Chapter 13
Verifying Runtime Reconfiguration Requirements on UML Models

Selim Ciraci, Pim van den Broek, and Mehmet Aksit

Abstract Runtime reconfiguration is a method used for changing the structure and the call pattern such that the software can adapt itself to the client's computing environment. The current practice of verifying software models with respect to the reconfiguration requirements is rather subjective: based on the stakeholders' needs, architects define a set of reconfiguration scenarios and manually trace the models. This chapter presents a novel process and a tool for automating the verification of the UML class and sequence diagrams with respect to runtime reconfiguration requirements. In this process, the models are simulated, which generates the execution tree. In the execution tree, each path from root to a leaf node is an execution sequence. The branching in this tree is caused by the reconfiguration of the structure and the call pattern. The runtime reconfiguration requirements are expressed with a visual state-based language which is verified against the execution tree. If the verification fails, feedback about the possible location of the problem is presented to the designers. The process has been tested with case studies and experiments conducted on the UML class and sequence diagrams of a software system from Philips Healthcare MRI.

13.1 Introduction

Reconfiguration allows software systems to be adapted to the clients' environment (Oreizy et al, 1998). Reconfiguration can be achieved at compile time by

Software Engineering Group
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
PO Box 217
7500 AE Enschede
The Netherlands
{s.ciraci, pimvdb, aksit}@ewi.utwente.nl

defining configuration semantics as program pieces to be compiled. For certain applications, however, runtime reconfiguration is preferable as the application may not be stopped for reconfiguration and the development environment may not be available at the client side. Runtime reconfiguration can be achieved through program generation, reflective programming and reconfiguration mechanisms (RMs). RMs are parts of the application whose purpose is to change the structure and the call patterns of the application. The change in the structure and the call patterns can be accomplished by polymorphism, conditional statements and dynamic type loading. This chapter focuses on reconfiguration with RMs as they are commonly used for tailoring embedded systems. Figure 13.1 presents an example software system having 3 components. Here, depending on the existence of the hardware at the client side, the structure and the call pattern is programmed to work with the component machine data reader, as shown in Figure 13.1-(a) or the component file data reader, as shown Figure 13.1-(b).

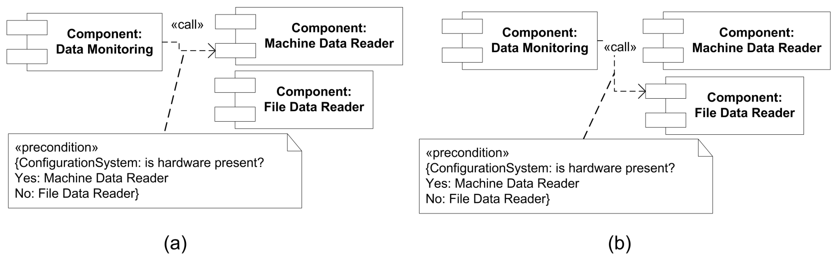
Component: Data Monitoring
«call»
Component: Machine Data Reader
Component: File Data Reader
«precondition»
{ConfigurationSystem: is hardware present?
Yes: Machine Data Reader
No: File Data Reader}
(a)
Component: Data Monitoring
«call»
Component: Machine Data Reader
Component: File Data Reader
«precondition»
{ConfigurationSystem: is hardware present?
Yes: Machine Data Reader
No: File Data Reader}
(b)

Fig. 13.1 A software with 3 modules a) a configuration with module Machine Data Reader b) a configuration with the module File Reader

Since runtime reconfiguration is carried out at the operational phase of the system, it is important that (a) the reconfiguration does not violate the invariants of the system, and (b) the system supports all the desirable reconfigurations. The verification can be realized at the model level or at the code level. Extracting calls/structure from the code can be time consuming and it is generally said that correcting the design errors at the implementation level can be costly. Therefore it is worthwhile to conduct the verification as early as possible such as at the model level.

The Unified Modeling Language (UML) is used to specify, visualize and document the designs of the OO software systems (uml, 2010). From these models, UML class and sequence diagrams are at a sufficiently high level of abstraction to capture the effects of the reconfiguration on the object structure and call patterns. More importantly, they are widely accepted and standardized ways of modeling OO software systems. This makes UML class and sequence diagrams adequate models to evaluate the effects of reconfiguration on the application. However, conducting this evaluation manually is still hard because one has to trace through complex class hierarchies, many conditional blocks and, possibly, many sequence diagrams.

13.1.1 State of the art in reconfiguration verification tools

In the literature, there are a number of approaches using UML models for specifying and verifying the reconfiguration of the software systems (Giese et al, 2004; Becker et al, 2006; Apvrille et al, 2004; Bucchiarone and Galeotti, 2008). For example, in the approach proposed in (Becker et al, 2006), the structure of the software system is modeled using a variant of UML class diagrams and the reconfiguration is simulated with application specific execution semantics. Although these approaches seem to be quite intuitive, we observe the following drawbacks: 1) their modeling languages (or UML profiles) are different from standard UML requiring the software system to be remodeled in the language of the model checker, 2) they require application specific execution semantics, 3) they only consider the structure (e.g. the changes in the connections between components); however, the call pattern leading the structural changes should also be considered.

The approaches for call graph generation (Grove et al, 1997) can be also applied to UML class and sequence diagrams for finding out the changes in the call patterns. However, the changes in call pattern should be supported by the object structure for correct runtime reconfiguration and these approaches do not consider the object structure. Model checking has also been applied in verifying the implementation of the software system (Kastenberg et al, 2006) (Visser et al, 2000). In the OO design phase with UML models, implementation level model checking is not suitable as the semantics provided by these model checkers are specific to a programming language. Whereas in the UML modeling most implementation level details are not known.

13.1.2 Our solution: simulation of UML models

Figure 13.2 presents the process for verifying the reconfiguration requirements of the UML models; in the rest of the chapter we refer to UML sequence and class diagrams as UML models. Here, the designer only inputs the design of the software system in UML; that is without providing additional models. The object structure and the call pattern conforming to a desired reconfiguration or the application invariant is expressed in the order of execution or as an execution sequence. This specification is done using a visual state-based language (VSL). We developed a converter tool that converts the reconfiguration specifications in VSL into formal verification specifications.

The input UML models are converted into a graph-based model. With generic execution and reconfiguration semantics modeled using graph transformation rules, the simulator simulates the input UML models. Two important aspects of these semantics are that they are not specific to an application and the execution semantics implemented by them are similar to actual execution of an OO software system.

The simulation generates a state-space showing all possible execution sequences supported by the input UML models; here, the simulation yields more than one

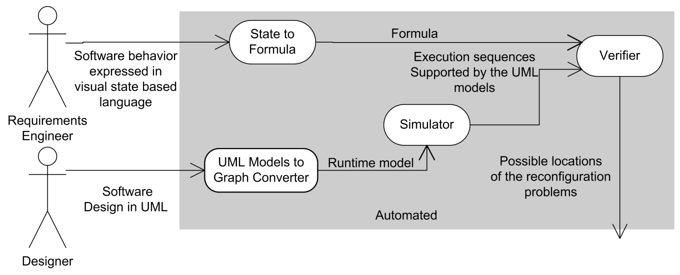
Requirements Engineer
Software behavior expressed in visual state based language
State to Formula
Formula
Verifier
Execution sequences Supported by the UML models
Simulator
Designer
Software Design in UML
UML Models to Graph Converter
Runtime model
Possible locations of the reconfiguration problems
Automated

Fig. 13.2 The process for verifying the reconfigured application behavior on the UML models. The ellipses represent the tools and the arrows are the inputs to the tools.

execution sequence due to reconfiguration. An evaluation algorithm evaluates the formula by searching the generated state-space to find whether the software design models support the specified execution sequence or the execution invariant. If the verification fails, a feedback algorithm outputs guidelines about the possible location of the problem. In this way, the designer is able to verify whether the application behaves in the desired way before/after reconfiguration at the design level without any implementation. In summary, the contributions of our work to the existing literature are:

- Generic execution semantics: the execution semantics can be applied to any complete UML model.
- No additional models are needed: The inputs to the model checker are UML models with RMs. Thus, the designers can use the approach by just adding these tags rather than re-modeling the whole system.
- The verification presents guidelines to the users: When the verification fails, our approach goes one step further and reports possible locations of the problem.

13.1.3 Reconfiguration mechanisms and running example

In the context of UML models, a RM consists of sets of calls with the receiving objects and *configuration parameters*. The reconfiguration is realized by the RM selecting a set of calls depending on the values of the configuration parameters.

Figure 13.3 presents the design of a sorter software system, which supports sorting of integer arrays. The clients use this software system by calling the method *Sorter.sort()*. The implementation of this method consists of a call to the method *sortAlgorithm.sort()* through the attribute *f*. Here, depending on the value of the attribute *f*, one of the subclasses, namely *quickSort* and *insertionSort*, of the interface *sortAlgorithm* receives the call. These subclasses implement different sorting algorithms in the method *sort()*.

The design of the sorter software system supports the reconfiguration of changing the sorting algorithms. The call *f.sort()* is a polymorphic call (i.e. the receiver of

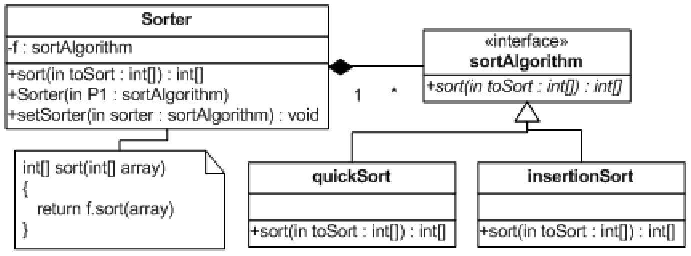
Sorter
-f : sortAlgorithm
+sort(in toSort : int[]) : int[]
+Sorter(in P1 : sortAlgorithm)
+setSorter(in sorter : sortAlgorithm) : void
«interface»
sortAlgorithm
+sort(in toSort : int[]) : int[]
1
*
int[] sort(int[] array)
{
return f.sort(array)
}
quickSort
+sort(in toSort : int[]) : int[]
insertionSort
+sort(in toSort : int[]) : int[]

Fig. 13.3 UML class diagram of the sorting software system.

the call can be changed) and depending on the value of the variable *f*, at runtime, the different subclasses of the interface *sortAlgorithm* might receive the call. Here, the RM polymorphism changes the structure and the call pattern by changing the receiver of the call.

With the sorter software system, we need to ensure that it supports all the desired reconfigurations. For example, a reconfiguration may require the sorter system to run with the merge sort algorithm; however, since this algorithm is not in the design the software system does not support this reconfiguration

13.2 Runtime Reconfiguration Mechanisms

The input to our approach is the UML models with the RMs. We designed extensions to UML for specifying 3 frequently used reconfiguration requirements. These extensions can be added to existing UML models; hence, the remodeling is not required. Below these mechanisms and their specifications in UML are detailed:

Polymorphic Reconfiguration: This RM has one configuration parameter which is the reference variable of the call. Using polymorphism, the receiver of the call changes according to the value of this variable. In a UML sequence diagram, a call action that can be reconfigured using this mechanism should be tagged as *[PolymorphicReconfiguration(reference variable name)]*. With this RM, the designer specifies that a call action can be received by any object-type that is a sub-type of the type of the reference variable. For the sorter system described in the previous section, we want to be able to switch between the different sorting algorithms at runtime. We can achieve this using polymorphic reconfiguration on the call action *f.sort()*, where the variable *f* is the reference variable of the call action. Figure 13.4 shows how this call action is tagged reconfigurable using polymorphism. With this tag, the design shows that the configuration system is able to change the object value the variable *f* is holding at runtime. This change causes the call to the method *sort()* to be received by other type-compatible objects.

Conditional Reconfiguration: This RM uses control-flow statements like *if-else* to select the calls to be executed. The frames labeled *ConditionalReconfiguration*

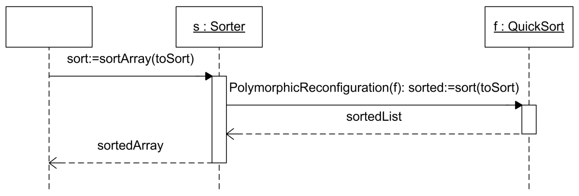

Fig. 13.4 The call f.sort() tagged as reconfigurable using polymorphism.

are used for modeling this RM in sequence diagrams. Similar to alternative frames, the conditional reconfiguration frame contains frame fragments with guards. The configuration parameters are the variables specified in the guards of the fragments.

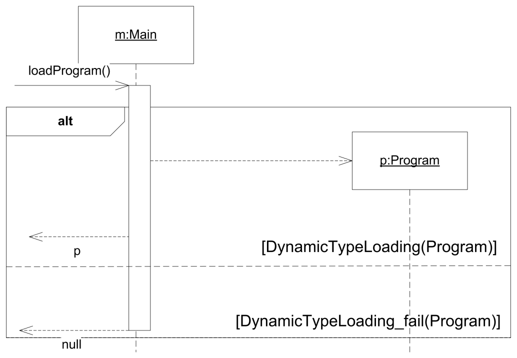
m:Main
loadProgram()
alt
p:Program
p
[DynamicTypeLoading(Program)]
[DynamicTypeLoading_fail(Program)]
null

Fig. 13.5 Dynamic type loading of the class Program.

Dynamic Type Loading: this is the RM where a class is loaded at runtime and it's configuration parameter is the name of the class to be loaded. In UML sequence diagrams, this mechanism can be represented by alternative frames, where one fragment shows the actions taken when the class is successfully loaded and another fragment shows the actions taken when the loading fails. The fragment that shows the successful load of the type has the guard *DynamicTypeLoading(className)* where the parameter *className* is the name of the class to be loaded. The fragment showing the actions taken when the load fails, on the other hand, has the guard *DynamicTypeLoading_fail(className)*. Figure 13.5 depicts a sequence diagram with dynamic type loading. Here, the method *loadProgram()* of the class Main is invoked. This invocation is received by the instance *m* of the same class, which in turn loads and creates an instance of the class Program.

13.3 Graph-based Model Checking of Reconfiguration Requirements

13.3.1 Specialization of graph-based model checking

In graph-based model checking a runtime state of the system is modeled as a graph and the execution semantics are modeled as graph transformation rules. Informally, a graph transformation rule consists of two graphs: a left-hand side graph and a right-hand side graph. A transformation rule is applied to a graph *G*, by recognizing the left-hand side graph in *G* and replacing the recognized left-hand side with the right-hand side graph. The graph production tool automatically applies all the applicable transformation rules to a graph; this generates graphs that represent different runtime states of the system. In this way, the graph production system generates all the states of the system.

In our approach, we specialized graph-based model checking by defining a graph-based modeling language that represents a runtime state of an OO software system at UML's level of abstraction. We call this modeling language Design Configuration Modeling Language (DCML). With DCML, we modeled transformation rules that simulate the execution of call, return and create actions of UML sequence diagrams. An important aspect of these transformation rules is that they achieve a simulation that is close to actual execution of OO software systems. For example, the simulation of a call action involves finding the latest implementation of a method in the inheritance hierarchy.

In addition to these execution semantics, we modeled the semantics of the 3 RMs that are described in Section 13.2. These rules can be applied to a software system when the simulation reaches an action that is tagged reconfigurable in the sequence diagrams.

13.3.2 Simulation in detail

The verification process starts by the designer inputting the UML models. We programmed extensions to the open source UML diagram editor ArgoUML (arg, 2010), where UML models can be imported and converted to DCML models. Once the conversion is completed, the converter tool launches the GROOVE graph production tool with the transformation rules used for simulating UML models. GROOVE is an open source graph production tool that has been used in various research projects (Kastenberg and Rensink, 2006).

DCML models are simulated by GROOVE, which automatically triggers the appropriate graph transformation rules that represent the operational (execution and reconfiguration) semantics of the UML models. This generates a state-space that we call the execution tree of the input UML models. The nodes in this tree are the various runtime states of the software system and the transitions are the applications of the transformation rules. The branching in the tree is caused by the simulation of reconfiguration mechanisms. Each path from the root to a leaf node is an execution sequence supported by the input models.

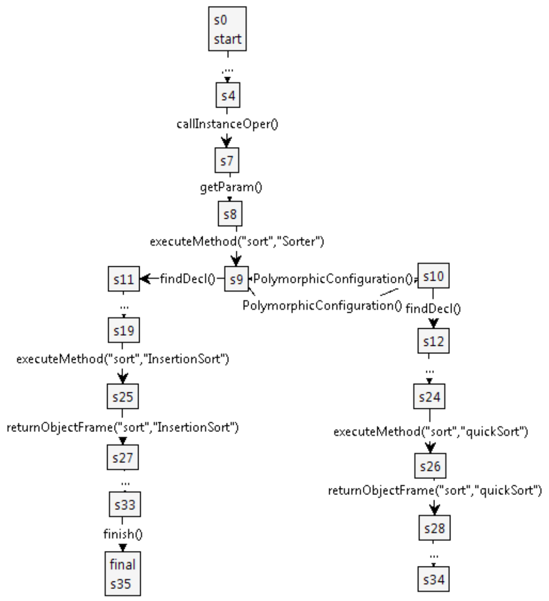
s0 start
s4
callInstanceOper()
s7
getParam()
s8
executeMethod("sort","Sorter")
s11
findDecl()
s9
PolymorphicConfiguration()
s10
PolymorphicConfiguration()
findDecl()
s19
executeMethod("sort","InsertionSort")
s25
returnObjectFrame("sort","InsertionSort")
s27
s33
finish()
final s35
s12
s24
executeMethod("sort","quickSort")
s26
returnObjectFrame("sort","quickSort")
s28
s34

Fig. 13.6 The execution tree of the sorter software system.

As discussed in Section 13.1.3, the sorter software system supports the reconfiguration of changing the sorting algorithm at runtime. The design of this software system, as presented in Figure 13.3, includes two sorting algorithms. The execution tree generated from the simulation is presented in Figure 13.6. Note the branching after the state $s9$: it can be seen with the transition labeled *executeMethod(sort, InsertionSort)* that the insertion sort algorithm has been executed in the left branch. In the right branch, on the other hand, the quick sort algorithm has executed, which can be seen with the transition labeled *executeMethod(sort, quickSort)*. The transition labeled *PolymorphicConfiguration()* between states $s9$ and $s10$ shows that the transformation rule modeling the semantics of the polymorphic reconfiguration is applied. This application, in turn, changed the value the attribute *f* is holding from an instance of the class *InsertionSort* to an instance of the class *quickSort*. Thus, the receiver of the call *f.sort()* is reconfigured

The verification involves finding an execution sequence in the execution tree that conforms an execution sequence of the reconfiguration requirement. The input sequence involves the names of the executed methods and their execution order. Thus, the names of the methods that are executed during simulation should also be included in the execution tree to realize the verification. We have designed parameterized transformation rules that output information about the executed methods and classes for this. A parameterized rule specifies a set of node attributes that should be output instead of the parameters. One of these rules is the rule executeMethod(), which outputs the names of the method and the class that started to execute. Other rules are as follows:

- *executes(O)*: object-type O has received a call.
- returnframe(M, O): method M of object-type O returns.
- *conditionalExecutes(C)*: the simulation executes the operand with guard C.
- *polymorphicReconfiguration(T,O)*: polymorphic reconfiguration has changed the receiver of a call from object-type T to object-type O.
- dynamicTypeLoading(O): dynamic type loading mechanism has loaded the object-type O.

It is important to note here that the user does not have to see the execution tree generated by simulation. The user only needs to enter the execution sequence to be verified as described in the next subsection.

13.3.3 Specification and verification of execution sequences

The execution sequence to be verified is expressed in terms of the executing/returning methods and the applied reconfiguration mechanisms. For this, the names of the parameterized transformation rules are used, where the names of desired methods are placed instead of the parameters. For example, the execution sequence of the quick sort algorithm executing can be expressed as:

After executeMethod(sort, Sorter) eventually (After executeMethod(sort, quickSort eventually returnframe(sort, Sorter)))

Here, the terms after and eventually are used for expressing the execution order of the two methods. Formally, these terms can be expressed using temporal logic; however, this requires some knowledge in formal specifications.

For our approach to be used by designers that are not knowledgeable in formal specifications, we developed a visual state-based language (VSL). VSL is a state machine with accept and reject states. The states are used for specifying actions that need to be observed during simulation, like the execution of a method: *executeMethod*. The transitions, on the other hand, are used for ordering the states. A transition between two states means after eventually. The states can also be labeled as accept or reject to show the property of the state. A sequence of states that ends with an accept state is used for expressing an execution sequence that we are interested in verifying: the simulation should generate this sequence. A sequence that ends with a reject state means an execution sequence that we are not interested in: the simulation can ignore generating this sequence. Figure 13.7 presents the execution sequence of quick sort executing as the sorting algorithm in VSL. Here, the last state is an accept state because it is the last action in a desired execution sequence. That is, we want to observe/verify an execution sequence where first the method Sorter.sort() executes, then, eventually the method *quickSort.Sort()* executes, then, eventually the method *Sorter.sort()* returns.

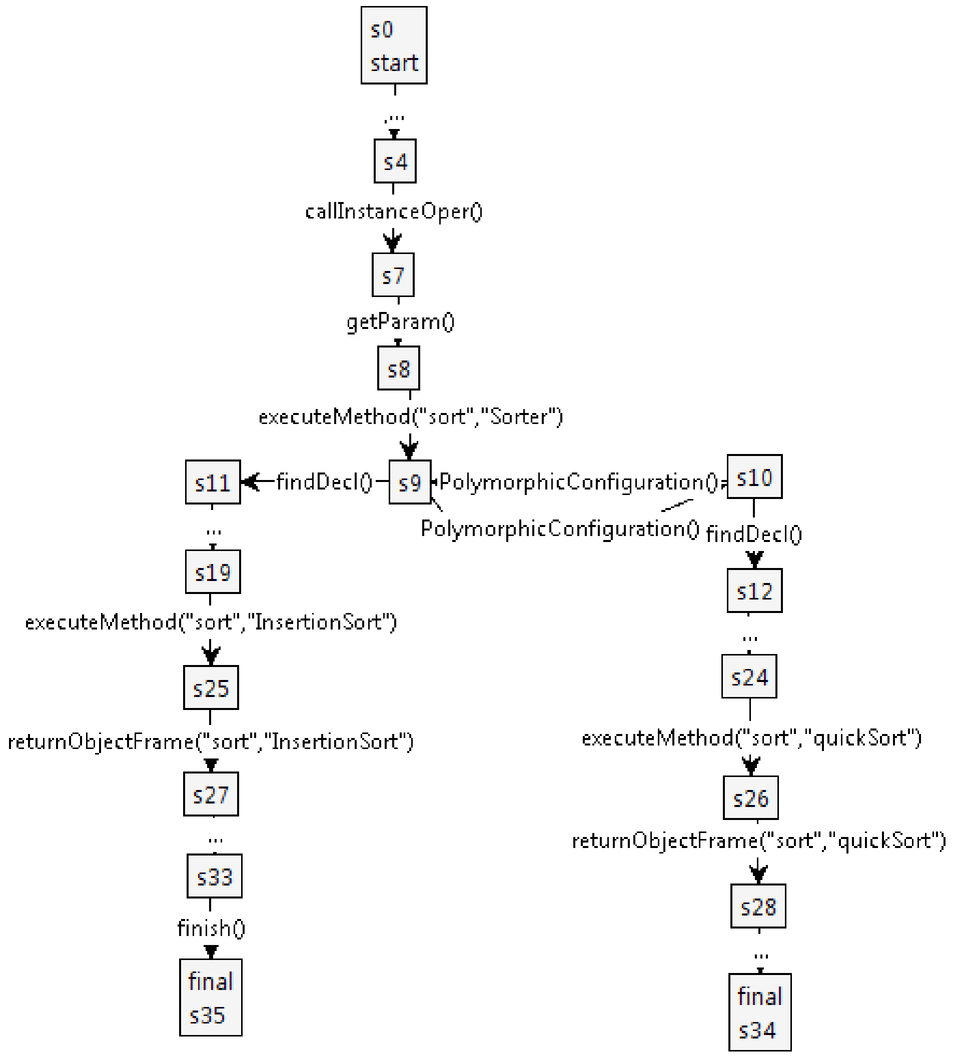
s0
start
....
s4
callInstanceOper()
s7
getParam()
s8
executeMethod("sort","Sorter")
s11
findDecl()
s9
PolymorphicConfiguration()
s10
PolymorphicConfiguration()
findDecl()
...
s19
executeMethod("sort","InsertionSort")
s25
returnObjectFrame("sort","InsertionSort")
s27
...
s33
finish()
final
s35
s12
...
s24
executeMethod("sort","quickSort")
s26
returnObjectFrame("sort","quickSort")
s28
...
final
s34

Fig. 13.7 The reconfiguration requirement of quick sort executing as the sorting algorithm.

The requirement of quick sort executing as a sorting algorithm is a supported reconfiguration requirement. Such requirements specify an execution sequence that should be generated by the simulation. The verification of these requirements is realized by the verification algorithm searching for the execution sequences (in the execution tree) that satisfy the VSL specification. If at least one such execution sequence is located, then the VSL specification is supported by the UML models. If, on the other hand, no such execution sequence can be located, then the verification of the VSL specification fails. For example, if during the simulation of the

UML models of the sorter system, after the execution of method *quickSort.sort()* the method *Sorter.sort()* does not return, then the verification algorithm fails in locating the sequence that satisfies the specification and, thus, the verification fails.

In addition to supported reconfiguration requirements, there are also reconfiguration invariants, which are application constraints that should not be violated by the reconfiguration. These requirements are expressed as execution sequences that violate the invariant. Similar to the verification of supported reconfigurations, the verification of the reconfiguration invariants is realized by searching for an execution sequence that satisfies the VSL specification. Since the VSL specification expresses a violation of the invariant, the verification succeeds if the verification algorithm cannot locate such an execution sequence.

13.3.4 Feedback mechanism

When the verification of a VSL specification fails, a feedback algorithm is executed. The aim of the feedback algorithm is to provide guidelines to the designers in finding the errors in the design. The feedback algorithm for supported reconfiguration requirements simply traces the execution sequences in the execution tree and outputs the execution sequence that supports the longest initial fragment of the specification. For example, if the VSL specification in Figure 13.7 is supported up to the execution of the method *quickSort.sort()*, the feedback algorithm outputs the VSL specification shown in Figure 13.8

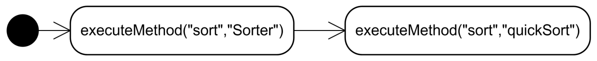
executeMethod("sort","Sorter")
executeMethod("sort","quickSort")

Fig. 13.8 The output of the feedback mechanism showing that the design only supports up to the execution of the method *quickSort.sort()*.

This shows that after the method quickSort.sort() starts executing, there is an error in UML models such that the method Sorter.sort() never returns.

For invariants, the VSL specification expresses an execution sequence that violates the invariant. The invariant is violated when there is at least one execution sequence in the execution tree that satisfies the specification. Once such an execution sequence is located, the feedback algorithm outputs the whole execution trace (from beginning to its end). The designer can, then, trace this execution sequence in the UML models and locate the error causing the violation of the invariant.

13.4 Evaluation of the Approach

We conducted two studies in evaluating our approach: in the first study, we verified the reconfiguration requirements of a software system belonging to the Philips Healthcare MRI software framework and compared the verification with the actual execution. In the second study, we conducted an experiment with computer science master students to test the effectiveness of the feedback mechanisms.

13.4.1 Case study with the designer from industry

We have applied the process for verification of reconfiguration requirements on UML models for verifying the reconfiguration requirements (current and expected in the near future) of a software system called Data Monitoring Viewer (DMV) which belongs to the Philips Healthcare MRI framework (see Chapter 1). This software system is used for displaying the received signals of an MRI system and it allows the user to manipulate certain parameters to display the effects on the signal.

The objective of this case study is to compare the verification results on the UML models with the implemented software. This case study is conducted together with the designer of the DMV. The DMV is written in C# (21K LOC, 139 classes) and is designed to be functionally extendable; it can be extended with signal viewers for specific signals, with user interface elements (e.g. buttons) that add extra functionality (e.g. loading the waveform from a file). Although, the DMV is stand alone software, there are also extensions that integrate the software to the MRI software framework so that it can interact with the MRI system. These extensions are implemented in a class and these classes are compiled as Dynamically Linked Library (DLL) files. These DLL files are placed in a specific directory; when the DMV is launched, before displaying the main window, it scans this directory and loads the classes.

The DMV is designed such that the extensions communicate with the main program through a set of event interfaces (similar to the observer pattern). Upon load of an extension, the extension asks the DMV for a set of events and it registers its handlers to the events it is interested in. Figure 13.9-(a) presents a sequence diagram showing the loading of the extension *Ext1* represented by the class with the same name. The class Main is responsible from the loading of the extensions in the DMV. The loading of an extension starts with the call *Main.createProgram()*. The object p of the class PE is the event interface and the class Ext1 registers its handler for the event *WindowLoad()* by calling the method *PE.registerWindowLoader()*. Here, the event *WindowLoad()* is fired by the DMV when it completes preparing the main window of the application. Handlers of this event can access the window and change it. For example, the extension Ext1 adds a button to the main window. Figure 13.9-(b) presents the sequence showing the notification of the event *WindowLoad()*. Here, the class Extension represents any extension that has registered its handler for this event.

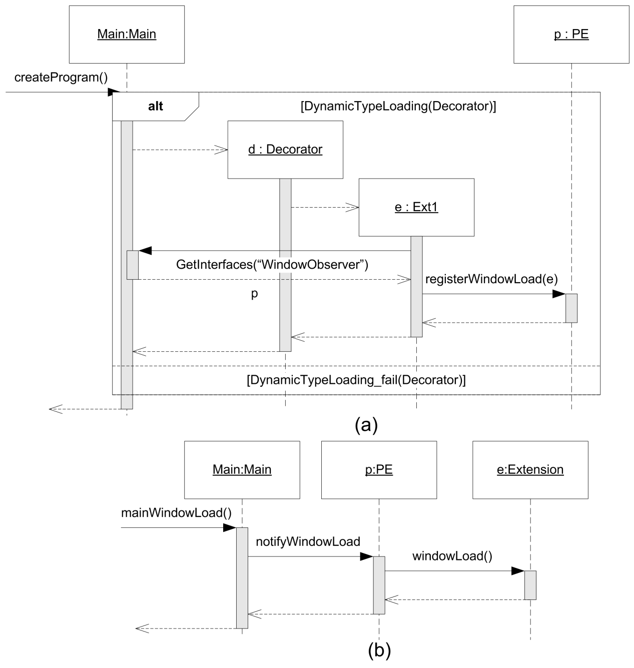

Fig. 13.9 a) Sequence diagram showing the load of the extension *Ext1*. b) Sequence diagram showing the notification of the extensions

The loading of extensions is handled through a dynamic type loading reconfiguration mechanism. In the current version of the DMV, 4 extensions are implemented and, thus, dynamic type loading is executed 4 times. In each load, the DMV has the option of loading one of the 4 extensions or failing. This in total makes 625 (5^4) different execution sequences. In addition to this, the configuration system reconfigures the DMV and is extensions through 5 conditional reconfiguration mechanisms and 1 polymorphic reconfiguration mechanism; these mechanisms also change the interactions in the DMV, causing new execution sequences. Obviously, the DMV supports many execution sequences due to reconfiguration and it is hard for the designer to know each one of these different execution sequences for verifying the reconfiguration requirements.

For the case study, we used the class diagram of the DMV tool and 4 sequence diagrams showing the successful execution of 4 extensions. The sequence diagrams in total contain 66 call/return actions. The execution tree generated after the simulation consists of 785 branches, 22579 states and 22578 transitions. The simulation took 1.6 minutes and 23MB memory with Intel Centrino 1.7GHz processor, 1GB memory and running Windows XP.

We verified 10 reconfiguration requirements of the DMV. In all the requirements the verification and the manual evaluation by executing the DMV agreed. This shows that the simulation can generate the right execution sequences. Below we provide two examples from the verified requirements in order to clarify the verification process:

The DMV should be able to execute when the loading of an extension fails – The DMV is designed to load extensions only when the dynamic type loading succeeds; otherwise, it ignores the loading of the extension as shown in Figure 13.9-(a). If there is an execution sequence where after a failing dynamic type load, none of the events are fired, then, this requirement is violated. This is because the failure in the type loading caused a problem in the DMV and it did not execute as it should. This failure execution sequence is expressed in VSL and put into to the verification algorithm. The verification algorithm did not find any execution sequence in the execution tree that satisfies the VSL specification for this requirement, meaning the requirement is supported by the DMV.

The tool should support extensions that are dependent on each other – In the near future, it is expected that the DMV will be extended with extensions that depend on other extensions. Currently, none of the implemented extensions is dependent on any other. Thus, we modeled two extensions *extV1* and *extV2*, where *extV2* is dependent on *extV1*. The reconfiguration requirement for these extensions is that when both extV1 and extV2 are registered as the handlers of the event *WindowLoad()*, then first *extV1.WindowLoad()* and, later on, *extV2.WindowLoad()* should execute. If there is an execution sequence in the execution tree, where *extV2.WindowLoad()* executes before extV1.WindowLoad() is executed, then the requirement is not supported. We expressed this sequence in VSL. The verification found an execution sequence in the execution tree where *extV2.WindowLoad()* is executed before *extV1.WindowLoad()*. Thus, the requirement is not supported by the DMV.

From the 10 requirements, the verification and manual evaluation showed that 2 are not supported by the DMV tool. These 2 requirements are for features that are planned for future releases of the tool. With our tool, we verified these requirements by just adding a new sequence diagram which is a copy of an existing sequence diagram where a class is renamed. Manual evaluation, on the other hand, required us to implement a new class by copying and adapting an existing class. More importantly, in a corporate environment, conducting such tests with implementation goes through development processes. In addition to this, our simulation has shown the designer that the execution orders of the extensions might change; the designers did not know this and implemented to the DMV without specifying any loading order. Although the design of the DMV contains only 66 actions, the simulation generated many execution sequences because the number of reconfiguration options (e.g. number of possibilities of the dynamic type loading reconfiguration mechanism) in the design is high. It may be argued that designs with a higher number of reconfiguration options, the simulation may require too many resources (or may not complete at all). GROOVE already implements techniques for detecting equal states and merging the branches that yield to the same state. Experiments on GROOVE have also shown that it can easily handle simulations resulting 1.5 million states (Rensink, 2005).

13.4.2 Experiment with students

The motivation of the experiment was to understand how effective the feedback algorithm is in helping the designers to locate the problems. That is, we wanted to know whether using this error diagnosis mechanism helps in correcting reconfiguration errors. The experiment is conducted with 22 master computer science students. Before the experiment the students were given a course on reconfiguration, UML class and sequence diagrams and manually tracing VSL specifications. We used the UML models of the DMV with three extensions; in total the students have received 4 sequence diagrams and 1 class diagram. To follow the non-disclosure agreements, the names of the classes, attributes, and methods have been changed. Because there are two types of requirements, the experiment is divided into two sub-experiments: Experiment 1 (*E1*) tested the effectiveness of the tool with supported reconfiguration requirements and Experiment 2 (*E2*) tested the effectiveness of the tool for reconfiguration invariants. The students were given 4 VSL specifications; two specifications of supported reconfigurations and two specifications of reconfiguration invariants. The students were randomly divided into two groups; the first group did all the specifications about supported reconfigurations manually and the specifications of reconfiguration invariants with tool support (i.e. with the feedback algorithm). The second group evaluated the specifications of supported reconfigurations with tool support and the specifications of reconfiguration invariants manually. For each requirements type, the students have received one easy and one hard specification. We injected errors into the UML sequence diagrams so that all 4 specifications were not supported by the models. The students were asked to evaluate the specifications and to correct only the UML sequence diagrams when the verification of the requirement fails. Here, correction involved removing the call action that is causing the problem. Each specification referred to a distinct set of sequence diagrams.

The data analysis shows that the tool supported group in both groups made 0 errors, while the manual evaluation group on average made 1.25 errors in *E1* and 1.12 errors in *E2*. For both experiments, we conducted formal statistical analysis; interested readers on the analysis are referred to the thesis published about this work (Ciraci, 2009). These analysis yielded a significance value that is lower than the aimed significance value ($p = 0.007 < p_aimed = 0.05$) showing that the experiment results of tool supported group and the manual evaluation group in both experiments is significantly different. With this conclusion, we can state that using the tool indeed affects the outcome of the evaluation and correction process. The statistical power tests for *E1* yielded 0.98 and for *E2* yielded 0.87. These values are greater than the minimum expected power 0.8 showing that the number of subjects are sufficient and the results have strong power for acceptance.

The students were also asked to fill out a survey about the output of the tool. The results of this survey show that 75% of the students found the trace provided by the tool useful; however, 25% said that they did not use the trace. From this 25% of the students the majority said that they wanted a better user interface for the tool.

13.5 Conclusions

In this chapter, we described the process and the supporting tool set for verifying the reconfiguration requirements on UML models. The major aspect of the process is that the model checker and formal processes are hidden from the user. The inputs are UML models and an execution sequence conforming to a reconfiguration requirement. The verification outputs whether the models support the provided the execution sequence. In case, the execution sequence is not support, an execution sequence (with the names of the methods) providing guidelines on the location of the problem is shown to the user.

The verification is realized by specializing the graph-based model checking. The main reason for selecting this model checking method is that UML models can be conveniently represented as graphs. For the specialization, we modeled graph transformation rules that provide execution semantics for UML models that are close to the actual execution of OO software systems. These rules are not application specific, enabling them to be used with any UML models that comply with the UML standards.

We applied our approach to two case studies. In one case study, we verified the reconfiguration requirements of an industrial software system from Philips Healthcare MRI framework (Section 13.4.1) and compared these results with manual execution tests. For all these requirements, the manual execution test and verification yielded the same result. This shows that the verification is able to generate the execution sequences correctly. We also evaluated two reconfiguration features that are planned to be implemented in future releases of the industrial software system. The evaluation using the verification approach required modifications of the UML models and the manual evaluation required changes in the implementation. In a corporate environment, changing the implementation is carefully governed by organizational processes. Conducting such tests on the implementation would also require these processes to be executed; however, such processes may not be required for changing/experimenting on UML models. In the second case study (Section 13.4.2), we conducted two experiments with computer science master students to test whether the guidelines provided by the tool set help in finding the location of the problem. These experiments showed that the tool set was indeed helpful to the students in correcting the errors related to reconfiguration.

References

(2010) Argouml. URL `http://argouml.tigris.org`

(2010) Unified modeling language (uml), version 2.2. URL `http://www.omg.org/technology/documents/formal/uml.htm`

Apvrille L, De Saqui-Sannes P, Sénac P, Lohr C (2004) Verifying service continuity in a dynamic reconfiguration procedure: Application to a satellite system. Automated Software Engg 11(2):167–191, DOI http://dx.doi.org/10.1023/B:AUSE.0000017742.47984.6c

Becker B, Beyer D, Giese H, Klein F, Schilling D (2006) Symbolic invariant verification for systems with dynamic structural adaptation. In: ICSE '06, pp 72–81

Bucchiarone A, Galeotti JP (2008) Dynamic software architectures verification using dynalloy. In: ECEASST '08, vol 10

Ciraci S (2009) Graph based verification of software evolution requirements. PhD thesis, University of Twente

Giese H, Burmester S, Schäfer W, Oberschelp O (2004) Modular design and verification of component-based mechatronic systems with online-reconfiguration. SIGSOFT Softw Eng Notes 29(6):179–188, DOI http://doi.acm.org/10.1145/1041685.1029920

Grove D, DeFouw G, Dean J, Chambers C (1997) Call graph construction in object-oriented languages. SIGPLAN Not 32(10):108–124, DOI http://doi.acm.org/10.1145/263700.264352

Kastenberg H, Rensink A (2006) Model checking dynamic states in groove. In: SPIN'06, Springer-Verlag, Berlin, vol 3925, pp 299–305

Kastenberg H, Kleppe AG, Rensink A (2006) Defining oo execution semantics using graph transformations. In: 8th IFIP, LNCS, vol 4037, pp 186–201

Oreizy P, Medvidovic N, Taylor RN (1998) Architecture-based runtime software evolution. In: ICSE '98: Proceedings of the 20th international conference on Software engineering, IEEE Computer Society, Washington, DC, USA, pp 177–186

Rensink A (2005) Time and space issues in the generation of graph transition systems. In: Mens T, Schürr A, Taentzer G (eds) GraBaTs 2004, Barcelona, Spain, Elsevier, Amsterdam, Electronic Notes in Theoretical Computer Science, vol 127, pp 127–139

Visser W, Havelund K, Brat G, Park S (2000) Model checking programs. In: ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering, IEEE Computer Society, Washington, DC, USA, p 3