Synthesis and Traceability of Scenario-based Executable Models

Ankit Goel Abhik Roychoudhury
Department of Computer Science, National University of Singapore
Email: {ankit,abhik}@comp.nus.edu.sg

***Abstract*— Message Sequence Charts (MSCs) or Sequence Diagrams are one of the behavioral diagram types in the Unified Modeling Language or UML. In system requirements modeling, MSCs are conventionally used for describing possible system scenarios. In the recent past, there have been concerted attempts to develop executable system modeling languages directly based on MSCs — Live Sequence Charts, Triggered Message Sequence Charts and Interacting Process Classes, to name a few. In this paper, we study the problem of model synthesis in these languages — how to translate informal requirements into formal models. We also discuss (a) test generation from these formal models, and (b) how the generated tests can be traced back to the informal requirements.**

Index terms— Computer aided software engineering, Software requirements and specifications.

I. Introduction

Message Sequence Charts (MSCs) or Sequence Diagrams 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 processes constituting a distributed reactive system may communicate among each other as well as with the environment. Due to their widespread usage in requirements engineering, MSCs have been integrated into the Unified Modeling Language (UML). In UML, they are popularly known as Sequence Diagrams.

Visually, a Message Sequence Chart (MSC) consists of a number of interacting processes each shown as a vertical line. Message communication between the processes are shown as horizontal or downward sloping arrows. Message communication can be synchronous (sender and receiver handshake) or asynchronous (non-blocking send events). In this paper, we always assume asynchronous message communication since it is less restrictive. Semantically, an MSC denotes a labeled partial order of events. This partial order is the transitive closure of (a) the total order of the events in each process and (b) the ordering imposed by a message's communication — a message is received *after* it is sent. An example MSC appears in Figure 1. Based on the partial order denoted by this MSC, we cannot conclude any ordering between the receive event of message $m1$ and the send event of message $m2$.

Since an MSC denotes a single scenario in system execution, it does not form a complete system description. This problem can be alleviated by MSC-graphs. Each node of an MSC-graph is an MSC. Thus, an MSC-graph represents execution traces formed by unfolding the MSC-graph from initial

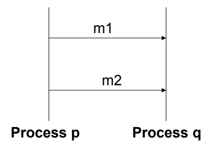


Fig. 1. *A Message Sequence Chart (MSC)*

state(s) and then linearizing the MSC sequences thus obtained. The model of MSC graphs is suitable for complete system descriptions; extended with hierarchy it forms the popular notation of High-level Message Sequence Charts (HMSCs). However, we cannot directly construct implementations for the participating processes by projecting out their behaviors within the MSCs of an HMSC, due to the presence of "implied scenarios" [1].

To overcome such problems, executable formalisms directly based on MSCs have been proposed in recent years. In particular, the Live Sequence Chart formalism [2], [3] extends MSCs (which only describe "possible behaviors") to describe mandatory behavior. The execution semantics of an LSC specification is of a centralized nature. In contrast, the work on Triggered Message Sequence Charts (TMSCs) [4] allows for a per-process execution semantics. Both LSCs and TMSCs suppress the computation/control flow within processes, emphasizing the inter-process communication. The overall system behavior is captured by expressing temporal constraints on the relative order of process interactions. To balance the intra-process and inter-process modeling styles we have proposed the Communicating Transaction Processes (CTP) formalism [5] and extended it to process classes via the Interacting Process Classes (IPC) model [6]. Here, the system modeling is done at two levels. At the top level, the processes are described by transition systems; however, the action labels in these transition systems correspond to guarded (collections of) Message Sequence Charts.

In this article, we take a fresh look at these different executable modeling languages based on MSCs. The formal syntax and semantics of these modeling languages have been studied in their respective books/papers [2], [3], [7], [5], [6], [4]. Our goal in this article is to study the suitability of these modeling languages for modeling various kinds of informal requirements found in practice. This is *different* from the play-in approach advocated in [3] where the user inputs the requirements by making certain choices and the user choices

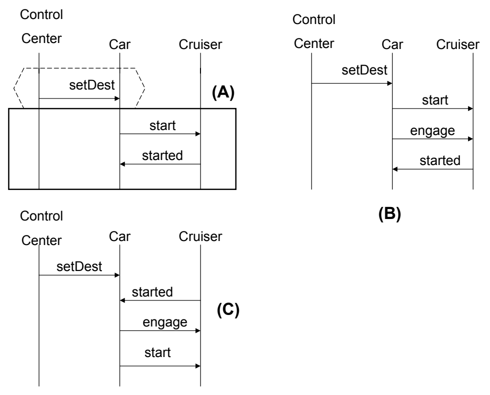


Fig. 2. *(A) An LSC universal chart, (B) An execution respecting the universal chart, and (C) An execution not respecting the universal chart*

internally get converted to Live Sequence Charts. In this paper, we only consider the (more modest but more common) task of converting an informal requirements document to a formal specification in a scenario-based modeling language. We also discuss how fragments of the model (captured by test cases) can be mapped back to the informal requirements.

In summary, the aim of this article is to initiate and further study into the issue of relating MSC-based informal requirements to MSC-based executable models. Currently, the requirements documents refer to MSCs (which capture inter-process communication), whereas the design models are given by state diagrams (capturing intra-process behavior). This creates a gap which needs to be filled in the manual synthesis of the design models. We believe that MSC-based executable design models will be more tightly linked to the requirements and hence will allow better traceability of models to informal requirements.

II. MSC-based Modeling Languages

In this section, we summarize three major efforts in building MSC-based executable languages — Live Sequence Charts (LSCs), Triggered Message Sequence Charts (TMSCs) and Interacting Process Classes (IPC).

A. Live Sequence Charts

Live Sequence Charts (LSCs) [2] is a powerful visual formalism which serves as an enriched requirements specification language. Descriptions in the LSC language are executable, and the execution engine which supports it is called the *Play Engine*. In the Live Sequence Chart (LSC) terminology, each chart is a concatenation of a pre-chart followed by a body chart. The notion of concatenation requires some explanation. Consider a chart $Pre \circ Body$ where $\circ$ denotes concatenation. This means that all processes first execute the chart Pre and then they execute the chart $Body$; no event of chart $Body$ takes place before any event of chart Pre.

In the LSC language, charts are classified as existential or universal. An existential chart $Pre \circ Body$ represents the following property : there exists a reachable state of the system model from which an outgoing trace executes a linearization of Pre followed by a linearization of $Body$. On the other hand, a system model satisfies a universal chart $Pre \circ Body$ iff : from every reachable state of the system model, if a linearization of the pre-chart Pre is executed, then it must be followed by a linearization of the body chart $Body$. In other words, for any execution trace of the system model, whenever Pre is executed, $Body$ *must* be executed. Along with the charts being universal or existential, LSCs also allow locations or events in a chart to be universal or existential in a similar fashion [2].

An example LSC universal chart, adapted from Harel and Gery's rail-car example [8], [2] appears in Figure 2(A). The requirements specified by this chart should be interpreted as follows — *whenever the control-center instructs a car to leave for certain destination, the car must eventually start its cruiser, followed by the cruiser confirming once it gets started*. Figure 2(B) shows an execution which obeys the event ordering specified by Figure 2(A), while Figure 2(C) shows an execution which does not obey the Figure 2(A)'s event ordering. Note that the occurrence of messages which do not appear in a universal chart (such as the message "engage" in Figure 2(A)) are not constrained by the chart.

B. Triggered Message Sequence Charts

Triggered Message Sequence Charts (TMSCs) [4] extend Message Sequence Charts with *conditional* and *partial* scenarios together with mathematically precise semantics for execution and refinement. A TMSC is visually similar to an MSC with events comprising of sending and receiving of messages amongst processes, and internal actions. However in addition, all the instances (vertical lines) in a TMSC are divided into two parts: a **trigger** and an **action**. In any system execution, if the sequence of events constituting the *trigger* of a process occurs, then subsequent behavior of that process is restricted to the sequence of events described by its *action* part. The execution of *action* part following the *trigger* in TMSCs, resembles the execution of body-chart following a pre-chart in a LSC. However, in the case of LSCs all processes in the chart synchronize at the end of the pre-chart, whereas for TMSCs the conditional execution is described per-process. An example TMSC appears in Figure 3(A). The dashed horizontal line divides each instance into the *trigger* (upper half) and *action* segments (lower half). According to this TMSC, once a *car* receives the 'setDest' message from the *control-center*, its *trigger* is complete; it must then ask its *cruiser* to 'start'. Similarly, the *cruiser* must confirm with message 'started' once it has been asked to 'start'.

The individual TMSCs (such as TM-A and TM-B in Figure 3) can be combined to generate larger and more realistic specifications (called TMSC expressions), using the process-algebraic operators: '$\|$' for parallel composition, '$\mp$' for delayed choice, ';' for sequential composition[1], and '*recX*' for expressing recursive behavior. The "logical" conjunction operator '$\wedge$' is used to constrain the behavior of one TMSC

[1]The sequential composition involves *asynchronous* concatenation of MSCs [9].

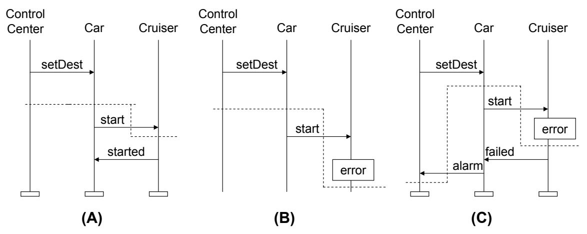


Fig. 3. *(A) TM-A: TMSC illustrating* trigger *and* action, *(B) TM-B: A partial TMSC, behavior of processes after this TMSC is unrestricted as shown by absence of bars at the foot of vertical lines, and (C) TM-C: A possible completion of TM-B*

expression with respect to another. The formal semantics of TMSCs is based on the concept of 'acceptance trees', which also help in defining a mathematically precise notion of one TMSC expression refining another.

C. Interacting Process Classes

Interacting Process Classes (IPC) [6], model a reactive system as a network of process classes where processes with similar functionalities are grouped into a class. For each process class p, a labeled transition system TS_p captures the control flow of the objects belonging to that class. However, a transition label rather than being an individual event such as a message send or receive, represents the part played by an object of that class in a short interaction protocol. Since we depict protocols using MSCs, a transition label thus becomes a lifeline of an MSC. Hence, each transition label is of the form γ_r, where γ names the interaction protocol (also called a **Transaction** in our terminology) and r is the role played by an object of the class in the transaction.

Formally, a *Transaction* is a guarded Message Sequence Chart. If a process class p has a transition label γ_r appearing in its transition system, then this label will correspond to an object of class p playing the role r in MSC γ. When we draw the MSC γ, we will often annotate this lifeline as p_r to emphasize that the role played is r and it is being played by an object of class p. In a transaction, the **guard** associated with lifeline p_r will specify the conditions that must be satisfied by an object belonging to the class p in order for it to be eligible to play the role r. These conditions consist of two components: (i) a *history* property of the execution sequence (sequence of transition labels) that the object has so far gone through (ii) a *propositional formula* built from boolean predicates regarding the values of the variables owned by the object.

Consider the transaction *DepartReqA* shown in Figure 4(A). This captures the same interaction which was depicted as LSC in Figure 2(A). However, in this case the initial receipt of 'setDest' message by a *car* from the *control-center* occurs in a separate transaction: either *SetDest* or *SelectDest* (not shown here). Thus, the Car object wishing to play the lifeline Car_{SndReq} must have last played the lifeline $Car_{RcvDest}$ in the transaction $SetDest$ or in the transaction $SelectDest$. This is captured by the regular expression guard $Act^{\star}_{car}.(SetDest_{RcvDest}|SelectDest_{RcvDest})$ shown in Figure 4(A); here Act_{car} denotes the set of lifelines in which objects of the *car* class can participate in. Thus, regular expressions are used to specify the history component of a guard. Snippets of the labeled transition systems of the Car and Cruiser classes are shown in Figure 4(B) and (C) (occurrences of the *DepartReqA* action label in the transitions systems are marked in bold).

III. Which Requirements Documents are Suitable for which Language?

In this section, we discuss the possible usage of MSCs in requirements documents and the issues involved in constructing MSC-based executable specifications from these informal requirements. In the following, we assume that the system being modeled is a reactive system consisting of several interacting processes.

A. Documents capturing overall system scenarios via MSCs

One obvious usage of MSCs in requirements is just to plainly describe system scenarios. In these kind of documents, usually

- the intra-process control flow for each process in the system is presented at a very high level using state machine like notation, and
- snippets of typical interactions between processes (as well as interactions between processes and environment) are presented as MSCs.

The informality in these kind of requirements stems from the per-process state machines not being detailed enough, and all possible system interactions not being given (only "important" use-cases are captured).

We have had some experience in system modeling from such documents, the most notable being the requirements document of *Media-oriented Systems Transport* or MOST. The MOST [10] 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. It has been designed to suit applications that need to network multimedia information along with data and control functions. The processes in this system consist of a network master and several network slaves.

From our experience in modeling MOST, we found that *the hybrid nature of the Interacting Process Classes (IPC) model* makes it suitable for this kind of requirements documents. In particular, the IPC model captures both state machine like and MSC like notations — albeit at different levels — the state machine like notation at the top level and the MSC like notations at the lower level. Thus, we could obtain the high-level Labeled Transition Systems (LTSs) of the processes by elaborating the control flow described in the requirements document. Similarly, the MSCs appearing in the requirements documents (system scenarios) will typically be broken into several MSCs appearing in the IPC model (these are the MSCs which are mentioned in the action labels of the LTSs). This

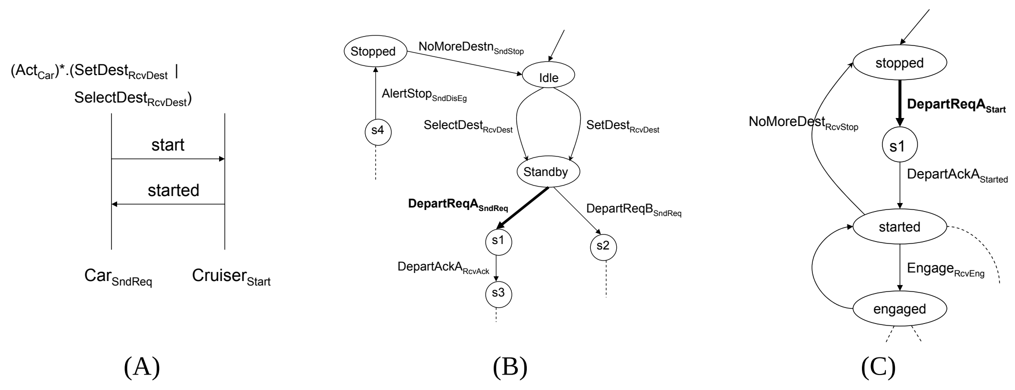


Fig. 4. *(A) Transaction* DepartReqA, *(B) Labeled Transition System snippet of Car class, and (C) Cruiser class*

process is of course manual, but the system scenarios in the requirements document form a useful guide about the alphabet of MSCs appearing in the IPC model.

Since LSCs/TMSCs *suppress intra-process control flow*, obtaining a LSC or TMSC specification from such requirements documents involves a higher-level of human ingenuity. It involves understanding the global control flow from the per-process control flow described in the documents and encoding the global control flow as pre-charts of LSCs, or triggers of TMSCs.

B. Documents describing system behavior as global constraints

Certain requirements documents structure the system behavior into phases. Usually, these documents model a system with a central controller which manages the communication between processes as well as with the environment. The system behavior is then given from the perspective of the controller, trying to split its possible behaviors into phases; for each phase the pre-condition for entering that phase as well as the actions to be executed in that phase are specified.

We have had experience in developing system specifications from such documents, the most notable being the weather-update controller from Center TRACON automation system (CTAS) tools developed by NASA to manage incoming air-traffic in busy airports [12]. The weather-update controller in CTAS consists of a central communication manager (CM) and several weather-aware clients. The CM manages the communication of weather updates to the clients. In the CTAS requirements document[2], the requirements are given from the view-point of the CM describing its behavior under given pre-conditions. The execution sequence *-2.6.2, 2.8.3, 2.8.5, 2.8.8-* of requirements taken from the requirements document, forms the scenario in which a weather aware client gets successfully connected to CM. The requirements *2.6.2* and *2.8.5* are shown in Table I. We now give their translation into LSCs, TMSCs and IPC modeling framework and comment on the relative effort required in doing so.

[2]http://scesm04.upb.de/case-study-2/requirements.pdf

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'
(b) set the weather-cycle status to
    'pre-initializing'
(c) disable the weather control panel ....
```

Requirement 2.8.5:*The CM should perform the following actions when the Weather Cycle status is 'initializing' and the newly connected weather-aware client has responded yes to the CTAS_GET_NEW_WTHR message...*

```
(a) set the Weather Cycle status to
    'post-initializing'
(b) set the weather status of the newly
    connected weather-aware client to
    'post-initializing'
(c) it should send a CTAS_USE_NEW_WTHR message
    to the newly connected weather-aware
    client.
```

TABLE I

Requirements from CTAS weather controller

Requirements of the form shown in Table I can be directly modeled as LSC universal charts. The pre-condition (marked in italics above) becomes the pre-chart, and the actions (marked as (a), (b), (c) above) become the body chart of the universal chart. A collection of such requirements becomes a collection of LSC universal charts — readily yielding an executable system model directly from requirements. The above two requirements modeled using LSCs appear in Figure 5. In order to represent various weather-aware clients, the lifeline corresponding to the weather-aware client has been marked as a symbolic-instance of class 'WAclient'. This would allow any object of this class to execute this lifeline, if it satisfies the the associated condition shown in the bubble: 'connected = false', i.e. client is not already connected to the controller.

For modeling such requirements as IPC or TMSCs, a bit more work is required. The TMSCs for the above given requirements appear in Figure 6. Note that for requirement *2.8.5*– in the case of LSCs the enabling condition for this requirement appears in the pre-chart, whereas for TMSCs we need to explicitly use process-algebraic operators(which

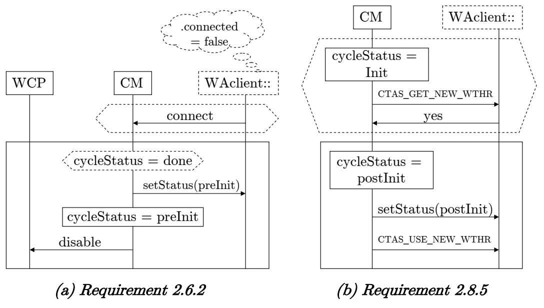


Fig. 5. *CTAS requirements modeled using LSCs.*

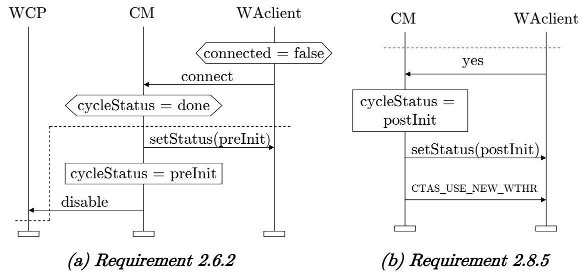


Fig. 6. *CTAS requirements modeled using TMSCs.*

are part of the TMSC language) to capture the enabling conditions and model the control flow. Assuming that TMSCs M_1, M_2, M_3, and M_4 model the requirements *2.6.2, 2.8.3, 2.8.5, 2.8.8* respectively, the complete TMSC specification would be $M_1; M_2; M_3; M_4$, where operator ';' represents the asynchronous sequential composition. In case of IPC, roughly speaking, the body charts in the LSC modeling of such requirements become the MSCs appearing in the IPC model. However, again the pre-charts (which can be directly lifted from such requirements) need to be encoded indirectly. The pre-charts basically convey which system phase can follow which phase — a high-level global control flow information. In the case of the IPC model, this is encoded via the intra-process control flow as shown by the high-level Labeled Transition Systems (LTSs) of the processes in the system. The two requirements shown earlier, together with the related part of CM's Labeled Transition System appears in Figure 7. Obtaining the LTSs for various process classes in this case needs some additional effort.

C. Documents involving mixture of both

It is conceivable that certain requirements documents employ a mixture of system scenarios (as MSCs) and global constraints on system behavior. Works on TMSCs [13] mention the requirements for a medical device involving blood-pressure measuring/pumping and presents this case as one where TMSCs may come in handy.

For these cases the IPC model is also directly useful since the system scenarios are encoded as MSCs appearing in the high-level LTSs of the processes, and the global constraints are encoded by the per-process control flow and the regular

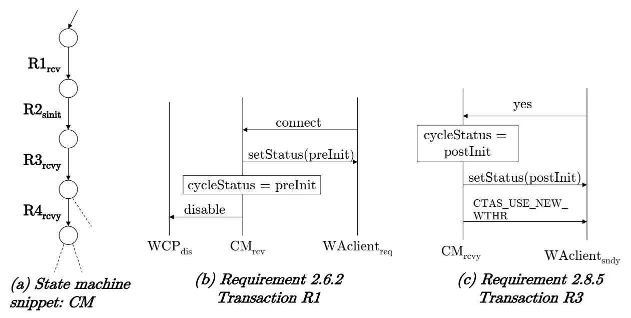


Fig. 7. *CTAS requirements modeled using IPC.*

expression guards in MSCs. We modeled the example medical device monitor of [13] in this manner. Currently, we are trying to see whether this kind of modeling scales up to more complicated requirements documents of this nature.

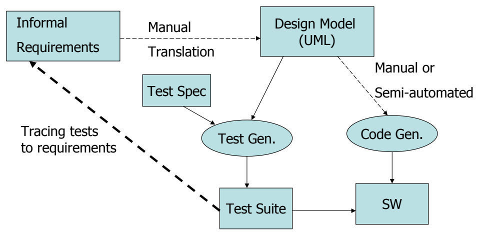
IV. Traceability of models to requirements

We now step back to look at the bigger picture: the role of MSC-based executable models in model-driven software development.

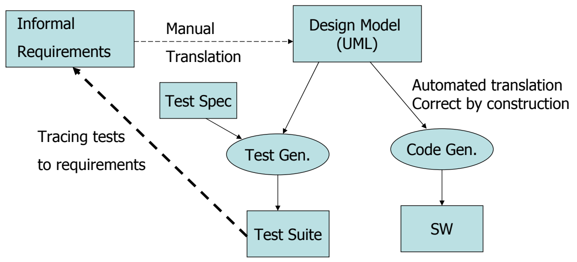
A. How MSC-based models fit into software design flows?

Conventionally, model-driven software development starts with the informal requirements from where a design model is manually synthesized. From the design model, the code is generated semi-automatically. Indeed, many UML-based tools (such as Mastercraft from TRDDC and Rational Rose from IBM) use the UML models as a guidance for code generation; the code generation from UML is, by no means, fully automated. However, the UML models can be used, along with test specifications, for automated test case generation. These test cases can be tried out on the semi-automatically generated code in order to gain confidence in the correctness of model-code translation. The overall software design flow is summarized in Figure 8 (a). The broken arrows involve manual or semi-automated steps, whereas a solid arrow denotes an automated step.

As shown in Figure 8 (a), the test cases tried out on software may need to be traced back to the informal requirements. This is indeed an important issue in many safety-critical application domains such as avionics. These domains require *certification* of software, and for the purposes of the certification, the software needs to be extensively tested. For example, the highest level of certification for flight-control software — the DO178-B level A from the Federal Aviation Administration or FAA — requires a strict criterion for structural testing of software called Modified Condition/Decision Coverage or MC/DC criterion. However, even if some offending test cases are found how does the development team discuss the intended behavior of these cases with the software design team or the requirements engineers? Currently, it is done in an ad-hoc fashion involving an informal dialogue between designers, developers and end-users. To enable more systematic design/testing of model-driven software, it is important to



(a)



(b)

Fig. 8. *Design Flows for Model-driven Software Development*

trace test cases back to informal requirements. However, the test cases are (automatically) derived from the design models. Thus, if the design models are not tightly coupled with the informal requirements, backwards traceability of tests to requirements remains a hard problem.

In Figure 8 (b), we show an alternative design flow for model-driven software development. In this flow, the code is automatically generated from the UML design models. Indeed this is the case in the Rhapsody tool from I-Logix where code can be generated fully automatically from Statechart models. Since both the code and the test cases are automatically generated from the design models, it is therefore meaningless in this situation to apply the test cases on the automatically generated code. Indeed, in the design flow of Figure 8 (b), the test generation and code generation are decoupled from each other. In such a design flow, the test cases serve a *completely different* purpose — they are used to gain high confidence in the model itself. Since the model is manually generated from the requirements, we need to understand whether the model faithfully captures the requirements. By automatically generating test-cases from the models, and tracing these tests back to the requirements — we can gain increased confidence in the models.

In the preceding, we have discussed two different design flows for model-driven software development. We observe that in both of them it is important to trace test cases (generated from models) to informal requirements. In other words, the design models should support (a) automated test generation, (b) traceability of generated tests to requirements and (c) at least semi-automatic code generation. Existing UML tools describe of the behavior of any class of active objects via a State Diagram; the relationship across classes is captured by a class diagram. For such a design model, automated test generation is easy — it proceeds by automated exploration of the State diagrams. However, as discussed earlier, it is difficult to synthesize the State Diagram models from the informal requirements (which are often inter-process in nature being described via MSCs). Conventional MSC-based system models such as MSC-graphs or HMSCs suffer from the reverse problem — they can be easily synthesized from the requirements, but since they are not executable in nature it may be difficult to synthesize tests which can be directly executed on the software.

We believe that hybrid design models (like the IPC model) offer a compromise —

- Since they are executable, we can use them for automated test generation. The test generation algorithms of course need to be modified.
- Since MSCs are the building blocks of these models, they are closely tied to the requirements. In particular, the MSCs in the system model can be (fragments of) MSCs appearing in the requirements document so that test cases generated from the models can be easily traced back to the requirements.

B. Test generation and traceability

We now outline an MSC-based approach to model-based test generation and traceability. The aim is not to provide one crisp solution, but rather highlight the issues that may arise in such an approach.

Modeling Language: First, we seek to outline the criteria that the system modeling language should satisfy. We feel that the system model should be amenable to symbolic execution — both in terms of symbolically representing data values as well as process states (when there are many behaviorally similar processes in the system). Why is symbolic execution important for model-based testing? We note that model-based testing typically proceeds by state-space exploration and hence lossless symbolic representations of the state transition graph are of immense importance. Moreover, such symbolic state space representation can enable grouping of similar test cases into a "symbolic" test case which is also very useful to the designer. In particular, symbolic execution of *process classes* seem to be important for model-based testing of distributed control systems. In such systems, a process class may capture a large number of objects of similar behavior; thus, the same state machine may be used to capture the behavior of many objects which are grouped into a class (such as the `phone` and `switch` classes in a telecommunication protocol).

The second issue concerns the granularity of the transitions for the system model that will be generated from the modeling language of choice. For MSC-based languages, there can be at least two choices.

- a transition corresponds to a single event — like a message send/receive.

- a transition corresponds to an entire Message Sequence Chart (MSC).

The second choice allows us to neatly define system execution traces as sequence of MSCs. This is the choice we have taken in our work, but more discussion is needed on this front. In our experiments on model-based testing, we have worked with the Interacting Process Classes (IPC) model. This is primarily because (a) we have been working with this model for number of years, (b) the IPC model provides symbolic execution semantics, and (c) the IPC model describes a transition as an MSC (rather than an event).

Test Specification and Coverage Criteria: The test criteria for model-based testing can be one of achieving coverage or to meet a test specification. For model-based testing, we feel that new notions of model coverage should be developed instead of re-using existing notions of code-coverage (which were originally developed for sequential programs). Thus, if we have a state-based specification where the behavior of each class is given by a state diagram, simply achieving coverage of all the local states in the state machines of the different classes *does not* suffice. The notion of test coverage needs to capture the interactions across state machines of different classes.

In the context of MSC-based models, this problem seems to have a natural solution. Consider an IPC system model where each class is described by a labeled transition system (LTS) and the action labels of the LTS correspond to MSCs. Since the MSCs form the units of execution for the entire system specification (which is given by a set of communicating LTSs), our test coverage can simply require the test cases to include all the MSCs appearing in the system specification. More discussion is needed on alternate notions of coverage in the context of model-based test generation.

Apart from test coverage, we also need to study how test specifications can be described. Again, in the context of MSC based models, we can have the test specification as a sequence of MSCs (all of which appear in the system specification). The user gives a sequence of MSCs $\tau_1\tau_2\cdots\tau_n$ as the test specification. At this level of abstraction it is much more intuitive and easier for the user to design test specifications corresponding to various *use case* scenarios. The test-case generation procedure aims at producing one or more test sequences of the form

$$\tau_1^1\cdots\tau_1^{i1}\tau_1\tau_2^1\cdots\tau_2^{i2}\tau_2\cdots\tau_n^{in}\tau_n$$

This can be viewed as finding a witness trace satisfying the Linear-time Temporal Logic (LTL) property

$$\mathbf{F}(\tau_1 \wedge \mathbf{F}(\tau_2 \wedge (\ldots(\mathbf{F}\tau_n)\ldots)))$$

We will 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.

Experiments: We now present some initial experiments on test case generation from MSC based system models. The examples used are standard reactive control systems from different application domains like avionics and rail-network control. For our initial experiments, we modeled a simple telephone switch drawn from [14]. Next we modeled the rail-car system whose behavioral requirements have been specified using Statecharts in [8] and using Live Sequence Charts in [2]. We have also modeled the requirement specification of two other systems - one drawn from the rail transportation domain and the other being a weather controller which is part of an air traffic control system (see `http://scesm04.upb.de/case-studies.html` for more details of these examples). The example from rail transportation domain is a railway shuttle system, where various shuttles bid for orders to transport passengers between various stations on a railway-interconnection network. The weather update controller [12] 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.

Examples	# Classes	# MSCs	Max. # Local States in any class
Phone	6	16	8
Railcar	12	31	16
Shuttle	4	29	28
CTAS	3	21	20

TABLE II
MODELING SUMMARY

For our experiments, we made the following choices. We chose IPC [6] as the modeling language. A test case specification is a sequence of MSCs appearing in the IPC model, and it is met by any global execution (which is a sequence of MSCs) containing the test case specification as a subsequence. Detailed modeling of our examples in the IPC model can be accessed from the web-site [11]. In Table II, we give some statistics about the modeling of these examples.

We compared the size of test-suite and test generation times with corresponding numbers where the test generation is performed by *concrete* model execution (*i.e.*, the state of each active object is maintained separately). All experiments were performed on a machine with 3GHz CPU and 1 GB of memory. The results appear in Table III. *The results serve as a measure of the importance of symbolic execution for model-based test generation.* For each test specification, the state-space of the models were explored (using both symbolic and concrete execution semantics) in *breadth-first manner* up to a given depth, and the total number of all possible witness-traces corresponding to the given test specification were recorded within that exploration tree. The time to generate the test cases is much lower in symbolic execution as compared to concrete execution. More importantly, using symbolic execution we can group together many behaviorally similar tests into a single test case. For example in the first test specification of the Telephone Network example (see Table III), 560 concrete test cases are grouped into 70 symbolic tests. Note that the number of concrete tests in Table III 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.

Example	Test spec. #	Exploration Depth	# Witnesses		Exec. Times(sec)	
			S	**C**	**S**	**C**
Telephone Network (5 phones)	1	5	70	560	1	28
	2	8	12	80	7	138
	3	9	12	80	17	212
RailCar (6 cars, 3 terminals)	1	18	6	—	380	$>$ 10 min.
	2	15	32	—	19	$>$ 10 min.
	3	15	62	—	19	$>$ 10 min.
Automated Shuttle (5 shuttles)	1	15	5	11	0.7	15
	2	15	9	18	0.72	15
	3	17	2	5	2	63
Weather Controller (10 clients)	1	20	5	129	0.1	2.5
	2	20	2	3	0.1	2.5
	3	25	3	7	0.15	12

TABLE III
COMPARING SYMBOLIC AND CONCRETE TEST GENERATION, S $\equiv$ SYMBOLIC, C $\equiv$ CONCRETE

Finally, we tried out our notion of MSC coverage for the IPC modeling of the four example controllers mentioned in Table III. Again, the MSC coverage was conducted for symbolic execution of the model. *If we try to achieve MSC coverage by concrete execution, the test-suite construction does not even terminate in reasonable time for most examples even with variations of the search strategy.* Using symbolic execution of the IPC model, coverage was achieved within 20 seconds for all the examples (on a machine with 3GHz CPU and 1 GB of memory). The test-suite sizes thereby produced ranged from 10 to 700.

Tracing back to requirements: We found that requirements which are given in a declarative form (*i.e.*, as a set of temporal constraints restricting order of events) can be relatively directly mapped to MSCs. This is the case for the CTAS weather controller [12] from NASA. Here, the individual requirement rules directly convert to LSC universal charts or to MSCs appearing in an IPC system model. Now note that our tests are sequences of MSCs, where the MSCs appearing in the tests are drawn from the system model. Since the MSCs in the system model directly map to requirement rules, this makes the task of mapping back tests (generated from models) to informal requirements easier. On the other hand, in many cases informal requirements are given in a more operational manner. The requirements may contain MSCs giving overall system scenarios involving several processes; moreover, the high-level behavior of each process may be given by a state machine. An example of such requirements is the MOST protocol from automotive domain [10] described earlier. In this case, the MSCs in the requirements document do not directly correspond to the MSCs in the executable system model. So, when we have a test case (*i.e.*, a sequence of MSCs taken from the system model), it may involve several MSCs in the requirements document. Clearly, this can occur when the requirements engineer had not envisioned "improbable" test cases (not captured as system scenario MSCs in the informal requirements, but found by exploration of the formal models). Tracing tests back to requirements is difficult in such situations. This remains an topic of our future work.

Overall Summary: In summary, our proposal for model-based test generation/traceability is as follows.

- *System Model*: An executable MSC based modeling language which helps us link requirements to formal models.
- *Test specification*: A sequence of MSCs $\tau_1, \ldots, \tau_n$ where for all i, τ_i is an MSC appearing in the system model.
- *Test case:* A sequence of MSCs from system model.
- *Meeting a Test specification:* Any test case which contains the test specification as a subsequence.
- *Achieving Test Coverage:* A set of test cases (test-suite) which contains all MSCs appearing in the system model.

V. DISCUSSION

In this paper, we have studied three different scenario-based system modeling languages — Live Sequence Charts (LSCs), Triggered Message Sequence Charts (TMSCs) and Interacting Process Classes (IPC). Specifically, we have looked at the issue of *synthesizing formal models* in these languages from informal specification documents. We have also studied the issue of generating test cases from these models and tracing these tests back to the informal requirements.

Scenario-based modeling languages are important in the context of UML-based system modeling. Traceability of test cases derived from such models is an exciting research topic of practical value. Different ways of modeling and analyzing timing constraints in such scenario-based modeling languages is another exciting research direction worth pursuing.

ACKNOWLEDGMENTS

We are thankful to the track organizers for inviting our paper. This work was partially supported by a Public Sector research grant from A*STAR, Singapore.

REFERENCES

[1] S. Uchitel, J. Kramer, and J. Magee, "Detcting implied scenarios in message sequence chart specifications," in *ESEC-FSE*, 2001.
[2] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," *Formal Methods in System Design*, 2001.
[3] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
[4] B. Sengupta and R. Cleaveland, "Triggered message sequence charts," in *FSE*, 2002.
[5] A. Roychoudhury and P. Thiagarajan, "Communicating transaction processes," in *ACSD*, 2003.
[6] A. Goel, S. Meng, A. Roychoudhury, and P. Thiagarajan, "Interacting process classes," in *ICSE*, 2006.
[7] D. Harel and P. Thiagarajan, "Message sequence charts," in *UML for Real: Design of Embedded Real-time Systems*, L. Lavagno, G. Martin, and B. Selic, Eds. Kluwer Academic Publishers, 2003.
[8] D. Harel and E. Gery, "Executable object modeling with statecharts," *IEEE Computer*, vol. 30, no. 7, 1997.
[9] R. Alur and M. Yannakakis, "Model checking of message sequence charts," in *CONCUR*, 1999.
[10] MOST cooperation, "Media oriented system transport," http://www.mostcooperation.com/.
[11] A. Goel, "Simulation tool for interacting process classes," National University of Singapore, 2006, http://www.comp.nus.edu.sg/~release/simulator.
[12] CTAS, "Center TRACON automation system," NASA, http://www.ctas.arc.nasa.gov.
[13] B. Sengupta and R. Cleaveland, "An integrated framework for scenarios and state machines," in *IFM*, 2005.
[14] G. Holzmann, *Modeling a Simple Telephone Switch*, ser. The SPIN Model Checker. Addison-Wesley, 2004, ch. 14.