Evolution of Requirements Models by Simulation

Christian Seybold, Silvio Meier, Martin Glinz
Department of Informatics, University of Zurich, CH-8057 Zurich, Switzerland
{seybold | smeier | glinz}@ifi.unizh.ch

Abstract

Simulation is a common means for validating requirements models. Simulating formal models is state-of-the-art. However, requirements models usually are not formal for two reasons. Firstly, a formal model cannot be generated in one step. Requirements are vague in the beginning and are refined stepwise towards a more formal representation. Secondly, requirements are changing, thus leading to a continuously evolving model. Hence, a requirements model will be complete and formal only at the end of the modeling process, if at all. If we want to use simulation as a means of continuous validation during the process of requirements evolution, the simulation technique employed must be capable of dealing with semi-formal, incomplete models.

In this paper, we present an approach how we can handle partial models during simulation and use simulation to support evolution of these models. Our approach transfers the ideas of drivers, stubs, and regression from testing to the simulation of requirements models. It also uses the simulation results for evolving an incomplete model in a systematic way towards a more formal and complete one.

1. Introduction

Validating requirements and removing detected errors as early as possible is quite important both for improving quality and reducing cost in software development. A requirements specification process consists of eliciting requirements from stakeholders, documenting them in an adequate way and then validating them by the stakeholders. This is normally not a linear process, but an evolutionary one, due to two reasons. Firstly, a requirements model usually is not created in a single step for size and complexity reasons. Secondly, requirements are changing as stakeholders bring up new requirements, change priorities, etc.

In order to detect ambiguous, missing and inconsistent requirements more easily, requirements should be written in a formal or at least semi-formal language. However, stakeholders usually do not understand formal notations at all and also need help for understanding semi-formal ones. Prototyping and simulation are two possible ways out of this dilemma. *Prototyping* is expensive, in particular if requirements change, because prototype development has to be done in addition to the requirements modeling effort, and a prototype must continuously be adapted if the requirements evolve. Demonstrating the expected behavior of a system by *simulating* a model of its requirements is much cheaper than prototyping, in particular when the requirements evolve. This is due to the fact that a simulation executes directly on the requirements model and therefore always reflects the latest changes. However, validating a requirements specification completely with simulation requires a complete, formal specification.

In practice, semi-formal models of requirements are preferred over formal ones, due to their better cost/benefit ratio. From a cost/benefit standpoint, it would be optimal to have requirements models with a varying degree of formality, where parts with a high risk of failure can be specified formally, while others are specified semi-formally or informally. Some parts may even not be specified at all, because there is a common understanding between the customers and the developers about these parts[1].

However, we still have the need for validating such models and for validating them early in the process. One would benefit most if errors could be found just when a requirement has been written. As simulation is a powerful means for finding errors, it would be extremely useful if a model fragment could be simulated as soon as it has been written and if the specification process could be accompanied by a continuous validation and re-validation of model fragments. Thus, an interesting research question arises: is it possible to extend the concept of simulating requirements models[2] from complete and formal models to partial and semi-formal ones?

[1] For example, when buying a car, the customer does not need to specify in the contract that the car must be equipped with an engine and four wheels with rubber tires.

[2] In this context, simulation means the execution of a system model. The language in which the model is described must rely on a defined execution semantics. Based on the semantics, a simulator tool can execute the model, either by direct interpretation or by code generation.

To appear. In Proceedings of IWPSE'04: International Workshop on Principles of Software Evolution, Kyoto, Japan, September 6-7, 2004.

In the field of testing, we know that we can test software which is not yet complete by using test drivers and test stubs. In testing, we also have the well-known concept of regression testing for dealing with evolving software.

In this paper, we present a concept for simulating partial, semi-formal requirements models which allows model-based validation of requirements at any stage of an evolutionary process. Our concept is based on carrying over the ideas of drivers, stubs, and regression from testing to the simulation of requirements models.

The remaining paper is organized as follows. In Sect. 2, we describe the language features required for our simulation concept. In Sect. 3, we outline an iterative modeling process in which validation by simulation, model evolution and revalidation is embedded. The technique of simulating partial models is described in Sect. 4. In Sect. 5, we show how to benefit from simulation runs to evolve a partial model towards a more complete and more formal one. Related work is discussed in Sect. 6. Our contributions are summarized and an outlook is given in Sect. 7.

2. Prerequisites

For the approach presented in this paper, we need a modeling language with specific features. In particular, the language must support hierarchical decomposition and provide constructs for modeling incomplete information. The modeling language ADORA [3], together with a recent extension [13], provides these required features. As we demonstrate our concepts with ADORA in this paper, we give a brief introduction to the key features of the language in this section, using a distributed heating control system as an example.

However, our approach would also work with other modeling languages providing the decomposition and partiality features that we need. For example, a properly defined UML 2.0 profile [9] would yield such a modeling language.

Fig. 1 shows a typical ADORA model. In contrast to UML, which basically is a loosely coupled collection of different modeling languages, ADORA uses an *integrated model* which unifies structural, behavioral, contextual, and user interaction aspects in a single modeling framework.

Another major difference between ADORA and other object-oriented modeling languages is that ADORA uses *abstract objects* instead of classes as the basic modeling elements, thus allowing hierarchical decomposition of models in a straightforward way with simple and clear semantics. Decomposition, in turn, yields abstraction and the possibility to visualize components in their context, thus making models both easier to understand and change, i.e. evolve.

Hierarchical structure is modeled by nesting abstract objects or object sets. The hierarchy is a whole-part-hierarchy; the part-of relationships being implicitly given by the hierarchical nesting of objects. *Associations* model information flow. Hence, associations may model directed structural relationships between components as well as sending events from one component to another. *States* and *state transitions* are integrated into the object hierarchy at all places where behavior has to be modeled. Objects are regarded to be a special sort of high-level states. We are using a simplified version of statechart semantics [2]. Events are broadcasted only within the boundary of an object. From an originating object to a destination object, events have to be sent explicitly over associations. User interaction is modeled by *type scenarios* (equivalent to use cases in UML). As interaction is frequently local, the scenarios are embedded in the object hierarchy at the position where they apply. Scenarios can also be decomposed, using an extended form of Jackson diagrams [6] as notation.

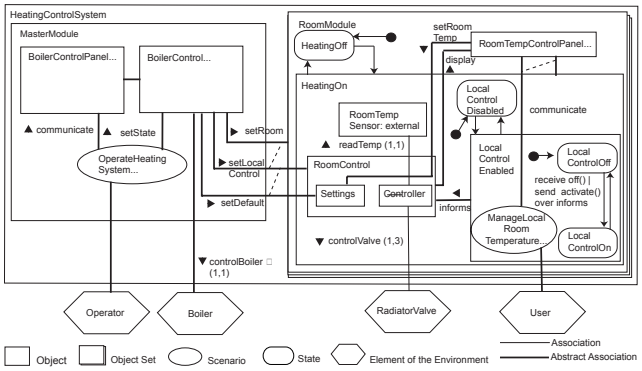

Figure 1. Heating Control System in ADORA

ADORA does not require drawing all model elements in a single diagram. In order to keep diagrams readable, ADORA provides *aspect views* (showing only a particular aspect, e.g. system behavior) and *hierarchical views* (omitting the details of lower level components).

The ADORA language allows both semi-formal and formal modeling. Semi-formal means that some elements of a model do not have formal semantics. A typical example is a state transition (a formal concept) where the triggering condition is given in natural language. Furthermore, ADORA supports *partial models*, i.e. models containing parts that are intentionally incomplete: some parts have not been modeled yet or will not be modeled at all. The difference to unintentional incompleteness is that the incomplete elements are marked as such. Partial modeling is particularly useful in an evolutionary requirements modeling process, where we want to evolve a model in a controlled way through a series of iterations.

In ADORA we have two constructs for describing partial models: the first one is the so-called *is-partial* property which indicates that a component is incomplete (indicated

by three dots following the name). This is especially useful if a system part will still evolve or is incomplete at this time. The second construct is the so-called *abstract association* which is represented as a bold line (e.g. the association from *BoilerControl* to *Settings* in Fig. 1). Abstract associations can be used if the modeler knows that there is some communication between components, but at the time of modeling it is not clear how the concrete communication will look.

Note that ADORA supports not only partial models, but also partial views, using the same notation for both. Partial views are diagrams that do not show all elements that exist in a model (for example, consider a high-level, abstract view of a system), while in a partial model, the model itself is incomplete.

3. A Process for Validating and Evolving Partial Models

In this section, we sketch an incremental process for creating and evolving requirements models which uses simulation as a means both for validating and evolving requirements. The process proceeds through a sequence of increments, each increment consisting of four major steps (Fig. 2). We assume that the process is enacted by requirements engineers who are professionals in elicitation, analysis, modeling, and validation of requirements.

The requirements model can either evolve through a series of requirements-only increments until the requirements specification is considered complete (and will then be used for designing and implementing a system), or the requirements can co-evolve with the design and implementation of the system. In the latter case, each requirements increment is followed by a design and implementation step before proceeding to the next requirements increment. We now describe the four steps of an increment of the process in more detail.

Step 1: Elicit. Elicit requirements using conventional techniques such as stakeholder interviews.

Step 2: Model. Construct a model of the elicited requirements. Try to identify the key sub-problems in the problem to be specified and model components reflecting this problem structure. Details are filled in where the elicitation step provides enough information. As the process is incremental, some parts of the model will deliberately remain incomplete and therefore are marked as such. In addition to this structural model, build a scenario model which describes the interaction between external actors and the system. Of course, there is a feedback-loop between the steps 1 and 2: building the model helps identify missing, ambiguous and contradictory requirements.

Step 3: Validate. Simulate those parts of the model that have been added in the current increment. The simulation

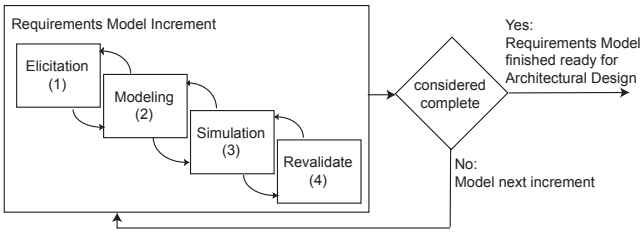

Figure 2. Possible process for evolving the requirements and the architecture of a system

works by executing the scenario models. As the model is incomplete, specific simulation techniques based on stub and driver simulation (see Sect. 4) are applied. The results are used for validating the requirements elicited in the current increment and for correcting the detected errors in the model. Furthermore, the simulation runs are recorded in form of sequence charts. These charts serve two purposes. Firstly, they are used in later increments for regression simulations. Secondly, they provide systematic guidance for evolving the model in the next increment (see Sect. 5).

Step 4: Re-validate. Run simulations for all recorded sequence charts to ensure that previously modeled parts were not affected by the current increment. This is done by comparing the recorded outputs with the actual outputs. If they all match, the re-validation passes and step 4 is finished. A mismatch either indicates an error in the current increment (which has to be corrected) or an intended change in the modeled behavior. In the latter case, the sequence chart must be re-recorded (see Sect. 4.1). After resolving the mismatch, step 4 has to be repeated until the re-validation passes.

4. Partial Simulation

In this section, we present our simulation technique for partial models with the purpose of validation. It is based on the well-known concept of test drivers and stubs [1] or mock objects [7] and today's standard simulation techniques for formal models. Test drivers and stubs are used in the context of software testing to drive unit and integration tests and substitute calls to incomplete components with stubs that have some default behavior. We adapt these terms to simulation units, driver simulation, and stub simulation. Test and simulation drivers have in common that they are utilized for the validation of parts of a model (instead of the complete model); both test and simulation stubs substitute yet unmodeled behavior. The difference is that simulation driver or stubs have not to be coded. Instead, they are played and recorded by interaction with the modeler. This recorded

information is also used for evolving the model towards a more complete and formal one (see Sect. 5) and for re-validation of the model after changes.

4.1. Driver Simulation

Driver simulation is used for simulating a formally specified part of a model.

The *simulation driver* triggers the simulation unit (see below) with events and receives events from the simulation unit. If the modeler validates the model, she or he drives the simulation interactively by inputs that are recorded. Recorded values can be taken to re-run the simulation automatically for re-validation of a changed model. As we suppose the model to evolve continuously, this so-called *regression simulation* is especially important.

The *simulation unit* (SU) is the candidate to be validated. It can consist of a single component or a group of them. Comparable to testing, small units are validated first, then larger units are composed out of them until the whole system forms a single simulation unit. In Fig. 3, an example SU is composed of components *A*, *B*, and *C* (in gray). Component *D* (in light gray) is implicitly included in this SU as it is a child component of *A*, while the parent component *X* and *Y* are not included. Scenarios (ellipses) lie logically outside the system and therefore are never included.

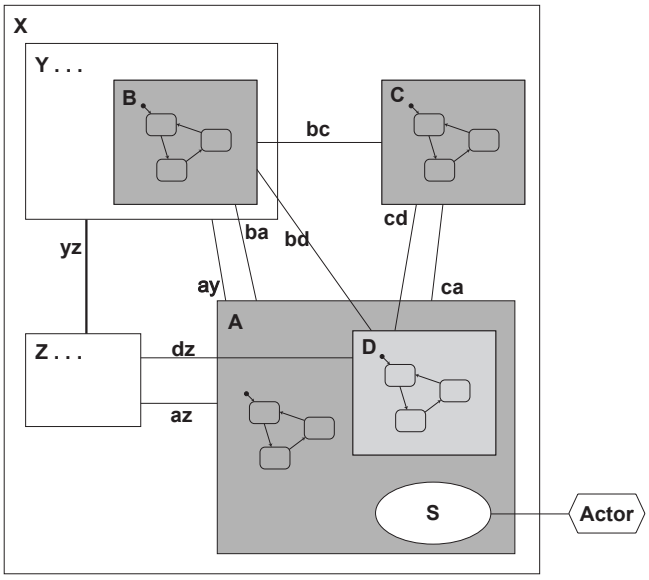

Figure 3. Simulation unit consisting of components A, B, C**, and** D

The possible communication channels to components of a SU form the *interface* between the SU and the simulation driver. There are four ways to communicate: (1) Concrete associations to SU components (*ay, az, dz* in Fig. 3), (2) abstract associations to SU components or parents of them (*yz*), (3) part-of relations between SU components and their parents (*A-X, B-Y, C-X*), and (4) scenario-relations between SU components and contained scenarios (*A-S*).

The interface is used by the simulation driver to enter events into and receive events from the SU. The simulator executes the behavior of the SU by processing the entered events. Both input and output events are recorded in a message sequence chart. The chart contains all events sent between the components taking part in the simulation. One chart represents the trace of one simulation run.

Fig. 4 gives an example of such a sequence chart. On the left side of the dashed line, which represents the interface of the SU, are the components played by the modeler. The components on the right side are executed by the simulation. All event parameters are assigned concrete values. For example, the modeler sends *msgA(25)* from scenario *S* to the simulated component *A*. After processing the internal message *msgB* (it does not leave the SU), *Z* receives *msgC(5)* from *D*. Another input event *msgD* results in *msgE(42)* arriving at *Z*.

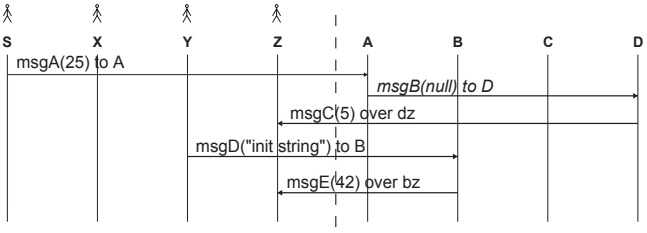

Figure 4. Example trace of a simulation run

A sequence chart which has been recorded in a simulation run is called a *simulation case* (SC), because, in analogy to test cases, it specifies input events and expected results.

The modeler creates new SCs by driving a simulation for a certain SU. She or he can trigger the events that the interface of the SU accepts and she or he has to accept all events from the SU. For each received event, the modeler has to decide whether this output is correct according to how the system should behave. An unexpected output stops the simulation and lets the modeler correct the model. This SC must be re-recorded in this case. After having validated all outputs, the recording of the SC can be stopped.

In regression simulation, the existing SCs are replayed after modification of the model. In this case, the recorded inputs drive the simulation. The received outputs are verified by comparing them to the recorded outputs. The SC passes if recorded and received outputs are equal. If the SC fails, this can have two reasons. Either the behavior of

the SU was unintentionally changed, then the problem in the model must be fixed. Or, the behavior of the SU was intended to change. Then, the SC has become useless and must be recorded again so that it reflects the new behavior.

4.2. Stub Simulation

In this section, we extend the driver simulation with simulation stubs so that also partial components can be included in a SU. A *simulation stub* is a partial component included in a SU. The behavior of these stubs has to be substituted by the modeler, similar to driver simulation. Requests to these components are delegated to the modeler who intervenes and plays the desired behavior.

For example, let's assume that the component *D* from Fig. 3 is not modeled yet, which is a typical situation when modeling top-down. When a simulation is performed on the SU *A*, *B*, and *C*, the component *D* must be represented by a simulation stub.

The interface of a simulation stub is defined in the same way as for a SU (see above). Here, the interface of the stub *D* is composed of the concrete associations *bd*, *cd*, *dz* and the part-of-relation *D-A*. The modeler has to control the interface of the stub as well as of the SU which is the same as in Fig. 3.

As soon as an event is sent to a stub, modeler interaction is required. The simulation is paused to let the modeler generate further events. Then, the simulation continues.

Both input and output events are recorded for two purposes. Firstly, modeler interaction can be replaced with a previously recorded set of interactions. This allows the automation of stub simulations as well. Secondly, the recorded interactions help specify the behavior of the simulation stub and thus evolve partial components to complete ones. This is described in the following section.

5. Model Evolution

In this section we present two techniques for evolving a model towards a more formal and complete one which are based on the process and the simulation approach presented in the previous two sections.

5.1. Evolution of Partial Components

After the simulation of a component which has incompletely specified sub-components, we might want to evolve the specification of one of these sub-components into a complete one. With respect to behavior, this means that we have to develop a statechart that models the behavior of this sub-component.

From stub simulation, we already have a set of sequence charts which describe the intended behavior of the sub-component. Principally, we could feed this information into one of the existing algorithms which synthesize state machines form sequence charts [12]. However, such generated state machines are hardly readable for humans and, hence, difficult to extend and adapt manually.

Therefore, we decided to develop a semi-automatic technique where the modeler draws the statechart manually, but with guidance by a tool which suggests the modeler how to proceed. Fig. 5 gives an idea how this technique works. Given a sequence chart (from stub simulation) and a partially modeled statechart, the tool determines that the first two events of the sequence chart can be handled by the existing statechart fragment, while the third event can't. Hence, the tool proposes the modeler to insert a transition into the statechart which handles this event. The modeler decides the location where to insert the transition and whether it will lead to a new state or to an existing one. So the resulting layout is determined by the modeler.

This procedure is repeated until the statechart is able to handle all events from all sequence charts that were recorded for the component during stub simulation. Such a manual, tool-guided process not only yields statecharts which are more readable than automatically generated ones. It also supports the modeler in finding errors or missing events in the sequence charts from which the statechart was derived.

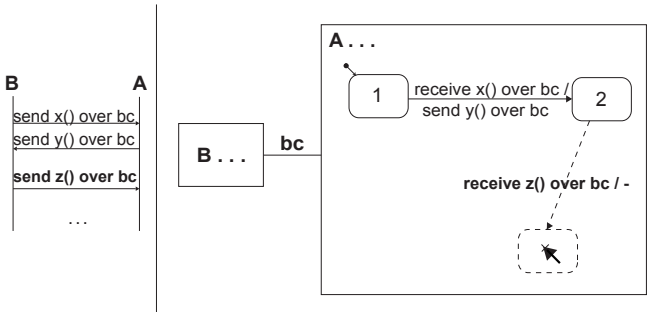

Figure 5. Semi-automatic generation of a statechart from a sequence chart

5.2. Evolution of Abstract Associations

As described in Section 2, we use *abstract associations* for modeling incompletely specified information flows between components. When evolving a partial model towards a more complete one, abstract associations will gradually evolve into *concrete* ones.

We support this kind of evolution also with a semi-automatic, tool-supported technique. The tool compares the

actual need for communication between components due to event flows between statecharts with the currently modeled associations (both abstract and concrete ones). From this comparison, the tool proposes where to replace abstract associations by concrete ones or adapt concrete associations to changes in the connected components. More details can be found in [10].

6. Related Work

There are several approaches that aim at the simulation of requirements models for validation purposes. Mostly, they require formal models. We briefly survey those approaches which are most similar to our one.

Labeled Transition Systems (LTS) [8] are utilized to prove safety and liveness properties of formal models. Partial LTS [11] help to identify undefined scenarios based on possible, but unmodeled transitions in a formal LTS by matching pre- and postconditions.

Whittle [12] provides an algorithm for automatic synthesis of statecharts from sequence charts. The resulting statecharts are readable thanks to the use of hierarchical structure. Modifying the statecharts breaks the link to the sequence charts and therewith prevents model evolution.

The Software Cost Reduction (SCR) method provides a simulator tool [5] that allows to validate SCR models by detecting the violation of invariants on execution and watching the behavior when entering scenarios. The models must be specified formally in dictionaries and tables.

The most similar approach is probably the Play-Engine by Harel et al. [4]. They record instance scenarios by playing-in and perform validation steps by playing-out. Existential and universal life sequence charts (LSC) are used as notation. Regression testing is performed by replaying recorded runs. For playing-in and -out, they require a graphical prototype that must be designed first. There is no focus on evolution of partial components.

7. Conclusions

In this paper, we presented a concept for simulating partial, semi-formal requirements models which allows model-based validation of requirements at any stage of an evolutionary process. The approach transfers the ideas of drivers, stubs, and regression from testing to the simulation of requirements models. It also uses the simulation results for evolving an incomplete model in a systematic way towards a more formal and complete one.

Our approach is limited with respect to proving formal liveness and safety properties. Furthermore, we did not focus on an animated model in the context of the modeled application domain yet.

We have already developed a modeling tool in Java that allows to draw and simulate formal ADORA models. The extension of this tool to the simulation of partial models is currently being implemented. We are also working on extensions of our approach towards the integration of further semi-formal properties of models.

Next, we are going to integrate the evolution techniques described in this paper into our tool. This allows us to perform real case studies demonstrating the usability of our approach. We also want to do further research in the field of semi-formal requirements modeling.

References

[1] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, Reading Mass., 1999.

[2] M. Glinz. Statecharts For Requirements Specification – As Simple As Possible, As Rich As Needed. In *Proceedings of the ICSE'02 Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, 2002.

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

[4] D. Harel. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, Berlin, 2003.

[5] C. L. Heitmeyer, J. Kirby, B. G. Labaw, and R. Bharadwaj. SCR*: A Toolset for Specifying and Analyzing Software Requirements. In *Computer Aided Verification*, pages 526–531, 1998.

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

[7] T. Mackinnon, S. Freeman, and P. Craig. Endo-Testing: Unit Testing with Mock Objects. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'00)*, pages 617–622, 2000.

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

[9] OMG. UML 2.0 Superstructure Specification. OMG document ptc/03-08-02. Tech. Rep., Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/2003-08-02, 2003.

[10] C. Seybold, S. Meier, and M. Glinz. Simulation of Semi-Formal Requirements Models as a Means for their Validation and Evolution. Technical Report 2004.02, Department of Informatics, University of Zurich, 2004.

[11] S. Uchitel, J. Kramer, and J. Magee. Modelling Undefined Behaviour in Scenario Synthesis. In *2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools at ICSE'03*, 2003.

[12] 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.

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