Scenario-Driven Modeling and Validation of Requirements Models

Christian Seybold, Silvio Meier, Martin Glinz
Requirements Engineering Group, Department of Informatics, University of Zurich
Binzmühlestrasse 14, CH-8050 Zurich, Switzerland
{seybold | smeier | glinz}@ifi.unizh.ch

ABSTRACT

Requirements models for large systems typically cannot be developed in a single step, but evolve in a sequence of iterations. We have developed such an iterative modeling process which is based on the interactive simulation of yet incomplete and semi-formal models. Missing parts are completed interactively by the user simulating the model. We start by modeling type scenarios (i.e. use cases) and simulate these interactively before having specified any system behavior. Such simulation runs yield exemplary system behavior in form of message sequence charts (MSCs). The modeler can then generalize this recorded partial behavior into statecharts. The resulting model is simulated again, (i) for validating that the modeled behavior matches the previously recorded behavior, and (ii) for recording new yet unspecified behavior in a next iteration step. Thus, recording MSCs by playing-through the scenarios and transforming MSCs to statecharts stimulate and drive each other.

In this paper we focus on two elements of our approach: firstly, we describe the syntax and semantics of our scenario language. Secondly, we give an example how our modeling process works.

Categories and Subject Descriptors

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

General Terms

Design, Languages, Human Factors

Keywords

Scenarios, Statecharts, Modeling, Simulation, ADORA

1. INTRODUCTION

When developing large requirements models in an iterative process, the models are typically incomplete and not completely formalized during the development process. Modeling proceeds by progressively making models more complete and more formal. The process stops when the model has reached the desired degree of formality and completeness (which will vary depending on risk, time and budget). Hence, a modeling language should support such a process by providing features for modeling intentional partial incompleteness and a variable degree of formality.

Such an iterative process crucially depends on early and frequent model validations. Reviews quickly become too expensive when they have to be performed frequently and repeatedly. On the other hand, classic automated techniques such as simulation or model checking cannot be applied because they require formal models.

Principally, simulation would be a quite appropriate technique for both requirements engineers and customers to validate whether a model behaves as desired: they can play with the model's dynamics by entering stimuli and receiving system reactions. Therefore, we have developed an interactive simulation technique that allows to simulate incomplete and semi-formal models by inquiring missing information interactively from the expert who runs the simulation. The information provided by the expert is recorded so that regression simulation becomes possible [12].

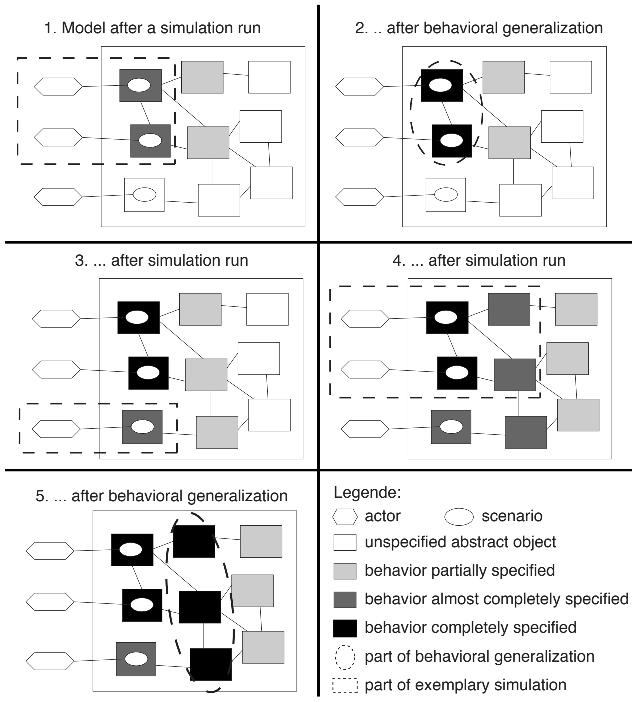
This technique also enables an iterative, outside-in development process for requirements models that starts with some external behavior specified by type-level user-system-interaction scenarios (aka use cases) and progressively elicits and defines system behavior with simulations that are driven by playing through the scenarios (Figure 1).

The process is based on the following observations. The behavior of a system can be specified in two different ways:

- It can be specified in an exemplary way by a set of typical play-throughs of the user-system-interaction scenarios. These play-throughs constitute message sequence charts (MSCs), which give an outline how the system should behave. The description is exemplary, because each run covers only a certain interaction with the system.

- When all potential ways how a component can behave are known, the component's behavior can be modeled generically, for example with statecharts. In contrast to MSCs, statecharts cover all potential interactions between the component and its environment.

Exemplary behavior is cognitively easier to develop and also better suited for discussing models with a customer for



1. Model after a simulation run
2. .. after behavioral generalization
3. ... after simulation run
4. ... after simulation run
5. ... after behavioral generalization
Legende:
actor
scenario
unspecified abstract object
behavior partially specified
behavior almost completely specified
behavior completely specified
part of behavioral generalization
part of exemplary simulation

Figure 1: Process of enriching model with behavior.

validation purposes. So it is a good starting point for system modeling. However, as exemplary behavior just gives an outline of certain situations, we eventually need generic behavior models. The problem is that generic modeling from scratch is difficult and requires considerable effort and expertise.

Hence, a modeling language should support both types of behavioral modeling, and an iterative modeling process should provide transitions from exemplary to generic modeling. The existing algorithms for transforming MSCs into statecharts (for example [13]) have drawbacks concerning the readability and the validation of the generated statecharts.

We therefore take an alternative approach, where the construction of statecharts from exemplary behavior is manual, but systematically guided and informed by simulation runs which in turn are produced by interactively playing through a set of exemplary scenario executions [11].

Vice versa, having generically specified the behavior of a component improves and eases the subsequent simulation runs as the modeled behavior is being integrated into the simulation, thus making it more precise and requiring less interactivity during simulation runs.

When used this way, the two modeling modes, viz. recording exemplary system behavior by playing through scenarios and modeling system behavior generically, stimulate each other and drive the model development towards more complete and more formal specifications. Fig. 1 illustrates how an alternating application of simulation runs and behavioral generalizations contribute to the evolution of a system model.

In this paper, we focus on the following two elements of our approach: In Chapter 2, we describe our scenario notation, as a well-suited and precisely defined scenario language is a critical building block of our approach. In Chapter 3, we demonstrate our modeling process with an example. Chapter 4 covers related work and Chapter 5 provides a discussion and some concluding remarks.

2. OUR SCENARIO NOTATION

In our modeling language ADORA [3], scenarios are mainly used for two purposes: (i) Scenarios describe the communication protocol between the modeled system and the actors in the context of the system, i.e. the behavior as it is system-externally seen by the actors. Therefore, scenarios are used to drive simulations. (ii) The evolution of a system model is done by the means of scenarios. Each model version of the system is derived from the corresponding versions of the scenarios. This procedure is embedded in an iterative requirements process described in [11].

A scenario can be hierarchically decomposed using *scenariocharts*. A scenariochart structures scenarios as a tree [2, 3]: each scenario is connected to all its children scenarios. The notation is derived from Jackson diagrams [7], with an extension to include concurrency. Note that scenarios in ADORA are type scenarios, i.e. they describe a set of possible execution paths.

2.1 Scenariochart Syntax and its Graphical Mapping

In contrast to UML, the syntax of the ADORA language is not defined by a graphical metamodel but by an EBNF [14]. ADORA models can be represented as an abstract syntax tree of the EBNF grammar. There exists a mapping between the production rules of the EBNF grammar (more precise: the nodes in the abstract syntax tree) and the graphical elements used to visualize the language [14]. Tab. 1 shows the EBNF grammar for the scenariocharts. For the sake of simplicity, we show an abstract grammar of the syntax which contains only the most important productions for the definition of scenariocharts.

In the following, we will explain how the production rules of Tab. 1 will be mapped to graphical elements. When discussing the elements of scenariocharts, we add in parentheses the name of the corresponding element in the grammar (see Tab. 1) in italics. Fig. 2 graphically exemplifies all the elements used for building a scenario tree. A scenario node *(ScenarioDefinition)* is graphically represented by the shape of an ellipse containing a name which describes the scenario. As an example, the scenario nodes with *Description A, Description B*, etc. can be found in in Fig. 2. A scenario node is of a specific type *(ScenarioType)* which determines how the node and its sub-nodes are interpreted. In Fig. 2 the unattributed node is the so-called root node, which denotes the top element of the scenario tree. Nodes attributed with two parallel lines denote concurrently executed scenarios. Nodes containing a number describe scenarios which are executed in sequence and last but not least, scenarios containing a circle shape denote alternatively executed nodes.

In contrast to Jackson diagrams, there are no iteration nodes in scenariocharts. Instead, iteration is specified by a so-called iteration property and an additional expression describing a predicate which must be true for visiting the node in a further iteration cycle. Every node type can have such an iteration property. Graphically, an iteration property is denoted by an asterisk. The node with *Description B* in Fig. 2 gives an example.

ScenarioDefinition	::=	[**'partial'**] ScenarioType **'scenarios'** ScenarioIdentifier [**'on'** GuardPart] [**'iteration'** Expression] [ScenarioConnections] [TransformationElements] **'end'** **'scenarios'** Identifier
Identifier	::=	Identifier
ScenarioType	::=	(**'alternative'** | **'sequence'** ([< **INTEGER_LITERAL** >] | **'parallel'** | **'root'** [Cardinality])
ScenarioConnections	::=	**'connections'** (ScenarioConnectionDefinition | AssociationDefinition)* **'end'** **'connections'**
ScenarioConnectionDefinition	::=	**'scenarioconnection'** **'to'** QualifiedIdentifier
GuardPart	::=	**'['** Expression (**';'** Expression)* **']'**
TransformationElements	::=	(TransformInput | TransformOutput) (**';'** (TransformInput | TransformOutput))*
TransformInput	::=	(**'transform'** **'input'** [**'cached'**]) (Identifier **'('** [ParameterList] **')'**) [**'over'** Identifier| **'to'** QualifiedIdentifier]
TransformOutput	::=	(**'transform'** **'output'** [**'cached'**]) (Identifier **'('** [ParameterList] **')'**) [**'over'** AssociationIdentifier | **'from'** QualifiedIdentifier]
QualifiedIdentifier	::=	((Identifier (**'.'** Identifier)*))
Identifier	::=	<**IDENTIFIER**>
ParameterList	::=	ParameterDefinition ((**','** ParameterDefinition))*
ParameterDefinition	::=	(Identifier **':'** DataTypeName)
DataTypeName	::=	(Identifier | PrimitiveDataTypeName)
PrimitiveDataTypeName	::=	(**'boolean'** | **'string'** | **'integer'** | **'float'** | **'time'** | **'id'**)

Table 1: The EBNF definition of ADORA scenariocharts.

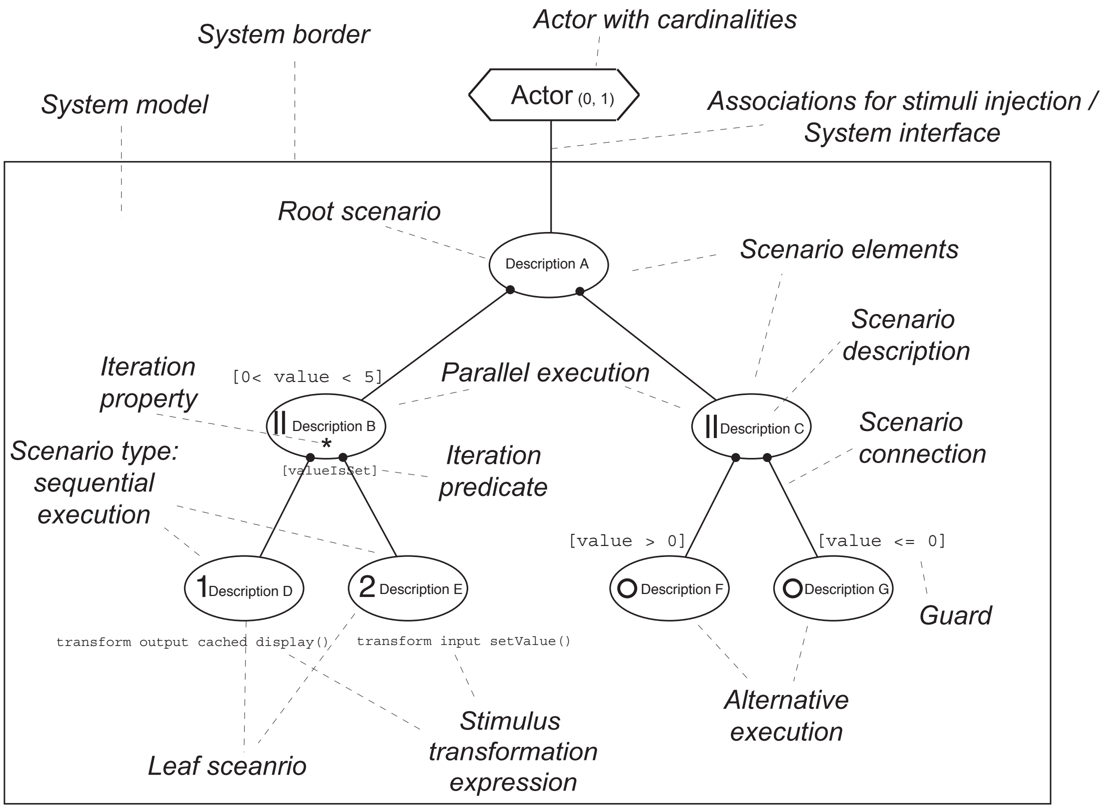


Figure 2: Notation of a scenario tree.

Scenarios are usually connected by an association from the root node to the corresponding actor. This association submits the input stimuli from the actor and sends responses back to the actor. Leaf nodes can contain so-called transformation expressions *(TransformationElements)*, i.e. either *(TransformInput)* elements which specify the kind of input stimuli for a system, or *(TransformOutput)* elements which specify the kind of responses for a system.

Each node can contain a guard *(GuardPart)* which must evaluate to true in order to execute. In Fig. 2 for example, the node with *Description G* has a guard.

Nodes are connected by scenario connections *(ScenarioConnection)*. In ADORA, diagrams have no orientation, i.e. there is no top or bottom in the diagrams. Therefore, in contrast to Jackson diagrams, we have to use directed connections to define an order in the scenario tree. This is done by denoting the parent node with a filled circle at the connection end point.

In ADORA, components (i.e. abstract objects or object sets) are used to describe the structure of a system. They can be decomposed hierarchically. We consider scenarios and components to be complementary in a specification. Scenarios are associated with components (by drawing them inside a component), but the association is not considered to be a a part-of relationship.

In the following two sections, we discuss the static and the dynamic semantics of scenariocharts. We use the following notations as a means for the semantics description. The function *type(s : scenario)* returns the type of the given scenario *s*. The function *iteration(s : scenario)* returns true if the given node *s* has the iteration property set. We denote a parent node as S whereas a sub-node is denoted as S'_i, where $0 \leqslant i < n$ and n is equal to the number of sub-nodes of S. In brief, we just write S for the parent node and S' for the set of child nodes and S'_i for one specific node out of the set S'.

2.2 Static Semantics

In addition to the syntax rules described above, there are several static semantic constraints that must be fulfilled by scenario models. For the sake of completeness, we give a brief description of the most important constraints.

- The scenariochart must be a tree. This can not be guaranteed by the grammar in Tab. 1, because the connections are just referring the connected elements by names. Therefore, there must be an additional constraint which guarantees that no cycles are contained in the scenariochart and that each node has only one parent.
- Scenarios are embedded in components. Let the scenario S be embedded in the component C. For the sake of information hiding [9], a sub-scenario S'_i of a scenario S must be embedded also in the component C or in one of the sub-components of C.
- The nodes on the same level of the scenario tree have to be of the same type as their siblings: If the node S has the sub nodes S', then the type of each sub node *type*(S'_i) has to be equal to the type of its siblings.
- Leaf nodes may not have an iteration property, i.e. if the number of sub-nodes of a given node S is zero, then *iteration(S)* must return false.
- The sequence numbering for nodes on the same level describing a sequence of scenarios must be unique: If the node S has the sub-nodes S' and the type for every sub node *type*(S'_i) equals *'sequence'*, then each sub-node S'_i has to have a unique sequence number.

2.3 Scenariochart Semantics

A scenario tree is executed by traversing each node *(ScenarioDefinition)* of an abstract syntax tree generated by the grammar in Tab. 1 in in-order sequence. The execution of a node S starts when S is entered. From that moment, S is called *in execution* until the node is left. The execution of node S ends if all sub nodes in S' have finished their execution (i.e. their traversal has been finished).

If a *(TransformInput)* element is found during the traversal, the system is waiting for an input stimulus from the user simulating the system. If a *(TransformOutput)* element is found during the traversal, the scenario tree waits for events from the system that can be transformed to a response. The execution of the scenario tree waits until such a stimulus is received. Input stimuli are caught and transformed into an event that could be handled by the behavioral description of the system model. Events that are caught by an output transformation are transformed into a response for the environment.

Transforming input stimuli and responses creates the need for handling continuous environment variables which have to be mapped on variables in the system. This problem is part of the so-called four variable problem [10]. Especially the sampling of environment variables or sending the responses back to the environment often occurs at a point of time when the values can not be processed by the system or the environment respectively. This is the case because the system or the environment is not in the correct state to handle the events or stimuli at the time when they occur. To overcome this problem we introduce the concept of *cached* stimuli which is described by the production rule *(TransformInput)* and *(TransformOutput)* respectively.

A cached stimulus is stored when it occurs. It is consumed by the corresponding transform expression and sent to the system or the environment as soon as the leaf node with the corresponding transform expression is traversed. After reading, the cache is cleared.

A scenario node is only visited if its guard *(GuardPart)* evaluates to true. If no guard is specified, the guard is interpreted as true. If the guard evaluates to false, the visiting of the current node is discarded, i.e. the node is not visited at all and execution proceeds with the next node in the traversal order.

The order of execution is specified by the type of the scenario node *(ScenarioType)*, as well as by the *iteration* property of the parent scenario as follows:

- **root**: This type marks a node as a root node of a scenario tree. A root node is only executed if its guard evaluates to true.
- **sequence**: The sub-nodes S' of the node S which fulfill the predicate *type*(S'_i) *equals 'sequence'*, will be visited sequentially. Therefore, each sub-node S'_i has to specify also a unique integer number, which defines the order in the sequential execution. A sequence scenario node is only executed if its guard evaluates to true. If this is not the case, the next sequence scenario

sibling or the successor of S in the visiting order is executed.

- **alternative**: The sub-nodes S' of the node S which fulfill the predicate *type*(S'_i) *equals 'alternative'* will be handled alternatively. This means that as soon as S is visited, every guard of the alternative scenarios in S' is evaluated. There are three distinct cases: (i) Exactly one guard of in S' evaluates to true. In this case, the corresponding alternative node is visited. (ii) The guards of more than one of the nodes in S' evaluate to true. In this case, the user simulating the model has to choose exactly one of the possible nodes to visit. The chosen node will be visited. (iii) No guard of the nodes in S' evaluates to true. In this case, the node S finishes its execution and the next node in the execution order of the tree is visited.

- **parallel**: The nodes in S' for which the predicate *type*(S'_i) *equals 'parallel'* will be handled in parallel. In this case, the execution is forked. Each sub scenario S'_i, $0 \leqslant i < n$ of the node S visits independently the corresponding scenario sub tree. The concurrent visiting happens in a asynchronous manner. The execution of S is finished (i.e. an execution join is done) as soon as the execution of each node in S' is finished. A node with the scenario type parallel is only executed if its guard evaluates to true. If this is not the case, only the parallel sibling nodes with guard conditions evaluated to true are executed. If no guard of the parallel scenarios of the nodes in S' evaluates to true, S is left without visiting any children.

- A scenario node S can contain an *iteration* property which means that the sub-scenario nodes in S' are visited as many times as the predicate *(Expression)* for the iteration evaluates to true.

3. SCENARIO-DRIVEN SIMULATION

3.1 Simulation Interface

When we talk about simulation, we mean an event-driven, discrete simulation. We do not consider real-time or continuous simulations.

Our system model is composed of hierarchically structured, abstract objects. Each object represents a state and may be further decomposed by other objects and by embedded statecharts. All objects and states together form one joint, hierarchical statechart.

The simulation engine executes the specified system behavior, i.e. on occurring events it performs transitions between states and executes specified actions. As soon as an event appears that cannot be handled, the simulation is interrupted to allow the user to interactively handle this event: whether it shall be received at all, by which object and which actions shall be performed on this event. Afterwards, the simulation continues as usual. More details on our interactive simulation engine can be found in [12].

For each modeled actor, the user simulating the model may create an instance launching the traversal of the connected scenariochart. The simulation stops at leaf nodes allowing the user simulating the model to enter stimuli or receive system reactions. The graphical interface to enter stimuli is automatically built from the transform expressions specified at the leaf nodes of the scenariocharts. For example, the following transform expression:

```
transform input SITZ_H(value : float) over Interface_S2
```

results in the dialog shown in Fig. 3.

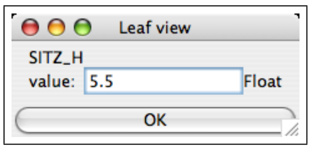


Figure 3: Dialog to enter a input stimulus.

3.2 Example: Car Door Control System

As an example application, we refer to the specification of a control system for car doors given in [6]. A first version of the model is given in Fig. 4. Some scenarios have been modeled already whereas the system is yet unspecified except the three main objects mentioned in the specification.

In a first step, we decide to focus on the seat positioning via the user management switches. There are four switches inside the car to recall four different seat positions (for different drivers). A seat may be adjusted in five different ways: seat height in the front and back, angle of the back, seat distance to the wheel, and tightness of the casing. While being seated, seat adjustments must be done in a comfortable way, i.e. the relaxing adjustments must happen before constrictions take place (in order not to trap the driver), and not more than two movements may be done at the same time. However, when unlocking the car with a radio transmitter (there are two different ones for two drivers), the seat position shall be adjusted as fast as possible to be ready before the driver enters the car. The switches inside the car are connected to the door control system via interface S1, the radio transmitter uses the CAN-bus for communication.

In our example, we start with a simulation of a typical sequence of interactions how the seat adjustment could take place, see Fig. 5. At that time, there is no behavior specified in the system. All objects taking part in the simulation (CAN, S1, User Management and Seat Adjustment) are being played by the user. This means that for each incoming event, the user specifies the corresponding actions that should take place. As we are currently concentrating on the user management, we are not interested in the actual seat positioning. That is why we do not continue to handle the messages in the Seat Adjustment object (marked with a cross in Fig. 5). This may be the focus of further simulations.

Therewith, we have outlined some exemplary behavior for the involved objects. We could either proceed by recording more simulation runs to enrich the system with exemplary behavior. Or we continue with the generalization of the existing behavior.

We have done the latter for the User Management object in Fig. 6. Up to now, it can exactly handle the recorded sequence chart, nothing else. However, for all following simulations, the specified behavior does not need to be played by the user any more. It will be taken from the statechart instead. Only new behavior must be played by the user.

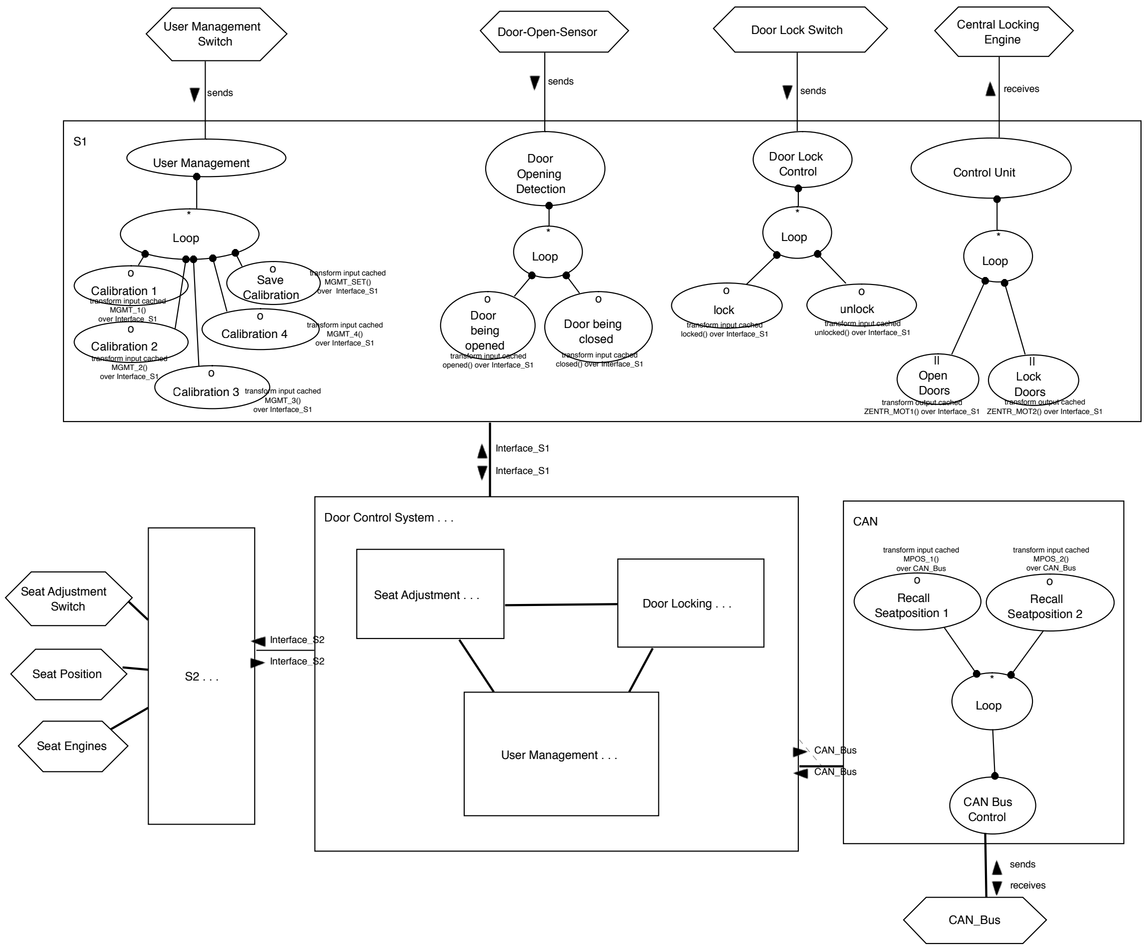

User Management Switch
Door-Open-Sensor
Door Lock Switch
Central Locking Engine
sends
sends
sends
receives
S1
User Management
Loop
Calibration 1
transform input cached MGMT_1() over Interface_S1
Save Calibration
transform input cached MGMT_SET() over Interface_S1
Calibration 2
transform input cached MGMT_2() over Interface_S1
Calibration 4
transform input cached MGMT_4() over Interface_S1
Calibration 3
transform input cached MGMT_3() over Interface_S1
Door Opening Detection
Loop
Door being opened
transform input cached opened() over Interface_S1
Door being closed
transform input cached closed() over Interface_S1
Door Lock Control
Loop
lock
transform input cached locked() over Interface_S1
unlock
transform input cached unlocked() over Interface_S1
Control Unit
Loop
Open Doors
transform output cached ZENTR_MOT1() over Interface_S1
Lock Doors
transform output cached ZENTR_MOT2() over Interface_S1
Interface_S1
Interface_S1
Door Control System . . .
Seat Adjustment . . .
Door Locking . . .
User Management . . .
Seat Adjustment Switch
Seat Position
Seat Engines
S2 . . .
Interface_S2
Interface_S2
CAN
transform input cached MPOS_1() over CAN_Bus
Recall Seatposition 1
transform input cached MPOS_2() over CAN_Bus
Recall Seatposition 2
Loop
CAN Bus Control
CAN_Bus
CAN_Bus
sends
receives
CAN_Bus

Figure 4: Modeled scenarios of a door locking system.

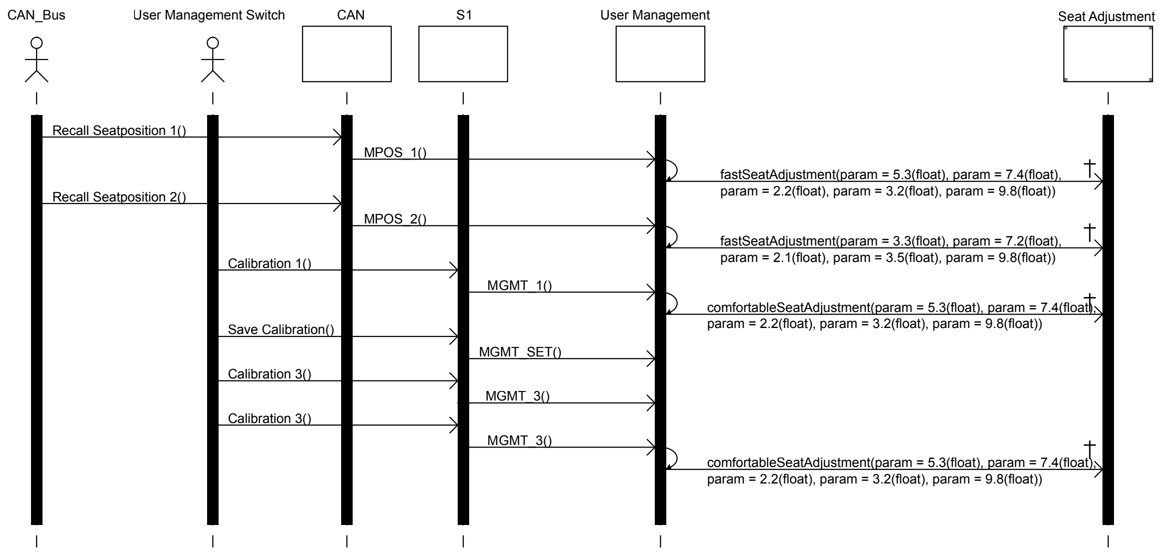

CAN_Bus
User Management Switch
CAN
S1
User Management
Seat Adjustment
Recall Seatposition 1()
MPOS_1()
fastSeatAdjustment(param = 5.3(float), param = 7.4(float), param = 2.2(float), param = 3.2(float), param = 9.8(float))
Recall Seatposition 2()
MPOS_2()
fastSeatAdjustment(param = 3.3(float), param = 7.2(float), param = 2.1(float), param = 3.5(float), param = 9.8(float))
Calibration 1()
MGMT_1()
comfortableSeatAdjustment(param = 5.3(float), param = 7.4(float), param = 2.2(float), param = 3.2(float), param = 9.8(float))
Save Calibration()
MGMT_SET()
Calibration 3()
MGMT_3()
Calibration 3()
MGMT_3()
comfortableSeatAdjustment(param = 5.3(float), param = 7.4(float), param = 2.2(float), param = 3.2(float), param = 9.8(float))

Figure 5: Trace of an exemplary play-through of the user management seat adjustment scenarios (the diagram is automatically generated by the ADORA tool).

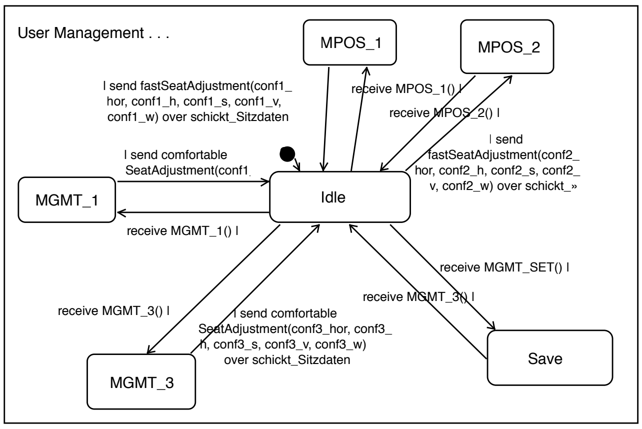

User Management . . .
MPOS_1
MPOS_2
l send fastSeatAdjustment(conf1_hor, conf1_h, conf1_s, conf1_v, conf1_w) over schickt_Sitzdaten
receive MPOS_1() l
receive MPOS_2() l
l send fastSeatAdjustment(conf2_hor, conf2_h, conf2_s, conf2_v, conf2_w) over schickt_»
l send comfortable SeatAdjustment(conf1
MGMT_1
Idle
receive MGMT_1() l
receive MGMT_SET() l
receive MGMT_3() l
receive MGMT_3() l
l send comfortable SeatAdjustment(conf3_hor, conf3_h, conf3_s, conf3_v, conf3_w) over schickt_Sitzdaten
MGMT_3
Save

Figure 6: Generalized behavior of sequence chart in Fig. 5 in object User Management.

This allows to focus on new behavior. Subsequent simulation runs may become stepwise more complex.

Simulations and behavioral generalizations can alternately take place to drive the model evolution to a certain degree of completeness that is desired.

4. RELATED WORK

To the best of our knowledge, there is only one approach which is closely related to ours: Harel et al. developed the Play-Engine [4] that allows to play and test behavior of an incomplete component interactively via a prototypically built user interface. Harel focuses on the interface being developed whereas we are focusing the model being developed. Our main focus lies on adequate support of the requirements engineer in the modeling process. The model can be executed in any state of completeness to drive the further development of the model.

Simulation as a means of validating a model is not a replacement for model checking approaches [8, 1, 5]. Simulation is used earlier in the process to validate the model and drive the further development of the model when it is not yet formal and complete enough to allow model checking.

5. CONCLUSIONS

Sequence charts and statecharts are complementary means for describing behavior that both help describing a system in a clearer way. Sequence charts are better suited for stakeholder discussions, whereas statecharts describe all valid behavior in a generalized way. Tied together in the way described in this paper, they support an evolutionary modeling process for requirements models and ease the way of validating the system model.

Simulations, on the one hand, are a means for producing exemplary interaction with a system and actors. On the other hand, simulations execute the modeled statecharts, thus enabling automatic regression validation and limiting the need for interactive simulations to those parts of the model that the modeler is currently working on. Both activities stimulate each other to further develop the model. The mixture of playing and generalizing is a good means to develop large systems iteratively.

Hence, a modeling language that is able to express incompleteness and semi-formality and a tool supporting a corresponding simulation technique provide strong support for modeling and validating requirements models in an evolutionary style.

6. REFERENCES

[1] S. M. Easterbrook and M. Chechik. Guest Editorial: Special Issue on Model Checking in Requirements Engineering. *Requirements Engineering*, 7(4):221–224, 2002.

[2] M. Glinz. An Integrated Formal Model of Scenarios Based on Statecharts. In W. Schäfer and P. Botella, editors, *Proceedings of the Fifth European Software Engineering Conference*, pages 254–271. Lecture Notes in Computer Science Nr. 989, Springer-Verlag, 1995.

[3] M. Glinz, S. Berner, and S. Joos. Object-Oriented Modeling with ADORA. *Information Systems*, 27(6):425–444, 2002.

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

[5] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[6] F. Houdek and B. Paech. *Das Türsteuergerät – eine Beispielspezifikation [The Car Door Control System – An Example Specification (in German)].* Technical report, Fraunhofer Institut für Experimentelles Software Engineering, Kaiserslautern, 2002.

[7] M. Jackson. *Principles of Program Design.* Academic Press, New York, 1975.

[8] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs.* John Wiley & Sons, Chichester, 1999.

[9] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 12(15):1053–1058, 1972.

[10] D. L. Parnas and J. Madey. Functional Documents for Computer Systems. *Science of Computer Programming*, 25(1):41–61, 1995.

[11] C. Seybold, S. Meier, and M. Glinz. Evolution of Requirements Models By Simulation. In *Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE'04)*, pages 43–48, 2004.

[12] C. Seybold, S. Meier, and M. Glinz. Simulation-based Validation and Defect Localization for Evolving, Semi-Formal Requirements Models. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, pages 408–417, 2005.

[13] J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'00)*, pages 314–323, 2000.

[14] Y. Xia. *A Language Definition Method for Visual Specification Languages.* PhD thesis, University of Zurich, 2004.