Simulation of Semi-Formal Requirements Models as a Means for their Validation and Evolution

Technical Report No. ifi-2004.02

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 from scratch 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 deal with partial models during simulation and how we can 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

Requirements constitute the fundamental basis for all later software development stages, i.e., the design, implementation, test, and maintenance of the software product to be built. Errors in requirements are costly: (1) Ultimately, they lead to products that do not satisfy the needs of their users, (2) errors are the more costly to remove the later they are discovered. Hence, 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 typically 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 is usually 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 typically 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.

Developing a complete formal specification of non-trivial size is so hard and expensive that it is typically neither feasible nor economically beneficial in practice. Moreover, it is extremely hard to produce a specification which is completely formalized right from the beginning. Instead,

a formal model normally will evolve by incrementally formalizing and assembling informal or semi-formal specification fragments.

Consequently, simulation as a means of validation is confined to a very narrow scope: it is applicable only to formal models and typically only at the end of the requirements engineering process.

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?

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. As a prerequisite, we need a requirements modeling language which is capable of systematically dealing with partial and semi-formal models. In our research group, we have developed the modeling language ADORA [5, 24] which supports various degrees of formality in the notation and provides constructs for expressing intentional incompleteness of model elements.

The remainder of this paper is organized as follows. In the next section, we describe the language features required for our simulation concept, using the ADORA modeling language as an example. In Section 3, we outline an iterative modeling process in which validation by simulation of partial models, model evolution and verification is embedded. The technique of simulating partial models is described in Section 4. In Section 5, we show how the results of simulation runs can be used for evolving a partial model towards a more complete and more formal one. In Section 6, we present a short case study. Related work is discussed in Section 7. Finally, we summarize our contributions and give an outlook in Section 8.

[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 [19].

2 Introduction into ADORA

This section gives a brief introduction into ADORA, the way we enforce the consistency of models, a set of important definitions and how we simulate formal ADORA models.

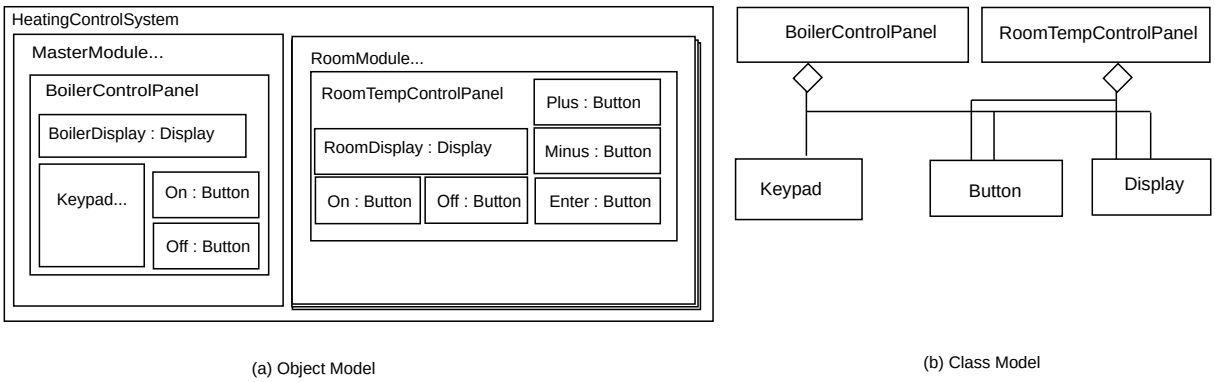
2.1 The ADORA language

The ADORA (Analysis and Description of Requirements and Architecture) language aims in modeling requirements and architectural models as described in [5, 10, 1].

On the first glance, the ADORA language is comparable with other modeling languages like UML [16]. However, there are fundamental differences between conventional modeling languages and ADORA. In the following, we will summarize the most important differences between them by using a *heating control system* as example:

Abstract Objects Class models, as they are used in conventional modeling languages, are inappropriate when modeling one or more objects of the same type, or when nesting objects [10]. In the case of modeling these situations, a lot of information is lost when using class diagrams. This information is especially important during requirements and architectural design phase, because of better understandability of the models, hence modeling languages based on class models are less suited in the mentioned phases. Instead of classes, ADORA uses abstract objects respectively object sets. Abstract objects are representatives for concrete objects, i.e., they define the attributes, functionality and behavior on an abstract level and therefore do not contain an object identity.

Fig. 1 shows an example of abstract objects. In this example, you will see the structural view of a modeled heating control system. Fig. 1(a), will show you the modeled system by using abstract objects, whereas Fig. 1(b) presents the corresponding class models in an UML like syntax. Abstract objects can have a type (following the name by a colon). The figure illustrates that you cannot visualize the information about the number of instances of each class are aggregated by another class which means that the context an ob-



(a) Object Model

(b) Class Model

Figure 1. Abstract Objects vs. Class Models

ject is lost. This problem can not be mitigated by using cardinalities or other workarounds like using inheritance [10].

Hierarchical Decomposition ADORA supports the possibility for hierarchical decomposition. We can recursively decompose objects in objects, but also other elements[3] as for examples states. Elements on a lower level describe the details of the system, whereas elements on higher level describe a more abstract view of the system. This is a good means for structuring large models by modularizing the system. It also provides the possibility for abstraction. You can find an example for hierarchical decomposition in Fig. 2, where *RoomModule* consists of the *RoomTempControlPanel* object, which again consists of other elements, and so on.

Integrated Models Conventional modeling languages like UML [16] consist of a set of loosely coupled sub languages for describing the different views (or also called projections) on a system's model. In such a case, a system's model consists of a patchwork of different submodels, which are not coherently integrated (e.g., behavioral model, structural model, etc.). Compared to a language using *one* integrated and coherent model, the conventional approach results in different problems:

1. The user has to contribute a higher intellectual effort to integrate all the different submodels into one coherent over-all model in his mind.
2. The submodels are more difficult for the communication with the stakeholder, when for example communicating the requirements of the system.
3. The resulting models contain more redundancy due to the fact that the models have to be handled separately.
4. The more redundancy the submodels have, the more consistency problems occur. The consistency has to be ensured by integrity rules that are not directly visible to the user of the language, which results in many cases in complicated constraint rules.

[3]The term elements comprises in the following text all language elements of ADORA, like abstract objects, states, scenarios

The ADORA language deals with the problems described above by using an integrated model, with a set of views:

i. **Base View**: The base view is the structure which is built by abstract objects and abstract object sets, called components. The base view can be hierarchical decomposed as described above. It forms the basis for the views described in the following. Fig. 2 shows the elements of the base view. Abstract objects are drawn as rectangle (e.g., *MasterModule*) and object sets are drawn as stack of rectangles (e.g., *RoomModule*).

ii. **Structural View**: The structural view is the combination of the base view and associations between objects and object sets. These associations are used as communication channels between objects. These channels can be used to send events to a destination component. An example for associations between abstract objects can be found in Fig. 2: *informs* is the association between the *Controller* and the *LocalControlEnabled* object. Associations are always directed binary relationships. Each direction can have a role name. In the *HeatingControlSystem* the *informs* association is an example for an association with one named role and explicitly modeled direction. Another kind of association are abstracted ones. These kind of association is described in more detail below in Subsection 2.2.

iii. **Behavioral View**: The behavior of systems is described in ADORA by a statechart like syntax and semantics [6]. The exact syntax and the behavioral semantics of the statecharts in ADORA is described in [4, 5, 10]. Differences in the syntax of ADORA are that parallel states are not explicitly modeled and

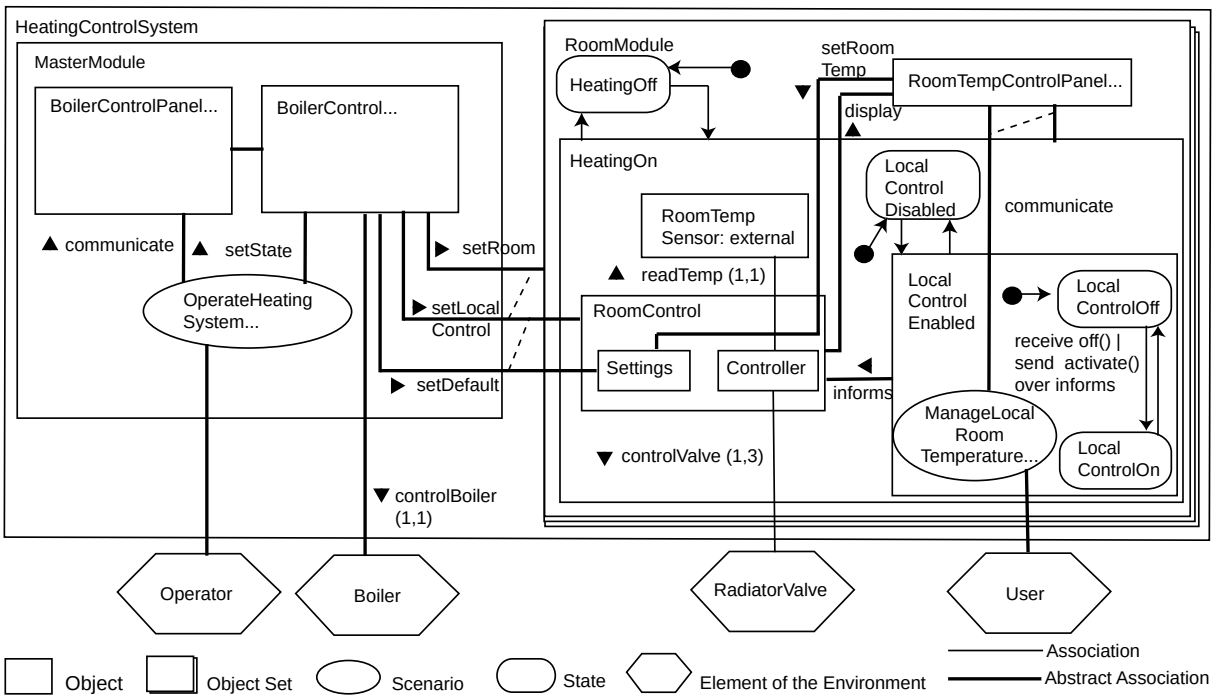


Figure 2. Heating Control System modeled in ADORA (hand-drawn)

that the states can be combined with the object hierarchy. The latter means that states or objects not being directly or indirectly connected by a transition are meant to execute in parallel. The former means that objects will be also interpreted as states if they have an in-going transition from another state or they are designated as a start state.

A transition can be annotated by a description of the firing condition and/or an event, as well as a call of an operation defined in the functional view of an object (see below) or a triggering of an event by a *send* statement. Fig. 2 shows these elements. States are visualized as rounded rectangles, transitions as arrows, start states are designated with special kind of in-going transition (a black dot is the starting point of the transition). States can also be nested.

Note that in Fig. 2 for the sake of simplicity, only transition annotations for one of the transition between *LocalControlOff* and *LocalControlOn* are shown. However, for the semantically correct interpretation of the model, most of the other transitions must also have such an annotation.

iv. **User Interaction View**: Scenario charts [24] are a means for describing the use cases occurring when an actor is communicating with the system. Scenario charts are an extended form of Jackson Diagrams [9] and are fully integrated in the user view of the ADORA language. They describe the possible use cases that can be executed by an actor. We extended the scenario charts in a straight forward approach for simulation by annotating them with conditions (for alternatives and iterations) and a language describing the type of input stimuli sent by the user to the system, respectively sent back from the system to the user [17]. The user view, i.e., the scenarios, describes the protocol used between system and actor to communicate. Hence, the user view plays a key role when simulating a system developed in ADORA.

An example for scenario chart can be found in Fig. 7, where you can see an alarm system with different simple scenarios.

v. **System Context View**: This view describes the system's context by showing the different actors represented by sexangular shapes. Fig. 2 shows an example of the user view. Actors are usually connected to root nodes of scenarios and in the case of simulating the model, they can inject some stimuli into the system and receive some reaction.

vi. **Functional View**: The functional view describes the detailed structure of an abstract object by declaring attributes and operations. The language describing

```
object specification Settings
provides        ActualTemp;
                'Operations to inspect / manipulate control intervals (consisting of start
                time and deired temperature), both default and user-defined';
requires        //nothing
type
        TempIntervals is list of TempInterval;
        TempInterval is structure of (start: Time, temp: Temperature);
        Temperature : typedef y : integer (y > -5 and y <= 40)
attribute
        public ActualTemp : Temperature;     //temperature measured by sensor
        DefaultIntervals  : TempIntervals;   //default temperature settings
        UserSetIntervals : TempIntervals     //user-defined temperature settings

syncoperation CurrentTemp(in time : integer, out ctemp : Temperature)
        pre     1 <= time <= 24 * 60 // minutes
        post    for all i : integer in (i >= 0 and i <= UserSetIntervals.length) (
                  ctemp == UserSetIntervals[i].temp and UserSetIntervals[i].start <= t and
                        (  not (exists j : integer in (j >= 0 and j <= UserSetIntervals.length)
                             (UserSetIntervals[i].start < UserSetIntervals[j].start <= t) ) )
                )
        statements
                ctemp = UserSetIntervals[time];
end CurrentTemp

syncoperation DefaultTemp(in t : Time; out dtemp : Temperature)
         'Same as CurrentTemp, but returns current default value'
end DefaultTemp

operation RevertToDefault()
        post UserSetIntervals@pre == DefaultIntervals
end RevertToDefault

'Settings must also provide operations for setting and deleting intervals and for
  browsing the currently defined intervals.'

end specification
```

Figure 3. Exampled of a detailed Object Specification in ADORA

the functional view is derived from the language ASTRAL [3]. Fig. 3 shows an example describing the detailed structure of the *Settings* object. in Fig. 2.

One coherent model does not mean that all the views will be drawn in the same diagram at the same time. In most cases, this would cause too much cognitive complexity for the user of the language. It rather means that the structural, behavioral, user, and context views can be visually combined freely with each other. These views must be drawn together with the base view. The functional view is a special case and is never be combined with one of the other views.

The integrated model of the ADORA language results in the formulation of less redundancy and enables the use of stronger rules for integrity checking of models. Also, the language is more easy to use, allows a more systematic model construction, fosters a better understanding of the model, and is a means for a better communication with the customer when communicating the requirements of the system.

2.2 Abstraction Mechanism

There are several abstraction mechanisms in ADORA available [10]. One possibility is the hiding of currently not used elements from the views described above. Another possibility is to zoom out in the model, which abstracts from inner content of an component or state of the model. Looking at the Fig. 2, we see there some abstracted components, e.g., the *BoilerControlPanel* or the *BoilerControl*. The hidden content is indicated by the trailing dots of the component name.

If the content of a component *A* is abstracted and this component *A* contains objects or object sets associated to another object/object set *B* located outside *A*, the association has to be abstracted, too. Abstract relationships are hierarchically ordered which is expressed by interrelationships (dashed line) [10] connecting them. An example for an abstracted relationships is the relation with the role name *setRoom* between *BoilerControl* and *RoomModule* in Fig. 2. There exists also an interrelationship between *setRoom*, *setLocalControl* and *setDefault*.

Variable Degree of Formality ADORA supports a variable degree of formality allowing to draw semi-formal or formal models. Semi-formal models contain parts which are described informally by the usage of natural language, possibly containing some kind of hyper links to other (formally defined) elements in the model. An example for a variable degree of formality can be found in Fig. 3. The object specification described in this figure contains informal and formal elements, e.g., the operation *CurrentTemp* is described completely formal whereas *RevertToDefault* is described informally. The informal description is done by means of natural language with links (in italics) to other formal elements in the model. Informal descriptions can also be placed in other elements of ADORA models, for example in the annotation of transitions in the behavioral view.

2.3 Partial Models

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. 2). *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 like.

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.

2.4 Formal and Semi-Formal Models

The definition of the terms formal, semi-formal and partial models are crucial for the understanding of the following content in the paper. This definitions are derived from the definitions in [10]. In the following, we distinguish between the degree of formality in the language and in the model. Fig. 4 gives an overview about the taxonomy respectively the meaning of formality used in this paper. Fig. 5 demonstrates its application.

Formal Language: A modeling language is called formal if the language is both syntactically and semantically precisely defined.

Semi-Formal Modeling Language: A semi-formal language has a well defined syntax, but one or more language elements have an imprecisely defined semantics. A language is called semi-formal language with a variable degree of formality if at least one language element can be described either in a formal or in a semi-formal way. ADORA is a semi-formal language with a variable degree of formality.

Formal Model: A formal model contains only syntactically and semantically precisely defined model elements. A formal model can automatically be interpreted.

Semi-Formal Model: We call a model semi-formal, if it has at least one model element that has an imprecise semantics, respectively if it fulfills one of the following conditions: It contains at least one syntactically correct, but semantically not well defined (semi-formal) construct *or* the model is intentionally incomplete *or* the model contains at least syntactically correct, but semantically wrong elements. We call intentionally incomplete models also partial models.

Partial Model: An intentional incomplete model is also called a partial model. A language should provide incompleteness indicator constructs enabling to distinguish intentional from unintentional incompleteness. Models containing at least one such indicator are called *partial*. ADORA provides two indicator constructs: manually abstracted relationships and is-partial indicators for components (see Sect. 2.3).

In this paper we will concentrate on the simulation of partial models in an evolutionary simulation process. The simulation of syntactically correct but semantically wrong models and models containing semi-formal language elements will be a concern in our future work. Unintentional incompleteness cannot be recognized by tools and therefore is not an issue when simulating requirements models.

2.5 Enforcement of Consistency

The enforcement of consistency is crucial for the simulation of formal and informal models. We use the ADORA integrity constraint language (ICL) [18] to check the consistency of formal and semi-formal ADORA models. The ICL is based on a first order predicate logic and is comparable to the OCL [15]. The predicates formulated in this language can access specific elements of the model. This is used to formulate consistency constraints.

Usually, consistency constraints verify that all formal properties of a model are met before being simulated. We have given an example constraint *AllComponentsAreFormallySpecified* in Fig. 6 (ln. 1-7) that checks for all components (called ADORA objects) in a model (ln. 4) whether the predicate *ComponentIsFormallySpecified* is met (ln. 6). This in turn (ln. 8-20) verifies for a single component several subpredicates, e.g., that it must have at least one start state (ln. 17-19).

However, these constraints will fail for partial components and become useless because required structures are missing. Nevertheless, there are properties that must be met also when simulating partial models. To benefit anyway from consistency checking, the constraints have to be adapted in a way that they tolerate exactly these semi-formal properties which the simulation can handle. Here, we would replace the first mentioned constraint with *AllComponentsArePartialOrFormallySpecified* (Fig. 6, ln. 21-31) that tolerates explicitly partial components (ln. 28). Reuse can be maximized, if constraints are grouped into meaningful units. In this example, we can reuse lines 8-20.

Future extensions of our simulation dealing with more semi-formal properties require also refinement of our constraints to keep consistency checking up to date.

2.6 Simulation of formal ADORA models

For the sake of clarity, we give first some definitions of what we understand under simulation and related terms [19].

Simulation – In the context of RE/SE, simulation is defined as execution of a system model. The language used to express the system model 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.

Animation – We define animation as the visualization of the behavior of the system model. Animation is often based on the (graphical) language/notation, in which the system

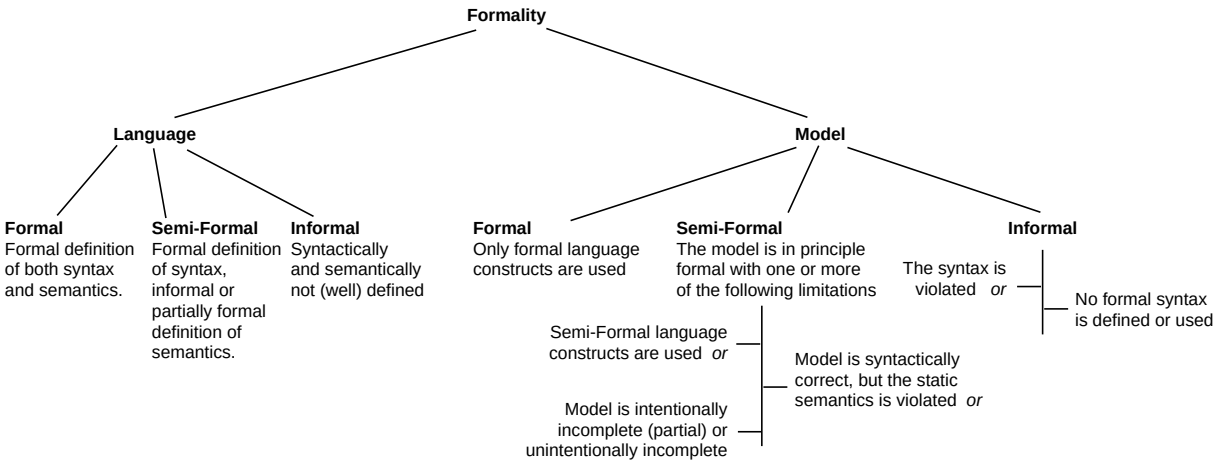


Figure 4. A Taxonomy of Formality

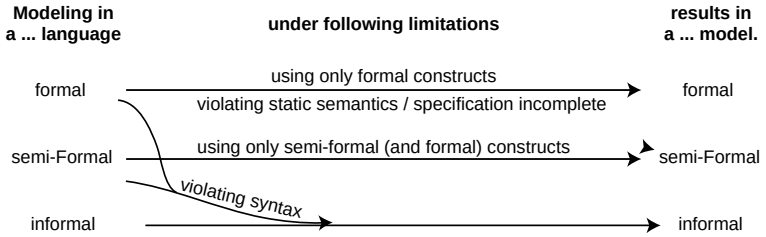


Figure 5. Formality degree in models depending on language and violated constraints.

model is expressed, e.g., by highlighting the model element currently executed. For better convenience, certain tools allow additionally the use of an application-specific graphical user interfaces (GUI), e.g., [14].

Validation – The process of evaluating software during, or at the end of, the development process to determine whether it satisfies the specified requirements [21], i.e., the process of questioning whether the right product is being built.

Verification – (1) The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase; (2) formal proof of correctness [21]. The process of questioning whether the product is being built correctly.

Simulating formal models is state-of-the-art. We have defined the semantics of formal ADORA models [11] that do include not only the behavioral description, but also all other language elements like actors and scenarios [17]. This enabled us to extend our modeling tool [20] to perform interpreted simulations on complete, formally specified, integrated ADORA models. We didn't focus yet on animation of these simulations.

The simulation of a formal ADORA model works as follows. After a model has been loaded, it is verified by running all defined consistency constraints and analyzing the static semantics. On start of the model execution, all start states are entered. Further actions are processed stepwise. For each actor, the user of the simulation can enter the connected scenario trees. They are traversed down to the leaf nodes where stimuli can be transformed into a system event and reactions can be received from the system. While the system is waiting for a user stimulus, the simulation is stopped. That means, the simulation does not run in real-time, but has its own time model with micro and macro steps. The detailed execution model is given in [4, 11, 17].

Stimuli input and system outputs can be recorded in the form of sequence charts. Inputs are required as driver for automatic re-execution of the simulation. The outputs are taken to validate the latest returned results. We call this regression simulation according to regression tests. It enables to detect unwanted changes in the behavior of the requirements model. Details are discussed in Section 4.

Fig. 7 shows a screenshot of an alarm system being simulated in our ADORA tool. The constraint window in the upper right shows that all defined constraints did succeed. The other windows provide a user interface to interact with

```
 1 constraint AllComponentsAreFormallySpecified
 2 on (ar : AdoraRepresentation) is
 3
 4 (for-all comp : AdoraObject in ElementEnumerator.casted
 5           (ar.getAdoraModel().getAllElementsExceptRoot(), AdoraObject.class ) |
 6    predicate ComponentIsFormallySpecified(comp#log)
 7 )
```

```
 8 constraint ComponentIsFormallySpecified
 9 on (comp : AdoraObject) is
10
11  predicate ComponentHasUnambiguousStartState(comp)
12      and
13  predicate ComponentHasProperlyConnectedStartStates(comp)
14      and
15  predicate ComponentHasNoTransitionsCrossingBorders(comp)
16      and
17  // must have at least one start state
18  (there-exists aSubComponent : Node in comp.children() |
19     aSubComponent.getClass().equals(StartState.class)
20  )
```

```
21 constraint AllComponentsArePartialOrFormallySpecified
22 on (ar : AdoraRepresentation) is
24
25 (for-all comp : AdoraObject in ElementEnumerator.casted
26     (ar.getAdoraModel().getAllElementsExceptRoot(), AdoraObject.class) |
27
28    not(comp.isManuallyPartial())
29        implies
30    predicate ComponentIsFormallySpecified(comp#log)
31 )
```

Figure 6. Constraints checking formal and partial properties

the system and with a recorder.

3 A Process for Validating, Evolving, and Verifying 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. 8). We assume that the process is enacted by requirements engineers who are professionals in elicitation, analysis, modeling, and validation of requirements. The requirements engineers closely work together with stakeholders for eliciting and validating 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 as a basis 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. Requirements are elicited using conventional techniques such as stakeholder interviews.

Step 2: Model. The requirements engineer constructs a model of the elicited requirements. She or he tries to identify key sub-problems in the problem to be specified and to 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 the requirements engineer marks these parts to be incomplete. In addition to this structural model, the requirements engineer also builds 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 (by simulation). Simulate those parts of the model that have been added in the current increment. The simulation works by executing the scenario models. As the model is incomplete, specific simulation techniques

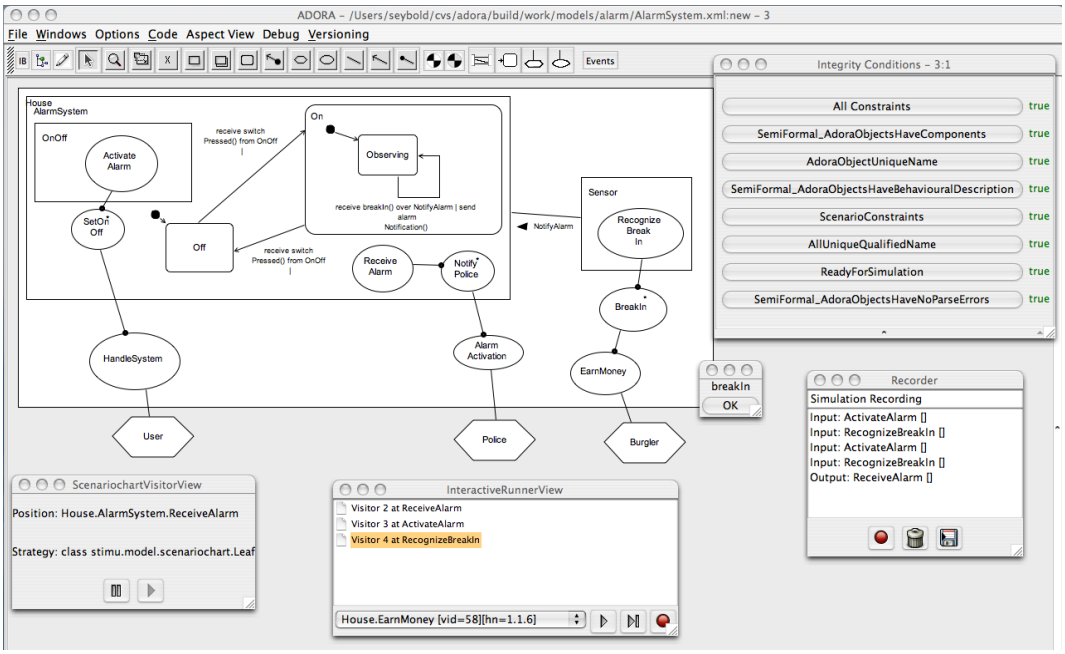


Figure 7. Running simulation of a formally specified alarm system in the ADORA modeling tool.

based on stub and driver simulation (see Section 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 Section 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 indicates a change in the modeled behavior. Either the current increment affected the existing behavior in an unwanted way or the behavior was intended to change. In the first case, the error has to be searched and corrected in the model. In the second case, the sequence chart got outdated and must be recorded again reflecting the new behavior. In both cases, step 4 has to be repeated after the error correction until the re-validation passes.

The process defined above is only sketched in a very rough way. Future investigations will be necessary to refine this process and to identify possibilities for finding good instance scenarios to play. However, the described process gives a hint how the simulation and evolution techniques are embedded. They are described in the following sections.

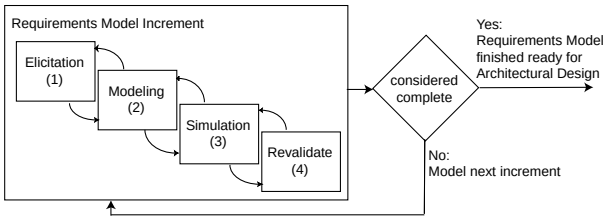


Figure 8. A possible process for evolving the requirements and the architecture of a system

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 [2] or mock objects [12] 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 model parts (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 of the modeler. This recorded information is also used to evolve the model to a more complete and formal one and to verify the model after changes whether it still fulfills the recorded behavior by applying regression simulations. More similarities and differences between software tests and requirements model simulation can be found in Table 1.

In the following two subsections, driver simulation and stub simulation are described in detail. The evolution techniques are presented in Section 5.

4.1 Driver Simulation

The driver simulation deals with formally specified model parts (instead of the whole model). The model itself may be partial.

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 his inputs that are recorded. Recorded values can be taken to re-run the simulation automatically for re-validation of a changed model. This is especially important as we suppose the model to evolve continuously. We call this procedure *regression simulation*.

The *simulation unit* 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. 9, an example simulation unit is composed of components *A*, *B*, and *C* (drawn in gray). If a component belongs to a simulation unit, all its child components are implicitly included. In our example *component D* (in light gray) is implicitly included as *A* belongs to the simulation unit. Parent components (*component X* and *Y*) are not included. Scenarios (ellipses) lie logically outside the system and therefore are never included even when drawn inside an included component.

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

The interface is used by the simulation driver to enter events into and receive events from the simulation unit. The simulator executes the behavior of the simulation unit by processing the entered events. Both input and output events

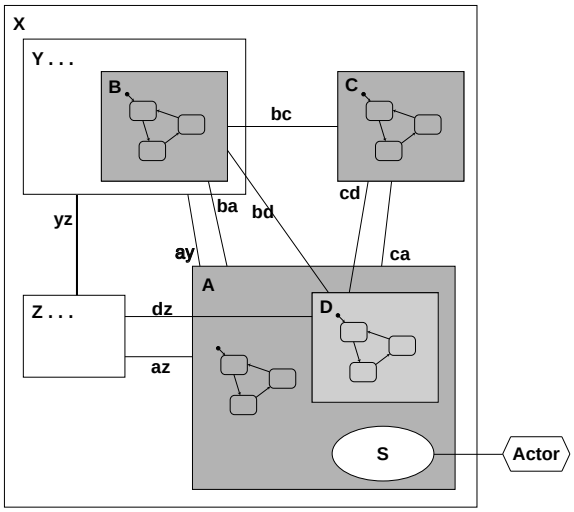


Figure 9. Simulation unit consisting of components *A*, *B*, *C*, and *D* (implicitly).

are recorded in a message sequence chart [16] according to instance scenarios. The chart contains all events sent between the components taking part in the simulation. One chart represents the trace of one simulation run. Recorded sequence charts can be used for regression simulation (see below) as well as for evolving the model towards a more complete and formal one (see Section 5).

Fig. 10 shows how a sequence chart could look like for the example model in Fig. 9. On the left side of the dashed line, which represents the interface of the simulation unit, 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. So, the modeler sends *msgA(25)* from *S* to the simulated *A*. After processing the internal message *msgB* (it does not leave the simulation unit), *Z* receives *msgC(5)* from *D*. Another input event *msgD* results in *msgE(42)* at *Z*.

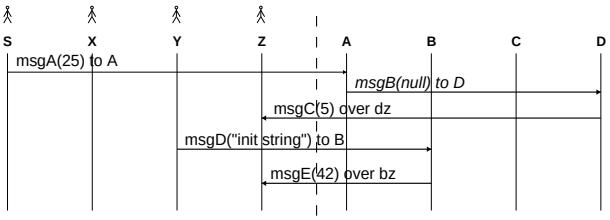


Figure 10. An example trace of a simulation run for the model in Fig. 9.

To be able to validate a certain behavior of the simulation unit, a particular sequence of input events must be en-

Table 1. Similarities and differences between software tests and requirements model simulation

Software Tests	Requirements Model Simulation
The purpose of testing is to verify the software code against the specification.	The purpose of simulation is (a) the validation of the requirements model with the customer and (b) the verification whether model changes did not affect the already validated behavior.
Software tests take place during and at the end of the software implementation.	Simulation for *validation* takes place during requirements elicitation and documentation. *Regression simulation* takes place during and at the end of requirements and architectural refinement.
Test drivers are pieces of software that contain the test cases composed of inputs and expected outputs of the test unit. Actual outputs are verified against the expected outputs.	Simulation driver is either the modeler who enters inputs for the simulation unit and validates the outputs. Or, when the previously recorded inputs drive the regression simulation, the actual outputs are verified against the recorded outputs.
Test stubs (mock objects) are pieces of software that replace uncoded software modules with some simplified default behavior for the test case execution.	Simulation stubs replace partial or unmodeled components with modeler interaction who plays the exact behavior of how the component shall behave during a simulation case execution.
Test cases run automatically by executing software code.	Simulation for *validation* runs interactively. *Regression simulation* runs automatically.
Testing does not guarantee the absence of errors, but reduces the probability of errors. Criteria like functional, statements, path coverage, etc. are taken to determine the quality of tests cases.	For simulation cases the same applies. Criteria like functional coverage of a simulation unit, state, transition, and path coverage can be measured to find good sets of simulation cases.

tered (*msgA* and *msgD*), so that one or several outputs are produced which can be validated (*msgC* and *msgE*). This resulting particular sequence chart is called *simulation case* according to test cases. Each simulation case is intended to validate some particular aspect of the simulation unit. For each simulation unit, one or more simulation cases can exist depending on the amount of functionality the unit provides. A reasonable set of simulation cases should reach a high coverage in the simulation unit for example regarding its functionality, state coverage, transition coverage, etc. If the simulation unit includes all components of the system, the simulation cases can be formed only by instance scenarios.

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

Existing simulation cases can be replayed after the model was changed to verify that they are still valid. This is called regression simulation. In this case, the recorded inputs drive the simulation. The received outputs are verified by comparing them to the recorded outputs. The simulation case passes if recorded and received outputs are equal. If the simulation case fails, this can have two reasons. Either the behavior of the simulation unit was unintentionally changed, then the problem in the model must be fixed. Or, the behavior of the simulation unit was intended to change. Then, the simulation case has become useless and must be recorded again reflecting the new behavior.

4.2 Stub Simulation

As mentioned in the previous section, a simulation unit has to be specified formally for the driver simulation. This is still unsatisfying, as we know that typically there are partial (including unspecified) components in the specification phase, because they are not specified yet (e.g., less important parts) or will not be specified at all (e.g., external components, very low risk parts). Therefore, we cannot wait for a completely specified model before simulating it.

In this section, we extend the driver simulation with simulation stubs so that also partial components can be included in a simulation unit. A *simulation stub* is a partial component included in a simulation unit. 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, we assume that the *component D* from Fig. 9 is not modeled yet and therefore partial, as shown in Fig. 11. This is a typical situation when modeling top-

down. When a simulation is performed on the simulation unit *A*, *B*, and *C*, then 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 simulation unit (see above). Here, the interface of the simulation 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 simulation stub as well as of the simulation unit which is the same as in Fig. 9.

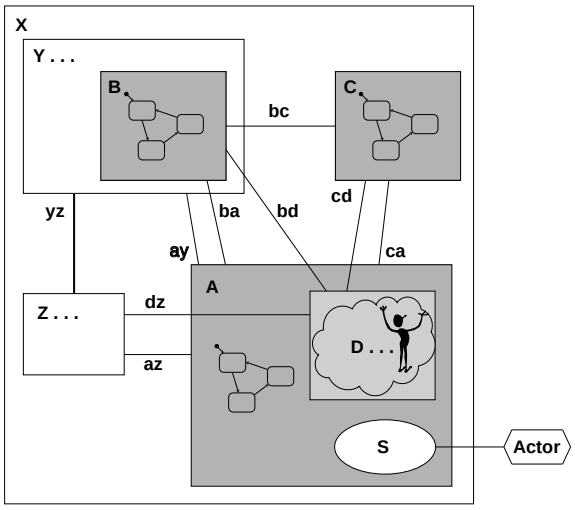


Figure 11. Simulation unit consisting of three components *A*, *B*, and *C*, implicitly including *D*, which is partial and therefore represented as a simulation stub.

As soon as an event is sent to a stub, modeler interaction is required. The simulation is paused to let the modeler send 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 specifying the behavior of the simulation stub and thus evolving partial components to complete ones. This is described in the following section.

5 Model Evolution

In the following, we present two techniques to evolve a model towards a more formal and complete one. Our intention is *not* to eventually arrive at a completely formally specified model, but to support formalization of selected parts. The modeler has to choose the parts for which evolution is reasonable.

In the previous section, stub simulation was used to validate partial components. The resulting sequence charts describe the unmodeled behavior by example. In the following subsection, we present a semi-automatic technique for deriving statecharts that model the behavior which has been captured in sequence charts during previous simulation runs.

Component evolution entails also evolution of their associations. When abstract components are refined to components with concrete behavior, abstract associations must also be concretized to preserve a consistent model. The details of this technique are given in the Subsection 5.2.

5.1 Evolution of Partial Components

Principally, a state machine model could be generated automatically from the recorded sequences. For example, we could apply the algorithm of [23] that takes a collection of sequence charts to generate hierarchically structured statecharts. However, automatic generation has serious disadvantages when further manual changes during evolution become necessary because these statecharts are hardly manually changeable or extensible.

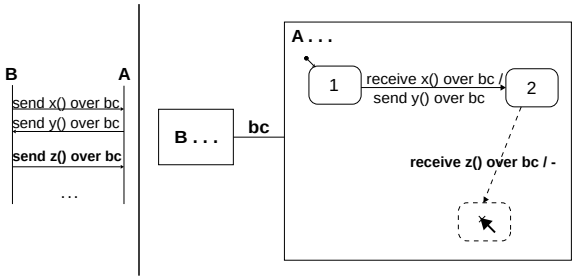


Figure 12. Semi-automatic generation of a statechart from a sequence chart.

Therefore, we offer the modeler a semi-automatic generation technique for evolving components in small steps. The resulting models are better and more valuable with respect to changeability and extensibility than automatically generated ones.

The modeler draws the statechart manually, but with guidance by a tool that suggests what is still missing or what should be drawn next. The whole problem is divided into many smaller, simpler problems. In each step, only one small problem is focused. In our context, the whole problem is to generate a statechart that can handle all recorded sequences. We do this by drawing one transition per step.

For any existing parts of a statechart, it can be easily evaluated which events in a message sequence chart are handled by the statechart. Regarding the example in Fig. 12, the first two events in the chart that were recorded for *component A* during a stub simulation, can be handled by the transition between the existing *states 1* and *2*.

This is the starting point for the transformation. A single step consists of the following procedure: The first unhandled event is transformed into a transition. 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 step is repeated until all events are handled. Between two steps, the modeler is free to perform any other editing operations. After the statechart was changed, the first unhandled event is searched to be the new starting point. As long as not all events are handled, the component is marked as partial. Otherwise, the statechart is finished and the modeler is asked to remove the is-partial indicator.

In the example, the event *z()* is going to be modeled. Our tool would suggest adding the dashed model elements to the existing model in order to handle the next event in the chart. The position of the new state is chosen by the modeler, symbolized in the diagram with a mouse cursor.

The resulting statechart is surely better extensible than a generated one because the modeler has drawn it and knows why she or he has drawn certain states and where to find them. Furthermore, the modeler has the chance to discover errors in a semi-automatic modeling process.

As the presented evolution technique can handle components with any degree of completion, it supports our evolutionary process well.

5.2 Evolution of Abstract Associations

Associations can be set *abstract* to express that some communication between the connected components respectively child components is planned but not modeled yet. It is a placeholder for future concrete associations. In diagrams, abstract associations are drawn as thick lines. In contrast to that, a *concrete association* shows that the two connected components actually can communicate with each other, i.e they can send / receive events over this association to each other.

Note that associations can also be used to express a more general relationship of two components. Communication via events is just one particular application of associations. But for refinement, we focus on these kind of associations.

In the example given in Fig. 13, the *abstract association bv* abstracts from a future *concrete association cy*. *Bv* superposes *cy* (indicated by dashed interrelationship).

The aim is to replace abstract associations with concrete ones when evolving a model. Both kinds of associations can get inconsistent with the modeled behavior of the connected components. An abstract association can get outdated when the behavior of its connected components or child components is modeled, and, hence, the abstract relationship should be replaced with the corresponding concrete associations. A concrete association can get outdated

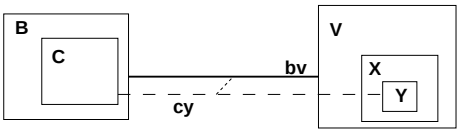


Figure 13. Hierarchy of concrete and abstract associations.

if components are restructured, so that the connected components do not communicate any more directly with each other, but child or sibling components do.

We do not consider the evolution of part-of-relations to parent components and child components, because they are not explicitly modeled using associations.

Whether a component communicates with another component can be easily found out by investigating its modeled behavior. The statements "*send event over association*" and "*receive event over association*" indicate communication over the specified associations. They appear in the modeled statechart of a component in the transition descriptions. For a consistent model, the following rules must apply:

- For all used association names in behavioral descriptions of components, the corresponding concrete association must exist.
- For each concrete associations, there must exist at least one send statement on one side and one receive statement on the other.
- There may be at most one abstract association between each pair of components.
- For each abstract association, at least one of the connected components or child components must be partial.

The consistency can be checked by calculating the required associations for each pair of components based on their modeled communication and comparing them to the actual modeled associations. Table 2 shows the possible cases that can occur. The upper half refers to the case that communication between two particular components is modeled, therefore a concrete association should exist. The left column shows whether there exists a concrete association, the middle one whether there exists an abstract association on this or a higher level, and the right one which action can be performed to attain a consistent model again. If several actions are possible, the modeler can choose. The lower half of the table refers correspondingly to the case where no communication is modeled and therefore no concrete association should exist.

Assuming in the example of Fig. 13 that the concrete association *cy* does not yet exist and the behavioral description of *C* and *Y* communication is modeled, cases 3 and 4

Table 2. Comparison between modeled communication and associations ($\surd$ = yes, – = no, * = doesn't matter).

	Exists concrete association?	Superposed by abstract assoc.?	Action to solve mismatch
Communication is modeled			
1	$\surd$	–	No changes
2	$\surd$	$\surd$	Ask to remove abstract assoc.
3	–	$\surd$	Concretize abstr. association
4	–	*	Insert concrete association
Communication is *not* modeled			
5	–	*	No changes
6	$\surd$	–	Abstract concrete association
7	$\surd$	*	Remove concrete association

in Table 2 would match. Which action should take place depends on the modeler's intention. If *bv* did abstract only from *cy*, then it should be concretized, i.e., the abstract association is replaced by a concrete one (case 3). If more communication is intended to take place between *B* and *Y* or between their child components, then the abstract association must be kept to show this intention. Instead, a new concrete association must be inserted (case 4).

6 Case study: Heating Control System

In this section, we apply the presented process to sketch the development of a part of the *heating control system* (Fig. 2) by using the simulation and evolution techniques. In reality, the process would actually require more iterations. We summarize here some interesting steps.

Suppose a customer interview yields the following specification: *"The heating control system consists of a master module and several room modules. The master module allows the operator to switch the system on and off and to request and adjust the default temperature for each room. Furthermore, the master module controls the connected boiler to heat if the water temperature falls under a fix threshold value of 60 degrees Celsius. Each room module controls the opening and closing of the radiator valve in a way that the room temperature is getting close to the room default temperature or to an individual temperature if set by the room user."*

We start with the identification of actors and the extraction of the type scenarios to draw scenario charts which are derived from Jackson Diagrams [9] and extended with parallelism (||), and numbered sequence (1, 2, ...). Next, we model the structure of the system by correlating the mentioned components. Intentionally incomplete components are marked as partial (three dots). Planned communication is indicated with abstract associations (bold lines). Assigning the scenariocharts to the component structure results in Fig. 14.

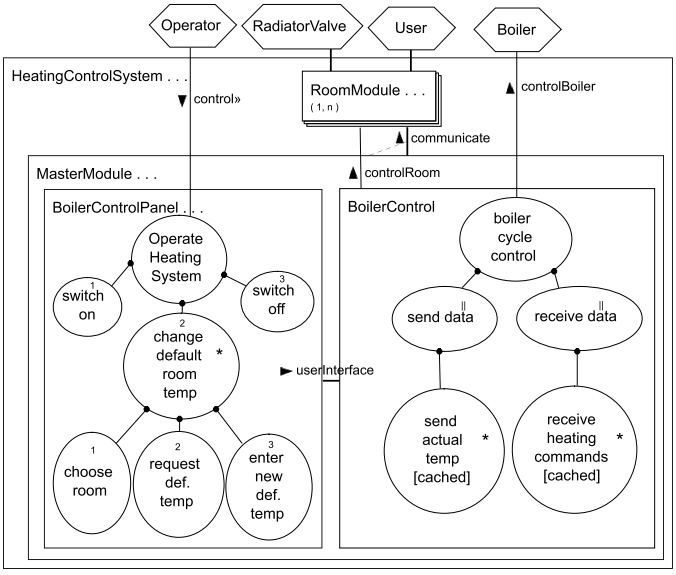


Figure 14. Structural and user aspects of the *Heating Control System* (tool generated).

For further refinement, we concentrate on development of the *MasterModule*. We can now perform a stub simulation recording typical instance scenarios for the *Operator* and *Boiler* actors. All components participate as stubs and are therefore under control of the modeler. Three possible simulation cases are shown in Fig. 15.

With these recorded sequences, we can evolve the *BoilerControl* to cover this behavior (Fig. 16). Still we want to keep the is-partial indicator as we expect more details to be modeled here. The other components can be evolved accordingly.

In the meantime, some associations have got outdated and we have also to evolve them. The association *userInterface* is converted into a concrete one, a new concrete association *controlRoomTemp* between *BoilerControl* and *RoomModule* is inserted and the abstract *controlRoom* is kept, as we expect further concrete associations there. The high level association *communicate* is removed, as we don't expect communication on this level any more, see Fig. 16.

Finally, we simulate the current system together with the customer to validate the modeled behavior before continuing with the *RoomModule*. While further refining, the recorded sequences charts are often revalidated to be sure that the existing behavior is still valid.

7 Related Work

There exist quite a large number of approaches that aim at the simulation of requirements models for validation pur-

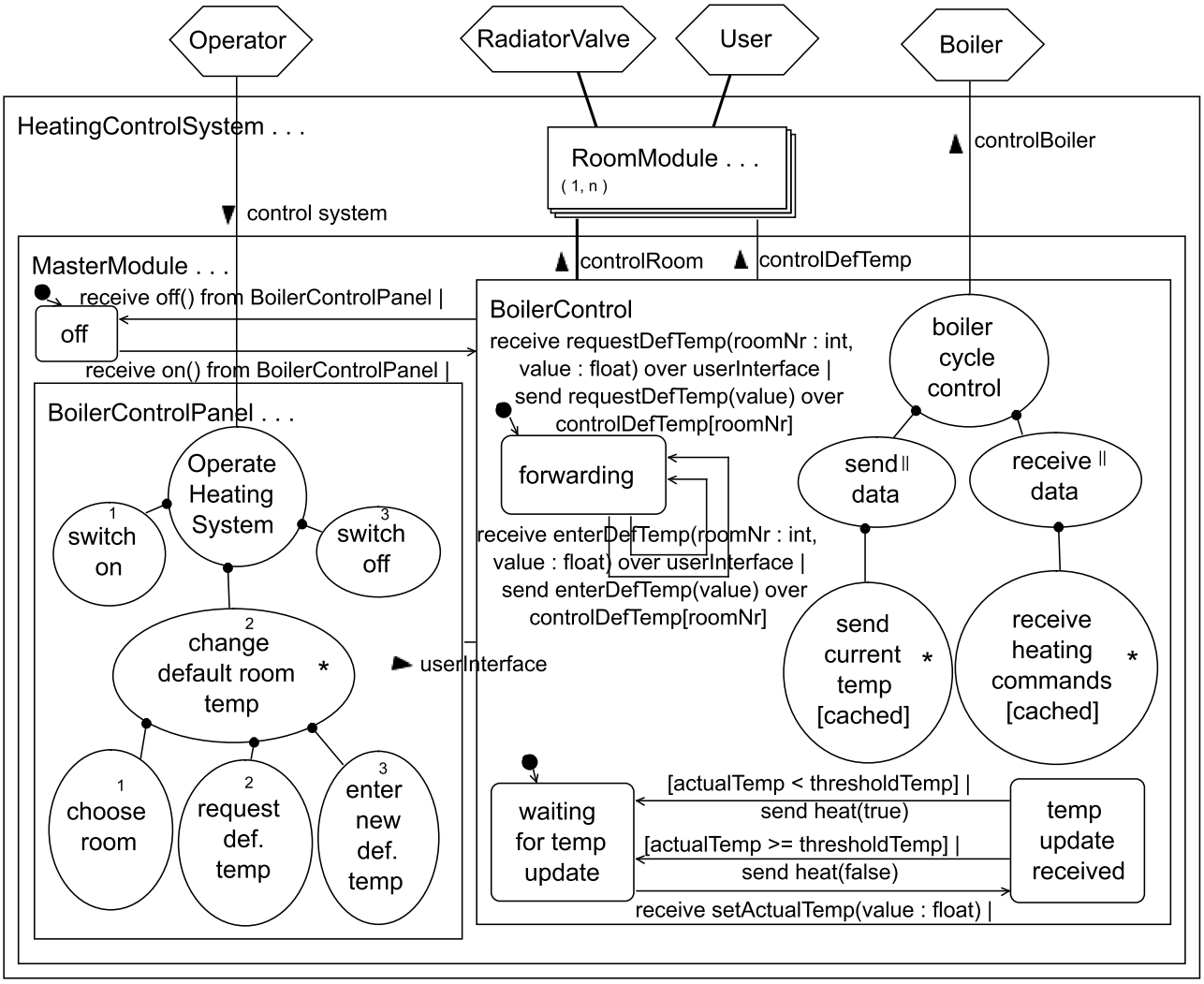


Figure 16. The *Heating Control System* after refinement containing behavioral aspects in *Boiler Control* (tool generated).

poses. Mostly, a formal model is required to perform simulations. We do not know any approach that uses simulation as a means for model evolution. Below, we briefly survey those approaches which are most similar to our one.

Labeled Transition Systems are used in the approach of Magee & Kramer [13] to prove safety and liveness properties of formal models. Partial Labeled Transition Systems [22] help to identify undefined scenarios based on possible, but unmodeled transitions in a formal LTS by comparing pre- and postconditions.

Whittle & Schumann [23] provide an algorithm for automatic synthesis from sequence charts to state machines. 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. Furthermore, the sequence charts must be enhanced with additional information to improve identification of common states.

The SCR method provides a simulator tool [8] 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. This approach aims mainly on model checking techniques; semi-formal models are not supported.

The most similar approach is probably the Play-Engine by Harel et al. [7]. 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 replay-

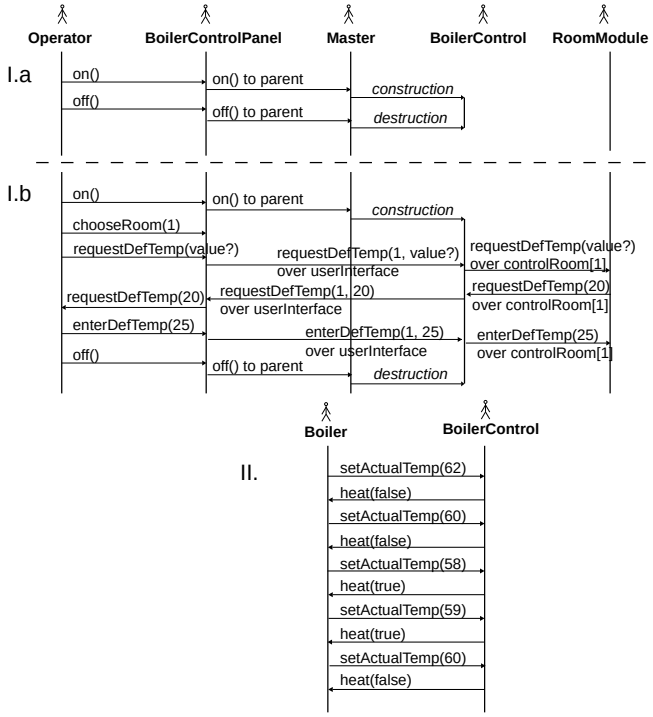


Figure 15. Three simulation cases showing instance scenarios of the *Operator* and the *Boiler*.

ing recorded runs and verified whether existential LSCs are still successful and universal charts are not violated. For playing-in and -out, they require a graphical prototype that must be designed first. There is no focus on evolution of partial components.

8 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] S. Berner. *Modellvisualisierung für die Spezifikationssprache* ADORA *[Model visualization for the specification language* ADORA *(in German)]*. PhD thesis, University of Zurich, 2002.

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

[3] A. Coen-Porisini, C. Ghezzi, and R. A. Kemmerer. Specification of realtime systems using ASTRAL. *IEEE Transaction on Software Engineering*, 23(23):704–736, 1997.

[4] 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.

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

[6] D. Harel. A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

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

[8] 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.

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

[10] S. Joos. ADORA-*L - eine Modellierungssprache zur Spezifikation von Software-Anforderungen [*ADORA-*L – A modeling language for specifying software requirements (in German)]*. PhD thesis, University of Zurich, Zürich, 1999.

[11] J. Krebs. Entwicklung einer Ausführungsmaschine für die Simulation / Animation von formalen ADORA-Modellen [Development of an execution machine for the simulation / animation of formal ADORA models (in German)]. Diploma Thesis, University of Zurich, To appear in 2004.

[12] 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.

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

[14] J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical animation of behavior models. In *International Conference on Software Engineering*, pages 499–508, 2000.

[15] OMG. UML 2.0 Object Constraint Language. Final adopted specification. OMG document ad/2003-10-14. Tech. Rep., Object Management Group, 2003. http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf.

[16] 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.

[17] F. Schenk. Konzeption und Umsetzung einer Stimuli-Ein-/Ausgabeschnittstelle für die Simulation von Anforderungsmodellen [Concept and implementation of a stimuli input/output interface for the simulation of requirements models (in German)]. Diploma Thesis, University of Zurich, To Appear in 2004.

[18] N. Schett. Konzeption und Realisierung einer Notation zur Formulierung von Integritätsbedingungen für ADORA-Modelle [A Notation for integrity constraints in ADORA models – Concept and implementation (in German)]. Diploma Thesis, University of Zurich, 1998.

[19] R. Schmid, J. Ryser, S. Berner, M. Glinz, R. Reutemann, and E. Fahr. A Survey of Simulation Tools for Requirements Engineering. Technical Report 2000.06, Department of Informatics, University of Zurich, 2000.

[20] C. Seybold, M. Glinz, S. Meier, and N. Merlo-Schett. An effective layout adaptation technique for a graphical modeling tool. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 826–827, Piscataway, NJ, 3–10 2003. IEEE Computer Society.

[21] The Institute of Electronical and Electronics Engineers. *Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990*. IEEE Computer Society Press, 1990.

[22] 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.

[23] 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.

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