PARAMETERIZED VALIDATION OF UML-LIKE MODELS FOR REACTIVE EMBEDDED SYSTEMS

ANKIT GOEL
(B.Tech. (Hons.), IT-BHU, India)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2009

Contents

Abstract

Distributed reactive systems consisting of classes of behaviorally similar interacting processes arise 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. Various existing modeling notations, such as those included in the UML standard (e.g. State-machines and Sequence diagrams), are not well equipped for requirements modeling of such systems, since they assume a fixed number of processes in the system.

Message Sequence Charts (MSCs) and its variants such as UML Sequence-diagrams are popular notations for modeling scenario-based requirements, capturing interactions among various processes in the system. In this thesis, we develop two UML-like *executable* modeling notations based on MSCs for parameterized validation of distributed reactive systems consisting of classes of interacting processes. These notations are– (i) *Interacting Process Classes (IPC)*, and (ii) *Symbolic Message Sequence Charts (SMSC)*, respectively. We propose an *abstract* execution semantics for both these notations, where we dynamically group together objects at runtime that will have similar future behaviors. We also capture static and dynamic association links between objects, and use class diagrams in a standard way to specify binary inter-class associations. Finally, we study automated test-generation techniques from our modeling notations. The test-cases generated from our MSC-based models provide a crucial link by enabling testing of final implementation with respect to the original requirements.

List of Tables

List of Figures

List of Publications

Journal

1. A. Goel, A. Roychoudhury, and P. Thiagarajan. Interacting process classes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(4), 2009.

Conference

2. A. Goel, B. Sengupta, and A. Roychoudhury. Footprinter: Roundtrip engineering via scenario and state based models. In *ACM International Conference on Software Engineering (ICSE)*, 2009. Short paper.

3. A. Roychoudhury, A. Goel, and B. Sengupta. 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*, pages 275-284. ACM, 2007.

4. A. Goel and A. Roychoudhury. 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*, pages 347-354. IEEE Computer Society, 2006. Invited Paper.

5. A. Goel, S. Meng, A. Roychoudhury, and P. S. Thiagarajan. Interacting process classes. In *ICSE 06: Proceeding of the 28th international conference on Software engineering*, pages 302-311, New York, NY, USA, 2006. ACM.

Others

6. A. Goel and A. Roychoudhury. Test Generation from Integrated System Models capturing State-based and MSC-based Notations. In K. Lodaya et al., editors, *Perspectives in Concurrency Theory (Festschrift for P.S. Thiagarajan)*. University Press (India), 2009.

Part I

Introduction

Chapter 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 popularity of Model Driven Architecture from the Object Management Group (OMG) [87], various model-driven development tools such as those from IBM (Telelogic [110] and Rational [95]), and open-source initiatives such as Eclipse Modeling Framework [33], strongly indicate a growing trend towards model-driven development.

The key idea behind the model driven system development is a clear separation of business and application logic from the underlying platform technologies. Specifically, the Model Driven Architecture (MDA) distinguishes between two kinds of models– (i) Platform Independent Models (or PIMs), capturing the system description free from the details of the underlying platform, and (ii) Platform Specific Models (or PSMs), which include various implementation specific details in addition to the functionality

captured by PIMs. This separation of concerns immediately offers several advantages. First of all, the system description captured by PIMs being independent of specific implementation details, can be *reused* across various implementation platforms. This results in a long lasting intellectual property, while the underlying technology keeps rapidly evolving. Further, PIMs are generally specified using open standard notations such as UML [77], and are therefore vendor neutral, thus allowing for easy migration across technologies. However, that is not all; the use of model based techniques offers various other advantages. Besides serving as initial design documents, system models are used for — (semi-) automated code generation for obtaining a system implementation (e.g. [100]), validation of functional and non-functional requirements through simulation, testing, model-checking etc. (e.g. [69]), and automated model-based test generation for testing system implementations derived separately from the same requirements (e.g. [20]).

Various modeling notations used in model-driven system development can be broadly classified based on– (i) whether they are *visual* (e.g. Statecharts [48], Message Sequence Charts [62]) or *textual* (e.g. CCS [78], Z-notation [121]), and (ii) whether they specify system *behavior* (e.g. Statecharts) or *structure* (e.g. Class-diagrams). In this work, our focus is on the use of UML-like *behavioral* modeling notations for modeling and parameterized validation of distributed systems requirements.

1.1 The Problem Addressed in this work

Distributed reactive systems consisting of classes of behaviorally similar interacting processes arise 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 can also be characterized as *parameterized systems*, the parameter being the number of processes in various classes, while the behavior of each class is specified using a finite state machine. These are well studied in the domain of automated verification (e.g. [28, 91]).

We find that various existing modeling notations, such as those included in the UML standard (e.g. State-machines and Sequence diagrams), are not well equipped for requirements modeling of such systems, since they assume a fixed number of processes in the system. Hence, while constructing a requirements model using the existing notations, number of processes of various types need to be fixed artificially. This has several drawbacks–

- **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 (*are we building the right product*) or verification (*are we building the product right*) results obtained for the restricted system cannot be guaranteed to hold for all implementations of the given system in general.

- **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.

- **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 [62], 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 [48], Live Sequence Charts [27]), 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 large number of objects may easily become error-prone or inefficient, severely limiting the use of model based techniques in such cases.

1.2 Solution Proposed in this dissertation

Message Sequence Charts (MSCs) [62] is a popular *visual* notation for modeling the scenario based requirements, capturing interactions among various processes in the system. In this thesis, we develop two *executable* MSC based notations for parameterized requirements modeling and validation of distributed reactive systems consisting of classes of interacting processes.

In the modeling frameworks 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 (or a set of events appearing along a lifeline in a MSC). In our setting, the constraints for selecting objects to participate in various event(s) may consist of one or more 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, and (iii) a constraint regarding an object's association links (with other

participating objects).

Thus, a requirements model in our framework may consist of a large or, even an unbounded number of objects in various process classes. If the execution semantics of such systems maintains the local state of each object, this will lead to an impractical blow-up during execution. Instead, we propose an *abstract* execution semantics, where we dynamically group together objects at runtime that will have similar future behaviors. While doing so, we keep track of only the number of objects in each equivalence class and not their identities. This results in considerable time and memory efficiency of our simulators.

We also capture static and dynamic association links between objects, and use class diagrams in a standard way to specify binary inter-class associations. Structural constraints imposed by the system are naturally captured via static associations. For instance, a node may be able to take part in a "transmit" transaction only with nodes with which it has a "neighbor-of" association. *Dynamic associations* on the other hand are needed to guarantee that proper combinations of objects take part in a transaction. For instance, when choosing a pair of phone objects to take part in a "disconnect" transaction, we may have to choose a pair which is currently in the "connected" relation. This relation has presumably arisen due to the fact that they took part last in a "connect" transaction. The combination of these features together with the imperative to develop an abstract execution semantics is a challenging task. We note that this issue does not arise in parameterized verification, since no associations

are maintained there.

1.3 Contributions of this thesis

Modeling notations. We develop two modeling frameworks, where one can efficiently simulate and validate a system with an arbitrary number of active objects, such as a telephone switch network with thousands of phones, an air traffic controller with hundreds of clients etc. The first modeling notation of *Interacting Process Classes* [41, 44] uses labeled transition systems to describe intra-process control flow of various classes in the system description, while using a high-level notion of transactions to capture interactions among various process classes. In our setting we use Message Sequence Charts (MSCs) [62] to capture transactions, since they form a natural candidate for describing inter-process interactions and are also widely used. The second notation of *Symbolic Message Sequence Charts* [101] is a light-weight extension of the MSC notation. While, in the case of MSCs, a lifeline can represent only a concrete object, SMSCs introduce the concept of a symbolic lifeline. Instead of representing a single object, a symbolic lifeline in a SMSC represents a collection of objects from a class. During execution, a set of objects to execute an event occurring along a SMSC lifeline is dynamically chosen based on the event guard.

Maintaining associations in abstract setting. our abstract execution semantics (for both IPC and SMSC notations) do not maintain the identity or

state of an individual object at runtime. Thus, challenges arise if static and/or dynamic association links need to be maintained between various objects at runtime. We address this issue by maintaining over-approximate association information at runtime, where we maintain links between groups of objects, with each object-group specifying an object state and number of objects currently in that state. Though maintaining over-approximate association information preserves all valid system behaviors, it may give rise to spurious behaviors.

Test-case generation. 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, it is likely to deviate from the requirements and 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 [45]. We study test-generation from our modeling notations which are scenario-based [43].

1.4 Organization of the Thesis

In the following chapter we present a discussion of the related work, also comparing our work with the existing literature. In Chapter 3, we discuss in detail our modeling notation of *Interacting Process Classes (IPC)*, while the notation of *Symbolic Message*

Sequence Charts (SMSCs) is discussed in Chapter 4. Besides presenting the syntax and abstract execution of semantics of the IPC and SMSC notations, the maintenance of association links in the abstract setting is also discussed in Chapters 3 and 4. Automated generation of test-cases from our IPC and SMSC models is described in detail in Chapters 7 and 8, respectively. Finally, concluding remarks appear in Chapter 9, along with a discussion on extensions and directions for future research.

Part II

Modeling Notations

Chapter 2

Related Work

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

We can broadly categorize various behavioral modeling notations into the following two categories: a) *Intra-object* (or *state-based*) notations, or b) *Inter-object* (or *scenario-based*) notations. The state-based notations specify the control flow of various classes of objects in a system description using finite state machine representations such as Statecharts [49]. The scenario-based notations, such as, Live Sequence

Charts [27] and Triggered Message Sequence Charts [106], are used to specify various interaction scenarios between system processes. There has also been work on using a combination of state-based and scenario-based notations, where the control flow for various processes is specified using labeled transition systems and the inter-object interactions are specified using Message Sequence Charts (MSCs) [103]. All these approaches deal with *concrete* objects and their interactions.

2.1 State-based models

Some of the widely used executable state-based notations in the design and analysis of reactive systems are Statecharts [48], Specification and Description Language (SDL) [2], and Petri-nets [97]. Statecharts provide a hierarchical state-based behavior description mechanism, allowing user to specify both concurrent (using AND-states) and sequential (using OR-states) behavior of a process. The execution of system is event-driven. A transition between two states is generally triggered by an external event (from the environment), or by a message sent by another process in the system. There are a number of design methodologies based on some variant of Statecharts, as exemplified in the commercial tools such as Rhapsody and RoseRT [100, 95].

A Specification and Description Language (SDL) system description consists of concurrent processes described using notation similar to the extended finite state machines, and communicating via signal (or message) exchange over FIFO channels. The following ITU standard [61] provides a formal syntax and semantics for SDL, support-

ing both textual as well as graphical specification of SDL models. The Telelogic SDL Suite [3] from IBM is a commercial tool offering a SDL-based design methodology for real-time communication systems. The tool supports visual modeling of SDL designs, their simulation, and automated code generation.

Petri-nets are a graphical and mathematical modeling tool, most commonly used for the description of concurrent distributed systems, and their analysis. A petri-net description consists of *places* and *transitions*, with *input-arcs* connecting places to transitions and *output-arcs* connecting transitions to places. Places generally represent process states, while transitions represent concurrent activities in which one or more processes participate together. A transition is enabled for execution if there are enough *tokens* (representing processes) in all the places connected to the given transition via input arcs. Several tools supporting petri-nets based modeling and analysis are available (see [90]).

There has also been work on using a combination of state-based and scenario-based notations, where the control flow for various processes is specified using labeled transition sys- tems and the inter-object interactions are specified using Message Sequence Charts (MSCs) [103].

Various design methodologies based on above notations do not provide an abstract execution semantics[1], which can facilitate scalable and efficient model-based simulation and validation of systems consisting of classes with a large (or even unbounded)

[1]Note that, in this work we only consider process abstraction– grouping together of behaviorally similar processes, and do not consider data abstraction over process variables.

number of objects. Further, inter-process interactions are specified at a fairly low level of granularity, typically a single message send/receive. Both our notations of Interacting Process Classes (IPC) and Symbolic Message Sequence Charts (SMSC), which we discuss in this thesis, support an abstract execution semantics, allowing for validation of distributed reactive systems consisting of a large number of behaviorally similar processes. The IPC is an object-oriented extension of the work in [103], and is a state-based notation (unlike SMSCs, which are purely scenario-based). The IPC notation allows inter-process interactions to be described at a fairly high-level of granularity, e.g. using Message Sequence Charts (MSCs).

The new standard UML 2.0 advocates the use of "structured classes" where interaction relationships between the sub-classes can be captured via entities such as ports/interfaces; Our present frameworks do not cater for structured classes but it can easily accommodate notions such as ports/interfaces.

Here, we also mention various programming frameworks, such as Lustre and Esterel[2], specifically targeted towards development of reactive control systems. However, they are more useful in the later stages of system development, when requirements have stabilized and focus is on obtaining the system implementation. Further, these frameworks assume the perfect synchrony hypothesis, where the computation/communication for processing all events that occur within one clock tick happen instantaneously. In our case, we do not make any such assumptions regarding

[2] http://www.esterel-technologies.com/

computations or, communication among processes. We target a more high-level initial system requirements, focusing on inter-process interaction protocols, rather than computational aspects (e.g. computing output signals in response to input signals).

2.2 Scenario-based Models

The distributed system requirements generally focus on specifying inter-process interactions and abstract away from the local computations, and are therefore more naturally captured using scenario-based notations such as Message Sequence Charts (MSCs) [62]. Examples of such requirements are often found in practice, for instance–

- Requirements document for Center-TRACON Automation System (or, CTAS) [1]. CTAS is a control system aiding in management of arrival air-traffic at busy airports (discussed in Section 4.2).

- Media Oriented Systems Transport (or, MOST) [79], a multimedia and infotainment networking standard for the automotive industry (discussed in Section 7.1). One of the specification documents, namely the 'MOST dynamic specification', contains the scenario based requirements for this standard.

The simplest form of MSC specification consists of a *basic* MSC (or bMSC), representing a single interaction scenario among a finite set of processes. Various processes participating in a bMSC are represented as vertical lines (called *lifelines*), which exchange messages among themselves and may also participate in local computation

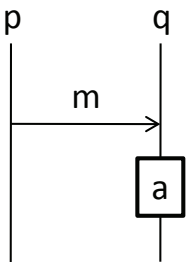


Figure 2-1: An example MSC.

actions [62]. In the following, we refer to a bMSC simply as MSC. A message exchanged between two lifelines in a MSC is represented by a horizontal or a downward sloping arrow, from the sending process to the receiving process. An example of a MSC is shown in Figure 2-1. It represents two processes p and q, exchanging a message m (from p to q). Further, process q participates in a local action a.

While an MSC represents a single execution scenario of the system, more complete system descriptions can be obtained in the form of High-level Message Sequence Charts (HMSCs). An HMSC is a directed graph, whose nodes are labeled with other (H)MSCs with a finite level of nesting. Various HMSC nodes can be flattened out such that, each node simply corresponds to an MSC — the resulting structure is also referred to as *Message Sequence Graph (MSG)*. In our discussions, we assume HMSC to be a flattened structure. Further, an HMSC has a unique *start* node, with a subset of its nodes designated as *final* nodes. A path from the start node to a final node in an HMSC represents an accepting path. The MSC corresponding to an accepting path in a HMSC [7] represents a valid run of the system.

Other extensions to HMSC notation, namely *Compositional* Message Sequence Charts [47] and *Causal* Message Sequence Charts [37], have also been studied. Both these notations improve the expressive power of HMSCs by allowing specification of MSC protocols which cannot be specified using HMSCs.

2.2.1 Analysis of MSC Specifications

The MSCs are equipped with a formal execution semantics [98], and can be subjected to various analyses such as detecting race conditions and timing conflicts [7], detecting non-local choice and process divergence [15], and model checking [8]. Race conditions in an MSC can exist when the visual event ordering described in the MSC may be violated at runtime due to the underlying communication architecture. For example, two messages sent by the same process to another process may not arrive in order, if the underlying communication is not FIFO. In [7], an MSC analyzer is presented for detecting such conflicts, given an underlying communication architecture. The non-local branching choice intuitively refers to the inability of the processes to locally make a (globally) consistent choice regarding which branch to pursue at a branching node in an HMSC specification. This leads to problems in directly obtaining a distributed implementation from an HMSC. Process divergence, on the other hand, refers to the possibility of a process making unbounded progress relative to other processes in a given HMSC. In [15], authors give syntax-based analysis algorithms for detecting both non-local choice and process-divergence. Model-checking HMSC specifications refers

to validating them against properties that may be described using logics (e.g. Linear Temporal Logic [23]), as automata, or as HMSCs. Model-checking HMSCs in general is shown to be *undecidable*, whereas it is *decidable* for the *bounded* subclass of HMSCs [8]. Consequently, model-checking has been studied for other subclasses of HMSCs where it becomes decidable (e.g. [38]).

We note that various decidability results for MSCs also apply to our notation of SMSCs (discussed in Chapter 4), which is an extension of the MSC notation.

2.2.2 Realizability and Implied Scenarios

An HMSC specification is said to be *realizable* (or, *implementable*), if there exists a distributed implementation that generates precisely the specification behaviors [6]. Since, the ultimate goal is to obtain a system implementation satisfying the given requirements, the synthesis of MSCs has been widely studied, for example, as Statecharts [71, 119] or Communicating Finite State Machines (CFSMs) [6, 81, 38]. A different approach in this direction is the work on Netcharts [82], which are a visual formalism for specifying collections of MSCs, similar to HMSCs. However, unlike HMSCs where control of processes is specified at a global level, Netcharts specify distributed process control. This leads to a more natural and direct translation of Netcharts into Communicating Finite State Machines. Further, Netcharts are more expressive than HMSCs, in the sense that they can specify all *regular* MSC languages [53], unlike HMSCs, which can only describe finitely-generated regular MSC lan-

guages. The above notion of regularity is studied for MSCs, since it directly relates to their realizability by means of bounded message passing automata.

However, implementations obtained from the HMSC specifications may give rise to *additional* behaviors, which are not present in the original scenario specification. These behaviors are referred to as *implied-scenarios* [114, 6], which mainly arise because various components have a local view of system behavior, and may not make globally consistent decisions (as per HMSC specification). An implied scenario may be desirable, i.e. an acceptable scenario has been overlooked and needs to be incorporated in the system specification, or undesirable, representing unacceptable behavior. An approach for detecting implied scenario is presented in [114]. A distinction between "positive" and "negative" implied scenarios is made in [115]. If a detected implied scenario corresponds to a negative scenario specified with the requirements, then it is not reported (as user is already aware of its presence). For a reported implied scenario, user classifies it as either positive or negative, accordingly updating the requirements model to eliminate the implied scenarios.

In our case, since we are adding to the expressive power of MSCs in our notation of SMSCs, various issues regarding realizability of HMSCs and existence of implied scenarios also extend to SMSCs. However, in the current work we do not investigate implementation of SMSC based specifications.

2.2.3 Scalability of MSC Specifications

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 (which we develop for our notation of SMSCs in Chapter 4), and thus such approaches only partially address the MSC scalability problems that allow similar interaction patterns to be concisely represented as in SMSCs. 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 [98]. On the other hand, we introduce event guards in SMSCs, which not only refer to conditions on variables of concrete objects, but also serve as an object selector from a collection of objects during execution.

2.2.4 Other Notations

In recent years, a number of MSC variants have been proposed (*e.g.* [18, 106, 27, 115]). Of these, Live Sequence Charts (LSCs) and Triggered MSCs (TMSCs) are equipped

with an execution semantics [52, 21]. The notation of Live Sequence Charts (LSCs) [27, 51] offers an MSC-based inter-object modeling and simulation framework for reactive systems. LSCs describe system behavior by prescribing existential and universal temporal properties that the overall system interactions should obey, rather than giving a per-process state machine. Consequently, the control flow information pertaining to individual processes is completely suppressed in LSCs. More importantly, we note that in the LSC framework, though the objects of a process class can be *specified* symbolically, the execution mechanism (the play-engine as described in [51]) does not support abstract execution. The symbolic instances must be instantiated to concrete objects during simulation. The approach taken in [117] 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. In the case of two modeling notations –IPC and SMSC– that we present in this thesis, we do not maintain any process identifiers, leading to a full symbolic execution semantics. Also, the work on Triggered Message Sequence Charts [106] allows for a non-centralized execution semantics, in comparison to the play-engine of LSCs. However, the TMSC language supports neither symbolic specifications nor abstract execution of process classes.

2.3 Parameterized System Verification

In this thesis, we deal with distributed systems consisting of classes of interacting processes, where a class may contain unboundedly many processes. The family of systems with many concurrent processes of the same type, are also known as *parameterized systems*. Such systems have mainly been studied in the context of *parameterized verification* (e.g. [24]).

Verification of parameterized systems is known to be undecidable [10], and two distinct approaches are considered to address this problem. We either look for restricted subsets of parameterized systems for which the verification problem becomes decidable, or we look for sound but not necessarily complete methods.

The first approach tries to identify a *restricted subset* of parameterized systems and temporal properties, such that if a property holds for a system with up to a certain number of processes, then it holds for every number of processes in the system. Moreover, the verification for the reduced system can be accomplished by model checking. Systems that are verified with this approach include systems with a single controller and arbitrary number of user processes [39], rings with arbitrary number of processes communicating by passing tokens [32, 31], systems formed by composing an arbitrary number of identical processes in parallel [60], and systems formed by unbounded processes of several process types where the communication mechanism between the processes is restricted to conjunctive / disjunctive transition guards [30].

The sound but incomplete approaches include methods based on synthesis of in-

visible invariant (*e.g.*, [9]) which can be viewed as a combination of assertion synthesis techniques with abstraction for verification; methods based on network invariant (*e.g.*, [75]) that relies on the effectiveness of a generated invariant and the invariant refinement techniques; regular model checking [66, 99] that requires acceleration techniques. Compositional proof methods have been studied in [14], while explicit induction based proof methods for parameterized families have been discussed in [102].

The abstract execution semantics that we develop for our IPC and SMSC modeling notations in this thesis, involve grouping together active objects at runtime into equivalence classes. This approach is related to *counter abstraction* schemes developed for grouping processes in parameterized systems (*e.g.*, see [28, 91]). In such systems, there are usually unboundedly many processes whose behavior can be captured by a single finite state machine or an extended finite state machine. It is then customary to maintain the count of number of processes in each state of the finite state machine; the names/identities of the individual processes are not maintained. For instance, in [91] a concrete system consisting of $n > 1$ identical processes is abstracted into a finite state system in which for each local state, the process count can be either 0, 1 or 2. The count of 0(1) indicates currently zero(single) process in the corresponding state, while count of 2 indicates 2 or more processes in a state. In our abstract execution semantics for IPC and SMSC, we also abstract away process identities, and maintain only the count of objects in each execution state at runtime. Further, for a class p with unboundedly many objects, a *cutoff* number c_p is used such that the

count of objects in a p-state can be a fixed number belonging to $[0, c_p]$, or it can be ω– representing $c_p + 1$ or more objects of class p. Note that, this is similar to the count-abstraction for parameterized systems as described above (with $c_p = 1$). However, in our setting, currently we do not employ our abstract execution semantics for the purpose of parameterized verification. Rather, the focus is on parameterized requirements modeling, validation and test-case generation for distributed systems consisting of interacting classes. Additionally, in our setting inter-object associations across classes have to be maintained — an issue that does not arise in parameterized system verification. This indeed is one of the key technical challenges in our work: How does one capture the information, say, "an object i is linked via some association asc to an object j" when we do not maintain object identities in our abstract execution semantics?

2.4 Model Checking and Data Abstraction

There are several works which employ data abstraction techniques for the purpose of model checking [25, 54, 22]. The main aim is to reduce the state space of a given system-model in order to make model checking tractable. In these approaches, various data types are replaced with smaller-sized types, thus obtaining an abstract model of the system in terms of behaviors, which is generally an over-approximation of the original system. Such an abstraction preserves the correctness of properties, in the sense that, a property proven correct for the abstract model, also holds for the

original model. In our present work, we only abstract away the process identities and dynamically group together various processes having identical states during run-time (a form of control abstraction [68]). Thus, currently we do not abstract away any other data types; but it can be easily integrated in to our framework.

2.5 The Semantics of a Class

Finally, we note that both the formalisms proposed in this thesis –*Interacting Process Classes (IPC)* and *Symbolic Message Sequence Charts (SMSCs)*– support object-oriented modeling of requirements. Hence, a specification in either of these two notations consists of a finite set of *classes*, whose instances (commonly referred to as *processes* in our work) constitute the interacting entities.

In general, two types of semantics are associated with a class in the object-oriented setting– *value-based*, or *reference-based* [107]. In the case of *value semantics*, a class is denoted by a set of values, where each value in such a set corresponds to an object of the class at some stage of its evolution. While, in the case of *reference semantics*, a class is denoted by a set of references to values (denoting objects). In our present work, we assume *value semantics* for various process-classes in a specification.

Chapter 3

Interacting Process Classes (IPC)

The notation of *Interacting Process Classes (IPC)* [41, 44] uses labeled transition systems to describe the behavior of classes of interacting objects. However, the unit of interaction rather than just being a message exchange between a process pair, may involve more than two participants. Further, an interaction unit can describe an abstraction of a protocol that involves bidirectional flow of signals and data between the objects taking part in the interaction. In our operational semantics, this unit of interaction, called a **transaction**, is executed in an atomic fashion. For illustrative purposes however, we use the notation of Message Sequence Charts (MSCs) to refine the transactions.

We develop the IPC modeling language in two steps. First we present the core modeling language without associations and develop its concrete execution semantics. We then formulate our abstract execution semantics. As a second step, we introduce

(static and dynamic) object associations and correspondingly extend the semantics. Further, we establish results relating the concrete semantics to the abstract semantics. Finally, we present experimental results demonstrating capabilities of the simulator for our model.

3.1 The Modeling Language

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

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

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

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

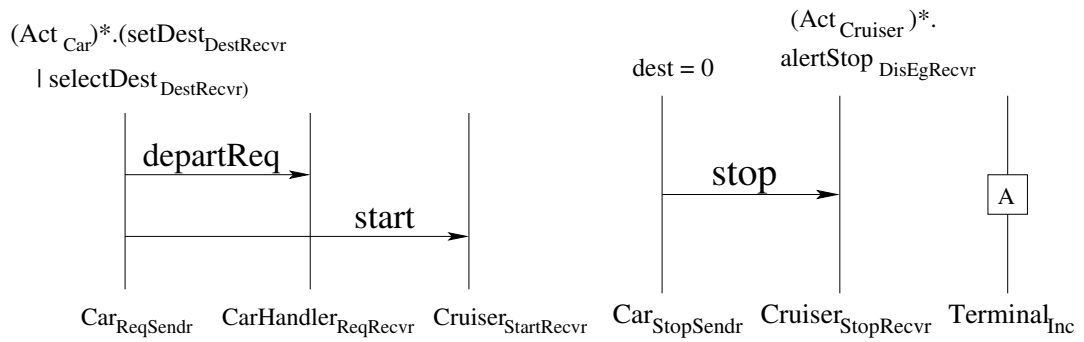
3. Ch represents the *behavior* corresponding to roles R in transaction γ.

The set of all transactions is denoted as Γ.

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

For the purpose of exposition we will use Message Sequence Charts (MSCs) for describing the behavior (Ch) of a transaction. For a transaction $\gamma = (R, I, Ch) \in \Gamma$, we view Ch as a labeled poset of the form $Ch = (\{E_r\}_{r \in R}, \leq, \lambda)$[1] where E_r is the set of events that the role r takes part in during the execution of γ. The labeling function λ, with a suitable range of labels, describes the messages exchanged by the instances as well as the internal computational steps during the execution of Ch. Finally, $\leq$ is

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



(a) departReqA (b) noMoreDest

Figure 3-1: Transactions *departReqA* & *noMoreDest*; A is an internal computation event in *noMoreDest*

the partial ordering relation over the occurrences of the events in $\{E_r\}_{r \in R}$. Within the MSCs, the communication via sending and receiving of messages can be synchronous or asynchronous. This issue is orthogonal to our model. In the operational semantics of our model, we assume for convenience that the execution of each transaction is atomic. As a consequence, the concatenation of MSCs in our setting, say, to depict the execution of a sequence of transactions, will be synchronous [7].

We show an example transaction in Figure 3-1(a). It occurs in our IPC model of Rail-car example presented in [27, 49]. For the moment, let us ignore the regular expression at the top portion of this chart. Note that as a notational shorthand we often write the role (p, ρ) as p_ρ. This convention has been followed in the transactions of Figure 3-1. In Figure 3-1(a), the roles are "$Car_{ReqSendr}$", "$CarHandler_{ReqRecvr}$" and "$Cruiser_{StartRecvr}$". This naming convention is intended to indicate that the functionality "ReqSendr" is played by an object drawn from the class "Car", the functionality "ReqRecvr" is played by an object belonging to the class "CarHandler", and so on.

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

As mentioned earlier, in a transaction $\gamma = (R, I, Ch)$, the **guard** I_r associated with the role $r = (p, \rho)$ will specify the conditions that must be satisfied by an object O_r belonging to the class p in order for it to be eligible to play the role r. These conditions will consist of two components: The first one is a *history* property of the execution sequence of actions that O_r has so far gone through. It will be captured using *regular expressions* over the alphabet Act_p, the set of all the action labels corresponding to process class p. We denote it using Λ. The second component is a *propositional formula* (denoted as Ψ) built from boolean assertions regarding the values of the variables owned by O_r. Thus, guard of a role r is of the form $I_r = (\Lambda, \Psi)$. For instance, consider the transaction "departReqA" shown in Figure 3-1(a)). A Car object wishing to play the role (*Car, ReqSendr*) must have last played the role (*Car, DestRecvr*) in the transaction *setDest* or in the transaction *selectDest*. This is

captured by the guard

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

shown at the top of the lifeline corresponding to role (*Car, ReqSendr*) in Figure 3-1(a). As this example shows, we use regular expressions to specify the history component of a guard. Also, note that in the transaction "departReqA" , the guard does not restrict the local variable valuation of participating objects in any way. On the other hand, in the transaction of Figure 3-1(b), the variable "dest" owned by the car-object intending to play the role (*Car,StopSendr*) must satisfy "dest = 0"; there is no execution history based guard for this role. Finally, if for some role no guard is mentioned (e.g. $Cruiser_{StartRecvr}$ in Fig.3-1(a)) then the corresponding guard is assumed to be vacuously true.

The transition system describing the common control flow of the objects belonging to the class p will be denoted as TS_p. It is a structure of the form

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

We first explain the nature of the components Act_p, V_p and v_{init_p}. As described earlier, Act_p is the set of action labels, with each label specifying a transaction and a role in that transaction, that the p-objects can play in the transactions in Γ. The effects of the computational steps performed by an object will be described with the help

of the set of variables V_p associated with p. Each object O in p of course will have its own copy of the variables in V_p. For convenience, we shall assume that for each variable $u \in V_p$, all the objects of class p assign the same initial value to u. This initial assignment is captured by the function v_{init_p}. We assume appropriate value domains for the variables in V_p exist but will not spell them out here. A computational step can be viewed as a degenerate type of transaction having just one role. Hence we will not distinguish between computational steps and transactions in what follows. Returning to $TS_p = \langle S_p, Act_p, \rightarrow_p, init_p, V_p, v_{init_p} \rangle$, S_p is the finite set of local states, $init_p \in S_p$ is the initial state and $\rightarrow_p \subseteq S_p \times Act_p \times S_p$ is the transition relation. In summary, our model can be defined as follows.

Definition 1. The IPC Model *Given a set $\mathcal{P}$ of process-classes, a set Γ of transactions and a set of action labels Act_p for $p \in \mathcal{P}$ involving transactions from Γ, a system of Interacting Process Classes (IPC) is a collection of $\mathcal{P}$-indexed labeled transition systems $\{TS_p\}_{p \in \mathcal{P}}$ where*

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

is a finite state transition system as explained above.

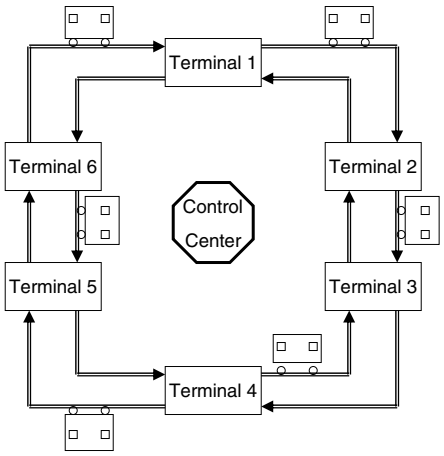


Figure 3-2: Rail-car system

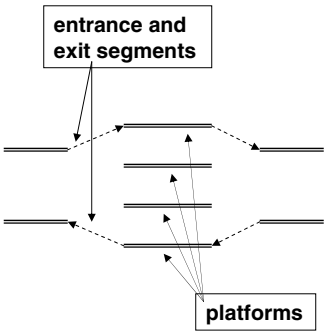


Figure 3-3: Terminal

3.2 Modeling A Rail-Car System: The First-Cut

We discuss here the preliminary modeling of a Rail-car example [27, 49] using the Interacting Process Classes (IPC) formalism. This example was initially developed in [49] and used subsequently in [27] to illustrate the modeling capabilities of Live Sequence Charts. It is a non-trivial distributed control system with many process classes such as cars, cruiser, terminal etc. The schematic structure is shown in Figure 3-2. There are six terminals located along two parallel cyclic tracks, one of them running clockwise and the other anti-clockwise. Each adjacent pair of these terminals is connected by the two parallel tracks. There is fixed number of rail cars for transporting passengers between the terminals. There is a control center which receives, processes and communicates data between various terminals and railcars.

As shown in Figure 3-3, each terminal has four parallel platforms. At any time at most one car can be parked at a platform. Further, there are two *entrance* and

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

Table 3.1: Process Classes & Object counts in Rail-car Example with 48 cars

two *exit* segments which connect the two main rail tracks to the terminal's platform tracks. Also, each terminal has a destination board for the use of passengers. It contains a push button and an indicator for each destination terminal. Each rail-car also has a similar destination-panel for the use by passengers. Further, a rail-car is equipped with an engine and cruise-controller to maintain the speed. The cruiser can be off, engaged or disengaged.

The list of process classes and the number of concrete objects in each process class for a rail-car system with 48 cars is shown in Table 3.1. Note that $tDestPanel$ represents the destination panel in the terminal and $cDestPanel$ represents the destination panel in the rail-car. Thus, with 48 cars and 6 terminals, we have 48 cDestPanel and only 6 tDestPanel (refer Table 3.1). We have only one $ControlCenter$ object which is related to all the $Terminal$ and Car objects. Also each Car is associated to a $ProximitySensor$, which notifies the car when it arrives within 100 and 80 yards of

some terminal, and also associated to a $Cruiser$ which maintains the car speed.

When a car is at a terminal or arrives in one, a unique $CarHandler$ gets associated with the Car to handle communication between the car and the terminal. Once the car leaves the terminal a $CarHandler$ is no longer associated with it. We have two $Entrance$ and two $Exit$ objects associated with each terminal. They represent the entrance and exit segments connecting the rail tracks to the terminal's platforms. $PlatformManager$ and $ExitsManager$ respectively allocate platforms and exits to $CarHandler$, which in turn notifies the Car of these events.

We show a fragment of the IPC model of the Rail-car example in Figure 3-4. Controlling the movement of the cars between the terminals involves a complex description. The classes shown in Figure 3-4 are Car, $Cruiser$, $Terminal$ and $CarHandler$. The Cruiser stands for the cruise control of a car. This will be captured as associations via Class Diagrams as discussed in Section 3.5. The CarHandler manages interaction between an approaching/departing car and the corresponding terminal.

In Figure 3-4, for each process class, we have shown a fragment of the transition system corresponding to that process class. As explained in the last section, the action labels of the transition system for a process class specify a transaction and a role in that transaction (which we model using a Message Sequence Chart). We have not shown all the transactions corresponding to the (transaction, role) pairs appearing as action labels in Figure 3-4; there are too many of them. However, two of these transactions, namely, *departReqA* and *noMoreDest* appear in Figure 3-1.

In this preliminary model, we do not discuss associations; the class associations for the rail-car example appear in Figure 3-5. The modeling and simulation of process classes with associations will be dealt with in Section 3.5 after developing the execution semantics of the core model.

3.3 Concrete Execution Semantics

We now formulate a **concrete execution semantics** of the IPC model. Given a set of transactions Γ and an IPC model $\{TS_p\}_{p\in\mathcal{P}}$ as defined in Section 3.1 (Definition 1), for any class p we define H_p to be the least set of minimal DFAs (Deterministic Finite State Automata) [58] given by: $\mathcal{A}$ is in H_p iff there exists a transaction $\gamma = (R, I, Ch)$ and a role $r \in R$ of the form (p, ρ) such that the guard I_r of r is (Λ, Ψ) and $\mathcal{A}$ is the minimal DFA recognizing the language defined by the regular expression Λ, the history part of the guard, i.e.

$$H_p = \{dfa(\Lambda) | \gamma = (R, I, Ch) \in \Gamma \wedge r = (p, \rho) \in R \wedge I_r = (\Lambda, \Psi)\}. \qquad (3.1)$$

Expression $dfa(\Lambda)$ in Eq. (3.1) above represents the minimal DFA corresponding to regular expression Λ.

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

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

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

Let $H_p = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ *be the set of minimal DFAs defined for class* p *(Eq.*(3.1)*). Then a* behavioral partition beh_p *of class* p *is a tuple* $(s, q_1, \ldots, q_k, v)$*, where*

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

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

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

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

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

where BEH_p *is the set of all behavioral partitions of class* p *(Definition 2, page 40). The set of all concrete configurations of* $\mathcal{S}$ *is denoted as* $\mathcal{C}^c_{\mathcal{S}}$*.*

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

1. $s \xrightarrow{(\gamma_r)} s'$ is a transition in TS_p

2. For all $1 \leq i \leq k$, if $\mathcal{A}_i$ is the DFA corresponding to the regular expression of Λ, then q_i is an accepting state of $\mathcal{A}_i$.

3. $v \in Val(V_p)$ satisfies the propositional guard Ψ.

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

- $s \xrightarrow{(\gamma_r)} s'$ is a transition in TS_p.

- for all $1 \leq i \leq k, q_i \xrightarrow{(\gamma_r)} q'_i$ is a transition in DFA $\mathcal{A}_i$.

- $v' \in Val(V_p)$ is the effect of executing γ_r on v.

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

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

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

Example For illustration, consider the example shown in Figure 3-4. Suppose c is a concrete configuration at which

- Two *Car* objects Oc_1 and Oc_2 are residing in state *stopped* and a third object, Oc_3, is in state $s2$ of TS_{Car}. Further suppose they have the values 0, 1 and 2 respectively for the variable *dest* and have no regular expression based history guards. Then we have– $pmap_{Car}(Oc_1) = \langle stopped, 0\rangle$, $pmap_{Car}(Oc_2) = \langle stopped, 1\rangle$ and $pmap_{Car}(Oc_3) = \langle s2, 2\rangle$.

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

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

while the history of O_3 satisfies the regular expression

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

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

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

Suppose we want to execute transaction *noMoreDest* shown in Figure 3-1(b) at configuration c. As for the role (*Car,StopSendr*), though Oc_1 and Oc_2 are in the appropriate control state, only Oc_1 can be chosen since it (and not Oc_2) satisfies

the guard $dest = 0$. For the cruisers, we observe that all the three *Cruiser* objects O_1, O_2, O_3 are in the "appropriate" control state at configuration c for the purpose of executing *noMoreDest*. However, only O_1 and O_2 have histories which satisfy the history part of the guard associated with the role (*Cruiser,StopRecvr*), i.e. they are in the accepting state q_2 of DFA $\mathcal{A}_1$ representing this history guard. Hence either one of them (but not O_3) can be chosen to play this role. For the role (*Terminal,Inc*), both the history and propositional guards are vacuous and hence we can choose any one of the 6 objects residing in the control state $s1$.

Assume that Oc_1, O_1 and Ot_1 are chosen to execute transaction *noMoreDest* in configuration c. In the resulting configuration c', all objects other than Oc_1, O_1 and Ot_1 will have their control states, histories and variable valuations unchanged from c, thus remaining in the same behavioral partitions as c. The objects Oc_1, O_1, Ot_1 will move to control states *idle*, *stopped*, $s1$ in their respective transition systems. Value of variable *dest* for Oc_1 remains unchanged, while the state of DFA $\mathcal{A}_1$ for O_1 is updated to q_1. Thus at the resulting configuration c' we have– $pmap'_{Car}(Oc_1) = \langle idle, 0\rangle$, $pmap'_{Cruiser}(O_1) = \langle stopped, q_1\rangle$ and $pmap'_{Terminal}(Ot_1) = \langle s1\rangle$.

3.4 Abstract Execution Semantics

We observe that various objects of a process class have same local state (i.e. they map to the same behavioral partition) at a given concrete configuration during execution, and are thus *behaviorally indistinguishable*. Further, for classes with unboundedly

many instances at run-time, the concrete execution semantics will produce an infinite-state system. Thus, we formulate an **abstract execution semantics** of the IPC model, where we do not maintain states and identities of individual objects. Instead, during execution, the objects of a class are grouped into *behavioral partitions*.

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

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

This implies that the computation trees of two objects in the same behavioral partition at a concrete configuration are isomorphic. We make use of this property for dynamically grouping together objects of a process class into behavioral partitions. This is a strong type of behavioral equivalence to demand. There are many weaker possibilities but we will not explore them here.

3.4.1 Abstract Execution of Core Model

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

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

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

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

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

- *BEH_p is the set of all behavioral partitions of class p,*

- *ω represents an unbounded number.*

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

We note that N_p can be a given positive integer constant or it can be ω (standing for an unbounded number of objects). If N_p is ω, our operational semantics remains unchanged provided we assume the usual rules of addition/subtraction (i.e. $\omega+1 = \omega$, $\omega-1 = \omega$ and so on). For a process class p, N_p can be ω if, either a) the initial object count for p is explicitly specified as ω, in order to model and validate a system with

any number of p objects, or b) objects of class p can be dynamically created and we specify a threshold object count, exceeding which the object count of p will become ω. We discuss the exact mechanism for object creation/deletion for a process class p later in Section 3.4.2. For convenience of explanation, we assume that N_p is a given constant in the rest of our discussion.

Our abstract execution efficiently keeps track of the objects in various process classes by maintaining the current abstract configuration; only the behavioral partitions with non-zero counts are kept track of. The system moves from one abstract configuration to another by executing a transaction. How can our simulator check whether a specific transaction γ is *enabled* at an abstract configuration cfg? Since we do not keep track of object identities, we define the notion of *witness partition* for a role r, from which an object can be chosen to play the role r in transaction γ.

Definition 5. Witness partition *Let $\gamma \in \Gamma$ be a transaction and $\mathsf{cfg_a} \in \mathcal{C}_S^a$ be an abstract configuration. For a role $r = (p, \rho)$ of γ where r has the guard (Λ, Ψ), we say that a behavioral partition $beh = (s, q_1, \ldots, q_k, v)$ is a witness partition, denoted as $witness(r, \gamma, \mathsf{cfg_a})$, for r at $\mathsf{cfg_a}$ if*

1. *$s \xrightarrow{(\gamma_r)} s'$ is a transition in TS_p*
2. *For all $1 \leq i \leq k$, if $\mathcal{A}_i$ is the DFA corresponding to the regular expression of Λ, then q_i is an accepting state of $\mathcal{A}_i$.*
3. *$v \in Val(V_p)$ satisfies the propositional guard Ψ.*

4. *$count_p(b) \neq 0$, that is there is at least one object in this partition in the configuration* $\mathsf{cfg_a}$.

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

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

- *If $beh \in BEH_p$ is assigned as witness partition of n roles in γ, then $count_p(b) \geq n$. This ensures that one object does not play multiple roles in a transaction.*

The "destination partition" — the partition to which an object moves from its "witness partition" after executing a transaction — can be defined as follows. We denote the destination partition of *beh* w.r.t. to transaction γ and role r as $beh' = dest(beh, \gamma, r)$. Thus, an object in behavioral partition *beh* moves to partition $dest(beh, \gamma, r)$ by performing role $r = (p, \rho)$ in transaction γ.

Definition 7. Destination Partition *Let γ be an enabled transaction at an abstract configuration $\mathsf{cfg_a} \in \mathcal{C}_S^a$ and $beh = (s, q_1, \ldots, q_k, v)$ be the witness partition for the role $r = (p, \rho)$ of γ. Then we define $dest(beh, \gamma, r)$ — the* destination partition *of beh w.r.t. transaction γ and role r — as a behavioral partition $beh' = (s', q'_1, \ldots, q'_k, v')$, where*

- $s \xrightarrow{(\gamma_r)} s'$ *is a transition in* TS_p.

- *for all* $1 \leq i \leq k, q_i \xrightarrow{(\gamma_r)} q_i'$ *is a transition in DFA* $\mathcal{A}_i$.

- $v' \in Val(V_p)$ *is the effect of executing* γ_r *on* v.

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

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

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

Example Consider $TS_{Cruiser}$ shown in Figure 3-4(b). Suppose we simulate the specification with 24 *Cruiser* objects (assume that other process-classes are also ap-

propriately populated with objects) using abstract execution semantics. In the transition system $TS_{Cruiser}$, only the transition $noMoreDest_{StopRecvr}$ is guarded using a non-trivial regular expression $Act^{\star}_{Cruiser}.alertStop_{RcvDisEng}$; the corresponding DFA, say $\mathcal{A}_1$, will have just two states as can be easily verified. Initially all the 24 objects will be in the *stopped* state of $TS_{Cruiser}$ with null history and this will correspond to the initial state, say $q1$, of $\mathcal{A}_1$. All these objects are in the same behavioral partition $\langle stopped, q1 \rangle$, where we have suppressed the valuation component since there are no local variables associated with this class in this example. Suppose now a cruiser object, say O_1, executes (in cooperation with objects in other classes) the trace:

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

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

from behavioral partition (ii) to (i) above. Thus, O_1 will be now indistinguishable from the 22 objects which have remained idle throughout. For all of these 23 objects, the action $departReqA_{StartRecvr}$ is now enabled. This is the manner in which objects migrate between different behavioral partitions during abstract execution.

Maximum Number of Partitions We shall assume in what follows that the value domains of all the variables are finite sets. Thus, the number of behavioral partitions of a process class is finite. In fact, the number of partitions of a process class p is bounded by

$$|S_p| \times |Val(V_p)| \times \prod_{\mathcal{A} \in H_p} |\mathcal{A}|$$

where $|S_p|$ is the number of states of TS_p, $|Val(V_p)|$ is the number of all possible valuations of variables V_p, $|\mathcal{A}|$ is the number of states of automaton $\mathcal{A} \in H_p$. Recall that H_p is the set of minimal DFAs accepting the regular expression guards of the various roles of different transactions played by class p (Eq.(3.1)). Note that the maximum number of behavioral partitions does not depend on the number of objects in a class. In practice, many regular expression guards of transactions are vacuous leading to a small number of partitions. For example, the *Cruiser* class of the Rail-Car Example shown in Figure 3-4(b) can have at most 14 behavioral partitions since — (i) $TS_{Cruiser}$ has seven (7) states (not all of them are shown in Figure 3-4(b)), (ii) the Cruiser class has no local variables that is $V_{Cruiser} = \emptyset$ and (iii) only one of the regular expression guards involving a Cruiser object results in a DFA with two

states[2]; all other regular expression guards involving the Cruiser class are accepted by a single state DFA. Thus, the number of behavioral partitions of the Cruiser class is at most $7 * 2 = 14$ while the number of objects can be very large. In fact, in Section 3.7 we report experiments that the number of behavioral partitions encountered in actual abstract execution runs is often lower than the upper bound on number of partitions (48 Cruiser objects are divided into less than 6 partitions, see Table 3.2).

3.4.2 Dynamic Process Creation/Deletion

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

[2]This is the guard for the role $Cruiser_{StopRecvr}$ in transaction $noMoreDest$, see Figure 3-1(b).

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

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

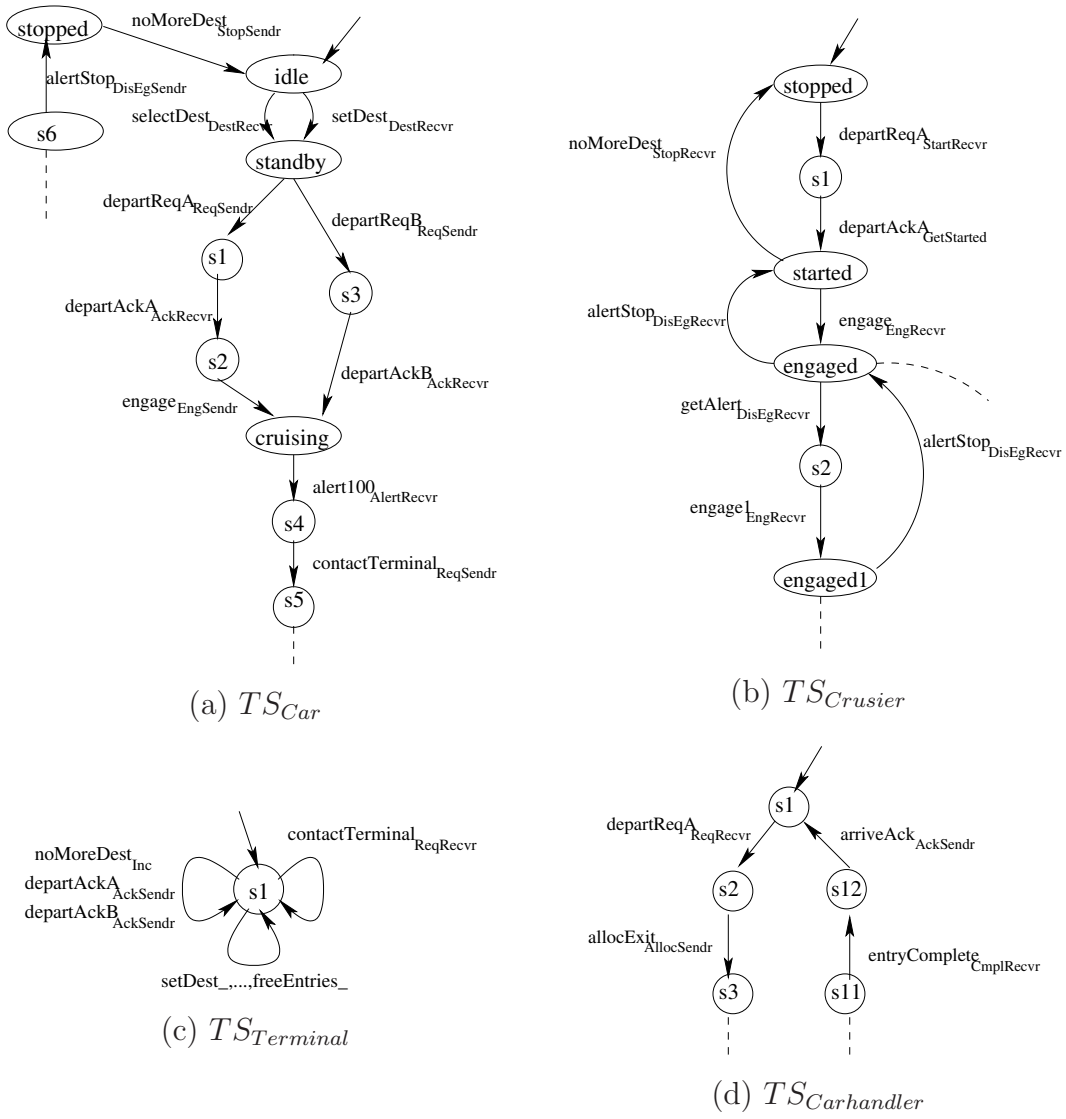


Figure 3-4: Fragment of Labeled Transition Systems for process classes of the Rail-car example — (a)Car (b)Cruiser (c)Terminal (d)Carhandler

3.5 Associations

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

3.5.1 Modeling Static and Dynamic Associations

Our notion of static and dynamic associations is similar to the classification presented in [108].

Static Associations A static association expresses a *structural relationship* between the classes. In a class-diagram, a static association is annotated with fixed multiplicities at both its ends. Static associations, as the name suggests, remain fixed and do not change at runtime. We can refer to static associations in transaction guards to impose the restriction that objects chosen for a given pair of agents should be statically related. The full class diagram for the Rail-car example with 24 cars appears in Figure 3-5. For example, the following pairs of classes: (PlatformManager, Terminal), (Terminal, ControlCenter), (Car, ControlCenter) and (Car, Cruiser) are statically associated in Figure 3-5. In particular, the association between Car class and the Cruiser class denotes the *itsCruiser* relation of a car with its cruiser. Note that we do not allow dynamic object-creation/deletion of a statically associated class–

such as Car, Terminal etc. in the Rail-car example.

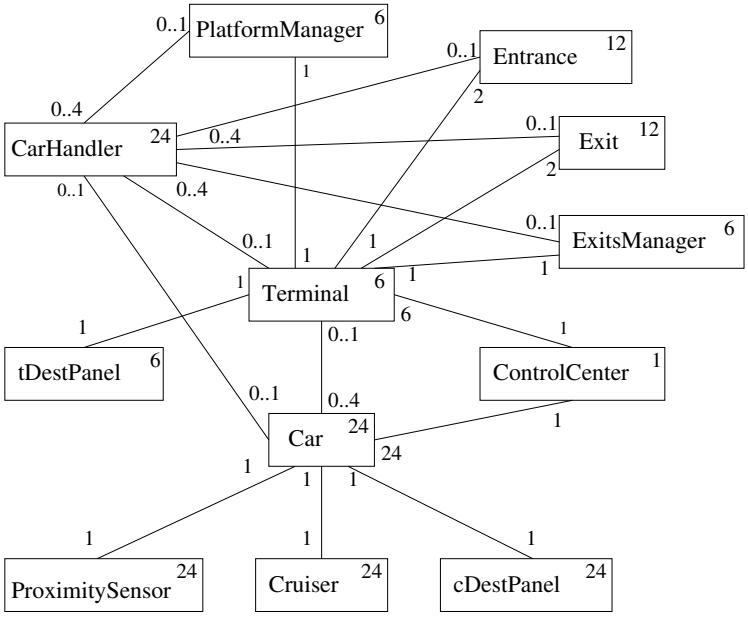


Figure 3-5: Class diagram for Rail-car example.

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

3.5.2 Concrete execution of IPC models with associations

In our concrete execution semantics, for a k-ary association *asc* relating objects of process classes $p_1, \ldots, p_k$, the k-tuple(s) of objects $\langle O_1, \ldots, O_k \rangle$ following the asso-

ciation are stored in relation *asc* during execution . We now describe the steps for handling an arbitrary association *asc* of arity k involving process classes $p_1, \ldots, p_k$ during concrete execution.

- **Initialization:** For a k-ary static association *asc*, the k-tuples of objects following the association are inserted in relation *asc*. For example, if $k = 2$ (binary association), and *asc* is a one-to-one binary association between two process classes p_1, p_2 each containing n objects $O_1, \ldots, O_n(O'_1, \ldots, O'_n)$, we populate *asc* with n object pairs, $\langle O_1, O'_1\rangle, \ldots, \langle O_n, O'_n\rangle$.

  If *asc* is a dynamic association, for convenience we assume that initially *asc* does not contain any k-tuple of objects. However, its content can change during execution.

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

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

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

Example We now illustrate the use of *dynamic associations* using the Rail-car example. During execution, various rail-cars enter and leave the terminals along their paths. When a car is approaching a terminal, it sends arrival request to that terminal by executing *contactTerminal* transaction and while leaving the terminal, its departure is acknowledged by the terminal by executing *departAckA* or *departAckB* transaction. Hence, the guard of *departAck(A/B)* requires that the participating *Car* and *Terminal* objects should have together executed *contactTerminal* in the past. Since this condition involves a relationship between the local histories of multiple objects, we cannot capture it via regular expressions over the individual local histories. Hence we make use of the dynamic relation *itsTerminal* between the *Car* and *Terminal* classes as part of our specification.

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

- *contactTerminal* has roles *(Car,ReqSendr)* and *(Terminal,ReqRecvr)*,

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

Figure 3-6: Dynamic Relation *itsTerminal*

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

In the concrete execution, if car object O_c and terminal object O_t play the roles (*Car,ReqSendr*) and (*Terminal,ReqRecvr*) in *contactTerminal*, then the effect of *contactTerminal* is to insert the pair $\langle O_c, O_t \rangle$ into the *itsTerminal* relation (refer to Figure 3-6). The *departAck(A/B)* transaction's guard now includes the check that the object corresponding to the role (*Car,AckRecvr*) and object corresponding to role (*Terminal,AckSendr*) be related by the dynamic relation *itsTerminal*; so if objects O_c and O_t are selected to play the (*Car,AckRecvr*) and (*Terminal,AckSendr*) roles in *departAck(A/B)*, the check will succeed. Furthermore, the effect of *departAck(A/B)* transaction is to remove the tuple $\langle O_c, O_t \rangle$ from *itsTerminal* relation.

Thus, for a dynamic relation, the specifications will include the effect of each transaction on the relation in terms of insertion/deletion of tuples of objects into the relation. Furthermore, the guard of a transaction can contain a membership

constraint ('check') on one or more of the specified static/dynamic relations. For execution of concrete objects, it is clear how our extended model should be executed. The question is how can we keep track of associations in the abstract execution semantics.

3.5.3 Abstract execution of IPC models with associations

For associations, the key question here is how we maintain relationships between objects if we do not keep track of the object identities. We do so by maintaining *associations between behavioral partitions.* To illustrate the idea, consider a binary relation D which is supposed to capture some dynamic association between objects of the process class p. In our abstract execution, each element of D will be a pair (b, b') where b and b' are behavioral partitions of class p; furthermore for all pairs $(b, b') \in D$ we also maintain a count indicating the number of concrete object pairs in behavioral partitions b, b' which are related via D. To understand what $(b, b') \in D$ means, consider the concrete execution of the process class p. If after an execution σ (a sequence of transactions), two concrete objects O, O' of p get D-related ($\langle O, O' \rangle \in D$) then the abstract execution along the same sequence of transactions σ must produce $\langle b, b' \rangle \in D$ where b (b') is the behavioral partition in which O (O') resides after executing σ. The same idea can be used to manage relations of larger arities. *Note that associations are maintained between behavioral partitions, but associations are not used to define behavioral partitions.* Hence there is no blow-up in the number of

behavioral partitions due to associations.

Formally, the set of abstract configurations (Definition 4, page 46) in our operational model remains unchanged. Recall from Definition 4 (page 46) that an abstract configuration is defined as $\mathsf{cfg} = \{count_p\}_{p \in \mathcal{P}}$ where $count_p(b)$ is the number of objects in partition $b \in BEH_p$; BEH_p is the set of all behavioral partitions of class p. In the presence of a k-ary association asc relating objects of process classes $p_1, \ldots, p_k$ we maintain asc in our abstract execution as

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

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

- **Initialization:** For each process class p we have an initial variable valuation v_p^{init} and an initial state $init_p$ in the high-level LTS of class p. Consequently, we can compute an initial behavioral partition beh_p^{init} for each process class p. Now, if asc is a static association, we initialize the counts of k-tuples of behavioral partitions in the following way. For every k-tuple other than $\langle beh_{p_1}^{init}, \ldots, beh_{p_k}^{init} \rangle$ we set the asc count to be zero. For the k-tuple $\langle beh_{p_1}^{init}, \ldots, beh_{p_k}^{init} \rangle$, the count is non-zero and is obtained from the class diagram annotations. For example, if

$k = 2$ (binary association), and *asc* is a one-to-one binary association between two process classes p_1, p_2 each containing n objects, we initialize the count for $\langle beh_{p_1}^{init}, beh_{p_2}^{init} \rangle$ to be n.

Now suppose *asc* is a dynamic association. For convenience we assume that initially *asc* does not contain any k-tuple of objects (which are in their initial state and have null histories). Consequently, the counts for *every* k-tuple of behavioral partitions is set to zero.

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

- **Insert:** Let γ be a transaction and $r_1, \ldots, r_k$ be the roles in transaction γ with the post-condition *insert* $(r_1, \ldots, r_k)$ *into asc*. Let $beh'_1, \ldots, beh'_k$ be the destination partitions of the roles $r_1, \ldots, r_k$ respectively, upon executing γ. We increment the *asc* count for the k-tuple $\langle beh'_1, \ldots, beh'_k \rangle$ by 1. Note that the insert operation is possible only if *asc* is a dynamic association.

- **Delete:** Let γ be a transaction and $r_1, \ldots, r_k$ be the roles in transaction γ with

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

- **Default:** Let γ be a transaction and BEH_γ be the set of witness partitions for the roles in γ. Let τ be a k-tuple in relation *asc* with association count greater than zero and BEH_τ represent the set of behavioral partitions it contains, such that, $BEH = BEH_\tau \cap BEH_\gamma \neq \emptyset$. Then for the k-tuple τ', obtained from k-tuple τ by replacing the partitions in BEH with the corresponding destination partitions upon executing γ, we increment its *asc* count by 1.

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

Example As discussed earlier, the dynamic relation *itsTerminal* is maintained between the objects of class *Car* and *Terminal* (as shown in Figure 3-6). This relationship is established between a *Car* and a *Terminal* object while executing *contactTer-*

minal and exists till the related pair executes either *departAckA* or *departAckB*. For illustration, suppose one object each from class *Car* and class *Terminal* plays the role (*Car,ReqSendr*) and (*Terminal,ReqRecvr*) respectively in the transaction *contactTerminal*. Let b_{Car} (b_{Term}) be the behavioral partitions in to which the objects of *Car* (*Terminal*) go by executing $contactTerminal_{ReqSendr}$ ($contactTerminal_{ReqRecvr}$). So in our abstract execution, corresponding to pairs of behavioral partitions of the Car and $Terminal$ class, we maintain a count indicating the number of pairs in the *itsTerminal* relation. Thus, for the pair $\langle b_{Car}, b_{Term} \rangle$ we increment its count by 1.

Now when we execute *departAck(A/B)* transaction, we will pick a pair from this relation as witness behavioral partitions for the roles (*Car,AckRecvr*) and (*Terminal,AckSendr*). We *have not* maintained information about which *Terminal* object in b_{Term} is related to which *Car* of b_{Car}. In our abstract execution, when we pick b_{Car} and b_{Term} as witness partitions of two roles in transaction *departAck(A/B)*, we are assuming that the corresponding objects of b_{Car} and b_{Term} which are associated via *itsTerminal* are being picked. Furthermore, after executing *departAck(A/B)* transaction, we decrement the count for pair $\langle b_{Car}, b_{Term} \rangle$ by 1.

3.6 Exactness of Abstract Semantics

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

for checking whether an abstract execution run is a spurious one.

3.6.1 Over-Approximation Results

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

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

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

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

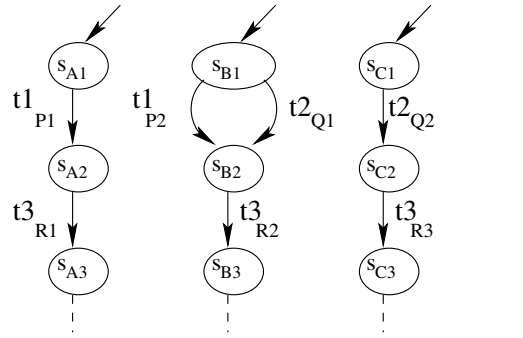
Theorem 2. *Suppose $\mathcal{S}$ is an IPC model which has no association relations appearing in the guards of any of the transactions. Then every finite sequence of transactions under the abstract execution semantics is also an execution sequence under the concrete execution semantics.*

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

Let σ be a finite sequence of transactions allowed in the abstract execution semantics of an IPC model $\mathcal{S}$. Let $beh \in BEH_p$ be a behavioral partition of class p with

count = n after the abstract execution of σ. Then σ is also a concrete execution. Furthermore, after the concrete execution of σ there exists exactly n concrete objects of class p which reside in partition beh based on their control state, execution history and variable valuation.

Let $\sigma = \sigma^{prev} \circ \gamma$ and the induction hypothesis holds for σ^{prev}. In the induction step, we need to show that the above holds after the execution of $\sigma = \sigma^{prev} \circ \gamma$ as well. Let $r_1, \ldots, r_m$ be the roles of transaction γ and let $beh_1, \ldots, beh_m$ be their witness behavioral partitions in the abstract execution of γ. By the induction hypothesis, we have concrete objects $o_1, \ldots, o_m$, whose states are given by the behavioral partitions $beh_1, \ldots, beh_m$, to play the roles $r_1, \ldots, r_m$ in the concrete execution of γ. Furthermore, if $beh'_1, \ldots, beh'_m$ are the destination partitions of $beh_1, \ldots, beh_m$ after the abstract execution of γ, we are guaranteed that $o_1, \ldots, o_m$ will move to $beh'_1, \ldots, beh'_m$ after the concrete execution of γ. This follows from- a) the definition of a destination partition, Def. 7, and b) the method for computing the behavioral partition representing the new state of an object in concrete execution semantics (Section 3.3). Now, in abstract execution, the object count of each of the witness (destination) partition $beh_i(beh'_i)$ will be decremented (incremented) by 1 after executing γ. Similarly, after the concrete execution of γ, the number of objects whose state is given by the behavioral partition $beh_i(beh'_i)$, will be decremented (incremented) by 1. Thus, the induction step is established. □



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

Figure 3-7: An example to show spurious runs in our abstract execution semantics

3.6.2 Spurious abstract executions

We now show that the converse of theorem 1 does not hold, i.e. *a finite sequence of transactions exhibited by the abstract execution of an IPC model $\mathcal{S}$ may not be exhibited by the concrete execution of $\mathcal{S}$*. Consider a fictitious system consisting of 3 process classes: *Cruiser*, *Car* and *BrakeControl*, such that each *Cruiser* and *BrakeControl* object is associated with a *Car* object via static associations Asc_1 and Asc_2. In other words, Asc_1 (Asc_2) captures the relationship between a car and *itsCruiser* (*itsBrakeController*). Fragments of the transition systems for these components are shown in Figure 3-7, along with the checks on the static associations by various transactions. Assume that there are no variables declared in these process classes and that all the action labels shown in the example have trivial guards, that is they do not

impose any restriction on the execution history of the object to play that role (of course the object should be in the appropriate control state). Suppose now, that we have an initial abstract configuration

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

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

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

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

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

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

Since the car object executing $t1$ (call it Car1 for convenience of explanation) is

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

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

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

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

In the concrete execution, however $t3$ cannot be executed after transactions $t1, t2$ are executed. After executing transactions $t1, t2$ there *cannot be any concrete car object* which has its cruiser (related by association Asc_1) as well its brake controller (related by association Asc_2) in the appropriate control states for executing transac-

tion $t3$. Thus, if trace σ is simulated in the concrete execution, it will get *deadlocked* after executing the transactions $t1, t2$. Though in this example we have only considered static associations, similar incompleteness of our abstract execution can be shown with dynamic associations.

3.6.3 Detecting spurious abstract executions

Detecting spurious abstract executions is similar in objective to detecting spurious counter-example traces in abstraction-refinement based software model checking (e.g. see [54]). In our setting, this can be done effectively.

Theorem 3. *There is an effective procedure which accepts as input an IPC* $S = \{TS_p\}_{p \in \mathcal{P}}$ *and a finite sequence* σ *which is an execution sequence under the abstract execution semantics, and determines whether or not* σ *is a spurious execution sequence; in other words,* σ *is not an execution sequence under the concrete semantics.*

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

For each process class p, let $num_{p,\sigma}$ denote an upper-bound on the number of p-objects required for exhibiting the execution sequence σ. We define $num_{p,\sigma}$ to be the total number of roles (p, ρ) appearing in transaction sequence σ s.t. for each such role (p, ρ) in a transaction occurrence γ^i in σ, γ^i_ρ is an outgoing transition from the initial state of TS_p (the transition system for process class p). This is because,

the number of unique p-objects that can participate in transactions in σ, must have initially executed a role (p, ρ) in a transaction γ^i (occurring in σ) s.t. γ^i_ρ is an outgoing transition from the initial state of TS_p. Note that, a transaction γ can occur more than once in an execution trace σ. For computing $num_{p,\sigma}$, we treat each transaction occurrence as distinct.

We now define $x_{p,\sigma} = min(N_p, num_{p,\sigma})$ if N_p, the number of objects in p is a given constant. Otherwise the number of objects of p is not fixed and we set $x_{p,\sigma} = num_{p,\sigma}$. It is worth noting that $x_{p,\sigma}$ serves as a cutoff on the number of objects of class p only for the purpose of exhibiting the behavior σ and not all the behaviors of the system. For the execution trace σ, we can say that σ is a concrete run in the given system iff it is a concrete run in the *finite state* system where each process class p has $x_{p,\sigma}$ objects. To show this we consider the following two cases:

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

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

more transactions of trace σ, even if the system has more than $num_{p,\sigma}$ objects of class p.

□

Using an illustrative example, we now show how a spurious run is detected for a given IPC system $\mathcal{S}$ and a trace σ, by first deriving a *finite state* system from $\mathcal{S}$ corresponding to σ. Again, we consider the example discussed in Section 3.6.2 (Figure 3-7), consisting of three process classes- *Cruiser*, *Car* and *BrakeControl*. Also, consider the trace $\sigma = t_1.t_2.t_3$ which can be exhibited in the given system following our abstract execution semantics, as was demonstrated in Section 3.6.2.

Suppose we now want to check whether or not σ is spurious. From Figure 3-7 we obtain the roles for transactions in σ– for t_1, roles are (*Cruiser,P1*) and (*Car,P2*), for t_2, roles are (*Car,Q1*) and (*BrakeControl,Q2*), and roles in transaction t_3 are (*Cruiser,R1*), (*Car,R2*) and (*BrakeControl,R3*). We now compute $num_{p,\sigma}$ for each process class in the system. First, we consider the $Cruiser$ class. It participates in transactions t_1 and t_3 in σ, playing the roles (*Cruiser,P1*) and (*Cruiser,R1*) in these transactions respectively. Only one of these roles, (*Cruiser,P1*) in transaction t_1, is played from the initial state s_{A1} of the transition system describing $Cruiser$ (Figure 3-7). Thus, $num_{Cruiser,\sigma} = 1$. Similarly we can determine that $num_{Car,\sigma} = 2$ and $num_{BrakeControl,\sigma} = 1$. As $N_{Cruiser} = N_{Car} = N_{BrakeController} = 2$ in the given system $\mathcal{S}$, we get $x_{Cruiser,\sigma} = 1$, $x_{Car,\sigma} = 2$ and $x_{BrakeControl,\sigma} = 1$. Now to detect whether or not σ is spurious in the given system $\mathcal{S}$, we only need to check if σ is a

valid execution trace of the finite state system obtained above. Note that we have already shown σ to be spurious for the given system $\mathcal{S}$ in Section 3.6.2. Following the similar reasoning, σ can easily be shown spurious for this finite state system as well, leading to a *deadlock* after the execution of transactions t_1, t_2.

We have implemented the above procedure using the Murphi model checker [83]. This model checker has in-built support for symmetry reduction [59] which can be exploited in the IPC setting. We present the details of the implementation in Appendix A.2.

3.7 Experiments

We have implemented our abstract execution method by building a simulator in *OCaml* [85], a general purpose programming language supporting functional, imperative and object-oriented programming styles.

3.7.1 Modeled Examples

For our initial experiments, we modeled a simple telephone switch drawn from [55]. It consists of a network of switch objects with the network topology showing the connection between different geographical localities. Switch objects in a locality are connected to phones in that locality as well as to other switches as dictated by the network topology. We modeled basic features such as local/remote calling as well as advanced features like call-waiting. Next we modeled the rail-car system whose

behavioral requirements have been specified using Statecharts in [49] and using Live Sequence Charts in [27]. As mentioned in Section 3.2, Rail-Car system is a substantial sized system with a number of process classes: car, terminal, cruiser (for maintaining speed of a rail-car), car-handler (a temporary interface between a car and a terminal while a car is in that terminal), etc.

We have also modeled the requirement specification of two other systems - one drawn from the rail transportation domain and another taken from air traffic control (see `http://scesm04.upb.de/case-studies.html` for more details of these examples). We now briefly describe these two systems. The automated rail-shuttle system [94] consists of various shuttles which bid for orders to transport passengers between various stations on a railway-interconnection network. The successful bidder needs to complete the order in a given time, for which it gets the payment as specified in the bid; the shuttle needs to pay the toll for the part of network it travels. If an order is delayed or not started in time, a pre-specified penalty is incurred by the responsible shuttle. A part of network may be disabled some times due to repair work, causing shuttles to take longer routes. A shuttle may need maintenance after traveling a specified distance, for which it has to pay. Also, in case a shuttle is bankrupt (due to payment of fines), it is retired. The weather update controller [26] is a an important component of the *Center TRACON Automation System*, automation tools developed by NASA to manage high volume of arrival air traffic at large airports. The case study involves three classes of objects: weather-aware clients, weather control panel and the

Example	Process Class	# Concrete Objects	# of partitions in Test Case		
			I	II	III
Telephone Switch	Phone	60	9	9	7
	Switch	30	9	9	9
Rail-Car Example	Car	48	12	10	11
	CarHandler	48	3	8	8
	Terminal	6	6	6	6
	Platform Mngr.	6	1	3	3
	Exits Mngr.	6	1	2	2
	Entrance	12	2	1	2
	Exit	12	1	2	2
	Cruiser	48	1	3	5
	Proximity Sensor	48	1	1	2
	cDestPanel	48	1	1	1
	tDestPanel	6	1	1	1
Automated Shuttle	Shuttle Agent	60	6	5	6
Weather Update	Clients	20	3	3	3

Table 3.2: Maximum Number of Behavioral partitions observed during abstract simulation

controller or communications manager. The latest weather update is presented by the weather control panel to various connected clients, via the controller. This update may succeed or fail in different ways; furthermore, clients get connected/disconnected to the controller by following an elaborate protocol.

3.7.2 Use Cases

We used guided abstract simulation on each of our examples to test out the prominent use cases. The details of these experiments appear in Table 3.2. For the Telephone

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

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

For each test case, we report the object count for each process class as well as the maximum number of behavioral partitions observed during simulation. We have reported the results for only process classes with more than one object. Since we are

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

C ≡ Concrete Exec., A ≡ Abstract Exec.

Table 3.3: Timing/Memory Overheads of Concrete Execution and Abstract Execution

simulating reactive systems, we had to stop the simulation at some point; for each test case, we let the simulation run for 100 transactions – long enough to exhibit the test case's behavior. From Table 3.2, we can see that the number of behavioral partitions is substantially less than the number of concrete objects. Furthermore, even if the number of concrete objects is increased (say instead of 48 cars in the Rail-car example, we have 96 cars), the number of behavioral partitions in these simulation runs remain the same.

3.7.3 Timing and Memory Overheads

Since one of our main aims is to achieve a simulation strategy efficient in both time and memory, a possible concern is whether the management of behavioral partitions introduces unacceptable timing and memory overheads. We measured timing and memory usage of several randomly generated simulation runs of length 1000(i.e. con-

taining 1000 transactions) in our examples and considered the maximum resource usage for each example. We also compared our results with a concrete simulator (where each concrete object's state is maintained separately). For meaningful comparison, the concrete simulator is also implemented in OCaml and shares as much code as possible with our abstract simulator. Simulations were run on a Pentium-IV 3 GHz machine with 1 GB of main memory. The results are shown in Table 3.3. For each example we show the time and memory usage for both the abstract and concrete simulation. Also, for a given example, we obtained results for two different settings, where the second setting was obtained by doubling the number of objects in one or more of the classes, e.g. in the *rail-car* example with 24 and 48 cars respectively.

We observe that for a given example and a given number of objects, the running time and memory usage for the concrete simulator are higher than that for the abstract simulator. Also for the same example but with higher number of objects, in case of abstract execution, the time/memory remain roughly the same, whereas they increase for the concrete case (as indicated by the increase in ratio C/A for higher number of objects in Table 3.3).

Furthermore, in the graphs shown in Figure 3-8, we compare the growth in timing and memory usage in the railcar example, for both concrete and abstract simulations. Each successive setting is obtained by increasing the number of cars and its associated components: "car-handler", "proximity-sensor", "cruiser" and "dest-panel" by 24. Clearly our abstract execution allows the designer to try out different settings

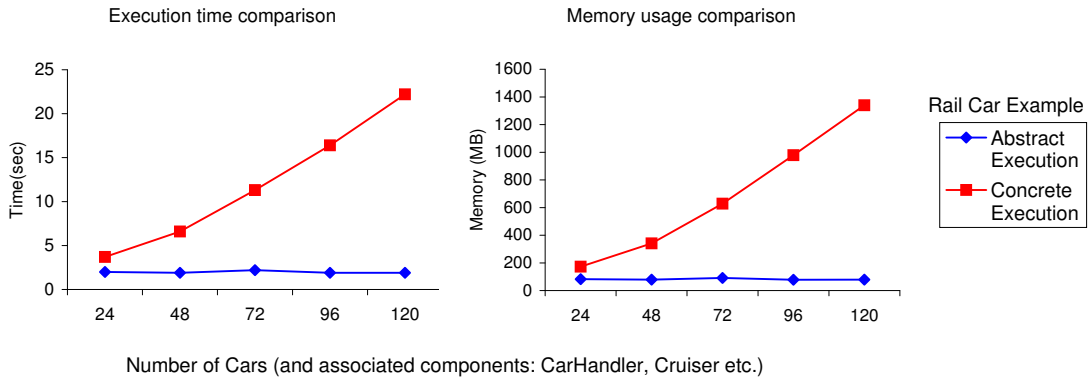


Figure 3-8: Execution Time and Memory Usage for different settings of the RailCar example.

of a model by varying the number of objects without worrying about time/space overheads.

3.7.4 Checking for spurious execution runs

Recall that in the presence of associations, our abstract execution semantics is sound but incomplete. Consequently, we may encounter execution runs which are "spurious", that is, do not appear in any concrete system execution. In Section 3.6.3, we have presented a decision procedure for checking whether a finite sequence of transactions produced by our abstract simulator is spurious. As mentioned, we used the Murphi model checker to implement this spuriousness check.

During our experiments, we found that the spuriousness check for all the test cases of all our examples was completed in less than 0.1 second using Murphi. Also, when simulating an example system against meaningful use cases, the execution run

produced by our abstract simulator was typically not spurious. In fact, there was only one false positive among all the test cases we tried for all the examples. This is to be expected, since we use our simulator to try out meaningful/prominent use-cases for a given system specification in the IPC model.

3.7.5 Debugging Experience

In this section, we share some experiences in reactive system debugging gained using our simulator tool. In particular, we describe our experiences in debugging the weather-update control system [26]. The weather-update control system consists of three process classes: the communications manager (call it CM), the weather control panel (call it WCP) and Clients. Both CM and WCP have only one object, while the Client class has many objects. In Figure 3-9, we show a snippet of the transition system for CM. Even for the snippet shown in Figure 3-9, the transition system shown is a slightly simplified version of our actual modeling. We have given the transactions names to ease understanding, for example *Snd_Init_Wthr* stands for "send initial weather" and so on.

We now discuss two bugs that we detected via simulation. The first one is an under-specification in the informal requirements document for the weather-update controller. In Figure 3-9, the controller CM initially connects to one or more clients by executing the transactions *Connect* and *Snd_Init_Wthr*. In the *Connect* transaction CM disables the Weather Control Panel (WCP). If the client subsequently reports that

that it did not receive the weather information (*i.e.* transaction *Not_Rcv_Init_Wthr* is executed), CM goes back to *Idle* state without re-enabling the Weather Control Panel (WCP). Hence no more weather-updates are possible at this stage. This results from an important under-specification of the weather-update controller's informal requirements document. This error came up in a natural way during our initial experiments involving random simulation. Simulation runs executing the sequence of transactions

Connect,Snd_Init_Wthr,Not_Rcv_Init_Wthr,Upd_from_WCP

got stuck and aborted as a result of which the simulator complained and provided the above sequence of transactions to us. From this sequence, we could easily fix the bug by finding out why *Upd_from_WCP* cannot be executed (*i.e.* the Weather Control Panel not being enabled). We note that since the above sequence constitutes a meaningful use-case we would have located the bug during guided simulation, even if it did not appear during random simulation. In this context it is worthwhile to mention that for every example, after modeling we ran random simulation followed by guided simulation of prominent test cases.

We found another bug during guided simulation of the test case where connected clients get disconnected from the controller CM since they cannot use the latest weather information. This corresponds to the connected clients executing the *Disconnect* transaction with the CM, and the CM returning from *Done2* to *Idle* by executing *Enable_WCP* (Figure 3-9). For this simulation run, even after all clients are

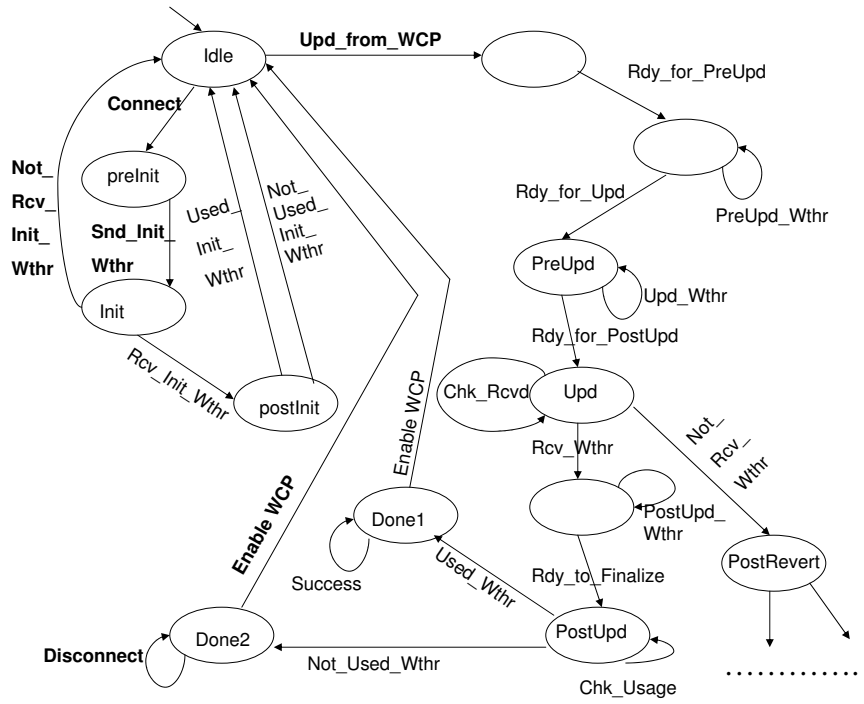


Figure 3-9: Snippet of Transition System for Weather-Update Controller

disconnected, the CM executes *Upd_from_WCP* (update from Weather Control Panel) followed by *Rdy_for_PreUpd* (ready for pre-update). The simulator then gets stuck at the *PreUpd_Wthr* (pre-update weather) transaction since there are no connected clients. From this run, we found a missing corner case in the guard for *Upd_from_WCP* transaction – no weather updates should take place if there are no connected clients. In this case, it was a bug in our modeling which was detected via simulation.

Currently, our simulator supports the following features to help error detection.

- Random simulation for a fixed number of transactions
- Guided simulation for a use-case (the entire sequence of transactions to be executed need not be given)

- Testing whether a given sequence of transactions is an allowed behavior.

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

3.8 Discussion

In this chapter, we have studied a modeling formalism accompanied by an execution technique for dealing with interacting process classes; such systems arise in a number of application domains such as telecommunications and transportation. Our models are based on standard notations for capturing behaviors and our abstract execution strategy allows efficient simulation of realistic designs with large number of objects. The feasibility of our method has been demonstrated on realistic examples.

The notion of "roles" played by processes in protocols have appeared in other contexts (*e.g.* [105]). Object orientation based on the actor-paradigm has been studied thoroughly in [74]. We see this work approach as an orthogonal one where the computational rather than the control flow features are encapsulated using classes and other object-oriented programming notions such as inheritance.

We note that, our model can be easily cast in the setting of Colored Petri nets [65] with our operational semantics translating into an appropriate token game. We feel however our formulation is simpler and better structured in terms of a network of communicating transition systems.

Chapter 4

Symbolic Message Sequence Charts (SMSC)

Message Sequence Charts (MSCs) are widely used by requirements engineers in the early stages of reactive system design. Conventionally, MSCs are used in the system requirements document to describe *scenarios* — possible ways in which the objects constituting a distributed reactive system may communicate among each other as well as with the environment. MSCs may be composed to yield complete behavioral descriptions. Such compositions may be captured as a High Level Message Sequence Chart or HMSC. The complete MSC language appears in a recommendation of the ITU [62]. The syntax and process theory based operational semantics of the language have also been outlined in [98].

The benefits of MSCs notwithstanding, it has been observed that while describing

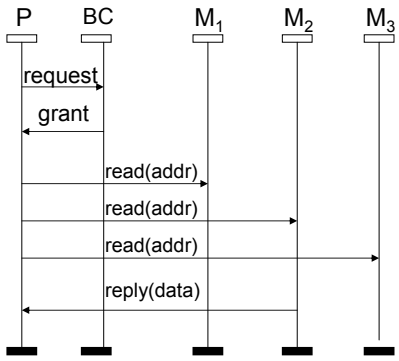


Figure 4-1: An MSC showing read request from a processor to various memory devices via a bus. The bus controller (BC) controls access to the bus.

requirements of systems containing many objects, MSC specifications tend to grow too large for human comprehension [98, 109]. We find this problem to be particularly acute when the system contains several objects which conform to a common behavioral protocol when interacting with other objects. Such objects may be considered as instances of a common process *class*. In the absence of suitable abstraction mechanisms, similar interactions involving different objects from the same class have to be repeated to convey all possible scenarios that may occur, thereby leading to voluminous MSC specifications.

Let us consider an example to illustrate this point. Consider a master-slave protocol interaction, where several master processes are competing to get service from the slave processes. An arbiter controls access to the slaves. Furthermore, whenever any master needs service and the arbiter grants access to the slaves, a specific slave will be allocated depending on the kind of service needed. A concrete realization of such

an interaction can be observed in bus access protocols. The master processes are the processors hooked to the bus. The slave processes are memory devices from which the processors are trying to read or write. The arbiter is the bus controller which decides, according to some scheduling policy, the processor to be granted bus access. Usually when a processor needs to access a memory address for reading/writing data, it broadcasts the appropriate address and control signals over the bus to various memory devices. Then after decoding the address, one of the devices will respond to the processor's read/write request.

Figure 4-1 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 4-1. Instead, the processor sends a read request separately to each of the memory units. In the MSC shown in Figure 4-1, the read request is for an address which is from the address space of memory unit M_2. Since there are two other memory devices, we would then need to repeat the same interaction to convey the cases when the read request is in the address space of M_1 or M_3.

Clearly, as we increase the number of memory devices and processors in the system, such an approach will not scale up. Individual MSCs will become large, containing many lifelines, and similar MSCs representing essentially the same interaction will

have to be repeated. Moreover, since a lifeline may appear in several MSCs, modifying the specification (for example by adding or removing memory devices) may be error-prone and will involve considerable effort. Finally, validation of such specifications will become inefficient as the number of memory devices and processors is increased.

Technical Contributions To address various shortcomings of the MSCs as discussed in the preceding, we introduce simple yet powerful extensions to the MSC language to support efficient specification and validation of systems involving classes of behaviorally similar objects. Our extensions are based on the meaning of a lifeline in an MSC. The MSC standard (now integrated into UML 2.0) suggests that a lifeline denotes a concrete object. As the above example indicates, this may lead to voluminous behavioral descriptions that scale poorly. In our extension, we first relax the meaning of a lifeline to consider three possibilities (i) a concrete object (ii) any k objects from a class for a given positive constant k (*existential abstraction*), or (iii) all objects from a class (*universal abstraction*). Moreover, *guards* may be used to further restrict the object(s) that may engage in the events depicted on the lifeline. Thus, if we have a universally abstracted MSC lifeline drawn from process class p with guard g_p, it will be played by all object(s) of class p which satisfy g_p. These extensions yield a concise MSC notation called Symbolic Message Sequence Chart (SMSC) [101].

Further, we note that the *inter-class associations* appear quite commonly in an object-oriented system description, and are used to convey structural information regarding static/dynamic relationships among various communicating processes. How-

ever, information regarding associations is not made explicit in MSCs. Similar to the case of IPC (Chapter 3), while modeling a distributed protocol using scenarios, a designer may want to explicitly specify establishing/removing of association links, and moreover, constrain the communicating processes to be linked via specific associations. We address this issue by introducing constraints in our SMSC modeling notation for- (i) *inserting* object pairs in a given association, (ii) *checking* that only object pairs related via a given association communicate with each other, / and (ii) *removing* existing association links between various object pairs.

In the following, we first present the syntax, concrete semantics, and abstract semantics of SMSCs without associations. The abstract semantics allows for efficient symbolic simulation of SMSCs showing interactions among process classes with large (or even an *unbounded*) numbers of objects. *It also serves as a formal execution semantics which can be used to reason about interactions between an unbounded number of objects.* We then present experimental results, discussing SMSC modeling and execution of a real-life controller- the CTAS weather controller [26] from NASA, which is part of a control system for managing high volume air traffic. Experimental results obtained from the CTAS controller allow us to better understand the issues in modeling, analyzing and debugging real-life control systems involving structurally similar interactions among many objects. Finally, we introduce associations in our SMSC modeling framework and discuss in detail an abstract SMSC execution semantics in the presence of associations.

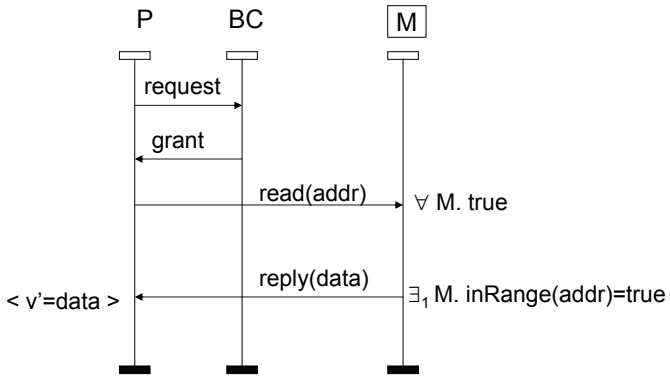


Figure 4-2: A Symbolic MSC

4.1 Syntax

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

4.1.1 Visual Syntax

Graphically, we represent SMSCs as in Figure 4-2. This SMSC depicts processor-memory interaction, corresponding to the MSC discussed earlier in Figure 4-1. One important difference between these representations is that the three concrete memory lifelines M_1, M_2 and M_3 appearing in Figure 4-1, have been merged into a single symbolic lifeline in Figure 4-2, 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 4-2. 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 4-2.

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

We use the shorthand $\exists_k$ to denote the existential abstraction of lifelines. This is to emphasize that exactly k objects will play such a lifeline. Clearly $k \geq 1$. *For the examples in this paper, we have only used* $\exists_1$ *(i.e. selecting one object) whenever we used existential abstraction. Henceforth we always assume* $\exists_1$ *since the extension of our semantics to the general case is trivial. We mention these extensions via footnotes when we present our semantics.* Moreover, in Section 4.8, we mention other variations of existential abstraction which can be incorporated with minimal modifications to our modeling language and its semantics.

In Figure 4-2, 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 4-1 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 [72]. 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 4-2 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 4-2, 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 4-2 since *for any variable x we show its updated value by its primed version x'.*

4.1.2 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

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

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. 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 $\mathcal{V}_p$ be its associated set of variables with function v_p^{init} giving the initial assignment of values to the objects of p. For convenience we assume that all the objects of class p assign the same initial value to any variable $u \in \mathcal{V}_p$.[2] Then, an assignment statement appearing in a postcondition corresponding to an event from class p is of the form $x' = aexpr$, where (i) $x \in \mathcal{V}_p$, and (ii) $aexpr$ represents an arithmetic expression involving variables from $\mathcal{V}_p$, integer constants, and the following arithmetic operators '$+, -, /, \times$'. Let $Post^p$ represent the set of postconditions for class $p \in \mathcal{P}$. We define $Post^{\mathcal{P}} = \bigcup_{p \in \mathcal{P}} Post^p \cup \{\epsilon\}$, where ϵ represents an empty postcondition.

We now define the sets of message send events A_{out}, receive events A_{in}, and local action events A_{act}, that are needed for defining the set A^{SMSC} of atomic actions in SMSCs.

Definition 8. *Let $\mathcal{P}$ denote the set of all process classes with $p, q \in \mathcal{P}$. Let $\mathcal{M}$ and $\mathcal{L}$ denote the set of all message names and local action names, respectively. Then the*

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

sets A_{out}, A_{in}, A_{act} and A^{SMSC} are defined as follows:

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

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

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

Definition 9 (SMSC)**.** *A SMSC sm can be viewed as a partially ordered set $sm = (L, E_L, \leq)$, where L is the set of lifelines in sm, $E_L = \bigcup_{l \in L} E_l$ where $E_l \subseteq A^{SMSC}$ is the set of events lifeline l takes part in sm, and $\leq$ is the partial ordering relation over the occurrences of events in E_L, such that $\leq = (\leq_L \cup \leq_{sm})^{\star}$ is the transitive closure*

of $\leq_L$ *and* $\leq_{sm}$*, where*

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

(b) $\leq_{sm}$ *is an ordering on message send/receive events in* E_L*. If* $e_s = \mathsf{out}(os_i, os_j, m, pc_1)$ *and the corresponding receive event is* $e_r = \mathsf{in}(os_i, os_j, m, pc_2)$*, we have* $e_s \leq_{sm} e_r$*.*

In our system model, SMSCs may be composed together to yield High-level SMSCs (or HSMSCs) in the same way that MSCs may be arranged in High-level MSCs (HMSCs). A HSMSC is essentially a directed graph whose each node is either a SMSC or (hierarchically) a HSMSC. Formally, a HSMSC is a tuple $H = (N, B, v^I, v^T, l, E)$ where (i) N is a finite set of nodes (ii) B is a finite set of boxes (or supernodes) representing (already defined) HSMSCs (iii) $v^I \in N \cup B$ is the initial node or box (iv) $v^T \in N \cup B$ is the terminal node or box (v) l is a labeling function that maps each node in N to a SMSC and each box in B to another HSMSC (vi) $E \subseteq (N \cup B) \times (N \cup B)$ is the set of edges that describe control flow.

We now define the system specification S.

Definition 10 (System)**.** *Given process classes* $\mathcal{P}$*, the system specification* $S = \langle H, \bigcup_{p \in \mathcal{P}} \{\mathcal{V}_p, v_p^{init}\} \rangle$ *where* $H = (N, B, v^I, v^T, l, E)$ *is a HSMSC describing the interactions among objects from process classes in* $\mathcal{P}$*.*

Expression Template In order to make specifications more readable, we identify a commonly occurring regular expression template that we have encountered in our modeling. Consider a process class $p \in \mathcal{P}$ and $e_1, e_2 \in A_p$. For regular expressions of the form $h_1 = A_p^\star.e_1$ we write it as $h_1 = last(e_1)$.

4.2 CTAS Case Study

We now discuss a well known example to illustrate system modeling using SMSCs. The weather update controller [26] 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 4-3. Various nodes of this HSMSC are labeled with the SMSC names. In the CTAS requirements document [1], the requirements are given from the viewpoint of the CM. All the clients are initially *disconnected* from the controller (CM) and the execution sequence -*2.6.2, 2.8.3, 2.8.5, 2.8.8*- taken from the requirements document, forms the scenario in which a client successfully gets connected to

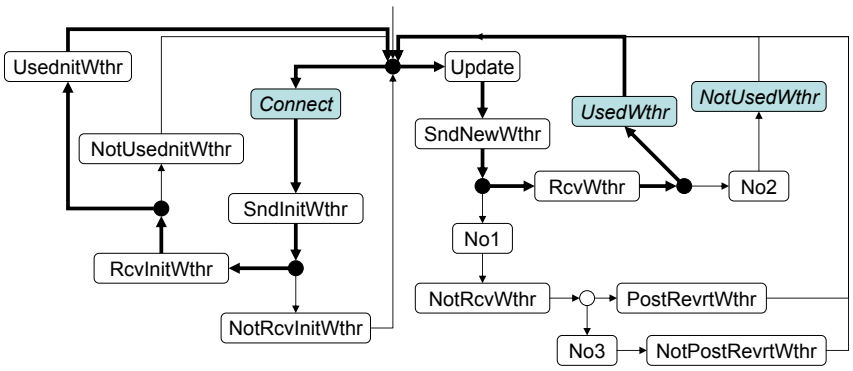


Figure 4-3: HSMSC for the CTAS case study. The *Connect*, *UsedWthr* and *NotUsedWthr* SMSCs are shown in Figure 4-4, 4-5.

CM. This behavior corresponds to the execution of the left loop (marked using bold lines) in Figure 4-3, such that the four SMSCs along this marked path correspond to the above four requirements. For example, SMSC *Connect* (shown in Figure 4-4) 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

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

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 4-4. This guard imposes the constraint that either a fresh Client object (having no execution history, and is therefore disconnected), or a Client object which has last been disconnected from CM (due to the receipt of *close* message) can send the 'connect' request to the CM. For the subsequent event of receiving *setStatus_1* message from the CM, the history based guard is $h = last(e_2)$, where $e_2 = \mathsf{out}(Client, CM, connect)$, which allows only the object(s) which have recently sent 'connect' message to CM to execute this event. Note that in specifying the regular expression guards, we have used the template expression $last(e)$ described earlier in Section 4.1.2.

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 4-3 (marked using bold lines) [4]. In case any client either fails to use the original data (in case it has failed to receive the new data), or update itself (having received the new weather information), all the connected clients get disconnected.

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

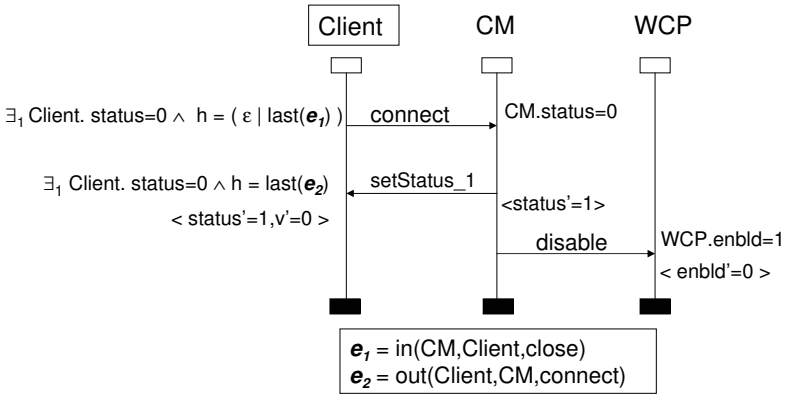


Figure 4-4: Connect SMSC from CTAS.

These latter scenarios correspond to other execution paths in the CTAS HSMSC.

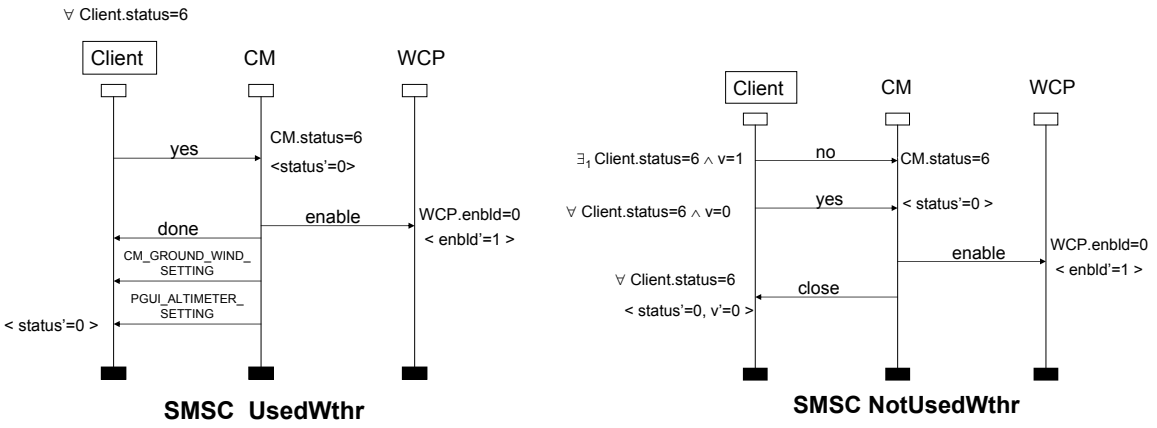


Figure 4-5: CTAS SMSCs showing successful and unsuccessful completion of weather update.

The two SMSCs shown in Figure 4-5 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.

The scenario shown in *NotUsedWthr* SMSC in Figure 4-5 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.*

4.3 Process Theory

The semantics of the ITU MSC language is defined using a process theory [98]. 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 [98] for details).

4.3.1 Configurations and Concrete Semantics

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

In a **concrete execution semantics**, a configuration captures the "local states" of all objects of all process classes. The question is, in a scenario-based modeling language like SMSC, how do we capture the local state of an object. Clearly, we need (a) the object's control flow (which SMSC it is currently executing and which events in the current SMSC have been executed), (b) the variable valuations of the object and (c) a bounded representation of the (unbounded) history of events which allows us to test the satisfaction of history-based regular expression guards in the specification. Here we note that the control flow of objects will be captured by terms in our process theory. The variable valuations will be explicitly captured. Finally, the history information for an object o can be captured by representing the regular expression guards as minimal DFAs and then maintaining the states in these DFAs in which object o lies. Maintaining such information for each individual object for each

Table 4.1: 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 4.2: 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. $\dfrac{C : x \xrightarrow{a} C' : x', C : y \xrightarrow{a} C' : y'}{C : x \mp y \xrightarrow{a} C' : x' \mp y'}$	

process class will give a notion of concrete configurations, and the transition between these concrete configurations (using the semantic rules presented below) provides a straightforward concrete execution semantics. However, we will later develop *abstract configurations* to enable (i) efficient simulation of systems with finite number of objects, and (ii) reasoning about systems with infinitely many objects. For this reason, at this stage, we do not impose any concrete structure on configuration C.

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

4.3.2 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 $\frac{H}{Concl}$, where H is a set of premises and $Concl$ is the conclusion. For ITU MSCs, the semantics for constants consists of two rules (i) $\frac{}{\epsilon \downarrow}$, which implies that the empty process may terminate successfully and immediately; there are no other rules for ϵ as it is unable to perform any action (ii) $\frac{}{a \xrightarrow{a} \epsilon}$, which expresses that a process represented by the atomic action a can perform a and thereby evolve into the empty process ϵ. The rules for constants in our theory are presented in Table 4.1. 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 4.2. Thus $x \mp y$ can terminate if either x (DC1) or y (DC2) is able to terminate. Except for the addition of the configuration element for the execution semantics rules, the rules are similar in spirit to the MSC rules for $\mp$. If $C : x$ can execute a and $C : y$ cannot (DC3), then on execution of a, the choice is resolved in favor of x. DC4 shows the complementary case when $C : y$ executes a, and $C : x$ cannot. Finally, if both $C : x$ and $C : y$ are able to execute a, then the execution of a does not resolve the choice, but rather, delays it (DC5). Semantic rules for the other operators may be defined similarly (see [98]).

To account for configurations, we also extend the *bisimilarity* based term equality definition in the MSC standard [98]. 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 [98]. We do not reproduce the details here, but use $\hookrightarrow$ for completeness of our modified term equality definition below.

Definition 11 (Bisimulation)**.** *Let $\mathcal{C}$ be the set of all possible configurations. A binary relation $B^{\mathcal{C}} \subseteq \mathcal{CT}(\Sigma) \times \mathcal{CT}(\Sigma)$ is called a bisimulation relation iff for all $a \in Act$ and $s, t \in \mathcal{CT}(\Sigma)$ with $sB^{\mathcal{C}}t$, the following conditions hold*

$$\forall_{s' \in \mathcal{CT}(\Sigma), C, C' \in \mathcal{C}} (C : s \xrightarrow{a} C' : s' \quad \Rightarrow \quad \exists_{t' \in \mathcal{CT}(\Sigma)} (C : t \xrightarrow{a} C' : t' \wedge s' B^{\mathcal{C}} t')) \tag{4.1}$$

$$\forall_{t' \in \mathcal{CT}(\Sigma), C, C' \in \mathcal{C}} (C : t \xrightarrow{a} C' : t' \quad \Rightarrow \quad \exists_{s' \in \mathcal{CT}(\Sigma)} (C : s \xrightarrow{a} C' : s' \wedge s' B^{\mathcal{C}} t')) \tag{4.2}$$

$$\forall_{s' \in \mathcal{CT}(\Sigma)} (s \overset{a}{\hookrightarrow} s' \quad \Rightarrow \quad \exists_{t' \in \mathcal{CT}(\Sigma)} (t \overset{a}{\hookrightarrow} t' \wedge s' B^{\mathcal{C}} t')) \tag{4.3}$$

$$\forall_{t' \in \mathcal{CT}(\Sigma)} (t \overset{a}{\hookrightarrow} t' \quad \Rightarrow \quad \exists_{s' \in \mathcal{CT}(\Sigma)} (s \overset{a}{\hookrightarrow} s' \wedge s' B^{\mathcal{C}} t')) \tag{4.4}$$

$$s \downarrow \iff t \downarrow \tag{4.5}$$

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 $B^{\mathcal{C}}$ such that $pB^{\mathcal{C}}q$. Note that even though we carry state information in the form of configuration C in above definition, our notion of bisimilarity is *stateless*, which is the most robust notion of equality for state-bearing processes [80].

Theorem 4. *$\leftrightarrow_{\mathcal{C}}$ 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 need to 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 $B^{\mathcal{C}}$ as defined in Definition 11 (page page 106), $\forall_{s\in\mathcal{CT}(\Sigma)}\, sB^{\mathcal{C}}s$ 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 a bisimulation $B^{\mathcal{C}}$ (see Defn. 11) such that $sB^{\mathcal{C}}t$. By exchanging the order of conditions (4.1) with (4.2) and (4.3) with (4.4), we can conclude that $tB^{\mathcal{C}}s$. 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 $B_1^{\mathcal{C}}$ and $B_2^{\mathcal{C}}$ such that $sB_1^{\mathcal{C}}t$ and $tB_2^{\mathcal{C}}u$. To show that $s \leftrightarrow_{\mathcal{C}} u$, we show that $B_3^{\mathcal{C}} = B_1^{\mathcal{C}} \circ B_2^{\mathcal{C}}$ is also a bisimulation relation (as per Definition 11).

Consider $(p, q) \in B_3^{\mathcal{C}}$. Then, there exists $r \in \mathcal{CT}(\Sigma)$ such that $pB_1^{\mathcal{C}}r$ and $rB_2^{\mathcal{C}}q$. Using condition (4.1) from Definition 11 we get–

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

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

From the above two conditions we get the following–

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

Note that $p'B_3^{\mathcal{C}}q'$ since, $p'B_1^{\mathcal{C}}r'$, $r'B_2^{\mathcal{C}}q'$ and $B_3^{\mathcal{C}} = B_1^{\mathcal{C}} \circ B_2^{\mathcal{C}}$. Hence, condition (4.1) of Definition 11 holds for $B_3^{\mathcal{C}}$. Similarly, we can show the remaining conditions of Definition 11 to hold for the relation $B_3^{\mathcal{C}}$. Therefore, $B_3^{\mathcal{C}}$ 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 4.2) over signature Σ. That is, for all $p, q, r \in \mathcal{CT}(\Sigma)$ $p \leftrightarrow_{\mathcal{C}} q$

implies $p \mp r \leftrightarrow_{\mathcal{C}} q \mp r$. We define relation

$$B^{\mathcal{C}} \quad = \quad \{(p \mp r, q \mp r) | p \leftrightarrow_{\mathcal{C}} q \text{ where } p, q, r \in \mathcal{CT}(\Sigma)\} \tag{4.6}$$

and show that $B^{\mathcal{C}}$ is a bisimulation. To show that condition (4.1) of Definition 11 holds, we consider the following three cases corresponding to rules –DC3, DC4 and DC5– of the delayed choice operator $\mp$ (see Table 4.2). Consider $(p \mp r, q \mp r) \in B^{\mathcal{C}}$.

Case I Let $C : p \mp r \xrightarrow{a} C' : p'$ as per rule DC3, Table 4.2 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 11). Further, from the definition of $B^{\mathcal{C}}$ above (see Eq. (4.6)), we get $(p' \mp r, q' \mp r) \in B^{\mathcal{C}}$. Hence,

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

Case II Let $C : p \mp r \xrightarrow{a} C' : r'$ as per rule DC4, Table 4.2 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

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

Case III Let $C : p \mp r \xrightarrow{a} C' : p' \mp r'$ as per rule DC5, Table 4.2 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 11). Further, from the definition of $B^{\mathcal{C}}$ above (see Eq. (4.6)), we get $(p' \mp r', q' \mp r') \in B^{\mathcal{C}}$. Hence,

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

From the three cases above, we can easily see that condition (4.1) of Definition 11 holds for the relation $B^{\mathcal{C}}$. Similarly, the remaining conditions of Definition 11 can be easily shown to hold for the relation $B^{\mathcal{C}}$. Hence $B^{\mathcal{C}}$ 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 [80] by noting that our term deduction system is in the *process-tyft* format (with negative premises) presented there and moreover, is *stratifiable* [98].

4.4 Abstract Execution Semantics

We develop an operational semantics for SMSCs using the process theory outlined so far. First, the set of atomic actions *Act* in the process theory is defined as A^{SMSC}, the set of SMSC events as defined in Section 4.1. In Section 4.4.1 we explain the

translation of SMSC specifications to terms in our process theory. Then, Section 4.4.2 presents our notion of configuration and an abstract execution semantics based on these configurations.

4.4.1 Translating SMSCs to process terms

In general, a SMSC may consist of an arbitrary (but finite) number of events. Semantics is provided for a SMSC body by sequentially composing the semantics of the first event definition with the semantics of the remaining part of the SMSC body. The generalized weak sequential composition operator is used to impose necessary ordering requirements across lifelines. For example, let us consider the *UsedWthr* SMSC in Figure 4-5. 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 [98] 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.

4.4.2 Representing/Updating Configurations

As mentioned in Section 4.3, the execution of terms in our theory is constrained by a *configuration*. When we developed our process theory in Section 4.3, we did not assume a specific definition of configurations, but required our configurations to provide these two functions *supports* and *migrates_to*. In particular, these functions capture the transition between system configurations thereby forming the core of

our execution semantics. We develop a notion of **abstract configurations** (Def. 14, page 115) and elaborate an abstract execution semantics over these abstract configurations.

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

Definition 12 (Behavioral Partition)**.** *Let V_p be a set of variables belonging to class p. Let R_p denote a set of regular expressions over events A_p (the set of events that may be performed by objects of class p), with $|R_p| = k$. For each $R_i \in R_p$, let D_i be the minimal DFA recognizing R_i. Then a behavioral partition $beh_p(V_p, R_p)$ of class p defined over V_p and R_p is a tuple $\langle q_1, q_2, \cdots, q_k, v \rangle$, where*

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

Q_i is the set of states of automaton D_i and $Val(V_p)$ is the set of all possible valuations

of variables V_p. We use $BEH_p(V_p, R_p)$ to denote the set of all behavioral partitions of class p defined over V_p and R_p. We use BEH to represent the set of all behavioral partitions.

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

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

Definition 13 (Signature)**.** *For any class p, let V_p be a set of variables belonging to p and R_p be a set of regular expressions over A_p. Then the set of tuples T=$\{(V_p, R_p)\}_{p\in\mathcal{P}}$ is called a signature.*

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

the notion of an abstract configuration.

Definition 14 (Abstract Configuration)**.** *Let each process class p contain N_p objects, and T=$\{(V_p, R_p)\}_{p\in\mathcal{P}}$ be a signature. An abstract configuration over T is defined as $cfg = \{count_p\}_{p\in\mathcal{P}}$ where $count_p : BEH_p(V_p, R_p) \rightarrow \mathbb{N} \cup \{\omega\}$ is a mapping s.t. $\Sigma_{b\in BEH_p(V_p,R_p)} count_p(b) = N_p$. The set of all configurations over signature T is $\mathcal{C}^T$.*

Thus, a configuration records the 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 [91]. *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 apply our 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:

$$n_1 \oplus n_2 = \begin{cases} \omega, & \text{if } n_1 = \omega \text{ or } n_2 = \omega \\ \omega, & \text{if } N_p = \omega \text{ and } n_1 + n_2 > cut_p \\ N_p, & \text{if } N_p \in \mathbb{N} \text{ and } n_1 + n_2 > N_p \\ n_1 + n_2, & \text{otherwise} \end{cases} \tag{4.7}$$

Next, for all $n_1, n_2 \in \mathbb{N} \cup \{\omega\}$ representing object counts of class p, such that $n_1 \geq n_2$:

$$n_1 \ominus n_2 = \begin{cases} \omega, & \text{if } n_1 = \omega, n_2 \neq \omega \\ 0, & \text{if } n_1 = n_2 = \omega \\ n_1 - n_2, & \text{otherwise} \end{cases} \tag{4.8}$$

Indeed, in Section 4.5 we present experiments detailing validation of SMSC systems with unbounded number of objects.

We now explain when a SMSC configuration *supports* an event a and the new configuration it *migrates to* on execution of a. These functions appear in rule Const2, Table 4.1, and are required for defining the transition between system configurations. We begin by defining a mapping $active : A^{SMSC} \rightarrow OS^{\mathcal{P}}$ that indicates which object

selector in an event descriptor "causes" the event to occur:

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

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

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

We also use a function $simple : A^{SMSC} \rightarrow A^{MSC}$ to convert a SMSC event to an MSC event; $simple(e)$ replaces each object selector occurring in e by the corresponding process class 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: Intuitively, a configuration supports an event defined on process class p if there is at least one p object[5] which satisfies the variable valuation and execution history criteria on the event guard. Since we do not maintain the states of individual objects, this is determined by verifying there is at least one behavioral partition of class p which satisfies the event guard and has a non-zero count of objects. We call such a behavioral partition a *witness partition*, which we formally define below.

Definition 15. *Let $e \in A_p^{SMSC}$ (i.e., active process class in e is p) be a SMSC event and $cfg \in \mathcal{C}^T$ be a configuration defined on signature T=$\{(V_q, R_q)\}_{q \in \mathcal{P}}$. Let ϑ and*

[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.

Λ *be the propositional and history based guards in event* e*. We say that behavioral partition* $beh_p = \langle q_1, q_2, \cdots, q_k, v \rangle$ *is a* **witness partition** *for event* e *at configuration* cfg *if*

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

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

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

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

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

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

Definition 16. *Let* $beh = \langle q_1, \cdots, q_k, v\rangle \in Witness(e, cfg)$ *for a configuration* cfg *and event* $e \in A^{SMSC}$. *Let* $simple(e)$[7] $= e'$. *We then define* $dest(beh, e)$ *(the* **destination partition** *of* beh *w.r.t to* e*) as a behavioral partition* $beh' = \langle q'_1, \cdots, q'_k, v'\rangle$, *where*

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

(b) v' *is the effect of* e*'s postcondition on* v.

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

Case 1: $mode(os) = \forall$. Then $cfg.migrates\_to(e)$ returns a unique new configuration that is computed as follows. Let $DP = \{d \mid \exists b \in \mathcal{B}.\ d = dest(b, e)\}$ the set of all destination partitions. Then $\forall d \in DP$, we first set $count'_p(d) = count_p(d) \oplus \Sigma_{b\ s.t.\ dest(b,e)=d}\ count_p(b)$. 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 $k = 1$, [8] $cfg.migrates\_to(e)$ returns a set of possible new configurations as follows. Let us choose any $b \in \mathcal{B}$, and let $dest(b, e) = d$. Then we set $count'_p(d) = count_p(d) \oplus 1$ and $count'_p(b) = count_p(b) \ominus 1$ to obtain a new

[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— 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.

configuration (where counts associated with all other partitions remain unchanged). Repeating this for each $b \in \mathcal{B}$ gives us the set of all possible new configurations.

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

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

4.4.3 Example

We revisit the CTAS example described in Section 4.2 and show the working of our abstract execution semantics.

In particular, we illustrate the execution of a message send event $e = out(\exists_1 Client.g_1^V \wedge g_1^H, CM, connect, \epsilon)$ in SMSC *Connect* shown in Figure 4-4, at a given system configuration, where $g_1^V : status = 0$ and $g_1^H : h_1 = (\epsilon \mid last(e_1))$. Assume that only the history based guards -$h_1 = (\epsilon \mid last(e_1))$ and $h_2 = last(e_2)$[9]- shown in SMSC *Connect* appear in the system description for 'Client' process class. Process class 'CM' has got no history based guards. Further, 'Client' class has variables *status* and v, and 'CM' has a single variable *status*. It can be easily seen that the DFAs corresponding to regular expressions h_1 and h_2 contain only two states. Let

[9] e_1 and e_2 appear at the bottom of MSC *Connect* in Figure 4-4. Expression $last(e)$ was described at the end of Section 4.1.2 under 'Expression Template'.

these be $\{q_0, q_1\}$ and $\{q'_0, q'_1\}$ respectively, where q_0 and q'_1 are accepting states. q_0 is the state of a Client object ready to connect to the CM. This may either be a fresh Client object (with no execution history), or a Client object that has last received a *close* message from CM. q'_1 is the state reached by a client object that has sent a *connect* message to CM in the immediate past.

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

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

The first two elements in a behavioral partition descriptor above correspond to the respective states in the two DFAs, while the next two numeric elements represent the values (0 in each case) of variables *status* and v respectively. By executing event e above, a disconnected client sends a connection request to CM. For event e, $active(e) = \exists_1 Client.g_1^V \wedge g_1^H$, i.e., e can be executed by any 'Client' object satisfying the guard $g_1^V \wedge g_1^H$. Following Definition 15, 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 play event e. After the execution of e, the selected client object will move to states q_1 and q'_1 in the two DFAs, and the destination partition (following Definition 16) is given by $dest(b_1, e) = \langle q_1, q'_1, 0, 0\rangle$. The new configuration cfg' for *Client* class (following 'Case 2' of **migrates_to** function described earlier) is as follows–

$cfg'_{Client}(b_1) = 14$, $cfg'_{Client}(b_2) = 5, cfg'_{Client}(b_3) = 1$, where $b_1 = \langle q_0, q'_0, 0, 0\rangle$, $b_2 = \langle q_1, q'_0, 0, 0\rangle$, $b_3 = \langle q_1, q'_1, 0, 0\rangle$.

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

4.4.4 Properties of SMSC Semantics

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

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 5. *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 $T(S) = \{(V_p(S), R_p(S))\}_{p\in\mathcal{P}}$, 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 Defn. 11) it is easy to see that we cannot guarantee that $S_1 \not\leftrightarrow_{\mathcal{C}^{\mathcal{T}}} S_2$ under a signature $\mathcal{T} \supseteq T(S_1) \cup T(S_2)$. □

Theorem 6. *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 $B^{\mathcal{C}^{\mathcal{T}}}$ such that $S_1 B^{\mathcal{C}^{\mathcal{T}}} S_2$ (see Section 4.3.2). Then, for an action $a \in Act$, from condition (4.1) of Definition 11 we get:

$$\forall_{S_1'\in\mathcal{CT}(\Sigma),C,C'\in\mathcal{C}^{\mathcal{T}}}(C : S_1 \xrightarrow{a} C' : S_1' \;\Rightarrow\; \exists_{S_2'\in\mathcal{CT}(\Sigma)}(C : S_2 \xrightarrow{a} C' : S_2' \wedge S_1' B^{\mathcal{C}^{\mathcal{T}}} S_2')) \quad (4.9)$$

Next, for any signature $\mathcal{T}'$ we define a binary relation $B^{\mathcal{C}^{\mathcal{T}'}} = B^{\mathcal{C}^{\mathcal{T}}}$. 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-guards and postcondition being part of an event itself, following condition (4.9) 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 $S_1' B^{\mathcal{C}^{\mathcal{T}'}} S_2'$ (since $B^{\mathcal{C}^{\mathcal{T}'}} = B^{\mathcal{C}^{\mathcal{T}}}$). Hence, we get

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

Thus, condition (4.1) of Definition 11 holds for relation $B^{\mathcal{C}^{\mathcal{T}'}}$. The remaining conditions of Definition 11 can be shown to hold in a similar manner. Hence, $B^{\mathcal{C}^{\mathcal{T}'}}$ 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.

4.5 Experiments

The operational semantics for SMSCs has been implemented as Prolog rules in the XSB logic programming system [4]. XSB supports *tabled* resolution for query evaluation. This speeds up execution by avoiding redundant computation. The operational semantics of SMSCs lend themselves naturally to Prolog rules leading to a straightforward implementation. On the other hand, the underlying well-engineered fixed point engine in XSB ensures that the evaluation of the rules is done efficiently as well. Both concrete and abstract execution semantics of SMSCs are implemented in XSB and they share as much code as possible. We now present the experimental results obtained for the 'CTAS weather controller' example which was described earlier in

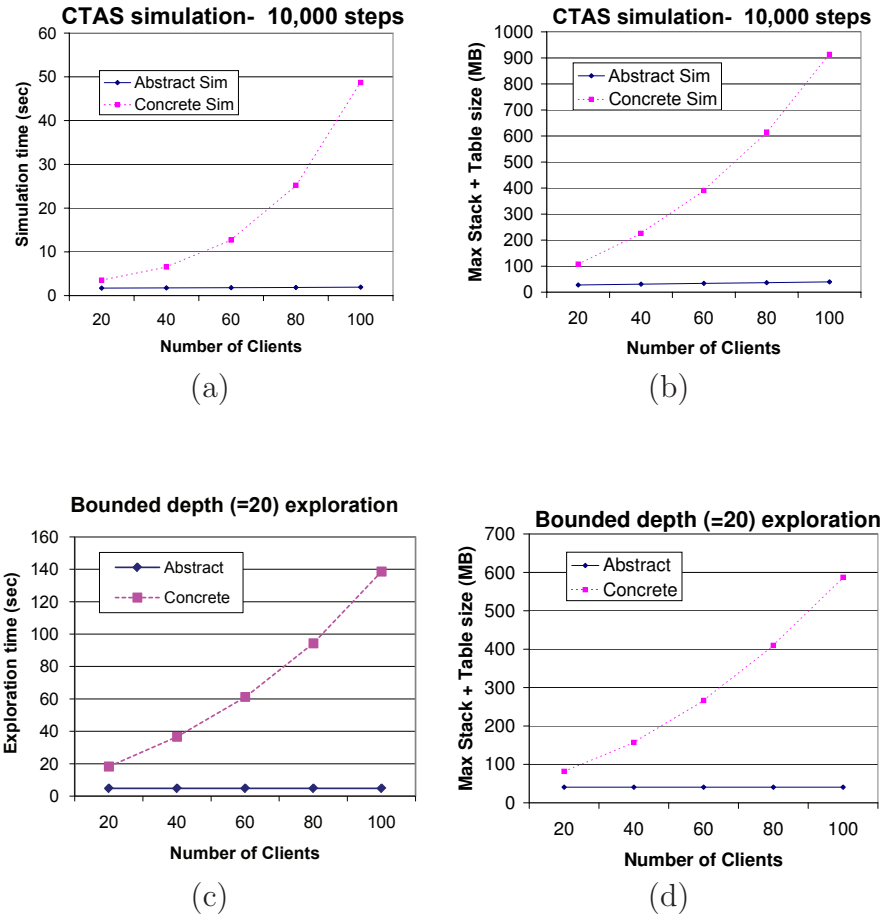


Figure 4-6: (a) Abstract vs Concrete Simulation Times for Different Settings of CTAS (b) Peak Memory Requirement for Abstract and Concrete Simulation, (c) Abstract vs Concrete State Space Exploration Times, (d) Peak Memory Requirement for Abstract and Concrete State Space Exploration.

Section 4.2. All experiments were conducted on a Pentium-IV 3GHz machine with 1GB of main memory.

Modeling effort First we briefly discuss the modeling effort required for the CTAS example from given requirements document. The requirements [1] are given as a set of scenarios from the perspective of the controller (or CM - communications manager),

with precondition(s) given for the occurrence of each scenario. We were able to directly translate each scenario into corresponding SMSC. The high level control flow (HSMSC) was easily obtained by following the preconditions given for various scenarios. Some characteristic features of the modeled example are shown below. Note that CTAS HSMSC is flat, i.e. each of its node corresponds to a SMSC.

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

Simulation The graphs in Figure 4-6(a) and (b) compare the simulation time/memory for abstract vs concrete semantics for the CTAS example. The use case used in simulation first connects all the Client objects to the controller (CM) and then performs the weather updates on all the connected clients (refer to Section 4.2). Different runs correspond to different settings of the CTAS model with number of client objects being varied (shown on the x-axis of the graph). As we can easily observe, for abstract simulation the time/memory remains almost constant ($\approx$ 1.9 sec/40 MB) even as the number of objects is increased from 20 to 100, while it increases at least linearly (3.5 $\rightarrow$ 48.7 sec/100 $\rightarrow$ 900 MB) for concrete simulation. We recall that in abstract simulation, various objects are grouped together into behavioral partitions, unlike in concrete simulation where the states of all objects have to be maintained and manipulated individually.

State Space Exploration We explored all possible traces up to a certain length, where each step in a trace is the execution of an SMSC event. The results appear in Figure 4-6(c) & (d), where the bound on the length of traces explored is set to 20. We find that abstract exploration time/memory is constant ($\approx$ 4.8 sec/41 MB) across different settings of CTAS, and increases linearly (19 $\rightarrow$ 139 sec/ 81 $\rightarrow$ 590 MB) for the concrete exploration. *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). 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.

4.6 Associations

In many distributed systems, various processes often come together to execute a short interaction scenario, establishing temporary links amongst them lasting over the period of communication. 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. Thus,

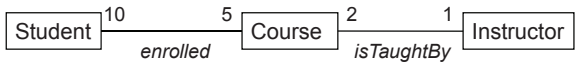


Figure 4-7: Class diagram- Course Management System.

while modeling such distributed protocols, a designer may want to explicitly specify establishing/removing of such association links, and moreover, constrain the communicating processes to be linked via such associations. In the object-oriented setting, such a link between the processes can be viewed as an instance of an association between the classes to which these processes belong, as depicted in a corresponding class diagram. An association can be *static*– representing a static (or permanent) relationship between the processes of the associated classes, or it can be *dynamic*– representing dynamically changing relationship amongst the processes.

In order to allow a designer to specify constraints over associations, we extend our language of SMSCs to include the following three types of constructs– i) an *insert* constraint for establishing association links between various processes, ii) a *check* constraint to restrict the pairs of communicating processes to be linked via a given association, and iii) a *delete* constraint for removing the existing association links between processes. Note that, the *insert* and *delete* constraints are only required in the case of dynamic associations. In the case of static associations we can only apply the *check* constraint. In the following, we first describe a case study which is used later while discussing the associations. Then, we discuss various association constraints in detail and present an extended SMSC execution semantics incorporating associations.

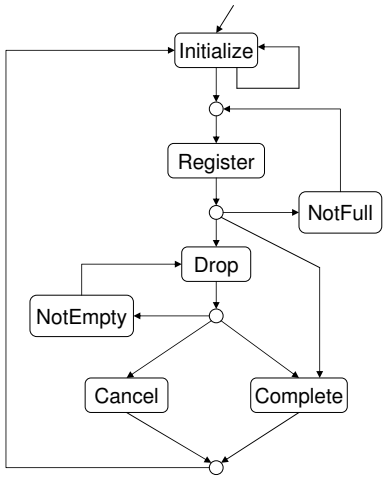


Figure 4-8: HSMSC specification- Course Management System.

4.6.1 Case Study– A Course Management System

In this case study we model the key aspects of a university course-management system (CMS) at a high level. It consists of three process classes– i) Instructor, ii) Course and iii) Student. The class diagram for this example is shown in Figure 4-7. A student can enroll in a maximum of 5 courses, while a course can be taken by a maximum of 10 students. Similarly, an instructor can teach up to 2 courses at a time and a course can be taught only by a single instructor. The HSMSC behavioral description for this case study appears in Figure 4-8.

We assume a fixed pool of courses, from which an instructor can select a course to teach (if it has not already been taken up). This selection occurs in the HSMSC node labeled *Initialize* (see Fig. 4-8). Once a course has been initialized, students can register for this course on a first-come first-serve basis. A successful registration occurs at the node labeled *Register*. Students can register for a course until its maximum

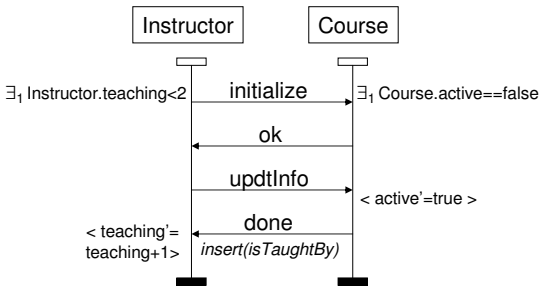


Figure 4-9: SMSC Initialize

capacity (i.e. 10) is reached. Once registration is over, some students might choose to drop the course as well (node labeled *Drop*, Fig. 4-8), and a course gets canceled (node labeled *Cancel*) if less than 3 students remain registered. Otherwise, the course is assumed to complete successfully as indicated by the node labeled *Complete*.

The SMSCs corresponding to the HSMSC nodes labeled *Initialize* and *Drop* are shown in Figures 4-9 and 4-10. To reduce the visual clutter, we have not shown all event guards.

4.6.2 Association constraints

For our purpose, we consider only binary associations relating pairs of process classes. In a SMSC system model, an association constraint is specified below a message arrow, representing a constraint over process pairs executing the corresponding message send/receive events.

Association Insert An association insert constraint is specified as *insert(asc)*, where *asc* is the name of an association, and appears as a post-condition written

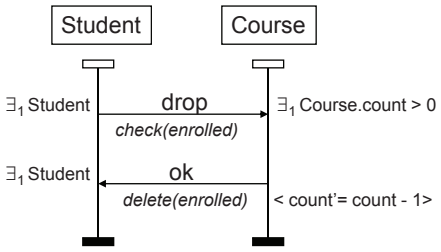


Figure 4-10: SMSC Drop

below a message m in a SMSC. It results in the insertion of object pairs, executing the send and receive events corresponding to message m, in the binary relation *asc*. For example, in SMSC *Initialize* (shown in Figure 4-9) from the course-management system (CMS) described earlier, message arrow *done* is labeled with a post-condition *insert(isTaughtBy)*. Thus, during execution, an object pair (O_c, O_i) will be inserted in the binary relation *isTaughtBy*, such that O_c is the Course object sending the *done* message and O_i is the corresponding Instructor object receiving the *done* message.

Association Check For an association *asc*, the check constraint is specified as a pre-condition *check(asc)*, and appears below a message m in a SMSC. This constraint requires that the object pairs selected for executing the send and receive events corresponding to message m, are in the *asc* relation. For example, in SMSC *Drop* appearing in the CMS example (see Figure 4-10), message arrow *drop* is labeled with the pre-condition *check(enrolled)*. This pre-condition requires that the object pair (O_s, O_c) is in relation *enrolled*, where O_s and O_c are Student and Course objects, respectively executing send and receive events corresponding to the message *drop*.

Association Delete Similar to an association *insert*, a delete constraint over an association *asc* is specified as a post-condition *delete(asc)* annotating a message, say m, in a SMSC. Further, this constraint implicitly requires a *check(asc)* constraint to be imposed as a pre-condition for message m. As the name suggests, the effect of a *delete(asc)* constraint labeling a message m is to remove the object pairs executing the send and receive events corresponding to message m from the binary relation *asc*. For example, in SMSC *Drop* shown in Figure 4-10, message *ok* is annotated with the delete constraint- *delete(enrolled)*. Thus, the object pair chosen for executing the send and receive events for message *ok*, is required to be in the *enrolled* relation. Moreover, this object pair will be removed from the *enrolled* relation after executing the above events.

For a message m annotated with an association constraint, *we require that the corresponding send and receive events occur synchronously*, i.e. the receive event occurs immediately after the send event. Let $e_s = \mathsf{out}(os_i, os_j, m, pc_1)$ and $e_r = \mathsf{in}(os_i, os_j, m, pc_2)$ be the send and receive events corresponding to message m annotated with an association constraint ctr. Then, synchronous execution of e_s and e_r is captured as a single synchronous event given by $\mathsf{synch}(os_i, os_j, m, pc_1, pc_2, ctr)$. Let A_{synch} denote the set of all such synchronous events. We extend the set of SMSC events A^{SMSC} (see Defn. 8) to also include the events in A_{synch}. Further, while translating a SMSC to the corresponding process term (as described earlier in Section 4.4.1), a synchronous event representing message m is added to the process term

using the weak sequential operator $\circ$, such that, all the events preceding send and receive events corresponding to message m in the SMSC event ordering (see Defn. 9) have already been added.

In the case of concrete execution semantics, the handling of associations is straightforward and follows the approach described in the preceding. However, recall that in case of our abstract execution semantics, we do not represent various objects as independent entities, but rather group them together in to *behavioral partitions* (ref. Section 4.4.2). Thus, maintaining associations in the presence of abstract execution semantics poses a challenge. We present one possible approach towards this end in the following Section.

4.7 Abstract execution semantics with Associations

To integrate associations with our abstract execution semantics, where various objects are grouped in to behavioral partitions, we maintain association relations between behavioral partition pairs instead of object pairs. Intuitively, if in a concrete execution two objects, say O and O', are present in a binary relation asc after executing a sequence of events σ. Then, in the abstract execution, following the same event execution sequence σ, the behavioral partition pair (B, B') will be present in the asc relation, such that B (B') is the behavioral partition containing the object O (O') in the abstract execution of σ.

Let $\mathcal{A}$ denote the set of all associations and $asc \in \mathcal{A}$ be an association between

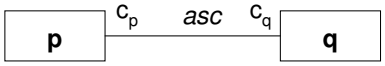


Figure 4-11: Example class diagram.

process classes p and q. Further, let c_p and c_q be the count annotations labeling association asc at class p's and q's end respectively (see class-diagram in Fig. 4-11). We now define the function $max_{asc} : \mathbb{N} \cup \{\omega\} \times \mathbb{N} \cup \{\omega\} \rightarrow \mathbb{N} \cup \{\omega\}$ such that for $n_1, n_2 \in \mathbb{N} \cup \{\omega\}$:

$$max_{asc}(n_1, n_2) = MIN\{n_1 \times c_q, n_2 \times c_p, n_1 \times n_2\}, \tag{4.10}$$

where we assume $\omega \times \_ = \omega$.

The value returned by $max_{asc}(n_1, n_2)$ is the maximum number of asc links that can be established between n_1 p-objects and n_2 q-objects. The first term $n_1 \times c_q$ in the MIN expression in Eq.(4.10) gives the maximum number of asc links that n_1 objects of class p can establish with (sufficient number of) objects of class q. Second term of the MIN expression gives the similar quantity for n_2 objects of class q. Finally, the third term gives maximum number of links that can be established between n_1 and n_2 objects. Thus, a minimum of these three quantities determines the maximum possible asc links that can be established in the given setting. We now define the notion of an *extended* abstract configuration.

Definition 17 (Extended Abstract Configuration)**.** *Let each process class p con-*

tain N_p objects, T=$\{(V_p, R_p)\}_{p\in\mathcal{P}}$ be a signature, and $\mathcal{A}$ be the set of all associations. An extended abstract configuration over T and $\mathcal{A}$ is defined as $cfg_x = (\{count_p\}_{p\in\mathcal{P}}, \{acount_{asc}\}_{asc\in\mathcal{A}})$ where

1. *$count_p : BEH_p(V_p, R_p) \rightarrow \mathbb{N} \cup \{\omega\}$ is a mapping s.t. $\Sigma_{b\in BEH_p(V_p,R_p)} count_p(b) = N_p$, and*

2. *$acount_{asc} : BEH_p(V_p, R_p) \times BEH_q(V_q, R_q) \rightarrow \mathbb{N} \cup \{\omega\}$, where $asc \in \mathcal{A}$ is an association between process classes p and q and $\forall(b, b') \in BEH_p(V_p, R_p) \times BEH_q(V_q, R_q). acount_{asc}(b, b') \leq max_{asc}(count_p(b), count_q(b'))$.*

The set of all extended abstract configurations over signature T and associations $\mathcal{A}$ is denoted as $\mathcal{C}^T_{\mathcal{A}}$.

Thus, besides maintaining the object count in each behavioral partition of various process classes, an extended abstract configuration also keeps track of count of object pairs in various behavioral partition pairs for each association.

We now discuss operational semantics of SMSCs in the presence of associations. Various semantic rules describing execution of SMSCs in the absence of associations (ref. Section 4.3, Tables 4.1 and 4.2), remain unchanged in the presence of associations, except for rule Const2, Table 4.1, which is modified as follows– (i) the configurations C, C', now represent extended abstract configurations from $\mathcal{C}^T_{\mathcal{A}}$ instead of simple object configurations from $\mathcal{C}^T$, and (ii) the *supports* and *migrates_to* methods are replaced by their extended versions– $supports_{ext}$ and $migrates\_to_{ext}$ respectively to

take associations into account. Hence, $supports_{ext}$ and $migrates\_to_{ext}$ methods form the core of our association handling in the abstract semantics. In the following, we discuss these *methods* for different association constraints.

4.7.1 Association Insert

An insert constraint *insert(asc)* is taken into account when executing a synchronous event of the form $e = \mathsf{synch}(os_i, os_j, m, pc_1, pc_2, insert(asc))$. Let $e_s = \mathsf{out}(os_i, os_j, m, pc_1)$ and $e_r = \mathsf{in}(os_i, os_j, m, pc_2)$ denote the send and receive events constituting the synchronous event e, and $C \in \mathcal{C}^T_{\mathcal{A}}$ be a given extended abstract configuration. Then the method call $C.supports_{ext}(e)$ returns *true* if there exist witness partitions (see Definition 15, page 117) for both e_s and e_r at C. Note that, in case a common behavioral partition, say $beh \in BEH_p$, is chosen as a witness partition for both e_s and e_r, we require that $count_p(beh) > 1$. This is to ensure that distinct objects execute e_s and e_r. If this is possible, then $C.migrates\_to_{ext}(e)$ returns a set of possible destination configurations considering the effect of postconditions pc_1, pc_2, and the *insert(asc)* constraint at C.

Let $C \equiv (cfg = \{count_p\}_{p\in\mathcal{P}}, acfg = \{acount_{asc}\}_{asc\in\mathcal{A}})$, $WP_s = Witness(e_s, cfg)$ and $WP_r = Witness(e_r, cfg)$. Here WP_s and WP_r represent the set of possible witness partitions for the events e_s and e_r at C. To simplify our discussion we assume that the send and receive events (i.e. e_s and e_r) are executed by two different process classes, say p and q s.t. $p \neq q$, and hence, $WP_s \cap WP_r = \emptyset$.

We now discuss the effect of $migrates\_to_{ext}$ method, which depends on the abstraction modes of the send and receive events (i.e. $mode(os_i)$ and $mode(os_j)$), giving rise to the following three cases–

1. $(mode(os_i), mode(os_j)) = (\exists, \exists)$, i.e. both send and receive events have *existential* abstraction modes.

2. $(mode(os_i), mode(os_j)) = (\exists, \forall)$ or $(mode(os_i), mode(os_j)) = (\forall, \exists)$, i.e. one of the send or receive events has *existential* abstraction mode, and the other has *universal* abstraction mode.

3. $(mode(os_i), mode(os_j)) = (\forall, \forall)$, i.e. both send and receive events have *universal* abstraction modes.

In the following, we discuss Case-2 from above in detail, the other two cases are handled in a similar manner.

One of the event (send or receive) has existential mode and the other one has universal mode- $(\exists, \forall)$. Let e_x and e_a represent the events having existential and universal execution modes respectively. Without any loss of generality, we assume that e_x is executed by an object of process class p, while e_a is executed by the objects of class q. Further, let WP_x/WP_a be the set of possible witness partitions corresponding to the event e_x/e_a. Recall that, when there are no association constraints, we can choose all objects from all the witness partitions in WP_a to execute e_a, and for executing e_x, we can select any partition from WP_x and an object from that partition. However,

due to the insert constraint, the number of objects that can execute e_a is bounded. In this case we need to take in to consideration the count annotations for association *asc* between classes p and q (*e.g.* see class diagram in Figure 4-11). This is used to determine the *maximum* number of objects of the class q that can be linked with a single object of the class p via *asc* (which is c_q in this case), and hence the maximum number of q-objects that be chosen to execute e_a. For illustration, consider the class diagram for the CMS example shown in Figure 4-7. If p (q) corresponds to Student (Course) class, then a maximum of 10 Student objects can be chosen to execute e_a.

In case $\sum_{b \in WP_a} count_p(b) \leq c_q$ then we can select all objects from all partitions in WP_a to execute e_a. Otherwise, we may have different choices for– a) selecting a subset of witness partitions from WP_a for executing e_a, and b) choosing the number of objects to execute e_a from each selected witness partition. Since, these non-deterministic choices may lead to a large number of alternatives, we *over-approximate* and consider all possible choices in a single execution step. For each partition $b \in WP_a$, we compute a minimum $min(b)$ and a maximum $max(b)$ number of objects that can be chosen to execute e_a. Clearly, for any partition $b \in WP_a$,

$$max(b) \quad = \quad MIN\{count_p(b), c_q\} \tag{4.11}$$

The minimum number of objects is determined as follows– $min(b) = MAX\{0, c_q - \sum_{b' \in WP_a \setminus \{b\}} count_p(b')\}$. Essentially, $min(b)$ is 0 if it is possible to select c_q objects from other partitions in WP_a. Otherwise, there are not enough number objects in

all partitions in WP_a (excluding b), and a minimum of $c_q - \sum_{b' \in WP_a \setminus \{b\}} count_p(b')$ objects are required from b. Note that, when $\sum_{b \in WP_a} count_p(b) \leq c_q$, for all $b \in WP_a$ we set $min(b) = max(b) = count_p(b)$. Let $DP_a = \{d | \, b \in WP_a \wedge d = dest(b, e_a)\}$ represent the set of destination partitions corresponding to WP_a. For a destination partition $d \in DP_a$, we define $B(d) = \{b | \, b \in WP_a \wedge dest(b, e_a) = d\}$. In case of existential event e_x, we do not over approximate and choose any partition $b_x \in WP_x$ as a witness. Let, $d_x = dest(b_x, e_x)$ be the corresponding destination partition.

The count of various behavioral partitions is updated via $migrates\_to_{ext}$ method as follows. For each witness partition $b \in WP_a$ we decrement its count by $min(b)$, i.e. $count'_p(b) = count_p(b) \ominus min(b)$. In case of a destination partition $d \in DP_a$, we determine the number of objects that can move in to d as

$$incr(d) = MIN\{c_q, \sum_{b \in B(d)} max(b)\}.$$

Then we compute $count'_p(d) = count_p(d) \oplus incr(d)$. The counts of b_x and d_x are updated as follows– $count'_q(b_x) = count_q(b_x) \ominus 1$ and $count'_q(d_x) = count_q(d_x) \oplus 1$. The object counts of all other partitions remain unchanged.

Next, we discuss the update of *asc* relation content which also occurs via $migrates\_to_{ext}$ method in our operational semantics. Consider the illustration shown in Figure 4-12, depicting various behavioral partitions involved in the execution of e_x/e_a and possible *asc* links existing among them. These links are classified into the following four categories, based on the pair of behavioral partitions involved in a link–

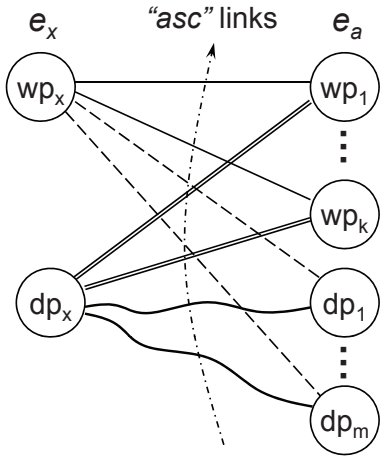


Figure 4-12: The $(\exists, \forall)$ case for associations.

(i) $\langle wp_x, wp\rangle, wp \in WP_a$ (linked via plain lines),

(ii) $\langle dp_x, dp\rangle, dp \in DP_a$ (linked via curved lines),

(iii) $\langle wp_x, dp\rangle, dp \in DP_a$ (linked via dashed lines), and

(iv) $\langle dp_x, wp\rangle, wp \in WP_a$ (linked via double lines).

We now describe the update of association counts for tuples in **(ii)** above, i.e. $\langle dp_x, dp\rangle, dp \in DP_a$ (linked via curved lines). Association count updates for various other tuples follow the similar approach.

$\mathbf{acount'_{asc}(dp_x, dp), dp \in DP_a}$: From the association links shown in Figure 4-12, we observe that there three potential sources from which *asc* links can be added to the tuple $\langle dp_x, dp\rangle$. These are– (a) $\langle wp_x, dp\rangle$, (b) $\langle wp_x, wp\rangle, wp \in B(dp)$, and (c) $\langle dp_x, wp\rangle, wp \in B(dp)$. We now determine the contribution from each of these sources–

From (a). $x_1 = MIN\{c_q - 1, acount_{asc}(wp_x, dp)\}$. Assuming that the object executing e_x is inserted in *asc* link with at least one object executing e_a, it can not be linked to more than $c_q - 1$ objects in *dp*.

From (b). $x_2 = MIN\{c_q, \sum_{wp \in B(dp)} max(wp)\}$, where $max(wp)$ is computed as defined earlier in Eq. (4.11). The summation term in the MIN expression here computes the maximum number of objects that can migrate in to *dp*, and it can not be greater than c_q objects. Note that, this quantity is same as the value x computed in Eq. (4.13).

From (c). Let $n = \sum_{wp \in B(dp)} max(wp)$, i.e. the maximum number of objects that can possibly migrate into *dp*. If $n \leq c_q$, then all these objects are chosen to move to *dp*, and their contribution to *asc* count from tuples $\langle dp_x, wp \rangle, wp \in B(dp)$ to tuple $\langle dp_x, dp \rangle$ is determined as–

$$x_3 = \sum_{b \in B(dp)} MIN\{acount_{asc}(dp_x, b), max(b) \times (c_p - 1)\}.$$

We know that for each $b \in B(dp)$, all objects in b executing e_a will be inserted in association *asc* with the object executing e_x. Hence, each such object in b can not be linked to more than $c_p - 1$ objects in dp_x.

In case $n > c_q$, we compute the *maximum* possible *asc* link contribution from any c_q objects out of these n objects. For this, we first compute for each $b \in B(dp)$– $s(b) = MIN\{acount_{asc}(dp_x, b), max(b) \times (c_p - 1)\}/max(b)$. The

term $s(b)$ determines per object *asc* link contribution for a given partition b. Let $B^s = \langle b_1, \ldots, b_{|B(dp)|} \rangle$, such that $1 \leq u < v \leq |B(dp)| \implies s(b_u) \geq s(b_v)$. Next, we determine the index value i, such that $\sum_{r=1}^{i} max(b_r) < c_q < \sum_{r=1}^{i+1} max(b_r)$, and compute–

$$x_3 = \sum_{r=1}^{i}(max(b_r) \times s(b_r)) + MIN\{(c_q - \sum_{r=1}^{i} max(b_r)) \times (c_p - 1), acount_{asc}(dp_x, b_{i+1})\}$$

Finally, the *asc* count for the tuple $\langle dp_x, dp \rangle$ is updated as–

$$acount'_{asc}(dp_x, dp) = MIN\{max_{asc}(count'_p(dp_x), count'_q(dp)), x\}, \text{ where}$$

$$x = acount_{asc}(dp_x, dp) + x_1 + x_2 + x_3.$$

A set of configurations will be returned by the *migrates_to*$_{ext}$ method due to choice in the selection of a witness partition for existential event e_x.

4.7.2 Association Check/Delete

We discuss the association *check* and *delete* constraints together since, a delete constraint *delete(asc)* specified as a message m's post-condition in a SMSC specification, implies a check constraint *check(asc)* as message m's pre-condition.

The *check(asc)* constraint requires that the object pairs selected to synchronously execute the send and receive events corresponding to message m, are related via *asc*. On the other hand, *delete(asc)* constraint removes the object pairs related via *asc* from the *asc* relation. A check constraint is taken into account when executing a synchronous event e with *check* or *delete* as the association constraint. Again, let e_s and e_r be send and receive events corresponding to the synchronous event e, and $C \in \mathcal{C}^T_{\mathcal{A}}$ be the extended abstract configuration when executing e. Then the method call $C.supports_{ext}$ checks whether there exist two (disjoint) sets of objects that can execute e_s and e_r at C, such that objects across the two sets are related via *asc*. If this is possible, then witness behavioral partitions for both these events are selected and corresponding destination partitions are updated by the $migrates\_to_{ext}$ method. In addition, $migrates\_to_{ext}$ updates the *asc* relation contents of various witness and destination partitions participating in the execution of e.

Similar to the *insert* constraint, handling of *check* and *delete* constraints depends on the abstraction modes of the send and receive events. Let $mode(os_i)$ and $mode(os_j)$ be the abstraction modes for the send (e_s) and receive (e_r) events respectively. Then, the following three cases arise.

1. $(mode(os_i), mode(os_j)) = (\exists, \exists)$, i.e. both send and receive events have *existential* abstraction modes.

2. $(mode(os_i), mode(os_j)) = (\exists, \forall)$ or $(mode(os_i), mode(os_j)) = (\forall, \exists)$, i.e. one of the send or receive events has *existential* abstraction mode, and the other has

universal abstraction mode.

3. $(mode(os_i), mode(os_j)) = (\forall, \forall)$, i.e. both send and receive events have *universal* abstraction modes.

We now describe handling of *check* and *delete* constraints for the case with one of the send or receive events having existential mode, and the other universal. Other cases are handled similarly.

Let e_x and e_a denote the events having existential and universal abstraction modes respectively. Without any loss of generality, we assume that e_x is executed by an object of process class p, while e_a is executed by the object(s) of class q. Further, let WP_x/WP_a be the set of possible witness partitions corresponding to the event e_x/e_a, such that for each partition wp in WP_x (WP_a) there is a partition wp' in WP_a (WP_x) with *asc* links between wp and wp', i.e. $acount_{asc}(wp, wp') > 0$. Then, we can choose any behavioral partition from WP_x, say wp_x, to act as the witness partition for e_x; and the subset of partitions $WP_a^x \subseteq WP_a$ for e_a, such that $\forall wp \in WP_a^x.(acount_{asc}(wp_x, wp) > 0)$.

Let dp_x be the resulting destination partition corresponding to wp_x after executing e_x. Then $migrates\_to_{ext}$ will decrement the object count of wp_x by 1 (i.e. $count'_p(wp_x) = count_p(wp_x) \ominus 1$) and increment the object count of dp_x by 1 (i.e. $count'_p(dp_x) = count_p(dp_x) \oplus 1$). Now, in the case of e_a, in order to satisfy the *check(asc)* constraint we can only select objects in WP_a^x which are linked via the *asc* association to the object executing e_x. However, due to over-approximation in

maintaining the association links– (a) the existence of linked object pairs in two behavioral partitions b and b' can not be guaranteed, even if $acount_{asc}(b, b') > 0$, and (b) we loose the precise structure of association links between objects. Thus, for each partition $wp \in WP_a^x$ we determine a minimum $min(wp)$ and a maximum $max(wp)$ number of objects that can potentially execute e_a. From condition (a) above we get $min(wp) = 0$, i.e. there might be no object in wp linked with the object executing e_x. Further, we compute

$$max(wp) \quad = \quad MIN\{c_q, count_q(wp), acount_{asc}(wp_x, wp)\}. \tag{4.12}$$

Above, first term in the MIN expression is c_q, which is the maximum number of q-objects, that an object of class p executing e_x can be linked with. However, from wp we *cannot* choose a number of objects greater than its count, i.e. $count_q(wp)$, or greater than the number of asc links between wp_x (partition from which object to execute e_x is chosen) and wp. Thus, $max(wp)$ is computed as the minimum of these three quantities.

We compute the new count of each partition $wp \in WP_a^x$ as $count'_q(wp) = count_q(wp) \ominus min(wp)$, which is same as $count_q(wp)$. Let DP_a^x be the set of all destination partitions corresponding to the partitions in WP_a^x with respect to the execution of e_a. Consider a behavioral partition $dp \in DP_a^x$. Let $B(dp) = \{wp |\, wp \in WP_a^x \wedge dest(wp, e_a) =$

$dp\}$. Then, we compute

$$x \;=\; MIN\{c_q, \sum_{wp \in B(dp)} max(wp)\} \tag{4.13}$$

as the maximum increment in the object count of dp. Note that, due to association check constraint we are sure that no more than c_q objects can migrate to a destination partition. Finally, $count'_q(dp) = count_q(dp) \oplus x$.

Next, we discuss the update of *asc* relation content which occurs via $migrates\_to_{ext}$ method in our operational semantics. Consider the illustration shown in Figure 4-12, depicting various behavioral partitions involved in the execution of e_x/e_a and possible *asc* links among them. As mentioned earlier, these links are classified into the following four categories, based on the pair of behavioral partitions involved in a link–

(i) $\langle wp_x, wp\rangle, wp \in WP_a^x$ (linked via plain lines),

(ii) $\langle dp_x, dp\rangle, dp \in DP_a^x$ (linked via curved lines),

(iii) $\langle wp_x, dp\rangle, dp \in DP_a^x$ (linked via dashed lines), and

(iv) $\langle dp_x, wp\rangle, wp \in WP_a^x$ (linked via double lines).

The association counts for tuples in **(ii)** above are updated as described in the following; updates for other cases are handled in a similar manner.

$\mathbf{acount'_{asc}(dp_x, dp), dp \in DP_a^x}$: From the association links shown in Figure 4-12, we observe that there three potential sources from which *asc* links might be added

to the tuple $\langle dp_x, dp \rangle$. These are– (a) $\langle wp_x, dp \rangle$, (b) $\langle wp_x, wp \rangle, wp \in B(dp)$, and (c) $\langle dp_x, wp \rangle, wp \in B(dp)$. We now determine the contribution from each of these sources–

From (a). $x_1 = MIN\{c_q - 1, acount_{asc}(wp_x, dp)\}$. Assuming that the object executing e_x is linked via asc to at least one object which is executing e_a, it can not be linked to more than $c_q - 1$ objects in dp.

From (b). $x_2 = MIN\{c_q, \sum_{wp \in B(dp)} max(wp)\}$, where $max(wp)$ is computed as defined earlier for a witness partition. The summation term in the MIN expression computes the maximum number of objects that can migrate in to dp, and it can not be greater than c_q objects. Note that, this quantity is same as the value x computed in Eq. (4.13).

From (c). Let $n = \sum_{wp \in B(dp)} max(wp)$, i.e. the maximum number of objects that can possibly migrate into dp. If $n \leq c_q$, then all these objects are chosen to move to dp, and their contribution to asc count from tuples $\langle dp_x, wp \rangle, wp \in B(dp)$ to tuple $\langle dp_x, dp \rangle$ is determined as–

$$x_3 = \sum_{b \in B(dp)} MIN\{acount_{asc}(dp_x, b), max(b) \times (c_p - 1)\}.$$

We know that for each $b \in B(dp)$, all objects in b executing e_a are linked via asc to the object executing e_x. Hence, each such object in b can not be linked to more than $c_p - 1$ objects in dp_x.

In case $n > c_q$, we compute the *maximum* possible *asc* link contribution from any c_q objects out of these n objects. For this, we first compute for each $b \in B(dp)$– $s(b) = MIN\{acount_{asc}(dp_x, b), max(b) \times (c_p - 1)\}/max(b)$. The term $s(b)$ determines per object *asc* link contribution for a given partition b. Let $B^s = \langle b_1, \ldots, b_{|B(dp)|}\rangle$, such that $1 \leq u < v \leq |B| \implies s(b_u) \geq s(b_v)$. Next, we determine the index value i, such that $\sum_{r=1}^{i} max(b_r) < c_q < \sum_{r=1}^{i+1} max(b_r)$, and compute–

$$\begin{aligned} x_3 &= \sum_{r=1}^{i}(max(b_r) \times s(b_r)) + \\ & \quad MIN\{(c_q - \sum_{r=1}^{i} max(b_r)) \times (c_p - 1), acount_{asc}(dp_x, b_{i+1})\} \end{aligned}$$

Finally, if there is no *delete(asc)* constraint present, the *asc* count for the tuple $\langle dp_x, dp\rangle$ is updated as–

$$acount'_{asc}(dp_x, dp) = MIN\{max_{asc}(count'_p(dp_x), count'_q(dp)), x\}, \text{ where}$$

$$x = acount_{asc}(dp_x, dp) + x_1 + x_2 + x_3.$$

Otherwise, in the presence of the *delete(asc)* constraint, we do not consider the *asc* link contribution from the tuples $\langle wp_x, wp\rangle, wp \in B$, and compute x above as–

$$x = acount_{asc}(dp_x, dp) + x_1 + x_3.$$

4.7.3 Default case

In general there might be no association constraints specified with a message in a SMSC specification. Then the execution of associated send and receive events remains asynchronous, and the object counts of the witness and the destination partitions are updated as in the case of no associations. However, if the witness partition(s) chosen for executing the event are involved in any association links, then we accordingly update the association contents.

4.8 Discussion

In this chapter, we have presented Symbolic Message Sequence Charts (SMSCs) as a lightweight syntactic and semantic extension to conventional MSCs. SMSCs are particularly suitable for behavioral description of systems with many behaviorally similar objects. In such systems, similar MSC specifications become too voluminous for human comprehension. First, we presented an abstract execution semantics for SMSCs without involving associations, in which objects are dynamically grouped together and executed following a process theory based operational semantics. Our approach was validated through a detailed case study and experiments involving a non-trivial weather controller specification. We then extended the notation of SMSCs with various association constraints and discussed in detail the abstract SMSC execution semantics in the presence of associations.

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. In future, we will investigate other requirement templates involving similar processes and integrate/support them systematically in our framework.

Finally, recall that, our notation of Interacting Process Classes, presented earlier in Chapter 3, is a hybrid notation in which the behavior of a process class is specified by a labeled transition system and unit interactions between processes are described by MSCs. However, the IPC model does not support universal abstraction of lifelines as in the case of SMSCs — only existential abstraction is supported. Also, there is no explicit structuring of MSCs in the IPC model — it is inherent in the control flow of process classes. This brings the model closer to state-based notations (rather than MSC-based notations).

Chapter 5

IPC vs SMSC

In this chapter, we present a comparative study between our IPC and SMSC modeling frameworks, which were discussed earlier in Chapters 3 and 4 respectively.

5.1 Local vs Global control

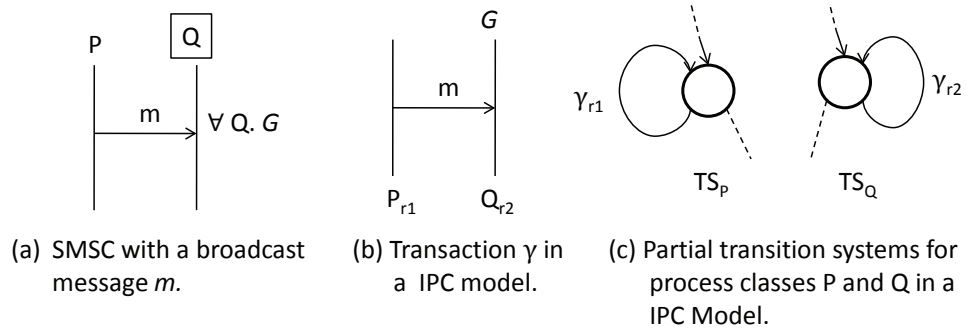
One distinction that can be immediately made when comparing the two notations is local (IPC) vs global (SMSC) control. While, in the case of IPC, the control flow of various process classes is explicitly specified by means of a Labeled Transition System, SMSCs provide a global view of system execution where local control flow of various process classes may not be inferred. Further, unlike MSCs, where an initial attempt towards determining local control flow of a process can be made by projecting events along the lifelines representing the process (e.g. [114]), in case of SMSCs events along a lifeline may not even be executed by the same (set of) objects, making it harder to

infer the control flow.

Having said that, we observe that the execution of IPC models is not entirely local — the transactions are selected for execution in a global manner. This is because, in an IPC transaction multiple processes can participate, and need to synchronize at the beginning of a transaction execution. Further, in the Labeled Transition Systems describing the control flow of various process classes, there may be non-deterministic choice between several outgoing transitions (labeling a role in a transaction) from a control state. Hence, problems similar to non-local choice as in case of HMSCs [15] may arise, with different processes making locally inconsistent choice of executing a transaction. This issue in IPC needs to be addressed, for instance by means of additional synchronization messages from a control process, for obtaining a distributed system implementation

5.2 Granularity of Execution

A key feature of IPC formalism is that it allows communication among processes to be specified as short protocol snippets (modeled as *transactions*), rather than a single message send or receive; which is a desirable property in modeling of various control protocols involving bi-directional flow of information. Taking advantage of this abstraction, the execution of IPC models is defined in terms of transactions (described using MSCs in our case), in which multiple processes from distinct classes may participate together. On the other hand, execution of SMSCs is defined at the



(a) SMSC with a broadcast message *m.*

(b) Transaction γ in a IPC model.

(c) Partial transition systems for process classes P and Q in a IPC Model.

Figure 5-1: Showing approximation of a SMSC broadcast message in IPC model.

level of individual send or receive events, with each event being executed by a subset of objects from a process class, as determined by the associated event guard. An exception to this asynchronous event execution in SMSCs occurs in the presence of association constraints, where the send and receive events corresponding to a message annotated with an association constraint are executed synchronously.

5.3 Lifeline Abstraction

Another important difference between IPC and SMSC notations is the introduction of *symbolic* lifelines in SMSCs. A lifeline in an MSC describing a transaction in an IPC model is always concrete, and represents a single object. While, lifeline in a SMSC description can be symbolic, representing a subset of objects belonging to the process class represented by that lifeline. Further, the *universal* ($\forall$) abstraction mode associated with an event appearing along a symbolic lifeline in a SMSC, allows specification of broadcast messages in a more direct and natural manner. The similar behavior cannot be captured exactly in our IPC framework. However, it can be ap-

proximated in IPC by repeatedly sending the broadcast message using loops in the transition systems of the involved process classes. For illustration, consider the SMSC shown in Figure 5-1(a), in which process-class P containing a single process, broadcasts message m to all processes of type Q satisfying the guard G. In Figures 5-1(b) and (c), we depict a possible way to model the broadcast of message m (shown in the SMSC in Fig. 5-1(a)) in a IPC model. An IPC transaction γ in Figure 5-1(b) captures the sending of message m from a process of type P to a process of type Q. Notice that lifeline Q_{r2} of transaction γ (in Fig. 5-1(b)) is guarded by the same guard G, as appearing in the guard of receive event for message m in the SMSC shown in Figure 5-1(a). Partial transition systems for process classes P and Q in an IPC model appear in Figure 5-1(c). The loops shown in these partial transition systems are labeled with roles γ_{r1} and γ_{r2} for process classes P and Q, respectively. These loops will allow repeated execution of transaction γ, and hence in sending of message m by P to all processes of type Q satisfying the guard G. Also, the executions corresponding to *existential* abstraction mode $\exists_k$, $k \in \mathbb{N}$, in SMSCs cannot be translated to an IPC execution. This is because k is a global parameter, and processes cannot decide locally whether or not a message has been sent k times.

5.4 Which is more expressive?

As discussed in the preceding, clearly models described using SMSCs may not be described using IPC notation. In the following, we show that it is possible to construct

a SMSC model corresponding to an IPC model such that, the execution of SMSC model includes the event traces from the execution of corresponding IPC model.

Let $\{TS_p = \langle S_p, Act_p, \rightarrow_p, init_p, V_p, v_{init_p}\rangle\}_{p\in\mathcal{P}}$ be an IPC model, with Γ denoting the transactions appearing in TS_p (see Definition 1, page 35). Then, we define $PR = \langle SP_{\mathcal{P}}, \rightarrow_{sp}\rangle$, the synchronous product of the Labeled Transition Systems describing the control flow of various process classes $p \in \mathcal{P}$ as follows–

- $(init_{p1}, \ldots, init_{pn}) \in SP_{\mathcal{P}}$ is the initial state of PR, where $p_i \in \mathcal{P}$ and $|\mathcal{P}| = n$.
- $\rightarrow_{sp} \subseteq SP_{\mathcal{P}} \times \Gamma \times SP_{\mathcal{P}}$, such that, $s \xrightarrow{\gamma}_{sp} s'$, where $S = (s_1, \ldots, s_n)$, $S' = (s'_1, \ldots, s'_n)$ iff,
  - $\exists s_{i_1}, \ldots, s_{i_k} \in S, \gamma \in \Gamma \cdot s_{i_j} \xrightarrow{\gamma_{i_j}} s'_{i_j}$, where $\gamma_{i_1}, \ldots, \gamma_{i_k}$ represent all distinct roles in transaction γ[1],
  - $s'_m = s'_{i_j}$ for $m = i_j$, and $s'_m = s_m$ otherwise.

Thus, PR represents a labeled transition system, whose edges are labeled with transactions from Γ. It is now straightforward to construct a HSMSC H corresponding to PR, which together with $V_p, v_{init_p}\}$ from the definition of TS_p, can be used to obtain a HSMSC specification $\langle H, \bigcup_{p\in\mathcal{P}}\{V_p, v_p^{init}\}\rangle$ (see Definition 10, page 97). Note that, the above translation is possible since, SMSCs are more expressive than the MSCs used for describing transactions in our IPC modeling framework. Further, since SMSC

[1]Note that, here we assume that all roles in a transaction correspond to distinct process classes. Extensions for considering transactions with multiple roles from the same class are straightforward, and will involve having multiple state components from the same process class in PR.

execution semantics is event based, the execution traces of the HSMSC specification obtained above will include the execution traces of the original IPC model.

Part III

Model-based Test Generation

Model-based testing is a well-known software development activity. The key idea in model-based testing is to develop an explicit behavioral model of the software from informal requirements. This forms a precise specification of the software's intended behaviors. The behavioral model is searched to generate a test-suite or a set of test cases. These test cases are tried on the software (which might have been constructed manually or semi-automatically) to check the software's behaviors and match them with the intended behaviors as described by the model. A collection of contributions in this area appears in the book [20].

Researchers have studied the methodological and technical issues in model-based test generation, focusing on how a given model can be best exploited for constructing test suites of real applications. In these works, different kinds of state machine like executable models have been used. Each process of the system-under-test is modeled as a (finite) state machine or I/O automata, or as a process algebraic expression (which can be compiled into a state machine). A common thread linking all these works is that they rely on *intra-process behavioral models*.

However, there has been little effort in utilizing *inter-process behavioral models*, such as the ones using Message Sequence Charts (MSCs) [62] based notations, for the purpose of test generation. Visually, a Message Sequence Chart depicts communicating processes as vertical lines; communication between processes are shown by horizontal or downward sloping message arrows between these vertical lines. Thus, MSCs emphasize *inter-process communication*. In the following chapter, we discuss

related work in the domain of model-based testing. In the next two chapters, we study test generation from our notations of *Interacting Process Classes* (Chapter 7) [43] and *Symbolic Message Sequence Charts* (Chapter 8) respectively, presented earlier in Chapters 3 and 4.

Chapter 6

Testing: Related Work

Model based testing of reactive systems is a well studied and an active area of research [20]. 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 [42]. Note that, our current work falls in the latter category.

6.1 State-based

These approaches use some variant of state-based notations for modeling system behaviors. In [93], underlying system behavior is described using Extended Finite State Machines (or *EFSM*) — the test generation involves translating the system model into a Constraint Logic Programing (CLP) specification, adding in constraints corresponding to a test-purpose, and executing resulting CLP program. Specifications in

AutoLink [70] are written in SDL [2], 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 [70] 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 [112, 64]. 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 [104, 35]. 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 [48] 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 [86], involving test generation based on coverage criteria such as transition and predicate coverage. In [56] 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 [57] uses symbolic model checking for test generation from Statecharts. In TesTor [88], 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 [89], 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 [36, 5].

6.2 Scenario-based

In TOTEM [19], test cases are derived from use cases, which are structured and detailed using UML activity and sequence diagrams. Sequence diagrams are used in SeDiTeC [34] and SCENTOR [120] for describing test specifications, which are extended with method parameters, and return values for method calls for actual testing. COWSuite [12] 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 [113] supports test generation from a HMSC model, in which test generation is primarily driven by the *edge-coverage* criteria in a HMSC. In [73], 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.

6.3 Combined notations

A testing framework involving use of state and scenario based notations is presented in [17]. A multi-paradigmatic approach towards model based testing was proposed in [46], which, besides state/scenario based notations, also allows models to be described using other diagrammatic notations and/or program fragments.

6.4 Symbolic Test Generation

A number of model-based testing approaches support symbolic test generation from system specifications involving data variables [104, 92, 35]. 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.

To the best of our knowledge, there is no existing work 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.

Chapter 7

Test Generation from IPC

As described earlier in Chapter 3, in the IPC notation we describe each process-class (*i.e.* a collection of processes) in the system-under-test as a labeled transition system. However, each action label in a labeled transition system denotes a guarded Message Sequence Chart (MSC) instead of an atomic action, such as a message send or receive. Thus, if a MSC γ involves processes p and q, γ will appear in the action labels in the transition systems for process p *and* process q. *A global system execution trace in this model is a sequence of MSCs.* Further, we can symbolically execute such models without maintaining the states of individual processes in a process class.

One of the main advantages of using such MSC-based models over State-based models for test generation is that, we can represent test cases and test case specifications at a higher level than conventional model-based test generation methods. A test case specification is a user input used for guiding the test generation process [112].

Given an IPC system model where the set of all MSCs appearing in the model is Γ, we present a test case as a finite sequence of MSCs drawn from Γ. A test case specification is also a sequence of MSCs drawn from Γ where any sequence of charts containing the test case specification as a subsequence is said to satisfy the test case specification. This high level view of a test case can be helpful in providing a quick and intuitive understanding of its behavior. The user can get a fairly good idea of a test case behavior by looking at the sequence of MSCs it contains, rather than having to examine the low level details involving message send/receive events or local computations.

Also, since our IPC models can be executed symbolically, we can efficiently generate symbolic test cases for process classes with many (active) objects. As we have already mentioned earlier, such interacting process classes are common in many application domains — telecommunication (phone and switch classes), avionics (class of aircrafts being controlled) and automotive infotainment (class of multimedia devices operating inside a car). We note that, our notion of symbolic test case groups together various concrete test-cases with similar behavior, differing only in the identities of objects of various classes participating in a test case. This makes our technique particularly suitable for testing control systems with many similar interacting processes.

Finally, since our behavioral model itself is MSC-based, we are able to exploit the (fragments of) MSCs present in the requirements as MSCs in the behavioral

model. This establishes a tighter connection between informal software requirements, formal software models, and the test cases generated from such models. Such relationships enable easy traceability of the test execution results back to the original requirements [42].

Finally, we would like to mention that our focus in the current chapter is on the *test case generation* in the form of MSCs from the MSC-based executable models of process classes. There are existing works (*e.g.* [89]) covering test case execution and assigning test verdicts for the test cases derived in the form of MSCs.

7.1 Case Study – MOST

In the following, we first present a case-study which is used as a running example in our following discussion. The MOST (Media Oriented Systems Transport) [79] is a networking standard that has been designed for interconnecting various classes of multimedia components in automobiles. It is currently maintained by the "MOST Cooperation", an umbrella organization consisting of various automotive companies and component manufacturers like BMW, Daimler-Chrysler 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.

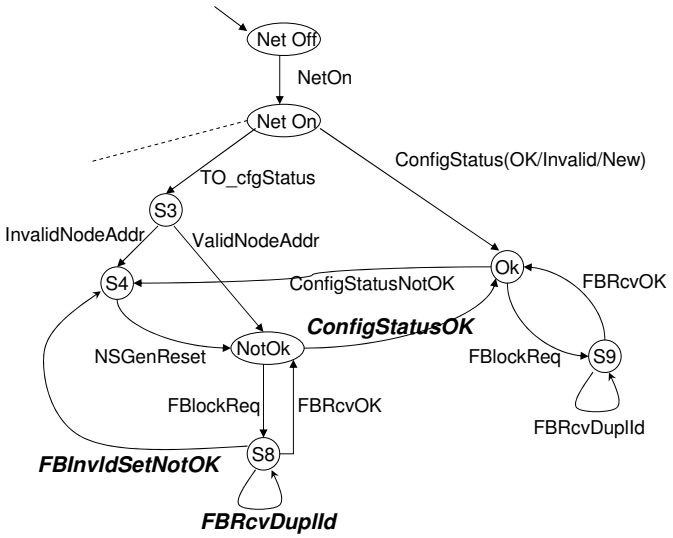
A node (or the device) in the MOST network is a physical unit connected to the network via a Network Interface Controller (NIC). A MOST system may consist

of up to 64 such nodes with identical NICs. A network device generally consists of various functional blocks, such as a tuner, an amplifier, CD player etc. Each such functional block provides a number of functionalities which may be directly available to user via human-machine interface or for use by other devices in the network; for example, a CD player provides Play, Stop and Eject functions. A special function block called the NetBlock is present in all the devices and provides functions related to the entire device. Various specification documents for MOST are available at `http://www.mostcooperation.com/publications/` One of the specifications, namely the 'MOST Dynamic Specification', presents the general description of the dynamic behavior in a MOST network, encompassing: a) Network Management, b) Connection Management, and c) Power Management. Network management ensures secure communication between applications over the MOST network by maintaining and providing most recent information about various nodes, whereas Connection Management deals with the protocols for establishing communication channels between nodes in the network. Network wake-up and shutdown are handled by the Power Management component.

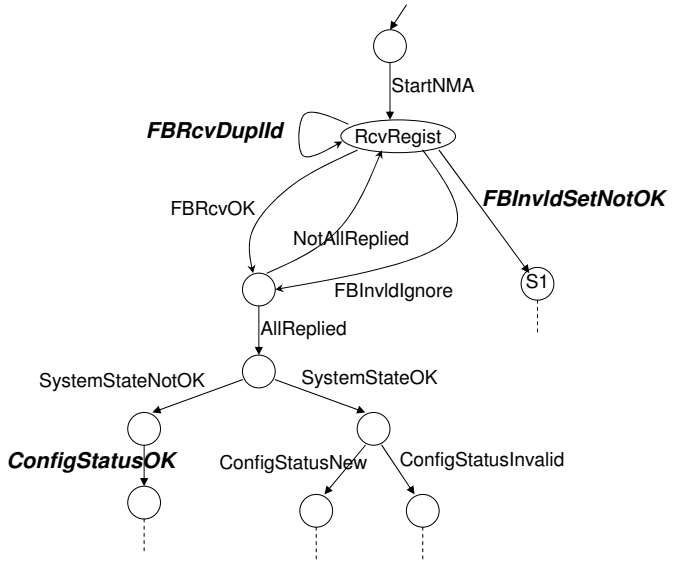
For our experiments, we modeled only the Network management part of MOST. From the network management perspective the system consists of: (i) Network Master (**NM**), a specific node which maintains the *central registry* containing network address information about various devices and their functional blocks in the network, and (ii) Network Slaves (**NS**), which are the remaining nodes in the network. The

NM has two main functions: (a) maintaining the *central registry* with most up-to date information, and (b) providing this information to various nodes when requested. The requirements document describes the NM using two parallel processes: one requests the configuration from NS when required, and other receives the registration information sent by them. The configuration information received from slaves is checked for various errors, such as invalid or duplicate functional block addresses etc. The validity of the central registry is reflected in the NM variable 'System State' which is set to 'OK' when registry is valid and 'NotOK' otherwise. At system startup the state is always 'NotOK', and subsequently becomes 'OK' once NM is able to complete the network scanning and update the central registry without any errors. Also, some network slave may enter/leave the network resulting in 'Network Change Event' (NCE), which causes the NM to re-scan the network and communicate the changes in the registry, if any.

Modeling MOST. The behavior description in the requirements document for MOST is given in the form of "high-level MSCs from the view-point of a particular process". Interestingly, the "high-level MSCs" in the requirements document are actually not HMSCs. In reality, they are rather *mistakenly* called as "high-level MSCs", and they reflect -at a very high level- the control flow *within* a process. In addition, several "scenario MSCs" in system execution are provided in the requirements document. The "scenario MSCs" of the requirements document correspond to a sequence of MSCs in our modeling, that is, a scenario in system execution. Using

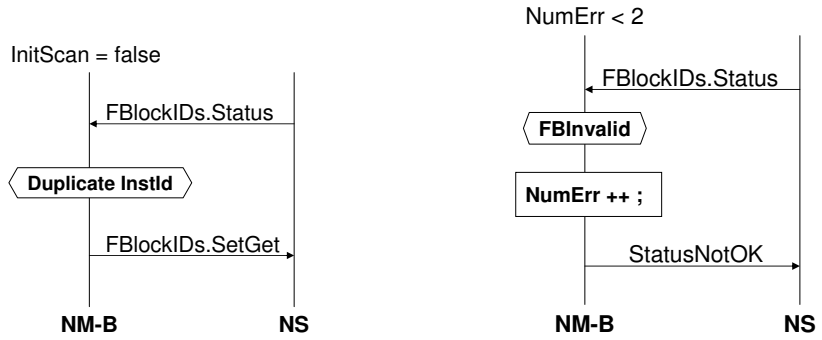


(a)



(b)

Figure 7-1: Transition system fragments: (a) Network Slaves: TS_{NS}, and (b) Network Master process responsible for receiving configuration from slaves: $TS_{\mathrm{NM-B}}$. Transaction roles are not shown in action labels to reduce visual clutter.



(a) (b)

Figure 7-2: MSCs (a) FBRcvDuplId, (b) FBInvldSetNotOK.

these scenarios, we elaborated on the very high-level per-process control flow given by the "high-level MSCs" of the document. This gave us the labeled transition system for each process class and the MSCs corresponding to the action labels of these transition systems. There was however some additional work involved in our modeling since the requirements document used both message passing as well as shared variables for inter-process communication, whereas our modeling language does not support shared variables.

As described earlier (Section 7.1), the MOST system consists of a network-master NM and several network-slaves (NS). We modeled NM using two process classes: 'NM-A', for requesting configuration from slaves and 'NM-B' for receiving responses from slaves. Also, environment is modeled as a separate process class, which takes part in network on/off events and causes the network change event (NCE) whenever a node leaves or joins the network. All network slaves are modeled as a single process class 'NS'; the number of slaves can be varied easily by changing the number of objects of this class.

Fragments of the transition system descriptions of the NS class, and the NM-B (process responsible for receiving the configuration information from slaves), are shown in Figure 7-1. The edges in these transition systems are labeled with transactions or guarded MSCs. Recall that the edges in the high-level transition systems of a process class in our IPC model are labeled with (transaction, role) pairs; here we have not shown the roles in each transaction to reduce visual clutter. The transactions corresponding to various transitions were easily obtained from the descriptions given in the requirements document. However obtaining the control flow for the per-process transition systems was not so straightforward. The "scenario MSCs" in the requirements document (which correspond to sequences of MSCs describing overall system behavior) were helpful in modeling the per-process control flow correctly.

For illustration, two transactions– *FBRcvDuplId* and *FBInvldSetNotOK*, from the MOST example are shown in Figure 7-2. Transaction *FBRcvDuplId* (Fig. 7-2 (a)) represents a scenario in which the address information provided by a network slave clashes with that of another slave which has replied earlier. Note that, in this transaction lifeline NM-B has a boolean guard $InitScan = false$. Thus, in order to execute this transaction, NM-B's local variable *InitScan* must be *false*. In case of transaction *FBInvldSetNotOK* (Fig. 7-2 (b)), a network slave replies with an invalid address information. This represents an error situation, such that NM-B keeps track of the error count using variable *NumErr*. If $NumErr < 2$, then this transaction is executed setting the system state to 'NotOK'.

Overall, the IPC specification for MOST comprises of *53 transactions* (or guarded MSCs) with process classes NS, NM-A and NM-B containing 18, 6 and 21 control states respectively.

Example For illustration, consider process class NS with 50 network-slaves in a MOST network. Assume that currently all slaves are residing in the control state *S8* of its transition system TS_{NS} in Figure 7-1 (a), while process NM-B is in the control state *RcvRegist* of its transition system $TS_{\mathrm{NM-B}}$ (Fig. 7-1 (b)). Let the values of NM-B's local variables *InitScan* and *NumErr* be *false* and *0* respectively. Further, assume that there are no local variables in the NS class and no regular expression guard for any transaction. Then, this configuration corresponds to

$$c = \{\langle S8 \rightarrow 50\rangle, \langle (\mathit{RcvRegist}, \mathit{InitScan} = \mathit{false}, \mathit{Numerr} = 0) \rightarrow 1\rangle\}.$$

For process class NS, its behavioral partition is described using only the control state *S8* from TS_{NS}. For process class NM-B, a behavioral partition consists of control state *RcvRegist* from $TS_{\mathrm{NM-B}}$ and the values of its local variables *InitScan* and *NumErr*. Note that we have only shown behavioral partitions with non-zero objects, which is how we keep track of them during simulation. Also, we have omitted NM-A's state for the purpose of illustration. We can now execute the transaction *FBInvldSetNotOK* (transition $S8 \rightarrow S4$ in Figure 7-1(a) and $\mathit{RcvRegist} \rightarrow S1$ in Figure 7-1(b)) which is shown in Fig. 7-2 (b), with any one slave object executing the lifeline marked NS.

The resulting configuration is

$$c' = \{\langle S4 \rightarrow 1, S8 \rightarrow 49\rangle, \langle (S1, \mathit{InitScan} = \mathit{false}, \mathit{Numerr} = 1) \rightarrow 1\rangle\}.$$

Thus, the slave object participating in *FBInvldSetNotOK* moves to control state *S4* in TS_{NS}, while NM-B moves to control state *S1* in $TS_{\mathrm{NM-B}}$ with the value of its variable *NumErr* getting updated to 1. This captures the basic idea in our symbolic execution semantics (note that we did not maintain the local states of 50 slave objects separately).

7.2 Meeting Test Specifications

In this section we describe the automatic generation of test cases from an IPC model based on a user-provided test case specification. The user gives *a sequence of transactions*, as a test specification. The test generation procedure makes use of guided search to generate a transaction sequence containing the user-provided test specification sequence as a *subsequence*. Note that it is possible that there is no execution sequence that can satisfy a given test specification.

7.2.1 Problem Formulation

The user gives a sequence of transactions $\tau_1, \tau_2, \ldots, \tau_n$ as the test specification. The test-case generation procedure aims at producing one or more test sequences of the

form– $\tau_1^1, \ldots, \tau_1^{i1}, \tau_1, \tau_2^1, \ldots, \tau_2^{i2}, \tau_2, \ldots, \tau_n^{in}, \tau_n$.

This problem can be viewed as finding a witness trace in the IPC model satisfying the Linear-time Temporal Logic (LTL) [23] property $\mathbf{F}(\tau_1 \wedge \mathbf{F}(\tau_2 \wedge (\ldots (\mathbf{F}\tau_n) \ldots)))$. We always generate only finite witness traces (*i.e.*, a finite sequence of transactions) such that any infinite trace obtained by extending our finite witness trace will satisfy the above-mentioned LTL property. This can be accomplished by standard search strategies like breadth-first or depth-first search. Breadth-first search produces shortest-possible test traces (the sequence of MSCs generated), but it is expensive in time and memory. On the other hand, *depth-first* search can help us find test-cases efficiently, but the generated sequence of MSCs may not be optimal in length. Hence, we investigate intelligent search heuristics for this problem.

7.2.2 A^* search

Various well-known heuristic search strategies such as best-first, and A^* (pronounced A-star) [84] have been shown to be useful in *test-case generation* [92], and *model checking* [29]. The heuristics mainly differ in the evaluation function used by them, which gives the "desirability" of expanding a node in the state graph. The search proceeds by expanding the graph choosing the most desirable node first. The evaluation function $f(s)$ used in A^*, which evaluates the state s during state-space exploration (lower score is better), consists of two parts– (i) $g(s)$, giving the shortest generating

path length for state s, i.e. length of the shortest path from start-state to s, and (ii) $h(s)$, the *estimated* cost of the cheapest path to the goal state from s. The evaluation function $f(s) = g(s) + h(s)$ gives the estimate of cheapest path from start state to the goal state, passing through s. If $h^*(s)$ is the actual cost of the cheapest path from a state s to the goal state and $h(s) \leq h^*(s)$, for all states s, then the heuristic is said to be *admissible* and guarantees to find the shortest path to goal state, if one exists.

We adapt and modify the A^* algorithm to guide our test-selection process. While searching for witness traces, we break the search into steps, such that each step aims at generating a transaction sequence up to the next uncovered transaction in the test specification. So if the test specification is $\tau_1, \tau_2, \ldots, \tau_n$ and so far the search has produced a witness for $\tau_1, \ldots, \tau_{i-1}$ (where $i \leq n$), the "goal" is the next transaction in the test specification — τ_i. The search for this "goal" will of course start from a state appearing at the end of the witness trace found for $\tau_1, \ldots, \tau_{i-1}$. Then, for a state s visited while searching for a path to τ_i, the value of function $g(s)$ in A^*'s evaluation function determines the length of the shortest path to s seen so far from the initial state, and covering the test specification transactions that have already executed (in the order of their occurrence). While, for computing the h function, we only focus on the next goal (τ_i) instead of the whole remaining sequence to be covered ($\tau_i, \ldots, \tau_n$).

During the search we maintain a global search tree $\mathcal{T}$ capturing the states visited so far. A **global state** and the **successor** of a global state, appearing in $\mathcal{T}$ during the test generation are defined as follows.

Definition 18. Global state *Given an IPC system model and a test case specification* $\tau_1, \ldots, \tau_n$, *a* **global state** $s = (c, i)$ *consists of: (i) an abstract system* **configuration** c *in the IPC model (see Section 3.4) and (ii)* i, *the index of the next* **goal transaction** τ_i *(while searching for the test case) in the test case specification, where* $1 \leq i \leq n$.

Definition 19. Successor state *Given a* global state $s = (c, i)$ *as defined above, a global state* $s' = (c', i')$ *is a successor state of* s *if and only if: (i)* c' *is obtained from* c *in our symbolic execution semantics by executing some transaction* τ', *and (ii)* $i' = i + 1$ *if* $\tau' = \tau_i$ *(i.e. the next goal transaction), and* $i' = i$ *otherwise. We use* **Succ(s)** *to denote the set of all possible successors of* s.

In the search tree $\mathcal{T}$, each edge from a state s to its successors is labeled with the transaction name that was executed at s leading to that successor. Also for each state s in $\mathcal{T}$ we maintain the values $g(s)$ and $h(s)$.

Computation of heuristic function h. To compute $h(s)$ for a given a state $s = (c, i)$, we consider the process classes involved in the transaction τ_i, that is, the process classes whose objects should appear as the lifelines of the MSC in τ_i (the next goal transaction). Let this set of classes be $classes(\tau_i)$. For each process class $p \in classes(\tau_i)$, we determine the length of the shortest path in TS_p from current control state(s)[1] of p-objects to the source state(s) of the transition(s) which appear

[1] TS_p is the transition system describing process class p.

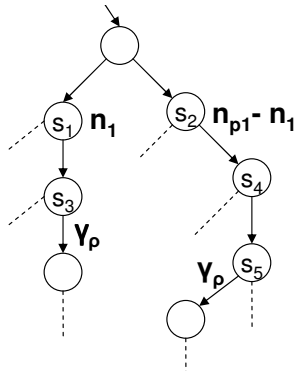


Figure 7-3: Transition system fragment for process class p_1

as a lifeline in τ_i. Note that, we pre-compute the shortest paths between all state-pairs in the Labeled Transition Systems (LTSs) of different process-classes once and for all, at the beginning of the test generation. Clearly, different objects of a process class p can be in different control states — so we need to consider all the control states in which any object of a process class p is currently in. Let the shortest distance computed in this way for process class p be denoted as $d_p^{\tau_i}$.

For illustration, consider a fragment of transition system describing a process class p_1 as shown in Figure 7-3. Assume that p_1 has n_{p1} objects such that n_1 (> 0) p_1-objects are currently in control state s_1, while remaining $n_{p1} - n_1$ objects are in control state s_2 (see Fig. 7-3). Let the next goal transaction be γ and (p_1, ρ) be the only lifeline involving process class p_1 in transaction γ (*i.e.* $p_1 \in classes(\gamma)$). Now, in the transition system of p_1 (fragment of which is shown in Fig. 7-3), the shortest paths to the control state(s) with an outgoing transition γ_ρ from states s_1 and s_2 respectively are— $s_1.s_3$ and $s_2.s_4.s_5$, having path lengths *one* and *two* (see Fig. 7-3). Hence, we get $d_{p1}^{\gamma} = 1$.

Finally, we define

$$h(s) \stackrel{def}{=} max_{p \in classes(\tau_i)} d_p^{\tau_i} \quad \text{and} \quad f(s) = g(s) + h(s) \tag{7.1}$$

Intuitively h gives a lower bound on the trace length required to execute τ_i from s, and therefore, gives us an *admissible* heuristic.

7.2.3 Test generation Algorithm

We now discuss the overall test generation procedure described using two functions, one top level or driver function **genTest** presented in Algorithm 1 which makes use of the second function **genTrace** described in Algorithm 4 (appears in Appendix B). In particular, the function *genTest* (Algorithm 1) takes as input three parameters: a set S of global states, i is the index of the current goal transaction and n is the length of the test-case specification $\tau_1, \ldots, \tau_n$. During test generation, states $s = (c, i)$ from the set S are evaluated (i.e. $f(s)$ is computed using Equation 7.1), and the one with the minimum value of function f is chosen and explored (i.e. all its successors are generated). The initial call to this procedure is `genTest({`$\mathtt{s}_{\text{init}}$`},1,n)` where $\mathtt{s}_{\text{init}} = (c_{init},1)$, c_{init} is the initial *configuration* in our IPC system model. The search tree $\mathcal{T}$ is also initialized, having a single root node s_{init}.

For finding a trace from states in the set S to the next goal transaction τ_i, *genTest* calls `genTrace(`S`,`τ_i`)` (line 2 of Algorithm 1). The function *genTrace* described in Algorithm 4 then tries to find a path leading to the execution of transaction τ_i from

Algorithm 1: **genTest**(S,i,n)

```
1: while S ≠ φ do
2:   ⟨GoalStates, UnExplored⟩ ← genTrace(S,τ_i)
3:   if GoalStates = ∅ then
4:     if i = 1 then
5:       report Failure and Exit
6:     else
7:       return
8:     end if
9:   else if i < n then
10:    genTest(GoalStates,(i + 1),n)
11:    S ← UnExplored
12:  else
13:    output witness(GoalStates) and Exit /*i = n*/
14:  end if
15: end while
```

states in S (see Appendix B). It returns: (a) *GoalSet*, the set containing the states reached after successfully finding a path from a state in S to the transaction τ_i, and (b) the set *UnExplored*, which contains states that were reached but not explored when searching for the trace. If *GoalSet* is empty, this means that *genTrace* failed to generate a trace from S reaching τ_i. In this case, if $i = 1$, the *genTest* algorithm reports failure and exits, since it cannot find any trace up to the first transaction in the test specification; otherwise, the *genTest* algorithm backtracks (line 7 of Algorithm 1) and tries to find another trace for the previous goal transaction using the remaining unexplored states from that step. This occurs by assigning the set *Unexplored* to S when a failed recursive call returns (line 11 of Algorithm 1). The new S is explored further due to outermost **while** loop. If *GoalSet* is not empty, *genTest* is called recursively using the set *GoalSet* as the starting states for finding the next transaction

(τ_{i+1}) in the test specification. In this way if we have encountered τ_n and *GoalSet* is still not empty, a witness trace satisfying the test specification has been found. The *genTest* algorithm then outputs the witness trace and exits.

7.3 Experimental Results

We now discuss various results from the experiments we performed for (a) generating test cases corresponding to various test specifications, and (b) comparing test case generation for symbolic vs concrete model execution for a given test specification.

Witness trace generation. Besides the MOST network-management protocol, we also performed experiments using the following four examples, which were discussed earlier in Section 3.7– *Telephone Switch*, *Rail-Car*, *Automated Shuttle*, and *Weather Controller*. For the purpose of experiments, we considered *three* test specifications for all examples modeled, each attempting to cover a meaningful use-case. Test cases were successfully derived for these test specifications using the A^*-based heuristic approach. Experimental results showing the witness trace lengths and test-generation times corresponding to various test specifications are shown in Table 7.1 in the columns under 'RESULTS A'. Recall that each witness trace contains the corresponding test specification as a subsequence. All experiments were performed on a machine with 3GHz CPU and 1 GB of memory.

We also generated witness traces for the same test specifications using breadth-

first (BFS) and depth-first strategies (DFS). In most of the cases A^* based approach generated optimal length traces, i.e. same as those generated using BFS[2], taking only a fraction of time. On the other hand, test cases generated using DFS were up to 3 times longer than those generated using A^* heuristic.

			RESULTS A		RESULTS B				
Example	Test Spec.		Witn. trace	Test gen.	Expl.	# Witnesses		Exec. Times(sec)	
	#	Length	length	Time(sec)	Depth	**S**	**C**	**S**	**C**
MOST (3 slaves)	1	2	28	67	30	16	–	272	> 10 min.
	2	2	19	4.5	19	7	21	28	480
	3	3	32	312	32	4	–	460	> 10 min.
Telephone Network (5 phones)	1	2	4	0.04	5	70	560	1	28
	2	4	14	8.1	14	–	–	> 10 min.	> 10 min.
	3	4	9	1.42	9	12	80	17	212
RailCar (6 cars, 3 terminals)	1	3	17	136	18	6	–	380	> 10 min.
	2	3	12	5	15	32	–	19	> 10 min.
	3	3	11	4	15	62	–	19	> 10 min.
Automated Shuttle (5 shuttles)	1	4	14	0.1	15	5	11	0.7	15
	2	3	12	0.07	15	9	18	0.72	15
	3	4	23	0.91	23	12	–	78	> 10 min.
Weather Controller (10 clients)	1	2	4	0.02	20	5	129	0.1	2.5
	2	3	13	0.03	20	2	3	0.1	2.5
	3	3	15	0.03	25	3	7	0.15	12

Table 7.1: RESULTS A: Witness test generation– test lengths & generation times. RESULTS B: Comparing Symbolic/Concrete test generation, S ≡ Symbolic, C ≡ Concrete.

For illustration, let us consider the following test specification from our modeling of MOST protocol (corresponding to the first test specification for MOST in Table 7.1): *FBRcvDuplId, ConfigStatusOk*. As discussed in Section 7.2, for a given test specification (given as a sequence of transactions), test generation procedure attempts to find a witness trace (another sequence of transactions) containing the test

[2]Given a test specification $\tau_1, \ldots, \tau_n$, we employ A^* based search to find a smallest path to τ_1, search from that occurrence of τ_1 to find a smallest path to τ_2 and so on. Clearly, adding up these smallest paths may not produce the minimal path containing $\tau_1, \ldots, \tau_n$ as a subsequence.

specification as a subsequence. The transaction *FBRcvDuplId* (transition $S8 \rightarrow S8$ in Figure 7-1(a) and *RcvRegist* $\rightarrow$ *RcvRegist* in Figure 7-1(b)) corresponds to the scenario shown in Figure 7-2 (a)– which takes place when the configuration information sent by slave to master clashes with the configuration information for some other slave node already registered with the master. In response, when it is not the first time that the system is being scanned after the network was powered on (as indicated by the guard 'InitScan = false' of lifeline NM-B in Figure 7-2 (a)), network master assigns a new id for this slave and sends this value for acceptance to the slave via message *FBlockIDs.SetGet*. This new value may be accepted or rejected by the slave node. In case it is accepted, then the new value is entered in the central registry by NM-B. The transaction *ConfigStatusOk* (whose MSC is not shown here) corresponds to the communication of 'SystemState = OK' by network master to the slave nodes, once all the nodes have responded correctly. As shown in Table 7.1, a test-case MSC sequence of length 28 was obtained within 67 seconds for this test specification. In this manner, the test specifications, though small in length, can be used to derive test cases representing complex system behaviors.

Symbolic nature of our tests. Recall that our IPC models inherently support *symbolic* execution of process classes (Section 3.4) and hence generation of symbolic tests where processes are grouped together in terms of behavior. To evaluate the advantage of symbolic test generation, we generated all possible witness traces corre-

sponding to the given test specifications using both *symbolic* and *concrete*[3] execution semantics. This was done by exploring various system models up to a given depth bound. For a given test specification, in order to guarantee the generation of at least one witness trace during exploration, the length of its witness trace derived from A^*-based test generation was used as a cut-off for the depth bound.

The results comparing the test-suite size and test generation times appear in Table 7.1 in the columns under 'RESULTS B'. As we can observe, the time to generate the test cases is much lower in symbolic execution as compared to concrete execution. In fact, for almost half of the cases the concrete execution did not even terminate within 10 minutes (shown as '–' in Table 7.1 under RESULTS B). More importantly, using symbolic execution many behaviorally similar tests were grouped together into a single test case. For example, in case of the first test specification for the Telephone Network example (see Table 7.1, RESULTS B), 560 concrete test cases are grouped into 70 symbolic tests. Note that the number of concrete tests is not always an exact multiple of the number of symbolic tests. This is because different symbolic tests may be blown up into different number of concrete tests.

Use of our tests We note that the tests generated from our IPC model can be executed on a distributed system implementation, that is, executable code in a programming language. This is the case where the system implementation is generated manually using the informal system requirements as a guide. In this case, the tests

[3]The state of each object is maintained separately in concrete execution semantics.

increase our confidence in system implementation. Alternatively, it is conceivable that the system implementation is generated from the system model, and so are the test cases. In this case, the tests can be used to increase our confidence in the system model itself (which is generated manually from the informal requirements). For more detailed discussion on the use of model-based tests in model-driven software development, the reader is referred to [42].

Chapter 8

Test Generation from SMSC

In this chapter, we present a method for model-based testing of reactive systems of process classes based on 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 objects. In general the requirements specification for such systems only describe the interactions between classes of objects and do not constrain their size, which may be large and can even vary dynamically with time. Therefore, 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 (which typically focus on inter-class communication) using SMSCs. As discussed earlier in Chapter 4, SMSCs extend Message Sequence Charts (MSCs) with the concept of a symbolic lifeline. Unlike MSCs, where a lifeline represents a concrete object, a symbolic lifeline in SMSCs may represent a group of objects from a class. This extension, along with an *abstract* execution semantics, allows SMSCs to succinctly specify and simulate inter-process behavior in systems consisting of classes of large (even unbounded) number of objects. Apart from a SMSC-based system model, a second input to our testing framework is a user-provided test-purpose that aids in selecting interesting behaviors against which the user wishes to test a system implementation. The test-purpose, which may include *negative* messages denoting forbidden behavior, is also modeled using SMSCs.

Given a system model and test-purpose, our test generation method first auto-

matically generates a set of *abstract* test cases that satisfy the purpose. These are generated by exploring the system model to determine execution traces that contain all the test-purpose events (except the negative events) in the appropriate order, possibly interspersed with other events from the model. 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 corresponding to the test-purpose is then generated by instantiating lifelines in the templates with 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.

To summarize, the main technical contribution of the current chapter is a model-based testing methodology, targeting reactive systems consisting of many behaviorally similar objects. It consists of automated generation of *abstract* test cases, their step-wise refinement to concrete test cases through *template generation* and test execution. Our approach avoids *(a) deriving different system models representing different configurations for test generation* and, *(b) redundant generation of behaviorally similar test cases.*

Organization In the next section, we introduce the case-study used for illustrating our approach and also describe our test-purposes, while Section 8.2 provides an end-to-end view of our test generation process using a running example. Section 8.3 discusses the technical details of our approach, including test generation algorithms. In Section 8.4 we elaborate our test execution setup and provide empirical validation of our approach in Section 8.5 on a real life case-study.

8.1 Test-purpose specification

In this section, we first present the CTAS case-study, which is used as a running example throughout the chapter. We then discuss a test purpose, which is specified using an extended version of SMSCs, and is used for selecting the relevant test case(s) from a SMSC model. A SMSC model and test purpose are the two inputs to our test

generation method.

8.1.1 CTAS Case Study

Note that, the CTAS case study was discussed earlier in Section 4.2, while discussing the SMSC notation. For the purpose of our discussion, we briefly describe it again in the following.

The CTAS or *Center TRACON Automation System* is a set of tools developed at NASA to aid the air traffic controllers in managing high volume of air traffic flows at large airports. Various processes such as TS (Trajectory Synthesizer), RA (Route Analyzer) etc. in the CTAS system require latest weather updates for their functioning. The weather updates are provided to these processes by WCP (Weather Control Panel) via the CM (Communications Manager) which is the central controller responsible for communications among these processes. Both WCP and CM are also part of the CTAS system. We refer to various processes requiring the weather updates simply as Clients. Thus, we consider the CTAS system to be consisting of three classes of objects– (i) WCP and (ii) CM classes with one object each, and (iii) Client class consisting of multiple client objects.

The CTAS requirements for the weather control logic are described informally in English in [1]. The requirements are structured as short snippets, describing communication scenarios among CM, WCP and Client objects. Each such requirement

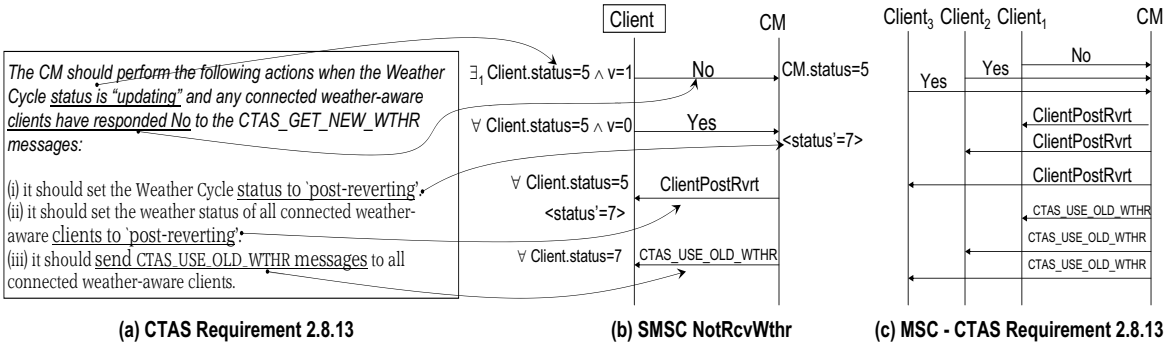


Figure 8-1: A CTAS requirement and its modeling as an MSC and a SMSC.

first states a pre-condition, followed by a set of events to be executed when the given pre-condition holds. For illustration, we partially reproduce a CTAS requirement (Requirement 2.8.13) in Figure 8-1(a). It describes the events, stated as items (i), (ii), and (iii), to be executed when the pre-condition (specified in *italics*) holds true.

As described in Chapter 4, the language of Symbolic Message Sequence Charts (SMSCs) [101] is a light-weight extension of the MSC notation, having the following key features– i) the notion of a *symbolic* lifeline, and ii) an *abstract* execution semantics. In the case of MSCs, a lifeline can represent only a concrete object [62], thus making it difficult to capture scenarios with a large number of concrete objects. For illustration, consider the MSC shown in Figure 8-1(b). It represents a scenario with 3 Client objects corresponding to the CTAS requirement 2.8.13 (Fig. 8-1(a)). Here one of the client objects ($Client_1$ in this case) replies in negative to a request sent by the controller CM, with the remaining clients replying in positive. The SMSC corresponding to the CTAS requirement 2.8.13 in Fig. 8-1(a) is shown in Figure 8-1(c). Comparing with the MSC in Fig. 8-1(b) capturing the same requirement, all Client

objects are now represented using a unique *symbolic* lifeline.

The complete CTAS system description is obtained by first deriving SMSCs corresponding to various requirements (as illustrated above) and then composing together these SMSCs to form a High-level SMSC (HSMSC). The HSMSC for CTAS case-study appears in Figure 8-2. This HSMSC is flat, i.e. all its nodes represent a SMSC (and not another HSMSC). For example, SMSC *NotRcvWthr* shown in Figure 8-1(c) corresponds to a node (encircled in bold lines) in the CTAS HSMSC.

Recall that the complete SMSC system model is represented as $\mathbf{Spec} = \langle H, \bigcup_{p \in \mathcal{P}} \{V_p, v_p^{init}\} \rangle$, where H is a HSMSC describing the interactions among process classes $p \in \mathcal{P}$, and V_p denotes the set of variables of a class p with v_p^{init} giving an initial assignment of values to objects of p (see Chapter 4).

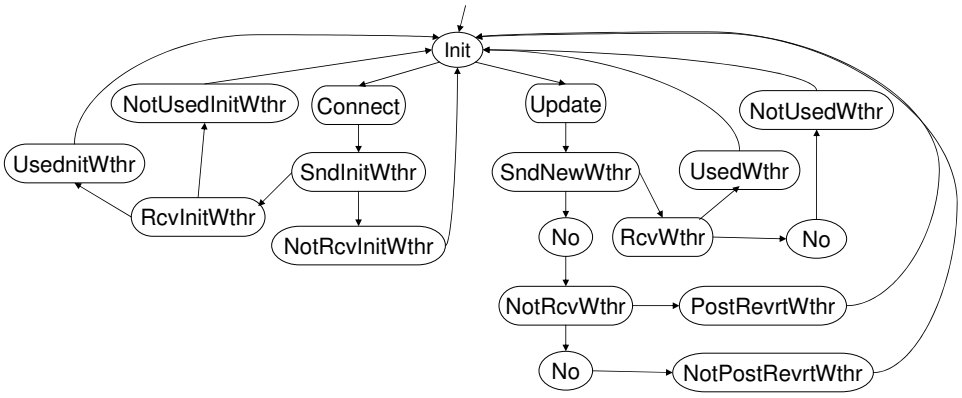


Figure 8-2: HSMSC for the CTAS case study.

Let $\mathcal{C}_{\text{Spec}}$ denote the set of all possible abstract system-states[1] of a SMSC model **Spec**, Σ_{Spec} be a set of process expressions describing the HSMSC model, and Act_{Spec}

[1]In our description of SMSCs in Chapter 4, abstract system-states were referred to as abstract configurations (4.4).

denote the set of events appearing in **Spec**. Then, in the current chapter, we use transition relation '$\rightarrow_{\text{Spec}} \subseteq (\mathcal{C}_{\text{Spec}} \times \Sigma_{\text{Spec}}) \times Act_{\text{Spec}} \times (\mathcal{C}_{\text{Spec}} \times \Sigma_{\text{Spec}})$' to describe the abstract execution semantics of **Spec**, moving from one abstract system-state to another by executing a SMSC event enabled at the current process expression (and also determining the resulting process expression). The details of SMSC operational semantics were discussed earlier in Chapter 4.

8.1.2 Test-purpose Specification

Once a system model **Spec** has been derived from the informal system requirements using SMSCs, a test-purpose is used to drive the test-generation process. A test-purpose [112] 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 8-3(a) 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-

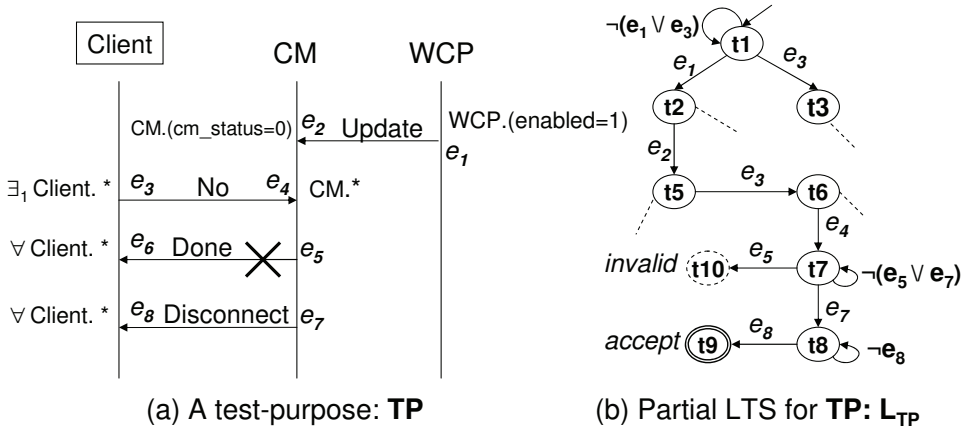


Figure 8-3: A test-purpose and its LTS.

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 8-3(a).

For illustration, consider the test-purpose SMSC shown in Figure 8-3(a). 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. 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.

For the purpose of test generation, we are checking whether an execution trace of a system model **Spec** contains a linearization of the test-purpose S_{TP} as a subsequence. To obtain the test-purpose linearizations we consider the operational semantics for MSCs [62, 98], and extend them to handle the forbidden events. All these event linearizations are captured using a labeled transition system (LTS) L_{TP}. A trace in L_{TP} either ends with a forbidden send event leading to an *invalid* state, or it contains all the test-purpose events except for the forbidden events and ends in an *accept* state. Thus, if the send event of a forbidden message is encountered in the system model (during test generation) and there is an outgoing transition labeling this send event from the current state of test-purpose LTS L_{TP}, then L_{TP} will move to an *invalid* end state. For example, the LTS corresponding to the test-purpose shown in Figure 8-3(a) appears partially in Figure 8-3(b). In state $t7$ of the LTS, we see three possibilities — (a) event e_5 occurs (which corresponds to sending of forbidden message *Done*) leading the test-purpose LTS to an invalid end state $t10$, (b) the next non-forbidden event in the test-purpose, event e_7 (sending of message *Disconnect*) occurs, progressing the test-purpose eventually to an accept state $t9$, or (c) any other event occurs, leaving the test-purpose LTS in its current state $t7$.

Let Act_{TP} be the set of events appearing in the test-purpose SMSC S_{TP}. Then, we define $L_{\text{TP}} = (T, \rightarrow_{\text{TP}}, t_1, I, A)$, where T is the set of L_{TP} states, $\rightarrow_{\text{TP}} \subseteq T \times Act_{\text{TP}} \times T$ is the transition relation describing L_{TP}, t_1 is the initial L_{TP} state, $I \subseteq T$ is the

(possibly empty) set of *invalid* states, and $A \subseteq T$ is the set of *accept* states such that $I \cap A = \emptyset$.

8.2 Test Generation Overview

In this section, we describe the steps in our test generation methodology (see Figure 8-4) with the help of an example based on the CTAS case-study discussed earlier in Section 8.1. The goal is to provide readers with a high-level overview of the end-to-end process, before the technical details of the approach are presented in Section 4. The test generation starts with the following two inputs– (i) a system model **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** (see Section 8.1). The overall flow of our test generation method appears in Fig. 8-4. We now briefly discuss the three steps of this method.

8.2.1 Deriving abstract test case SMSC

Given the system model (as an HSMSC) and the test-purpose (as an SMSC), we first generate a set of abstract test cases in the form of SMSCs. The abstract test generation procedure (described in Section 4.1) involves execution of system model **Spec** guided by the test-purpose **TP** (oval $\boxed{1}$, Fig. 8-4). 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

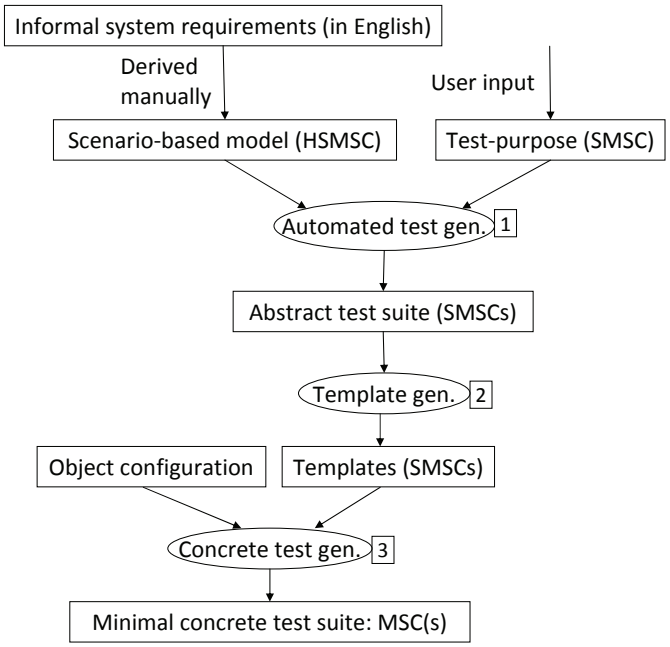


Figure 8-4: Overall test generation flow.

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 8-3(a) appears in Figure 8-5(a). The message names appearing in bold italics in the test case SMSC (Fig. 8-5(a)) represent the matching events in the test-purpose (Fig. 8-3(a)). To reduce visual clutter, we have omitted the object selectors and post-conditions 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 8-5(a). The abstract test case shown in Figure 8-5(a) represents an unsuccessful weather update scenario for the Clients connected to CM.

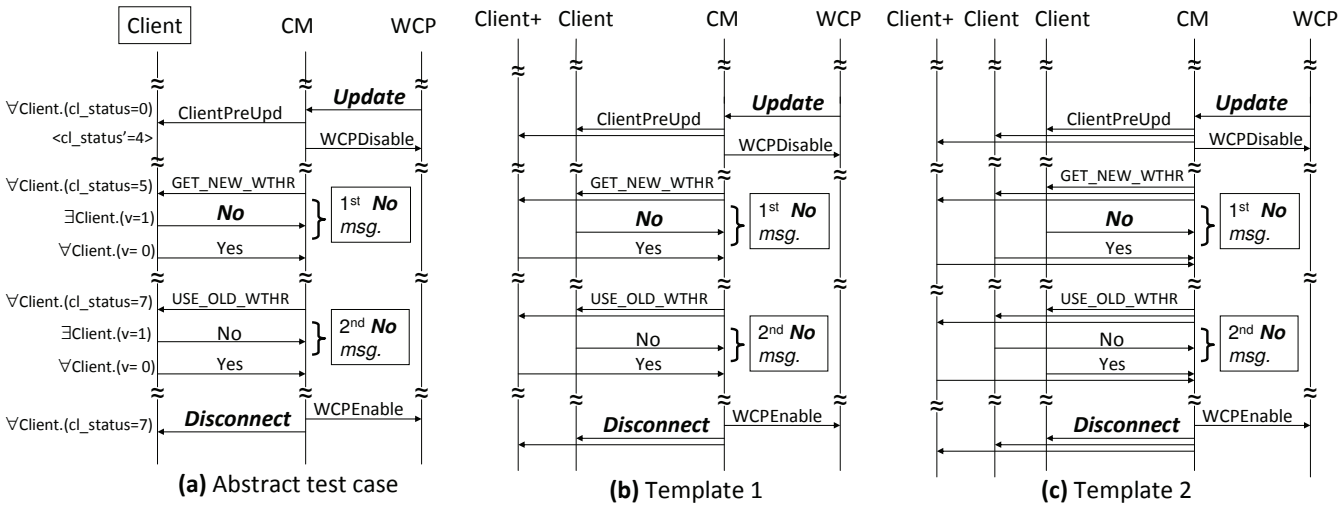


Figure 8-5: An abstract test case (corresponding to test-purpose shown in Fig. 8-3(a)) and two templates for it.

8.2.2 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. A straightforward approach for doing this is by executing an abstract SMSC S with an initial configuration comprising of concrete objects. During execution, we maintain the individual state of each object, instead of monitoring only the count of objects in a given state as in the abstract SMSC semantics (explained in Section 2). We will thereby obtain different MSC execution traces that will represent various possible concrete instantiations of the abstract test case, for the given configuration. In such a concrete test case MSC, a symbolic lifeline representing process class p in S, is replaced by concrete lifeline(s) representing the p objects specified in the initial configuration. However, this 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 clients shown in Fig. 1(b), we will get a new MSC scenario, which will however, depict the same core behavior (a client object sending a *No* message).

Third, the whole test generation 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.

In order to avoid the above drawbacks, we introduce an intermediate representation between abstract and concrete test cases, called *templates* (oval 2, Fig. 8-4). 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 8-5(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 8-5(a) are shown in Figures 8-5(b) and 8-5(c). Figure 8-5(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 8-5(c) depicts the case when two different Client objects (both labeled 'Client') send the two *No* messages to the CM. In both these cases, all other Client objects are symbolically represented by the *marked lifeline* ('Client+'); they execute the other events shown on the Client lifeline in the abstract test case, 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 is evident 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 8-5(b) is 2, and that from the template in Figure 8-5(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. Further, in Section 4.2 we will present an algorithm to automatically generate *all* possible templates from an abstract test case.

3. Concrete test cases for different object configurations (differing in number of objects in certain classes) are 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, as explained in Section 4.3.

We now formally define templates and present some key properties.

Definition 1 (Templates)**.** *Let S be an abstract test case SMSC. The* **templates** *derived from S is a set $\mathcal{T}$ of MSCs where — a template captures a projection of events from a symbolic lifeline l_p representing process class p in S, to one or more lifelines from process class p (represented as C_p) such that,*

1. *The projected events (from l_p to lifelines in C_p) follow the top-down event ordering along l_p.*

2. *An existential event from l_p appears along exactly one lifeline in C_p, while a universal event from l_p appears along all lifelines in C_p such that– the event sequence (from top to bottom) along a lifeline l in C_p captures a feasible execution path in the control-flow of process class p (i.e. events along l can be executed by a single p-object).*

A lifeline with only universal events projected along it is called a **marked** *lifeline, while we refer to the remaining non-marked lifelines as* **concrete** *lifelines.*

Note that, while a concrete lifeline denotes exactly one object, a marked lifeline from process class p represents *all* objects of class p other than those representing the concrete 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 assigned to any concrete p-lifeline) participate in the given template by executing some common events.

Since our template generation involves checking feasible control flows, a comment on this matter is in order. Our template generation procedure does not involve expensive static checks on infeasible/feasible control flows. Instead, we *execute* the abstract test case SMSC (using SMSC operational semantics), and infeasible flows are found in course of the execution. The detailed description of our template generation algorithm appears later in Section 8.3.2.

We now state some key properties of our templates.

Theorem 1. *Let S be an abstract test case SMSC and $\mathcal{T}$ be the set of templates derived from S. Then,*

1. *For a process class p, a template in $\mathcal{T}$ contains at most one marked lifeline.*
2. *The number of lifelines in each template in $\mathcal{T}$ is finite.*
3. *The number of templates in $\mathcal{T}$ is finite (i.e. $|\mathcal{T}| \in \mathbb{N}$).*

Proof. Let n_e^p (n_u^p) be the total number of existential (universal) events from process class p appearing in the abstract test case SMSC S.

1. *For a process class p, a template in $\mathcal{T}$ contains at most one marked lifeline.* We prove this by contradiction. Let there be more than one marked lifeline from process class p in a template. Recall that each lifeline in a template represents a feasible execution path in p's control flow. Further, events along a marked lifeline will be executed by all the p-objects other than those assigned to the concrete (or unmarked) lifelines. Now, all p-objects are initially in the same execution sate and it is not possible for any p-object to simultaneously execute along more than one path in p's control flow. Hence, the contradiction.

2. From the definition of templates (Defn. 1, p. 202) we know that each existential event in the abstract test case SMSC S appears along exactly one (concrete) lifeline in a template. Hence, the total number of concrete lifelines from process class p in a template is bounded by the number of existential p-events in S, i.e. $|n_e^p|$. Further, from (1) above we know that there can be at most one

marked lifeline from class p in a template. Hence, the number of p-lifelines in a template lie between 1 to $1+|n_e^p|$, thus bounding the total number of lifelines in a template.

3. From (2) 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 templates (see Defn. 1, p. 202) we know that a universal p-event appears along all lifelines (including the marked p-lifeline) if permitted by the p's control flow. Therefore, to determine an upper bound on the number of templates of 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 we denote this quantity as U_p^k, which is same as 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 $S_{|n_e^p|,k}$ defined as follows [67]:

$$(1) \quad S_{n,1} = 1, S_{n,n} = 1$$

$$(2) \quad S_{n,m} = m \times S_{n-1,m} + S_{n-1,m-1}$$

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 processes appearing in the abstract test case S.

□

8.2.3 Deriving concrete tests

The final step in our test generation process involves deriving concrete test case MSCs from various templates (oval 3, Fig. 8-4). 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 2 (Object configuration)**.** *Let* $\boldsymbol{Spec} = \langle H, \bigcup_{p\in\mathcal{P}}\{V_p, v_p^{init}\}\rangle$ *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 M by simply assigning concrete objects from O_p to lifelines corresponding to class p in the template. For each concrete lifeline l in template M 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 (Thm. 1).

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 8-5. In the first template (Fig. 8-5(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. 8-5(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+'.

We now elaborate on our test generation method.

8.3 Test Generation Method

In the following, we elaborate the various steps in our test generation methodology. The steps include abstract test generation (oval 1, Fig. 8-4), template generation (oval 2, Fig. 8-4) and concrete test generation (oval 3, Fig. 8-4).

8.3.1 Abstract test-case generation

Let $\mathbf{Spec} = \langle H, \bigcup_{p\in\mathcal{P}}\{V_p, v_p^{init}\}\rangle$ be a system model with process class $p \in \mathcal{P}$ having $n_p \in \mathbb{N} \cup \{\omega\}$ objects, and **TP** be a user provided test-purpose. Abstract test case generation involves exploring various paths in the HSMSC H modeling the system requirements up to a given depth bound. A path in H is reported as a test-case if it contains all the test-purpose events (except for the *forbidden* events) according to the partial order specified by the test-purpose SMSC S_{TP}, possibly interspersed with other events appearing in H.

For generating abstract test cases, we exploit the abstract execution semantics of

Algorithm 2: *testGen(n,C,t,d,p)*: abstract test generation

Input: n – current node of HSMSC H
Input: $C \in \mathcal{C}_{\mathbf{Spec}}$ – current abstract system-state
Input: t – current state of test-purpose LTS $L_{\mathbf{TP}}$
Input: d – current exploration depth of HSMSC H
Input: p – current path being generated in HSMSC H
Output: Set of abstract test-cases

```
1  if d ≤ D then                                   /* D: user given depth-bound */
2      (t', C') ← execute(t, C, n);
3      if t' = accept then
4          addTest(p.n) ;  /* Add abstract test-case corresponding to the
                              current path in H, obtained by concatenating node n to path
                              p generated so far */
5          return;
6      else if t' = invalid then
7          genWarning();
8          return;
9      else
10         forall sn ∈ succ(n) do /* succ(n) returns all successor nodes of
                                     n in HSMSC H */
11             testGen(sn, C', t', d + 1, p.n);
```

SMSCs [101]. This allows us to generate abstract test cases for system configurations with process classes having an unbounded (i.e. ω) number of objects. At the core of our test generation process is the comparison of test-purpose events with the events appearing in the system model. Let Act_{TP}, Act_{Spec} denote the set of events appearing in **TP** and **Spec** respectively. We define a relation '$\approx \subseteq Act_{\mathrm{TP}} \times Act_{\mathrm{Spec}}$' such that $(a, a') \in \approx$ iff the two events match exactly, or differ only in guards specified as '*' (don't care) for the test-purpose event a. Recall that, $\mathcal{C}_{\mathrm{Spec}}$ denotes the set of all abstract system-states of **Spec** (see Section 8.1).

The abstract test generation proceeds by exploring paths of increasing lengths

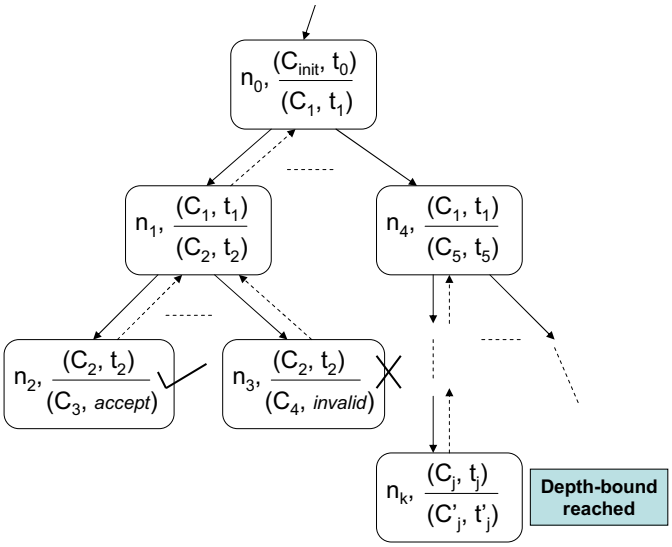


Figure 8-6: Abstract test case generation.

in the HSMSC H, from length 1 up to a user given depth bound D. A path $l = n_0.n_1 \ldots n_k$ in H corresponds to a sequence of HSMSC nodes (n_i's), where n_0 is the initial node. The overall abstract test generation procedure is described in Algorithm 2. It takes as input five parameters– (i) a node n in the HSMSC H, (ii) an abstract system-state $C \in \mathcal{C}_{\text{Spec}}$, (iii) a node t in labeled transition system (LTS) L_{TP} describing the event linearizations of test-purpose SMSC S_{TP} (see Section 8.1.2), (iv) current exploration depth d of HSMSC H, and (v) current path p (of length $d-1$) being generated in H. The procedure is initially invoked using $testGen(n_0, C_{init}, t_0, 1, \epsilon)$, where n_0 is the initial node of H, $C_{init} \in \mathcal{C}_{\text{Spec}}$ is the initial abstract system-state, t_0 is the initial state of LTS $L_{\mathbf{TP}}$ and ϵ represents the empty path.

At each step during exploration, when a node n_i is visited, the algorithm considers the SMSC S_i associated with node n_i, the current abstract system-state $C \in \mathcal{C}_{\text{Spec}}$, and current state t in L_{TP}. SMSC S_i is then executed at state C following the SMSC

operational semantics (using transition relation $\rightarrow_{\text{Spec}}$ as described in Section 8.1), resulting in abstract system-state $C' \in \mathcal{C}_{\text{Spec}}$. While executing SMSC S_i, the test generation algorithm tries to find events in S_i, which are *equivalent* (as per the relation $\approx$) to the test-purpose events emanating from the current state t in the test-purpose LTS L_{TP}; the execution of these events advances the test-purpose LTS to a new state t' (abstracted as line 2 in Algorithm 2). It is possible that none of the events in S_i is equivalent to an outgoing event from t in LTS L_{TP}, in which case $t' = t$. This is essentially what is captured in the node annotations of the form '$n_i, \frac{(C,t)}{(C',t')}$' in Figure 8-6, which represents a sample exploration for test generation. One of the following cases may arise while processing a HSMSC node n_i visited during search.

Case 1: (Alg. 2, lines 3–5) All the test-purpose events are matched, i.e. an *accept* state is reached in L_{TP}. In this case, the SMSC obtained by the asynchronous concatenation [8] of SMSCs corresponding to HSMSC nodes along the current path being explored, is reported as a test case. Moreover, the current path is not further explored. For example, in Figure 8-6, a test case is found at a depth of 3 at node labeled with n_2, and corresponds to a SMSC $S = S_0.S_1.S_2$.

Case 2: (Alg. 2, lines 6–8) Execution of an event in S_i matches a forbidden event in L_{TP} leading to an *invalid* state in L_{TP}. At this point a warning message is generated (Alg. 2, line 7), so that user may inspect the system model for the possibility of any error. The search then backtracks and continues along an alternate path. For example, at node labeled with n_3 in Figure 8-6, a forbidden event is matched causing

the search process to backtrack.

Case 3: (Alg. 2, lines 9–11) Otherwise, the current path is continued to be explored in a similar manner up until the depth bound of D is reached, at which point it backtracks (see node labeled with n_k in Figure 8-6).

Thus, the abstract test case SMSCs derived contain all the test-purpose events (except for the *forbidden* events) according to the partial order specified by test-purpose SMSC S_{TP}, possibly interspersed with other events appearing in the SMSC model.

The running time for Algorithm 2 is $O(|E_H|^D.N_H.|E_T|.D)$, where E_H is the maximum number of outgoing edges from a node in the HSMSC H, E_T is the maximum number of outgoing edges from a node in the test-purpose LTS L_{TP}, and N_H denotes the maximum number of events in a SMSC corresponding to a node in HSMSC H. The term $|E_H|^D$ determines the maximum number of paths that can be explored in HSMSC H for a depth bound of D. While for each path explored, $N_H.|E_T|.D$ determines the maximum possible comparisons between the system-model and test-purpose events.

To formally describe the test generation process discussed above, we define a *satisfaction* relation between a test-purpose **TP** and a system model **Spec** in the following. Recall that, Σ_{Spec} represents the set of process expressions describing **Spec**, $\mathcal{C}_{\text{Spec}}$ denotes the set of system-states of **Spec** and $\rightarrow_{\text{Spec}}$ captures the abstract execution semantics of **Spec** (Section 8.1).

Definition 20. *Satisfaction relation. Let* ***Spec*** *be a given system model and* $L_{TP} = (T, \rightarrow_{TP}, t_1, I, A)$ *be the test-purpose LTS corresponding to a test-purpose* ***TP****. Then, relation* $\simeq \subseteq T \times (\Sigma_{Spec} \times \mathcal{C}_{Spec})$ *is a* **satisfaction** *relation s.t.* $\forall t \in T \backslash I, s_1 \in \Sigma_{Spec}$, $c_1 \in \mathcal{C}_{Spec}$*:* $t \simeq (s_1, c_1)$ *iff*

1- $t \in A \vee$

$(\exists\, a \in Act_{TP},\ t' \in T \backslash I,$

$w = b_1 \ldots b_n \in Act^{+}_{Spec},\ s_i \in \Sigma_{Spec},\ c_i \in \mathcal{C}_{Spec},\ 1 < i \leq n+1 \bullet$

2- $(c_1 : s_1 \xrightarrow{b_1}_{Spec} c_2 : s_2 \ldots \xrightarrow{b_n}_{Spec} c_{n+1} : s_{n+1}$

3- $\wedge\ \ \forall a' \in Act_{TP}, t'' \in T, 1 \leq i < n \cdot\ (t \xrightarrow{a'}_{TP} t'' \implies \neg(a' \approx b_i))$

4- $\wedge\ \ t \xrightarrow{a}_{TP} t'$

5- $\wedge\ \ a \approx b_n$

6- $\wedge\ \ t' \simeq (s_{n+1}, c_{n+1})$

$)$

$)$.

Intuitively, the *satisfaction* relation holds between a test-purpose LTS L_{TP}'s state $t \in T$ and a process expression $s_1 \in \Sigma_{\mathrm{Spec}}$ describing the system model **Spec** at a given system-state $c_1 \in \mathcal{C}_{\mathrm{Spec}}$ (i.e. $t \simeq (s_1, c_1)$) in the following cases–

a) $t \in A$ (Defn. 20, line 1), *i.e.* t is an accepting state in the test-purpose LTS L_{TP}. This indicates that all (non forbidden) test-purpose events have been matched, and hence the test-purpose is satisfied.

b) There exists an event execution sequence $b_1 \ldots b_n$ from s_1 at system-state c_1, resulting in process expression s_{n+1} and system-state c_{n+1} with $s_2, \ldots, s_n$ $(c_2, \ldots, c_n)$ as the intermediate process expressions (system-states)– see (Defn. 20, line 2). In this case, none of the following events $-b_1, \ldots, b_{n-1}-$ are equivalent (as per relation $\approx$) to any test-purpose event labeling an outgoing transition from state t in the test-purpose LTS L_{TP} (Defn. 20, line 3). Only event b_n is equivalent to a test-purpose event a, labeling an outgoing transition from t, with t' as the destination state in L_{TP} (Defn. 20, lines 4 & 5). Further, the satisfaction relation $t' \simeq (s_{n+1}, c_{n+1})$ holds recursively (Defn. 20, line 6).

8.3.2 Template generation

Once abstract test cases are obtained in the form of SMSCs as described above, we derive a set of *templates* (see Def. 1, p. 202) corresponding to each abstract test case. The template generation takes place in two phases– (i) execution of the given abstract test case SMSC S using our abstract SMSC execution semantics (see Sec. 8.1), followed by (ii) derivation of templates from various system states reachable after execution of S (from a given initial state) in step (i). We now discuss these two phases in more detail.

Abstract test case execution

In the *first* phase of template generation, the given abstract test case SMSC S is executed using our abstract execution semantics for SMSCs (see Section 8.1). However, for template generation, an *extended* version of the abstract system-states is used during the execution. For convenience, we refer to an extended abstract system-state simply as **extended-state**. Compared to the abstract system-state, in addition to maintaining the local state information, an extended-state also maintains the list of events executed by various objects. During template generation, this additional information enables distinguishing among the objects based on the events executed by them.

We now describe the concept of extended-states with the help of an example. Consider the Client class consisting of variables cl_status and v from the CTAS case study. An abstract system-state for the Client class consists of tuples of the form $\langle val, n\rangle$, where $val \in Val(V_{Client})$ and $n \in \mathbb{N} \cup \{\omega\}$ (ref. Section 8.1). Let E_{Client} denote the set of all Client events occurring in the CTAS system model. Then, an extended-state for the Client class will consist of triples of the form $\langle val, h, n\rangle$, where $val \in Val(V_{Client})$, $h \in E^*_{Client}$ and $n \in \mathbb{N} \cup \{\omega\}$. For a given triple $\langle val, h, n\rangle$, h is the list of events executed by the objects represented by this triple. Consider the execution of an event e corresponding to the receive of message $ClientPostRvrt$ in SMSC *NotRcvWthr* (see Fig. 8-1(c)), at an extended-state for Client class given by– $\{\langle(cl\_status = 5, v = 0), h_1, 3\rangle, \langle(cl\_status = 5, v = 1), h_2, 1\rangle\}$, where h_1, h_2

are some execution histories. The resulting extended-state after execution of e is $\{\langle(cl\_status = 7, v = 0), h'_1, 3\rangle, \langle(cl\_status = 7, v = 1), h'_2, 1\rangle\}$, where h'_1 and h'_2 are obtained by concatenating event e to event list h_1 and h_2 respectively.

The algorithm for executing a SMSC using extended-states is outlined in Algorithm 3. It takes as input an abstract test case SMSC S and set of initial extended-states $\mathcal{ES}$ given by– $\{\bigcup_{p\in\mathcal{P}}\{\langle v_p^{init}, [], n_p\rangle\}\}$, where v_p^{init} is an initial valuation of variables of process class p, [] represents an empty event list, and $n_p \in \mathbb{N} \cup \{\omega\}$ is the number of objects in class p, where ω represents an unbounded number of p objects in their initial state. At any point during the execution of the abstract test case SMSC S, the procedure maintains the set of all reachable extended-states from the initial set of extended-states. At each step during execution while considering an event e from S, the set of all reachable extended-states is derived from the current set of extended-states (Alg. 3, lines 3–31). This is done by considering each extended-state E in the current extended-state set $\mathcal{ES}$ one by one, and various triples within it that can execute e. If e is an *existential* event (Alg. 3, lines 5–18), for each triple in E which satisfies the guard of e, an object from it is chosen to execute e. This results in a new extended-state, with local state (val) and event history (h) of the executing object updated with the effect of execution of e (Alg. 3, lines 7–18). In this case, multiple extended-states may be generated from a single extended-state. This is because multiple object-states in an extended-state may satisfy the guard of e. If e is a *universal* event (Alg. 3, lines 19–30), then a single extended-state is generated

Algorithm 3: SMSC execution using extended-states

Input: S – abstract test case SMSC
Input: $\mathcal{ES}$ – initial set of extended-states
Output: Updated set of extended-states

$\mathcal{TES} \leftarrow \emptyset$; /* *Temporary set of extended states* */
while *not all* S *events have executed* **do**
  $e \leftarrow$ next S event to be executed ; /* e: an enabled-event chosen according to partial ordering of S */
  forall $E \in \mathcal{ES}$ **do** /* E: *extended-state* */
    if e *is an existential event* **then**
      forall $\langle val, h, n\rangle \in E$ **do**
        if *val satisfies the guard of event* e **then**
          $val' \leftarrow$ effect of execution of e on val;
          if $\langle val', h.[e], k\rangle \in E$ **then** /* $k \in \mathbb{N}$ */
            $E' \leftarrow E \backslash \{\langle val', h.[e], k\rangle\} \cup \{\langle val', h.[e], k+1\rangle\}$;
          else
            $E' \leftarrow E \cup \{\langle val', h.[e], 1\rangle\}$;
          $n' \leftarrow n - 1$; /* $\omega - 1 = \omega$ */
          if $n' == 0$ **then**
            $E'' \leftarrow E' \backslash \{\langle val, h, n\rangle\}$
          else /* $n' > 0$ otherwise */
            $E'' \leftarrow E' \backslash \{\langle val, h, n\rangle\} \cup \{\langle val, h, n'\rangle\}$;
          $\mathcal{TES} \leftarrow \mathcal{TES} \cup \{E''\}$;
    else if e *is a universal event* **then**
      bool check $\leftarrow$ false;
      $E' \leftarrow \emptyset$; /* *Temporary extended-state* */
      forall $\langle val, h, n\rangle \in E$ **do**
        if v *satisfies the guard of event* e **then**
          $v' \leftarrow$ effect of execution of e on val;
          $E' \leftarrow E' \cup \{\langle val', h.[e], n\rangle\}$;
          $E \leftarrow E \backslash \{\langle val, h, n\rangle\}$;
          check $\leftarrow$ true;
      if *check* == *true* **then**
        $E' \leftarrow E' \cup E$;
        $\mathcal{TES} \leftarrow \mathcal{TES} \cup \{E'\}$;
  $\mathcal{ES} \leftarrow \mathcal{TES}$; $\mathcal{TES} \leftarrow \emptyset$;
return $\mathcal{ES}$;

from a source extended-state containing triple(s) satisfying e's guard; the local states and event histories of all objects that can execute e are updated with the effect of execution of e in the resulting extended-state.

We now illustrate the above process using a small example. Consider a sample abstract test case SMSC shown in Figure 8-7. It contains a lifeline representing process class A with an unspecified number of objects, and a lifeline representing process class B with a single object. Assume, that A contains a local variable x initialized to 0, while B has no local variables. We now consider execution of the SMSC in Figure 8-7, from the initial set of extended-states given by $\{\{\langle x = 0, [], \omega\rangle_A, \langle \epsilon, [], 1\rangle_B\}\}$. For convenience, we have added process-class name as subscript to each object-state triple within an extended-state. The first element of B's object state is empty (or ϵ) since there are no local variables in B. For executing event e_1 (see Fig. 8-7), since its guard is *true*, any A object can be chosen to execute it. An object from A-state $\langle x = 0, [], \omega\rangle$, present in the only available initial extended-state, is chosen to execute e_1. The resulting extended-state with the state and event-set of the executing object updated is $\{\langle x = 1, [e_1], 1\rangle_A, \langle x = 0, [], \omega\rangle_A, \langle \epsilon, [], 1\rangle_B\}$. Next, e_2 is executed by the object in B-state $\langle \epsilon, [], 1\rangle$. The resulting set of extended-states is shown as the first entry in Table 8.1. In a similar manner various events in Figure 8-7 are executed one by one. Some other extended-state sets reached during execution are shown in Table 8.1. Note that, the final set of extended-states (third entry, Table 8.1) contains two extended-states. These arise due to two different choices of object-states of A for

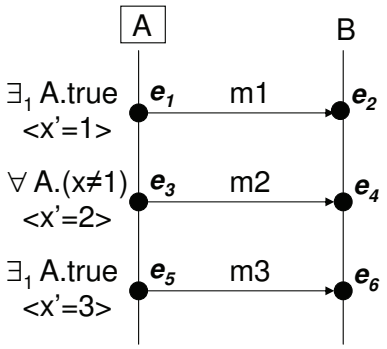


Figure 8-7: Example test case SMSC.

Table 8.1: Extended-states reachable during template generation.

	Sets of reachable Extended-states
1. After e_1, e_2	$\{\{\langle x = 1, [e_1], 1\rangle_A, \langle x = 0, [], \omega\rangle_A, \langle \epsilon, [e_2], 1\rangle_B\}\}$
2. After e_1–e_4	$\{\{\langle x = 1, [e_1], 1\rangle_A, \langle x = 2, [e_3], \omega\rangle_A, \langle \epsilon, [e_2, e_4], 1\rangle_B\}\}$
3. Finally	$\{\{\langle x = 3, [e_1, e_5], 1\rangle_A, \langle x = 2, [e_3], \omega\rangle_A, \langle \epsilon, [e_2, e_4, e_6], 1\rangle_B\},$ $\{\langle x = 1, [e_1], 1\rangle_A, \langle x = 3, [e_3, e_5], 1\rangle_A, \langle x = 2, [e_3], \omega\rangle_A,$ $\langle \epsilon, [e_2, e_4, e_6], 1\rangle_B\}\}$

executing event e_5, at the extended-state reached after executing events e_1 to e_4.

Constructing templates

In the *second* phase, corresponding to each final extended-state E obtained after executing abstract test case SMSC as described above, we construct a template (see Def. 1, page 202) T_E as follows. For each triple $\langle \_val, h, k\rangle$ in E we do the following. If k is finite, we add k lifelines (along with the events appearing in the execution history h) in the template T_E. If however, $k = \omega$ (*i.e.*, $\langle \_val, h, k\rangle$ represents unbounded number of objects), we create a single lifeline representing one or more objects for this triple provided the execution history is non-empty (*i.e.*, $h \neq []$). This corresponds to our interpretation of ω as $n \geq 1$ objects, where the number n is unbounded. Note

that, the event execution list of a triple with an unbounded number of objects will contain only universal events. This is because, in an abstract test case an existential event can occur only a finite number of times, with each occurrence executed by a single object. We call a lifeline representing one or more objects as a **marked lifeline** (see Def. 1, p. 202) and annotate it with a + sign in the template. During concrete test generation, these marked lifelines are blown up into several lifelines depending on the supply of objects in the concrete system. *There is at most one marked lifeline for each process class.*

Once the lifelines are created in the template along with the events they participate in, completing the template SMSC is trivial. We simply connect the send events with their corresponding receive events as per the partial order prescribed by the abstract test case SMSC.

Theorem 2. *A template derived corresponding to an abstract test case S using the approach described in Section 8.3.2 satisfies the properties specified in Definition 1 (page 202. That is, a template T captures a projection of events from a symbolic lifeline l_p representing process class p in S, to one or more lifelines from process class p in T (represented as C_p) such that,*

1. *The projected events (from l_p to lifelines in C_p) follow the top-down event ordering along l_p.*

2. *An existential event from l_p appears along exactly one lifeline in C_p, while a universal event from l_p appears along all lifelines in C_p such that– the event se-*

quence (from top to bottom) along a lifeline l in C_p captures a feasible execution path in the control-flow of process class p (i.e. events along l can be executed by a single p-object).

Proof. 1. Template T is constructed from a final extended-state E reached after execution of abstract test case S as described in Sections 8.3.2 and 8.3.2. Each lifeline l in T corresponds to a triple $tr = \langle val, h, n \rangle$ in E such that $h \neq []$, with the sequence of events in execution list h appearing along the lifeline l. Assuming that triple tr represents objects of process class p– (i) events in h will be the events appearing along symbolic lifeline l_p from class p in S, and (ii) the event sequence (in h) would follow the top-down ordering along l_p as per our SMSC operational semantics [101].

2. From lines 4–18 of Algorithm 3 we observe that exactly one object is chosen to execute am existential event from a triple satisfying the event guard (lines 6–7, Alg. 3) in a given extended state E (line 4, Alg. 3). Therefore, each occurrence of an existential event can appear in execution history of exactly one object. Note that, a triple $tr = \langle \_val, h, k \rangle$ containing existential events in its event execution list h represents $k \in \mathbb{N}$ objects executing k distinct occurrences of an existential event (in h) appearing along a symbolic lifeline in S. Further, tr will be blown up into k concrete lifelines in T. Hence, each occurrence of an existential event in S will appear along exactly one lifeline in T. The different object choices (modulo object identity) for executing an existential event are

mutually exclusive and lead to different possible final extended states (and thus resulting in different templates).

On the other hand, while executing a universal event e, all object choices that can execute e from a given extended-state E are considered together to execute e, resulting in a new unique extendbed state E' (lines 19–30, Alg. 3).The execution histories of all triples whose state satisfies e's guard (line 22, Alg. 3) will be appended with e after its execution. Thus, eventually when template lifelines are created from the final extended states, the event e will appear along *all* lifelines that executed e.

Furhter, at any point during the execution of abstract test case SMSC S for template generation (Section 8.3.2), the execution history of a triple withing an extended-state contains only the sequence of events that can be executed by the objects of that triple from their initial states (thus representing a feasible execution in their control flow). This can be easily shown using induction on the event execution sequence.

□

Lemma 3. *For an abstract test case S, our template generation procedure constructs the set of templates $\mathcal{T}$ as defined in Definition 1 (page 202).*

Proof. As shown in Theorem 2, a tempate generated by our procedure satisfies the propperties specified in Definition 1 (page 202). We now only need to show that our template generation procedure constructs the complete set $\mathcal{T}$. From the definition

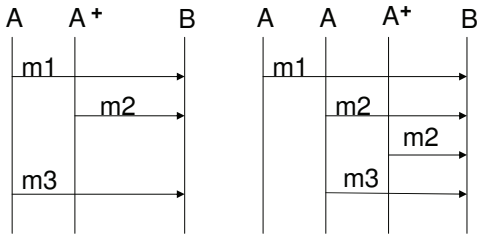


Figure 8-8: Templates for abstract test case shown in Fig. 8-7.

of templates and Theorem 1 we can easily see that different templates arise due to different possible projections of existential events from a symbolic lifeline in S to corresponding lifelines (of the same process class) in a template in $\mathcal{T}$. Our template generation procedure constructs a template from a final extended-state (see Sec. 8.3.2) reahced after executing S (see Sec. 8.3.2). Now, while executing S, for each existential event e we consider all reachable extended states at that point and each object within an extended state (modulo the object identity) that can execute e (lines 4–18, Alg. 3). These choices capture all possible projections of e in the final resulting templates. □

For illustration, recall that the final extended-states of the abstract test case SMSC in Figure 8-7 were captured in the third entry of Table 8.1. Now, one template test case each is generated for the two final extended-states in Table 8.1. The two template test cases appear in Figure 8-8.

8.3.3 Concrete test case generation

Once templates are obtained from an abstract test case, concrete test cases in the form of MSCs are then derived from the templates. Given (i) a template derived from

an abstract test case T, corresponding to a system model **Spec** and test-purpose **TP**, and (ii) an object configuration $\bigcup_{p \in \mathcal{P}} O_p$ (see Def. 2, p. 206) corresponding to **Spec**, we generate concrete test case MSC(s) from the given template as follows. Given the set of objects O_p for each process class p, we simply assign concrete objects to the lifelines of the template which correspond to class p. For each lifeline l involving p we assign one concrete p-object, provided l is not a marked lifeline. Recall that there can be at most one marked lifeline for a process class p. If class p has a marked lifeline we assign all *remaining* p-objects to it once the unmarked lifelines involving p are assigned objects. 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. Next, we define the set of *minimal concrete tests*, which are derived from the set of templates (see Def. 1, p. 202).

Definition 3 (Minimal concrete tests)**.** *Given an abstract test case SMSC S derived from system model* ***Spec****, and an object configuration OC (see Def. 2, p. 206) 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.

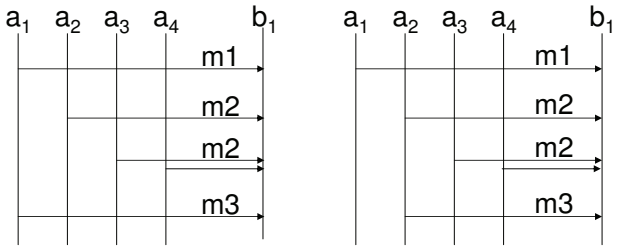


Figure 8-9: Minimal concrete test cases.

For the abstract test case shown in Figure 8-7, two template tests were generated (Figure 8-8). Then, for a concrete configuration with four A objects, two concrete test cases are generated (shown in Fig. 8-9) for these two templates. As can easily be inferred from the guards of A-events e_1, e_3 and e_5 (see Figure 8-7), any object can be chosen to execute events e_1 and e_5, while e_3 is to be executed by all objects that have *not* executed e_1. Thus, with four A objects, a total of $4 \times 1 \times 4 = 16$ concrete test cases can be generated, out of which we only test the two test cases of Figure 8-9.

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

Proof. Let $\mathcal{T}$ be the set of templates derived from S. From the definition of templates (see Def. 1, p. 202) we know that each template captures a projection of events from

a symbolic lifeline in S to one or more lifelines in a template in $\mathcal{T}$. For a template $t \in \mathcal{T}$ and a process class p, let C_p^t denote the set of lifelines from class p in t. Then, following holds–

$$\forall t_1, t_2 \in \mathcal{T}, \exists p \in \mathcal{P} \text{ such that :}$$

$$(i)\, \exists l \in C_p^{t_1} \cdot \forall l' \in C_p^{t_2}, l \neq l', \text{ and}$$

$$(ii)\, \exists l \in C_p^{t_2} \cdot \forall l' \in C_p^{t_1}, l \neq l'.$$

Thus, clearly a concrete test derived from t_1 cannot be derived from t_2, since here will be at least one lifeline in a concrete test case MSC derived from t_1 (t_2) which will be behaviorally distinct from all lifelines in t_2 (t_1). Hence, the minimal concrete tests (see Def. 3, p. 223) derived from S are all behaviorally distinct. Further, we know that for a given object configuration (see Def. 2, p. 206) 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. □

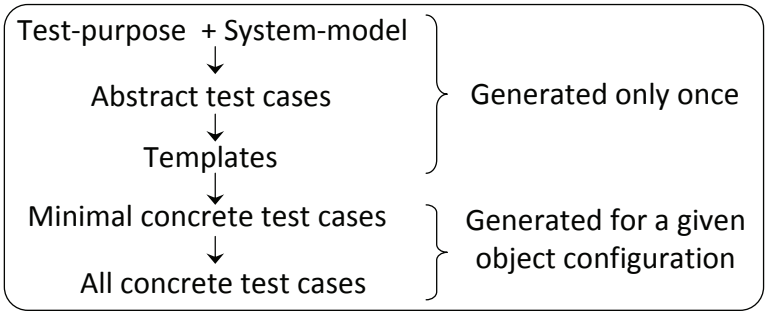


Figure 8-10: Summary of our test generation flow.

8.3.4 Summary

To summarize, we recapture our overall test generation flow in Figure 8-10. 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. 2, p. 206). 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 8.5.2).

8.4 Test-execution Setup

Having derived the concrete test case MSCs as described in the last section, we briefly discuss the experimental setup for testing a system implementation using these test cases. As mentioned earlier, the implementation under test (or, IUT) can either be constructed manually, or generated (semi-) automatically from another system model.

Initially, 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

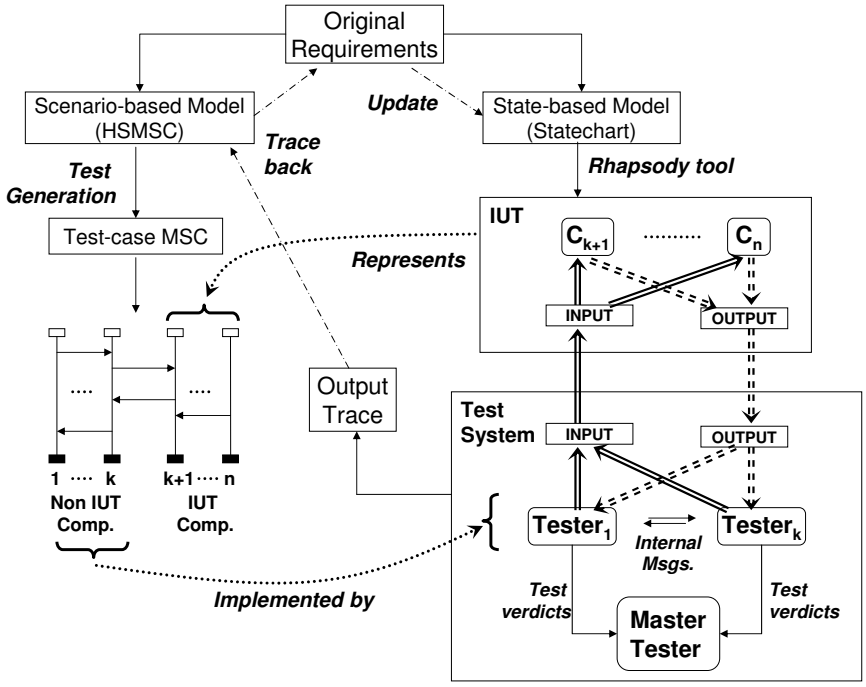


Figure 8-11: Design flow with detailed Test execution architecture.

the purpose of giving the test verdicts.

The overall test-architecture consisting of IUT components, tester-components and the master-tester, is shown in Figure 8-11. The IUT components labeled as $C_{k+1} \ldots C_n$, correspond to the IUT lifelines in a test-case MSC (numbered $k+1 \ldots n$ in the test-case MSC in Figure 8-11). The *double* solid (dashed) arrows represent the flow of input (output) messages with respect to IUT, among various components in the test setup. The IUT input events are also referred to as *controllable* events, as sending inputs is under the control of the tester-components. Similarly, the output events from IUT are called *observable* events, since they can only be observed by the tester components. This setup corresponds to a distributed testing architecture [116], where multiple testers are used to stimulate and observe the IUT components. We

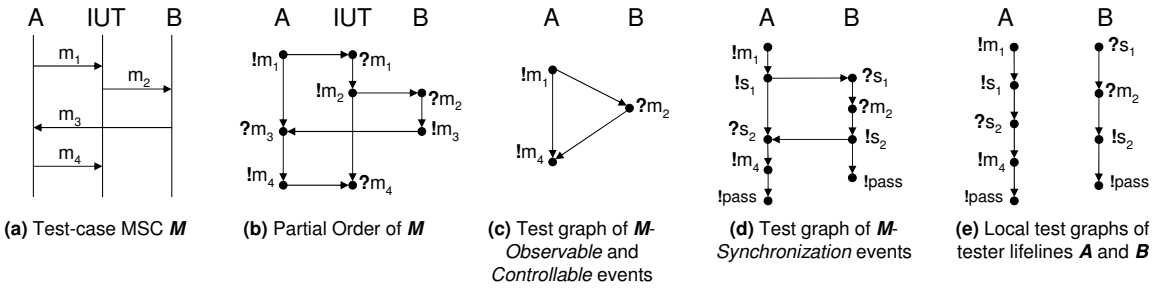


Figure 8-12: Generation of tester-components from a test-case MSC.

now discuss the derivation of tester components corresponding to tester lifelines.

Tester Component To obtain the tester components from a test case MSC, we follow a distributed tester synthesis approach similar to [63]. The test case MSC is viewed as a partial order $\langle E, \leq \rangle$ over various events E appearing in it. The partial order $\leq \equiv (\leq_l \cup \leq_m)^*$ is the transitive closure of $\leq_l$ and $\leq_m$, where $\leq_l$ is the linear ordering of events from top to bottom along all lifelines, and $\leq_m$ represents the ordering between a message send e_s and its corresponding receive e_r, s.t. $e_s \leq_m e_r$. A sample test-case MSC and its corresponding partial order are shown in Figures 8-12(a) and 8-12(b). Note that a send (receive) event corresponding to a message m is shown as $!m$ ($?m$) in Figure 8-12.

For generating the tester components, a reduced partial order $\langle E_T, \leq_T \rangle$, called *test graph*, is obtained from the test-case MSC's partial order. It contains only controllable and observable events E_T ($\subseteq E$) with respect to the IUT components in test case MSC, and a partial ordering $\leq_T$ over them such that, $\forall e, e' \in E_T, e \leq_T e' \text{iff } e \leq e'$. For the test-case example shown in Figure 8-12(a), where lifelines A and B represent the tester lifelines, its test graph is shown in Figure 8-12(c).

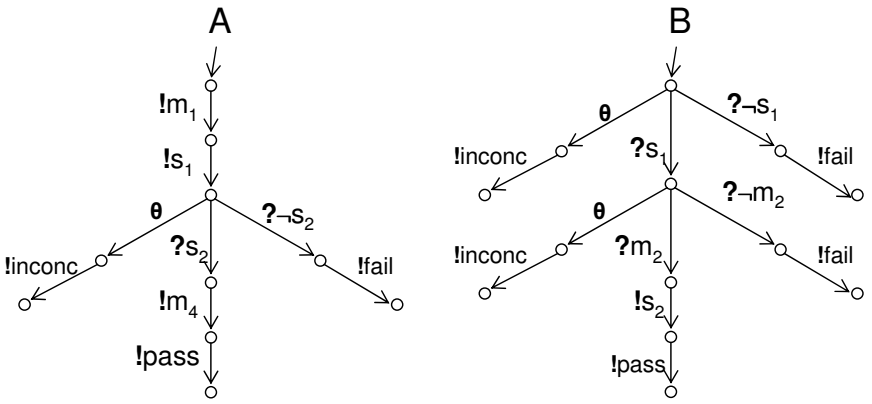


Figure 8-13: Generated tester-components for test-case MSC shown in Fig. 8-12(a).

In the next step, *synchronization* messages are introduced in the test graph to preserve the causality constraints between the events appearing along the distinct tester lifelines. For a direct ordering between two events appearing along different lifelines, a synchronization send is introduced after the first event along its lifeline, while the corresponding receive is added before the second event along its lifeline. Further, after the last event along each tester lifeline in the test graph, sending of a *pass* message is also added. These messages are received by the master-tester (not shown here) based on which it gives test verdicts. The test graph for the above example, with synchronization messages (s_1 and s_2), and *pass* verdicts appears in Figure 8-12(d).

From the resulting test graph detailed in the preceding, a *local* test graph for each tester lifeline is derived by taking a projection over the events appearing along that lifeline (shown in Figure 8-12(e)). A *tester component* is then derived as a sequential automaton from each local test graph by considering all possible event linearizations.

Further, in these automatons, from each node with an outgoing transition labeled with a receive event $?e_r$, two outgoing edges labeled with θ and $?\neg e_r$ respectively, are added. Here θ represents a timeout event, which occurs if no input is received within a given timeout value. On the other hand, $?\neg e_r$ represents the receipt of a test case event other than e_r. Both these events (θ and $?\neg e_r$) result in sending of *fail* verdict to the master-tester. The automata for tester lifelines A, B are shown in Fig. 8-13.

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* [111]. 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.

8.5 Experiments

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.

8.5.1 Test generation

Since we use SMSCs for system modeling as well as for test-purpose specification, our test generation engine is built on top of SMSC operational semantics [101]. The SMSC operational semantics was encoded earlier as Prolog rules in the XSB logic programming system [4]. We build our test generation framework (also in Prolog) on top of these Prolog rules encoding SMSC operational semantics. All experiments were conducted on a Pentium-IV 3GHz machine with 1GB of main memory.

We consider the overall test generation flow as summarized in Figure 8-10. For the CTAS example, we derived five test-purposes with the aim of covering its major use-cases. 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. TP-3 represents a scenario where some Clients are unable to receive the weather update. Subsequently, all Clients revert back to using the old weather information. In TP-4, all Clients

are successfully updated with the new weather information. Finally, TP-5 (shown in Fig. 8-3(a)) 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.

Abstract test generation For all five test-purposes, abstract test cases were generated using the procedure outlined in Section 8.3.1. Abstract test-generation results are shown in Table 8.2 (columns 2–4). For each test-purpose, paths in the CTAS HSMSC were explored up to a depth of 20. Execution times are reported for generating either a *single* (Tab. 8.2, col. 2), or *all* (Tab. 8.2, col. 3–4) abstract test cases corresponding to a test-purpose.

Table 8.2: Symbolic Test Generation for CTAS example with exploration depth set to 20. For generating the concrete tests, we consider 3 Clients.

TP #	1 Abst. test case	All Abst. test cases		# Templates	All Concrete Test Cases
	Time(s)	Time(s)	Total #		
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

Template and concrete test generation As discussed in Section 8.3.2, 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 8.3.3). In Table 8.2 (col. 5), we report the total number of templates generated for all abstract test cases for each test-purpose. Concrete test cases were generated

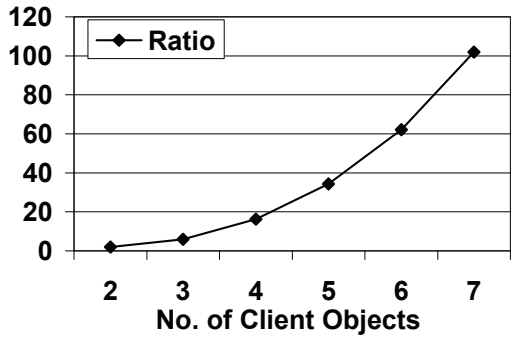


Figure 8-14: Ratio of All/Minimal no. of concrete test cases for Test-purpose 4.

from these templates for an object configuration consisting of– *three* Client objects, *one* CM object and *one* WCP object. The number of minimal concrete test cases obtained for this configuration is same as the number of templates (Tab. 8.2, col. 5), since all the templates can be instantiated to a concrete test for the given configuration with three clients. We also show the total number of all possible concrete test cases for each test-purpose (Tab. 8.2, col. 6).

By comparing columns 5 and 6 in Table 8.2 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 Sec. 8.3.3). On the other hand, the number of all possible tests keeps on increasing with the increasing number of objects. For illustration, ratio of the *all* to the *minimal* number of concrete test cases for test-purpose 4 (TP-4), with increasing number of clients, is plotted in Figure 8-14.

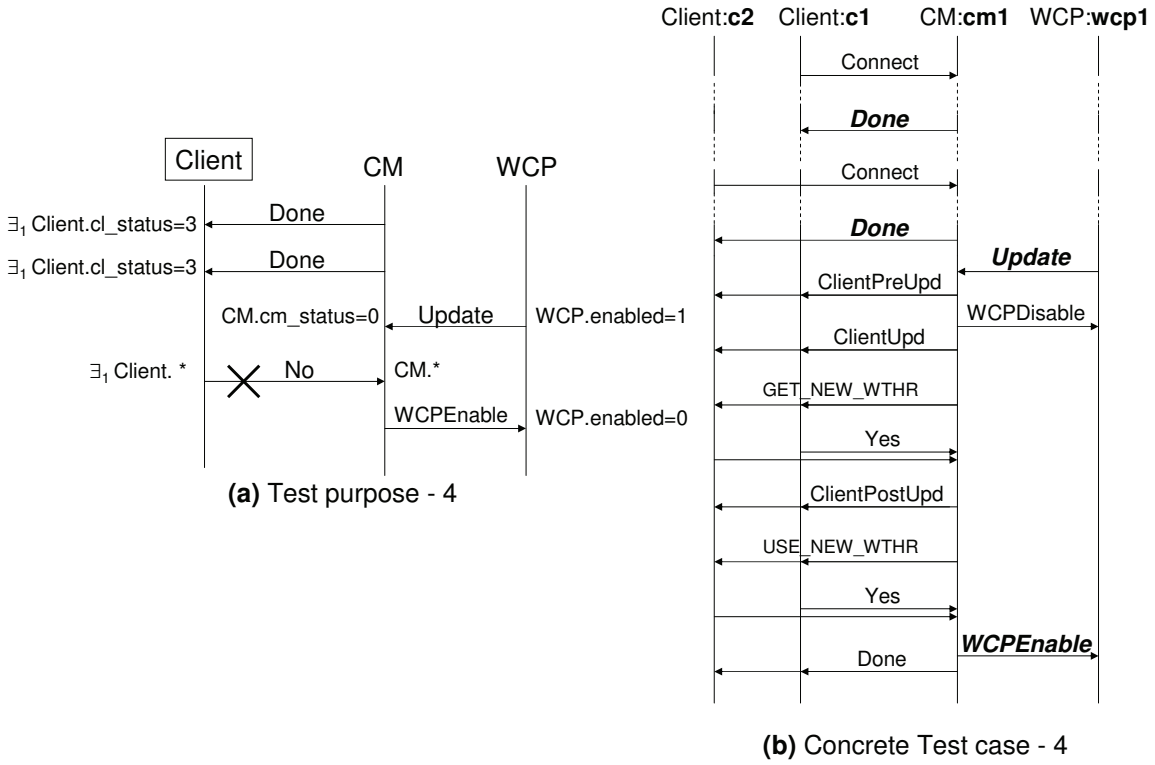


Figure 8-15: Test-purpose 4 and a corresponding Concrete test case.

Example. Consider test-purpose 4 (TP-4), shown in Figure 8-15(a). The first two *Done* messages in TP-4 reflect the successful connection of two clients to controller CM (more clients can also be connected, but we need at least two clients for this test-purpose). The subsequent *Update* message corresponds to the weather update request initiated by WCP. Then we have a forbidden message *No* with don't care guards ($*$) for both the Client and CM lifelines. Thus, it would match, and hence avoid generating test cases with any *No* message sent by a client to CM during the weather update.

A concrete test case MSC corresponding to TP-4 is partially shown in Figure 8-15(b). The message names appearing in bold italics represent the matching events in the test-

Table 8.3: 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

purpose (Fig. 8-15(a)). Initially, two clients c_1 and c_2 get connected to CM (cm_1) by sending the *Connect* message, eventually receiving the *Done* message. Various intermediate messages exchanged during these connection setups are not shown; these are represented via dotted line segments along lifelines in Figure 8-15(b). Subsequently, the two connected clients are successfully updated with the latest weather information via CM.

8.5.2 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 8.3, 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. 8-10). In Table 8.4, we compare "the time taken to generate the minimal set of concrete tests (see Def. 3, page 223) 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).

Table 8.4: 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

Only the results corresponding to test-purposes 1 and 2 are shown in Table 8.4, 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.

8.5.3 Test execution

We also 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 [100] 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.*. We begin by first explaining our test execution setup.

Table 8.5: 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

In our experiments, we focused on testing the central controller (CM) component of CTAS C++ implementation. The implementation code was generated automatically from the Statechart model of the CTAS requirements using the Rhapsody tool [100]. In Table 8.5, we present the key features of the CM's Statechart model. The C++ code for CM was tested against our minimal set of concrete test cases, derived using the five test purposes discussed earlier. This testing process led to the discovery of some significant bugs in the Statechart model of CM, such as missing transitions, states etc.

We now discuss one of the more subtle bugs discovered while testing against one of the test cases (see Figure 8-15(b), this was a test case corresponding to the fourth test-purpose in Fig. 8-15(a)). The execution of this test case resulted in an *fail* verdict. The output log produced by the test driver indicated that the tester components corresponding to– a) Client objects c_1 and c_2 timed-out while waiting for

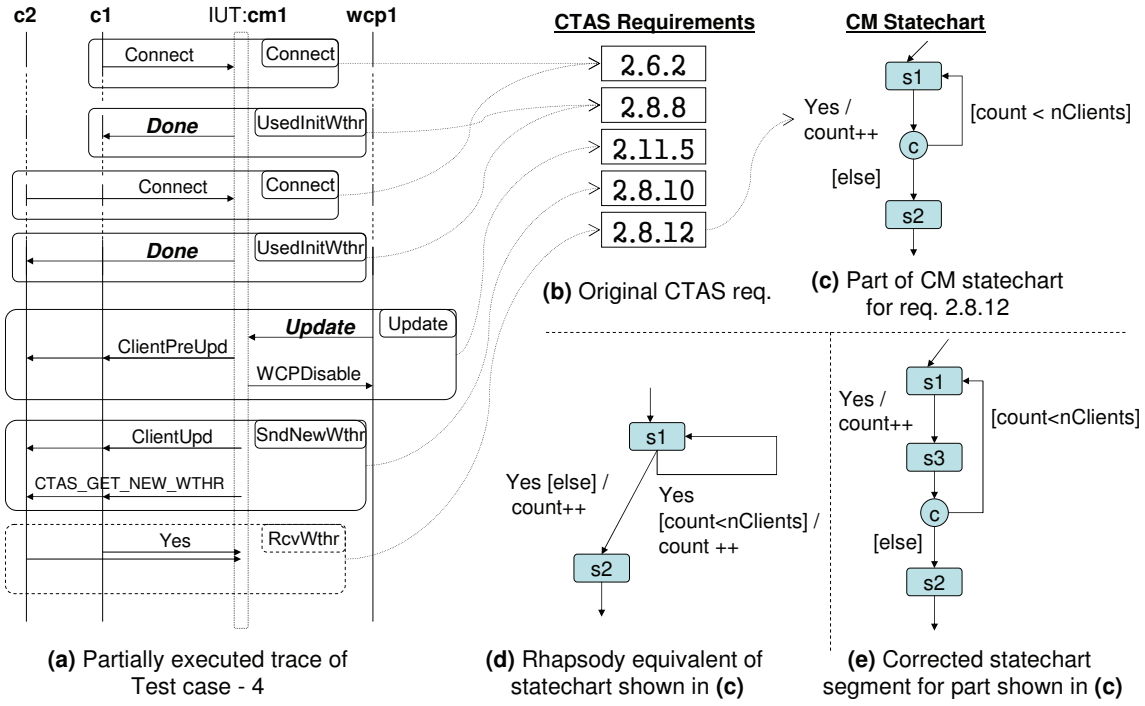


Figure 8-16: Test execution and tracing of test results for Test case 4.

message *ClientPostUpd* from CM (cm_1), and b) WCP object wcp_1 timed-out waiting for the *WCPEnable* message from cm_1. The partially executed trace depicting the tester components is shown in Figure 8-16(a). The boxes shown group together the events that correspond to a particular node in the CTAS HSMSC, with the node name appearing at the top right corner of the box. The last box drawn using dashed line in Figure 8-16(a) represents an incomplete set of events executed from SMSC RcvWthr.

For convenience, we show the mapping of various SMSCs to the original CTAS requirements from which they were derived. From there, the source of error was discovered in the Statechart description of the CM. The part of the Statechart where

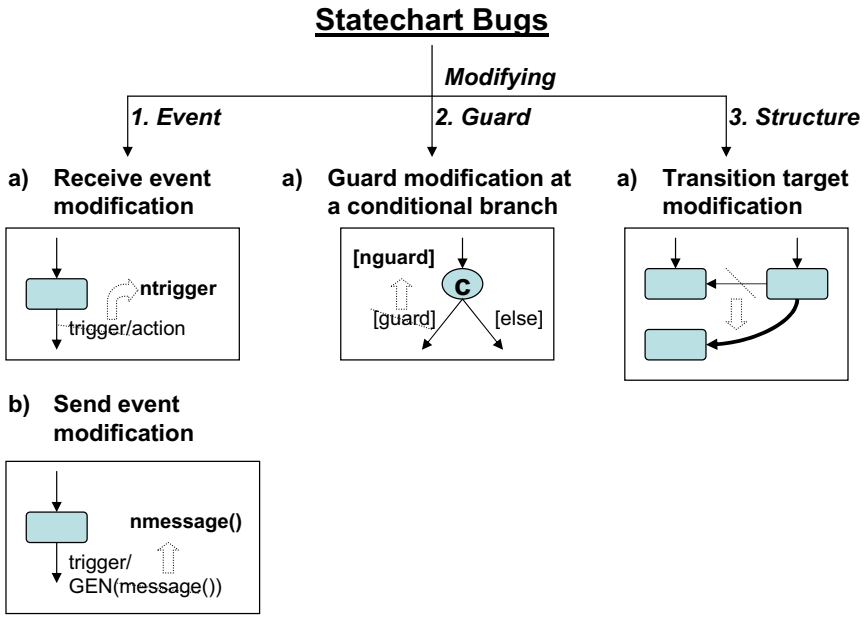


Figure 8-17: Taxonomy of bugs introduced in Statechart models for the CTAS case study.

fault was located is shown in Figure 8-16(c). In state s_1 the CM waits to receive the message *Yes*, and updates a counter *count* to check if all the clients have replied. However, we found that Rhapsody interprets this as another (supposedly equivalent) structure (shown in Figure 8-16(d)). Now, in this case the counter is incremented after checking the guard $count < nClients$, instead of doing so before this check. To ensure that the guards of outgoing transitions from the condition node (marked with c) are evaluated after *count* is incremented, a new state s_3 was introduced between state s_1 and the condition node, which corrected this fault. The updated structure is shown in Figure 8-16(e).

Fault-injection and debugging To further 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 different category of bugs (see Table 8.6) were systematically introduced in the CM's Statechart for this purpose. The taxonomy of these bugs is shown in Figure 8-17. 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, assuming a concrete system with two (2) client objects, 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 8.6.

Table 8.6: 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[2] 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 8.3.3), 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.

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

8.6 Discussion

In this chapter, we have presented a model-based test generation methodology for distributed reactive systems with *many behaviorally similar objects*, based on our notation of 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.

Our approach benefits the designer by grouping together behaviorally similar tests, thereby helping him/her comprehend the implementation under test. More importantly, if the system configuration changes (due to change in number of objects in one or more classes), the concrete tests can be generated from our templates with very minimal effort. Various experimental results illustrate the efficacy of our approach.

Acknowledgments Liang Guo constructed the StateChart model of the CTAS case study.

Part IV

Conclusion

Chapter 9

Conclusions and Future Work

The Model-driven design and development of systems is becoming increasingly popular, and gaining widespread usage due to its various advantages. Various modeling notations, such as UML, which were earlier used mainly for requirements gathering and documentation purposes, have now become the main focus of various model-driven development methodologies. However, it is also becoming clear, that in order to fully utilize various benefits of model-driven development, directly using existing notations may not suffice. This is because, the modeling notations are often too generic, or may simply lack features to support development for a certain category of systems. The use of Domain Specific Languages (or DSLs) is seen as an important step in this direction, with major organizations providing initial tools to support design and development of DSLs, for example, Microsoft's Domain Specific Language

Tools[1] and Eclipse Modeling Framework[2].

In this thesis, we have developed two scenario-based behavioral modeling notations, targeted towards distributed reactive systems consisting of classes of interacting processes. Both our notations– Interacting Process Classes (IPC) and Symbolic Message Sequence Charts (SMSC), are equipped with an abstract execution semantics (unlike most of the existing notations), which allow for efficient and scalable validation of initial system requirements. While IPC supports state-based modeling of systems with inter-process interactions being specified at a higher level of granularity than a single message send/receive, SMSC supports purely scenario-based modeling. We note that, often deciding which notation to use for modeling, depends on the requirements specification itself [42].

Another interesting and challenging aspect that we have attempted to address in this thesis is the maintenance of associations in the abstract execution setting. Though, it is relatively straightforward to handle associations when dealing with concrete objects, in our abstract execution where objects' identities or states are not maintained individually, we over-approximate association information by maintaining association links among groups of objects. Consequently, spurious behaviors may arise during abstract execution, which can be detected by checking if there exists any concrete realization of the given execution trace. Interestingly, we have not seen association based constraints being explicitly specified or used in various popular

[1] `http://www.domainspecificdevelopment.com/`
[2] `http://www.eclipse.org/modeling/emf/`

notations such as Statecharts or Live Sequence Charts (LSCs).

Finally, we observe that majority of model-based test generation works use state-based notations as the underlying system model, while scenario-based notations may be used to specify a test-purpose. In our case, we support automated test case generation from both IPC and SMSC, where SMSC is entirely a scenario-based notation, while in case of IPC, inter-process interactions are modeled as transactions specified using MSCs. Since, distributed reactive system requirements are more naturally captured using a scenario-based notation, the test cases generated from our models can serve as a means to test a system implementation against the original requirements. Further, abstraction at both syntactic and semantic level in case of SMSCs, allows us to generate a minimal set of concrete test cases capturing all distinct relevant behaviors corresponding to a given test-purpose (modulo the maximum test-case length).

9.1 Future Work

9.1.1 Extensions

Timing Constraints. In order to be able to capture and reason about the real-time behavior, we need to include support for modeling and execution of timing constraints in our modeling notations. For instance, integrating timing features in our modeling frameworks to enable us to specify timing constraints such as: message delays and upper/lower bounds on a process to engage in certain actions. Note that, handling

of real time constraints becomes more challenging in the presence of our abstract execution semantics.

Behavioral Subtyping. Also, currently we do not support class inheritance, either structural or behavioral. The structural inheritance primarily aims at reusing existing class definitions and possibly adding new behaviors, or redefining existing ones. However, this does not guarantee any form of behavioral conformance between the subtype and the supertype. On the other hand, the notion of behavioral subtyping plays a crucial role in object oriented systems by allowing an object of a subtype to replace the object of its parent type, without changing the overall system behavior. One of the early works in this area is by Liskov and Wing [76] which focuses on passive objects objects whose state change is only via method invocation by other objects. Subsequently, behavioral subtyping of active objects has been studied in many works (e.g. [13, 50, 118]). These works mostly exploit well-known notions of behavioral inclusion (such as trace containment or simulation relations) to define notions of behavioral subtyping. In future, we aim to incorporate similar notion(s) of behavioral inclusion to allow reuse of existing process classes, and define an efficient mechanism for checking substitutability of a subtype for its supertype. Behavioral subtyping has mainly been studied in the intra-object setting, where the behavior of each class is explicitly specified, for example, using a finite state machine. It would be interesting to study the behavioral notion of a subtype in the scenario-based setting (such as SMSCs), where intra-object behavior is not explicitly described.

Model checking. Another interesting extension to our present work can be the development of a verification frameworks centered on our abstract execution semantics, that will exploit the abstraction-refinement based approach to software model checking. The abstraction refinement framework can be used to find which associations need to be tracked in the abstract execution semantics, in order to avoid spurious run(s). For instance, in case of IPC we may have transaction guards of the form $(r1, r2) \in asc1 \wedge (r2, r3) \in asc2$ where $r1, r2, r3$ are transaction roles. Consequently, it will not be sufficient to track only associations $asc1$ and $asc2$ appearing in the system specification. Instead, we also need to track "derived" associations during abstract execution; in the above example the relation formed by the join of the $asc1$, $asc2$ relations is one such association. This is similar in flavor to predicate abstraction based abstraction refinement [11, 54] — where tracking the predicates/conditions appearing in the program is not sufficient, and abstraction refinement gradually finds out the additional predicates to track.

9.1.2 Applications

In recent years, the concept of providing computing as a service has become increasingly popular, with several organizations such as IBM[3] and Microsoft[4], working towards providing various services and development tools in this direction. Various Cloud computing and Service Oriented Architecture (SOA) frameworks present a

[3] `http://www-01.ibm.com/software/solutions/soa/`
[4] `http://www.microsoft.com/azure/default.mspx`

major step in this direction, where the main idea is to provide software as services (computing) over the Internet/network (cloud) in a homogeneous and trustworthy manner. Thus, in the framework of service based computing, a software functionality (possibly implementing a service itself) is realized by utilizing other services over the network. This creates a kind of global distributed computing environment with various services collaborating to accomplish a common goal. Clearly, several challenges arise in this framework, for instance those pertaining to security, reliability, performance etc. which need to be investigated. We observe that various model-based design and analysis techniques from the domain of distributed reactive systems can be applied directly or adapted to the domain of cloud computing. For instance, scenario-based notations such as MSCs can be used to specify global interactions between collaborating services, while state-based notations such as Finite State Machines are useful for high level specification of individual service behaviors [96].

Based on our modeling notations and the associated abstract execution semantics, we are interested in pursuing the following research directions:

Symbolic pattern discovery in Service Oriented Architectures (SOA). Discovering or identifying service engineering patterns in existing service oriented systems can be beneficial in a number of ways, such as, for behavioral validation of services, for service-reuse by identifying services that follow specific execution pattern, etc. Since, interaction patterns can be described quite

naturally using scenario-based notations, we would like to investigate the use of our notations for generic pattern description and discovery.

Parameterized Model-based Performance and Security Analysis of Web Services. Similar to the class of distributed reactive systems targeted this thesis, many web-services may involve an arbitrary number of services at run-time. For example, an e-business system where a supplier-service may consult various warehouse-services in relation to an order placed by a customer-service. Works on performance and security analysis of web-services (e.g. [40, 16]) generally fix the number of various services in the system for the purpose of analysis. We would like to investigate approaches for performance and/or security analysis in the parameterized setting for the number of services in a given system.

Model-based testing of services. Study model-based testing of services, where the number of participating services of a certain type is not known beforehand (e.g. warehouse-service in the e-business scenario mentioned above).

Finally, we note that, even though here we have only focused on Services, our modeling notations can be utilized in any other suitable domain.

Bibliography

[1] CTAS requirements document. `http://scesm04.upb.de/case-study-2/requirements.pdf/`.

[2] Specification and Description Language. `http://www.sdl-forum.org/`.

[3] Telelogic SDL Suite. `http://www.telelogic.com/products/sdl/index.cfm`.

[4] XSB logic programming system. `http://xsb.sourceforge.net/`.

[5] S. Ali, L. C. Briand, et al. A state-based approach to integration testing based on UML models. *Information and Software Technology*, 49(11-12), 2007.

[6] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. *Theor. Comput. Sci.*, 331(1):97–114, 2005.

[7] R. Alur, G. Holzmann, and D. Peled. An Analyzer for Message Sequence Charts. In T. Margaria and B. Steffen, editors, *Intl. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 35–48. Springer Berlin / Heidelberg, 1996.

[8] R. Alur and M. Yannakakis. Model Checking of Message Sequence Charts. In *Concurrency Theory (CONCUR)*, volume 1664 of *Lecture Notes In Computer Science*, pages 114–129, 1999.

[9] Y. F. anf K. L. McMillan, A. Pnueli, , and L. D. Zuck. Liveness by Invisible Invariants. In *Formal Techniques for Networked and Distributed Systems - FORTE*, volume 4229 of *Lecture Notes In Computer Science*, pages 356–371, 2006.

[10] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[11] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, New York, NY, USA, 2001. ACM.

[12] F. Basanieri, A. Bertolino, and E. Marchetti. The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects. In *UML 2002 The Unified Modeling Language*, volume 2460 of *Lecture Notes In Computer Science*, pages 275–303, 2002.

[13] T. Basten and W. van der Aalst. Inheritance of behavior. *Journal of Logic and Algebraic Programming*, 47(2):47–145, 2001.

[14] S. Basu and C. R. Ramakrishnan. Compositional Analysis for Verification of Parameterized Systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes In Computer Science*, pages 315–330, 2003.

[15] H. Ben-Abdallah and S. Leue. Syntactic Detection of Process Divergence and Non-local Choice in Message Sequence Charts. In *Proceedings of the Third Intl. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 259–274. Springer-Verlag, 1997.

[16] A. Bertolino, G. Angelis, L. Frantzen, and A. Polini. Model-Based Generation of Testbeds for Web Services. In *Testing of Software and Communicating Systems*, volume 5047 of *Lecture Notes In Computer Science*, pages 266–282, 2008.

[17] A. Bertolino, E. Marchetti, et al. Introducing a Reasonably Complete and Coherent Approach for Model-based Testing. In *Electr. Notes in Theor. Comput. Sci.*, volume 116, pages 85–97, 2005.

[18] B.Genest, M.Minea, A.Muscholl, and D.Peled. Specifying and verifying partial order properties using template MSCs. In *Foundations of Software Science and Computation Structures (FOSSACS)*, volume 2987 of *Lecture Notes in Computer Science*, pages 195–210, 2004.

[19] L. C. Briand and Y. Labiche. 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*, pages 194–208, 2001.

[20] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Prestchner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[21] B.Sengupta. Triggered message sequence charts. *Ph.D Thesis, State University of New York (SUNY) Stony Brook*, 2003.

[22] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[23] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.

[24] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In *Concurrency Theory (CONCUR)*, volume 962 of *Lecture Notes In Computer Science*, pages 395–407, 1995.

[25] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Prog. Lang. Syst.*, 16(5):1512–1542, 1994.

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

[27] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[28] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes In Computer Science*, pages 53–68, 2000.

[29] S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *SPIN workshop on Model checking of software*, volume 2057, pages 57–79. Springer-Verlag, 2001.

[30] E. A. Emerson and V. Kahlon. Reducing Model Checking of the Many to the Few. In *CADE-17: Proceedings of the 17th International Conference on Automated Deduction*, pages 236–254, 2000.

[31] E. A. Emerson and V. Kahlon. Parameterized Model Checking of Ring-Based Message Passing Systems. In *Computer Science Logic*, volume 3210 of *Lecture Notes In Computer Science*, pages 325–339, 2004.

[32] E. A. Emerson and K. S. Namjoshi. On Reasoning About Rings. *Int. J. Found. Comput. Sci.*, 14(4):527–550, 2003.

[33] EMF. Eclipse modeling framework. `http://www.eclipse.org/modeling/emf/`.

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

[35] L. Frantzen, J. Tretmans, and T. A. C. Willemse. A Symbolic Framework for Model-Based Testing. In *Formal Approaches to Software Testing and Runtime Verification (FATES/RV)*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54, 2006.

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

[37] T. Gazagnaire, B. Genest, L. Helouet, P. S. Thiagarajan, and S. Yang. Causal Message Sequence Charts. In *Concurrency Theory (CONCUR)*, volume 4703 of *Lecture Notes In Computer Science*, 2007.

[38] B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-state high-level MSCs: Model-checking and Realizability. *J. Comput. Syst. Sci.*, 72(4):617–647, 2006.

[39] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

[40] S. Gilmore, V. Haenel, L. Kloul, and M. Maidl. Choreographing Security and Performance Analysis for Web Services. In *Formal Techniques for Computer*

Systems and Business, volume 3670 of *Lecture Notes In Computer Science*, pages 200–214, 2005.

[41] A. Goel, S. Meng, A. Roychoudhury, and P. S. Thiagarajan. Interacting process classes. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 302–311. ACM, 2006.

[42] A. Goel and A. Roychoudhury. 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*, pages 347–354. IEEE Computer Society, 2006. Invited Paper.

[43] A. Goel and A. Roychoudhury. Test Generation from Integrated System Models capturing State-based and MSC-based Notations. In K. Lodaya et al., editors, *Perspectives in Concurrency Theory (Festschrift for P.S. Thiagarajan)*. University Press (India), 2009.

[44] A. Goel, A. Roychoudhury, and P. Thiagarajan. Interacting Process Classes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(4), 2009. To appear.

[45] A. Goel, B. Sengupta, and A. Roychoudhury. Footprinter: Roundtrip Engineering via Scenario and State based Models. In *ACM International Conference on Software Engineering (ICSE)*, 2009. Short paper.

[46] W. Grieskamp. Multi-paradigmatic Model-Based Testing. In *Formal Approaches to Software Testing and Runtime Verification (FATES/RV)*, volume 4262 of *Lecture Notes in Computer Science*, pages 1–19, 2006.

[47] E. Gunter, A. Muscholl, and D. Peled. Compositional Message Sequence Charts. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 496–511, 2001.

[48] D. Harel. Statecharts: A visual formalism for complex ystems. *Science of Computer Programming*, 8(3):231–274, 1987.

[49] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[50] D. Harel and O. Kupferman. On object systems and behavioral inheritance. *IEEE Trans. Softw. Eng.*, 28(9):889–903, Sep 2002.

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

[52] D. Harel and R. Marelly. Specifying and executing behavioral requirements: The play-in/play-out apprroach. *Software and System Modeling (SoSyM)*, 2(2):82–107, 2003.

[53] J. G. Henriksen, M. Mukund, K. N. Kumar, M. Sohoni, and P. S. Thiagarajan. A theory of regular MSC languages. *Inf. Comput.*, 202(1):1–38, 2005.

[54] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70. ACM, 2002.

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

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

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

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

[59] C. Ip and D. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(2):41–75, 1996.

[60] C. N. Ip and D. L. Dill. Verifying systems with replicated components in Murφ. *Formal Methods in System Design*, 14(3):273–310, 1999.

[61] ITU-T Z.100. Specification and description language (SDL), ITU-T Recommendation Z.100. *Telecommunication Standardization Sector of ITU*, 1999.

[62] ITU-T Z.120. Message Sequence Chart (MSC), ITU-T Recommendation Z.120. *Telecommunication Standardization Sector of ITU*, 1999.

[63] C. Jard. Synthesis of distributed testers from true-concurrency models of reactive systems. *Information and Software Technology*, 45(12):805–814, 2003.

[64] C. Jard and T. Jéron. 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, 2005.

[65] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer-Verlag, 1995.

[66] B. Jonsson and M. Saksena. Systematic Acceleration in Regular Model Checking. In *Computer Aided Verification*, volume 4590 of *Lecture Notes In Computer Science*, pages 131–144, 2007.

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

[68] Y. Kesten and A. Pnueli. Control and data abstraction: the cornerstones of practical formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):328–342, 2000.

[69] A. Knapp, S. Merz, and C. Rauh. Model Checking Timed UML State Machines and Collaborations. In W. Damm and E. R. Olderog, editors, *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–416, 2002.

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

[71] I. Krueger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In *International Workshop on Distributed and Parallel Embedded Systesms (DIPES)*, pages 61–71, 1999.

[72] I. Kruger, W. Prenninger, and R. Sander. Broadcast MSCs. *Formal Aspects of Computing*, 16(3):194–209, 2004.

[73] H. Kugler, M. Stern, and E. Hubbard. Testing Scenario-Based Models. In *Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *Lecture Notes In Computer Science*, pages 306–320, 2007.

[74] E. Lee and S. Neuendorffer. Classes and subclasses in actor-oriented design. *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 161–168, 23-25 June 2004.

[75] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 346–357, 1997.

[76] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[77] S. Mellor and M. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.

[78] R. Milner. *A Calculus of Communication Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[79] MOST-cooperation. Media oriented system transport. `http://www.mostcooperation.com/`.

[80] M. R. Mousavi, M. A. Reniers, and J. F. Groote. Congruence for SOS with data. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 303–312, 2004.

[81] M. Mukund, K. N. Kumar, and M. A. Sohoni. Synthesizing Distributed Finite-State Systems from MSCs. In *Concurrency Theory (CONCUR)*, volume 1877 of *Lecture Notes In Computer Science*, pages 521–535, 2000.

[82] M. Mukund, K. N. Kumar, and P. Thiagarajan. Netcharts: Bridging the Gap between HMSCs and Executable Specifications. In *Concurrency Theory (CONCUR)*, volume 2761 of *Lecture Notes In Computer Science*, pages 296–310, 2003.

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

[84] N. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 1993.

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

[86] J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. In *UML The Unified Modeling Language*, volume 1723 of *Lecture Notes In Computer Science*, page 76, 1999.

[87] OMG. Object Management Group. website: `http://www.omg.org`.

[88] P. Pelliccione, H. Muccini, A. Bucchiarone, and F. Facchini. TeStor: Deriving Test Sequences from Model-Based Specifications. In *Component-Based Software Engineering (CBSE)*, volume 3489 of *Lecture Notes In Computer Science*, pages 267–282, 2005.

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

[90] PN. Petri Nets Tools Database.

http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/.

[91] A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, ∞)-counter abstraction. In *International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes In Computer Science*, pages 107–122, 2002.

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

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

[94] RailShuttle_System. New rail-technology Paderborn.

http://nbp-www.upb.de/en/.

[95] Rational Rose. IBM Rational Rose.

website: http://www-01.ibm.com/software/rational/.

[96] I. Rauf, M. Z. Iqbal, and Z. I. Malik. UML Based Modeling of Web Service Composition - A Survey. In *Sixth international Conference on Software Engineering Research, Management and Applications*, pages 301–307, 2008.

[97] W. Reisig. Petri Nets: An Introduction. vol. 4 of EATCS Monographs in Theoretical Computer Science, 1985.

[98] M. A. Reniers. Message sequence chart: Syntax and semantics. *PhD Thesis, TU/e*, 1999.

[99] Y. Resten, O. Maler, M. Marcus, A. Pnueli, , and E. Shahar. Symbolic model checking with rich assertional languages. In *Computer Aided Verification*, volume 1254 of *Lecture Notes In Computer Science*, pages 424–435, 1997.

[100] R. Rhapsody. Model driven development tool from IBM. `www.ibm.com/software/awdtools/rhapsody/`.

[101] A. Roychoudhury, A. Goel, and B. Sengupta. 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*, pages 275–284. ACM, 2007.

[102] A. Roychoudhury and I. Ramakrishnan. Automated inductive verification of parameterized protocols. In *International Conference on Computer Aided Verification (CAV), LNCS 2102*, pages 25–37, 2001.

[103] A. Roychoudhury and P. S. Thiagarajan. Communicating transaction processes. In *ACSD '03: Proceedings of the 3rd International Conference on Application of Concurrency to System Design*, page 157, Washington, DC, USA, 2003. IEEE Computer Society.

[104] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *IFM '00: Proceedings of the Second International Conference on Integrated Formal Methods*, pages 338–357, 2000.

[105] B. Selic. Using UML for Modeling Complex Real-Time Systems. In *Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, volume 1474 of *Lecture Notes in Computer Science*, pages 250–260, 1998.

[106] B. Sengupta and R. Cleaveland. Triggered message sequence charts. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 167–176, 2002.

[107] G. Smith. Introducing Reference Semantics via Refinement. In *ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods*, pages 588–599. Springer-Verlag, 2002.

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

[109] H. Storrle. Semantics of interactions in UML 2.0. In *IEEE Symposium on Human Centric Computing Languages and Environments*, pages 129–136, 2003.

[110] Telelogic. IBM Telelogic. website: `http://www.telelogic.org`.

[111] J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

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

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

[114] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 74–82, 2001.

[115] S. Uchitel, J. Kramer, and J. Magee. Negative scenarios for implied scenario elicitation. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 109–118, 2002.

[116] A. Ulrich and H. König. Architectures for Testing Distributed Systems. In *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems*, pages 93–108, 1999.

[117] T. Wang, A. Roychoudhury, R. Yap, and S. Choudhary. Symbolic execution of behavioral requirements. In *Practical Aspects of Declarative Languages*, volume 3057 of *Lecture Notes In Computer Science*, pages 178–192, 2004.

[118] H. Wehrheim. Behavioral subtyping relations for active objects. *Formal Methods in Sys. Design*, 23(2):143–170, 2003.

[119] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proceedings of the 22nd international conference on Software engineering (ICSE)*, pages 314–323. ACM, 2000.

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

[121] Z-Notation. The formal specification notation Z.

website: `http://formalmethods.wikia.com/wiki/Z`.

Appendix A

IPC

A.1 Proof of Theorem 1

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

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

- **Property (2)** After the concrete execution of σ, let there be n tuples of the form $\langle o_1, \ldots, o_k \rangle$ following association asc, where $o_1, \ldots, o_k$ are concrete objects, and

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

the states of $o_1, \ldots, o_k$ are defined by the behavioral partitions $beh_1, \ldots, beh_k$. Then, after the abstract execution of σ, the association information maintained is such that the *asc* count for the k-tuple $\langle beh_1, \ldots, beh_k \rangle$ is $\geq n$.

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

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

- $beh'_1, \ldots, beh'_m$ are the destination partitions of $beh_1, \ldots, beh_m$ in the abstract execution of γ. This follows from the definition of destination partition (Definition 7, page 48).

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

by the behavioral partition beh (beh') before (after) execution of γ. Then, after the execution of γ, the count of beh (beh') will be decremented (incremented) by $n_1(n_2)$.

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

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

- Case A: We have a guard $(r_{i_1}, \ldots, r_{i_k}) \in asc$ as part of γ. Suppose that there are n_1 and n_2 number of k-tuples of concrete objects in association asc, whose local states are given by the behavioral partitions $beh_1, \ldots, beh_k$ and $beh'_1, \ldots, beh'_k$ respectively, before concrete execution of γ. Now, suppose a k-tuple of concrete objects $\langle o_1, \ldots, o_k \rangle$ in association asc is chosen for concrete execution of γ, s.t. the local states of objects $o_1, \ldots, o_k$ are given by the behavioral partitions $beh_1, \ldots, beh_k$ ($beh'_1, \ldots, beh'_k$) before (after) execution of γ. Thus, after

the concrete execution of γ, the count of k-tuples in *asc* whose objects' states are given by $beh_1, \ldots, beh_k$ ($beh'_1, \ldots, beh'_k$) is decremented (incremented) by 1, resulting in association counts of '$n_1 - 1$' ($n_2 + 1$).

By the induction hypothesis, the *asc* count for the k-tuple $\langle beh_1, \ldots, beh_k \rangle$ ($\langle beh'_1, \ldots, beh'_k \rangle$) in the abstract execution is $n_a(n_b)$ after σ^{prev}, such that $n_a \geq n_1$ ($n_b \geq n_2$). Further, we can choose the behavioral partitions $beh_1, \ldots, beh_k$ as the witness partitions for roles $r_{i_1}, \ldots, r_{i_k}$ to execute γ. Then, from Property (1), the corresponding destination partitions after executing γ are $beh'_1, \ldots, beh'_k$. Thus, after the abstract execution of γ, the *asc* count for the k-tuple $\langle beh_1, \ldots, beh_k \rangle$ ($\langle beh'_1, \ldots, beh'_k \rangle$) is updated to '$n_a - 1$' ($n_b + 1$) (see the case for "Check" in the handling of associations, Section 3.5.3).

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

k-tuple is incremented by 1 in the abstract execution as well.

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

- Case D: Suppose O_γ is the set of objects chosen to play the roles in the concrete execution of γ. Let τ_c be a k-tuple consisting of objects O_{asc} in association asc such that, $\{o_1, \ldots, o_j\} = O_\gamma \cap O_{asc}$, and the local states of objects $o_1, \ldots, o_j$ are given by the behavioral partitions $beh_1, \ldots, beh_j$ $(beh'_1, \ldots, beh'_j)$ before (after) concrete execution of γ. Further let the local states of objects in k-tuple τ_c (i.e. O_{asc}) be given by the k-tuple τ (τ') of behavioral partitions before (after) concrete execution of γ. Then the k-tuple τ (τ') will contain the behavioral partitions $beh_1, \ldots, beh_j$ $(beh'_1, \ldots, beh'_j)$ representing the states of objects $o_1, \ldots, o_j$ before (after) concrete execution of γ. Note that τ and τ' may only differ in behavioral partitions corresponding to the objects $o_1, \ldots, o_j$. Thus, after the

concrete execution of γ, the count of k-tuples of objects in asc whose objects' states are given by the k-tuple τ (τ') of behavioral partitions, is decremented (incremented) by one.

By the induction hypothesis, we can choose the witness partitions BEH_γ representing the local states of objects O_γ chosen above, to play the respective roles in abstract execution of γ. Further, let BEH_τ be the set of behavioral partitions in the k-tuple τ. Then BEH_γ will contain behavioral partitions $\{beh_1, \ldots, beh_j\}$ ($= BEH_\gamma \cap BEH_\tau$) corresponding to objects $o_1, \ldots, o_j$ above. Then, from Property (1), the corresponding destination partitions after executing γ are given by $beh'_1, \ldots, beh'_j$. Thus, after the abstract execution of γ, k-tuple τ will result in τ', and the asc count for τ' is incremented by 1 (see the case for "Default" in the handling of associations, Section 3.5.3).

Note that we do not decrement the asc count of k-tuple τ in the abstract execution of γ. This is to consider the case where the objects chosen from witness partitions $beh_1, \ldots, beh_j$ may not be in association asc. In this case there will be no update in asc content in concrete execution. However, since precise association information is lost in abstract execution, we consider both the possibilities –the participating objects may or may-not be in asc– in updating the association content for asc.

This concludes the induction step for Property (2).

□

A.2 Checking spuriousness of execution runs in Murphi

Here we elaborate how we can check whether an execution run produced by our abstract simulator is spurious (*i.e.* cannot be realized in concrete executions). We have implemented the spuriousness check using the Murphi model checker [83]. The reason for using Murphi is its inherent support for symmetry reduction via the *scalarset* data type. We now discuss how Murphi's support of symmetry reduction is exploited to perform our spuriousness check efficiently.

We define the following data types for each process class.

- A *scalarset* type to act as an object identifier having the cut-off number[2] as its upper limit. For example, for Car class containing N objects, following type will be declared:

  `Car: Scalarset(N);` –index for process class Car

- Enumeration variable types which define *sets of states* of its LTS and various DFAs. Assuming that the LTS of process class Car contains M states and one of its DFAs, say dfa_i has D_i states, the following translation will result:

  `stCar: Enum {st_car1,...,st_carM};` –states for LTS of Car

  `dfai_Car: Enum {d_car_i1,...,d_car_iD`$_i$`};` –states for dfa_i of Car

[2]The number $x_{p,\sigma}$ for process class p and execution run σ, defined in Section 3.6.3

Based on the types defined above, following variables are declared for each process class:

- An array of *enumeration type representing the LTS states*, indexed by the scalarset type corresponding to this process class. For example, LTS states for objects of process class Car will be represented using the following array variable:

```
Car_lts:  Array [Car] of stCar;
```

- Similarly, array variables are defined to represent the DFA states.

- Arrays corresponding to the variables in the IPC model. Murphi supports only integer/boolean variables and the range for integer type needs to be specified in declaration. For example, variable *mode*[3] for the Car is declared as follows:

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

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

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

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

An array entry of 1 will indicate the existence of an association between the objects of A and B, whose identities are represented by the index values of that particular array element.

For each *transaction-occurrence* in the trace σ being checked, a corresponding *rule* is defined in Murphi (representing a guarded command) using the witness and destination partitions' information: *control states, dfa states* and *variable valuations* for the participating agents, obtained from the abstract execution. The initial configuration for the Murphi execution is given as the "Startstate" declaration, where the initial control states, dfa states and variable valuations for various objects are defined. If an execution run σ produced by our abstract simulation is suspected to be spurious by the user, (s)he can submit it to Murphi for spuriousness check. Given our encoding of the reduced concrete IPC model to Murphi, if σ is indeed spurious it will correspond to a deadlocked run in Murphi, with Murphi getting stuck at a spurious transaction. By slight modification in the Murphi code we are able to precisely identify the transaction that σ got stuck at and furthermore report the system state at that point. This can then be analyzed by the user to determine the cause of spuriousness.

For illustration, we analyzed the trace $\sigma = t_1.t_2.t_3$, corresponding to the example discussed in Section 3.6.3, for spuriousness using Murphi-based approach outlined in the preceding; as expected, σ was indeed found to be spurious.

Appendix B

IPC Test generation Algorithm

genTrace

The function *genTrace* described in Algorithm 4 takes as input two parameters: a set S of states to be explored, and the current goal transaction τ. It then tries to generate a trace to the current goal transaction τ from a state in S. It maintains the set *Open* containing the states yet to be explored and the set *GoalSet* to store the state(s) reached after executing the *goal* transaction τ. The set *GoalSet* is initially $\emptyset$. At the end of each iteration of the **while** loop (lines 2–35, Algorithm 4) if *GoalSet* is not empty, then we have found a trace up to the current goal transaction. We return the set *GoalSet* and also the set *Open* containing the unexplored states (lines 32–33, Algorithm 4).

In each iteration of the **while** loop, a state s with minimum value of $f(s)$ $(= g(s) + h(s))$ is chosen and removed from *Open* (line 3, Algorithm 4). All successors of

Algorithm 4: **genTrace(S, τ): adapted from** A^*

```
1: Open ← S; GoalSet ← ∅
2: while Open ≠ ∅ do
3:    s ← getMin_f(Open); Open ← Open − {s}
4:    for all m ∈ successors(s) do
5:       setSuccessor(m,s) /* Add m as a successor of s*/
6:       if notVisited(m) then
7:          /* Case 1: m not visited yet */
8:          if (m is a destination state of a τ transaction) then
9:             GoalSet ← GoalSet ∪ {m}
10:            h(m) ← distance from m to next goal transaction (after τ), use
               Equation (7.1)
11:         else
12:            Open ← Open ∪ {m}
13:            h(m) ← distance from m to τ, use Equation (7.1)
14:         end if
15:         g(m) ← g(s) + 1 /* length of generating path */
16:         setParent(m,s)
17:      else if m ∈ Open then
18:         /* Case 2: m reached earlier, successors not explored */
19:         if (g(s) + 1) < g(m) then
20:            /* we have found a shorter path to m via s */
21:            g(m) ← g(s) + 1; setParent(m,s)
22:         end if
23:      else
24:         /* Case 3: m reached earlier and successors explored */
25:         if (g(s) + 1) < g(m) then
26:            /* we have found a shorter path to m via s */
27:            g(m) ← g(s) + 1; setParent(m,s);
28:            updateAll_g(m) /*Update value of function g and parent of nodes from
               m downwards*/
29:         end if
30:      end if
31:   end for
32:   if GoalSet ≠ ∅ then
33:      return (GoalSet,Open)
34:   end if
35: end while
36: return (∅,∅)
```

s are generated and the search tree capturing states visited so far is updated. Note that in the search tree capturing the states visited so far, we maintain *one* of the predecessors of each visited state s as the "parent" of s. This is done to remember the shortest path from the start state to state s, such that it includes all previous goal transactions (i.e. transactions appearing in the test specification that have already been executed) in the order of their occurrence.

Thus, the **parent pointer** of each state s is set to a state x such that (a) s is a successor of x and (b) the current shortest path from start state to s consists of the current shortest path from start state to x (covering the previous goal transactions in the order executed) and the edge $x \rightarrow s$. For each successor m of state s, the following updates need to be performed.

Case 1: (line 7, Algorithm 4) If m has *not* been reached earlier, it is added either to *GoalSet* or *Open* (lines 9 and 12, Algorithm 4), depending on whether m is reached by executing the current goal transaction τ or not. The value of h function (appearing in A^*'s evaluation function) is also computed accordingly. Finally, we compute the value of g function of A^*'s evaluation function, and set the node s as m's parent in the search tree capturing visited states.

Case 2: (line 18, Algorithm 4) Otherwise, we check if $m \in Open$ and if so, we know that m has been reached earlier via an alternate path. In this case if the new path to m is shorter, we accordingly update the function g for m. Note that the function h remains unchanged since it estimates the length of the path from m to τ, which is

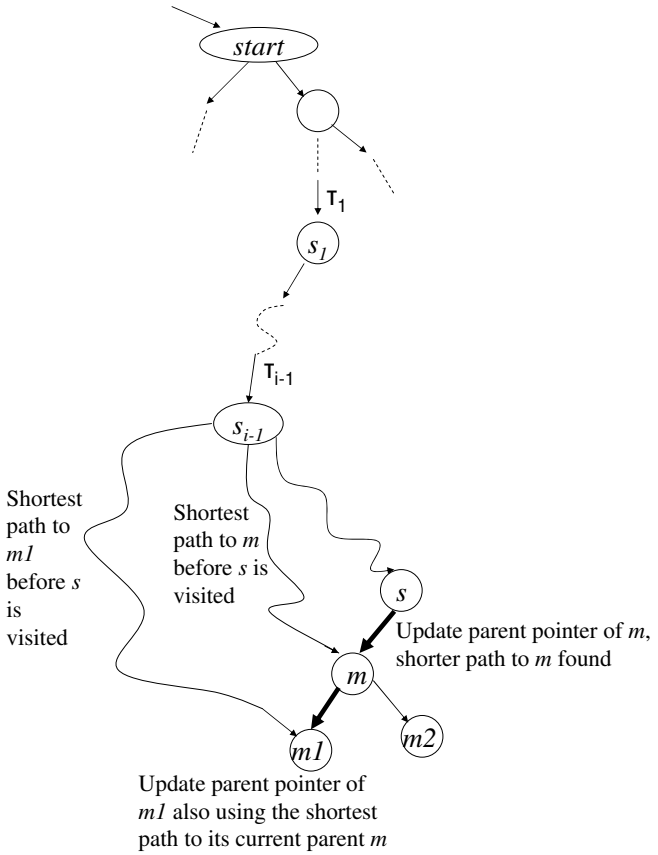


Figure B-1: Updating parent pointers in search tree — line 28 of *genTrace* (Alg. 4).

not affected by the path through which m is reached from the start state.

Case 3: (line 24, Algorithm 4) In case m is not present in *Open* also, then it was reached *and* its successors were explored earlier. Again, we check if the new path to m is shorter. If it is so, then its g value is updated and parent pointer set to s. Further, in this case we check the children of m in the search tree capturing the visited states. For all the successors already pointing to m as their parent, their g value is updated. For all other successors, if the new path to them via m is shorter, then m is set as their parent, also updating their g value. This process is repeated for successors of these successors and so on (see Figure B-1 for a pictorial explanation).