Synthesis of Behavioral Models from Scenarios

Sebastian Uchitel, *Member*, *IEEE Computer Society*, Jeff Kramer, *Member*, *IEEE Computer Society*, and Jeff Magee, *Member*, *IEEE*

Abstract—Scenario-based specifications such as Message Sequence Charts (MSCs) are useful as part of a requirements specification. A scenario is a partial story, describing how system components, the environment, and users work concurrently and interact in order to provide system level functionality. Scenarios need to be combined to provide a more complete description of system behavior. Consequently, scenario synthesis is central to the effective use of scenario descriptions. How should a set of scenarios be interpreted? How do they relate to one another? What is the underlying semantics? What assumptions are made when synthesizing behavior models from multiple scenarios? In this paper, we present an approach to scenario synthesis based on a clear sound semantics, which can support and integrate many of the existing approaches to scenario synthesis. The contributions of the paper are threefold. We first define an MSC language with sound abstract semantics in terms of labeled transition systems and parallel composition. The language integrates existing approaches based on scenario composition by using high-level MSCs (hMSCs) and those based on state identification by introducing explicit component state labeling. This combination allows stakeholders to break up scenario specifications into manageable parts and reuse scenarios using hMCSs; it also allows them to introduce additional domain-specific information and general assumptions explicitly into the scenario specification using state labels. Second, we provide a sound synthesis algorithm which translates scenarios into a behavioral specification in the form of Finite Sequential Processes. This specification can be analyzed with the Labeled Transition System Analyzer using model checking and animation. Finally, we demonstrate how many of the assumptions embedded in existing synthesis approaches can be made explicit and modeled in our approach. Thus, we provide the basis for a common approach to scenario-based specification, synthesis, and analysis.

Index Terms—Requirements specification, scenario-based specification, Message Sequence Charts, sequence chart combination, requirements analysis.

1 Introduction

The software engineering community has long understood the importance of requirements elicitation. Stakeholder involvement in the elicitation process and tools to help build a common ground between stakeholders and developers are essential to obtain a good requirements definition. Consequently, it is not surprising that scenarios have become increasingly popular as part of a requirements specification. Scenarios describe how system components (in the broadest sense) and users interact in order to provide system level functionality. Each scenario is a partial story which, when combined with other scenarios provides a more complete system description. Thus, stakeholders may develop descriptions independently, contributing their own view of the system to those of other stakeholders.

A widespread notation for scenarios is that of message sequence charts (MSCs) [22] and UML sequence diagrams [23]. These notations in their most basic form are highly intuitive and have a well-understood and widely accepted semantics. However, one scenario conveys relatively little information. Many scenarios are generally required to provide a significant system description. This makes scenario synthesis—the combination of a number of scenarios into a coherent whole—a central issue. How should a set of scenarios be interpreted? How do they relate to each other?

There are two ways of tackling this issue. One is to try to infer the relations between scenarios; the other is to require these relations to be explicitly stated by stakeholders. In the latter case, what abstractions should be provided to specify these relationships? Unsurprisingly, there are many different answers to this last question. For instance, the International Telecommunication Union (ITU) [22] introduces a graph-like notation that shows how the system evolves from one scenario to another. The underlying notion used by the ITU standard is that of scenario composition: New scenarios can be defined in terms of other scenarios by composing with sequential, choice, and iteration operators. In this way, complex system behavior can be described.

A different approach taken by Krüger et al. [28] provides state conditions instead of hMSCs. State conditions identify common states throughout different scenarios. Thus, two state conditions equally labeled on different scenarios indicate that the scenarios have a common point in the interactions they describe. This potentially allows the system to switch scenario when it reaches the common state.

There are a number of advantages and disadvantages between composition and state labeling approaches. On the one hand, composition mechanisms such as hMSCs promote scenario reuse, give a high-level view of the relation between scenarios and do not require stakeholders to specify scenarios with some kind of state machine in mind. However, scenario composition can lead to a large number of very short scenarios that must be composed in complex ways to describe the system's overall behavior.

• *The authors are with the Department of Computing, Imperial College, 180 Queen's Gate, London, SW7 2BZ, UK.*
E-mail: {su2, jk, jnm}@doc.ic.ac.uk.

Manuscript received Feb. 2002; revised July 2002; accepted Sept. 2002.
Recommended for acceptance by M.-J. Harrold.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 117521.

This mitigates the advantage of using scenario notations for depicting significant portions of system behavior. On the other hand, state identification approaches are a convenient way of introducing complex behaviors without having to split scenarios into parts. A key system run can be described in one scenario and then embellished with state information in order to relate it with other scenarios. In addition, state identification can be used, as we shall see, to introduce varied information on the system throughout a scenario specification and may provide means for progressively moving into a more detailed system description. On the negative side, requiring explicit identification of states requires much more consistency from stakeholders when constructing scenarios and forces them to reason about their system in terms of state rather than sequences of actions. *In this paper, the MSC language we define integrates approaches based on scenario composition and state identification.*

Scenario relations do not necessarily have to be given explicitly using hMSCs or state labels. Synthesis algorithms can be used to infer how scenarios are to be merged. These algorithms can include complex domain-specific and general assumptions of how scenarios are used and can sometimes incorporate additional information provided in other specifications. For example, Whittle and Schumann [50] use the Object Constraint Language (OCL) to express pre and postconditions for messages. These are traversed with scenarios to infer from the valuation of OCL predicates how scenarios are to be related. Using this information, a statechart model is constructed for each component of the scenario description.

If the assumptions implemented in the synthesis algorithm are appropriate, they help to simplify scenario notations and reduce stakeholders' workload. However, some drawbacks are that important explicit knowledge may be lost within the synthesis algorithms and that the consequences of the embedded assumptions can be obscured and produce misleading synthesis results. In addition, there is a significant loss of flexibility; if assumptions change, the complete synthesis procedure must be changed too. *In this paper, we show how the consequences of many assumptions on how to integrate scenarios can be described explicitly in the MSC language we propose. Thus, providing a common, intermediate representation for other approaches to scenario synthesis.*

Regardless of the way in which the relation between scenarios is defined, the purpose of a scenario specification is to describe how a system is intended to behave. Thus, analyzing the described system behavior should play a central role in the development of scenario-based specifications. To enable such analysis, synthesis algorithms build state-machine based behavior models. In addition to providing an alternative view, there is benefit to be gained by experimenting with [13] and replaying analysis results from behavior models in order to help correct, elaborate, and refine scenario-based specifications.

However, building a statechart model [14], for example, will not produce these benefits on its own. Synthesis procedures must integrate with existing behavior analysis technology to provide useful feedback to stakeholders. *In this paper, we describe an implementation of a synthesis algorithm, which is integrated into the Labeled Transition System Analyzer (LTSA)* [32], *which can be used to analyze the resulting behavior model by using its model checking and animation features.* Some of the benefits of this integration are illustrated in Section 4.2, where we use the model checking features of LTSA to detect a potential deadlocking scenario of a system described using MSCs.

Summarizing, in this paper, we aim to provide a common approach to scenario-based specification, synthesis, and behavior analysis. Our approach is based on a clear sound semantics, produces behavior models amenable to analysis, and can support and integrate many of the existing approaches to scenario synthesis.

We define an MSC language with sound abstract semantics in terms of labeled transition systems and parallel composition [32]. The language integrates scenario composition and state identification approaches by providing hMSCs and explicit component state labeling. This allows stakeholders to break up scenario specifications into manageable and reusable parts using hMCSs; while also having the possibility of using state labeling to avoid the need to break up scenarios into excessively small parts. Additionally, state labels support approaches that assume specific criteria for identifying component and system states, thereby providing a simple mechanism for making the effects of these assumptions explicit. In this way, we aim to support existing approaches such as [2], [6], [17], [22], [27], [28], [39], [40], [43], [50]. In particular, we demonstrate how this can be done for two different scenario synthesis algorithms.

Our synthesis algorithm, which is integrated into the Labeled Transition System Analyzer (LTSA) [32], translates a scenario specification into a Finite Sequential Processes (FSP) specification [32]. From the FSP specification, LTSA can build a composite behavior model in the form of a labeled transition system (LTS), which can then be analyzed using LTSA's model checking and animation features. This integration is important in terms of our more general objective, which is to facilitate the development of behavior models in conjunction with scenarios.

The paper is structured as follows: In Section 2, we present a short survey on scenarios, focusing mainly on message sequence charts, synthesis, semantics, and analysis. It also serves as a discussion justifying the work presented in this paper. In Section 3, we present the MSC language, giving both syntax and semantics. MSC specifications are defined as a set of basic MSCs and a high-level MSC. We also present a simple ATM system that is used throughout the paper to illustrate the presentation. We then introduce the synthesis algorithm for producing a behavioral model from a MSC specification (Section 4) and discuss the soundness of the algorithm (Section 5). In Section 6, we mention the some of the experiences—including an industrial case study—we have had using our approach. We comment on the implementation of the ideas presented in the paper in Section 7 and, in Section 8, we refer to two case studies we have performed to illustrate how our work can support other approaches to scenario synthesis. Section 9 includes a more specific discussion on related work than the survey. Conclusions and future directions of our work are given in Section 10.

2 Background

2.1 Scenarios and Message Sequence Charts

A scenario is a narrative description of how users, system components, and the environment interact in order to

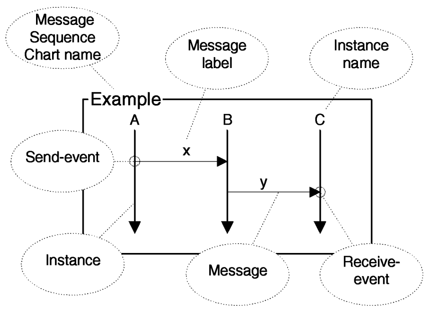


Fig. 1. Elements of a (basic) MSC.

achieve a (possibly implicit) goal (e.g., [23]). The need to document scenarios has motivated the development of many scenario-based notations. Although these languages include a variety of different features and have been designed for possibly different domains, there is a common core. *Message sequence charts* (MSCs) and their UML counterpart called *sequence diagrams* are a widespread graphical notation for documenting scenarios. In MSCs (see Fig. 1), system components, environment and users are represented as vertical arrows called *instances* (*lifelines* in UML). Instances model a system *components* participating in a scenario. We shall use component and instance interchangeably. Interactions between instances are shown as horizontal arrows called *messages*. The direction of a message indicates which instance initiates the interaction. *Messages labels* indicate the type of interaction that is occurring. The points on instances where an arrow starts and finishes are called (*send* and *receive*) *events*. An event can be considered the phenomena observed by an instance because of an interaction. MSCs are interpreted time-wise in a top-down fashion; an event on an instance occurs before all other events that appear below it on the same instance. Message direction also provides information on the order in which events occur; a send-event can never occur after its corresponding receive-event. For this reason, as a convention, messages are required to be drawn horizontally or with a downward slope. This is the generally accepted core of MSCs and, from here on, many variations abound. Messages can be considered to represent synchronous (handshaking) or asynchronous communication. Notation for and assumptions on queues that impose restrictions on event orderings can be included, (e.g., [2], [36]), as can explicit use of time to describe delays, timeouts, and deadlines (e.g., [22], [26], [31]); dynamic creation and termination of instances (e.g., [22]); reference to component and system states (e.g., [28]); and parametric message and use of data (e.g., [10]).

In this paper, we take the core aspects of scenario descriptions, components, and their interactions (considering them synchronous, handshaking communication for simplicity), and focus on the key issue of combining multiple scenarios into one system model.

2.2 Managing Multiple Scenarios

Scenarios are partial descriptions that are combined together to provide a more complete view of how a system is expected to behave. Typically, scenarios are provided by different stakeholders and address different system functionalities. The conjunction of all scenarios provides a system description. Choosing the right abstractions for combining scenarios is a critical issue. There are very distinct approaches to this problem; nevertheless, they can be explained in terms of three main concepts: scenario composition, state identification, and triggers.

2.2.1 Scenario Composition

In the approach adopted by the International Telecommunication Union (ITU) [22], the focus is on providing tools for managing complexity. Simple sequences of behavior are described using *Basic Message Sequence Charts* (bMSCs) (see Fig. 1). In addition, three fundamental constructs for combining bMSCs are provided: vertical and alternative composition and loops. Vertical composition of two bMSCs combines them sequentially. The system behavior is determined by the behavior of the scenario resulting from the syntactical concatenation of both bMSCs. Alternative composition defines a set of possible MSCs from which the system can choose which to follow. Loops compose a given bMSC sequentially with itself. The underlying notion of scenario composition is that scenarios can be used as building blocks to describe more complex behavior. They can be composed to define new, more complex, scenarios.

Several syntactic constructs, equivalent in terms of expressiveness [39], are provided by the ITU standard for specifying scenario composition: inline expressions, MSC reference expressions, and high-level MSCs, the last being the most widely adopted (e.g., [2], [39], [40]). *High-level Message Sequence Charts* (hMSCs) are directed graphs where each node references either an hMSC or a bMSC (for example, see Fig. 3, right). Edges indicate the acceptable ordering of scenarios, thus allowing stakeholders to reuse scenarios within a specification and to introduce sequences, loops, and alternatives of bMSCs. The advantage of the hMSC approach is that it allows stakeholders to break up a scenario specification into manageable parts in a simple, intuitive, and operational way, and to show how these different parts relate. On the other hand, as explained by Rudolph et al. [40], if alternative composition is the only mechanism of introducing branching system behavior, scenarios must typically be broken down into short bMSCs and composed to model the many alternative behaviors. Thus, obtaining a specification with very short scenarios that might be meaningless without the context of the hMSC. This plays against the intuitiveness of scenario notations that depict in one diagram a significant portion of system behavior.

2.2.2 State Identification

An approach that differs significantly from the scenario composition approach is the identification of common component or system states throughout a set of scenarios (e.g., [27], [28], [40], [43], [50]). The assumption here is that the scenario specification is describing a state-machine that models the behavior of system components. Thus, instances in a bMSC are considered to model both a set of states in the state-machine (which we call *component states*) and the events that fire state change (usually called *labeled transitions*). Thus, in the scenario specification, every space

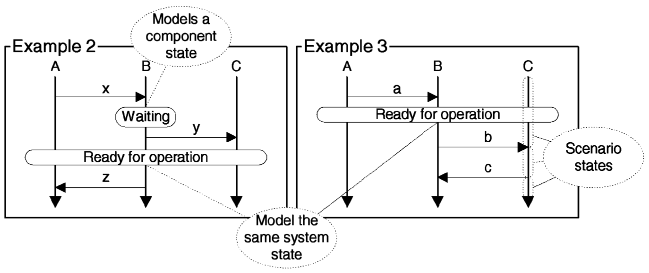


Fig. 2. Scenario, component, and system states.

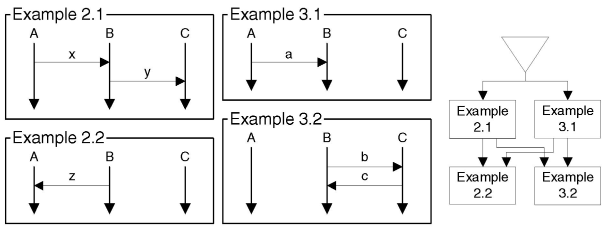


Fig. 3. bMSCs and hMSC.

between consecutive events is called a *scenario state* (see Fig. 2, right) and is considered to refer to *component state*. The relation between scenario and component states is many to one, meaning that several scenario states can refer to the same component state. Thus, scenario states in different bMSCs that refer to the same component state provide information on how the scenarios are related.

There are two basic mechanisms for identifying component states. The first is to allow stakeholders to tag scenario states (e.g., [28]). Typically, labels that describe the state of the component are placed on scenario states (see Fig. 2, left); if two states in a scenario appear with the same label, they are considered to refer to the same component state.

The second approach avoids the needs for explicit state labeling in scenarios and instead provides rules for identifying component states. These rules are usually based on domain-specific knowledge and additional information of the system being specified. For example, SCED [27] synthesises statecharts [14] while applying some assumptions in order to decide whether two scenario states represent the same statechart state. Another example is the work of Whittle and Schumann [50] which uses an Object Constraint Language (OCL) specification that states pre and postconditions for scenario messages. The OCL specification is traversed with the MSCs to produce a valuation of state variables for each scenario state. Scenario states that have equivalent valuations are considered to represent the same component states.

We shall be discussing modeling component states in MSC notations in more detail throughout the paper.

A variation of identifying component states is that of system states [40]. Instead of identifying component states on instances, labels that cover all instances of a scenario are used to mark a specific system state (see Fig. 2), which essentially models the state in which each component is in at a particular moment. Identically labeled system states refer to the same system state.

An advantage of explicitly labeling component or system states is that they can be used to enrich scenario descriptions with additional information. This information can come from specifications that have different system viewpoints or from domain-specific knowledge. In addition, incorporating component or system state information may provide means for progressively moving into a more detailed design description. Compared with scenario composition, identifying states allows complex component behavior including alternative behaviors to be described in bMSCs of any length. For example, in Fig. 2, we have two bMSCs with a system state labeled *Ready for operation*. This means that, if the system is in bMSC *Example 2* and messages x and y occur, then, instead of message z occurring, the system could now continue with b and c. Thus, the system state has introduced alternative behavior.

To introduce the same alternative using scenario composition, we would need to split the scenarios exactly where the "Ready for operation" state is and then introduce some composition mechanisms to establish all possible alternative behavior (see Fig. 3). Although the specified behavior is the same, we have had to split scenarios into smaller parts (some with only one message in them!) that may not be meaningful on their own.

The ITU provides a notation that resembles that of state labeling. Its MSC standard [22] includes the notion of local and global conditions, which could be interpreted as component and system states. However, ITU has not assigned any semantics to these syntactic elements. Several meanings to these constructs have been proposed. Component and system state identification is one of the possibilities being studied.

In UML [23], sequence diagrams allow introducing state information; however, it is not clear how this information affects the composition of different scenarios [16], [23], [38], [44]. To understand the relation between sequence diagrams, it is usually necessary to refer to the statechart descriptions.

2.2.3 Triggers

The third mechanism for combining scenarios is through the use of triggers or preconditions. Instead of relating scenarios to each other, information on when each scenario can occur is provided. This approach is popular in informal development methods [8], [38], [44], where scenarios are provided with a precondition normally stated in natural language. The precondition can refer informally to a state in which the scenario may occur or to a sequence of events that trigger the scenario. Other possibilities for describing preconditions are OCL or temporal logic, while scenario triggers can be specified using temporal logic or bMSC-like notations [15].

An advantage of using triggers is that scenarios are loosely coupled. In contrast with scenario composition methods, such as hMSCs, where the whole set of scenarios must fit together in one graph, triggers permit expression of the context of scenarios independently of existing scenarios. However, this same characteristic is a disadvantage in handling a scenario specification as a whole. This is especially acute when triggers are the only mechanism for specifying relations between scenarios.

In the work presented here, we define a scenario-based language that supports two main approaches to managing multiple scenarios, namely, those approaches based on scenario composition and on state identification. We do not support scenario triggers; although we do plan to look into this in future work.

2.2.4 *Overlapping Scenarios*

Many approaches assume that a scenario describes all the concurrent behavior of participating components at a given time (e.g., [27], [50]). This means that, for example, when the system is going through the scenario of Fig. 1, component B does not interact with other components between messages x and y. Other approaches allow further interactions on message types that do not appear in the scenario [15]. So, for example, component B, after receiving x in Fig. 1, may be allowed to receive a message a, but not another message x before it sends message y. Other approaches allow scenarios to be composed in parallel, meaning that scenarios with common participating components can occur simultaneously. For instance, the horizontal composition operator defined in the MSC standard [22] has been introduced for this purpose. Composition of scenarios overlapping in time introduces a series of complex issues such as events with the same label appearing in different scenarios: Do they represent the same event and, thus, the same moment in time, or are they different occurrences of the sending or receiving of a message of the same type?

In this paper, we do not consider composition of overlapping scenarios. However, this could form a future extension to our work.

2.3 Semantics

Independently of the mechanisms provided for managing multiple scenarios, there are also subtle yet significant differences in the semantics of scenario languages. These differences have important practical consequences.

2.3.1 *Semantic Definitions*

In terms of how semantics for scenario-based languages are defined, existing work is quite varied. We identify three broad categories: informal, algorithmic, and abstract semantics. The first category corresponds to scenario languages with no precise semantics that are used in the context of informal development methods (e.g., [8]) and in some UML-based development methods such as the Unified Software Development Process [23] and others (e.g., [16], [38], [44]). Although useful for documentation and informal analysis, the lack of a precise interpretation of scenarios makes rigorous analysis of scenarios and formal verification of system compliance to requirements extremely difficult.

The second category includes approaches in which the semantics of a scenario specification is implicit, given by means of a translation algorithm. Using an algorithm to translate scenario specifications into other notations can determine a precise interpretation if the target notation has a well-defined semantics. However, this procedure is rather operational and does not provide an intuitive and abstract meaning to scenario specifications. Subtle aspects of the semantics may be, and usually are, buried in the synthesis algorithm. Some approaches translate scenario specifications into statecharts (e.g., [27], [28], [50]). In these cases, it is important to distinguish between the several different interpretations of statecharts that exist [11]; however, this is not always the case. Furthermore, approaches that build individual statecharts for each component do not always explain how these statecharts are to be composed to provide the overall system behavior. We believe that this is crucial as many of the subtle design issues can lead to errors when concurrent interacting components are composed. Other work based on translation have focused on producing SDL specifications (e.g., [21]), hierarchical state machines (e.g., [6]), and other state machine based formalisms (e.g., [25]).

Finally, the third category includes work in providing a formal abstract semantics for scenario-based languages. The difference with the previous category is that the semantics is defined in an abstract manner rather than by a translation algorithm. Formalization work includes the use of process algebras [22], [39], partial orders [1], pomsets [24], büchi automata [29], and petri-nets [19]. In many cases, synthesis algorithms are also provided but not as a means for defining scenario semantics. In some cases, they are developed for producing the input of model checkers [20]. If so, algorithms must be shown to preserve the semantics of the specification.

2.3.2 *Design vs. Early Requirements Oriented Semantics*

Although the differences among approaches to scenario semantics can be considered a technical issue, it can strongly impact the role of scenario-based specification within the development process. Consequently, it also impacts on practical and research issues. There are two fundamentally different approaches to scenario semantics that we now discuss.

Many authors consider that scenario specifications describe high-level design of system components, i.e., that a set of scenarios directly determines a state machine for each system component. This is particularly true for most approaches that use synthesis algorithms as a means of providing scenario semantics, for those in which component or system state identification is considered or those approaches that use state machine semantics (for e.g., [15], [21], [22], [27], [28], [39], [40], [49], [50]). These approaches take on a design-oriented perspective, in which scenario descriptions are a design document in their own right. This design-oriented or late-requirements perspective tends to support the move from scenario-based notations to traditional design notations and techniques more easily.

A different interpretation of scenario specifications considers them to be describing system functionality not design. In other words, that a scenario specification determines a set of acceptable behaviors for which many designs can be found. This view is taken by some informal approaches to scenarios (e.g., [8]), as well as approaches using semantics based on partial ordering of events [1], [4], [24]. This perspective introduces the problem of finding a design (or possibly many) for a given scenario specification and also of proving that the design satisfies the requirements specification [1], [47]. Compared to the design-oriented approach discussed previously, this approach seems to be more suitable in an early requirements view.

We believe that both approaches to scenario semantics can be useful, and we are involved in conducting research in both. The work presented here corresponds to the design-oriented view of scenario-based descriptions. We define a semantics based on labeled transition systems (LTSs) and provide a synthesis algorithm for building such models using Finite State Processes (FSP) and the Labeled Transition System Analyzer (LTSA). Our work on the early requirements view of scenarios-based specifications can be found in [47], [48].

2.4 Analysis

Providing scenario-based languages with clear syntactic and semantic constructs for specifying requirements is a helpful first step. Substantial benefit can then be obtained from tools which support analysis of such specifications. There are a number of efforts in this direction, many of which focus on checking for syntactic consistency (e.g., MSC and POGA tools [2]). However, since scenario-based specifications describe inherently concurrent systems, analyzing syntactic correctness is not sufficient and analysis of the semantic implications of specifications is crucial. Many authors have focused on specific system properties and have produced ad hoc algorithms that can check their validity. Some examples of specific properties are *process divergence* [4] where, in an asynchronous communication setting, a component can flood another by sending it an unbounded number of messages that the receiver cannot process. *Nonlocal choice* [4] can result from alternatives in hMSCs. These choices can be under-specified if asynchronous communication is considered and the initial events of the different alternatives correspond to different components. Both process divergence and nonlocal choice are implemented in the MESA tool [4]. *Race conditions* [2] can also arise if assumed queuing mechanisms fail to enforce event orderings determined by scenarios. *Template matching* has also been developed as a mechanism for querying for behavior patterns in scenario-based specifications [35]. The TEMPLE tool implements template matching [20]. Semantic analysis based on additional model information is performed in many cases with specifically tailored integration methods, such as in [45], [50]. Scenarios can also be used to derive system performance models as in [3].

An attractive approach to scenario specification analysis is the use of standard model checking tools to verify properties of models. Thus, if scenario-based specifications are used to synthesise behavior models, properties such as process divergence can be detected using standard model checking techniques. In addition, the constructed behavior models can be used for general property analysis (such as deadlock) and for verifying domain specific, user defined properties. Some tools can directly generate the input for model checkers from scenario-based specifications (e.g., [20]). We have adopted this approach and implemented a synthesis algorithm integrated into an existing model checker. The MSC specification is translated into a Finite Sequential Processes (FSP) specification [32], which can then be analyzed using the Labeled Transition System Analyzer [32] (LTSA) by model checking for deadlock, safety, and liveness properties and by model animation.

3 Message Sequence Charts

In this section, we briefly describe the syntax and semantics of our *Message Sequence Charts* (MSCs) language, based on scenario composition, state identification, and labeled transition system (LTS) based semantics. An ATM example (see e.g., [41]) is introduced to illustrate the different aspects of our approach. This example has several scenarios showing how a customer operates a bank account through an ATM machine and a consortium. For the sake of brevity, we use a reduced set of scenarios.

3.1 Syntax

Syntactically (but not semantically, as explained later), the language is a subset of the MSC ITU language [22]. A *basic MSC* (bMSC) describes a finite interaction between a set of components (for example see Fig. 4, *Bad Bank Account*). Each vertical line represents a component and is called an *instance*. Each horizontal arrow represents a synchronous *message*, its source on one instance corresponds to a *message output,* and its target on a different instance corresponds to a *message input*. Ovals on instances represent *states*, where the label appearing within the oval identifies a particular component state. Some approaches (e.g., [22]) refer to state labels as conditions. Placing MSC events (message inputs, message outputs, or state labels) further down on an instance means that they occur later on. For simplicity, throughout the paper, we assume that message labels are types; that is, they are used consistently to identify only one outputting component and only one (different) inputting component.

Definition 1 (Instances). *An instance is a structure (E, L, <, lbl), where*

- $E = In \cup Out \cup Cond$ *is a set of MSC events that is partitioned into message inputs, message outputs, and states.*
- *L is a finite set state labels. We shall assume that the label* Init *denotes the initial component state.*
- $< \subseteq (E \times E)$ *is a total ordering of events. We denote the minimal event* e' *such that* $e < e'$ *as* $suc(e)$.
- $lbl\colon E \rightarrow L$ *is function that describes each event's label. We shall sometimes use* $lbl(A)$ *to denote the set* $\{lbl(e) \mid e \in A\}$.

Definition 2 (bMSCs). *A basic message sequence chart (bMSC) is a structure* $B = (I, tgt)$, *where* I *is a finite set of bMSC instances* $(E_j, L_j, <_j, lbl_j)$ *with* $0 < j < n$ *and disjoint sets of events* E_j *and tgt:* $\bigcup_{i=1}^{n} In_j \rightarrow \bigcup_{i=1}^{n} Out_j$ *is a bijective function that maps output and input events such that:*

- *If* $i \in In_j$, *then* $tgt(i) \notin Out_j$.
- *If* $tgt(i) = o$, $i \in In_j$, *and* $o \in Out_k$, *then* $lbl_j(i) = lbl_k(o)$.
- *If the transitive closure of* $(\bigcup_{i=1}^{n} <_i) \cup tgt \cup tgt^{-1}$ *contains* $\{(a,b),(b,a)\}$, *then* $tgt(a) = b$, $tgt(b) = a$, *or* $a = b$.

Note that the condition for the transitive closure defined above is to ensure that the bMSC is consistent in terms of time. In other words, no messages arrows cross each other. We use tgt and tgt^{-1} because messages are synchronous, and $<_i$ as it is the ordering of events over time on an

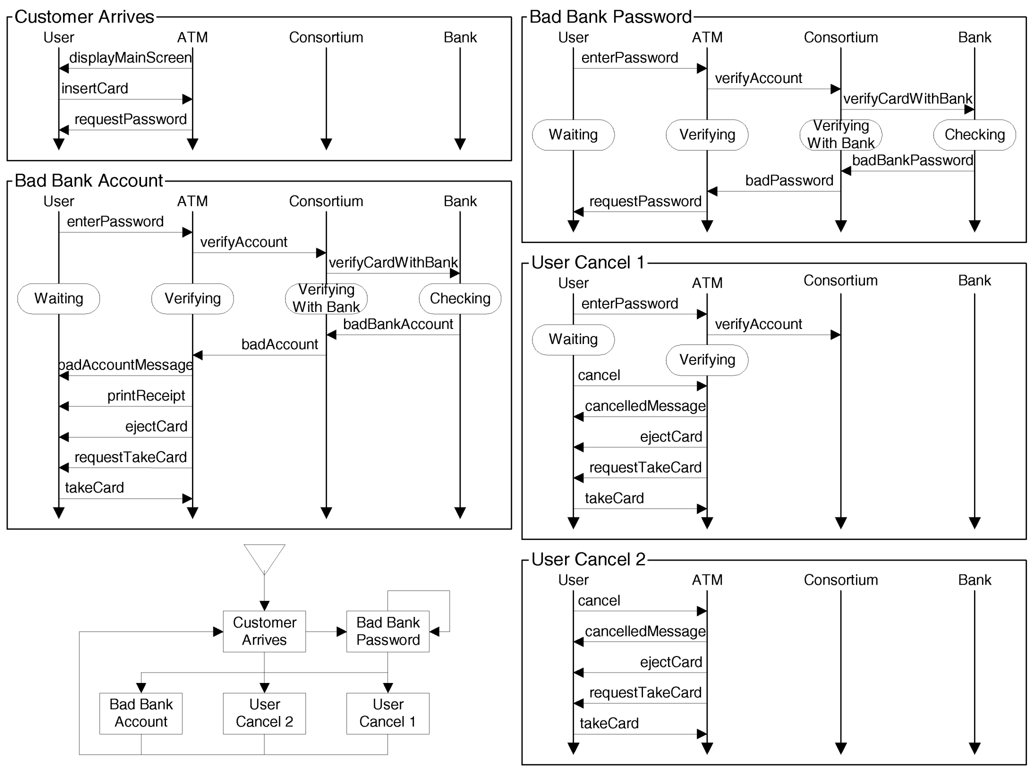


Fig. 4. MSC specification of the ATM system.

instance. Thus, if two events mutually precede each other, they must either be the same event or be the send and receive events of the same message.

A *high-level MSC* (hMSC) provides the means for composing bMSCs: It is a directed graph where nodes are bMSCs and edges indicate their possible continuations. An hMSC can have a special initial node that corresponds to the initial system states. We define hMSCs within the definition of MSC specifications and assume that message labels are used consistently as message types throughout all bMSCs of the specification. A portion of the MSC specification of the ATM example is shown in Fig. 4. It consists of five bMSCs and one hMSC.

Definition 3 (MSC Specification). *An MSC Specification is a structure S = (B, H, C, name), where*

- *B is a finite set of bMSCs* (I_j, tgt_j) *with disjoint sets of events.*
- $H = B \cup \{Init\} \rightarrow 2^B$ *is the hMSC function that determines the possible continuations of the bMSCs.*
- *C is a finite set of components.*
- name *is a family of bijective functions* $name_j$: $I_j \rightarrow C$ *that determines to which component each instance belongs.*

The ITU MSC standard [22] includes several more features. We have excluded some of them for simplicity, as they are variations for expressing scenario composition as the case, inline expressions, and bMSC references [39]. We have excluded other features such as coregions in order to simplify our presentation; however, they could be included without introducing substantial change to our results. We do not consider aspects such as timers, gates, process creation and termination, and incomplete messages, as we wish to focus on the key issue of combining multiple scenarios into one system model. As discussed in the previous section, we do not include horizontal composition either. Finally, we do not consider asynchronous messages and queues; these could however be modeled by using additional components for buffering and, hence, decoupling the message passing.

3.2 Semantics

The semantic of the MSC language is not a subset of the ITU MSC language. There are two major differences. First, the ITU semantics does not provide any meaning to state labels (called conditions by the ITU). Second, the ITU version introduces delayed choice, which assumes that components can delay the choice of which scenario to follow in a bMSC in order to prevent deadlocking situations. We do not make this assumption and, rather, require delayed choice to be modeled explicitly where required using state labels.

We define the semantics of MSC specifications in terms of labeled transition systems (LTSs) and parallel composition [34]. In other words, we take a design-oriented view in which a MSC is considered to define a system resulting from the parallel composition of LTSs, one for each component.

Definition 4 (Labeled Transition Systems). *A finite labeled transition system (LTS) P is a structure (S, L, Δ, q), where:*

- *S is a set states.*
- $L = \propto(P) \cup \{\tau\}$ *is a set of labels where* $\propto(P)$ *denotes the alphabet of P and* τ *denotes internal actions that cannot be observed by the environment of an LTS.*

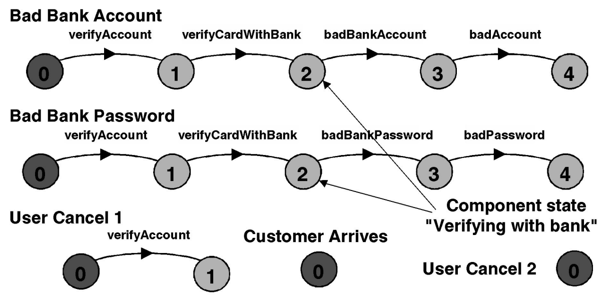


Fig. 5. Instance LTSs of consortium.

- $\Delta \subseteq (S \times L \times S)$. *We use* $s \xrightarrow{a} s'$ *to denote* $(s, a, s') \in \Delta$, *and* $s \xrightarrow{w} s'$ *to denote that there are* l_i *and* s_i, *such that* $w = l_1 \ldots l_n$, $s_n = s'$, *and* $s \xrightarrow{l_1} s_1 \cdots \xrightarrow{l_n} s_n$.
- $q \in S$ *is the initial state.*

Given two LTSs P and Q, we denote $P||Q$ the LTS that models their joint behavior. The joint behavior is the result of both LTSs executing asynchronously but synchronizing on all shared message labels. This means that any of the two LTSs can perform a transition independently of the other LTS, so long as the transition label is not shared with the alphabet of the other LTS. Shared *observable* labels have to be performed simultaneously. Observability of labels signifies that the LTSs never synchronize on τ since they represent internal behavior.

We first define the semantics of an instance, then that of components, and finally we define the system that is determined by an MSC specification.

There are the two types of information that an MSC specification provides: sequences of message inputs and outputs, and information on states. Information on sequences of messages is provided by instances. For example, reading from the top to bottom, it is natural to say that, in the *badBankAccount* bMSC of Fig. 4, the Consortium inputs a *verifyAccount* message, then after sending *verifyCardWithBank* and receiving a *badBankAccount* message, it forwards the message *badAccount* to the ATM. If each of these events is considered instantaneous, the instance can be viewed as a labeled transition system (LTS) as shown in the Bad Bank Account instance LTS of Fig. 5. In addition, note that state 2 corresponds to the *Verifying With Bank* state label of the ATM.

To simplify the presentation of semantics, we shall assume normalized instances. A normalized instance is an instance in which there are no two consecutive states and no two consecutive message events, i.e., for all events e, e' such that $e' = suc(e)$, then ($e \in Cond$ and $e' \in \{In \cup Out\}$) or ($e \in \{In \cup Out\}$ and $e' \in Cond$). In addition, a normalized instance has states as the first and last events, i.e., if e is the minimal or maximal event in E, then $e \in Cond$. Normalized instances have events of a special kind, τ-events, that represent internal changes of a component's state. Normalizing an instance is done by using τ-events to separate consecutive states and states labeled with ε to separate consecutive message events.

Definition 5 (Instance LTS of a Component). *Let I =(E, L, <, lbl) be a normalized instance. The instance LTS of I is a labeled transition system (S, A, Δ, Start), where*

- *S = Cond.*
- $A = lbl(In) \cup lbl(Out)$.
- Start *is the minimal event in E.*
- $\Delta \subseteq (S \times A \times S)$ *is the transition relation where* $(q, a, q') \in \Delta$ *if and only if there is a message event* $e \in E$ *such that* $suc(q) = e$, $suc(e) = q'$, *and* $lbl(e) = a$.

We shall refer to the maximal state in E as Stop.

Note that the semantics abstracts away the fact that events are outputs and inputs. A naming convention can be introduced to differentiate in the LTS send-events from receive-events. This, however, would then require some renaming when components are composed in parallel, in order to force synchronization between correspond send- and receive-events.

State labels and hMSCs provide information on states of instance LTSs. State labels identify component states indicating that, although they appear as distinct in instances, they are actually the same internal component state. For example, there are three different bMSCs in Fig. 4, where the ATM reaches a state called *Verifying*. In terms of component behavior, this means that the ATM could "switch," between bMSCs when arriving at that state. hMSCs provide information on how components can continue once they have completed a bMSC. In other words, they determine a relation between the Start and Stop states of instance LTSs. For example, according to the hMSC in Fig. 4, the states in which components are once the *Customer Arrives* bMSC concludes can continue as the initial states of the bMSCs *Bad Bank Password*, *Bad Bank Account*, *User Cancel 1*, and *User Cancel 2*. Thus, given an MSC specification, using state labeling and hMSCs, it is possible to define a continuation relation between the instance states of a component plus component initial and final states as follows:

Definition 6 (Continuation Relation). *Let (B, H, C, name) be an MSC specification and let* $(S_j, A_j, \Delta_j, Start_j)$ *with* $0 < j < n$ *be the instance LTSs of component* $c \in C$. *If q and q′ are states in* S_j *and* S_k, *the continuation relation of c is* $R \subseteq S \times S$, *where*

- $S = \bigcup_{i=1}^{n} S_j \cup \{Init\}$.
- $(q, q) \in R$.
- *If* $lbl(q) = lbl(q')$ *and* $lbl(q) \neq \varepsilon$, *then* $(q, q') \in R$ *and* $(q', q) \in R$.
- *If* $B_k \in H(B_j)$, *then* $(Stop_j, Start_k) \in R$.
- *If* $B_k \in H(Init)$, *then* $(Init, Start_k) \in R$.
- *If* $lbl(q) = Init$, *then* $(Init, q) \in R$.

In conclusion, components defined by an MSC specification are the result of putting together their instance LTSs and their continuation relation.

Definition 7 (Component LTS). *Let (B, H, C, name) be an MSC specification, let* $(S_j, A_j, \Delta_j, Start_j)$ *with* $0 < j < n$ *be the instance LTSs of component* $c \in C$, *and* R^+ *is the transitive closure of the continuation relation of c. The component LTS of c is a labeled transition system* $(S, A, \Delta, Init)$, *where*

- $S = \bigcup_{i=1}^{n} S_j \cup \{Init, End\}$.
- $A = \bigcup_{i=1}^{n} A_j$.
- $(s, a, s') \in \Delta$ *if and only if* $(q, a, q') \in \Delta_j$ *for some j,* $(s, q) \in R^+$, $(q', s') \in R^+$.

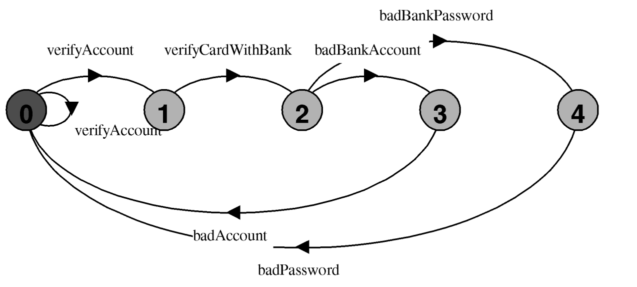


Fig. 6. Minimized component LTS for consortium.

The resulting component LTS may have a large number of states and transitions. In the case of the Consortium component, we have 15 states and over 20 transitions. In order to simplify and improve comprehension of these models, it is possible to find an equivalent LTS that has a minimal number of states. The equivalence relation we use the standard observational equivalence [34].

In Fig. 6, we show the LTS that has the least number of states and that is observationally equivalent to the Component LTS model for the Consortium component. Note that state 2 corresponds to the state modeled by label *Verifying with Bank*. The states with this label have been merged into one unique state. Had we not used the state label *Verify with Bank* the resulting model for the consortium component would have had a nondeterministic choice on state 1: Two different transitions labeled *verifyCardWithBank* enabled, leading to the two different states.

Note that the LTS semantics defines a model that captures the state *Verify with Bank* but not its label. The semantics could be changed to include such labels; however, the loss of the state label does not impact on the behavior of the overall system: The observable behavior of the system remains the same.

Finally, the semantics of an MSC specification is the parallel composition of its components.

Definition 8 (System LTS). *Let (B, H, C, name) be an MSC specification $C = \{c_1, \ldots, c_n\}$ and C_i the component LTS of c_i. The system LTS defined by the MSC specification is the parallel composition of all component LTSs: $(C_1 || \ldots || C_n)$.*

Note that the introduction of bounded asynchronous communication does not require a major change to the semantics: port abstractions could be introduced into the final parallel composition as explained in [32].

4 Synthesis of Behavior Models

In this section, we show how an LTS model can be synthesised from an MSC specification. We translate the MSC specification into a model specification in the form of FSP [32], a simple process algebra with precise LTS semantics that provides a concise way of describing LTSs. Additionally, we use the Labeled Transition System Analyzer (LTSA) [32] to build a composite LTS model from our FSP specification. LTSA is a verification tool for concurrent systems. It mechanically builds LTS models from FSP specifications and can be used to check that the specification of a concurrent system satisfies the behavioral properties required. In addition, LTSA supports specification animation to facilitate interactive exploration of system behavior.

```
void Synthesise(MSC S, String Component) {
    SetOfProduction P;
    Relation R;
    SetOfInstance I;

    labelInstances(S, Component);              //Step 1
    R = getContinuationRelation(S, Component);
                                               //Step 2
    Split(I);                                  //Step 3
    I = Derive(I, R);                          //Step 4
    P = MergeAndBuildFSPProductions(I);        //Step 5
    outputFSP(Component, P);
}

labelInstances(MSC S, String Component) {
    ForEach bMSC b in S {
        Instance I = b.getInstance(Component);
        if (I.firstState().unlabelled())
            I.labelFirstState(“B_” + b.name());
        if (I.lastState().unlabelled())
            I.labelLastState(“E_” + b.name());
    }
}


SetOfInstance Derive(SetOfInstance Instances, Relation R) {
    SetOfInstance D;
    ForEach Instance I in Instances {
        ForEach State s1 in R.Domain(I.FirstState())
            ForEach State s in R.Codomain(I.LastState())
                Instance Aux = I
                Aux.ReplaceFirst(s1);
                Aux.ReplaceLast(s2);
                D.add(Aux);
            }
        }
    }
    return D;
}
```

Fig. 7. FSP synthesis algorithm.

4.1 Synthesis Algorithm

The synthesis algorithm builds an FSP process for a given component according to a MSC specification. It consists of five steps and is outlined in Fig. 7. We provide a detailed explanation while applying each step to the synthesis of the ATM component of Fig. 4.

The general idea of the algorithm is to build local FSP processes that correspond to portions of component behavior and to combine them in such a way as to provide the complete component behavior. Each local FSP process corresponds to the behavior described on a component instance between labeled states, the top and bottom of the instance. The local process names can then be used when specifying how the local processes are combined according to the MSC specification.

The first step of the algorithm is to add labeled states at the top and bottom of all the ATM instances that do not already have such states. We use the convention *B_<bMSC.Name>* to label the top (Begin) state of an instance and *E_<bMSC.Name>* for the bottom (End) state. For example, the ATM instance corresponding to the *Bad Bank Account* bMSC is shown in Fig. 8.

The algorithm then constructs a relation on the set of state labels. The relation only includes labels corresponding to top and bottom instance states and an additional label representing the initial component state: *Init*. The construction of such a relation is straightforward; an edge in the hMSC defines a pair in the relation. Fig. 9 shows the resulting relation for the ATM component. The relation constructed by the algorithm represents a subset of the Continuation relation described in Section 3.

Third, the algorithm then works on the component instances. Each instance is split into several subinstances in such a way that all subinstances start and end with labeled states and have no other labeled states in between. Fig. 10

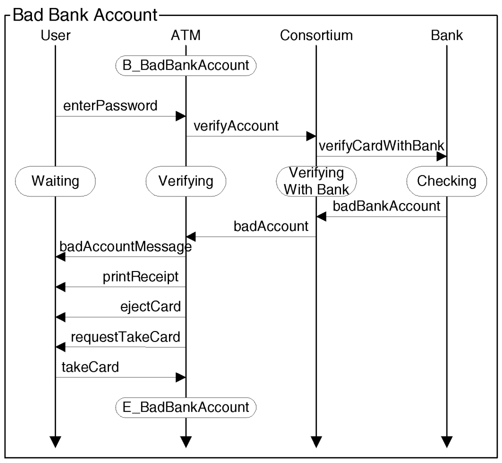
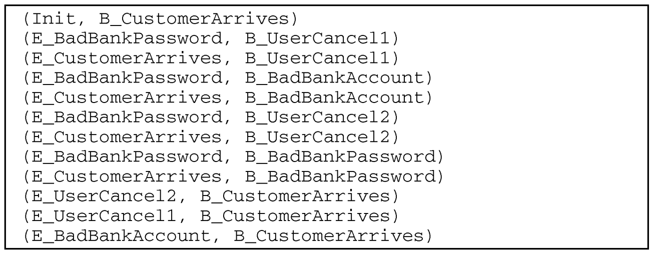


Fig. 8. Annotated ATM instance.

shows the two subinstances that result from breaking up the ATM instance of the *Bad Bank Account* bMSC.

The fourth step is to combine the continuation relation with the instances. The leftmost instance of Fig. 10 shows how the ATM component being in state *B_BadBankAccount* can end in state *Verifying* if events *enterPassword* and *verifyAccount* occur. From Fig. 9, we also know that the pair (*E_CustomerArrives, B_BadBankAccount*) is in the ATM's continuation relation. This means that the state *E_CustomerArrives* can continue as state *B_BadBankAccount.* Thus, if the ATM is in state *E_CustomerArrives* and events *enterPassword* and *verifyAccount* occur, the ATM must also reach *Verifying* state. The same reasoning can be applied to the rightmost instance of Fig. 10, where *B_CustomerArrives* can replace *E_BadBankAccount*. These new derived instances are shown in Fig. 11. The *Derive* method follows this reasoning and for each instance it creates a set of derived instances by replacing initial and final states according to the continuation relation.

In the fifth step, every instance is trivially translated into a local FSP process by using the label of its first state as the process name and the sequence of events as the process behavior. The final state of the local process must be another local process; we use the label of the instance's last state. Thus, in the case of the leftmost instance of Fig. 11, the resulting FSP definition is: `E_CustomerArrives = (enterPassword -> verifyAccount -> Verifying)`. Note that the "->" operator is the action prefix operator that defines a local process as the occurrence of an action (starting with lower-case label) and another local process (starting with upper-case label).



```
(Init, B_CustomerArrives)
(E_BadBankPassword, B_UserCancel1)
(E_CustomerArrives, B_UserCancel1)
(E_BadBankPassword, B_BadBankAccount)
(E_CustomerArrives, B_BadBankAccount)
(E_BadBankPassword, B_UserCancel2)
(E_CustomerArrives, B_UserCancel2)
(E_BadBankPassword, B_BadBankPassword)
(E_CustomerArrives, B_BadBankPassword)
(E_UserCancel2, B_CustomerArrives)
(E_UserCancel1, B_CustomerArrives)
(E_BadBankAccount, B_CustomerArrives)
```

Fig. 9. R for the ATM component.

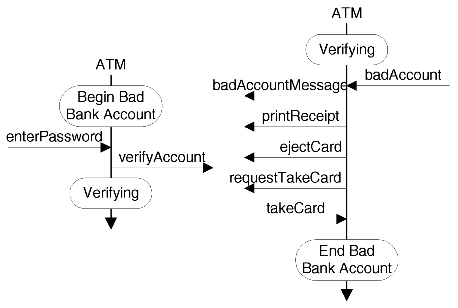


Fig. 10. ATM instance split in two.

We now have a series of local processes that can be put together to form the component behavior. However, multiple definitions of local processes must first be handled. For instance, due to the fact that the CustomerArrives bMSC has two continuations, we obtain two definitions of local process `E_CustomerArrives`. These definitions are combined using the choice operator (|), meaning that the component, having finished the bMSC CustomerArrives, has two different possible behaviors. The final definition for the local process is:

```
E_CustomerArrives =
 (cancel -> canceledMessage -> ejectCard ->
     requestTakeCard -> takeCard -> Init
 |enterPassword -> verifyAccount -> Verifying)
```

The resulting set of local processes together with the definition of the ATM component as the local process Init define the ATM behavior. However, before outputting the final FSP code, we apply some simple procedures for eliminating local processes with identical behavior. In addition, we utilize the LTSA keyword `minimize` in order to obtain the minimal observationally equivalent LTS to the behavior described by the FSP process. The final FSP specification and actual output of our implementation is shown in Fig. 12. The FSP code for the ATM component is at the top. Note that the complete system (appearing at the end of Fig. 12) is the parallel composition of all FSP components:

```
||System = (ATM || Consortium || Bank || User).
```

4.2 Analysis

Once an FSP specification has been generated, LTSA can generate an LTS model of each component and of the complete system. Furthermore, LTSA can minimize models

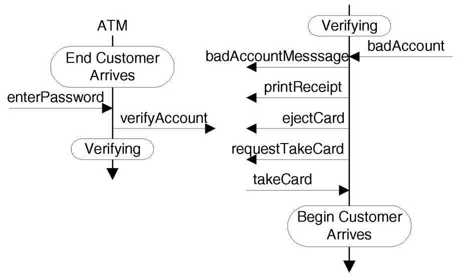


Fig. 11. Derived instances from Fig. 10.

```
minimal User = Init,
E_BadBankPassword =   (cancel -> canceledMessage -> ejectCard
                          -> requestTakeCard -> takeCard -> Init
                      |enterPassword -> Waiting),
Init =      (displayMainScreen -> insertCard -> requestPassword ->
                    E_BadBankPassword),
Waiting =   (cancel -> canceledMessage -> ejectCard ->
                 requestTakeCard -> takeCard -> Init
            |requestPassword -> E_BadBankPassword
            |badAccountMessage -> printReceipt -> ejectCard
                -> requestTakeCard -> takeCard -> Init).

minimal ATM = Init,
Init =      (displayMainScreen -> insertCard -> requestPassword ->
                    E_BadBankPassword),
E_BadBankPassword =   (cancel -> canceledMessage -> ejectCard ->
                           requestTakeCard -> takeCard -> Init
                      |enterPassword -> verifyAccount -> Verifying),
Verifying =   (badPassword -> requestPassword -> E_BadBankPassword
              |cancel -> canceledMessage -> ejectCard ->
                  requestTakeCard -> takeCard -> Init
              |badAccount -> badAccountMessage -> printReceipt ->
                  ejectCard-> requestTakeCard -> takeCard -> Init).

minimal Consortium = Init,
Init =  (verifyAccount -> verifyCardWithBank -> VerifyingWithBank
        |verifyAccount -> Init),
VerifyingWithBank =   (badBankPassword -> badPassword -> Init
                      |badBankAccount -> badAccount -> Init).

minimal Bank = Init,
Init =  (verifyCardWithBank -> Checking),
Checking =  (badBankAccount -> Init
            |badBankPassword -> Init).

||System = (Consortium || ATM || Bank || User).
```

Fig. 12. FSP specification for an ATM component.

with respect to observational equivalence (defined previously), which provides a more compact model and potentially clearer insight into its behavior. One problem is that the number of system states grows exponentially with respect to the number of components [9]. For the small example used in this paper, the number of states of the minimised system LTS is 43. Graphical representation of large systems does not favour rigorous analysis, even if hierarchical states such as those used in statecharts are introduced. Tools such as LTSA provide mechanisms for automated, rigorous analysis of behavior models.

In particular, as the synthesis algorithm preserves the semantics of the MSC specification, the synthesized model can be analyzed to provide sound and useful feedback to those who wrote the MSC specification. An immediate result when the complete ATM example is analyzed is that the system may deadlock. In Fig. 13, we show LTSA output of a trace that takes the system to deadlock: If the User cancels just after entering a password but before receiving an answer, the ATM, which has requested the account to be verified, does not wait for the answer from the Consortium. Eventually, when the ATM serves the User again, it cannot communicate with the Consortium as the latter is still trying to communicate the results of verifying the previous account.

Note that the deadlock detection mechanism works independently of the semantics of the message labels. LTSA applies model checking algorithms that traverse the system LTS verifying specific user defined properties or standard properties such as deadlock freedom. In the case of the latter, the state space of the LTS is traversed to find a state with no outgoing transitions.

This section has illustrated the kind of benefits that can be obtained by combining scenario-based notations such as MSCs with behavior models. By doing so, it is possible to provide scenario-developers with feedback that will help them to correct and elaborate their specifications.

```
Progress violation for actions: . . .
Trace to terminal set of states:
        displayMainScreen
        insertCard
        requestPassword
        enterPassword
        verifyAccount
        cancel
        canceledMessage
        ejectCard
        requestTakeCard
        takeCard
        displayMainScreen
        insertCard
        requestPassword
        enterPassword
        verifyCardWithBank
        badBankPassword
Actions in terminal set:
        {}
```

Fig. 13. LTSA output with deadlocking trace.

5 Experience

In order to validate our approach, we have experimented with several variations of scenario-based specifications that are available in the literature. These include a more complete ATM system, an alarm clock [33], an elevator system [12], a gas station [42], a journal editing process [51], a parcel router [32], a rail-car control system [15], and a boiler control system [47].

In addition, we have conducted one medium-sized industrial case study. The case study involved specifying the behavior of instances of a software architectural style being developed at Philips. The architectural style involves a horizontal communication protocol [37] that supports building product families of television sets. We first constructed a scenario specification for the simplest instance of the architectural style, and then tackled two more complex instances. Fortunately, we were able to validate the synthesized LTS models obtained from our scenario specifications against a separate behavior model, developed independently by Rob van Ommering and Jeff Magee. From the case study, we have been able to increase our confidence in scenarios-based specifications as a means for specifying and constructing behavior models of concurrent systems.

6 Implementation

The synthesis algorithm presented in Section 4 has been implemented in Java and integrated with the LTSA tool we have mentioned in previous sections. It inputs MSC specifications in textual format [22] and outputs an FSP specification, which can then be processed by LTSA. The implementation, together with the examples used throughout this paper, is available at [46].

Table 1 gives the execution times for the examples presented in this paper and other case studies we have conducted. All examples were run on a Pentium III, 300Mhz, 256Mb with Windows NT 4.0, and Java 1.3. This implementation of the synthesis algorithm was developed fundamentally as a proof of concept rather than to build an efficient implementation. Consequently, the execution times of our naïve implementation could be made more efficient. The implementation builds component FSP processes one at a time. The complexity of this algorithm resides in the

TABLE 1
Synthesis Algorithm Execution Times

	bMSCs	Edges in hMSC	Labelled States	LTS States	Time (ms)
ATM Section 3	5	11	4	10	181
ATM Section 8.1	5	15	21	125	200
ATM Section 8.2	5	4	33	98	171
Complete ATM	14	35	22	146	541
Alarm Clock	6	10	21	15	140
Gas Station	12	21	0	63	2564
Elevator System	8	12	42	190	371
Journal Editing Process	15	36	0	51	4276
Railcar System	11	28	0	475	300
Industrial Case Study	18	33	88	658	24191

number of FSP productions that must be built. Given B bMSCs and S labeled states of the component being synthesized, (B+S) split bMSCs are constructed. In addition, each split bMSC can have at most the (B+S) continuations and be the continuation of at most another (B+S) split bMSCs. Thus, the number of FSP productions that are initially built is bounded by $(B+S)^3$. This theoretical bound could be reduced significantly in a nonnaïve implementation by detecting equivalent FSP productions on the fly rather than implementing clean-up procedures such as the ones we have implemented.

7 Support for Other Approaches to Synthesis

So far, we have presented an MSC specification language that integrates approaches based on hMSCs and on identifying component states. We have also provided a synthesis algorithm that generates LTS behavior models. In this section, we illustrate how this approach can be used to support different approaches to behavior model synthesis.

Many synthesis algorithms have been proposed and, although they agree on the basic interpretation of MSCs, they differ greatly in terms of the algorithms and the results obtained. This is because these approaches embed assumptions in their synthesis algorithms. These can be domain-specific assumptions, assumptions on how to include additional information provided in alternative specification languages, or other assumptions based on, for example, characteristics of the stakeholders, organization, or development process. Synthesizing behavior models with certain assumptions in mind is not a problem; however, having assumptions embedded in the algorithms results in less flexibility and loss of explicit knowledge. States in MSC specifications can be used to make the explicit the effect that these assumptions have on the semantics of the scenario specification.

The purpose of this section is to demonstrate through two different cases, how our approach can be used to support synthesis approaches with different kinds of assumptions. By supporting these approaches, we are providing an intermediate step between the original specification with which these approaches start from and the final state machine generated by the synthesis algorithms. This intermediate step is a new MSC specification using syntax and semantics described above, which shows how states are to be merged and where to loops are being introduced. In addition, by mapping these approaches onto our language, we are providing a sound, common synthesis procedure and the possibility of rigorous analysis using LTSA.

We now discuss in detail two approaches we have studied.

7.1 Case 1

Whittle and Schumann [50] present an algorithm for automatically generating UML statecharts from the combination of scenarios with a set of message pre- and postconditions given in UML Object Constraint Language (OCL). In fact, a LTS is first synthesized and then some abstraction techniques used to build statecharts. We describe this approach to synthesis in detail using the same example given by authors in [50].

Suppose we start with a set of bMSCs that describe the ATM system and that differ with the four bMSCs of Fig. 4 in that no states—*Waiting, Verifying, Verifying With Bank,* and *Checking*—have been labeled ([50] does not support component state labeling nor hMSCs).[1] In addition, suppose that we have some domain knowledge regarding the ATM component that has been specified in OCL (see Fig. 14). The OCL describes a set of ATM state variables—*cardIn, cardHalfway, passwdGiven, card*, and *passwd*—together with pre and postconditions for ATM interactions. By finding messages in scenarios for which we have pre and postcondition information, it is possible to infer the value that OCL state variables have at specific points in the scenario.

For example, in bMSC *Bad Bank Account*, the first message is *Display Main Screen.* In addition, the OCL specification states a precondition for such a message, thus we know that the value of cardIn and cardHalfway is false at the beginning of bMSC *Bad Bank Account*. By taking into account the available OCL for all components and using the unification and frame action techniques defined in [50], it is possible to infer further information on the value of state variables throughout the available scenarios.

Consequently, it is possible to assign a (possibly partial) valuation of state variables to every state of a bMSC. In [50], these valuations are used in two different ways. First, within an instance, two states with the same valuation are considered to determine a loop. However, as not all message occurrences provoke a change in the state variable values, it is also required that valuations that determine loops be the result of state-changing messages. Second, between two instances of the same component, two states that follow messages with the same label and that have the same valuation are considered to refer to the same state.

According to the criteria described above and the assumption that stakeholders describe system behavior from its initial state, the authors provide an algorithm for building a LTS model from a MSC and OCL specification. The algorithm, of course, has been specifically tailored for the criteria presented above and produces an LTS model

1. We also change message labels to concord with the authors naming conventions.

```
cardIn, cardHalfway, passwdGiven : Boolean
card : Card
passwd : Sequence

Insert card(c : Card)
pre : cardIn = false
post: cardIn = true and card = c

Enter password(p : Sequence)
pre : passwdGiven = false and
              p->forAll(p->includes(d)=>digit(d))
post: passwdGiven = true and passwd = p

Take card()
pre : cardHalfway = true
post: cardHalfway = false and cardIn = false
Display main screen()
pre: cardIn = false and cardIn = false
post:

Request password()
pre : passwdGiven = false
post:

Eject card()
pre : cardHalfway = true
post: cardIn = false and cardHalfway = false
              and card = null and passwd = null
              and passwdGiven = false

Request take card()
pre : cardHalfway = true
post:

Canceled message()
pre : cardIn = true
post:
```

Fig. 14. OCL for the ATM component.

directly from the specifications. Information on valuations and loops is not visible to the scenario provider.

We now show how this approach can be used in our setting, making all assumptions explicit. This is achieved by using our MSC language as an intermediate representation of the criteria of [50]. There are three different issues and we address each of them separately: loops, references to same states, and behaviors starting at initial system state.

First of all, we construct an hMSC to make explicit the assumption that scenarios describe system behavior from the very start of it. This can be done trivially and is shown in Fig. 16.

Second, after valuations have been inferred (using techniques described in [50]), all states in bMSCs are labeled with two pieces of information: the message label that precedes the bMSC state and the valuation that corresponds to the state. By doing so, we identify component states using the same criteria as Whittle and Schumann. In Fig. 15, we show bMSC *Bad Bank Account* before adding state labels and part of the same bMSC annotated as explained above. Valuations are shown as vectors where each position represents the value of a variable in the OCL specification in the following order: <*cardIn, cardHalfway, passwdGiven, card, passwd*>. We have coded the valuations with letters A to D (see reference in Fig. 15) and used message label initials to make the annotated bMSC easier to read.

Finally, every loop, detected using the criteria provided in [50], is used to split bMSCs into three: the initial part that occurs before the loop, the looping part, and the part that occurs after the loop is exited. The hMSC is modified to reflect the relation between the new bMSCs resulting in an hMSC as in Fig. 17.

The final MSC specification contains all relevant information derived from the authors' criteria and all assumptions have been made explicit through a process that can be automated. The final step is to apply the synthesis algorithm defined in previous sections to obtain an LTS model that is equivalent to that obtained in [50]. The ATM synthesized from the annotated scenarios using the algorithm presented in this paper models the same behavior as the statechart built using the Whittle and Schumann approach (Fig. 18).

In conclusion, we have shown that both the ideas and algorithms developed in [50] can be complemented with our approach giving the benefit of a potentially standardized synthesis algorithm and an intermediate step in which all information and assumptions are in one specification.

7.2 Case 2

Koskimies et al. [27] have developed a tool for building statechart models of components from scenario-based specifications. The actual synthesis algorithm is based on

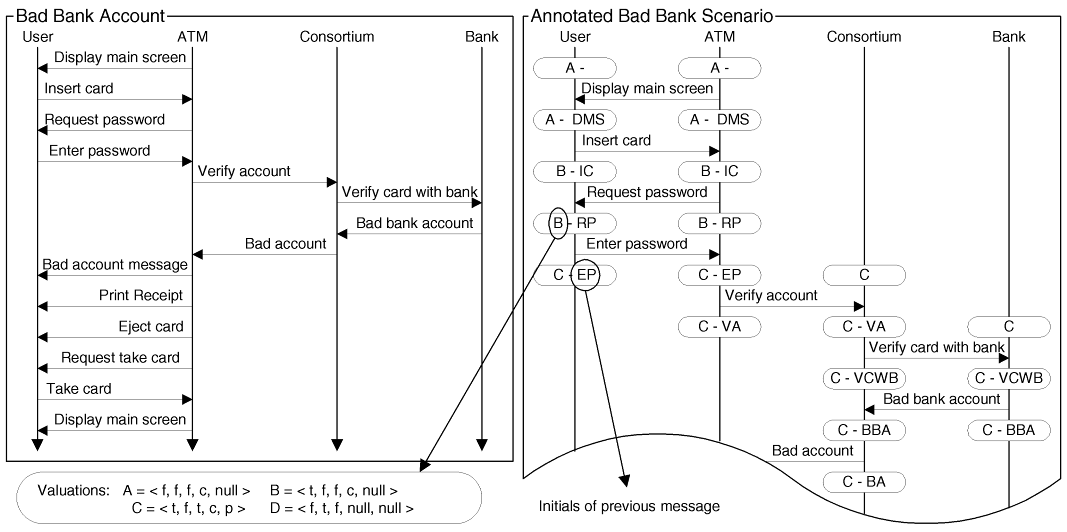


Fig. 15. Original and annotated bMSCs.

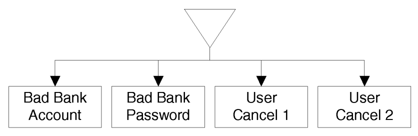


Fig. 16. Explicit specification of initial system state.

the BK-algorithm [5] for constructing programs from example computations. In the synthesized statecharts, states are labeled with component actions (message outputs) and transitions with sequences of events (message inputs).

The main assumption of the approach is that all states with the same action correspond, in principle, to the same component state. The underlying rationale for this approach is that scenarios describe components in which the capability of outputting a particular message uniquely identifies its state. However, if states introduce nondeterminism when merged, then they must remain separate; this is done to avoid overgeneralizations.

We now show how the assumption can be made explicit in an MSC specification using state labels. However, one aspect we cannot model in our setting is that in [27], the resulting statecharts have no initial state. We cannot reproduce this in our setting because LTSs must have an initial state.

Suppose we start with a set of bMSCs that describe the ATM system and that differ with the four bMSCs of Fig. 4 in that no states have been labeled ([27] does not support component state labeling nor hMSCs). We first introduce an initial system state through an hMSC as shown in Fig. 16. In addition, to implement the criteria for identifying states, we add two sets of state labels. Essentially, on every instance, when a message is output, we introduce a state label *B_<Message Label>* just before the output and a state label *E_<Message Label>* just after. This guarantees two unique states for every output message in the resulting LTS. These pairs of states model statechart states, one for the entrance of the component to the statechart state and the other for its exit. The transition between both states models the action being performed by the component when in the statechart state. In Fig. 19, we show the bMSC *Bad Bank Account* annotated with labels *B_<Message Label>* and *E_<Message Label>*.

Summarizing, using state labels, an MSC specification can be built that has some of the information on how states are (according to [27]) to be merged. LTS synthesis is performed using the algorithm described in previous sections. The resulting ATM component models the same behavior as that of the statechart synthesized by the SCED tool (Fig. 20) that implements the approach described in [27].

Note that, if the component being synthesized had nondeterministic choices, the simple mapping described in this section would have not been sufficient to emulate the approach in [27]. However, it would be possible to extend the ideas presented in this section to also incorporate the assumptions on nondeterminism taken in the SCED approach. We would probably need to incorporate a backtracking algorithm as described in [27] to calculate which states are not to be merged.

Mäkinen and Systä have developed a more recent approach and tool called MAS [33] that address the issue of merging states by using a different synthesis algorithm and an iterative approach that requires a user to accept or reject new scenarios provided by the tool. In this way, the authors can tackle the overgeneralizations that may appear when merging states as done in the SCED approach.

We believe that the general approach taken in [27] and [33] can prove to be beneficial in the construction of state-based formalisms. Inferring from common behavior and posing questions to stakeholders to build and refine the resulting state machine has been shown to aid behavior model development. However, we believe that it is important that the conclusions drawn from the inference

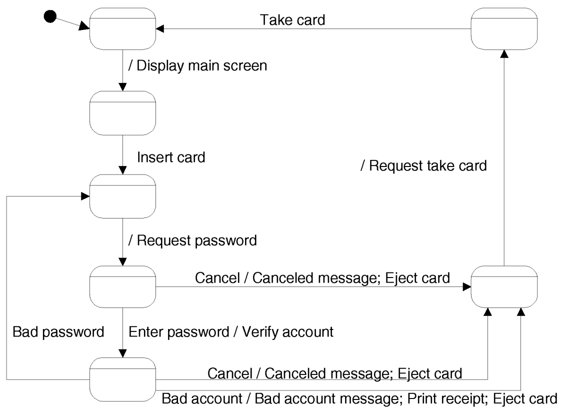

Take card
/ Display main screen
Insert card
/ Request take card
/ Request password
Cancel / Canceled message; Eject card
Bad password
Enter password / Verify account
Cancel / Canceled message; Eject card
Bad account / Bad account message; Print receipt; Eject card

Fig. 18. Statechart synthesised using Whittle and Schumann approach.

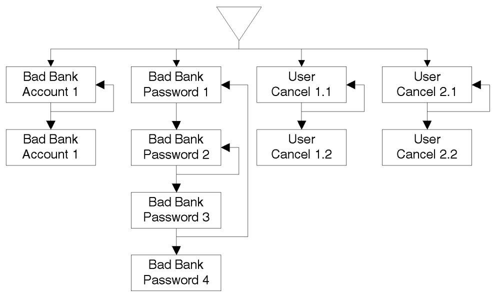


Fig. 17. Explicit specification of detected loops.

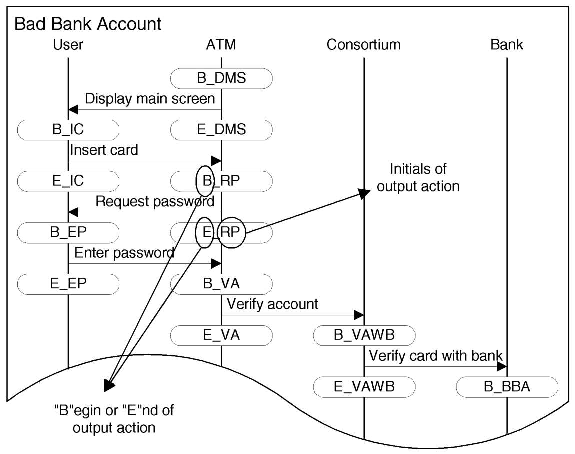


Fig. 19. Annotated bMSC.

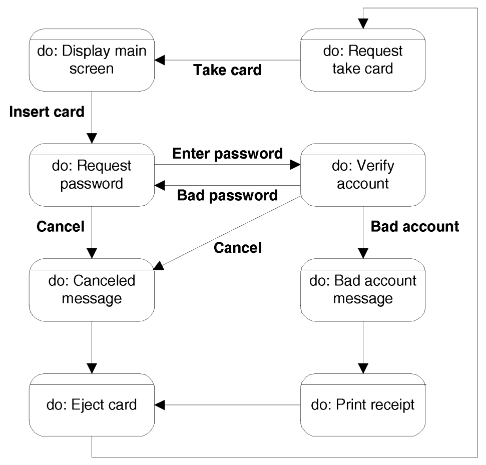


Fig. 20. Statechart synthesized using SCED.

procedures should be made explicit within scenario notations, thus allowing for feedback to stakeholders in the same format as the input they provided.

8 Discussion and Related Work

We have discussed some related work in Section 2, focusing on scenario semantics, management of multiple scenarios, and analysis; we now revisit some of this work going into more detail and also commenting on related work on a broader scale.

Several semantics for scenario-based languages have been proposed and, also, a number of synthesis techniques for building models from a scenario description have been developed. Our work focuses on integrating some of these approaches and supporting other approaches by providing a mechanism for representing their assumptions explicitly.

There are many approaches that generate statechart models from MSCs [27], [28], [50]. Authors argue that statecharts provide a more structured and, therefore, understandable view of behavior. Automatically synthesizing this structure does require that some design decisions be embedded into the synthesis process. However, we argue that this can be counterproductive. Design decisions should be explicit and changeable, particularly as they may vary a great deal according to the system, designer, and organization. Our approach allows some of these decisions to be made explicit with state labels. In addition, we give special importance to producing analyzable models and tools to support such analysis. In particular, we use standard minimization techniques to help to provide compact, comprehensible models.

A difference with approaches using statecharts is that the semantics we use does not retain the labels used to identify states within the scenarios. We use labeled transition systems to define the semantics of MSC; these do not allow state labeling, only transition labeling. However, a labeling convention for state names could be added to our approach. On the other hand, we interpret state labels as a way of explicitly declaring which states in scenarios represent the same state in the state machine model. The actual label is given no further meaning. Thus, each state label determines a state in the LTS and, consequently, there is no loss in terms of behavioral analysis (i.e., observational, trace equivalence).

Many approaches do not explicitly provide semantics for the scenario language they use, providing instead a synthesis algorithm to some other notation. Krüger et al. [28] present a statechart synthesis algorithm in which interpretation of conditions is similar to our use of state labels. However, their approach does not support hMSCs. We share the authors' view of MSC specifications as an *exact* representation of interaction sequences and also the synchronous communication setting.

We have discussed Whittle and Schumann's [50] work in detail in the previous sections. Somé et al. [43] also use additional information to infer equivalences between states. We believe that they too might benefit from making the results of their inference process explicit. We also believe that some aspects of the approach taken in [45] for semantic analysis of sequence diagrams could also be supported by our approach to scenario synthesis. Koskimies et al.'s approach [27] discussed in previous sections also focuses on synthesizing readable and understandable statecharts. The approach has strong assumptions embedded into the synthesis algorithm as to when states should be merged. We have shown how these assumptions can also be made explicit by mapping the approach to ours. The synthesis algorithm of [27], tries to avoid nondeterminism by only unifying states with the same action if they do not have common events leading to different states. Our approach allows for nondeterminism. In fact, in order to synthesize a model with a deterministic choice, the choice must be explicitly modeled in the high-level MSC or using state labels. An example of this has been described in Section 3.

Some approaches give a different semantics to hMSCs and state labels. The latter are often called conditions [28], [39], [22], [40]. Rudolph et al. [40] allow conditions in hMSCs for referencing system states as opposed to component states. A referenced bMSC in an hMSC must have the same initial and final condition as its reference. The redundancy introduced by conditions does not provide an alternative composition mechanism; rather, it provides a double check for consistency between bMSCs and hMSCs. In addition, as pointed out by the authors, all alternatives must be placed in the hMSC, as choices are not allowed in bMSCs. This leads to short bMSCs. The way we use conditions (state labels) addresses this, as it allows many ways of expressing the systems behavior, using long or short, and several or few bMSCs as appropriate. In [28], conditions are interpreted in the same way as we do, however, hMSCs are not considered in the synthesis process.

The formal semantics of MSCs proposed by Reniers [39] is part of the Z.120 recommendations for MSCs [22]. The semantics of hMSCs differs slightly as a *late decision* assumption is used. Late decision means that a component, when choosing between two different possible scenarios, will postpone the decision if both scenarios have common initial events. In our approach, this needs to be explicitly stated using state labels. The advantage of the late decision assumption is that it can reduce the size of specifications. However, we again prefer to make this assumption explicit. Late decision semantics could be translated automatically into state labels in our approach. Furthermore, the Z.120 formal semantic definition is given in terms of process algebra, with nonstandard operators of delayed choice and delayed parallel composition. We prefer the more standard model of LTS with parallel composition. Other formalizations similar to [39] are given in [18], [24].

Buhr and his group have developed a notation called Use Case Maps [7] (UCMs) that allow the description of scenarios at a more abstract level in terms of sequences of responsibilities over a set of components. UCMs do not model explicit intercomponent communication as scenario notations such as MSCs do. UCMs can be used as an abstract specification for the construction of MSC specifications [6].

Van Lamsweerde and Willemet [30] present a very different approach to scenarios. A set of examples and counterexamples expressed as scenarios is used to infer a temporal logic specification. Thus, generating explicit declarative requirements from an operational description. Combining these requirements with LTS models may be an interesting possibility for future work.

9 Conclusions

We have presented a language for MSC specifications with sound abstract semantics in terms of labeled transitions systems and parallel composition that integrates approaches based on scenario composition and on states identification. We have defined and implemented a synthesis algorithm that generates behavioral models for analysis in LTSA and illustrated how this approach can be used to support other approaches to model synthesis and analysis. Using hMSCs, we help to manage complexity of MSC specifications, promoting scenario reuse, and providing a simple, intuitive, operational way of showing how scenarios relate. Using state labels to provide information on component states we help to make explicit any additional information, and domain-specific or general assumptions in MSC specifications. By generating FSP specifications, our approach integrates with LTSA, thus supporting model checking of deadlock, safety, and liveness properties. There is also the potential for model animation as a means of including further domain constraints and of making the models more comprehensible to stakeholders and developers.

Finally, by taking two dissimilar approaches with their own assumptions and their own means of adding information to MSC specifications and, by showing how they can be built upon our approach, we have demonstrated how our approach provides the basis for a common approach to scenario-based specification, synthesis, and analysis. This means that our approach could be used as an intermediate step in approaches where complex assumptions are embedded in synthesis procedures. This intermediate step would provide a notation for making some of the effects these assumption have explicit and providing a common ground for the final synthesis of behavior models that can be analyzed rigorously using LTSA.

One direction for future work is to study how triggered scenarios can be integrated into our approach. Use of triggers and preconditions for describing how scenarios fit together complements well with approaches based on scenario composition and state identification. Another direction is that of managing overlapping or concurrent scenarios.

Finally, scenarios have proven to be a good tool for bridging the gap between stakeholders and developers. However, up to now, this is mainly a one-way bridge in which developers gain more insight of stakeholders' domain knowledge. We believe an important direction of future work should focus on building a bridge in the other direction, in other words, building mechanisms to provide feedback of the developer's world to stakeholders. Preliminary work in this direction is promising. We are automating the construction of alternative system views from synthesised LTS models. Interestingly, many views can be generated by taking advantage of the semantic overlap between hMSCs and state labels. The latter identify component states across scenarios, while the former provide information about all components by relating bMSCs. Moving information from state labels to hMSCs allows for a large number of possible views that vary from long bMSCs that start at the system's initial state to short bMSCs that optimise reuse. These views can allow stakeholders to gain more insight into their own MSC specifications or be used by designers to show the impact of their changes to behavioral models in a language that stakeholders manage.

Acknowledgments

The authors would like to thank the referees for their thorough reviews, helpful comments, and corrections that have improved this paper. Finally, they would like to acknowledge that this research was supported, in part, by the EPSRC under grant BR/M24493 (BEADS).

References

[1] R. Alur, K. Etessami, and M. Yannakakis, "Inference of Message Sequence Charts," *Proc. 22nd IEEE Int'l Conf. Software Eng. (ICSE '00),* pp. 304-313, 2000.

[2] R. Alur, G.J. Holzmann, and D. Peled, "An Analyser for Message Sequence Charts," *Proc. Second Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96),* pp. 35-48, 1996.

[3] F. Andolfi, F. Aquilani, S. Balsamo, and P. Inverardi, "Deriving QNM from MSCs for Performance Evaluation of Software Architectures," *Proc. Second Int'l Workshop Software and Performance (WOSP '00),* pp. 47-57, 2000.

[4] H. Ben-Abdhallah and S. Leue, "MESA: Support for Scenario-Based Design of Concurrent Systems," *Proc. Fourth Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98),* pp. 118-135, 1998.

[5] A.W. Biermann and R. Krishnaswamy, "Constructing Programs from Example Computations," *IEEE Trans. Software Eng.,* vol. 2, pp. 141-153, 1976.

[6] F. Bordeleau, "A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical Finite State Machines," PhD Thesis, Carleton Univ., Ottawa, 1999.

[7] R.J.A. Buhr, "Use Case Maps as Architectural Entities for Complex Systems," *IEEE Trans. Software Eng.,* vol. 24, pp. 1131-1155, 1998.

[8] J.M. Carroll, *Scenario-Based Design: Envisioning Work and Technology in System Development.* New York: Wiley, 1995.

[9] E.M. Clarke and J.M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys,* vol. 28, pp. 626-643, 1996.

[10] A. Engels, L. Feijs, and S. Mauw, "MSC and Data: Dynamic Variables," *Proc. Ninth SDL Forum,* pp. 105-120, 1999.

[11] M. Glinz, "Statecharts For Requirements Specification—As Simple as Possible, as Rich as Needed," *Proc. ICSE 2002 Workshop Scenarios and State Machines: Models, Algorithms, and Tools,* 2002.

[12] H. Gomaa, "Designing Concurrent, Distributed and Real-Time Applications with UML," *Proc. 23rd Int'l Conf. Software Eng. (ICSE'01),* pp. 737-738, 2001.

[13] D. Harel, "From Play-In Scenarios to Code: An Achievable Dream," *IEEE Software,* vol. 34, pp. 53-60, 2001.

[14] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming,* vol. 8, pp. 231-274, 1987.
[15] D. Harel and W. Damm, "LSCs: Breathing Life into Message Sequence Charts," *Proc. Third IFIP Int'l Conf. Formal Methods for Open Object-Based Distributed Systems,* pp. 293-312, 1999.
[16] P. Harmon and M. Watson, *Understanding UML: The Developers Guide.* San Fransisco: Morgan Kaufmann, 1998.
[17] O. Haugen, "From MSC-2000 to UML 2.0—The Future of Sequence Diagrams," *Proc. 10th Int'l SDL Forum,* pp. 38-51, 2001.
[18] S. Heymer, "A NonInterleaving Semantics for MSC," *Proc. SAM '98, First Workshop SDL and MSC,* 1998.
[19] S. Heymer, "A Semantics for MSCs Based on Petri Net Components," *Proc. Second Workshop SDL and MSC (SAM '00),* 2000.
[20] G.J. Holzmann, D. Peled, and M.H. Redberg, "Design Tools for Requirement Engineering," *Bell Labs Technical J.,* vol. 2, pp. 86-95, 1997.
[21] H. Ichikawa, M. Itoh, J. Kato, A. Takura, and M. Shibasaki, "SDE: Incremental Specification and Development of Communications Software," *IEEE Trans. Computers,* vol. 40, pp. 553-561, 1991.
[22] ITU, *Message Sequence Charts,* Recommendation Z.120, Int'l Telecomm. Union., Telecomm. Standardization Sector, 1996.
[23] I. Jacobson, J. Rumbaugh, and G. Booch, *The Unified Software Development Process.* Harlow: Addison Wesley, 1999.
[24] J.P. Katoen and L. Lambert, "Pomsets for Message Sequence Charts," *Proc. First Workshop SDL and MSC (SAM '98),* pp. 197-208, 1998.
[25] J. Klose and H. Wittke, "An Automata Based Interpretation of Live Sequence Charts," *Proc. Seventh Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01),* pp. 512-527, 2001.
[26] P. Kosiuczenko and M. Wirsing, "Towards an Integration of Message Sequence Charts and Timed Maude," *Proc. Fifth World Conf. Integrated Design and Process Technology (IDPT '00),* pp. 23-44, 2000.
[27] K. Koskimies, T. Männistö, T. Systä, and J. Tuonmi, "Automated Support for Modeling OO Software," *IEEE Software,* vol. 15, pp. 87-94, 1998.
[28] I. Krüger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to Statecharts," *Distributed and Parallel Embedded Systems,* F.J. Rammig, ed., Kluwer Academic Publishers, pp. 61-71, 1999.
[29] P.B. Ladkin and S. Leue, "Interpreting Message Flow Graphs," *Formal Aspects of Computing,* vol. 7, pp. 473-509, 1995.
[30] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios," *IEEE Trans. Software Eng.,* vol. 24, pp. 1089-1114, 1998.
[31] S. Leue, L. Mehrmann, and M. Rezai, "Synthesizing ROOM Models from Message Sequence Charts Specifications," *Proc. 13th IEEE Conf. Automated Software Eng. (ASE '98),* pp. 192-195, 1998.
[32] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs.* John Wiley and Sons, 1999.
[33] E. Mäkinen and T. Systä, "MAS—An Interactive Synthesizer to Support Behavioral Modeling in UML," *Proc. 23rd IEEE Int'l Conf. Software Eng. (ICSE '01),* pp. 15-24, 2001.
[34] R. Milner, *Communication and Concurrency.* Prentice-Hall, 1989.
[35] A. Muscholl and D. Peled, "Analyzing Message Sequence Charts," *Proc. SAM '00, Second Workshop SDL and MSC,* pp. 108-122, 2000.
[36] A. Muscholl, D. Peled, and Z. Su, "Deciding Properties of Message Sequence Charts," *Proc. First Int'l Conf. Foundations of Software Science and Computation Structure, (FOSSACS '98),* pp. 226-242, 1998.
[37] R.V. Ommering, "Examples of Horizontal Communication," Internal Report Philips Research, Eindhoven, 2000.
[38] T. Quatrani, *Visual Modeling with Rational Rose 2000 and UML.* Addison Wesley, 1998.
[39] M.A. Reniers, "Message Sequence Chart. Syntax and Semantics," PhD Thesis, Eindhoven Univ. of Technology, Eindhoven, 1999.
[40] E. Rudolph, P. Graubmann, and J. Grabowski, "Tutorial on Message Sequence Charts '96," *Proc. IFIP TC6 WG6.1 Int'l Conf. Formal Description Techniques (FORTE '96),* pp. 1629-1641, 1996.
[41] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design.* Prentice Hall, 1991.
[42] F. Shull, I. Russ, and V.R. Basili, "Improving Software Inspections by Using Reading Techniques," *Proc. Tutorial at 23rd Int'l Conf. Software Eng. (ICSE '01),* pp. 726-727, 2001.
[43] S. Somé, R. Dssouli, and J. Vaucher, "From Scenarios to Timed Automata: Building Specifications from User Requirements," *Proc. Asia Pacific Software Eng. Conf. (APSEC '95),* pp. 48-57, 1995.
[44] P.P. Texel and C.B. Williams, *Use Cases Combined with Booch, OMT, and UML.* Prentice-Hall, 1997.
[45] A. Tsiolakis, "Integrating Model Information in UML Sequence Diagrams," *Electronic Notes in Theoretical Computer Science,* vol. 50, 2001.
[46] S. Uchitel, *MSC-FSP Synthesiser,* http://www.doc.ic.ac.uk/~su2/, 2000.
[47] S. Uchitel, J. Kramer, and J. Magee, "Detecting Implied Scenarios in Message Sequence Chart Specifications," *Proc. Joint Eighth European Software Eng. Conf. (ESEC '01) and Ninth ACM SIGSOFT Symp. Foundations of Software Eng. (FSE '01),* pp. 74-82, 2001.
[48] S. Uchitel, J. Kramer, and J. Magee, "Negative Scenarios for Implied Scenario Elicitation," *Proc. 10th ACM SIGSOFT Symp. Foundations of Software Eng. (FSE '02),* 2002.
[49] Y. Wakahara, Y. Kayuda, A. Ito, and E. Utsunomiya, "ESCORT: An Environment for Specifying Communication Requirements," *IEEE Software,* vol. 6, pp. 38-43, 1989.
[50] J. Whittle and J. Schumann, "Generating Statechart Designs from Scenarios," *Proc. 22nd Int'l Conf. Software Eng. (ICSE '00),* pp. 314-323, 2000.
[51] M.D. Zisman, "Use of Production Systems for Modeling Asynchronous, Concurrent Processes," *Pattern-Directed Inference Systems,* Waterman and Hayes-Roth, eds., Academic Press, pp. 53-68, 1978.

Sebastian Uchitel received the computer science degree from the University of Buenos Aires, Argentina. He is a research associate and PhD candidate in the Department of Computing, Imperial College, London. His research interests include requirements engineering, design methods, analysis techniques, and behavior modeling, particularly as applied to the engineering of concurrent and distributed software-based systems. He is a member of the IEEE Computer Society and ACM.

Jeff Kramer is head of the Department of Computing at Imperial College. His research interests include requirements engineering, software architectures, and analysis techniques, particularly as applied to concurrent and distributed software. He was a principal investigator in the various research projects which led to the development of the CONIC environment for configuration programming and the Darwin architectural description language. His current research work is on behavior analysis, the use of models in requirements elaboration, and architectural approaches to self-organizing software systems. He is a chartered engineer, fellow of the IEE, and fellow of the ACM. He was program cochair of the 21st ICSE in Los Angeles in 1999, chair of the steering committee for ICSE from 2000 to 2002, and associate editor and member of the editorial board of *Transactions on Software Engineering and Methodology* from 1995 to 2001. He is a coauthor of a recent book on concurrency, coauthor of a previous book on distributed systems and computer networks, and the author of more than 150 journal and conference publications. He is a member of the IEEE Computer Society.

Jeff Magee is head of the Software Architecture Research Group in the Department of Computing at Imperial College, London. His research is primarily concerned with the software engineering of distributed systems, including design methods, analysis techniques, operating systems, languages, and program support environments for these systems. His work on software architecture has lead to the commercial use by Phillips of a novel architectural description language in their next generation of consumer television products. He is the author of more than 100 refereed conference and journal publications and has recently written a book on concurrent programming entitled *Concurrency—State Models and Java Programs*. He is a member-at-large of the ACM SIGSOFT committee and currently chairs the steering committee of the International Conference on Software Engineering. He is a member of the IEEE and the IEEE Computer Society.