Simulation-based Validation and Defect Localization for Evolving, Semi-Formal Requirements Models

Christian Seybold, Martin Glinz, Silvio Meier

Department of Informatics, University of Zurich, Switzerland
{seybold | glinz | smeier }@ifi.unizh.ch

Abstract

When requirements models are developed in an iterative and evolutionary way, requirements validation becomes a major problem. In order to detect and fix problems early, the specification should be validated as early as possible, and should also be revalidated after each evolutionary step.

In this paper, we show how the ideas of continuous integration and automatic regression testing in the field of coding can be adapted for simulation-based, automatic revalidation of requirements models after each incremental step. While the basic idea is fairly obvious, we are confronted with a major obstacle: requirements models under development are incomplete and semi-formal most of the time, while classic simulation approaches require complete, formal models. We present how we can simulate incomplete, semi-formal models by interactively recording missing behavior or functionality.

However, regression simulations must run automatically and do not permit interactivity. We therefore have developed a technique where the simulation engine automatically resorts to the interactively recorded behavior in those cases where it does not get enough information from the model during a regression simulation run.

Finally, we demonstrate how the information gained from model evolution and regression simulation can be exploited for locating defects in the model.

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. In traditional linear processes, the complete requirements specification was written in a single phase and the specification was validated at the end of this phase. Typically, the validation was done with manual techniques such as reviews.

However, with today's iterative and evolutionary processes, such a simple validation process is no longer appropriate. In order to identify problems and faults early, validation should be performed early and frequently, i.e. at least after each iteration step. On the other hand, manual validation is expensive and time-consuming. Hence, it cannot be performed so frequently. The problem is aggravated when we specify requirements by formal or semi-formal models and develop these models in an interactive and evolutionary style. Such models typically evolve in a sequence of small, but frequently occurring incremental modifications. As every small modification of a requirements model can unintentionally destroy required properties that held in the model prior to the modification, a model should be revalidated after each incremental step. However, this is not feasible without automation.

In this paper, we present an approach that validates requirements models by simulating them. Revalidation after incremental modeling steps is done by regression simulation, i.e. automatically re-executing previously recorded simulation runs. While this idea is fairly obvious, we have to deal with a major obstacle: requirements models under development are incomplete and semi-formal most of the time, while classic simulation approaches require complete, formal models.

The main contribution of this paper is extending the concept of simulation from formal models to semi-formal ones. We interactively specify missing behavior or functionality in a simulation case, but nevertheless allow regression simulations to run automatically. The latter is achieved by letting the simulation engine automatically resort to the interactively recorded behavior in those cases where it does not get enough information from the model during a regression simulation run.

As an additional benefit, simulation execution traces as well as evolution information can be used for visualizing failed regression runs and for localizing defects in a model. The features described in this paper have been implemented in the ADORA [7] modeling and simulation tool.

The remainder of the paper is organized as follows: In Section 2.1, we describe some prerequisites about simulation, the modeling language we use and the specification

To appear in: Proceedings of the 12th Asia-Pacific Software Engineering Conference, Taipei, Taiwan, December 2005.

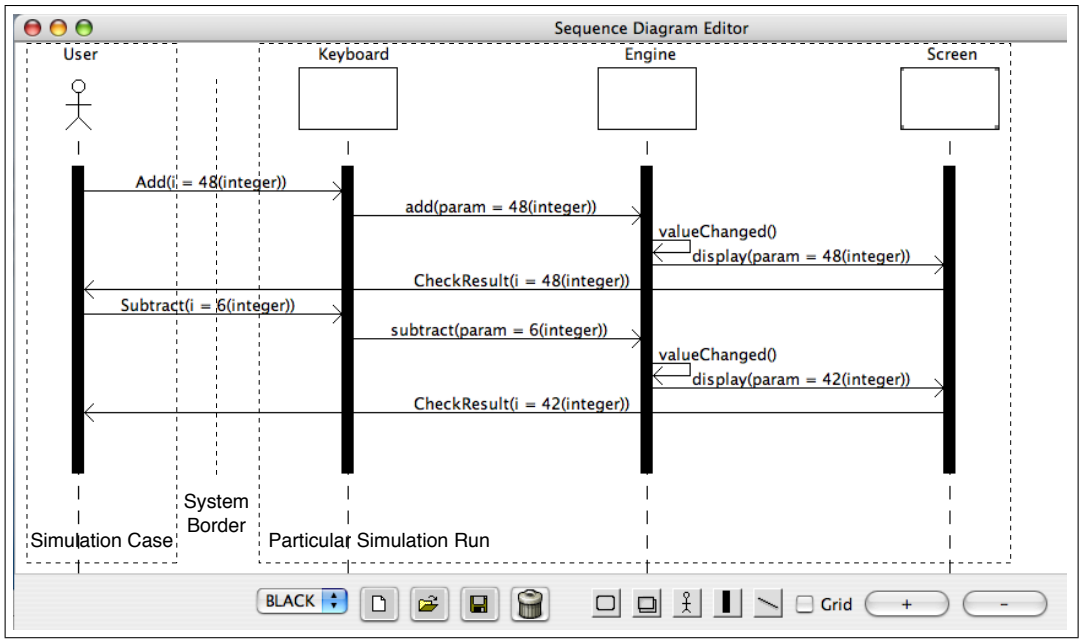

Figure 1. An example simulation run visualized as a sequence diagram for the model in Fig. 2.

process we assume. In Section 3, we show how a simulation can be performed interactively on an informally and incompletely specified model. In Section 4, we present how a regression simulation can be used to revalidate also incomplete specified models. In Section 5, we demonstrate how the results of failing regression simulation runs can be analyzed in order to locate the problems that caused the failures. Related work is described in Section 6. The paper concludes with a summary of the main results and some remarks about future work.

2 Prerequisites

2.1 Simulation of requirements models

Simulation of formal requirements models is a well-known technique for validating requirements [8, 12, 10]. The purpose of a simulation is to validate the reactions of the modeled system to given stimuli, which are entered into the simulation by the modeler via the model interface.

In order to do simulations systematically, we define *simulation cases*. Every execution of a simulation case yields a *simulation run*. A simulation case can be regarded as an instance scenario of the communication protocol between an actor and the system. When a simulation case is executed, one determines whether the scenario behaves as intended and yields the desired results. In order to achieve a comprehensive validation, the intended behavior of a model must be sufficiently covered by simulation cases.

A simulation case consists of a set of input stimuli and system reactions. It is constructed by an interactive simulation run when a required functionality of the specified system is validated. Usually, a modeler would do this together with a stakeholder.

A simulation run can be represented as a sequence diagram, see Fig. 1. The actor lifeline in this diagram represents the external actor. The messages sent and the reactions received by the actor represent the scenario of required system behavior given by the simulation case. The other lifelines and the messages they exchange represent the execution trace of the simulation run through the model.

A requirements specification may be complete and formal at the end of a specification process, if at all. Most of the time, however, it will be both semi-formal and incomplete with respect to the modeler's intentions. Semi-formality is meant not as absence of semantics, but as constructive integration of informality and incompleteness in a formal modeling framework. This allows a systematic refinement and evolution of an initially informal and rather incomplete specification model.

A simulation engine for requirements models should support semi-formality, so that validation by simulation becomes applicable also for models that are not yet completely formalized. For that purpose, the simulation engine must provide suitable interaction mechanisms for resolving uninterpretable statements during a simulation run with the help of the modeler.

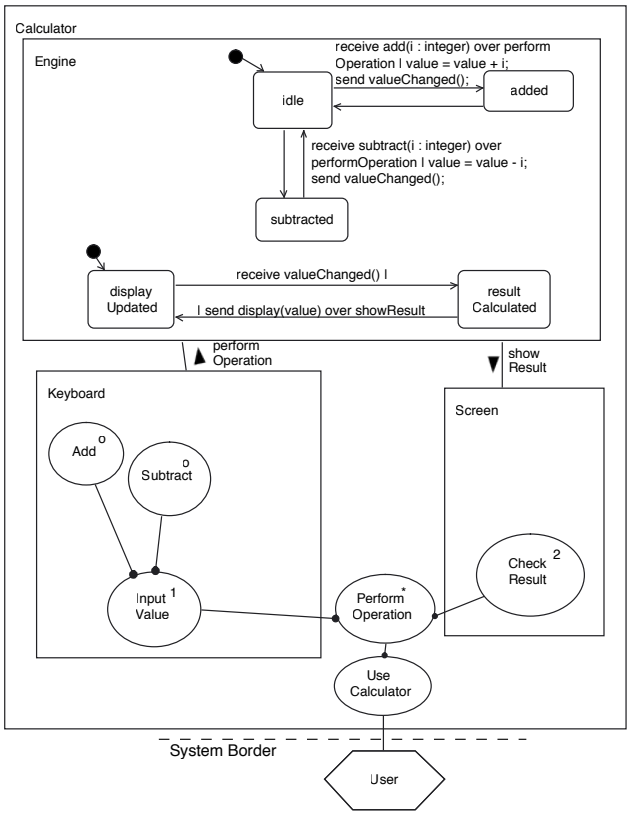

Figure 2. The corresponding simple calculator model to Fig. 1.

2.2 The modeling language

In order to be simulatable, a requirements model must be constructive. In our approach, we are using the modeling language ADORA [7] based on a hierarchy of objects which encapsulate operations, data, states, and scenarios. Operations and data model the functionality of the system. States represent explicit states in which the system reacts to some expected stimuli and ignores others. Scenarios[1] model the required sequence of interaction between a particular actor and the system. Both states and scenarios can be decomposed hierarchically. In contrast to UML, ADORA integrates all these aspects into a single, coherent logical model. The ADORA tool provides mechanisms for visualizing multiple aspects together in a single diagram (Fig. 2) or creating single aspect views of a model. ADORA also provides constructs for expressing intentional incompleteness and, hence, supports a variable, but controlled degree of formality and completeness in a model. This is needed if we want to develop and maintain a requirements model in an incremental and evolutionary style (see Section 2.3).

[1]These are type scenarios, i.e. use cases in UML terminology

Fig. 2 shows a simple ADORA model, which we will use as running example in this paper. The object Calculator interacts with an external actor User. Calculator is decomposed into the objects Engine, Keyboard, and Screen. Its interaction with User is specified in the type scenario Use Calculator. Use Calculator is an iteration of Perform Operation, which in turn consists of a sequence of Input Value and Check Result. Input Value has two alternative subscenarios: Add and Subtract. The object Engine performs the calculation. It consists of two concurrently running statecharts. The first one is responsible for the event reception and the calculation itself. The second one sends a message to the Display with the new result according to the observer pattern.

The simulation of an ADORA model is driven by executing the type scenarios that specify the interaction between the actors and the system. An executing scenario generates events that may trigger state transitions and/or invokes operations on objects. These transitions and operations in turn change the internal state of the system and eventually produce results that are passed back to actors.

Example: Assume that we have elicited the requirements for a Calculator as modeled in Fig. 2. For validating this model, we instantiate the Use Calculator type scenario with a set of instance scenarios that cover the functionality of the model and execute them together with the customer. Fig. 1 shows the execution trace of the scenario that first adds 48 and then subtracts 6. If the customer agrees that this simulation run exhibits the expected functionality and behavior, the execution trace is stored as a simulation case. Additionally, we can also define and execute negative instance scenarios that represent sequences of unintended or forbidden input events.

The selection of instance scenarios is a problem which is analogous to the selection of test cases in black-box testing. Hence, the same techniques are applicable.

Our approach is not confined to ADORA. Alternatively one could use a sufficiently formalized subset of UML. We use ADORA here, because it already has the features we need, whereas with UML, we first had to define and formalize a suitable subset.

2.3 Iterative requirements engineering process

As already discussed in the introduction, simulation is a potential means for automatic revalidation of models in an iterative requirements engineering process. Fig. 3 shows such a process and illustrates how simulation and revalidation interact with the tasks of elicitation and modeling.

This process supports an evolutionary style of eliciting, documenting and validating requirements as well as an evolutionary software development process (where we have a sequence of iterations with each iteration consisting of spec-

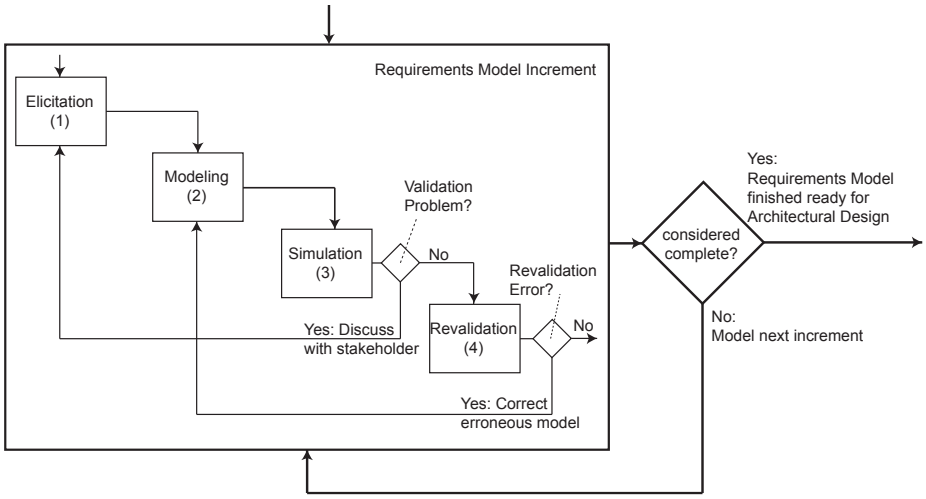

Figure 3. Iterative process modeling requirements increments.

ification, design, implementation, and deployment) [13].

In order to keep track of the evolving artifacts, model versions, simulation cases and simulation runs must be put under configuration control. In our approach, we are using cvs [3] for configuration management.

3 Simulating Semi-formal Models interactively

An interactive simulation run is triggered by the injection of stimuli via the modeled system interface. Actors are connected to scenarios trees that specify the communication protocol with the system, see Fig. 2.

The current approaches to model simulation assume a formal, fully executable model. However, with an incremental, evolutionary style of model development, this assumption is no longer valid: when a requirements specification model is developed this way, it will typically be semi-formal most of the time.

3.1 Incomplete Semi-Formal Simulation

In this section, we extend our simulation approach to incomplete and semi-formally specified models. Incompleteness means that required behavior and/or functionality is missing. Semi-formality means informal parts are embedded in the model structure in a human understandable way; but they are not machine interpretable.

As an example, assume that we want to extend our calculator (see Fig. 2) such that it also can multiply and divide. Fig. 4 shows the result of an incremental modeling step, where we have extended the Use Calculator scenario by two additional alternatives Multiply and Divide. The model of the calculator engine has been adapted partially: multiply is specified informally, divide is not specified.

Now, let's run a simulation case that contains a multiplication and a division. A conventional simulation engine would fail on the multiplication and would ignore the divide event. The latter would lead to unexpected behavior and to a difficult task to locate and correct the problem if we had a real-world size model.

Therefore, we have developed a simulation technique that enables us to detect and handle informally specified or incomplete behavior during a simulation run. If an event is received and there are informally specified transitions outgoing from one of the active states of an object, the simulation is temporarily stopped and an informal transition dialog box (Fig. 5) is displayed. One of the transitions could be suitable to handle the event. By choosing the multiplication transition, a corresponding formal description to receive the multiply event is added to the informal description. Next time in this situation, the event can be handled automatically. If none of the informal transitions match, a new transition can be specified. This is similar to the reception of an unhandled event.

When an unhandled event is detected, the simulation is temporarily stopped to let the user react to this event. A dialog box (Fig. 6 left) presents the user-relevant information about the unhandled event: name, origin and parameter values. S/he may either ignore the message (i.e. let the simulator ignore the event) or accept it. Local variables may be defined to store the given parameter values enabling their further processing. As an unhandled event may indicate an incompleteness, the simulator suggests making the intentional incompleteness explicit by marking the corresponding component as partial (this is a feature provided by our modeling language ADORA).

The next dialog box (Fig. 6 right) allows the user to spec-

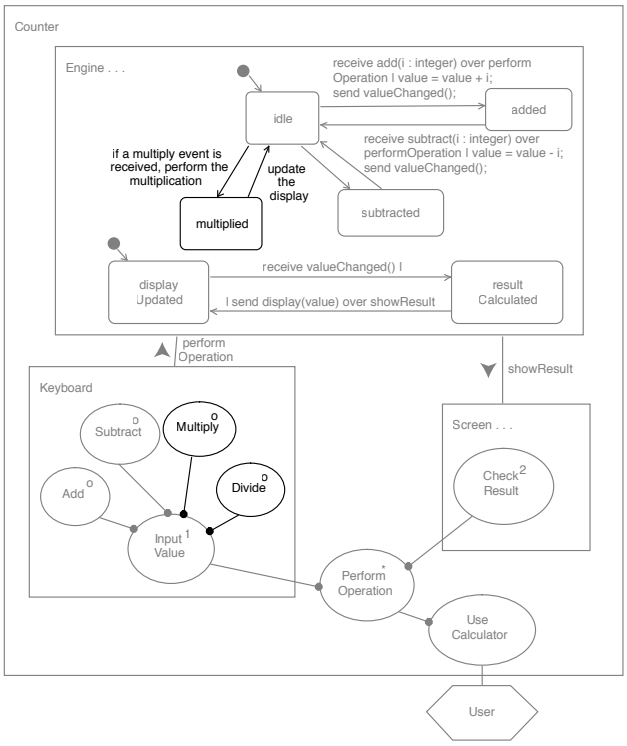

Figure 4. Incomplete and informally specified extended model. New parts are drawn in black.

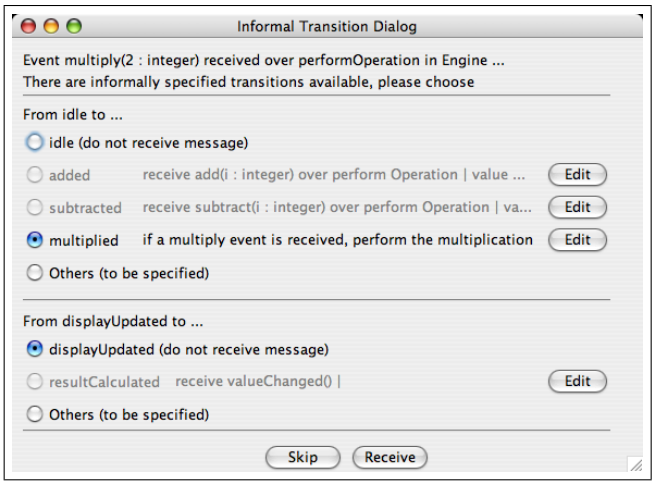

Figure 5. Event assigned to an informal transition.

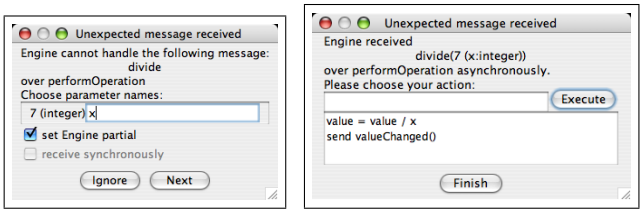

Figure 6. Reception and reaction dialog on an unhandled event.

ify several actions to be performed on this event, i.e. attribute manipulations or sending further events. In our example, the user enters the required behavior for a division. The simulation then continues as usual until another unhandled event is detected or the simulation is finished.

One might argue that this kind of simulation technique requires too much user interactivity. However, a validation anyway is interactive for the input of stimuli, alternative decisions and the observation of outputs. The additionally required interactivity just depends on the degree of completeness in the model.

The model remains unchanged after such an interactive simulation run concerning unhandled messages. The newly recorded behavior can only be found in the recorded sequence chart (Fig. 7). There, it is not apparent whether the behavior was simulated or interactively played. We will see later that we have to store some meta-information to be able to run useful regression simulations.

4 Regression simulation

Regression simulation adapts the conceptual ideas from regression testing. Instead of testing implementations with the purpose of verification, requirements models are simulated with the purpose of their revalidation. Instead of test cases, simulation cases manifest previously validated system behavior in a formalized form.

While the idea of regression simulation is rather obvious, it has not been described or used before in the context of continuously validating evolving requirements models. Furthermore, our extension of regression simulation to semi-formal models (see Section 3) is novel and non-trivial.

As we are using regression simulation to determine whether a requirements model meets the intentions of the stakeholders, regression simulation basically is a *validation* technique. On the other hand, regression simulation *verifies* that the functionality and behavior of a requirements model has not been changed unintentionally during a model evolution step.

4.1 Storing cases under version control

Models are intended to evolve as part of the process; the configuration management system tracks all modifications. The simulation cases are not intended to evolve, but should remain stable, because their purpose is to validate a certain functionality. Only if the systems interface is intentionally

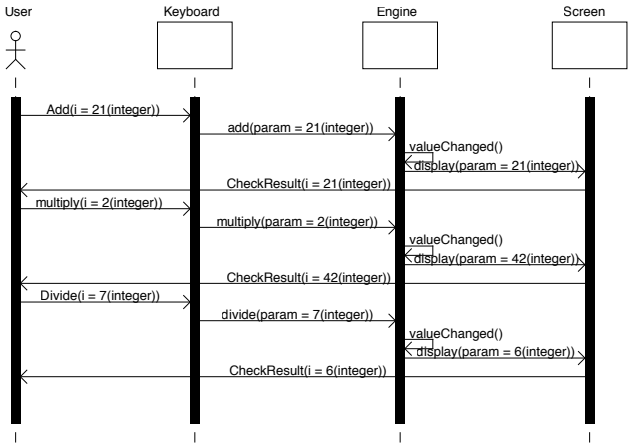

Figure 7. Simulation run for model in Fig. 4.

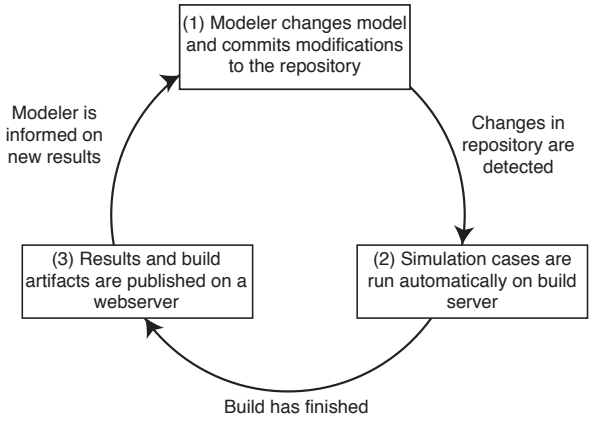

Figure 8. Continuous Integration Cycle.

changed, the existing simulation cases may not run correctly any more and therefore need adaption. That means, each model version belongs to a set of versioned simulation cases that validate that particular version.

In order to simplify the error localization, see Section 5, the model and simulation cases should not be modified at the same time. Either the model is changed, then the simulation cases remain stable. Or the simulation cases are adapted, then the changes of the model are restricted to modifications at the interface.

The re-execution of a simulation case is successful (i.e. does not reveal a problem) if the sequence of messages between the actors and the system components is identical to the strict sequence, whereas the system-internal trace may be different.

4.2 Running the regression

Rerunning a simulation case is performed by replacing the user interface with a replay engine that takes the necessary inputs from the stored simulation case and compares the produced outputs with the stored ones. The current simulation run is stored again as a sequence diagram and compared to its last successful run, the so-called *reference trace*.

To benefit from regression simulation, one cannot rely on manual executions of the simulation cases. This is a time-consuming and error-prone work. The only reliable solution is to establish a framework which continuously reruns the automated simulation cases after each modification.

In the field of coding, continuous integration [6] frameworks are well established, see [1] for example. We adapted this iterative feedback cycle for our purpose to revalidate requirements models. Our process of continuous integration with regression simulation is illustrated in Fig. 8. We adapted the Cruisecontrol framework [1] to visualize the results of our regression runs. An example is given in Fig. 9.

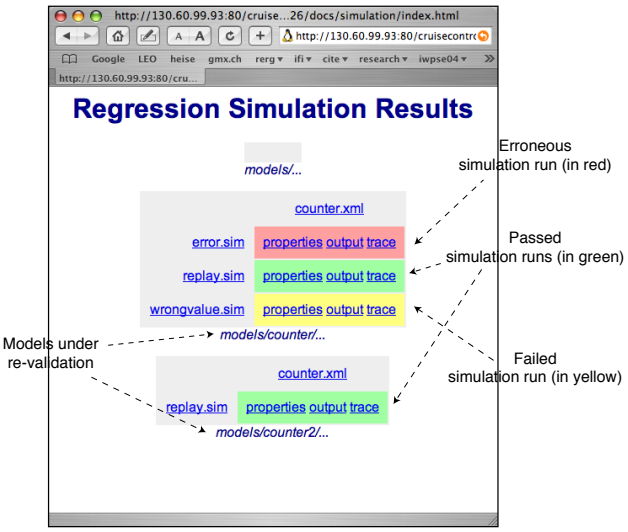

Figure 9. Regression Simulation Results published on a webpage by Cruisecontrol.

4.3 Possible outcomes of a regression run

Table 1 shows the possible results of executed simulation cases. The results are classified into three degrees of severity: Passed, Failure, and Error. The difference between failure and error is that in a failure run, certain controlled properties under proof fail, whereas unexpected incidents result in errors.

Below, we present a detailed explanation of the possible incidents and their causes:

Non-conforming outputs values. In this case, one or more output values received from the system do not match the values that have been recorded. This reveals a problem either in the system, if the behavior was changed accidentally, or in the simulation case, if the behavior of the system

Table 1. Possible incidents in a regression simulation

Incident	Result
No incident executing simulation case	Passed
Non-conforming output values	Failure
Assertions in model violated	Failure
System reaction timeout	Error
Error binding simulation case to model	Error

was intended to change, but the simulation case was not adapted yet.

Assertions in model violated. Assertion may be specified for example as contracts for interfaces to assure their proper use. During a simulation run, these assertions are checked.

Reaction timeout. Executing parallel tasks may lead to deadlocks or the simulation may end in a endless loop. In both cases, no messages are received from the system any more. If no reaction is observed, it must be assumed the simulation got stuck. It will be terminated after some given timeout period.

Error binding simulation case to model. For every simulation run, the recorded simulation case must be bound to the model. This happens by matching the element names occurring in the simulation case to the corresponding model elements (e.g. the name of an actor, message, ...). If no matching element can be found in the model, the binding fails and the simulation run cannot be executed. This problem could be caused by a changed system interface.

4.4 Incomplete Regression Simulation

Next, we investigate how a regression simulation can be established on such interactively recorded simulation cases. On the one hand, we have to avoid interactivity for automatic execution. On the other hand, the interactively recorded behavior is not yet available in the model and therefore, a usual execution (ignoring unhandled messages) would lead to a deviating behavior.

Ideally, we would expect a regression simulation to work on a simulation case including interactively recorded behavior as follows: 1. Running on an unchanged model, the regression simulation has to produce exactly the same results as the interactive simulation did. This can be done by looking up the behavior for unhandled messages in the recorded sequence chart. 2. As soon as the model is extended with the missing behavior or functionality, we want the regression simulation to execute as given by the model *instead of* looking up the behavior in the sequence chart. 3. Functional changes to existing model behavior that had been validated before, lets the regression simulation fail. In this case, we explicitly do not want to use previously recorded behavior from the sequence chart as model faults would remain undetected. 4. When a component that previously was marked as partial becomes complete (i.e. the "is partial" mark is removed), the re-execution of a simulation case must behave as specified in the model. No lookup in the sequence chart is allowed.

These requirements show that we have to distinguish interactively played behavior from general one in the recordings. We do this by storing some meta-information for each event: We must record whether an event was received and the corresponding actions were produced interactively. Furthermore, we must record the effects of an interactively handled event to be able to reproduce them, i.e. the actions that were performed due to that event. Therefore, we store for each action its causing event.

With this information in hand, we are able to build a regression simulation that fulfills the above mentioned expectations. The regression simulation executes as usual until it detects an unhandled event. If the corresponding component is not marked as partial, the message is ignored and the regression simulation continues. Then, the simulation run will fail when this message is relevant for any observable system reaction.

Otherwise, the recorded behavior for the component is taken to search for a suitable reaction. A suitable reaction is one that has been interactively produced upon receiving an equivalent message and has not been used before. Each recording may be used at most once. If there is more than one suitable reaction, the first unused reaction is taken. In contrast to this, the stored message order is not regarded to reach more flexibility in reacting to a changed model.

A regression simulation can produce two additional results:
1. A list of messages that have been recorded interactively, but were not used in the simulation run. This indicates that additional behavior was added to the model so that interactively recorded behavior is not required any more.
2. A list of messages that could not be handled because of the absence of any suitable, interactively recorded message. This indicates that the model was changed in a way that the recordings are not sufficient to handle all messages.

Example: Re-executing the simulation run given in Fig. 7 as regression will lead to a successful result. The statechart of Engine can handle the Add message as usual. The multiply message can be handled as the informal transition was formalized during the interactive simulation run (cp. Fig. 5). For the Divide message that cannot be handled, the reaction can be taken from the sequence chart, as Engine is marked as partial. The reactions are sending valueChanged() to Engine itself and display(6) to Screen. Hence, all messages are exactly handled as they were recorded.

5 Locating model defects

If a regression simulation run fails, the modeler should not only receive a message stating the kind of the fault (as described in Section 4.3 above). S/he should also be provided with information about its *location* in order to help her or him find and remove the cause(s) of the fault. Both the erroneous trace and the configuration management provide information for localizing faults.

There are two facts that help locating the cause for a failure or error of a particular simulation run. On the one hand, the cause lies somewhere in the *trace of the simulation run*. Otherwise, it would not have been noticed. On the other hand, the model evolved since the last successful run of the erroneous simulation case. The evolution manifests in a number of modified elements including deleted and added ones. One or more modifications may have caused the fault.

Both the behavioral trace of the simulation run and the set of modified elements can be used in isolation for the localization of a fault. In order to narrow the defect space, we combine the two. As they are orthogonal to each other, their intersection yields a small set of defect candidates.

5.1 Evolved Model Parts

If a simulation case S fails on a model version B which passed when executing on the previous version A, we assume that the modifications caused the error. From the configuration management system, we can extract the following information:
The two model versions, $Model_A$ and $Model_B$, the reference trace $Trace_{ref,S}$ which has been recorded for a successful run of simulation case S, and the failing trace $Trace_{prod,S,B}$ which has been produced when running the simulation case S on model version B. From this information, we can compute two Diffs: the $ModelDiff_{A,B} = Model_A \cap Model_B$ and the $TraceDiff_{S,B} = Trace_{ref,S} \cap Trace_{prod,S,B}$.

Example: We evolve again our calculator from Fig. 4 by implementing the divide functionality in the statechart (see Fig. 10). Unfortunately, we inserted a copy/paste error that causes our first simulation case from Fig. 1 to fail because of 'Non-conforming outputs values'.

The model difference between failing and working version is $ModelDiff_{A,B}$ = {State:divided, Transition:idle → divided, Transition:divided → idle}

We can see that the transition description idle → divided was copied from the subtraction one (idle → subtracted), however the event name was not changed. This leads to the non-deterministic behavior in Engine when a subtract event is received. It either performs the subtraction or the division operation.

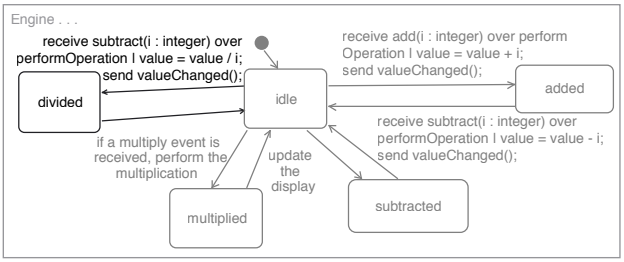

Figure 10. Newly inserted elements in Fig. 4 drawn in black, old ones in gray (clipping of model).

5.2 The Simulation Trace

If the simulation run fails, the noticed failure or error appears at some location in the trace. As we know from testing, the defect normally does not occur at the location where it is caused. The same applies to model simulations.

As a first means for narrowing the defect space, the erroneous $Trace_{prod,S,B}$ can be compared to the recorded reference trace $Trace_{ref,S}$[2]:

$Trace_{ref,S}$ = { {idle → added → idle → subtracted → idle},
{dU → rC → dU → rC → dU},
{User → Add → CR → Subtract → CR} }
$Trace_{prod,S,B}$ = { {idle → added → idle → divided → idle},
{dU → rC → dU → rC → dU},
{User → Add → CR → Subtract → CR }}
dU = displayUpdated, rC = resultCalculated, CR = CheckResult

For better understandability, we visualize the sequence traces within the model. This can be done by connecting the visited model elements according to the Use Case Maps approach [2]. We have done this in Fig. 11 for our two simulation traces. The passed trace is drawn solid, the failed trace is dashed. For each of the three execution threads, we get separate lines beginning at the start state / scenario. For long execution traces, this method does not scale as the model would become overloaded with traces. In the following section, we show how to reduce the load to a reasonable amount of information.

The difference between the two traces formally manifests in the $TraceDiff_{S,B}$ = { {idle → divided / subtracted → idle}, {}, {} }. In the reference trace, state 'subtracted' was visited, in the current execution, it was state 'divided'.

[2]We restrict here to the visited states and scenarios. Usually, we would also consider transitions and parameter values.

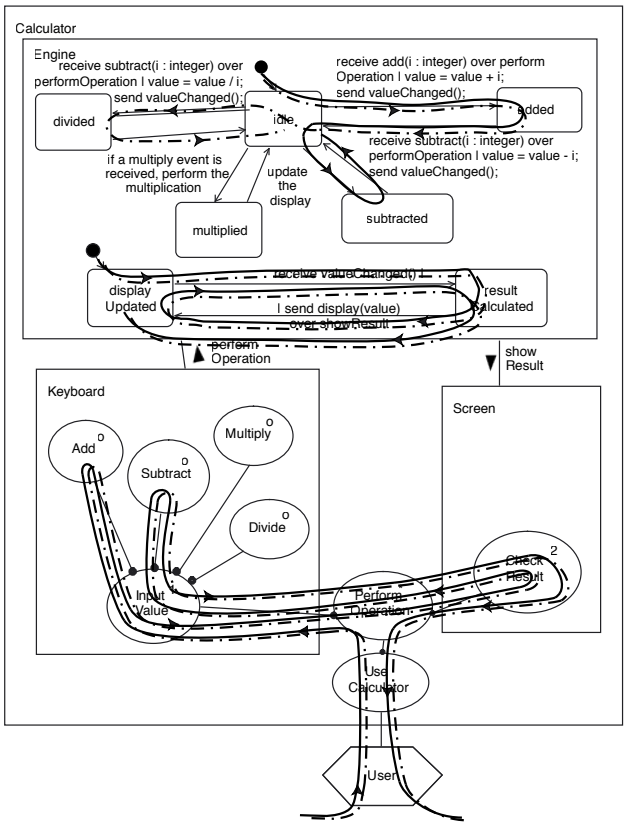

Figure 11. Passed (solid) and erroneous (dashed) traces mapped into the model.

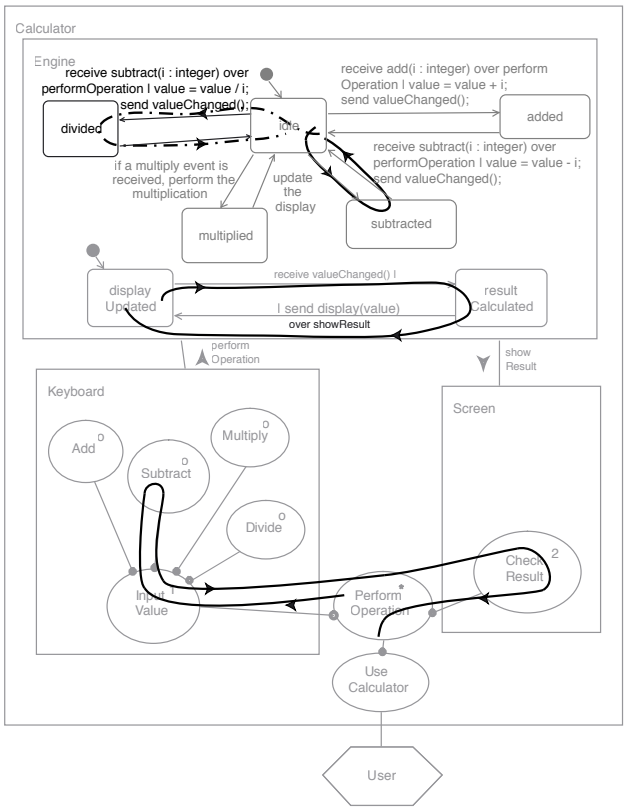

Figure 12. Changed elements and differing traces (dashed vs. solid) visualized in model.

5.3 Combination of Trace and Evolved Model

The combination of the execution trace and the model difference is an ideal means for localizing errors as they consider orthogonal aspects. The intersection of $ModelDiff$ and $TraceDiff$ confirms the newly added state divided is involved. It was changed in the model as well as visited by the failing simulation run. This gives us the strong indication that the recently changed model elements may be responsible for the deviating execution order of the failing trace. Formally:

$TraceDiff \cap ModelDiff = \{\text{idle} \rightarrow \text{divided} \rightarrow \text{idle}\}$

The intersection is displayed graphically in Fig. 12. We can clearly see how the failing execution thread passes through the divided state.

To make our approach scalable, we reduce the long traces in Fig. 11 to the moment where a difference occurs. For the calculation thread this is: idle $\rightarrow$ divided / subtracted $\rightarrow$ idle. The two other non-differing threads could be left out. For better understandability what is going on in the error situation, we leave them in, but reduced to this moment. We can see, while a deviation in the calculation thread was noticed, the user executed the scenarios Subtract and CheckResult and one display update was performed. The diagram now easily reveals the error. As we cut the traces to the first noticeable deviation, this approach works for any length of execution.

Summarizing, the failing trace deviates in one point from the passed trace (got from the $TraceDiff$). The first transition after that point is a newly added transition (got from the $ModelDiff$). With this information in hand, we are able to detect the copy/paste error easily and correct the transition description to 'receive divide() ...'

6 Related Work

In our approach, we are using simulations as a testing procedure of requirements models especially in an early stage. Well considered inputs are entered into a discrete, state-based, semi-formal model to validate its reactions.

There are a number of simulation approaches available based on formal languages like a process calculus for Labeled Transition Systems (LTS) [12] or formally specified dictionaries and tables in the SCR method [10]. These lan-

guages are as complex as programming languages. Models must be complete and totally formal before the first simulation run can be performed. While these approaches are appropriate for simulating models of high-risk systems when models have been completed, they do not work for continuously validating a specification that evolves in a series of semi-formal steps.

In contrast, stochastic simulations (e.g. [5]) perform calculations on qualitative models with randomly chosen inputs.

With Statemate [8], Harel et al. presented an approach and tool that allows to simulate and generate code from formal statecharts. An incomplete design can be compensated by stub components. The Play-Engine [9] allows to interactively play and test behavior of an incomplete component via a prototypically built user interface. However, informality is not considered in these approaches.

Model checking [4] is another approach to verify models. There, the state space of a model is explored to prove certain properties like safety and liveness or provide counter-examples. However, the state space of typical models is too large to be explored. Remodeling to reduce the state space must be performed manually by experts. It must be repeated after each evolutionary step, thus preventing automatic regression model checking. Again, model checking is based on totally formal languages.

Approaches to formally express incompleteness, e.g. by introducing may and must modalities [11], enable an evolutionary model development. Still, the modeling remains difficult and informal descriptions cannot be handled.

In contrast to model checking approaches, simulations are computationally much less expensive. As simulators can be built flexible enough to deal with evolving, semi-formal models [13], they are a suitable means to validate requirements models and to detect errors early [10].

7 Conclusions

We have demonstrated that regression simulation is a useful means for continuously assuring the quality of a requirements model that is developed in an incremental, evolutionary style.

The contribution of this paper is threefold. Firstly, we exploit the idea of regression simulation (which is a rather obvious one) systematically for the first time.

Secondly, we extend the notion of simulation from formal models to semi-formal ones, concentrating on the situation where models are not yet complete. This is our main contribution: the ability to interactively specify missing or informal behavior or functionality in a simulation case, but nevertheless allowing regression simulations to run automatically makes regression simulation applicable in practice, where models are not completely formal most of the time.

Thirdly, we have demonstrated how execution traces as well as evolution information can be used to visualize failing regression runs and how this information can be used for localizing defects in a model.

Our future research will concentrate on (i) the evaluation of performance and usability for larger models, and (ii) the investigation of simulation and regression simulation for aspect-oriented requirements models.

References

[1] A. Almagro and P. Julius. *Cruisecontrol, Continuous Integration Toolkit*. http://cruisecontrol.sourceforge.net, 2004.

[2] R. J. A. Buhr. Use Case Maps as Architectural Entities for Complex Systems. *IEEE Transactions of Software Engineering*, 24(12):1131–1155, 1998.

[3] P. Cederqvist et al. *Version Management with CVS*. Online Manual, 1.11.19, http://www.cvshome.org/docs/manual/, 2005.

[4] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[5] M. S. Feather and T. Menzies. Converging on the Optimal Attainment of Requirements. In *Proceedings of the 10th IEEE Joint International Conference on Requirements Engineering (RE'02)*, pages 263–272. IEEE Computer Society, 2002.

[6] M. Fowler. *Continuous Integration*. http://www.martinfowler.com/articles/continuousIntegration.html, 2004.

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

[8] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. B. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *Software Engineering*, 16(4):403–414, 1990.

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

[10] C. Heitmeyer, J. Kirby, and B. Labaw. Tools for Formal Specification, Verification, and Validation of Requirements. In *Proceedings of COMPASS '97*, pages 35–47, 1997.

[11] K. G. Larsen. Modal Specifications. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems: Proceedings*, volume 407 of *LNCS*, pages 232–246. Springer, 1989.

[12] J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical Animation of Behavior Models. In *International Conference on Software Engineering*, pages 499–508, 2000.

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