Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements*

David Harel, Hillel Kugler, and Amir Pnueli

Department of Computer Science and Applied Mathematics,
The Weizmann Institute of Science, Rehovot, Israel
{dharel,kugler,amir}@wisdom.weizmann.ac.il

Abstract. Constructing a program from a specification is a long-known general and fundamental problem. Besides its theoretical interest, this question also has practical implications, since finding good synthesis algorithms could bring about a major improvement in the reliable development of complex systems. In this paper we describe a methodology for synthesizing statechart models from scenario-based requirements. The requirements are given in the language of live sequence charts (LSCs), and may be played in directly from the GUI, and the resulting statecharts are of the object-oriented variant, as adopted in the UML. We have implemented our algorithms as part of the Play-Engine tool and the generated statechart model can then be executed using existing UML case tools.

1 Introduction

Constructing a program from a specification is a long-known general and fundamental problem. Besides its theoretical interest, this question also has practical implications, since finding good synthesis algorithms could bring about a major improvement in the reliable development of complex systems.

Scenario-based inter-object specifications (e.g., via live sequence charts) and state-based intra-object specifications (e.g., via statecharts) are two complementary ways to specify behavioral requirements. In our synthesis approach we aim to relate these different styles for specifying requirements. In [10] the first two coauthors of this paper suggested a synthesis approach using the scenario-based language of live sequence charts (LSCs) [7] as requirements, and synthesizing a state-based object system composed of a collection of finite state machines or statecharts. The main motivation for suggesting the use of LSCs as a requirement language in [10] is its enhanced expressive power. LSCs are an extension of message sequence charts (MSCs; or their UML variant, sequence diagrams) for rich inter-object specification. One of the main additions in LSCs is the notion of universal charts and hot, mandatory behavior, which, among other things, enables one to specify forbidden scenarios. Synthesis is considerably harder for

* This research was supported in part by the John von Neumann Minerva Center for the Verification of Reactive Systems, by the European Commission project OMEGA (IST-2001-33522) and by the Israel Science Foundation (grant No. 287/02-1).

H.-J. Kreowski et al. (Eds.): Formal Methods (Ehrig Festschrift), LNCS 3393, pp. 309–324, 2005.

LSCs than for MSCs, and is tackled in [10] by defining consistency, showing that an entire LSC specification is consistent iff it is satisfiable by a state-based object system. A satisfying system is then synthesized.

There are several issues that have prevented the approach described in [10] from becoming a practical approach for developing complex reactive systems. A major obstacle is the high computational complexity of the synthesis algorithms, that does not allow scaling of the approach to large systems. Additional problems are more methodological, related to the level of detail required in the scenarios to allow meaningful synthesis, the problem of ensuring that the LSC requirements are exactly what the user intended, and a lack of tool support and integration with existing development approaches.

In this paper we revisit the idea of synthesizing statecharts from LSCs, with an aim of addressing the limitations of [10] mentioned above. Our approach benefits from the advances in research made since the publication of [10] – mainly the play-in/play-out approach [13], which supplies convenient ways to capture scenarios and execute them directly, and our previous work on smart play-out [11], which allows direct execution and analysis of LSCs using powerful verification techniques. We suggest a synthesis methodology that is not fully automatic but rather relies on user interaction and expertise to allow more efficient synthesis algorithms. One of the main principles we apply is that the specifier of the requirements provide enough detail and knowledge of the design to make the job easier for the synthesis algorithm. The algorithm tries to prove, using verification methods, that a certain synthesized model satisfies all requirements; if it manages to do so, it can safely synthesize the model. We have developed a prototype statechart synthesis environment, that receives as input LSCs from the Play-Engine tool [13] and generates a statechart model that can then be executed by Rhapsody [15], and in principle also by other UML tools, see e.g., [24, 27].

The paper is organized as follows. Section 2 describes the main challenges in synthesizing statecharts from scenarios and the main principles we adopt to address them. Section 3 shows how to relate the object model of LSCs as supported by the Play-Engine tool with standard UML object models, and describes how this is supported by our prototype tool. Section 4 addresses the notion of consistency of LSCs and introduces a game view for synthesizing reactive systems. Section 5 describes our approach to statechart synthesis, while Section 6 explains the actual statechart synthesis using an example of a cellular phone system. We conclude with a discussion of related work in Section 7.

2 Main Challenges in Synthesis

In this section we discuss some of the main challenges that need to be addressed in order to make a method for synthesizing statechart models from scenarios successful. The challenges are of different nature, varying from finding a scenario-based language that is powerful and easy for engineers to learn, to dealing with the inherent computational complexity of synthesis algorithms that must handle large complex systems.

2.1 Appropriate Scenario-Based Language

An important usage of scenario notations is for communicating ideas and for documentation. For such purposes sketching an inter-object scenario on a blackboard or diagram editor can be very helpful. When our goal is synthesizing a statechart model and eventually production code from the scenarios, we need a powerful and expressive inter-object scenario-based language with rigorously defined semantics. The language should still retain the simplicity and intuitive feel that made scenario-based languages popular among engineers. In our approach we use the language of live sequence charts (LSCs) introduced in [7]. LSCs extends classical message sequence charts, which have very limited expressive power. Among other things, LSCs distinguish between behaviors that may happen in the system (existential) from those that must happen (universal). An example of a universal chart appears in Fig. 1. A universal chart contains a *prechart* (dashed hexagon), which specifies the scenario which, if successfully executed, forces the system to satisfy the scenario given in the actual chart body. For more details on LSCs see [7, 13, 14].

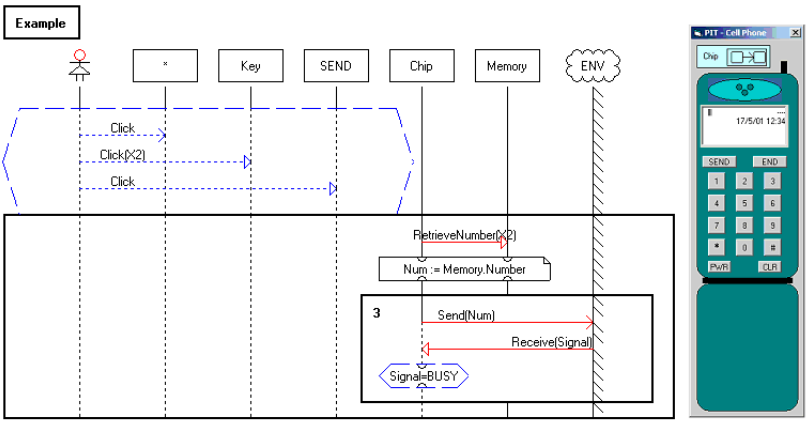

Fig. 1. Example of a universal LSC.

2.2 Sufficiently Detailed Scenario-Based Specification

Specifying requirements of a system is a very difficult task, which must be carried out in a careful and accurate manner. For this reason, it may be claimed that requirements in general, and scenario-based ones in particular, will only be partial and will focus on certain important properties and concepts of the system. According to this argument it is not possible to beneficially apply a synthesis approach for deriving a system implementation since the requirement model provides insufficient details.

We attempt to overcome this challenge by using the play-in/play-out approach introduced in [13, 14]. In play-in the user starts with a graphical representation of the system and specifies various scenarios by interacting with

the GUI and demonstrating the required behavior. As this is being done, the Play-Engine tool constructs the LSC that captures what was played in. Play-in enables non-technical stake holders to participate in the requirement elucidation phase, and to contribute to building a detailed scenario model. Our experience in several projects [12] shows that the play-in/play-out approach enhanced to a large extent the efficiency of this process and allowed building rich and detailed scenario-based requirements, which can serve as a solid starting point for synthesis algorithms.

2.3 Correct Scenarios

As mentioned earlier, specifying requirements is a difficult job, and the user must be sure that the property specified is exactly what is intended. In the context of formal verification, many times when verifying a system with respect to a specified property the result shows that the system does not satisfy the property, and then the user realizes that the property specified was not exactly the intended one and refines it. In a synthesis approach the requirements themselves must be accurate, otherwise even if the synthesis algorithms work perfectly the obtained system will not be what was actually intended.

We try to address this challenge in several complementary ways. First, the requirement language of LSCs, being an extension of classical MSCs, has intuitive semantics, and allows users who are not very technical to express complex behavioral requirements, while other formalisms, e.g., temporal logic, may prove to be trickier even for advanced users. Second, play-out, the complementary process to play-in, allows one to execute the LSCs directly, giving a feeling of working with an executable system. This makes it possible to debug the requirements specification and gain more confidence that what is specified is exactly what is required.

2.4 The Complexity of Synthesis Algorithms

Solving the problem of synthesis for open reactive systems is an inherently difficult problem. In various settings the problem is undecidable, and even in more restricted settings when it is becomes decidable, the time and space requirements of the synthesis algorithm may be too large to be practical for large systems.

One way we attack this problem is by applying methods from formal verification, in ways that will be discussed below. We have in mind mainly model-checking algorithms, which in recent years – due to intensive research efforts and tool development – have scaled nicely in terms of the size of the models they can handle. Nevertheless, the models that can be treated even using state of the art technologies are still limited in size and much more work is needed here to make synthesis a practical approach.

In our current work, one of the main principles we apply is that the specifier of the requirements provide enough detail and knowledge of the design to make the job easier for the synthesis algorithm. The algorithm tries to prove that a certain synthesized model satisfies all requirements; if it manages to do that it

can safely synthesize the model. This approach is not complete, since some other model may be correct and the synthesis algorithm will fail to find it. However, our hope is that for many interesting cases the synthesis will succeed.

2.5 Integration with Existing Code and System Modification

In order to make a new system development approach practical, an important requirement is that it should fit in nicely with other existing approaches. In our context of designing complex embedded software, the synthesized statechart-based model may need to interact with other software that was developed in other diverse ways. By synthesizing into a UML-based framework, we attempt to address this issue and thus to take advantage of the integration capabilities of existing commercial UML tools.

Related to this issue is our recent work on InterPlay [3]. InterPlay is a simulation engine coordinator that supports cooperation and interaction of multiple simulation and execution tools. It makes it possible to connect several Play-Engines to each other, and also to connect a statechart-based executable model in Rhapsody to the Play-Engine. A model synthesized using algorithms described in this paper can thus be linked to the Play-Engine, allowing the scenarios to be monitored as they occur. It also supports an environment in which some subsystems run a statechart or code-based model and others execute LSCs directly, say, by play-out.

3 Transferring the Structure

Scenario-based inter-object specifications (via LSCs) and state-based intra-object specifications (via statecharts) are two complementary ways for specifying behavioral requirements. In our synthesis approach we aim to relate these different styles for specifying requirements.

According to the play-in/play-out approach the user specifies behavioral requirements by playing on a GUI representation of the system, as this is being done the Play-Engine automatically constructs corresponding requirements in LSCs.

3.1 The Play-Engine Object Model

We now introduce the object model used by the Play-Engine, which is the basis for the LSC specifications. We later explain how this object model is related to standard UML models, allowing our prototype tool to connect to models in existing UML tools, and allowing to synthesize statechart-based UML models. For a detailed explanation of the Play-Engine framework and object model see [13, 14].

An object system $\mathcal{S}ys$ is defined as

$$\mathcal{S}ys = \langle \mathcal{D}, \mathcal{C}, \mathcal{O}, \mathcal{F} \rangle$$

where $\mathcal{D}$ is the set of application types (domains), $\mathcal{C}$ is the set of classes, $\mathcal{O}$ is the set of objects, $\mathcal{F}$ is the set of externally implemented functions. We refer to the user of the system as $User$ and to the external environment as Env.

A type $D \in \mathcal{D}$ is simply a (finite) set of values. The basic types supported are range, enumeration and string.

A class C is defined as:

$$C = \langle Name, \mathcal{P}, \mathcal{M} \rangle$$

where *Name* is the class name, $\mathcal{P}$ is the set of class properties and $\mathcal{M}$ is the set of class methods.

An object O is defined as:

$$O = \langle Name, C, \mathcal{PV}, External \rangle$$

where *Name* is the object's name, C is its class, $\mathcal{PV} : C.\mathcal{P} \rightarrow \bigcup_i D_i$ is a function assigning a value to each of the object's properties and $External$ indicates whether the object is an external object. We define the function $class : \mathcal{O} \rightarrow \mathcal{C}$ to map each object to the class it is an instance of. We also use $Value(O.P) = O.\mathcal{PV}(O.C.P)$ to denote the current value of property P in object O.

An object property P is defined as

$$P = \langle Name, D, InOnly, ExtChg, Affects, Sync \rangle$$

where *Name* is the property name and D is the type it is based on. $InOnly \in \{True, False\}$ indicates whether the property can be changed only by the user, $ExtChg \in \{True, False\}$ indicates whether the property can be changed by the external environment, $Affects \in \{User, Env, Self\}$ indicates the instance to which the message arrow is directed when the property is changed by the system, and $Sync \in \{True, False\}$ indicates whether the property is synchronous.

An object method M is defined as:

$$M = \langle Name(D_1, D_2, \ldots, D_n), Sync \rangle$$

where Name is the method name, $D_i \in \mathcal{D}$ is the type of its i^{th} formal parameter and $Sync \in \{true, false\}$ indicates whether calling this method is a synchronous operation.

An implemented function is defined as:

$$Func = Name : D_1 \times D_2 \times \ldots, \times D_n \rightarrow D_F$$

where $Name$ is the function name, $D_i \in \mathcal{D}$ is the type of its i^{th} formal parameter and $D_F \in \mathcal{D}$ is the type of its returned value.

3.2 Importing a UML Model into the Play-Engine

The usual work-flow in the play-in/play-out approach as supported by the Play-Engine is that the user starts by building a GUI representation and the corresponding object model. As part of our current work we support an alternative starting point, in which a UML model is imported into the Play-Engine, (say, from RHAPSODY), and can then be used while specifying the behavior using

LSCs and the play-in process. This shows the relation between the Play-Engine object model and a standard UML model, and also from the more practical point of view it provides an easy link to models developed in existing UML tools and a good starting point for applying our synthesis approach.

The import procedure is quite straightforward, we describe here only its general principles. Types in the UML model are converted to Play-Engine types, as defined in the previous section. Currently the Play-Engine supports only simple type definitions – range, enumeration and string. A type that cannot be defined in terms of these basic type definitions is declared as `EngineVariant`, the default Play-Engine type. The Play-Engine currently does not support packages, the UML construct for grouping classes, so that when importing UML classes they all appear in a flat structure. UML attributes are mapped to Play-Engine properties, preserving their corresponding type. For each UML class, the operations are imported as Play-Engine methods, with the arguments preserving their corresponding types.

Instances in the UML model are defined as internal objects, preserving their base class. In the Play-Engine, internal objects are visualized using something resembling class diagrams, and play-in is supported by clicking and manipulating this kind of diagram in a convenient way. This allows rapid development of requirements without a need to construct a GUI. Building a GUI has many benefits in terms of visualizing the behavior, but as a first approximation importing the model and playing-in using internal objects works fine.

3.3 Synthesizing a Skeleton UML Model

Complementary to the UML to Play-Engine import described in the previous subsection, we also support the synthesis of a skeleton UML model from the Play-Engine; that is, a UML model containing the object model definitions, but without taking the LSC specifications into account and without synthesizing any statecharts. This skeleton synthesis can be useful if we have a complex Play-Engine model we have developed, and now want to go ahead and build a corresponding UML model. We can apply the synthesis of the skeleton model, thus automating the straightforward part, and then proceed to do the interesting and creative part, regarding dynamic behavior, by defining the UML statecharts manually. We may want to use this approach when we have special motivation to create the statechart model manually (see, e.g., [9] for an example), or when the automatic synthesis algorithms do not work properly. Using the InterPlay approach [3] mentioned earlier, we can then execute the statechart-based UML model linked to the Play-Engine, allowing the scenarios to be monitored as they occur.

4 Consistency of LSCs

Before being able to synthesize a statechart based model we must ensure that the LSCs are consistent. Consider the two charts `OpenAntGrad1` and `OpenAntGrad2` in Fig. 2. When the user opens the Antenna both charts are activated. However,

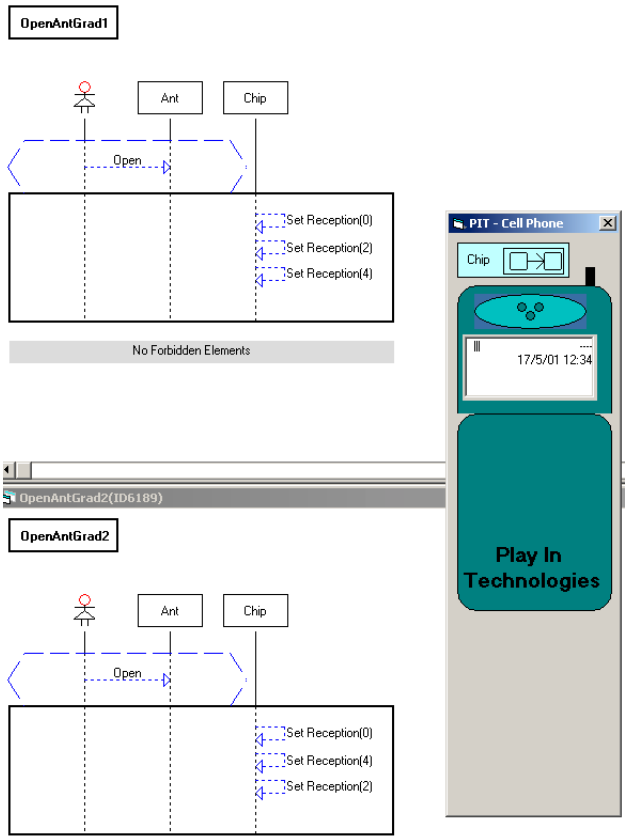

Fig. 2. Inconsistent LSCs.

there is no way to satisfy them both since after changing the reception level of the Chip to 0 (as required by both charts), the first chart requires that the reception level change to 2 and only later to 4, while the second one requires that the reception level change first to 4 and only later to 2. These are clearly contradictory. While this is a very simple example, such contradictions can be a lot more subtle, arising as a result of the interaction between several charts. In large specifications this phenomena can be very hard to analyze manually. Our tool can automatically detect some of these inconsistencies and provide information to the user. After the relevant LSCs are fixed the synthesis algorithm can again be applied. This can lead to an iterative development process at the end of which a consistent LSC specification is obtained, and a statechart model can be synthesized.

4.1 A Game View

In the study of synthesis of reactive systems a common view is that of a game between two players [6, 22]. One of the players is the environment and the other

is the system. The players alternate turns each one making a move in his turn, and the requirements define the winning condition. If there exists a strategy for the system under which for any moves the environment makes the system always wins, we say the specification is realizable (consistent) and we can attempt to synthesize a system implementation.

In the Play-Engine tool while using LSCs as the requirement language, the environment can be a **User** object, as in the prechart of Fig. 1, or a more explicit environment object, as represented by the **ENV** object appearing in the main chart of Fig. 1, or an external object, an object that is designated as being implemented outside the Play-Engine specification. In principle, the clock object, which represents global time, should also be considered external, but the treatment of time is beyond the scope of this paper. All other objects are assumed to be part of the system.

The game is played as follows: the environment makes a move, consisting of performing a method call or modifying the value of an externally changeable property. The system responds by performing a superstep, a finite sequence of system events, and then it is again the environment's turn. The system is the winner of the game if all LSC requirements are satisfied, otherwise the environment is the winner.

For the finite state case, when the number of objects is finite, all types are of finite domain, and the number of different simultaneously active copies of a chart is bounded, the game can be solved using model-checking methods. An implementation of the game problem is now part of the Weizmann Institute model-checker TLV [23]. The computation complexity of the algorithms is still a major limitation in applying this game approach.

In our current work, one of the main principles we apply is that the specifier of the LSCs provide enough detail and knowledge of the design, to make the job easier for the synthesis algorithm. LSCs as a declarative, inter-object behavior language, enables formulating high level requirements in pieces (e.g., scenario fragments), leaving open details that may depend on the implementation. The partial order semantics among events in each chart and the ability to separate scenarios in different charts without having to say explicitly how they should be composed are very useful in early requirement stages, but can cause underspecification and nondeterminism when one attempts to execute them.

In play-out, if faced with nondeterminism an arbitrary choice is made. From our experience in several projects, by providing a detailed enough LSC requirement play-out can get very close to solving the game problem, and sometimes can even solve it directly. Assuming the user provided enough knowledge for the synthesis algorithm, the algorithm tries to prove that a certain synthesized model will satisfy all requirements, and if it manages to do this it can safely synthesize the model. This approach is not complete, thus a different synthesized model may be correct and the synthesis algorithm may fail to find it, but our hope is that for many interesting cases the synthesis will succeed. In a situation where for a synthesized model we have not managed to prove it correct or to

find some problem with it, synthesizing a state-based model opens possibilities to try to prove its correctness using other tools and techniques, e.g., [25, 2].

5 The Synthesis Approach

In order to apply the synthesis approach we encode play-out in the form of a transition system and then apply model-checking techniques. We construct a transition system which has one process for each actual object. A state in this system indicates the currently active charts and the location of each object in these charts. The transition relation restricts the transitions of each process only to moves that are allowed by all currently active charts. We now provide some more of the details on how to translate LSCs to a transition system. The encoding of the transition relation was developed as part of our work on smart play-out [11].

An LSC specification LS consists of a set of charts M, where each chart $m \in M$ is existential or universal. We denote by $pch(m)$ the prechart of chart m. Assume the set of universal charts in M is $M^U = \{m_1, m_2, ..., m_t\}$, and the objects participating in the specification are $\mathcal{O} = \{O_1, ..., O_n\}$.

We define a system with the following variables:

act_{m_i} determines if universal chart m_i is active. It gets value 1 when m_i is active and 0 otherwise.

$msg^s_{O_j \to O_k}$ denoting the sending of message msg from object O_j to object O_k. The value is set to 1 at the occurrence of the send and is changed to 0 at the next state.

$msg^r_{O_j \to O_k}$ denoting the receipt by object O_k of message msg sent by object O_j. Similarly, the value is 1 at the occurrence of the receive and 0 otherwise.

l_{m_i,O_j} denoting the location of object O_j in chart m_i, ranging over $0 \cdots l^{max}$ where l^{max} is the last location of O_j in m_i.

$l_{pch(m_i),O_j}$ denoting the location of object O_j in the prechart of m_i, ranging over $0 \cdots l^{max}$ where l^{max} is the last location of O_j in $pch(m_i)$.

We use the asynchronous mode, in which a send and a receive are separate events, but we support the synchronous mode too. The details of encoding the transition relation are rather technical, for more information see [11].

Given this encoding we claim that play-out is correct if the following property holds.

$$\neg(EF(AG(\bigvee_{m_i \in M^U} (act_{m_i} = 1))))$$

The property specified above is a temporal logic property [8]. The operators E, A are the existential and universal path quantifiers respectively, while F and G are the eventually and always temporal logic operators. Intuitively, this formula claims that it is not the case that eventually play-out may get stuck, not being able to satisfy the requirements successfully.

We now apply the model-checker to prove this property, and if it is indeed correct we can go on and synthesize the system. The basic synthesis scheme

generates a statechart for each of the participating objects, using orthogonal states for implementing different scenarios and making use of additional events to guarantee synchronization of the distributed objects along each behavioral scenario. More details are given in the next section. If the property does not hold we can apply model-checking to a variation of this property and can sometimes obtain more information on how the LSCs can be fixed so that play-out will be correct.

6 An Example of Statechart Synthesis

We use an example of a cellular phone system to illustrate our synthesis algorithms. A GUI representation of the system appears on the right-hand side of Fig. 2. The system is composed of several objects, including the `Cover`, `Display`, `Antenna` and `Speaker`. We consider a specification consisting of several universal charts.

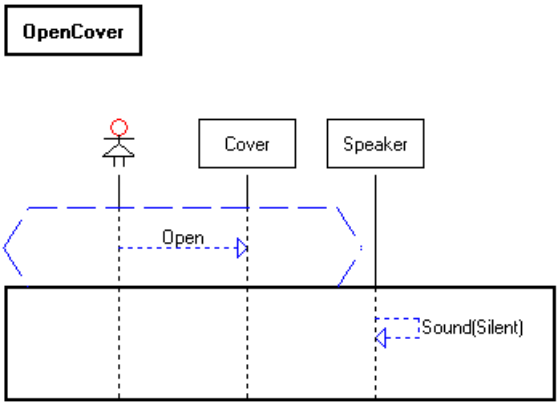

Fig. 3. Open Cover.

The chart `OpenCover`, appearing in Fig. 3, requires that whenever the user opens the `Cover`, as specified in the prechart, the `Speaker` must turn silent.

The charts `OpenAnt`, `CloseAnt`, appearing in Fig. 4, specify that whenever the user opens the `Antenna` the `Display` shows that the reception level is changed to 4, and whenever the user closes the `Antenna` the `Display` shows that the reception level is changed to 1.

The resulting statecharts for the `Antenna` and the `Display` obtained by applying the synthesis algorithms appear in Fig. 5 and Fig. 6 respectively. Consider the `Antenna` statechart of Fig. 5. The AND-**state** named *Top* contains two orthogonal states *OpenAnt* and *CloseAnt*, corresponding to the scenarios of opening and closing of the `Antenna`.

The orthogonal state *OpenAnt* has three substates, $P0, P1$ and $S0$, where $P0$ is the initial state entered, as designated by the default transition into $P0$.

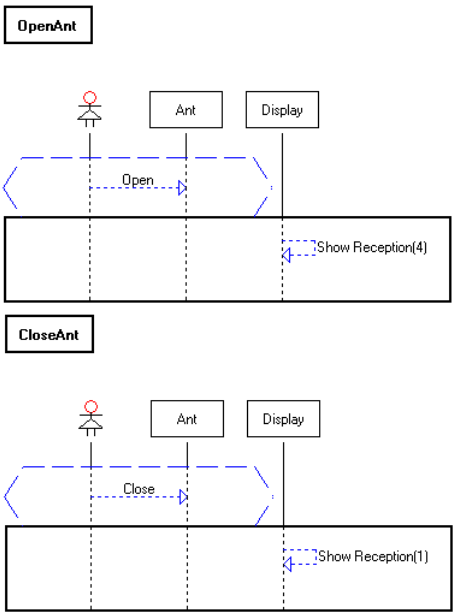

Fig. 4. Opening and Closing the Antenna.

The states $P0, P1$ and $S0$ correspond to progress of the `Antenna` object along the *OpenAnt* scenario, where we use the convention that P states correspond to prechart locations while S states correspond to main chart locations. If the `Antenna` object is in state $P0$ of the *OpenAnt* orthogonal component, and it receives the event `Open`, it takes a transition to state $P1$ and performs the action written in the label of the transition. The action has the effect of telling the other objects that the scenario of opening of the `Antenna` has been activated. This is done by sending the event `activeOpenAnt` to the other objects, i.e., the command `getItsDisplay_C()->GEN(activeOpenAnt)` generates an event `activeOpenAnt` and sends it to the `Display` object. In a similar way the event `activeOpenAnt` is generated and sent to the `Cover` and `Speaker` objects. The `Antenna` object, which is now in the sub-state $P1$ of the *OpenAnt* component, takes the null transition to state $S0$. Null transitions are transitions with no trigger event, and are taken spontaneously.

The `Display` object is originally in state $P0$ of the *OpenAnt* orthogonal state. It receives the event `activeOpenAnt` (sent by the `Antenna`), causing the transition to state $S0$ to be taken, meaning that now the object has progressed to the main chart of the scenario. From state $S0$ a null transition to state $S1$ is taken, and the reception level of the `Display` is set to volume level 4. This is done by performing the method `setReception(V_4)`, which sets the value of the attribute `reception` to `V_4`. As part of the action of the transition from state $S0$ to state $S1$ the other objects are notified that the scenario of opening of the `Antenna` is over, this is done by performing the command `getItsAnt_C()->GEN(overOpenAnt)`, and similarly for other objects. The `Display` object then takes the null transition

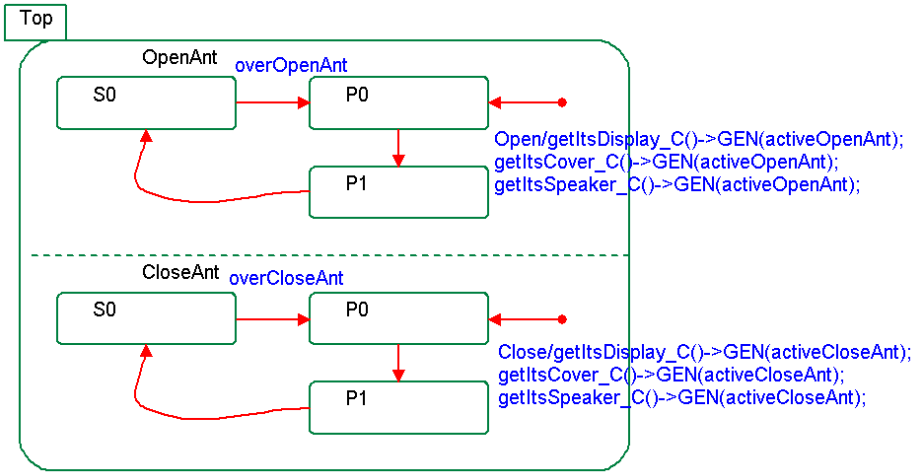

Fig. 5. Synthesized Antenna statechart.

back to state $P0$. The `Antenna` object on receiving the event `overOpenAnt` takes the transition from state $S0$ back to state $P0$. At this point the scenario of opening the `Antenna` has completed successfully. The statechart synthesis algorithm implements the scenario of closing the `Antenna` in a similar way, as reflected by the *CloseAnt* components of the `Antenna` and `Display` objects.

There are several points were the synthesis algorithm can be optimized to produce more efficient and readable models, and indeed we have a first version of such an improved algorithm. When sending an event to all other objects to notify them of some occurrence (for example when taking the transition from state $P0$ to state $P1$ in orthogonal component *OpenAnt* of the `Antenna`) it is enough in our case to send the event only to the `Display` object, since the objects `Cover` and `Speaker` do not participate and are not affected by the opening `Antenna` scenario.

A related issue is the architecture of the synthesized model: In this example, we allow each object to communicate directly with each of the other objects in the system, and we synthesize the relations in the UML model to allow this. Thus, for example, the `Antenna` object can relate to the `Speaker` by the `getItsSpeaker_C()` command. Using an optimized algorithm, if this communication is not used the corresponding relations will not be synthesized. For improved readability, if an action contains several commands of similar nature, e.g., sending an event to various objects, an optimized synthesis algorithm will define a method performing these related commands, and the label of the transition will include a call to this method, thus resulting in more readable and elegant statecharts than those of Fig. 5 and 6.

As mentioned earlier, an important part of the synthesis is to apply the play-out consistency check, as described in Section 5, which guarantees the correctness of the synthesis algorithm.

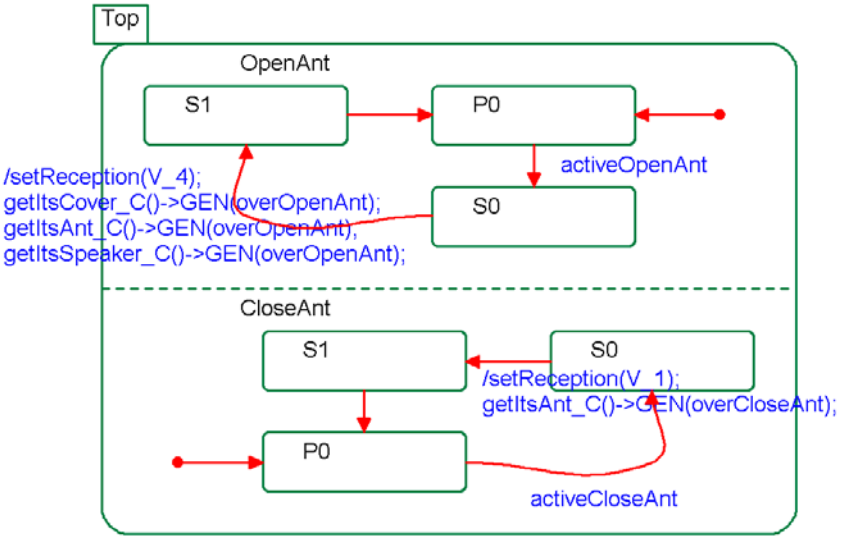

Fig. 6. Synthesized Display statechart.

7 Related Work

The idea of deriving state-based implementations automatically from scenario-based requirements has been the subject of intensive research efforts in recent years; see, e.g., [17, 18, 20, 19, 29]. Scenario-based specifications are very useful in early stages of development, they are used widely by engineers, and a lot of experience has been gained from their being integrated into the MSC ITU standard [21] and the UML [28]. The latest versions of the UML recognized the importance of scenario-based requirements, and UML 2.0 sequence diagrams have been significantly enhanced in expressive capabilities, inspired by the LSCs of [7].

There is also relevant research on statechart synthesis. As far as the case of classical message sequence charts goes, work on synthesis includes the SCED method [17] and synthesis in the framework of ROOM charts [20]. Other relevant work appears in [4, 26, 1, 19, 29]. In addition, there is the work described in [16], which deals with LSCs, but synthesizes from a single chart only: an LSC is translated into a timed Büchi automaton (from which code can be derived).

While the work in [10, 5] addressed the synthesis problem of LSCs from a theoretical viewpoint, the current paper applies new verification-based techniques and also reports on a prototype implementation. Other aspects special to our approach were described in Section 2 above. In addition to synthesis work directly from sequence diagrams of one kind or another, one should realize that constructing a program from a specification is a long-known general and fundamental problem. For example, there has been much research on constructing a program from a specification given in temporal logic (e.g., [22]).

References

1. R. Alur and M. Yannakakis. Model checking of message sequence charts. In *10th International Conference on Concurrency Theory (CONCUR99)*, volume 1664 of *Lect. Notes in Comp. Sci.*, pages 114–129. Springer-Verlag, 1999.
2. T. Arons, J. Hooman, H. Kugler, A. Pnueli, and M. van der Zwaag. Deductive Verification of UML Models in TLPVS. In *Proc. 7th International Conference on UML Modeling Languages and Applications (UML 2004)*, Lect. Notes in Comp. Sci., pages 335–349. Springer-Verlag, October 2004.
3. D. Barak, D. Harel, and R. Marelly. InterPlay: Horizontal Scale-Up and Transition to Design in Scenario-Based Programming. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lect. Notes in Comp. Sci.*, pages 66–86. Springer-Verlag, 2004.
4. A.W. Biermann and R. Krishnaswamy. Constructing programs from example computations. *IEEE Trans. Softw. Eng.*, SE-2:141–153, 1976.
5. Y. Bontemps and P.Y. Schobbens. Synthesizing open reactive systems from scenario-based specifications. In *Proc. of the 3rd Int. Conf. on Application of Concurrency to System Design (ACSD'03)*. IEEE Computer Science Press, 2003.
6. J.R. Buchi. State-strategies for games in $F_{\sigma\delta} \cap G_{\delta\sigma}$. *J. Symb. Logic*, 48:1171–1198, 1983.
7. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001. Preliminary version appeared in Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99).
8. E.A. Emerson. Temporal and modal logics. In J. van Leeuwen, editor, *Handbook of theoretical computer science*, volume B, pages 995–1072. Elsevier, 1990.
9. J. Fisher, D. Harel, E.J.A. Hubbard, N. Piterman, M.J. Stern, and N. Swerdlin. Combining state-based and scenario-based approaches in modeling biological systems. In *Proc. 2nd Int. Workshop on Computational Methods in Systems Biology (CMSB 2004)*, Lect. Notes in Comp. Sci. Springer-Verlag, 2004.
10. D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. *Int. J. of Foundations of Computer Science (IJFCS)*., 13(1):5–51, Febuary 2002. (Also,*Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000)*, July 2000, Lecture Notes in Computer Science, Springer-Verlag, 2000.).
11. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *Proc. 4^{th} Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'02), Portland, Oregon*, volume 2517 of *Lect. Notes in Comp. Sci.*, pages 378–398, 2002. Also available as Tech. Report MCS02-08, The Weizmann Institute of Science.
12. D. Harel, H. Kugler, and G. Weiss. Some Methodological Observations Resulting from Experience Using LSCs and the Play-In/Play-Out Approach. Tech. Report MCS04-06, The Weizmann Institute of Science, 2004.
13. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
14. D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: The Play In/Play-Out Approach. *Software and System Modeling (SoSyM)*, 2(2):82–107, 2003.
15. Rhapsody. I-Logix, Inc., products web page. http://www.ilogix.com/products/.

16. J. Klose and H. Wittke. An automata based interpretation of live sequence chart. In *Proc. 7^{th} Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), volume 2031 of Lect. Notes in Comp. Sci., Springer-Verlag*, 2001.
17. K. Koskimies and E. Makinen. Automatic synthesis of state machines from trace diagrams. *Software – Practice and Experience*, 24(7):643–658, 1994.
18. K. Koskimies, T. Mannisto, T. Systa, and J. Tuomi. SCED: A Tool for Dynamic Modeling of Object Systems. Tech. Report A-1996-4, University of Tampere, July 1996.
19. I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In *Proc. Int. Workshop on Distributed and Parallel Embedded Systems (DIPES'98)*, pages 61–71. Kluwer Academic Publishers, 1999.
20. S. Leue, L. Mehrmann, and M. Rezai. Synthesizing ROOM models from message sequence chart specifications. Tech. Report 98-06, University of Waterloo, April 1998.
21. ITU-TS Recommendation Z.120 (11/99): MSC 2000. ITU-TS, Geneva, 1999.
22. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, pages 179–190, 1989.
23. A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In R. Alur and T. Henzinger, editors, *R. Alur and T. Henzinger, editors, Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96), volume 1102 of Lect. Notes in Comp. Sci., Springer-Verlag*, pages 184–195, 1996.
24. Rational Rose Technical Developer. Rational, Inc., web page. http://www-306.ibm.com/software/awdtools/developer/technical/.
25. I. Schinz, T. Toben, and B. Westphal. The Rhapsody UML Verification Environment. In *2nd Int. Conf. on Software Engineering and Formal Methods*. IEEE Computer Society Press, 2004.
26. R. Schlor and W. Damm. Specification and verification of system-level hardware designs using timing diagram. In *European Conference on Design Automation*, pages 518–524, Paris, France, 1993. IEEE Computer Society Press.
27. Telelogic TAU. Telelogic, Inc., web page. http://www.telelogic.com/products/tau/.
28. UML. Documentation of the unified modeling language (UML). Available from the Object Management Group (OMG), http://www.omg.org.
29. J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *22nd International Conference on Software Engineering (ICSE 2000)*, pages 314–323. ACM Press, 2000.