a} C' : y'}$	DC5. $\dfrac{C : x \xrightarrow{a} C' : x', C : y \xrightarrow{a} C' : y'}{C : x \mp y \xrightarrow{a} C' : x' \mp y'}$	

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 by means of deduction rules. A deduction rule is of the form $\dfrac{H}{Concl}$, where H is a set of premises and $Concl$ is the conclusion. For ITU MSCs, the semantics for constants consists of two rules (i) $\dfrac{}{\epsilon \downarrow}$, 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) $\dfrac{}{a \xrightarrow{a} \epsilon}$, 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 are presented in Table I. While the empty process may terminate immediately, the execution of an atomic action a has to be supported by the associated configuration C, and this leads 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 operator $\mp$ are shown in Table II. For closed terms $x, y \in \mathcal{CT}(\Sigma)$, $x \mp y$ 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 [Reniers 1999]).

To account for configurations, we also extend the *bisimilarity* based term equality definition in the MSC standard [Reniers 1999]. 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 [Reniers 1999]. We do not reproduce the details here, but use $\hookrightarrow$ for completeness of our modified term equality definition below.

DEFINITION 4 BISIMULATION. *Let $\mathcal{C}$ be the set of all possible configurations. A binary relation $B \subseteq \mathcal{CT}(\Sigma) \times \mathcal{CT}(\Sigma)$ is called a bisimulation relation iff for all $a \in Act$ and $s, t \in \mathcal{CT}(\Sigma)$ with sBt, the following conditions hold*

$$\forall_{s' \in \mathcal{CT}(\Sigma), C, C' \in \mathcal{C}}(C : s \xrightarrow{a} C' : s' \;\Rightarrow\; \exists_{t' \in \mathcal{CT}(\Sigma)}(C : t \xrightarrow{a} C' : t' \wedge s'Bt')) \quad (1)$$

$$\forall_{t' \in \mathcal{CT}(\Sigma), C, C' \in \mathcal{C}}(C : t \xrightarrow{a} C' : t' \;\Rightarrow\; \exists_{s' \in \mathcal{CT}(\Sigma)}(C : s \xrightarrow{a} C' : s' \wedge s'Bt')) \quad (2)$$

$$\forall_{s' \in \mathcal{CT}(\Sigma)}(s \overset{a}{\hookrightarrow} s' \;\Rightarrow\; \exists_{t' \in \mathcal{CT}(\Sigma)}(t \overset{a}{\hookrightarrow} t' \wedge s'Bt')) \quad (3)$$

$$\forall_{t' \in \mathcal{CT}(\Sigma)}(t \overset{a}{\hookrightarrow} t' \;\Rightarrow\; \exists_{s' \in \mathcal{CT}(\Sigma)}(s \overset{a}{\hookrightarrow} s' \wedge s'Bt')) \quad (4)$$

$$s\downarrow \iff t\downarrow \quad (5)$$

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 such that pBq. 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 [Mousavi et al. 2004].

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.

PROOF. (i) **$\leftrightarrow_{\mathcal{C}}$ is an equivalence relation.**
To prove this we show that following three properties hold–

(a) $\leftrightarrow_{\mathcal{C}}$ is *reflexive*,

(b) $\leftrightarrow_{\mathcal{C}}$ is *symmetric*, and

(c) $\leftrightarrow_{\mathcal{C}}$ is *transitive*.

$\leftrightarrow_{\mathcal{C}}$ is reflexive. It is easy to see that for any bisimulation B as defined in Definition 4, $\forall_{s \in \mathcal{CT}(\Sigma)}\, sBs$ and hence, $\leftrightarrow_{\mathcal{C}}$ is reflexive.

$\leftrightarrow_{\mathcal{C}}$ is symmetric. Let $s, t \in \mathcal{CT}(\Sigma)$ and $s \leftrightarrow_{\mathcal{C}} t$. Thus, there exists at least one bisimulation B (see Definition 4) such that sBt. By exchanging the order of conditions (1) with (2) and (3) with (4), we can conclude that tBs. Hence, $t \leftrightarrow_{\mathcal{C}} s$ and $\leftrightarrow_{\mathcal{C}}$ is symmetric.

$\leftrightarrow_{\mathcal{C}}$ is transitive. Let $s, t, u \in \mathcal{CT}(\Sigma)$ such that $s \leftrightarrow_{\mathcal{C}} t$, $t \leftrightarrow_{\mathcal{C}} u$. Thus, there exist bisimulation relations B_1 and B_2 such that sB_1t and tB_2u. To show that $s \leftrightarrow_{\mathcal{C}} u$, we show that $B_3 = B_1 \circ B_2$ is also a bisimulation relation (as per Definition 4).

Consider $(p,q) \in B_3$. Then, there exists $r \in \mathcal{CT}(\Sigma)$ such that pB_1r and rB_2q. Using condition (1) from Definition 4 we get–

$$\forall_{p' \in \mathcal{CT}(\Sigma), C, C' \in \mathcal{C}}(C : p \xrightarrow{a} C' : p' \Rightarrow \exists_{r' \in \mathcal{CT}(\Sigma)}(C : r \xrightarrow{a} C' : r' \wedge p' B_1 r'))$$
$$\forall_{r' \in \mathcal{CT}(\Sigma), C, C' \in \mathcal{C}}(C : r \xrightarrow{a} C' : r' \Rightarrow \exists_{q' \in \mathcal{CT}(\Sigma)}(C : q \xrightarrow{a} C' : q' \wedge r' B_2 q'))$$

From the above two conditions we get the following–

$$\forall_{p' \in \mathcal{CT}(\Sigma), C, C' \in \mathcal{C}}(C : p \xrightarrow{a} C' : p' \Rightarrow \exists_{q' \in \mathcal{CT}(\Sigma)}(C : q \xrightarrow{a} C' : q' \wedge p' B_3 q'))$$

Note that $p'B_3q'$ since, $p'B_1r'$, $r'B_2q'$ and $B_3 = B_1 \circ B_2$. Hence, condition (1) of Definition 4 holds for B_3. Similarly, we can show the remaining conditions of Definition 4 to hold for the relation B_3. Therefore, B_3 is also a bisimulation.

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

In the following we show that $\leftrightarrow_{\mathcal{C}}$ is a congruence with respect to the delayed choice operator $\mp$ (see Table II) over signature Σ. That is, for all $p, q, r \in \mathcal{CT}(\Sigma)$ $p \leftrightarrow_{\mathcal{C}} q$ implies $p \mp r \leftrightarrow_{\mathcal{C}} q \mp r$. Since $p \leftrightarrow_{\mathcal{C}} q$, there exists a bisimulation B_1 such that pB_1q. We define relation

$$B = \{(p \mp r, q \mp r) | p \leftrightarrow_{\mathcal{C}} q \wedge p, q, r \in \mathcal{CT}(\Sigma)\} \cup B_1 \cup \{(r,r) | r \in \mathcal{CT}(\Sigma)\} \quad (6)$$

and show that B is a bisimulation. To show that condition (1) of Definition 4 holds, we consider the following three cases corresponding to rules –DC3, DC4 and DC5– of the delayed choice operator $\mp$ (see Table II). Consider $(p \mp r, q \mp r) \in B$.

Case I Let $C : p \mp r \xrightarrow{a} C' : p'$ as per rule DC3, Table II for the delayed choice operator, which implies $C : p \xrightarrow{a} C' : p'$ and $C : r \not\xrightarrow{a}$. Now, since $p \leftrightarrow_{\mathcal{C}} q$, we know that there exists q' such that $C : q \xrightarrow{a} C' : q'$ and $p' \leftrightarrow_{\mathcal{C}} q'$ (see Definition 4). Further, from the definition of B above (see Eq. (6)), we get $(p', q') \in B$. Hence,

$$\forall_{p' \in \mathcal{CT}(\Sigma), C, C' \in \mathcal{C}}(C : p \mp r \xrightarrow{a} C' : p' \Rightarrow \exists_{q' \in \mathcal{CT}(\Sigma)}(C : q \mp r \xrightarrow{a} C' : q' \wedge (p', q') \in B)).$$

Case II Let $C : p \mp r \xrightarrow{a} C' : r'$ as per rule DC4, Table II for the delayed choice operator, which implies $C : r \xrightarrow{a} C' : r'$ and $C : p \not\xrightarrow{a}$. Now, since $p \leftrightarrow_{\mathcal{C}} q$, we know that $C : q \not\xrightarrow{a}$. Hence, we get

$$\forall_{r' \in \mathcal{CT}(\Sigma), C, C' \in \mathcal{C}}(C : p \mp r \xrightarrow{a} C' : r' \Rightarrow (C : q \mp r \xrightarrow{a} C' : r' \wedge (r', r') \in B)).$$

Case III Let $C : p \mp r \xrightarrow{a} C' : p' \mp r'$ as per rule DC5, Table II for the delayed choice operator, which implies $C : p \xrightarrow{a} C' : p'$ and $C : r \xrightarrow{a} C' : r'$. Now, since $p \leftrightarrow_{\mathcal{C}} q$, we know that there exists q' such that $C : q \xrightarrow{a} C' : q'$ and $p' \leftrightarrow_{\mathcal{C}} q'$ (see Definition 4). Further, from the definition of B above (see Eq. (6)), we get $(p' \mp r', q' \mp r') \in B$. Hence,

$$\forall_{p', r' \in \mathcal{CT}(\Sigma), C, C' \in \mathcal{C}}(C : p \mp r \xrightarrow{a} C' : p' \mp r' \Rightarrow \exists_{q' \in \mathcal{CT}(\Sigma)}(C : q \mp r \xrightarrow{a} C' : q' \mp r' \wedge (p' \mp r', q' \mp r') \in B)).$$

From the three cases above, we can easily see that condition (1) of Definition 4 holds for the relation B. Similarly, the remaining conditions of Definition 4 can be easily shown to hold for the relation B. Hence B is a bisimulation.

Congruence of $\leftrightarrow_{\mathcal{C}}$ with respect to other function symbols from the signature Σ can be shown in a similar manner. □

The result also follows automatically from [Mousavi et al. 2004] by noting that our term deduction system is in the *process-tyft* format (with negative premises) presented there and moreover, is *stratifiable* [Reniers 1999].

5. EXECUTION SEMANTICS FOR SMSCS

In the following, we develop a concrete and an abstract 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 actions 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 concrete configuration and a concrete execution semantics based on these configurations. To enable efficient simulation of systems with large or even unbounded number of objects, we develop an abstract execution semantics for SMSCs in Section 5.4.

5.1 Translating SMSCs to process terms

In general, a SMSC may consist of an arbitrary (but finite) number of events (see Definition 2). Semantics is provided for a SMSC by sequentially composing the semantics of the first event in the textual description of a SMSC with the semantics of the remaining SMSC textual description. 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=$\mathsf{out}(\forall Client.(status = 6), CM.(status = 6), yes, \epsilon)$. The corresponding receive by CM is represented by

$$e_2 = \mathsf{in}(\forall Client.(status = 6), CM.(status = 6), yes, \langle status' = 0\rangle)$$

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

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

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 [Reniers 1999] 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. A detailed description of HSMSC translation to corresponding process term appears in Appendix C

5.2 Concrete Execution Semantics

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 **concrete configurations** (Def. 7) and elaborate a concrete execution semantics over these concrete configurations.

Since SMSC events carry guards, we maintain a configuration to keep track of the state of objects during execution, and for each event determine the objects which satisfy its guard. 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. We now define the notion of a *behavioral partition* for capturing the execution state of an object. A behavioral partition for class p represents one possible state of a p object in terms of its execution history and variable valuation.

Definition 5 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 the language of 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. We use BEH to represent the set of all behavioral partitions.

Let us consider any object O of class p with execution history σ. Then, O is in the execution state given by 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 6 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 $T_{\mathcal{P}}$=$\{(V_p, R_p)\}_{p\in\mathcal{P}}$ is called a signature.*

Given a signature $T_{\mathcal{P}}$=$\{(V_p, R_p)\}_{p\in\mathcal{P}}$, the set of all variables in $T_{\mathcal{P}}$ is then given by $T^v_{\mathcal{P}} = \bigcup_{p\in\mathcal{P}} V_p$. Similarly, the set of all regular expressions in $T_{\mathcal{P}}$ is given by $T^r_{\mathcal{P}}$=$\bigcup_{p\in\mathcal{P}} R_p$. For any two signatures $T_{\mathcal{P}} = \{(V_p, R_p)\}_{p\in\mathcal{P}}$ and $T'_{\mathcal{P}} = \{(V'_p, R'_p)\}_{p\in\mathcal{P}}$, we define their union as a signature $T_{\mathcal{P}} \cup T'_{\mathcal{P}} = \{(V_p \cup V'_p, R_p \cup R'_p)\}_{p\in\mathcal{P}}$. Also, we

say $T_{\mathcal{P}} \supseteq T'_{\mathcal{P}}$ if for all p, $V_p \supseteq V'_p$ and $R_p \supseteq R'_p$. Intuitively, this means that $T_{\mathcal{P}}$ is defined over a larger state space (variables and execution history) than $T'_{\mathcal{P}}$. We now define the notion of a concrete configuration.

DEFINITION 7 CONCRETE CONFIGURATION. *For each process class p let O_p represent the set of objects of class p, and $T_{\mathcal{P}}$=$\{(V_p, R_p)\}_{p\in\mathcal{P}}$ be a signature. A concrete configuration over $T_{\mathcal{P}}$ is defined as $cfg_c = \{obj_p\}_{p\in\mathcal{P}}$ where $obj_p : O_p \rightarrow BEH_p(V_p, R_p)$, with $BEH_p(V_p, R_p)$ denoting the set of all behavioral partitions of class p (Definition 5). The set of all concrete configurations over signature $T_{\mathcal{P}}$ is $\mathcal{C}_C^{T_{\mathcal{P}}}$.*

We now explain when a concrete SMSC configuration *supports* an event a and the new configuration it *migrates to* on execution of a. These functions appear in rule Const2, Table I, 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:

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

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

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

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 and also removes e's postcondion. For example, $simple(\mathsf{out}(\forall p.v_1 = 0, q, m, pc)) = \mathsf{out}(p, q, m)$.

supports function. A concrete 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. More precisely, let $e \in A_p^{SMSC}$ (i.e., active process class in e is p) be a SMSC event and $cfg_c = \{obj_p\}_{p\in\mathcal{P}} \in \mathcal{C}_C^{T_{\mathcal{P}}}$ be a concrete configuration defined on signature $T_{\mathcal{P}}$=$\{(V_q, R_q)\}_{q\in\mathcal{P}}$. Let ϑ and Λ be the propositional and history based guards in event e. Further, let $O \in O_p$ be a p object whose state is given by $obj_p(O) = \langle q_1, q_2, \cdots, q_k, v\rangle$. Then, object O satisfies the guard of event e at configuration cfg_c if

(a) $\exists R_i \in R_p$ s.t. $L(R_i) = L(\Lambda)$, ($L(R_i)$ and $L(\Lambda)$ are the languages of R_i and Λ, respectively), and q_i is an accepting state of the minimal DFA accepting Λ.

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

We can now define the *supports* function. Let e be a SMSC event and $cfg_c \in \mathcal{C}_C^T$ be a concrete configuration. Then, $cfg_c.supports(e)$=$true$ iff there exists an object whose state satisfies the guard of e.[6]

[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 at least one object satisfying the guards in cfg_c. If event e's mode is existential abstraction $\exists_k$ with $k > 1$, we require at least k objects to satisfy e's guard at cfg_c.

migrates_to function. We next describe the function $cfg_c.migrates\_to(e)$, which returns the set of possible destination configurations that result when e is executed at configuration cfg_c. We first describe the state update of an object O executing event e. Let the state of object O at concrete configuration cfg_c be given by the behavioral partition $\langle q_1, \cdots, q_k, v\rangle$, and $simple(e)$[7] $= e'$. Then, the updated state of object O, resulting after the execution of event e at concrete configuration cfg_c, is denoted as $update(O, cfg_c, e)$ and given by the behavioral partition $\langle q'_1, \cdots, q'_k, v'\rangle$, 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 e's postcondition on v.

Let $O(e)$ denote the set of all objects whose states satisfy the guard of event e at cfg_c, and $active(e) = os$. Then $cfg_c.migrates\_to$ function is defined as follows.

Case 1: $mode(os) = \forall$. Then $cfg_c.migrates\_to(e)$ returns a unique new configuration cfg'_c that is computed as follows. For all objects other than those in $O(e)$, their states at cfg'_c remain unchanged from cfg_c, while for each object $O \in O(e)$, its state at cfg'_c is determined by $update(O, cfg_c, e)$.

Case 2: $mode(os) = \exists_k$. For $k = 1$, [8] $cfg_c.migrates\_to(e)$ returns a set of possible new concrete configurations as follows. Any object $O \in O(e)$ is chosen to execute the event e, and a new configuration is obtained by changing the state of O to $update(O, cfg_c, e)$ (states of all other objects remain unchanged). Repeating this for each object in $O(e)$ 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 $mode(os) = \forall$, then all objects satisfying the guard of event e are chosen to execute it, with their states updated accordingly in the resulting configuration. On the other hand, if $mode(os) = \exists$, then any one object which satisfies e's guard is selected to execute e, resulting in a new configuration with its state updated.

5.3 Example Illustrating Concrete Execution

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

For illustration, consider the execution of a message send event $e = out(\exists_1 Client.g_1^V \wedge g_1^H, CM, connect, \epsilon)$ in SMSC *Connect* shown in Figure 4, at a given concrete 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 does not have any 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

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

[8] The case for $\exists_k$ with $k > 1$ is handled in a similar fashion.

[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'.

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. State 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. State q'_1 is the state reached by a client object that has sent a *connect* message to CM in the immediate past.

Let cfg_c be a concrete configuration at which the state of a Client object O_C is given by the behavioral partition $\langle q_0, q'_0, 0, 0\rangle$. The first two elements of the O_C's state correspond to the states in the two DFAs h_1 and h_2, while the next two numeric elements represent the values (0 in each case) of variables *status* and *v*. 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 the description of *supports* function given earlier in Section 5.3, we can easily see that O_C's state satisfies the guard of event e, and hence O_C can be chosen to execute e. After executing e, O_C will move to states q_1 and q'_1 in the two DFAs, with its updated state given by $\langle q_1, q'_1, 0, 0\rangle$ at the resulting configuration (following 'Case 2' of *migrates_to* function described in Section 5.3).

5.4 Abstract Execution Semantics

As described in the preceding section, our concrete execution semantics maintains the state of each individual object during execution. However, maintaining the state of each individual object during simulation can be computationally expensive, and lead to state explosion. To overcome this, we develop an *abstract* execution semantics for SMSCs in the following. In our abstract execution semantics, instead of maintaining individual states of various objects, we group together objects whose states are determined by the same *behavioral partition* (see Def. 5) at runtime, and maintain only the count of objects (and not their identities) in each behavioral partition. We consider any two objects O_1 and O_2 of a process class p to be in the same *behavioral partition* in $BEH_p(V_p, R_p)$, if and only if they have the same valuation of local variables in V_p, and their current histories lead to the same state for DFAs corresponding to the regular expressions in R_p.

We now define the notion of an abstract configuration.

Definition 8 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 $T_\mathcal{P}$ is defined as $cfg = \{count_p\}_{p\in\mathcal{P}}$ where $count_p : BEH_p(V_p, R_p) \rightarrow \mathbb{N} \cup \{\omega\}$ is a mapping s.t. $\Sigma_{b\in BEH_p(V_p,R_p)} count_p(b) = N_p$. The set of all abstract configurations over signature $T_\mathcal{P}$ is denoted as $\mathcal{C}^{T_\mathcal{P}}$.*

Thus, an abstract configuration records 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 [Pnueli et al. 2002]. *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 $N_p = \omega$, with ω representing an unbounded number of objects in class p. Further, for class p with $N_p = \omega$, we define a cutoff number

$cut_p \in \mathbb{N}$ such that $cut_p + n = \omega$, $n \geq 1$. By default, we assume $cut_p = 1$. To formulate our abstract execution semantics, we define the following two arithmetic operations—$\oplus, \ominus : \mathbb{N} \cup \{\omega\} \times \mathbb{N} \cup \{\omega\} \rightarrow \mathbb{N} \cup \{\omega\}$—representing addition and subtraction involving ω. For all $n_1, n_2 \in \mathbb{N} \cup \{\omega\}$, representing object counts of class p:

$$n_1 \oplus n_2 = \begin{cases} \omega, & \text{if } N_p = \omega \text{ and } n_1 + n_2 > cut_p \\ n_1 + n_2, & \text{otherwise} \end{cases} \quad (7)$$

$$n_1 \ominus n_2 = \begin{cases} \omega, & \text{if } n_1 = \omega, n_2 \neq \omega \\ n_1 - n_2, & \text{otherwise} \end{cases} \quad (8)$$

Indeed, in Section 7 we present experiments detailing validation of SMSC systems with unbounded number of objects.

As in the case of concrete execution semantics discussed earlier in Section 5.3, we now describe the abstract versions of the *supports* and *migrates_to* functions for our abstract execution semantics. Again, these functions form the core of our abstract semantics (see rule Const2, Table I), required for defining the transition between abstract configurations.

supports function. Intuitively, an abstract configuration supports an event defined on process class p, if there is at least one p object[10] which satisfies the variable valuation and execution history criteria on the event guard. Since we do not maintain the states of individual objects in an abstract configuration, this is determined by verifying that 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 9 Witness Partition. *Let $e \in A_p^{SMSC}$ (i.e., active process class in e is p) be a SMSC event and $cfg \in \mathcal{C}^{T_\mathcal{P}}$ be an abstract configuration defined on signature $T_\mathcal{P}$=$\{(V_q, R_q)\}_{q \in \mathcal{P}}$. 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. $L(R_i) = L(\Lambda)$, ($L(R_i)$ and $L(\Lambda)$ are the languages of R_i and Λ, respectively), and q_i is an accepting state of the minimal DFA accepting Λ.

(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 for abstract execution. Let e be a SMSC event and $cfg \in \mathcal{C}^T$ be an abstract configuration. Then, $cfg.supports(e)$=*true* iff there exists a behavioral partition *beh*, such that *beh* is a witness partition for e at cfg.[11]

[10] 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.

[11] 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.

migrates_to function. We next describe the function $cfg.migrates\_to(e)$ for abstract execution, which returns the set of possible abstract 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 with respect to e as $dest(beh, e)$.

Definition 10 Destination Partition. *Let* $beh = \langle q_1, \cdots, q_k, v\rangle \in Witness(e, cfg)$ *for an abstract configuration* cfg *and event* $e \in A^{SMSC}$. *Let* $simple(e)$[12] $= e'$. *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* e*'s postcondition on* v.

Let abstract 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 abstract configuration that is computed as follows. Let $DP = \{d \mid \exists b \in \mathcal{B}.\ d = dest(b, e)\}$ be the set of all destination partitions. Then $\forall d \in DP$, we first set $count'_p(d) = count_p(d) \oplus \Sigma_{b\ s.t.\ dest(b,e)=d}\ count_p(b)$. 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 $k = 1$,[13] $cfg.migrates\_to(e)$ returns a set of possible abstract configurations as follows. Let us choose any $b \in \mathcal{B}$, and let $dest(b, e) = d$. Then we set $count'_p(d) = count_p(d) \oplus 1$ and $count'_p(b) = count_p(b) \ominus 1$ 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 abstract configurations.

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

Thus, if $mode(os) = \forall$, all objects belonging to all witness partitions for e at cfg migrate to corresponding new destination partitions. On the other hand, if $mode(os) = \exists$, then any one object belonging to any one witness partition for e at cfg will migrate to a new destination partition.

5.5 Example Illustrating Abstract Execution

We illustrate our abstract execution semantics using the CTAS example (see Section 3). Again, we consider the execution of the message send event $e = out(\exists_1 Client.g_1^V \wedge g_1^H, CM, connect, \epsilon)$ in SMSC *Connect* shown in Figure 4, at a given abstract configuration, where $g_1^V : status = 0$ and $g_1^H : h_1 = (\epsilon \mid last(e_1))$.

[12] Recall that $simple(e)$ replaces the object selectors in event e by the corresponding process class names and removes e's postcondition.

[13] 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.

As described earlier in Section 5.3, we assume two history based guards h_1 and h_2 for the Client process class and none for the CM class. Further, the DFAs corresponding to the regular expressions h_1 and h_2 contain only two states, given by $\{q_0, q_1\}$, and $\{q'_0, q'_1\}$, respectively, where q_0 and q'_1 are the accepting states. State q_0 is the state of a Client object ready to connect to the CM. Also, two variables *status* and v are defined in the Client class, while CM contains a single variable *status*.

Assuming 20 Client objects in a given system specification, such that, 15 of them are ready to connect to CM (i.e., are in state q_0 of the DFA representing guard h_1), we consider the following abstract configuration for Client class–

$cfg_{Client}(b_1) = 15$, $cfg_{Client}(b_2) = 5$, where $b_1 = \langle q_0, q'_0, 0, 0\rangle$ $b_2 = \langle q_1, q'_0, 0, 0\rangle$.

Here, b_1 and b_2 are the behavioral partitions of the Client class. The first two elements in b_1 and b_2 correspond to the states in the minimal DFAs representing the history guards h_1 and h_2, respectively, while the next two numeric elements represent the values (0 in each case) of variables *status* and v. By executing event e above, a disconnected Client sends a connection request to CM. Following Definition 9, 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 to execute 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 10) is given by $dest(b_1, e) = \langle q_1, q'_1, 0, 0\rangle$. The new abstract 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.

5.6 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 concrete/absrtact 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 $T(S) = \{T_p(S) = (V_p(S), R_p(S))\}_{p\in\mathcal{P}}$. Then $T(S)$ represents *a necessary and sufficient signature* to simulate S.

Given such a mechanism for obtaining a signature from SMSC specifications, it is reasonable to ask the following question. 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* $\forall \mathcal{T} \supseteq T(S_1) \cup T(S_2)$ $S_1 \not\leftrightarrow_{\mathcal{C}^{\mathcal{T}}} S_2$.

PROOF. Note that, for a SMSC specification S having signature $T(S) = \{T_p(S)\}_{p\in\mathcal{P}}$, the variables in $V_p(S)$ and regular-expressions in $R_p(S)$ are part of various events appearing in S. Now, considering S_1 and S_2, clearly there exists an event in S_1 not equal to any event in S_2 (since, $T(S_1) \neq T(S_2)$), or vice-versa. Hence, from the definition of bisimulation (see Definition 4) it is easy to see that we cannot guarantee that $S_1 \leftrightarrow_{\mathcal{C}^{\mathcal{T}}} S_2$ under a signature $\mathcal{T} \supseteq T(S_1) \cup T(S_2)$. □

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 $\mathcal{T}'$, $S_1 \leftrightarrow_{\mathcal{C}^{\mathcal{T}'}} S_2$.*

PROOF. Since, $S_1 \leftrightarrow_{\mathcal{C}^{\mathcal{T}}} S_2$, there exists a bisimulation relation B such that $S_1 B S_2$ (see Section 4.2). Then, for an action $a \in Act$, from condition (1) of Definition 4 we get:

$$\forall_{S_1'\in\mathcal{CT}(\Sigma),C,C'\in\mathcal{C}^{\mathcal{T}}}(C : S_1 \xrightarrow{a} C' : S_1' \Rightarrow \exists_{S_2'\in\mathcal{CT}(\Sigma)}(C : S_2 \xrightarrow{a} C' : S_2' \wedge S_1' B S_2')) \quad (9)$$

Next, for any signature $\mathcal{T}'$ we define a binary relation $B' = B$. Let action a be executable from S_1 at some configuration $C_1 \in \mathcal{C}^{\mathcal{T}'}$, leading to configuration $C_1' \in \mathcal{C}^{\mathcal{T}'}$. Then, a can also be executed from S_2 at configuration C_1 and leading to configuration C_1'. This is because, the event-guard and postcondition being part of an event itself, following condition (9) above if an event can be executed from S_1 at any given configuration C, it can also be executed from S_2 at configuration C, with both executions leading to the same destination configuration. Note that, the set of actions executable from S_1 at a configuration $C_1 \in \mathcal{C}^{\mathcal{T}'}$ will be a subset of actions executable from S_1 at a configuration $C \in \mathcal{C}^{\mathcal{T}}$. Furthermore, we have $S_1' B' S_2'$ (since $B' = B$). Hence, we get

$$\forall_{S_1'\in\mathcal{CT}(\Sigma),C,C'\in\mathcal{C}^{\mathcal{T}'}}(C : S_1 \xrightarrow{a} C' : S_1' \Rightarrow \exists_{S_2'\in\mathcal{CT}(\Sigma)}(C : S_2 \xrightarrow{a} C' : S_2' \wedge S_1' B' S_2')).$$

Thus, condition (1) of Definition 4 holds for relation B'. The remaining conditions of Definition 4 can be shown to hold in a similar manner. Hence, B' is a bisimulation and $S_1 \leftrightarrow_{\mathcal{C}^{\mathcal{T}'}} S_2$. □

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

6. TEST GENERATION FROM SMSCS

Model-based testing is a well-known software development activity. A key aspect of model-based testing is the generation of test-cases from a behavioral model of the system, derived from the informal system requirements. Scenario-based notations, such as MSCs, lend themselves easily for modeling distributed reactive system requirements and play an important role in the reactive systems testing (*e.g.*, [Lee and Cha 2003; Pickin et al. 2007]). In the following, we present a method for model-based testing of reactive systems, consisting of process classes, based on our notation of Symbolic Message Sequence Charts (SMSCs).

Various challenges arise in the model-based test case generation of distributed reactive systems involving *process classes*, or collections of behaviorally similar interacting processes. To begin with, there is the need for *flexibility*; we cannot impose an artificial limit on the number of process class objects in the requirements specification and derive test cases only for that configuration, as the runtime configuration can differ. Secondly, test cases need to be *reusable*; if objects are added or

removed, then we should not have to regenerate test cases from scratch provided the *test-purpose* (or, property to be tested), remains the same. Finally, the set of test cases should be *optimal*: they should include all interesting behaviors corresponding to the test-purpose, but at the same time, the test cases should be *behaviorally distinct*. This last property is important because process class objects being behaviorally similar, there is always a possibility of redundant test cases—depicting the same interaction pattern but involving different combinations of objects—being generated. This may lead to significant wastage of resources, both in the generation of redundant test cases, as well as their subsequent detection/removal or their execution.

Our approach begins with modeling process class requirements using SMSCs. Apart from a SMSC-based system model, a second input to our testing framework is a user-provided test-purpose. A test-purpose, which is also modeled using SMSCs, aids in selecting interesting behaviors from a system-model against which the user wishes to test a system implementation. Given a system model and a test-purpose, our test generation method first automatically generates a set of *abstract* test cases as SMSCs that satisfy the test-purpose. The lifelines in an abstract test case being symbolic, concrete test cases need to be derived, containing precise information regarding the objects executing various events. The abstract test cases are generated by exploring the system model to determine execution traces that contain all the test-purpose events (except the forbidden events) in the appropriate order, possibly interspersed with other events from the model. This step effectively involves parallel composition of the Labeled Transition System (LTS) capturing the system model behavior, with the LTS describing the test-purpose. Next, we perform a novel step called *template generation*. Here, an abstract test case is refined into set of *templates*, where each template represents a behaviorally distinct realization of the abstract test case, and also encodes the minimum number of concrete objects of each class that would be needed to realize the test case fully. Moreover, taken together, the generated templates represent all possible realizations of the abstract test case that may occur in practice. The templates thus bring to our test generation framework the much desired characteristics of flexibility, reusability and optimality – they do not impose any limit on the number of process class objects, they may be re-used to generate concrete test cases for different configurations, and they are behaviorally distinct from each other, while together representing all system behaviors that satisfy the abstract test case. Finally, for generating a set of concrete test cases, the user has to provide a set of concrete objects for various process classes. A minimal set of concrete test cases in the form of MSCs, corresponding to the test-purpose, is then generated by instantiating lifelines in each template once with the concrete objects. Further, all possible concrete test cases can be generated from the minimal set by simply exchanging different object identities. The concrete MSC test cases are used for testing the system implementation, which is derived either manually, or generated (semi-automatically) from a system model different from the one used for test-case generation.

6.1 Test Generation Methodology

We now describe the steps in our test generation methodology (see Figure 7) with the help of an example based on the CTAS case-study discussed earlier in Section 3.

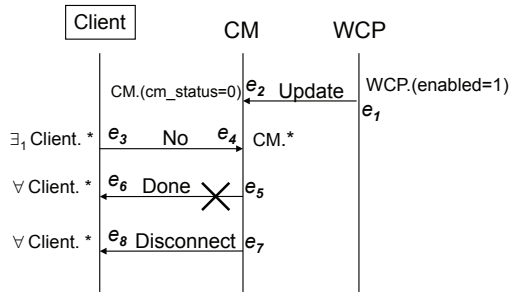


Fig. 6. A test-purpose: **TP**.

The goal is to provide readers with a high-level overview of the end-to-end process. The test generation starts with the following two inputs– (i) a system model $\mathbf{Spec} = \langle H, \bigcup_{p\in\mathcal{P}}\{V_p, v_p^{init}\}\rangle$ with each process class $p \in \mathcal{P}$ having $n_p \in \mathbb{N} \cup \{\omega\}$ number of objects (ω represents an unbounded number), and (ii) a user provided test-purpose **TP**, as described in the following.

Test-purpose Specification. A test-purpose [Tretmans and Brinksma 2002] usually corresponds to an important use-case, or some corner-case scenario more likely to contain errors. In our setting, a test-purpose **TP** is specified as a SMSC S_{TP}, and represents a template behavior, for which the user wants to generate the test-case(s). In *addition* to the usual SMSC elements, a test-purpose SMSC may contain following elements.

(1) A *forbidden* message, say m, that is not allowed to occur at specified locations in a test case satisfying the given test-purpose. Visually this corresponds to a *cross* appearing on the message m's arrow. For example, message *Done* (from *CM* to *Client*) in the test-purpose shown in Figure 6 is a forbidden message.

(2) The guard g of an object selector $[m]p.[g]$ in a test-purpose event can be specified as '*' (or don't care), indicating that it represents *any* guard expression. A test-purpose event with a *don't care* guard can match multiple events differing only in guard(s). For example, consider the object selector '$\exists_1$`Client.*`' corresponding to the sending of message *No* by lifeline *Client* in the test-purpose shown in Figure 6.

For illustration, consider the test-purpose SMSC shown in Figure 6. This test-purpose corresponds to a use-case for an unsuccessful weather update of the Clients connected to CM in the CTAS system described earlier in Section 3. The initial *Update* message from WCP to CM represents a weather update request initiated by WCP. The following message *No* indicates the failure of a Client object to either receive new weather information, or revert back to using old weather information — eventually resulting in all Clients getting disconnected (indicated by the *Disconnect* message sent by CM to all connected Clients). The forbidden *Done* message appearing between the exchange of *No* and *Disconnect* messages checks against the erroneous possibility of a *Done* message being sent by CM to the connected Clients, indicating a successful weather update.

The overall flow of our test generation method appears in Figure 7. We now briefly discuss the three steps of this method.

6.1.1 *Deriving abstract test case SMSC.* In the initial step, we generate abstract test cases in the form of SMSCs from the system model corresponding to a given test-purpose. The abstract test generation procedure involves execution of system model **Spec** guided by the test-purpose **TP** (oval 1, Fig. 7). An abstract test case SMSC corresponds to a finite path in HSMSC H describing the system model and contains all the test-purpose events (except for the *forbidden* events) according to the partial order specified by the test-purpose SMSC (possibly interspersed with other events appearing in H). For example, an abstract test case SMSC generated from the CTAS HSMSC corresponding to the test-purpose shown in Figure 6 appears in Figure 8(a). The message names appearing in bold italics in the test case SMSC (Fig. 8(a)) represent the matching events in the test-purpose (Fig. 6). To reduce visual clutter, we have omitted the object selectors and postconditions for certain events. Further, various intermediate messages exchanged are also not shown in the test case SMSC; these are represented as broken line segments ($\approx$) along lifelines in Figure 8(a). The abstract test case shown in Figure 8(a) represents an unsuccessful weather update scenario for the Clients connected to CM in the CTAS system.

The worst case time complexity for generating all the abstract test cases from a system model **Spec** corresponding to a test-purpose **TP** is as follows. Let,

- E_H be the maximum number of outgoing edges from a node in the HSMSC H describing **Spec**,
- D be the maximum length of a path explored in HSMSC H[14],
- N_H be the maximum number of events in a SMSC corresponding to a node in H, and
- N_{TP} be the number of events in **TP**.

Then, the worst case time complexity for abstract test generation step is $O((E_H)^D.N_H.N_{TP}.D)$. Here, the term $(E_H)^D$ determines the maximum number of paths of length D that can be explored in H. And for each path explored in H, $N_H.N_{TP}.D$ determines the maximum possible comparisons between the system model and test-purpose events.

6.1.2 *Deriving templates.* For testing a system implementation, concrete test cases need to be derived in the form of MSCs from the abstract test case SMSCs. For this, we need to determine concrete objects for executing various events in the abstract test case SMSC. A straightforward approach for doing this is by executing an abstract test case SMSC S using our concrete SMSC execution semantics (see Section 5.3), specifying a concrete set of objects for each process class. We will thereby obtain different MSC execution traces that will represent various possible concrete instantiations of the abstract test case.The different instantiations will arise due to different possible object choices for executing various events. In such a concrete test case MSC, a symbolic lifeline s_p representing process class p in S, is

[14]The depth bound D is a user specified parameter.

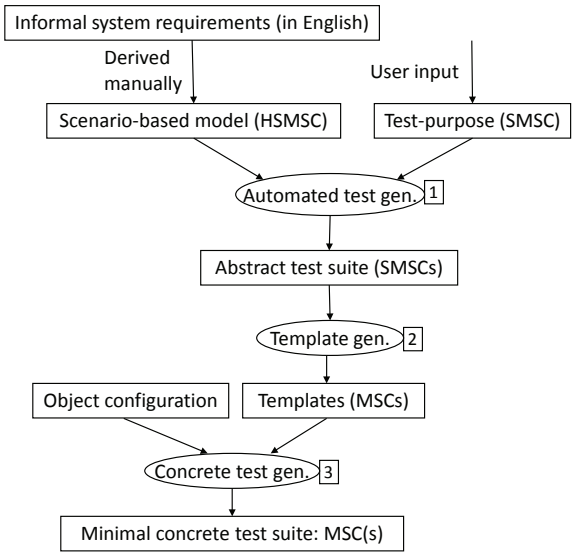


Fig. 7. Overall test generation flow.

replaced by a non-empty set of concrete lifeline(s) representing the p objects, and containing events from s_p. We capture various concrete instantiations of an abstract test case SMSC via the refinement relation $\sqsubseteq_C$, which is described in Appendix A.1. We then define a concrete test case as follows.

Definition 11 Concrete Test Case. *Let S be an abstract test case SMSC derived from a system model **Spec**. Then a concrete test case corresponding to S is an MSC M, such that, $S \sqsubseteq_C M$ (ref. Appendix A.1).*

However, the preceding approach of directly deriving concrete tests from the abstract test cases suffers from several major drawbacks.
First, it may be difficult to determine the minimum number of objects for various process classes that guarantees generation of at least one concrete test case from an abstract test case. Note that, a single object in a process class may not be able to execute all the events shown on the process class lifeline.
Second, many of the concrete test cases generated corresponding to an abstract test case may be redundant, differing only in the identities of objects playing various lifelines, but representing essentially the same behavior. For example, if we switch the lifeline names of the 3 memory devices shown in Fig. 1, we will get a new MSC scenario, which will however, depict the same core behavior (a memory device replying with the requested data).
Third, this process of deriving concrete test cases from abstract test cases has to be repeated each time there is a new object configuration, consisting of a different number of objects. This will make testing very inefficient, since it will involve repeated execution of the abstract test case, while maintaining the individual states of a potentially large number of objects.

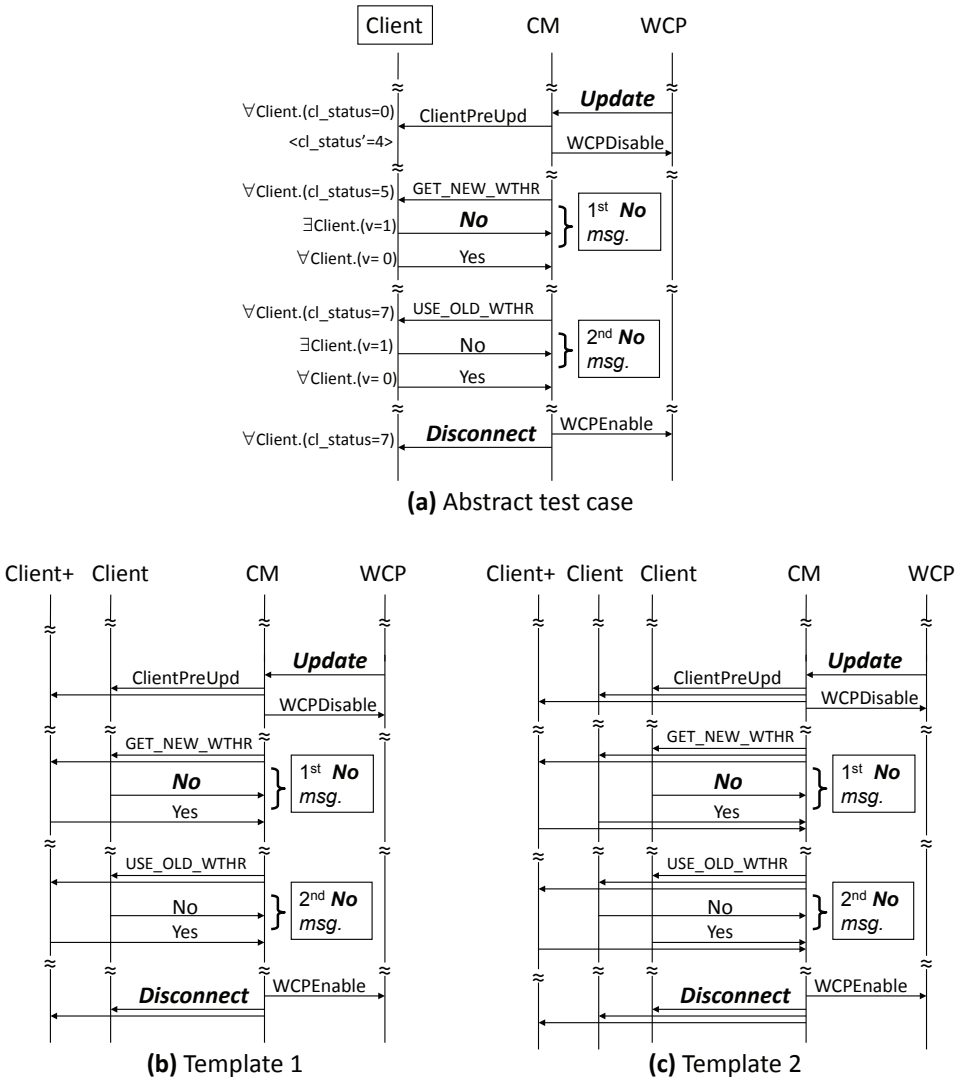


Fig. 8. An abstract test case (corresponding to test-purpose shown in Fig. 6) and two templates for it.

In order to avoid the above drawbacks, we introduce an intermediate representation between abstract and concrete test cases, called *templates* (oval 2, Fig. 7). A set of templates derived from an abstract test case represent behaviorally distinct realizations of the test case, without requiring object identities to be maintained. For example, let us consider the abstract test case in Figure 8(a). It is obtained from the CTAS model with an unbounded number of Client objects (i.e. $n_{Client} = \omega$) and, one object each in the CM and WCP classes (i.e. $n_{CM} = n_{WCP} = 1$). On the Client lifeline, the two *No* events with existential abstraction indicate the possibility of two distinct ways in which this abstract test case may be realized; in

one case, we can have the same concrete client object executing the two *No* events, while in the other, we may have two different client objects for the two events. The exact identities of these objects are irrelevant, what matters is whether the same or different concrete object(s) are selected, since from a testing perspective, they represent two different system behaviors. We capture these behaviors through *templates*.

The two templates corresponding to the abstract test case in Figure 8(a) are shown in Figures 8(b) and 8(c). Figure 8(b) depicts the case when a single Client object sends two *No* messages to the CM. This object is represented by the concrete lifeline labeled Client. In contrast, Figure 8(c) depicts the case when two different Client objects (both labeled Client) send the two *No* messages to the CM. In both these cases, the behavior of all other Client objects is represented by the *marked lifeline* (Client+); all of which will execute the events shown along the marked Client lifeline, and are behaviorally similar.

Templates offer a number of advantages in generating concrete test cases from abstract ones:

(1) The minimum number of objects required to derive concrete test cases from each template can be determined directly from the template itself. For each process class, this is the number of lifelines of that class (including the marked lifeline) present in the template. For example, the minimum number of Client objects required to generate a concrete test case from the template in Figure 8(b) is 2, and that from the template in Figure 8(c) is 3.

(2) Each template represents a behaviorally distinct realization of the corresponding abstract test case such that, a concrete test case derived from one template cannot be derived from another.

(3) Concrete test cases for different object configurations (differing in number of objects in certain classes) can be obtained directly from templates (which need to be generated only once). This involves simple instantiation of the templates with concrete objects, and involves no execution of behavior.

Intuitively, for an abstract test case SMSC S, a template T is described as an MSC, and captures a projection of events in top-down order from a symbolic lifeline s_p, representing process class p in S, to lifelines from process class p (say, C_p) in T. Further, an event e having existential abstraction mode from s_p is projected as an MSC event, $simple(e)$, along exactly one lifeline l in C_p, such that, e forms a *feasible sequence*[15] along with the events from s_p already projected along l. On the other hand, a universal event e from s_p is projected as an MSC event, $simple(e)$, along all possible lifelines (at least one) in C_p, such that for each lifeline l where e appears, it forms feasible sequence along with the events from s_p already projected along l. Finally, for a process class p, T has at most one lifeline (called *marked* lifeline) in C_p containing events with only universal abstraction mode from s_p projected along it. Note that, a marked lifeline from process class p in a template represents the

[15] We say that a sequence of SMSC events $\sigma = e_1 \dots e_n$, containing events from class p, is *feasible*, if an object (say, O) of class p can potentially execute the event sequence σ. That is, for each event e_i in σ, O satisfies the guard of e_i with its state updated with the effect of postconditions of events $e_1 \cdots e_{i-1}$, applied sequentially to its initial state.

behavior of *all* objects of class p other than those representing the concrete (or, *unmarked*) p-lifelines. Consequently, for each class p, a template has either one or zero marked lifeline, depending on whether or not the remaining p-objects (*i.e.* those not representing any concrete p-lifeline) participate in the given template by executing some common events.

We define a refinement relation $\sqsubseteq_T$, formally capturing the relationship between a template MSC and an abstract test case SMSC, in Appendix A.2. The notion of a template is defined as follows.

DEFINITION 12 TEMPLATE. *Let S be an abstract test case SMSC derived from a system model **Spec**. Then a template T corresponding to S is an MSC, such that, $S \sqsubseteq_T T$ (ref. Appendix A.2)*

We now state some key properties of our templates. For $n, m \in \mathbb{N}$ we define expression $\mathcal{Z}_{n,m}$ recursively as follows:

- $\mathcal{Z}_{n,1} = 1, \mathcal{Z}_{n,n} = 1$
- $\mathcal{Z}_{n,m} = m \times \mathcal{Z}_{n-1,m} + \mathcal{Z}_{n-1,m-1} \quad (n > m)$
- $\mathcal{Z}_{n,m} = 0 \quad (n < m)$

THEOREM 4. *Let S be an abstract test case SMSC and $\mathcal{T}$ be the set of templates corresponding to S. Let n_e^p denote the number of existential events from process class $p \in \mathcal{P}$ appearing in S. Then,*

(1) The number of lifelines in each template in $\mathcal{T}$ is finite.

(2) The number of templates in $\mathcal{T}$ is bounded by: $\prod_{p\in\mathcal{P}} \sum_{i=1}^{n_e^p} \mathcal{Z}_{n_e^p,i}$.

(3) The worst case time complexity for template generation is $O(\prod_{p\in\mathcal{P}}(\sum_{i=1}^{n_e^p} \sum_{j=1}^{i} \mathcal{Z}_{i,j}))$.

The proof of Theorem 4 appears in Appendix B.1.

6.1.3 *Deriving Concrete Tests from Templates.* The final step in our test generation process involves deriving concrete test case MSCs from various templates (oval 3, Fig. 7). It takes as input a user provided object configuration (defined in the following), specifying the number of objects in a process class p, if the number of objects in p is originally unbounded.

DEFINITION 13 OBJECT CONFIGURATION. *Let **Spec** $= \langle H, \bigcup_{p\in\mathcal{P}}\{V_p, v_p^{init}\}\rangle$ be a system model with $n_p \in \mathbb{N} \cup \{\omega\}$ objects in class p. An* **object configuration** *with respect to **Spec** is defined as $\bigcup_{p\in\mathcal{P}} O_p$, where O_p is a set of objects of class p in their initial state (determined by v_p^{init}) such that $|O_p| \in \mathbb{N} : |O_p| = n_p$, if $n_p \in \mathbb{N}$ and $|O_p| \geq 1$ otherwise (i.e. when $n_p = \omega$).*

Given an object configuration $\bigcup_{p\in\mathcal{P}} O_p$, a concrete test case MSC is derived from a template T by simply assigning concrete objects from O_p to lifelines corresponding to class p in the template. For each concrete lifeline l in template T involving p we assign one concrete p-object. Once the (unmarked) concrete p-lifelines have been assigned objects, all the *remaining* p-objects are assigned to the marked lifeline in p (replicating the marked lifeline and the events appearing along it). Recall that there can be at most one marked lifeline for a process class p. Note that, the number of objects in O_p should be equal to or greater than the total number lifelines

representing process class p in the given template in order to obtain concrete test case(s) from it.

For example, for an object configuration with three Client objects (and one object each of type CM and WCP) concrete test cases are obtained from the two CTAS templates shown in Figure 8. In the first template (Fig. 8(b)), one Client object is assigned to the concrete Client lifeline labeled 'Client' and, the remaining two objects are assigned to the marked lifeline labeled 'Client+'. In the second template (Fig. 8(c)), two objects are assigned to the two concrete lifelines labeled 'Client', and the only remaining Client object is assigned to the marked lifeline labeled 'Client+'.

The refinement of template MSCs into concrete test case MSCs is formally captured via the refinement relation $\sqsubseteq_{TC}$ (see Appendix A.3).

THEOREM 5. *Let S be an abstract test case SMSC, T be a template MSC, and M be a concrete test case MSC. Then, $S \sqsubseteq_T T \wedge T \sqsubseteq_{TC} M \iff S \sqsubseteq_C M$.*

The proof of Theorem 5 appears in Appendix B.2.

Next, we define the notion of *minimal concrete tests*, which can be derived from a set of templates corresponding to an abstract test case.

DEFINITION 14 MINIMAL CONCRETE TESTS. *Given an abstract test case SMSC S derived from system model **Spec**, and an object configuration OC for **Spec**, the set of* **minimal concrete tests** *consists of concrete test cases obtained by instantiating each template T derived from S exactly once (if for each process class p, the configuration OC has at least as many p-objects as the number of p-lifelines in T), or zero times (otherwise).*

The set of *all concrete test cases* can be generated from the minimal concrete tests by considering all possible object choices for various lifelines.

We deem two concrete test cases as *behaviorally distinct*, if one cannot be derived from another simply by switching object identities. Further, we call a set C of concrete test cases generated from an abstract test case S to be **optimal**, if all test cases in C are (pair-wise) behaviorally distinct and no new behaviorally distinct test case (with respect to C) can be generated from S.

THEOREM 6. *Given an abstract test case S, a set of minimal concrete tests derived from S is optimal.*

The proof of Theorem 6 appears in Appendix B.3.

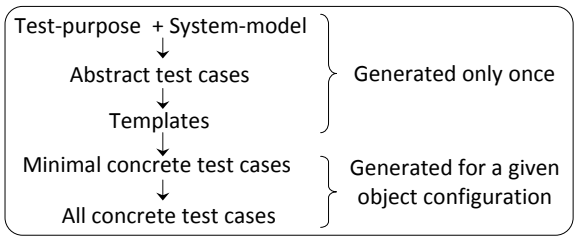


Fig. 9. Summary of our test generation flow.

From the discussion above, we can easily see that templates provide a useful mechanism for grouping together behaviorally similar concrete test cases corresponding to a given abstract test case.

6.2 Summary

To summarize, we recapture our overall test generation flow in Figure 9. While the left column outlines the steps involved in test case generation, the right column indicates if a step needs to be repeated to generate tests for different object configurations (see Def. 13). As can be easily seen, once templates have been generated for a given abstract test case, concrete test cases for different object configurations can be obtained directly from these templates without re-modeling or re-executing the system. This makes our approach highly *portable*, as evidenced by our experiments (see Section 7.4.2).

7. EXPERIMENTS

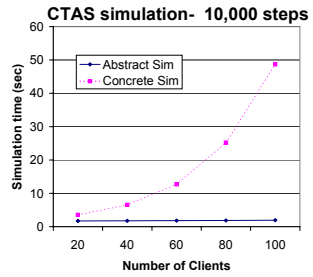
The operational semantics for SMSCs has been implemented as Prolog rules in the XSB logic programming system [XSB]. XSB supports *tabled* resolution for query evaluation. This speeds up execution by avoiding redundant computation. The operational semantics of SMSCs lend themselves naturally to Prolog 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. Further, our test generation framework, which utilizes SMSC operational semantics, is also encoded as Prolog rules in XSB.

In the following, we first briefly discuss another case study (called 'MOST'), followed by the experimental results obtained for the 'CTAS weather controller' example described earlier in Section 3, as well as the 'MOST' example.

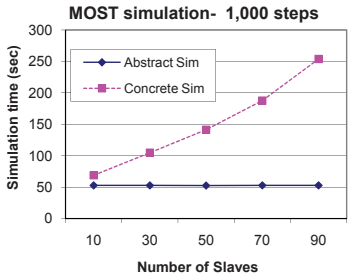
Case Study: Media Oriented Systems Transport (MOST)

The MOST (Media Oriented Systems Transport) [MOST] is a networking standard designed for interconnecting various classes of multimedia components in automobiles. It is maintained by the organization known as "MOST Cooperation", consisting of various automotive companies and component manufacturers like BMW, Daimler and Audi. The MOST network employs a ring topology, supporting easy plug and play for addition and removal of any new devices. It has been designed to suit applications that need to network multimedia information along with data and control functions.

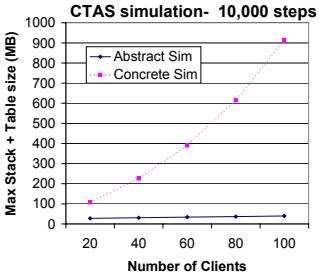
A node in a MOST network can be a multimedia device, such as a tuner, an amplifier, CD player etc. Each such device may provide a number of functionalities which can be directly available to user via human-machine interface, or for use by other devices in the network. For example, a CD player providing Play, Stop and Eject functions. Various specification documents for MOST are available under the **Publications** section at '`http://www.mostcooperation.com/`'. One of the specifications, namely the 'MOST Dynamic Specification', describes the dynamic behavior of a MOST network, encompassing: (a) Network Management, (b) Connection Management, and (c) Power Management. For our experiments, we



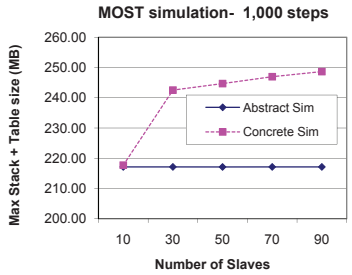
(a) CTAS Simulation: Time



(b) MOST Simulation: Time



(c) CTAS Simulation: Memory



(d) MOST Simulation: Memory

Fig. 10. Abstract vs Concrete Simulation Times for Different Settings of CTAS (a) and MOST (b). Peak Memory Requirement for Abstract and Concrete Simulation for CTAS (c) and MOST (d).

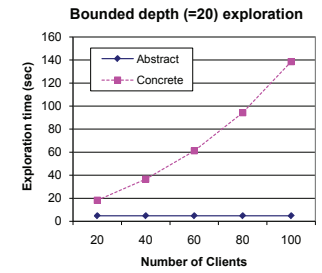
modeled only the Network Management part of the MOST protocol. The network management in MOST ensures secure communication between applications over the network by maintaining and providing most recent information about various nodes in the network.

From the network management perspective the system consists of (i) Network Master (NM), a specific node which maintains information about various device nodes in the network and their functionalities, and (ii) Network Slaves (NS), which are the remaining device nodes in the network. In our MOST system model, we have a **NM** class consisting of one concrete object representing the network master, and a **NS** class containing various network slave objects.

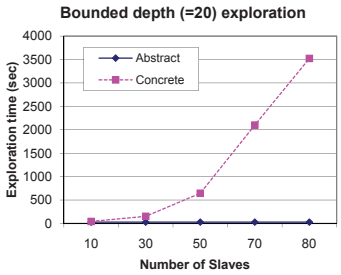
7.1 Modeling effort

We now describe in brief the modeling effort required for the CTAS and MOST examples from given requirements documents.

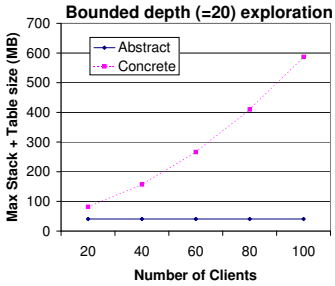
CTAS. The requirements [CTRD] are given as a set of scenarios from the perspective of the controller (or CM - communications manager), with preconditions given for the occurrence of each scenario. We were able to directly translate each scenario into corresponding SMSC. The high level control flow (HSMSC) was easily obtained by following the preconditions given for various scenarios.



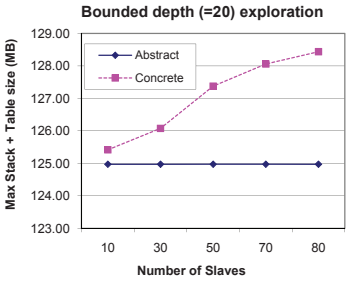
(a) CTAS Exploration: Time



(b) MOST Exploration: Time



(c) CTAS Exploration: Memory



(d) MOST Exploration: Memory

Fig. 11. Abstract vs Concrete State Space Exploration Times for CTAS (a) and MOST (b), Peak Memory Requirement for Abstract and Concrete State Space Exploration for CTAS (c) and MOST (d).

MOST. The requirements for MOST, as described in 'MOST Dynamic Specification' are given as high-level MSCs from the view-point of a particular process, describing -at a very high level- the control flow *within* a process. In addition, several "scenario MSCs" in system execution are provided. These "scenario MSCs" of the requirements document would correspond to a sequence of SMSCs in our modeling, that is, a scenario in system execution. Using these scenarios, we elaborated the high-level per-process control flow given by the "high-level MSCs" of the document and derived our HSMSC specification along with constituent SMSCs.

Some characteristic features of the CTAS and MOST HSMSCs are shown below.

	CTAS	MOST
# HSMSC nodes = # SMSCs	17	15
# HSMSC edges =	23	23
Total # Events	103	39
# Events with non-trivial reg. expr. guards	3	4
# Events with non-trivial propositional guards	65	20

7.2 Simulation

The graphs in Figure 10 compare the simulation time/memory for abstract vs concrete semantics for the CTAS and MOST examples. The use case used in the CTAS 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). In case of MOST, the simulation runs through the network initialization process,

followed by the network master (NM) requesting and successfully receiving device information from various slave nodes (NS). Different simulation runs correspond to different settings of the CTAS/MOST models with number of client/NS objects being varied (shown on the x-axis of various graphs in Figure 10). As we can easily observe, in case of CTAS (Figures 10 (a) & (c)), for abstract simulation the time/memory remains almost constant ($\approx$ 1.9 sec/40 MB) even as the number of client 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. Similar variations in time/memory can be observed for MOST simulations (Figures 10 (b) & (d)). 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 and manipulated individually.

7.3 State Space Exploration

We explored all possible traces up to a certain length, where each step in a trace is the execution of an SMSC event. The results appear in Figure 11, where the bound on the length of traces explored is set to 20 both for CTAS (Figures 11 (a) & (c)) and MOST (Figures 11 (b) & (d)). 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. *Moreover*, 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 a bounded number of client objects for abstract execution semantics (4.8 sec/41 MB). Again, similar time and memory usage patterns are observed in the case of MOST explorations. 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.

7.4 Test Generation and Execution

In this section, we report on three classes of experiments. The first deals with the performance of our test generation algorithms, the second explores the portability of our approach across different system configurations, and the third reports on the efficacy of the concrete test cases we generate in debugging system implementations.

7.4.1 *Test generation.* We consider the overall test generation flow as summarized in Figure 9. For both the CTAS and MOST examples, we derived five test-purposes each with the aim of covering their major use-cases. In case of CTAS, test-purposes (TP) 1 and 2 were designed to generate test cases for successful and unsuccessful connection requests respectively, from a Client object. The remaining three test-purposes (TP-3, 4, and 5) were used to elicit test cases corresponding to (un)successful weather updates of connected clients via CM. For instance, TP-5 (shown in Fig. 6) captures the scenario where at least one Client fails to either use the new weather update, or revert back, leading to all Clients getting disconnected. On the other hand, for MOST example, the first two test-purposes covered scenarios for successful and unsuccessful information receipt by network master (NM)

Table III. Symbolic Test Generation for CTAS and MOST with exploration depth set to 20 for CTAS and 25 for MOST. For generating the concrete tests, we consider 3 Clients for CTAS and 4 Network Slaves for MOST.

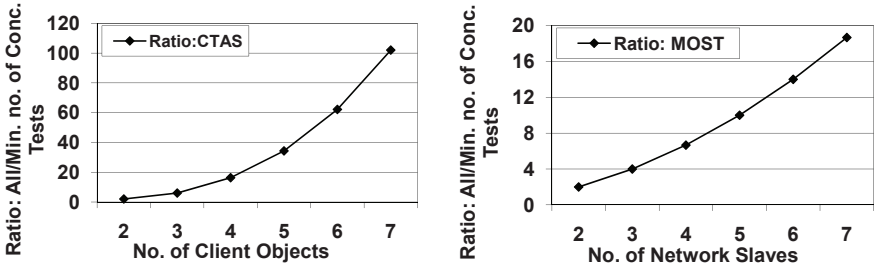
	TP #	1 Abst. test case	All Abst. test cases		# Templates	All Concrete Test Cases
		Time(s)	Time(s)	Total #		
CTAS	1	11.92	46	1	1	3
	2	0.02	33	2	2	6
	3	3.8	55	4	14	84
	4	3.8	58	8	14	84
	5	3.9	56	5	20	120
MOST	1	0.02	0.47	1	1	1
	2	0.1	0.16	4	4	16
	3	0.1	0.7	13	31	304
	4	0.06	0.2	6	9	60
	5	0.05	0.2	8	12	80

from all slaves (NS) during network initialization phase. The remaining three test-purposes for MOST were used to generate test-cases covering different scenarios where network master re-requests device information from various slaves in case some slave joins or leaves the network.

Abstract test generation. For both the examples, abstract test-generation results are shown in Table III (columns 3–5) for all five test-purposes. For each CTAS test-purpose, paths in the CTAS HSMSC were explored up to a depth of 20. While, in the case of MOST, exploration depth of 25 was used. Execution times are reported for generating either a *single* (Tab. III, col. 3), or *all* (Tab. III, col. 4–5) abstract test cases corresponding to a test-purpose.

Template and concrete test generation. As discussed in Section 6.1.2, from an abstract test case we first generate a set of templates. A set of minimal/all concrete test cases is then derived from these templates for a given object configuration (Section 6.1.3). In Table III (col. 6), we report the total number of templates generated for all abstract test cases for each test-purpose. For CTAS, concrete test cases were generated from these templates for an object configuration consisting of– *three* Client objects, *one* CM object and *one* WCP object. While in the case of MOST, object configuration consisted of *four* network slaves and *one* network master. The number of minimal concrete test cases obtained for these configuration is same as the number of templates (Tab. III, col. 6), since all the templates could be instantiated to a concrete test for the given configurations with three clients (for CTAS) and four network slaves (for MOST). We also show the total number of all possible concrete test cases for each test-purpose (Tab. III, col. 7).

By comparing columns 6 and 7 in Table III we see that for a given test-purpose the number of all concrete test cases generated can be significantly greater than the minimal number of test cases. Moreover, this gap increases as the number of objects in the system configuration is increased. This is because, the number of minimal test cases is bounded by the number of templates generated for a given test-purpose (see Def. 14). On the other hand, the number of all possible tests



(a) CTAS: Test-purpose 4 (b) MOST: Test-purpose 5

Fig. 12. Ratio of All/Minimal no. of concrete test cases for (a) CTAS: Test-purpose 4, and (b) MOST: Test-purpose 5.

Table IV. No. of nodes and edges in the (S)MSC-based models constructed for CTAS.

	HSMSC	HMSC (3 Clients)	HMSC (4 Clients)
# Nodes (# SMSCs/MSCs)	18	108	201
# Edges	24	157	285

keeps on increasing with the increasing number of objects. For illustration, ratio of the *all* to the *minimal* number of concrete test cases for CTAS test-purpose 4 (TP-4), with increasing number of clients, is plotted in Figure 12(a), and for MOST test-purpose 5 (TP-5), with increasing number of slaves, is plotted in Figure 12(b).

7.4.2 *Portability.* We now evaluate one of the key benefits of our approach — *portability*. By portability, we mean the relative ease of generating concrete tests for *different* object configurations, once our templates are generated.

For the purpose of these experiments we constructed two concrete models of the CTAS case-study using HMSCs. These two models differed in the number of Client objects– consisting of *three* and *four* Clients respectively . In the Table IV, we report the number of nodes and edges in our SMSC based CTAS system model and the two HMSC models constructed above.

At this point we note that, our SMSC based test generation approach already saves us considerable time and effort by avoiding re-modeling the system requirements for different object configurations. Moreover, we measure the time saved in terms of test generation for a given object configuration, due to our template-based approach (see Fig. 9). In Table V, we compare "the time taken to generate the minimal set of concrete tests (see Def. 14) from our SMSC based model" with "the time taken to generate the corresponding set of test cases from the HMSC models". Note that, for our SMSC based approach, we report the execution times for generating test cases directly from the templates for different object configurations (since templates need to be generated only once).

Only the results corresponding to test-purposes 1 and 2 are shown in Table V, since exploration did not even terminate (after running for 30 min.) for the other three test-purposes in the case of HMSC models. This is because of the blow-up in the number of paths in the concrete HMSC models.

Table V. Comparison of test generation times between a regular MSC-based and our SMSC-based approach.

	HSMSC	HMSC	
		3 Clients	4 Clients
TP1	0.033 s	38 s	148 s
TP2	0.12 s	35 s	144 s

7.4.3 *Test execution.* For the purpose of testing, the lifelines in a test-case MSC are divided into two categories– (a) those representing the components constituting the implementation under test (IUT), and (b) tester lifelines, representing the test environment for the IUT components. The *tester-components* are then generated from these tester lifelines to interact with, and test the IUT components. Further, a *master-tester* component is also generated for the purpose of giving the test verdicts.

The master tester gives the final test verdict based on the test verdicts received from various tester-components during test execution. Possible verdicts given by the master-tester are– (i) **pass**, if master-tester receives a pass verdict from *all* tester components, (ii) **fail**, if master-tester receives a fail verdict from any tester component, or (iii) **inconclusive**, if master-tester receives an inconclusive verdict from some tester components and no tester component sends a fail verdict. These verdicts are assigned in accordance with the formal conformance relation *ioco* [Tretmans 1996]. An implementation *ioco*-conforms to a specification (or, system model), if after the execution of an implementation trace allowed by the specification, the possible implementation outputs are those allowed by the specification. The absence of any output (*e.g.*, due to a deadlock) is also treated as an observable output.

Note that we distinguish between the IUT and the tester lifelines in a test case MSC at the time of testing an implementation. A test case MSC, in our setting, represents a particular execution scenario in our system model. It is possible that, for certain inputs sent by the tester to the IUT lifelines, there are non-deterministic responses from the IUT components, not necessarily captured by the given test case. Hence, for various inputs from the tester to the IUT components, we take into account such a possibility while assigning the test verdicts. More specifically, if the response of an IUT component differs from the expected output according to a test case, but is a possible output due to non-determinism in our system model, we assign inconclusive verdict to the test execution run in this case.

We performed experiments to evaluate the efficacy of our generated tests for debugging system implementations. In this set of experiments, we worked with two models of CTAS—a Statechart model was used to automatically generate a C++ implementation using the Rhapsody tool [Rhap] and a SMSC model was used to generate test cases which were tried on the C++ implementation. Note that, the Statechart model was derived separately, *by a person other than the authors.* In our experiments, we focused on testing the central controller (CM) component of CTAS C++ implementation. In Table VI, we present the key features of the CM's Statechart model.

To test the bug-detection capability of the test cases derived using our approach, we derived various buggy versions of CM's implementation by manually injecting bugs in CM's Statechart and generating code from that via Rhapsody tool. Three

Table VI. Key features of the CM's Statechart.

No. of States	No. of Trans.	No. of Guards	No. Unique events sent	No. Unique events recvd
25	34	3	13	6

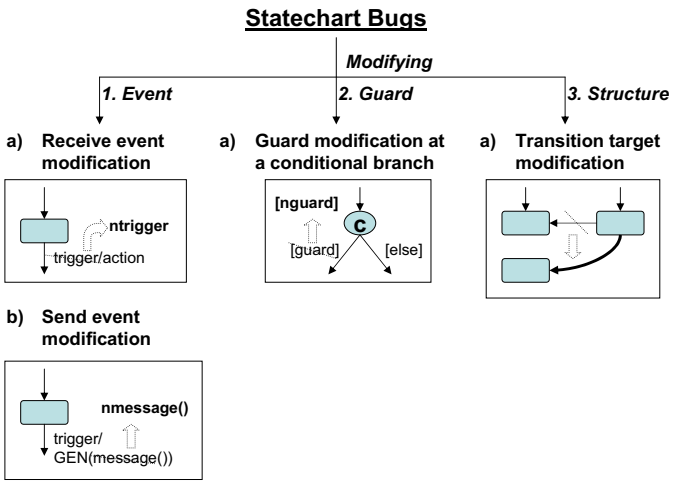


Fig. 13. Taxonomy of bugs introduced in Statechart models for the CTAS case study.

different category of bugs (see Table VII) were systematically introduced in the CM's Statechart for this purpose. The taxonomy of these bugs is shown in Figure 13. Majority of the bugs were introduced by modifying a transition label in the Statechart, which is of the form `trigger[guard]/action`. Here `trigger` represents the event whose reception triggers this transition, provided the `guard` evaluates to true. The `action` is a set of events (C++ code) executed if this transition is taken. All these three parts of a transition label are optional.

The first category of bugs involved modification of transition triggers, or of actions which involved sending event triggers. A total of 33 buggy versions were constructed in this category. The second category of bugs (6 in all) were constructed by modifying the guard of each outgoing transition individually from a condition node in the Statechart. To obtain the third category of bugs, the structure of the Statechart was modified. Specifically, the targets of various transitions were changed to point to another node in the Statechart, such that the original target node was still reachable from the initial Statechart node via an alternate path. A total of 19 buggy versions were thus constructed.

In order to test the above-mentioned buggy implementations, we used the five (5) test-purposes discussed earlier. These test-purposes correspond to the main use-cases of the CTAS example. For these five test-purposes, we derived seven (7) concrete test cases using our test generation method. Note that the generation of these seven concrete test cases involved (a) constructing an abstract test case SMSC for each test-purpose (b) deriving template test case MSCs from the abstract test case SMSCs, and (c) finally deriving concrete test case MSCs from the template

test MSCs by considering the objects in the concrete implementation being tested.

All the buggy CM implementations mentioned in the preceding were tested against our minimal set of concrete tests, derived using the *five* test-purposes discussed earlier. We deem to have *detected* a buggy implementation, if it fails at least one test from the set of minimal concrete test cases we generate. We summarize the results in Table VII.

Table VII. Use of our generated concrete tests for detecting bugs in C++ implementation

Bug category	Total # of buggy versions	# of buggy versions detected
1. Event	33	33
2. Guard	6	4
3. Structure	19	16

All bugs in the first category were detected. This is because our test-cases covered all the main use-cases in CTAS model, thereby covering all messages exchanged in the system at least once. Since bugs in the first category involved modifications relating to send or receive of various messages in the system, they were all detected. For the second category two buggy versions remained undetected. Recall that, this category of bugs involved modifying transition guards. For the two buggy versions which remained undetected, the guard modification caused them to ignore some inputs[16] sent by the tester components. However, their response to various tester components was still in accordance with the test cases. Thus, they *appeared* to behave correctly based on their outputs and passed all test cases. In the third category three cases were left undetected. For these three buggy versions none of our test cases exercised the CM's code beyond the point where bugs could be detected. CTAS being a reactive system involves non-terminating executions; detection of these bugs would require executing more than one test-cases in succession.

As we can observe, using our test generation methodology, we detected over 90% of the buggy implementations. Note that these implementations were derived from a Statechart model (*independently* constructed by a person other than the authors), different from the SMSC based model used for deriving the test cases.

Instead of trying out minimal number of concrete tests for the five test-purposes, if we had tried out all possible concrete tests—no more buggy implementations would have been detected. This is because the total set of concrete tests is simply obtained by switching the object identities of the minimal set of concrete tests (ref. Section 6.1.3), and do not test any more behavior than the minimal set of concrete tests. Thus, our strategy of computing/testing the minimal number of concrete tests for a given test-purpose can lead to *significant productivity enhancement* in testing.

[16]This is allowed by the Statechart semantics in the Rhapsody tool.

8. RELATED WORK

8.1 Modeling Notations

In our discussion, we focus on *visual* and *executable* modeling frameworks. Using *visual* notations provides a more intuitive basis for modeling and communication of initial system requirements among the designers and stakeholders. On the other hand, *executable* modeling notations easily lend themselves to various behavioral analyses (e.g. through simulation and model checking), as well as enabling automated model-based test generation. Various visual and executable modeling notations can be broadly classified into two main categories– *state-based* or *scenario-based*. In the following, we briefly discuss state-based notations, with a more detailed discussion on scenario-based notations.

8.1.1 *State-based Notations.* State-based notations are used for system behavior description either by means of a global state-machine, or by specifying the local control flow of various processes in the system (i.e. *intra-process behavior*), such that their composition determines the global system behavior. Some of the popularly used state-based notations in the design and analysis of distributed reactive systems are Statecharts [Harel 1987], Specification and Description Language (SDL) [ITU-T Z.100 1994], and Petri-nets [Reisig 1985]. A key advantage of using a state-based notation for system description is that it makes explicit the behavior of each process (or, a class in the object-oriented setting), enabling automated code generation from the model (e.g. [Niaz and Tanaka 2003; Rauchwerger et al. 2005; Mortensen 2000]).

8.1.2 *Scenario-based notations.* Complementary to the state-based notations, scenario-based notations are used for specifying global *inter-process* interaction scenarios among various system processes. In the case of distributed reactive systems, the requirements generally focus on specifying inter-process interactions while abstracting away from the local computations, and can therefore be more naturally modeled using scenario-based notations such as Message Sequence Charts (MSCs) [ITU-T Z.120 1999]. Examples of such requirements are often found in practice, for instance–

—Requirements document for Center-TRACON Automation System (or, CTAS) [CTRD]. CTAS is a control system aiding in management of arrival air-traffic at busy airports (discussed in Section 3).

—Media Oriented Systems Transport (or, MOST) [MOST], a multimedia and infotainment networking standard for the automotive industry. One of the specification documents, namely the 'MOST dynamic specification', contains the scenario based requirements for this standard.

The MSC language 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 decomposing 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 [Reniers 1999]. 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.

In recent years, a number of MSC variants have been proposed (*e.g.* [B.Genest et al. 2004; B.Sengupta and R.Cleaveland 2002; Damm and Harel 2001; Uchitel et al. 2002]). Of these, Triggered MSCs (TMSCs) and Live Sequence Charts (LSCs) are equipped with an execution semantics [B.Sengupta 2003; Harel and Marelly 2003]. However, TMSCs do not support symbolic specification, while LSCs support symbolic specification, but *not* symbolic execution. The symbolic lifelines are instantiated to concrete objects during execution in the LSC Play Engine [Harel and Marelly 2003]; 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. The approach taken in [Wang et al. 2004] maintains constraints on concrete process identities to alleviate this problem of LSCs. However, it falls short of fully symbolic execution (as in this paper where no process identities are maintained), and also requires additional annotations about process identities in the LSC specification.

8.1.3 *Hybrid Notations.* The work of [Goel et al. 2006] presents a hybrid notation called Interacting Process Classes (IPC), in which the behavior of a process class is specified by a labeled transition system and unit interactions between processes are described by MSCs. While, in case of IPC, the control flow of a process class is explicit (as specified by its labeled transition system), SMSCs provide a global view of system execution where local control flow of various classes may not be inferred. This brings the IPC model closer to state-based notations, rather than MSC-based notations. Further, the granularity of execution in IPC is at the level of transactions, whereas it is event based execution in case of SMSCs. A transaction in IPC may itself consist of multiple events. Another important difference between IPC and SMSC notations is that IPC model does not support universal abstraction of lifelines—only existential abstraction is supported. Therefore, while SMSCs allow specification of broadcast messages in a more direct and natural manner, the similar behavior cannot be captured exactly in our IPC framework.

8.2 Model-based Test Case Generation

Model based testing of reactive systems is a well studied and an active area of research [Broy et al. 2005]. Majority of these approaches use state-based notations, which are more suited for detailed and complete system descriptions. On the other hand, there are relatively few scenario based notations suited for complete behavior descriptions [Goel and Roychoudhury 2006]. Note that, our SMSC based testing methodology, as proposed in this article, falls in the latter category.

To the best of our knowledge, there are no existing works dealing with symbolic

test generation for systems with large number of behaviorally similar processes. Various approaches towards symbolic testing via data abstraction address a *different* problem—explosion in number of tests due to large number of data values. In the following, we briefly discuss various works in the domain model-based testing, categorized by the type of notations used for system modeling.

8.2.1 *State-based.* These approaches use some variant of state-based notations for modeling system behaviors. In [Pretschner et al. 2005], underlying system behavior is described using Extended Finite State Machines (or *EFSM*) — the test generation involves translating the system model into a Constraint Logic Programming (CLP) specification, adding in constraints corresponding to a test-purpose, and executing resulting CLP program. Specifications in AutoLink [Koch et al. 1998] are written in SDL [ITU-T Z.100], wherein a system is specified as a set of interconnected abstract machines which are extensions of finite state machines. On the other hand, a test-purpose in [Koch et al. 1998] is described using a MSC– the test generation proceeds by state space exploration of system model, while trying to satisfy the test-purpose. The notation of Labeled Transition Systems (or *LTS*) and its derivatives, such as IOTS (Input-Output LTS), are used for formally describing specifications, implementations, as well as test-purposes in [Tretmans and Brinksma 2002; Jard and Jéron 2005]. The test-generation is automated and driven by a conformance relation, which formally defines the notion of an implementation being correct with respect to a given specification. The state space explosion problem due to the presence of data variables is addressed in [Rusu et al. 2000; Frantzen et al. 2006]. They introduce symbolic versions of IOTS, called IOSTS (Input Output Symbolic Transition Systems), which include explicit notion of data and data-dependent control flow.

Statecharts [Harel 1987] and its UML variant are popular notations widely used for behavioral system description. As such, there is a large body of work dealing with test generation from them; here we mention a few of them. An initial work in this direction is [Offutt and Abdurazik 1999], involving test generation based on coverage criteria such as transition and predicate coverage. In [Hong et al. 2000] Statecharts are transformed into flow-graphs capturing both control and data flow in a Statechart. Tests are generated from resulting flow-graphs based on conventional data (control) flow analysis techniques. A complementary approach in [Hong et al. 2001] uses symbolic model checking for test generation from Statecharts. In TesTor [Pelliccione et al. 2005], test cases are generated from UML state diagrams by recovering missing information in partially specified test-purposes, represented using sequence diagrams. UML state machines are used for specifying a system model in [Pickin et al. 2007], and a test-purpose consists of a set of *Accept* and *Reject* scenarios, respectively specifying positive and negative criteria for generating test-cases. Interaction testing among classes modeled as state-machines, also taking into account the states of collaborating objects, is studied in [Gallagher et al. 2006; Ali et al. 2007].

8.2.2 *Scenario-based.* In TOTEM [Briand and Labiche 2001], test cases are derived from use cases, which are structured and detailed using UML activity and sequence diagrams. Sequence diagrams are used in SeDiTeC [Fraikin and Leon-

hardt 2002] and SCENTOR [Wittevrongel and Maurer 2001] for describing test specifications, which are extended with method parameters, and return values for method calls for actual testing. COWSuite [Basanieri et al. 2002] uses UML sequence diagrams for describing system use-cases. Corresponding to each use case, a set of test cases for testing that use case are then derived. The UBET tool [UBET 1999] supports test generation from a HMSC model, in which test generation is primarily driven by the *edge-coverage* criteria in a HMSC. In [Kugler et al. 2007], the play-engine tool for Live Sequence Charts (LSCs), which are an extension of MSCs, has been extended to support testing of scenario-based requirements.

8.2.3 *Combined notations.* A testing framework involving use of state and scenario based notations is presented in [Bertolino et al. 2005]. A multi-paradigmatic approach towards model based testing was proposed in [Grieskamp 2006], which, besides state/scenario based notations, also allows models to be described using other diagrammatic notations and/or program fragments.

8.2.4 *Symbolic Test Generation.* A number of model-based testing approaches support symbolic test generation from system specifications involving data variables [Rusu et al. 2000; Pretschner 2001; Frantzen et al. 2006]. All these approaches use state-based description of system behaviors and involve abstraction over data-domains as a means for avoiding state-explosion during test generation for data-intensive systems. However, various processes in a system are still represented in a concrete manner as individual entities. Hence these approaches do not scale as the number of processes (say of a class) is increased. One *cannot* directly use these approaches for symbolic test generation of systems with large number of behaviorally similar processes.

9. DISCUSSION

In this article, we have proposed Symbolic Message Sequence Charts (SMSCs) as a framework for scenario-based modeling, simulation, and testing of process classes, the systems consisting of many behaviorally similar objects. SMSCs are a lightweight syntactic and semantic extension to conventional MSCs. In case of systems consisting of process classes, similar MSC specifications become too voluminous for human comprehension. We discussed process theory based concrete and abstract operational semantics for SMSCs. While, in concrete execution semantics, the execution states of various objects are maintained individually, objects are dynamically grouped together into equivalence classes at runtime in our abstract execution semantics. We also presented a model-based test generation methodology for process classes based on SMSCs. The key aspect of our approach is the automated generation of abstract test cases from a system model, followed by generating test case templates from them. A minimal set of behaviorally distinct concrete test cases can then be derived directly from these templates for various system configurations. The efficacy of our approach was validated through a detailed case study and experiments involving a non-trivial weather controller specification.

Finally, we observe that there are several extensions of SMSCs which add substantial modeling power but involve minimal changes in the execution semantics. One 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. Yet another kind of requirement might be "some objects (an unknown number) will play a certain role". If the total number of objects is bounded, our execution semantics requires minimal modification to handle such requirements. Another extension can be to consider SMSCs with "associations" where objects of different classes are statically or dynamically related. For example, in a telephone network consisting of thousands of phones and switches, communication links are established between the caller and the called phones via the intermediate switches for the duration of the call; this constitutes a dynamic association between phones. In future, we will investigate these and other requirement templates involving similar processes and integrate/support them systematically in our framework.

Acknowledgments. This work was partially supported by an IBM Faculty Award and NUS Faculty Research Council grants R252-000-416-112, R-252-000-385-112. Liang Guo constructed the Statechart model of the CTAS case study.

APPENDIX

A. REFINEMENT RELATIONS

Let **Spec** be a SMSC system model with process class $p \in \mathcal{P}$ containing $n_p \in \mathbb{N} \cup \{\omega\}$ objects, and $OC = \cup_{p \in \mathcal{P}} O_p$ be an object configuration with respect to **Spec** (see Def. 13). We now list some notations used in our discussion further.

- BEH_p – set of behavioral partitions of process class p (ref. Def. 5).
- $\mathcal{C}$ – set of abstract configurations for **Spec** (ref. Section 5.4).
- $\mathcal{C}_C$ – set of concrete configurations for **Spec** and OC (ref. Section 5.6).
- $L_p(M)$ – set of lifelines from process class p appearing in an MSC M.
- $\mathcal{CT}(\Sigma_{\mathcal{S}})$ and $\mathcal{CT}(\Sigma_{\mathcal{M}})$ – sets of closed terms describing SMSCs and MSCs, respectively (ref. Section 5.1).
- A^{MSC} – set of events appearing in MSCs.
- $\rightarrow_M \subseteq \mathcal{CT}(\Sigma_{\mathcal{M}}) \times A^{MSC} \times \mathcal{CT}(\Sigma_{\mathcal{M}})$ – transition relation describing the MSC operational semantics as defined in Reniers [1999].
- A_p^{SMSC} – set of SMSC events executed by process class p.

A.1 Refinement Relation $\sqsubseteq_C$

In this section, we define a refinement relation $\sqsubseteq_C$ between SMSCs and MSCs to relate the concrete test case MSCs corresponding to an abstract test case SMSC.

Definition 15 Refinement Relation $\sqsubseteq_C$. *Let S be an abstract test case SMSC derived from **Spec** and M be an MSC, such that, $s^{init} \in \mathcal{CT}(\Sigma_{\mathcal{S}})$ and $t^{init} \in \mathcal{CT}(\Sigma_{\mathcal{M}})$ denote the process terms capturing S and M, respectively. Further, let $c^{init} \in \mathcal{C}_C$ denote the initial concrete configuration. Then, we say that $S \sqsubseteq_C M$, if and only if the following hold–*

(1) For a process class p, there exists a mapping $id_p : L_p(M) \rightarrow O_p$ such that, $\forall p_1, p_2 \in L_p(M) \cdot id_p(p_1) \neq id_p(p_2)$.

(*2*) *For a process class p there exists a mapping* $proc_p : A_p^{SMSC} \rightarrow 2^{L_p(M)}$ *such that:*

(*a*) $\forall e \in A_p^{SMSC}$, $|proc_p(e)| = 1$ *if* $mode(e) = \exists$; $|proc_p(e)| \geq 1$ *otherwise*[17].

(*b*) *Let* $PR = \{pr | \forall e \in A_p^{SMSC} \cdot pr \in proc_p(e) \implies mode(e) = \forall\}$, *and for a lifeline* $l \in L_p(M)$ *let* $events_p(l) = \{e | e \in A_p^{SMSC}, l \in proc_p(e)\}$. *Then, for all* $pr, pr' \in PR \cdot events_p(pr) = events_p(pr')$.

(*3*) *There exists a binary relation* $\sim_C \subseteq (\mathcal{C}_C \times \mathcal{CT}(\Sigma_\mathcal{S})) \times \mathcal{CT}(\Sigma_\mathcal{M})$, *satisfying that*

(*a*) $(c^{init}, s^{init}) \sim_C t^{init}$.

(*b*) *If* $(c, s) \sim_C t$, *then* $s \downarrow \Leftrightarrow t \downarrow$ *(i.e. both s and t should terminate together).*

(*c*) *Let* $(c, s) \sim_C t$, *and for a local action event* $e = action(os, l, pc)$, $c : s \xrightarrow{e} c' : s'$ *(as per concrete SMSC execution semantics, ref. Section 5.2), such that* $os = \forall\, p.G$ *and e is executed by all p objects, say* $PR = \{o_1, \ldots, o_N\} \subseteq O_p$, *satisfying the guard G of e. Then,*

—$proc_p(e) = \{p_1, p_2, \ldots, p_N\}$ *such that,* $id_p(p_i) = o_i$.

—$\exists t_1, \ldots, t_N \in \mathcal{CT}(\Sigma_\mathcal{M})$, $t \xrightarrow{action(p_1,l)}_M t_1 \cdots t_{(N-1)} \xrightarrow{action(p_N,l)}_M t_N$.

—$(c', s') \sim_C t_N$.

(*d*) *Let* $(c, s) \sim_C t$, *and for a send event*[18] $e = out(os_1, os_2, f, pc)$, $c : s \xrightarrow{e} c' : s'$, *such that,* $os = \forall\, p.G$, *e is executed by all p processes, say* $PR = \{o_1, \ldots, o_N\} \subseteq O_p$, *satisfying the guard G of e. Further, let* e' *be the matching receive event executed by process class q corresponding to event e in S, such that,* $proc_q(e') = \{q_1, \ldots, q_K\}$. *Then,*

—$proc_p(e) = \{p_1, p_2, \ldots, p_N\}$ *such that,* $id_p(p_i) = o_i$.

—$\exists t_1, t_2, \ldots, t_{N.K} \in \mathcal{CT}(\Sigma_\mathcal{M})$, *such that,* $t \xrightarrow{out(p_1,q_1,f)}_M t_1 \cdots t_{(K-1)} \xrightarrow{out(p_1,q_K,f)}_M t_K \xrightarrow{out(p_2,q_1,f)}_M t_{(K+1)} \cdots t_{(N.K-1)} \xrightarrow{out(p_N,q_K,f)}_M t_{N.K}$.

—$(s', c') \sim_C t_{N.K}$.

Note that, items (c) and (d) above can be easily modified to handle the case where event e has an existential abstraction mode (i.e. $os = \exists\, p.G$*) — instead of multiple p processes, only a single p process will execute event e in this case.*

A.2 Refinement Relation $\sqsubseteq_T$

In this section, we define a refinement relation $\sqsubseteq_T$ between MSCs describing templates and abstract test case SMSCs. We use a function $state_p : L_p(M) \rightarrow BEH_p$ to associate an execution state with each lifeline in $L_p(M)$. Let $state = \{state_p\}_{p \in \mathcal{P}}$, and $\mathcal{ST}$ denote the set of all such functions. Further, we use $st^{init} \in \mathcal{ST}$ to denote the initial mapping, such that, for a process class p, for all lifelines in $L_p(M)$, $state_p \in st^{init}$ maps them to the initial behavioral partition in BEH_p representing the initial state of p processes.

Definition 16 Refinement Relation $\sqsubseteq_T$. *Let S be an abstract test case SMSC derived from* ***Spec*** *and M be an MSC, such that,* $s^{init} \in \mathcal{CT}(\Sigma_\mathcal{S})$ *and* $t^{init} \in \mathcal{CT}(\Sigma_\mathcal{M})$ *denote the process terms capturing S and M, respectively. Further, let* $c^{init} \in \mathcal{C}$ *denote the initial abstract configuration. Then, we say that* $S \sqsubseteq_T M$, *if and only if the following hold–*

[17] We use $mode(e)$ as a shorthand for $mode(active(e))$.

[18] A receive event is treated in the similar manner.

(*1*) *For a process class p there exists a mapping* $nproc_p : A_p^{SMSC} \rightarrow 2^{L_p(M)}$ *such that:*

(*a*) $\forall e \in A_p^{SMSC}$, $|nproc_p(e)| = 1$ *if* $mode(e) = \exists$; $|nproc_p(e)| \geq 1$ *otherwise.*

(*b*) *for a process class p with an unbounded number of objects (i.e.* $n_p = \omega$*), there is at most one lifeline in* $L_p(M)$ *such that, only SMSC events having the universal abstraction mode are mapped to it via* $nproc_p$*, that is,*

$$|\{pr | \forall e \in A_p^{SMSC} \cdot pr \in nproc_p(e) \implies mode(e) = \forall\}| = 1.$$

We call such a p lifeline marked lifeline*, and refer to all other p lifelines as* concrete*.*

(*2*) *There exists a binary relation* $\sim_T \subseteq (\mathcal{C} \times \mathcal{CT}(\Sigma_{\mathcal{S}})) \times (\mathcal{ST} \times \mathcal{CT}(\Sigma_{\mathcal{M}}))$*, satisfying that*

(*a*) $(c^{init}, s^{init}) \sim_T (st^{init}, t^{init})$.

(*b*) *If* $(c, s) \sim_C (st, t)$*, then* $s \downarrow \Leftrightarrow t \downarrow$ *(i.e. both s and t should terminate together).*

(*c*) *Let* $(c, s) \sim_T (st, t)$*, where* $st = \{state_p\}_{p \in \mathcal{P}}$*, and for a local action event* $e = action(os, l, pc)$*,* $c : s \xrightarrow{e} c' : s'$ *(as per abstract SMSC execution semantics, ref. Section 5.4), such that* $os = \forall\, p.G$ *and e is executed by all p processes from the behavioral partitions in* $B = \{b_1, \ldots, b_m\}$*, satisfying the guard G of e. Then,*

—*there exists a partition of the set* $nproc_p(e) = \{p_1, p_2, \ldots, p_N\}$*, containing m subsets, say* $Pt_1, \ldots, Pt_m$*, such that (i)* $count_p(b_i) = |Pt_i|$*, if* $count_p(b_i) \in \mathbb{N}$*, and (ii) for* $1 \leq i \leq m$*,* $\forall pr \in Pt_i, state_p(pr) = b_i$.

—$\nexists pr \in L_p(M) \backslash nproc_p(e) \cdot state_p(pr)$ *satisfies the guard G of event e.*

—$\exists t_1, \ldots, t_N \in \mathcal{CT}(\Sigma_{\mathcal{M}})$*,* $t \xrightarrow{action(p_1, l)}_M t_1 \cdots t_{(N-1)} \xrightarrow{action(p_N, l)}_M t_N$.

—*for* $1 \leq i \leq m, \forall pr \in Pt_i$*,* $state'_p(pr)$ *is set to to* $dest(b_i, e)$ *after the execution of* $action(pr, l)$.

—$(c', s') \sim_T (st', t_N)$*, where* $st' = \{state'_p\}_{p \in \mathcal{P}}$.

(*d*) *Let* $(c, s) \sim_T (st, t)$*, where* $st = \{state_p\}_{p \in \mathcal{P}}$*, and for a send event*[19] $e = out(os_1, os_2, f, pc)$*,* $c : s \xrightarrow{e} c' : s'$*, such that,* $os = \forall\, p.G$*, e is executed by all p processes from the behavioral partitions in* $B = \{b_1, \ldots, b_m\}$ *satisfying the guard G of e. Further, let* e' *be the matching receive event corresponding to event e, executed by class q, with* $proc_q(e') = \{q_1, \ldots, q_K\}$*. Then,*

—*there exists a partition of the set* $nproc_p(e) = \{p_1, \ldots, p_N\}$*, containing m subsets, say* $Pt_1, \ldots, Pt_m$*, such that (i)* $count_p(b_i) = |Pt_i|$*, if* $count_p(b_i) \in \mathbb{N}$*, and (ii) for* $1 \leq i \leq m$*,* $\forall pr \in Pt_i, state_p(pr) = b_i$.

—$\nexists pr \in L_p(M) \backslash nproc_p(e) \cdot state_p(pr)$ *satisfies the guard G of event e.*

—$\exists t_1, t_2, \ldots, t_{N.K} \in \mathcal{CT}(\Sigma_{\mathcal{M}})$*, such that,* $t \xrightarrow{out(p_1, q_1, f)}_M t_1 \cdots t_{(K-1)} \xrightarrow{out(p_1, q_K, f)}_M t_K \xrightarrow{out(p_2, q_1, f)}_M t_{(K+1)} \cdots t_{(N.K-1)} \xrightarrow{out(p_N, q_K, f)}_M t_{N.K}$.

—*for* $1 \leq i \leq m, \forall pr \in Pt_i$*,* $state'_p(pr)$ *is set to* $dest(b_i, e)$ *after the execution of* $out(pr, q_K, f)$.

—$(s', c') \sim_T (st', t_{N.K})$*, where* $st' = \{state'_p\}_{p \in \mathcal{P}}$.

[19] A receive event is treated in the similar manner.

Note that, items (c) and (d) above can be easily modified to handle the case where event e has an existential abstraction mode (i.e. os = ∃p.G) — instead of multiple p processes, only a single p process will execute event e in this case.

A.3 Refinement Relation $\sqsubseteq_{TC}$

We define relation $\sqsubseteq_{TC}$ to formally capture the refinement of template MSCs in to concrete test case MSCs. Let T be a template MSC derived from **Spec** and M be a concrete test case MSC, such that, $|L_p(T)| \leq |L_p(M)| \leq |O_p|$.

DEFINITION 17 REFINEMENT RELATION $\sqsubseteq_{TC}$. *Let T be a template MSC and M be a concrete test case MSC, with t^{init} and $m^{init} \in \mathcal{CT}(\Sigma_{\mathcal{M}})$ denoting the process terms capturing T and M, respectively. Then, we say that $T \sqsubseteq_{TC} M$, if and only if the following hold–*

(1) For a process class p, there exists a mapping $nid_p : L_p(M) \rightarrow O_p$, such that, $\forall p_1, p_2 \in L_p(M) \cdot nid_p(p_1) \neq nid_p(p_2)$.

(2) For a process class p, there exists a mapping $F_p : L_p(T) \rightarrow 2^{L_p(M)}$ such that:
- *(a) $\forall p_1, p_2 \in L_p(T) \cdot F_p(p_1) \cap F_p(p_2) = \emptyset$.*
- *(b) $\cup_{pr \in L_p(T)} F_p(pr) = L_p(M)$.*
- *(c) $\forall pr \in L_p(T)$, such that pr is a concrete lifeline, $|F_p(pr)| = 1$.*
- *(d) If $\exists pr \in L_p(T)$, such that pr is a marked lifeline, then $|F_p(pr)| \geq 1$.*

(3) There exists a binary relation $\sim_{TC} \subseteq \mathcal{CT}(\Sigma_{\mathcal{M}}) \times \mathcal{CT}(\Sigma_{\mathcal{M}})$, satisfying that
- *(a) $s^{init} \sim_{TC} m^{init}$.*
- *(b) If $t \sim_{TC} m$, then $t \downarrow \Leftrightarrow m \downarrow$ (i.e. both t and m should terminate together).*
- *(c) If $t \sim_{TC} m$, and for a local action event $e = action(pr, l)$, $t \xrightarrow{e}_M t'$, such that $pr \in L_p(T)$ and $F_p(pr) = \{p_1, \ldots, p_N\}$ — then there exist $m_1, \ldots, m_N \in \mathcal{CT}(\Sigma_{\mathcal{M}})$ such that $m \xrightarrow{e_1}_M m_1 \cdots m_{N-1} \xrightarrow{e_N}_M m_N$, where $e_i = action(p_i, l)$ and $t' \sim_{TC} m_N$.*
- *(d) If $t \sim_{TC} m$, and for a send event[20] $e = out(pr, pr', f)$, $t \xrightarrow{e}_M t'$, such that $pr \in L_p(T)$, $pr' \in L_q(T)$, $F_p(pr) = \{p_1, \ldots, p_{N_1}\}$ and $F_q(pr') = \{q_1, \ldots, q_{N_2}\}$ — then there exist $m_1, \ldots, m_{(N_1.N_2)} \in \mathcal{CT}(\Sigma_{\mathcal{M}})$ such that $m \xrightarrow{out(p_1,q_1,f)}_M m_1 \cdots m_{(N_2-1)} \xrightarrow{out(p_1,q_{N_2},f)}_M m_{N_2} \xrightarrow{out(p_2,q_1,f)}_M m_{(N_2+1)} \cdots m_{(N_1.N_2-1)} \xrightarrow{out(p_{N_1},q_{N_2},f)}_M m_{N_1.N_2}$, and $t' \sim_{TC} m_{(N_1+N_2)}$.*

B. THEOREMS' PROOFS

B.1 Proof of Theorem 4

PROOF. Let n_e^p be the total number of existential events from process class p appearing in the abstract test case SMSC S. From the definition of templates (Def. 12), each template $T \in \mathcal{T}$ is an MSC such that, $S \sqsubseteq_T T$, where $\sqsubseteq_T$ relation is defined in Appendix A.2.

(1) From the description of refinement relation $\sqsubseteq_T$ in Appendix A.2, we know that each existential event in the abstract test case SMSC S appears along exactly one unmarked (or, concrete) lifeline in a template This follows from

[20] A receive event is considered in the similar manner.

the definition of mapping $nproc_p$ (point (1), Def. 16 in Appendix A.2), and the definition of simulation relation $\sim_T$ (point (2), Def. 16 in Appendix A.2). On the other hand, from the definition of the simulation relation $\sim_T$, we can determine that– corresponding to the execution of an existential event e from SMSC S, only one lifeline (determined by the mapping $nproc_p$) from a template corresponding to S will participate in executing the MSC version of the event e. Hence, the maximum number of concrete lifelines from process class p in a template, that correspond to some existential event in S (via the mapping $nproc_p$), is bounded by the number of existential p-events in S, i.e. n_e^p. Further, we know that there can be at most one marked lifeline from class p in a template. This again follows from the definition of the mapping $nproc_p$ (point (1) in Def. 16). Finally, from the description of simulation relation $\sim_T$ (point (2), Def. 16 in Appendix A.2), we know that there can be no more lifelines in a template other than those mapped to by the function $nproc_p$ corresponding to the events appearing in S (otherwise, condition (b) of point (2) in Def. 16 will be violated). Hence, the number of p-lifelines in a template lie between 1 to $1+n_e^p$.

(2) From (1) above, the existential events from a process class p can appear along $k \in [1, n_e^p]$ lifelines in a template corresponding to the abstract test case S. Further, from the definition of the $\sqsubseteq_T$ relation (Def. 16, point (2), $\sim_T$ relation), we know that a universal p-event e appears along all lifelines (at least one p-lifeline) in a template, if permitted by the p's control flow, as determined by the satisfaction of e' guard by various p processes. Therefore, to determine an upper bound on the number of templates for a given abstract test case, it is sufficient to determine for each process class p all possible ways in which existential p-events can appear along $k \in [1, n_e^p]$ lifelines. For a given k and p we define the quantity U_p^k, which determines the number of ways in which n_e^p distinct objects can be distributed into k identical boxes such that each box contains at least one object. Value of U_p^k is determined by the expression $\mathcal{Z}_{n_e^p,k}$ defined as follows [Joshi 1989]:

$$\begin{array}{ll} (1)\ \mathcal{Z}_{n,1} = 1, \mathcal{Z}_{n,n} = 1 & \\ (2)\ \mathcal{Z}_{n,m} = m \times \mathcal{Z}_{n-1,m} + \mathcal{Z}_{n-1,m-1} & (n > m) \\ (3)\ \mathcal{Z}_{n,m} = 0 & (n < m) \end{array}$$

Since k varies from 1 to n_e^p, there are a maximum of $U_p = \sum_{i=1}^{n_e^p} U_p^i$ ways in which existential p-events can appear along concrete lifelines in a template. Hence, the total number templates is bounded by $\prod_{p \in \mathcal{P}} U_p$ where $\mathcal{P}$ is the set of process classes appearing in the abstract test case S.

(3) The worst case time complexity for template generation is $O(\prod_{p \in \mathcal{P}} (\sum_{i=1}^{n_e^p} \sum_{j=1}^{i} \mathcal{Z}_{i,j}))$. We prove this using induction on n_e, the number of existential events appearing in the abstract test case S, i.e. $n_e = \sum_{p \in \mathcal{P}} n_e^p$. The base case $n_e = 0$ holds trivially. Next, we assume that induction hypothesis holds for $n_e = \sum_{p \in \mathcal{P}} n_e^p$ number of events. From (2) above we know that the maximum possible number of templates at this stage is $\prod_{p \in \mathcal{P}} \sum_{i=1}^{n_e^p} \mathcal{Z}_{n_e^p,i}$. Next, for process class p and $k \in [1, n_e^p]$, we determine the maximum number of templates having k or $k+1$ lifelines from class p ($k+1$ lifelines if a marked

p lifeline is also present) as follows:

$$t_k^p = (\prod_{q \in \mathcal{P} \setminus \{p\}} \sum_{i=1}^{n_e^q} \mathcal{Z}_{n_e^q,i}).\mathcal{Z}_{n_e^p,k}$$

We now consider execution of another existential event e_p from process class p. For $k \in [1, n_e^p]$, in worst case event e_p can be executed by all the $k+1$ lifelines from process class p in each of the t_k^p templates. Thus, event e_p can cause a maximum number of $\sum_{k=1}^{n_e^p} t_k^p.(k+1)$ executions. So, the worst case time complexity for $n_e + 1$ existential events will be:

$$\begin{aligned}
&\prod_{p \in \mathcal{P}} (\sum_{i=1}^{n_e^p} \sum_{j=1}^{i} \mathcal{Z}_{i,j}) + \sum_{k=1}^{n_e^p} t_k^p.(k+1) \\
\equiv\ &\prod_{p \in \mathcal{P}} (\sum_{i=1}^{n_e^p} \sum_{j=1}^{i} \mathcal{Z}_{i,j}) + (\prod_{q \in \mathcal{P} \setminus \{p\}} \sum_{i=1}^{n_e^q} \mathcal{Z}_{n_e^q,i}).(\sum_{k=1}^{n_e^p} \mathcal{Z}_{n_e^p,k}.(k+1)) \\
&\text{(replacing } t_k^p \text{ from above and taking } \sum_{k=1}^{n_e^p} \text{ inside)} \\
\equiv\ &\prod_{p \in \mathcal{P}} (\sum_{i=1}^{n_e^p} \sum_{j=1}^{i} \mathcal{Z}_{i,j}) + (\prod_{q \in \mathcal{P} \setminus \{p\}} \sum_{i=1}^{n_e^q} \mathcal{Z}_{n_e^q,i}). \sum_{k=1}^{n_e^p+1} \mathcal{Z}_{n_e^p+1,k} \\
&\text{(from the definition of } \mathcal{Z}) \\
\leq\ &\prod_{p \in \mathcal{P}} (\sum_{i=1}^{n_e^p} \sum_{j=1}^{i} \mathcal{Z}_{i,j}) + (\prod_{q \in \mathcal{P} \setminus \{p\}} \sum_{i=1}^{n_e^q} \sum_{j=1}^{i} \mathcal{Z}_{i,j}). \sum_{k=1}^{n_e^p+1} \mathcal{Z}_{n_e^p+1,k} \\
\leq\ &\prod_{q \in \mathcal{P} \setminus \{p\}} \sum_{i=1}^{n_e^q} \sum_{j=1}^{i} \mathcal{Z}_{i,j}.(\sum_{i=1}^{n_e^p+1} \sum_{j=1}^{i} \mathcal{Z}_{i,j})
\end{aligned}$$

Hence, for $n_e + 1$ existential events the worst case time complexity of template generation is $O(\prod_{q \in \mathcal{P} \setminus \{p\}} \sum_{i=1}^{n_e^q} \sum_{j=1}^{i} \mathcal{Z}_{i,j}.(\sum_{i=1}^{n_e^p+1} \sum_{j=1}^{i} \mathcal{Z}_{i,j}))$.

□

B.2 Proof of Theorem 5

Proof.

1. $\underline{S \sqsubseteq_T T \wedge T \sqsubseteq_{TC} M \Longrightarrow S \sqsubseteq_C M}$

Let S be an abstract test case SMSC derived from system model **Spec**, T be a corresponding template MSC, and M be a concrete test case MSC derived from the template T for an object configuration $OC = \cup_{p \in \mathcal{P}} O_p$ corresponding to **Spec** (ref. Section 6.1.2), that is, $S \sqsubseteq_T T$ and $T \sqsubseteq_{TC} M$. To show that $S \sqsubseteq_C M$, we need to show that the conditions 1, 2 and 3, describing the $\sqsubseteq_C$ relation hold (ref. Appendix A.1). Since, $S \sqsubseteq_T T$, there exists a mapping $nproc_p : A_p^{SMSC} \rightarrow 2^{L_p(T)}$

satisfying[21] condition (1) of Definition 16. Similarly, since $T \sqsubseteq_{TC} M$, there exists (i) a mapping $nid_p : L_p(M) \rightarrow O_p$, satisfying condition (1) of Definition 17 (p. 49), and (ii) a mapping $F_p : L_p(T) \rightarrow 2^{L_p(M)}$, satisfying condition (2) of Definition 17. We now consider the three conditions of relation $\sqsubseteq_C$ in the following.

(1) A mapping $id_p = nid_p$ is defined corresponding to condition 1 of $S \sqsubseteq_C M$ relation (Definition 15).

(2) A mapping $proc_p : A_p^{SMSC} \rightarrow 2^{L_p(M)}$ is defined corresponding to condition 2 of $S \sqsubseteq_C M$ relation (Definition 15) as follows: $\forall e \in A_p^{SMSC} \cdot proc_p(e) = \cup_{pr \in nproc_p(e)} F_p(pr)$.
If event e has an existential abstraction mode, i.e. $mode(e) = \exists$, then $nproc_p(e)$ will be a singleton set, say $\{pr'\}$, representing a concrete lifeline from $L_p(T)$ (condition (1), Definition 16, p. 47). Furthermore, from the definition of F_p (condition (2), Definition 17, p. 49), $F_p(pr')$ will also be a singleton set, i.e. $|proc_p(e)| = 1$ when e is an existential event. Similarly, it can be easily shown that, for a universal event e', $|proc_p(e')| \geq 1$. Thus, condition 2-(a) for relation $S \sqsubseteq_C M$ holds.
From condition 1-(b) of relation $\sqsubseteq_T$ (Defn. 16), the template MSC T has at most one lifeline (the *marked* lifeline) from process class p, say t_p, such that, only events with the universal abstraction mode are mapped onto it via the mapping $nproc_p$. Let $PR = F_p(t_p)$ and $E_p = \{e | e \in A_p^{SMSC}, t_p \in nproc_p(e)\}$. Then, from the definition of $proc_p$, along all lifelines in $L_p(M)$ as determined by PR, only events with the universal abstraction mode from E_p will appear. Further, for all remaining lifelines $L_p(M) \backslash F_p(t_p)$, some existential event from A_p^{SMSC} is mapped onto them via $proc_p$. Thus, condition 2-(b) of relation $\sqsubseteq_C$ holds.

(3) We define a relation $\mathcal{R} \subseteq \mathcal{ST} \times \mathcal{C}_C$ between the state functions, defined corresponding to $\sqsubseteq_T$ relation (ref. Appendix A.2), and concrete configurations as follows. For $st = \{state_p\}_{p \in \mathcal{P}}$ and $c = \{obj_p\}_{p \in \mathcal{P}} \in \mathcal{C}_C$, we say that $st \mathcal{R}\, c$ if and only if for all $p \in \mathcal{P}$, for all behavioral partitions $b \in BEH_p$,

$$l \in L_p(T) \wedge state_p(l) = b \wedge p_i \in F_p(l) \wedge o = nid_p(p_i) \implies obj_p(o) = b.$$

We now construct the simulation relation $\sim_C \subseteq (\mathcal{C}_C \times \mathcal{CT}(\Sigma_{\mathcal{S}})) \times \mathcal{CT}(\Sigma_{\mathcal{M}})$, corresponding to condition 3 for relation $S \sqsubseteq_C M$. Let $s^{init} \in \mathcal{CT}(\Sigma_{\mathcal{S}})$ denote the process term capturing SMSC S, $t^{init} \in \mathcal{CT}(\Sigma_{\mathcal{M}})$ denote the process term capturing template MSC T and $u^{init} \in \mathcal{CT}(\Sigma_{\mathcal{M}})$ denote the process term capturing the concrete test case MSC M. Since, $S \sqsubseteq_T T$, there exists a simulation relation $\sim_T$ satisfying condition 2 of Definition 16. Similarly, since $T \sqsubseteq_{TC} M$, there exists a simulation relation $\sim_{TC}$ satisfying condition 3 of Definition 17. Assume that $(c^{init}, s^{init}) \sim_C u^{init}$. Let $(c_a, s) \sim_T (st, t)$ and $t \sim_{TC} u$, such that $s \downarrow$, $t \downarrow$, and $u \downarrow$. Then, it is easy to see that $(c, s) \sim_C u$, where $c \in \mathcal{C}_C$ is a concrete SMSC configuration. Thus, conditions 3-(a) and 3-(b) of Definition 15 hold. Next, let $(c_a, s) \sim_T (st, t)$ such that, for a local action event

[21] The description of various notations, such as A_p^{SMSC}, $L_p(T)$ etc., appears at the beginning of Appendix A.

$e = action(os, l, pc) \in A_p^{SMSC}$, $c_a : s \xrightarrow{e} c'_a : s'$, where event e has the universal abstraction mode. Further, let c be a concrete configuration such that $st\mathcal{R}\, c$. Assume that objects from behavioral partitions $\{b_1, \ldots, b_m\}$ are chosen to execute the event e. Then, from condition 2-(c) of $\sqsubseteq_T$ (Definition 16), N_1 lifelines from process class p, say p_1 to p_{N_1}, in template T are partitioned into m subsets $Pt_1 \ldots Pt_m$ such that, (i) $count_p(b_i) = |Pt_i|$, if $count_p(b_i) \in \mathbb{N}$, and (ii) for $1 \leq i \leq m$, $\forall pr \in Pt_i, state_p(pr) = b_i$. These$N_1$ p-objects will be chosen to execute the local action event in T corresponding to event e (with process p_i executing the event $action(p_i, l)$). Also, let $t \sim_{TC} u$. Then, from condition 3-(c) of $\sqsubseteq_{TC}$ (Definition 17), lifelines from process class p in the concrete test case MSC M, determined by $\bigcup_{i\in[1,N_1]} F_p(p_i) = \{pr_1, \ldots, pr_{N_2}\}$, will execute the corresponding local action event (process pr_i executing the event $action(pr_i, l)$). Let these N_2 lifelines in M correspond to the p objects $o_1, \ldots, o_{N_2}$ as determined by the mapping nid_p. Then, from the definition of relation $\mathcal{R}$, we know that in the concrete configuration c, there are N_2 objects o_1 to o_{N_2}, each of whose states is determined by one of the behavioral partitions in b_1 to b_m. Then, all these objects can be chosen to execute the event e in the concrete execution of S, i.e. there exists a concrete configuration c' such that, $c : s \xrightarrow{e} c' : s'$. Thus, condition 3-(c) of relation $\sqsubseteq_C$ (Definition 15) holds.
The condition 3-(d) of relation $\sqsubseteq_C$ in Definition 15 can be shown to hold in a similar manner. Hence, $S \sqsubseteq_T T \wedge T \sqsubseteq_{TC} M \Longrightarrow S \sqsubseteq_C M$

2. $S \sqsubseteq_C M \Longrightarrow \exists$ an MSC T s.t. $S \sqsubseteq_T T \wedge T \sqsubseteq_{TC} M$

Let S be an abstract test case SMSC derived from system model **Spec**, and M be a concrete test case MSC derived from S for an object configuration $OC = \cup_{p\in\mathcal{P}} O_p$ corresponding to **Spec**, such that, $S \sqsubseteq_C M$. We now show that there exists a template MSC T that can be derived from S, such that, the concrete test case MSC M can be derived from T for object configuration OC, that is, $S \sqsubseteq_T T$ and $T \sqsubseteq_{TC} M$. Since, $S \sqsubseteq_C M$, there exists (i) a mapping $id_p : L_p(M) \rightarrow O_p$, satisfying condition (1) of Definition 15 (p. 46), and (ii) a mapping $proc_p : A_p^{SMSC} \rightarrow 2^{L_p(M)}$ satisfying condition (2) of Definition 15. Then, we define the following.

(1) We construct a template MSC T from the concrete test case MSC M as follows. For a process class p with a finite number of processes in the abstract setting, we set $L_p(T) = L_p(M)$. On the other hand, for a process class p with an unbounded number of processes in the abstract setting, let $PR_p \subseteq L_p(M)$ be the set of lifelines as defined in the condition 2-(b) of $\sqsubseteq_C$ relation (Definition 15). Let $m_p \in PR_p$ be one of the lifelines in PR_p. Then, we set $L_p(T) = L_p(M) \backslash PR_p \cup \{m_p\}$. Finally, we add all the events from M to T corresponding to the lifelines from M appearing in T.

(2) A mapping $nproc_p : A_p^{SMSC} \rightarrow 2^{L_p(T)}$ corresponding to condition 1 of $S \sqsubseteq_T T$ relation (Definition 16) as follows: $\forall e \in A_p^{SMSC} \cdot nproc_p(e) = proc_p(e)$, if the number of objects in process class p is finite in the abstract setting, i.e. $n_p \in \mathbb{N}$. Otherwise, for a process class p with an unbounded number of processes (i.e., $n_p = \omega$):

$$\forall e \in A_p^{SMSC} \cdot nproc_p(e) = \begin{cases} proc_p(e) \backslash PR_p \cup \{m_p\}, & \text{if } PR_p \subseteq proc_p(e) \\ proc_p(e), & \text{otherwise.} \end{cases}$$

Note that, for a process class p with unbounded number of processes, a single lifeline $m_p \in PR_p$, if included, corresponds to the marked p lifeline.

(3) A mapping $nid_p = id_p$ corresponding to condition 1 of $T \sqsubseteq_{TC} M$ relation (Definition 16), where id_p is as defined in condition 1 of $T \sqsubseteq_{TC} M$ relation (Definition 15).

(4) A mapping $F_p : L_p(T) \rightarrow 2^{L_p(M)}$ corresponding to condition 2 of $T \sqsubseteq_{TC} M$ relation (Definition 16), such that

$$F_p(pr) = \begin{cases} \{pr\}, & \text{if } pr \neq m_p \\ PR_p, & \text{otherwise,} \end{cases}$$

where the set PR_p is as defined in point (1). It can be easily verified that F_p as defined in the preceding, satisfies various constraints specified in condition 2 of $\sqsubseteq_{TC}$ relation.

(5) We use the relation $\approx \subseteq \mathcal{C} \times \mathcal{C}_C$, to relate the abstract and the concrete configurations as follows. For an abstract configuration $c_a = \{count_p\}_{p \in \mathcal{P}} \in \mathcal{C}$ and a concrete configuration $c = \{obj_p\}_{p \in \mathcal{P}} \in \mathcal{C}_C$, we say that $c_a \approx c$ if and only if, $\forall p \in \mathcal{P}, b \in BEH_p \cdot\ count_p(b) \in \mathbb{N} \implies |\{o \mid o \in O_p, obj_p(o) = b\}| = count_p(b)$.
Let $\mathcal{L}$ denote the set of all MSC lifelines. We define a relation $\mathcal{R}_1 \subseteq \mathcal{CT}(\Sigma_{\mathcal{M}}) \times 2^{\mathcal{L}} \times \mathcal{CT}(\Sigma_{\mathcal{M}})$, between process terms representing two MSCs and a subset of MSC lifelines, as follows. We say that $(u_1, l, u_2) \in \mathcal{R}_1$ if and only if u_1 is a process expression representing an MSC M_1 and u_2 is a process expression representing an MSC M_2, with M_1 containing lifelines and events from M_2 except for the lifelines from l appearing in M_2. Note that, $(t_{init}, L', u_{init}) \in \mathcal{R}_1$, where (a) t_{init} and u_{init} are the process terms capturing template MSC T and concrete test case MSC M, respectively, and (b) $L' = \cup_{p \in \mathcal{P}}(PR_p \backslash \{m_p\})$.
Then, based on the simulation relation $\sim_C$ defined in condition 3 of relation $\sqsubseteq_C$ (Definition 15), we define another relation $\mathcal{R}_2 \subseteq (\mathcal{C} \times \mathcal{CT}(\Sigma_{\mathcal{S}})) \times (\mathcal{ST} \times \mathcal{CT}(\Sigma_{\mathcal{M}}))$. Let $(c, s) \sim_C u$, where $c = \{obj_p\}_{p \in \mathcal{P}} \in \mathcal{C}_C$ is a concrete configuration. For class p, we define the function $state_p : L_p(T) \rightarrow BEH_p$ as follows. For a lifeline $l \in L_p(T)$, $state_p(l) = b$ if $pr \in F_p(l) \wedge o = id_p(pr) \wedge obj_p(o) = b$. Then, given an abstract configuration c_a, such that $(c_a, c) \in \approx$, we say that $(c_a, s)\mathcal{R}_2(st, t)$, where $st = \{state_p\}_{p \in \mathcal{P}}$, if there exists a t such that, $(t, L', u) \in \mathcal{R}_1$.
Finally, we derive simulation relation $\sim_{TC}$, corresponding condition 3 of relation $\sqsubseteq_{TC}$, from $\mathcal{R}_1$ as follows: $(t, u) \in \sim_{TC} \Leftrightarrow (t, \_l, u) \in \mathcal{R}_1$. Further, using relation $\mathcal{R}_2$ as simulation relation $\sim_T$ corresponding to condition 2 of relation $\sqsubseteq_T$, it is easy to see that $S \sqsubseteq_T T$ and $T \sqsubseteq_{TC} M$ hold.

□

B.3 Proof of Theorem 6

PROOF.
Let $\mathcal{T}$ be the set of templates derived from S. From the definition of templates (Def. 12, p. 31), a template $T \in \mathcal{T}$ will be related to S via the refinement relation $\sqsubseteq_T$, i.e. $S \sqsubseteq_T T$. Then, from the definition of relation $\sqsubseteq_T$, we can see that the two templates corresponding to S will differ (modulo the lifeline identities) if and only

Table VIII. Grammar describing Basic SMSCs

⟨smsc_document⟩	::=	**smsc_document** ⟨smsc_document_id⟩**;** ⟨process_info⟩ ⟨smsc_document_body⟩
⟨process_info⟩	::=	**process_info;** ⟨process_list⟩ **end;** | ⟨⟩
⟨process_list⟩	::=	⟨process⟩ | ⟨process⟩ ⟨process_list⟩
⟨process⟩	::=	**proc** ⟨process_id⟩ **:** ⟨count⟩ **;** ⟨var_init_list⟩
⟨count⟩	::=	**inf** | ⟨INT⟩
⟨var_init_list⟩	::=	⟨var_id⟩ **:=** ⟨INT⟩ **;** ⟨var_init_list⟩ | ⟨var_id⟩ **[** ⟨INT⟩ **]** **:=** ⟨INT⟩ **;** ⟨var_init_list⟩ | ⟨⟩
⟨smsc_document_body⟩	::=	⟨smsc⟩ ⟨smsc_document_body⟩ | ⟨⟩
⟨smsc⟩	::=	**smsc** ⟨smsc_id⟩ **;** ⟨smsc_body⟩ **endsmsc;**
⟨smsc_body⟩	::=	⟨event_definition⟩ ⟨smsc_body⟩ | ⟨⟩
⟨event_definition⟩	::=	⟨instance_name⟩ **:** ⟨instance_event⟩ ⟨post_condition⟩ **;**
⟨instance_name⟩	::=	⟨mode⟩ ⟨instance_id⟩ ⟨guard⟩
⟨mode⟩	::=	∀ | ∃ | ⟨⟩
⟨guard⟩	::=	**(** ⟨bool_expr⟩ **,** ⟨reg_expr⟩ **)** | **(** ⟨bool_expr⟩ **)** | **(** ⟨reg_expr⟩ **)** | ⟨⟩
⟨instance_event⟩	::=	⟨action⟩ | ⟨message_event⟩
⟨action⟩	::=	**action(** ⟨action_id⟩ **)**
⟨message_event⟩	::=	⟨message_output⟩ | ⟨message_input⟩
⟨message_output⟩	::=	**out** ⟨message_id⟩ **to** ⟨instance_name⟩
⟨message_input⟩	::=	**in** ⟨message_id⟩ **from** ⟨instance_name⟩
⟨post_condition⟩	::=	**{** ⟨assignment_list⟩ **}** | ⟨⟩
⟨assignment_list⟩	::=	⟨lvalue⟩ **:=** ⟨prim_expr⟩ **;** ⟨assignment_list⟩ | ⟨lvalue⟩ **:=** ⟨prim_expr⟩ **;**
⟨lvalue⟩	::=	⟨var_id⟩ | ⟨var_id⟩ **[** ⟨INT⟩ **]** | ⟨var_id⟩ **[** ⟨index_var_id⟩ **]**

if the function $proc_p$ differs for the two templates. Now, if the function $proc_p$ is different, clearly events from the symbolic p lifeline in S are not mapped identically to the p lifelines in the two templates (determined by $L_p(T)$ for a template T). Thus, a concrete test derived from one template cannot be derived from another. Hence, the minimal concrete tests (see Def. 14, p. 32) derived from S are all behaviorally distinct.

Next, we know that for a given object configuration (see Def. 13, p. 31) the concrete tests derived from a template will only differ in object identities of various lifelines, and hence are not behaviorally distinct. Thus, if the set of minimal concrete tests is not optimal, this implies there exists another template corresponding to S not present in $\mathcal{T}$, which is a contradiction. □

C. TRANSLATING HSMSC SPECIFICATIONS TO PROCESS TERMS

In this section give a brief outline of the translation of a HSMSC specification into a process term. The translation process follows a similar approach as described in [Reniers 1999]. To recall, the signature Σ over which we describe our process terms consists of a set of constants and a set of operators as follows. The constants consist of (i) the empty process ϵ, (ii) deadlock δ, and (iii) atomic actions from the set A^{SMSC} (see Definition 1). The set of operators we consider consist of unary operators – iteration $\star$ and unbounded repetition ∞, and the binary operators – delayed choice $\mp$, weak sequential composition $\circ$ and its generalized version $\circ^S$. The set of *closed* Σ *terms* is denoted by $\mathcal{CT}(\Sigma)$.

C.1 Translating a SMSC to a process term

The grammar describing a ⟨smsc_document⟩, comprising a finite set of smsc descriptions is given in Table VIII. The definition of ⟨smsc_document⟩ consists of a ⟨process_info⟩ part and ⟨smsc_document_body⟩. The ⟨process_info⟩ contains information about various process classes in the system specification, giving the number of objects in various process classes, as well as initial valuation of their vari-

ables. It is used for obtaining an initial system configuration. On the other hand, ⟨smsc_document_body⟩ specifies various SMSCs in a system description. Various grammar elements of the form ⟨X_id⟩ (e.g. ⟨smsc_id⟩, ⟨var_is⟩ etc.) represent *identifiers* and are of *string* type). In the following, we define a family of mappings $\mathbb{T}$ which translate a SMSC described using ⟨smsc⟩ in Table VIII into a process term over the signature described in the preceding.

First, we define a mapping $\mathbb{M}$ that associates with a process term the set of message exchange actions appearing in it. Let $A_{out} = \bigcup_{p\in\mathcal{P}} A^p_{out}$, $A_{in} = \bigcup_{p\in\mathcal{P}} A^p_{in}$ and $A_{msg} = A_{out} \cup A_{in}$, where A^p_{out} and A^p_{in} are message output and input actions from process class p (see Definition 1).

DEFINITION 18. *Let* $\otimes \in \{\circ^S, \mp | S \subseteq A_{msg} \times \mathbb{N} \times A_{msg}\}$ *and* $\odot \in \{\star, \infty\}$. *The mapping* $\mathbb{M} : \mathcal{CT}(\Sigma) \to \mathbb{P}(A_{msg})$ *for* $x, y \in \mathcal{CT}(\Sigma)$ *and* $a \in A_{msg}$ *is defined as follows:*

$$\begin{aligned}
\mathbb{M}(\epsilon) &= \emptyset,\\
\mathbb{M}(\delta) &= \emptyset,\\
\mathbb{M}(a) &= \begin{cases} \{a\} & \text{if } a \in A_{msg},\\ \emptyset & \text{otherwise,}\end{cases}\\
\mathbb{M}(x \otimes y) &= \mathbb{M}(x) \cup \mathbb{M}(y),\\
\mathbb{M}(x^{\odot}) &= \mathbb{M}(x).
\end{aligned}$$

Next, we define a mapping $MsgOrd$ to obtain the ordering requirements when two SMSC fragments are composed to ensure that a message output precedes the corresponding input.

DEFINITION 19. *For* $x, y \in \mathcal{CT}(\Sigma)$, $os \in OS^p, os' \in OS^q, m \in \mathcal{M}, pc \in Post^p$, $pc' \in Post^q$ *the mapping* $MsgOrd : \mathcal{CT}(\Sigma) \times \mathcal{CT}(\Sigma) \to \mathbb{P}(A_{out} \times A_{in})$ *is defined as follows:*

$$\begin{aligned}
MsgOrd(x,y) &= \{o \overset{0}{\mapsto} i | o = \mathit{out}(os, os', m, pc) \in \mathbb{M}(x), i = \mathit{in}(os, os', m, pc') \in \mathbb{M}(y)\}\\
&\cup \{o \overset{0}{\mapsto} i | o = \mathit{out}(os, os', m, pc) \in \mathbb{M}(y), i = \mathit{in}(os, os', m, pc') \in \mathbb{M}(x)\}
\end{aligned}$$

The ordering requirements for two actions $a, b \in A^{SMSC}$ are of the form $a \overset{n}{\mapsto} b$, where $n \geq 0$. Here, counter n is used to keep track of the difference in the number of times that a and b have been executed already. Each time a executes, n is incremented by 1, while n is decremented by 1 on execution of b. When $n = 0$, only a can execute and not b. This ensures that occurrences of a are always greater than those of b.

Next, we define the vertical composition for SMSCs using $\bullet$ operator, which takes into account the message ordering requirements between the SMSC fragments being composed.

DEFINITION 20. *For all* $x, y \in \mathcal{CT}(\Sigma)$, *the operator* $\bullet : \mathcal{CT}(\Sigma) \times \mathcal{CT}(\Sigma) \to \mathcal{CT}(\Sigma)$ *is defined as follows:*

$$x \bullet y = x \circ^{MsgOrd(x,y)} y$$

Let$\mathcal{L}(\langle \mathrm{X} \rangle)$ denote the language of the grammar element ⟨X⟩.

DEFINITION 21. *For all* smsc $\in \mathcal{L}(\langle smsc \rangle)$, smsc_body $\in \mathcal{L}(\langle smsc\_body \rangle)$ *and* eventdef $\in \mathcal{L}(\langle event\_definition \rangle)$, *the mapping* $\mathbb{T} : \mathcal{L}(\langle smsc \rangle) \to \mathcal{CT}(\Sigma)$ *is defined*

Table IX. Grammar describing HSMSCs

⟨hsmsc⟩	::=	**hsmsc** ⟨hsmsc_id⟩**; expr** ⟨smsc_expression⟩ **endhsmsc**
⟨smsc_expression⟩	::=	⟨start⟩ ⟨node_expression_list⟩
⟨start⟩	::=	⟨label_name_list⟩ **;**
⟨label_name_list⟩	::=	⟨label_name⟩ **alt** ⟨label_name_list⟩ | ⟨label_name⟩
⟨node_expression_list⟩	::=	⟨node_expression⟩ ⟨node_expression_list⟩ | ⟨⟩
⟨node_expression⟩	::=	⟨label_name⟩ **:** ⟨node_list⟩
⟨node_list⟩	::=	⟨smsc_id⟩ **seq (** ⟨label_name_list⟩ **);** | **end ;**

inductively by:

$$\begin{aligned} \mathbb{T}(\text{smsc}) &= \mathbb{T}(\text{smsc\_body}) \\ \mathbb{T}(\text{eventdef smsc\_body}) &= \mathbb{T}(\text{eventdef}) \bullet \mathbb{T}(\text{smsc\_body}) \\ \mathbb{T}(\langle\rangle) &= \epsilon \end{aligned}$$

The translation of an event definition *eventdef* $\in$ $\mathcal{L}(\langle\text{event\_definition}\rangle)$, i.e. $\mathbb{T}$(eventdef), is straightforward and results in an atomic action in A^{SMSC} corresponding to *eventdef*. We omit the translation details here. Finally, for a *smsc* $\in \mathcal{L}(\langle\text{smsc}\rangle)$, *smscid* $\in \mathcal{L}(\langle\text{smsc\_id}\rangle)$ and *smsc_body* $\in \mathcal{L}(\langle\text{smsc\_body}\rangle)$, such that:

$$smsc \equiv \textbf{smsc}\ smscid;\ smsc\_body\ \textbf{endsmsc};$$

we define a constant $\overline{smscid} = \mathbb{T}(smsc)$ to identify the process term associated with a *smsc* having id *smscid*.

C.2 Translating a HSMSC to a process term

The textual description of a HSMSC is as per the grammar rules described in Table IX, starting from ⟨hsmsc⟩. Each node in the HSMSC is identified using a label except for the *start* node. The start node is described as a list of labels of its successor nodes. All other nodes are described by stating the corresponding label name, the SMSC name represented by that node, and a list of labels of the successor nodes of the node being described.

For obtaining a process term corresponding to a HSMSC, the HSMSC is treated as an edge-labeled graph, and transformed into another HSMSC with a very specific structure. As an edge labeled graph, the nodes (edges) of the graph correspond to the nodes (edges) of the HSMSC. The start node in the HSMSC is not present in the graph; instead the successor nodes of the start node in the HSMSC appear as the initial nodes in the graph. The end nodes of the HSMSC are represented in the corresponding graph by the final nodes. Each node in the graph is uniquely identified by the corresponding node label in the HSMSC. The label of an edge in the graph representing a HSMSC between the nodes labeled with l_1 and l_2 is specified as the process term corresponding to the SMSC represented by l_1.

DEFINITION 22. *An edge labeled graph is a quadruple* (V, E, I, F) *where*

—V *is a finite set of nodes,*
—$E \subseteq V \times \mathcal{CT}(\Sigma) \times V$ *is finite set of labeled edges,*
—$I \subseteq V$ *is a set of initial nodes,*
—$F \subseteq V$ *is a set of final nodes.*

We now define the mappings *Nodes*, *Edges*, *Initial* and *Final*, which are used for obtaining an edge labeled graph corresponding to a textual HSMSC description. Let $\mathcal{V} = \mathcal{L}(\langle \text{label\_name} \rangle)$ denote the set of label names.

DEFINITION 23. **(Successor nodes)**
For $\text{l} \in \mathcal{V}$ *and* $\text{labels} \in \mathcal{L}(\langle \textit{label\_name\_list} \rangle)$*, the mapping* $\text{Succ} : \mathcal{L}(\langle \textit{label\_name\_list} \rangle) \rightarrow \mathbb{P}(\mathcal{V})$ *is defined inductively by*

$$\begin{aligned} \text{Succ(l)} &= \{\text{l}\}, \\ \text{Succ(l labels)} &= \{\text{l}\} \cup \text{Succ(labels)}. \end{aligned}$$

DEFINITION 24. **(Graph Nodes)**
For $\text{start} \in \mathcal{L}(\langle \textit{start} \rangle)$, $\text{nodexprs} \in \mathcal{L}(\langle \textit{node\_expression\_list} \rangle)$, *the mapping* $\text{Nodes} : \mathcal{L}(\langle \textit{smsc\_expression} \rangle) \rightarrow \mathbb{P}(\mathcal{V})$ *is defined by*

$$\text{Nodes(start nodexprs)} = \text{Nodes(nodexprs)}.$$

For $\text{nodexpr} \in \mathcal{L}(\langle \textit{node\_expression} \rangle)$, $\text{nodexprs} \in \mathcal{L}(\langle \textit{node\_expression\_list} \rangle)$, *the mapping* $\text{Nodes} : \mathcal{L}(\langle \textit{node\_expression\_list} \rangle) \rightarrow \mathbb{P}(\mathcal{V})$ *is defined inductively by*

$$\begin{aligned} \text{Nodes()} &= \emptyset \\ \text{Nodes(nodeexpr nodexprs)} &= \text{Nodes(nodexpr)} \cup \text{Nodes(nodexprs)}. \end{aligned}$$

For $\text{l} \in \mathcal{V}$ *and* $\text{nl} \in \mathcal{L}(\langle \textit{node\_list} \rangle)$*, the mapping* $\text{Nodes} : \mathcal{L}(\langle \textit{node\_expression} \rangle) \rightarrow \mathbb{P}(\mathcal{V})$ *is defined by*

$$\text{Nodes(l : nl)} = \{l\}.$$

DEFINITION 25. **(Graph Edges)**
For $\text{start} \in \mathcal{L}(\langle \textit{start} \rangle)$, $\text{nodexprs} \in \mathcal{L}(\langle \textit{node\_expression\_list} \rangle)$, *the mapping* $\text{Edges} : \mathcal{L}(\langle \textit{smsc\_expression} \rangle) \rightarrow \mathbb{P}(\mathcal{V} \times \mathcal{CT}(\Sigma) \times \mathcal{V})$ *is defined by*

$$\text{Edges(start nodexprs)} = \text{Edges(nodexprs)}.$$

For $\text{nodexpr} \in \mathcal{L}(\langle \textit{node\_expression} \rangle)$, $\text{nodexprs} \in \mathcal{L}(\langle \textit{node\_expression\_list} \rangle)$, *the mapping* $\text{Edges} : \mathcal{L}(\langle \textit{node\_expression\_list} \rangle) \rightarrow \mathbb{P}(\mathcal{V} \times \mathcal{CT}(\Sigma) \times \mathcal{V})$ *is defined inductively by*

$$\begin{aligned} \text{Edges()} &= \emptyset \\ \text{Edges(nodeexpr nodexprs)} &= \text{Edges(nodexpr)} \cup \text{Edges(nodexprs)}. \end{aligned}$$

For $\text{l} \in \mathcal{V}$, $\text{smscid} \in \mathcal{L}(\langle \textit{smsc\_id} \rangle)$ *and* $\text{labels} \in \mathcal{L}(\langle \textit{label\_name\_list} \rangle)$*, the mapping* $\text{Edges} : \mathcal{L}(\langle \textit{node\_expression} \rangle) \rightarrow \mathbb{P}(\mathcal{V} \times \mathcal{CT}(\Sigma) \times \mathcal{V})$ *is defined by*

$$\begin{aligned} \text{Edges(l : smscid } \textbf{seq} \text{ (labels);)} &= \{(l, \overline{\textit{smscid}}, l') | l' \in \text{Succ(labels)}\}, \\ \text{Edges(l : } \textbf{end}\text{;)} &= \emptyset. \end{aligned}$$

where mapping Succ *is as defined in Definition 23.*

DEFINITION 26. **(Initial Nodes)**
For $\text{start} \in \mathcal{L}(\langle \textit{start} \rangle)$, $\text{labels} \in \mathcal{L}(\langle \textit{label\_name\_list} \rangle)$, $\text{nodexprs} \in \mathcal{L}(\langle \textit{node\_expression\_list} \rangle)$*, the mapping* $\text{Initial} : \mathcal{L}(\langle \textit{smsc\_expression} \rangle) \rightarrow \mathbb{P}(\mathcal{V})$ *is defined by*

$$\begin{aligned} \text{Initial(start nodexprs)} &= \text{Initial(start)}, \\ \text{Initial(labels ;)} &= \text{Succ(labels)}. \end{aligned}$$

DEFINITION 27. **(Final Nodes)**
For start $\in \mathcal{L}(\langle start \rangle)$, nodexprs $\in \mathcal{L}(\langle node\_expression\_list \rangle)$, *the mapping* Final : $\mathcal{L}(\langle smsc\_expression \rangle) \to \mathbb{P}(\mathcal{V})$ *is defined by*

$$\text{Final(start nodexprs)} = \text{Final(nodexprs)}.$$

For nodexpr $\in \mathcal{L}(\langle node\_expression \rangle)$, nodexprs $\in \mathcal{L}(\langle node\_expression\_list \rangle)$, *the mapping* Final : $\mathcal{L}(\langle node\_expression\_list \rangle) \to \mathbb{P}(\mathcal{V})$ *is defined inductively by*

$$\begin{array}{ll} \text{Final()} & = \emptyset \\ \text{Final(nodeexpr nodexprs)} & = \text{Final(nodexpr)} \cup \text{Final(nodexprs)}. \end{array}$$

For $\mathrm{l} \in \mathcal{V}$, smscid $\in \mathcal{L}(\langle smsc\_id \rangle)$ *and* labels $\in \mathcal{L}(\langle label\_name\_list \rangle)$, *the mapping* Final : $\mathcal{L}(\langle node\_expression \rangle) \to \mathbb{P}(\mathcal{V})$ *is defined by*

$$\begin{array}{ll} \text{Final(l : smscid } \textbf{seq} \text{ (labels);)} & = \emptyset, \\ \text{Final(l : } \textbf{end}\text{;)} & = \{\mathrm{l}\}. \end{array}$$

DEFINITION 28. *For all* smscexpr $\in \mathcal{L}(\langle smsc\_expression \rangle)$, *the mapping* Graph *is defined by*

$$\begin{array}{rl} \text{Graph(smscexpr)} = (& \text{Nodes(smscexpr)}, \\ & \text{Edges(smscexpr)}, \\ & \text{Initial(smscexpr)}, \\ & \text{Final(smscexpr)}, \\ ). & \end{array}$$

The graph corresponding to a HSMSC obtained above is next transformed into a normal form for which the translation to the process term becomes easy. This transformation is based on node elimination from the graph.

DEFINITION 29. **Node Elimination** *Let* $G = (V, E, I, F)$ *be a graph. Let* $v \in V \backslash I$ *such that* $succ(v) \not\subseteq \{v\}$. *Then* v *is eliminated through the following transformation:* $(V, E, I, F) \mapsto_v (V', E', I', F')$ *where*

$$\begin{array}{rcl} V' & = & V \backslash \{v\}; \\ E' & = & \{n_1 \xrightarrow{t} n_2 \in E | n_1 \neq v \wedge n_2 \neq v\} \\ & \cup & \{n_1 \xrightarrow{e_1 \circ e_2} n_2 | n_1 \xrightarrow{e_1} v \in E \wedge v \xrightarrow{e_2} n_2 \in E \wedge v \notin succ(v)\} \\ & \cup & \{n_1 \xrightarrow{t} n_2 | n_1 \xrightarrow{e_1} v, v \xrightarrow{e_2} n_2 \in E \wedge v \in succ(v) \wedge t = e_1 \circ (\mp_{v \xrightarrow{e} v \in E} e)^\star \circ e_2\}; \\ I' & = & I; \\ F' & = & F. \end{array}$$

The number of nodes in a graph will reduce by one with every application of the preceding rewrite rule. Repeated application of the above rule to a HSMSC graph will reduce it to a normal form such that there is no node v left to which the rewrite rule can be applied.

Finally, a process expression is associated with the HSMSC graph reduced to a normal form as follows. The translation is obtained by composing the translations of the initial nodes using the delayed choice operator.

DEFINITION 30. *Let $G = (V, E, I, F)$ be a HSMSC graph in normal form. Then, the following expression is associated with the graph:*

$$\mathbb{T}(G) \;=\; \mathop{\mp}_{i \in I} [\![i]\!]_G,$$

where

$$[\![i]\!]_G \;=\; \begin{cases} \epsilon^{\infty} & \text{if } succ(i) = \emptyset \wedge i \notin F, \\ \epsilon & \text{if } succ(i) = \emptyset \wedge i \in F, \\ \left(\mathop{\mp}_{i \xrightarrow{e} i \in E} e \right)^{\infty} & \text{if } succ(i) = \{i\} \wedge i \notin F, \\ \left(\mathop{\mp}_{i \xrightarrow{e} i \in E} e \right)^{\star} & \text{if } succ(i) = \{i\} \wedge i \in F, \\ \left(\mathop{\mp}_{\substack{i \xrightarrow{e} j \in E, \\ i \neq j}} e \right) \circ [\![j]\!]_G & \text{if } i \notin succ(i) \wedge succ(i) \not\subseteq \{i\}, \\ \left(\mathop{\mp}_{i \xrightarrow{e} i \in E} e \right)^{\star} \circ \left(\mathop{\mp}_{\substack{i \xrightarrow{e} j \in E, \\ i \neq j}} e \circ [\![j]\!]_G \right) & \text{if } i \in succ(i) \wedge succ(i) \not\subseteq \{i\}. \end{cases}$$

REFERENCES

ALI, S., BRIAND, L. C., ET AL. 2007. A state-based approach to integration testing based on UML models. *Information and Software Technology 49,* 11-12, 1087–1106.

BASANIERI, F., BERTOLINO, A., AND MARCHETTI, E. 2002. The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects. In *UML 2002 The Unified Modeling Language*. Lecture Notes In Computer Science, vol. 2460. Springer Berlin / Heidelberg, 275–303.

BERTOLINO, A., MARCHETTI, E., ET AL. 2005. Introducing a Reasonably Complete and Coherent Approach for Model-based Testing. *Electr. Notes in Theor. Comput. Sci. 116,* 19, 85–97.

B.GENEST, M.MINEA, A.MUSCHOLL, AND D.PELED. 2004. Specifying and verifying partial order properties using template MSCs. In *Foundations of Software Science and Computation Structures*. Lecture Notes in Computer Science, vol. 2987. Springer Berlin / Heidelberg, 195–210.

BRIAND, L. C. AND LABICHE, Y. 2001. A UML-Based Approach to System Testing. In *UML'01: Proceedings of the 4th Intl. Conf. on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*. Springer-Verlag, 194–208.

BROY, M., JONSSON, B., KATOEN, J.-P., LEUCKER, M., AND PRESTCHNER, A., Eds. 2005. *Model-Based Testing of Reactive Systems*. Lecture Notes in Computer Science, vol. 3472. Springer.

B.SENGUPTA. 2003. Triggered message sequence charts. Ph.D. thesis, State University of New York (SUNY) Stony Brook.

B.SENGUPTA AND R.CLEAVELAND. 2002. Triggered Message Sequence Charts. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering (FSE)*. ACM, 167–176.

CTAS. Center-tracon automation system (CTAS) for air traffic control. `http://ctas.arc.nasa.gov/`.

CTRD. Requirements document. `http://scesm04.upb.de/case-study-2/requirements.pdf/`.

DAMM, W. AND HAREL, D. 2001. LSCs: Breathing Life into Message Sequence Charts. *Form. Methods Syst. Des. 19,* 1, 45–80.

DELZANNO, G. 2000. Automatic verification of parameterized cache coherence protocols. In *Computer Aided Verification*. Lecture Notes in Computer Science, vol. 1855. Springer Berlin / Heidelberg, 53–68.

Fraikin, F. and Leonhardt, T. 2002. SeDiTeC-Testing Based on Sequence Diagrams. In *International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 261–266.

Frantzen, L., Tretmans, J., and Willemse, T. A. C. 2006. A Symbolic Framework for Model-Based Testing. In *Formal Approaches to Software Testing and Runtime Verification (FATES/RV)*, K. Havelund, M. Nunez, G. Rosu, and B. Wolff, Eds. Lecture Notes in Computer Science, vol. 4262. Springer Berlin / Heidelberg, 40–54.

Gallagher, L., Offutt, J., and Cincotta, A. 2006. Integration testing of object-oriented components using finite state machines. *Softw. Test. Verif. Reliab. 16,* 4, 215–266.

Goel, A., Meng, S., Roychoudhury, A., and Thiagarajan, P. S. 2006. Interacting process classes. In *28th international conference on Software engineering (ICSE)*. ACM, 302–311.

Goel, A. and Roychoudhury, A. 2006. Synthesis and Traceability of Scenario-Based Executable Models. In *ISOLA '06: Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. IEEE Computer Society, 347–354. Invited Paper.

Goel, A., Sengupta, B., and Roychoudhury, A. 2009. Footprinter: Round-trip engineering via scenario and state based models. In *ICSE Companion*. 419–420.

Grieskamp, W. 2006. Multi-paradigmatic Model-Based Testing. In *Formal Approaches to Software Testing and Runtime Verification (FATES/RV)*. Lecture Notes in Computer Science, vol. 4262. Springer Berlin / Heidelberg, 1–19.

Harel, D. 1987. Statecharts: A visual formalism for Complex Systems. *Science of Computer Programming 8,* 3, 231–274.

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

Hong, H. S., Kim, Y. G., et al. 2000. A test sequence selection method for Statecharts. *Software Testing, Verification and Reliability 10,* 4.

Hong, H. S., Lee, I., et al. 2001. Automatic Test Generation from Statecharts Using Model Checking. In *Workshop on Formal Approaches to Testing of Software (FATES)*. 15–30.

ITU-T Z.100. 1994. Specification and Description Language (SDL), ITU-T Recommendation Z.100.

ITU-T Z.120. 1999. Message Sequence Chart (MSC), ITU-T Recommendation Z.120.

Jard, C. and Jéron, T. 2005. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf. 7,* 4, 297–315.

Joshi, K. D. 1989. *Foundations of Discrete Mathematics*. Wiley-Interscience.

Knapp, A., Merz, S., and Rauh, C. 2002. Model Checking Timed UML State Machines and Collaborations. In *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems*, W. Damm and E. R. Olderog, Eds. Lecture Notes in Computer Science, vol. 2469. Springer Berlin / Heidelberg, 395–414.

Koch, B., Grabowski, J., et al. 1998. Autolink: A Tool for Automatic Test Generation from SDL Specifications. In *IEEE Workshop on Industrial Strength Formal Specification Techniques (WIFT)*. 114–125.

Kruger, I., Prenninger, W., and Sander, R. 2004. Broadcast MSCs. *Formal Aspects of Computing 16,* 3, 194–209.

Kugler, H., Stern, M., and Hubbard, E. 2007. Testing Scenario-Based Models. In *Fundamental Approaches to Software Engineering (FASE)*. Lecture Notes In Computer Science, vol. 4422. Springer-Verlag, 306–320.

Lee, N. H. and Cha, S. D. 2003. Generating test sequences from a set of MSCs. *Comput. Netw. 42,* 3, 405–417.

Marelly, R., Harel, D., and Kugler, H. 2002. Multiple instances and symbolic variables in executable sequence charts. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 83–100.

Mauw, S. and Reniers, M. A. 1999. Operational semantics for MSC'96. *Computer Networks 31,* 17, 1785–1799.

Mortensen, K. H. 2000. Automatic code generation method based on coloured petri net models applied on an access control system. In *ICATPN'00: Proceedings of the 21st international conference on Application and theory of petri nets*. Springer-Verlag, 367–386.

MOST. Media oriented system transport. `http://www.mostcooperation.com/`.

Mousavi, M. R., Reniers, M. A., and Groote, J. F. 2004. Congruence for SOS with data. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 303–312.

Niaz, I. A. and Tanaka, J. 2003. Code generation from uml statecharts. In *Intl. Conf. on Software Engineering and Application (SEA)*. 315–321.

Offutt, J. and Abdurazik, A. 1999. Generating Tests from UML Specifications. In *UML The Unified Modeling Language*. Lecture Notes In Computer Science, vol. 1723. Springer Berlin / Heidelberg, 76–76.

Pelliccione, P., Muccini, H., Bucchiarone, A., and Facchini, F. 2005. TeStor: Deriving Test Sequences from Model-Based Specifications. In *Component-Based Software Engineering (CBSE)*. Lecture Notes In Computer Science, vol. 3489. Springer Berlin / Heidelberg, 267–282.

Pickin, S., Jard, C., Jeron, T., Jezequel, J.-M., and Le Traon, Y. 2007. Test Synthesis from UML Models of Distributed Software. *IEEE Trans. Softw. Eng. 33,* 4, 252–269.

Pnueli, A., Xu, J., and Zuck, L. 2002. Liveness with (0,1,∞)- Counter Abstraction. In *Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 2404. Springer Berlin / Heidelberg, 93–111.

Pretschner, A. 2001. Classical search strategies for test case generation with constraint logic programming. In *In Proc. Formal Approaches to Testing of Software*. BRICS, 47–60.

Pretschner, A., Prenninger, W., et al. 2005. One Evaluation of Model-based Testing and its Automation. In *International conference on Software engineering (ICSE)*. ACM, 392–401.

Rauchwerger, Y., Kristoffersen, F., , and Lahav, Y. 2005. Cinderella SLIPPER: An SDL to C-Code Generator. In *SDL 2005: Model Driven Systems Design*. Lecture Notes in Computer Science, vol. 3530. 210–223.

Reisig, W. 1985. Petri nets: An introduction. *EATCS Monographs in Theoretical Computer Science 4*.

Reniers, M. A. 1999. Message sequence chart: Syntax and semantics. Ph.D. thesis, Eindhoven University of Technology. `http://www.win.tue.nl/~michelr/Files/proefschrift.pdf`.

Rhap. Telelogic rhapsody. `http://modeling.telelogic.com/modeling/products/rhapsody/`.

Roychoudhury, A., Goel, A., and Sengupta, B. 2007. Symbolic message sequence charts. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 275–284.

Rusu, V., du Bousquet, L., and Jéron, T. 2000. An approach to symbolic test generation. In *IFM '00: Proceedings of the Second International Conference on Integrated Formal Methods*. Lecture Notes in Computer Science, vol. 1945. Springer Berlin / Heidelberg, 338–357.

Storrle, H. 2003. Semantics of interactions in UML 2.0. In *HCC '03: Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*. IEEE Computer Society, 129–136.

Tretmans, J. 1996. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software - Concepts and Tools 17,* 3, 103–120.

Tretmans, J. and Brinksma, E. 2002. Côte de Resyste – Automated Model Based Testing. In *Progress 2002 – 3^{rd} Workshop on Embedded Systems*. STW Technology Foundation, 246–255.

UBET. 1999. Ubet. `http://cm.bell-labs.com/cm/cs/what/ubet/`.

Uchitel, S., Kramer, J., and Magee, J. 2002. Negative scenarios for implied scenario elicitation. *SIGSOFT Softw. Eng. Notes 27,* 6, 109–118.

Wang, T., Roychoudhury, A., Yap, R., and Choudhary, S. 2004. Symbolic Execution of Behavioral Requirements. In *Practical Aspects of Declarative Languages (PADL)*. Lecture Notes in Computer Science, vol. 3057. Springer-Verlag, 178–192.

Wittevrongel, J. and Maurer, F. 2001. Using UML to Partially Automate Generation of Scenario-Based Test Drivers. In *7th International Conference on Object Oriented Information Systems (OOIS)*. 303–306.

XSB. Logic programming system. `http://xsb.sourceforge.net/`.