Checking the Correspondence Between UML models and Implementation

Selim Ciraci[1], Somayeh Malakuti[1], Shmuel Katz[2], and Mehmet Aksit[1]

[1]Software Engineering Group
University of Twente
Enschede, The Netherlands
{s.ciraci, s.malakuti, m.aksit}@ewi.utwente.nl

[2]Department of Computer Science
The Technion
Haifa, Israel
katz@cs.technion.ac.il

Abstract. UML class and sequence diagrams are used as the basis for runtime profiling along with either offline or online analysis to determine whether the execution conforms to the diagrams. Situations where sequence diagrams are intended to characterize all possible executions are described. The approach generates an execution tree of all possible sequences, using a detailed collection of graph transformations that represent a precise operational semantics for sequence diagrams, including treatment for polymorphism, multiple activations, reference to other diagrams, and the use of frames in sequence diagrams. The sequence diagrams are also used to determine the information that should be gathered about method calls in the system. Aspects that can follow the flow of messages in a distributed system, are generated and the results of execution are recorded. The execution tree is used to automatically check the recorded execution to detect operations that do not correspond to any of the diagrams. These represent either new types of sequence diagrams that should be added to the collection, or implementation errors where the system is not responding as designed. In either case, it is important to identify such situations.

Keywords: runtime verification, UML class and sequence diagrams, execution semantics, graph transformation, aspect-oriented profiling.

1 Introduction

Software models are increasingly used in different phases of the software development life cycle. Here we consider class and sequence diagram models from the UML [1] suite of design models, and use them as the basis for run-time verification and analysis of implemented code. UML is a widely accepted/standardized modeling language for Object-Oriented (OO) systems. Although many diagrams exist, the two most popular are generally acknowledged to be class diagrams to describe the structure and interrelationships among the classes, and sequence diagrams to describe sequences of method calls among objects that together describe important usage scenarios.

Class diagrams list the fields and methods in each class, and show which classes use other classes, which inherit from others, and multiplicities of classes in various relationships with other classes. Sequence diagrams are commonly used for partially

documenting the behavior of the software system as interaction patterns among objects. Due to polymorphism, conditional frames, and multiple activations, these diagrams can represent many possible execution sequences and complex interactions.

Other notations for describing constraints, such as temporal logic, or even state diagrams and the Object Constraint Language (OCL) notations within UML, suffer from low availability: they simply are not widely used in industrial contexts, and are less likely to be updated as the system evolves. For example, UML state diagrams further describe the behavior of a system by defining the internal states and transitions among them for the classes (and objects derived from them). They are sometimes used to describe abstract versions of key protocols. However, it is actually rare to provide a full set of state diagrams, since they are viewed as extraneous and considered to be low-level.

We need to make sure that the implementation is consistent with the diagrams. Since user intervention, reaction, and decision-making are often involved in the use of such reactive systems, runtime verification of conformance is needed, in addition to testing and formal verification of the code. As explained in more detail in the related work section, most other works on using UML to generate monitors concentrate on the class diagrams. Even the few that use sequence diagrams do not treat necessary and common features such as polymorphism, multiple activations, and tracing over multiple sequence diagrams where one diagram refers to another.

Sequence diagrams have traditionally been used as a source for test cases that can be executed on the implemented system to see whether an expected sequence of method calls and responses occurs. This is reasonable if the sequence diagrams are seen simply as describing some sample scenarios of interaction, that in no way prevent other uses and orderings of method calls in the system. Sometimes this is indeed the approach used by software developers to describe expectations from a particular class or method within it: the overall behavior is implicitly understood by describing typical, but by no means exhaustive, cases of its use.

However, there are many software systems with UML designs where the sequence diagrams are more exhaustive: each sequence diagram provides a transaction-like description of scenarios that once begun, should be completed, and the collection of sequence diagrams is intended to show the entire possible use of the system. For a banking system, for example, the provided user actions of initiating a cash withdrawal, a transfer of funds, a deposit, etc, must be followed by an entire sequence of method calls described in one of the sequence diagrams. In the Crisis Mangagement case study presented in the following section, the types of crisis (car accident, fire, etc.) are all assumed to have corresponding sequence diagrams that design the proper response after analyzing the needs. When a situation is encountered that does not correspond to any of the diagrams, either it is a new type of crisis, or the system is not responding as designed. In either case, it is important to identify such situations. Such a use of sequence diagrams is also supported by the extension to live sequence charts [2] where after a pre-sequence occurs, an instance of the main sequence must follow.

When the sequence diagrams are used in this way, they can be used to automatically generate run-time profiling aspects and offline or online analyzers that detect when the system is not being used in conformance with any of the sequence diagrams. As noted above, such detection could mean either that the collection of sequence diagrams is

incomplete, and new scenarios need to be added, or indeed that the implemented system is reacting incorrectly, and some object does not respond as expected to a method call that is part of a scenario.

The approach taken here is to use the sequence diagrams directly to generate a state graph and then an execution tree of all possible abstract states and method calls in the system based on the available knowledge. Since the internal behavior of the methods in the objects is not available, we only check for key visible events, interleaved with internal events and possibly actions from other scenarios. The implemented code and the sequence diagrams are analyzed automatically to determine which information must be collected, and to generate run-time non-invasive aspects that log messages and relevant information, including tracking of the connections between threads in different processors that correspond to a single execution of a sequence diagram. For our implementation, the gathered information is automatically analyzed offline to detect non-conformance with the collection of scenarios, and provide helpful feedback. We also describe how to detect the deviations during execution, simply by sending the gathered information directly to an online version of the analyzer.

This paper introduces an additional automatic traceability and verification step to detect inconsistencies between specified UML sequence diagrams and the actual run-time behavior. One of the key difficulties of run-time verification is knowing what to check, and providing a practical way for users to express that. Since sequence diagrams are already widely used to specify what is possible in a system, their direct use for run-time verification is both natural and useful.

Our approach is fully automated with a set of integrated simulation, run-time verification and feedback generation tools. Furthermore, it can be applied to systems that are composed of a set of distributed sub-systems developed in multiple languages.

The remainder of this paper is organized as follows: In the following section, we present a motivating example on the design of a crisis management system (CMS) to illustrate the problem. Section 3 describes related work in greater detail. In Section 4, we provide an overview of the approach, while Sections 5, 6 and 7 detail the simulation of the sequence diagrams, the extraction of information to generate run-time monitors, and the automatic analysis to detect deviations. Section 8 has a summary of application to the motivating example, while Section 9 concludes the paper.

2 Motivating Example

Helping the victims of a crisis situation requires rigid management and allocation of resources. For example, the allocated resources should be accounted for and free resources should be assigned to a crisis by their location. The Crisis Management System (CMS) case study [3] provides requirements for designing and implementing a software system to manage the resources. In this section, we provide one design alternative to illustrate how our approach can help in finding inconsistencies between UML models and the implementation.

Our design alternative focuses on *coordination* of the crisis resolution process. To facilitate this, we added support for *scenarios* for recurring crises in our design. A scenario prescribes the actions that should be taken in order to resolve the specific

crisis like a car accident. As the users become experienced with recurring crises, they may want to extend existing scenarios and/or add new scenarios to the system. Thus, we modularized the scenarios in our design. The scenarios can be viewed as reactive systems that act upon outside events about the state of the crisis, which are: crisis start, resource allocate, resource dispatch, initial report about the crisis, resolution failed and crisis resolved. Both the scenarios and the system framework that organizes them can be described using class and sequence diagrams.

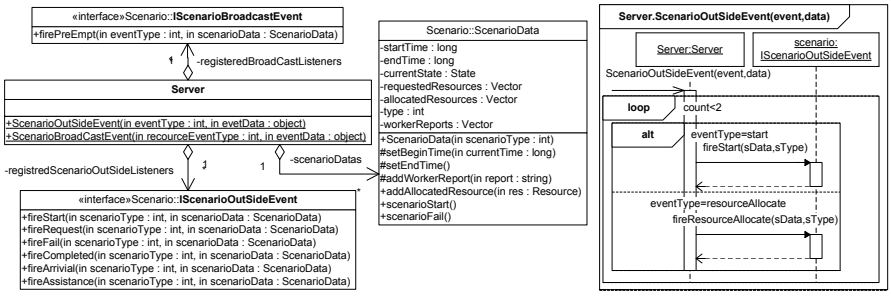

Fig. 1. The class diagram and a sequence diagram of the CMS

The event-based communication and the modularization of the scenarios can be supported by using the observer pattern. Here, the observer provides the interface of events to which the scenarios should react. Figure 1 illustrates the class diagram of the design with these patterns. Here, the class *Server* provides methods where the users (e.g., the user interface) communicate with the scenarios. The interface *IScenarioOutSideEvent* lists the events to which the scenarios should react. A scenario implements this interface and in each method, the actions it should take are specified. The class *ScenarioData* holds the data about the crisis, such as the allocated resources and the time the crisis started. Each crisis has a distinct type designated with the attribute *ScenarioData.type*.

The sequence diagram *Server.ScenarioOutSideEvent* in Figure 1 shows the Server looping through the list of scenarios and letting them know about the start of a crisis. Note that the call to the method *ScenarioOutSideEvent* is asynchronous; the class *Server* executes as a different process so that events arriving at the same time from different sources can be handled. The call to the method *fireStart* is polymorphic and any instance of a scenario can receive this call. Upon receiving an event, a scenario decides if it is interested in the crisis or not with the value of this attribute. This is illustrated with the sequence diagram *CarCrashScenario.fireStart()* in Figure 2.

The CMS has constraints essential for correct operation that emerge from the diagrams. For example, from the class diagrams it is clear that they should handle the list of events. The reactions to the events can be seen in the sequence diagrams of the scenarios. These constraints should be correctly reflected in the implementation, as a violation of a constraint may have catastrophic effects in CMS. In addition, scenarios may react to the same event and interfere with each other, which is considered additional source of complication.

As an example, assume that the CMS is deployed in an environment where only car accidents are managed. In due time, the users realize some accidents need prioritization. One such example is presidential emergencies coordinated by the class *Pre-*

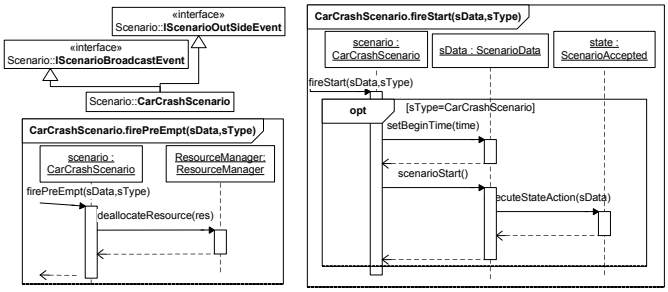

Fig. 2. The class *CarCrachScenario* and the sequence diagram showing the handling of the event *fireStart()*

sidentialEmergency. Prioritization then becomes a requirement, where a high priority scenario may require another scenario to release its resources. The design allows releasing of the resources through *scenario broadcast events*; a high-priority scenario asks the other scenarios to release resources by calling the method *Server.ScenarioOutSideEvent()*, which in turn calls the method *IScenarioOutSideEvent.firePreEmpt()*. The designer models the sequence diagrams showing the object *CarCrashScenario* handling the event *firePreEmpt*. Now assume that the developers do not correctly implement this previously unnecessary sequence diagram, so the resources are in fact not released. As shown in Figure 3, the scenario *PresidentialEmergency* throws the broadcast event for releasing resources when it starts. Because the sequence diagram of *CarCrashScenario* responding to this event is not correctly implemented, it does not release the resources causing the coordination of the presidential emergencies to fail. The error is caused by the code not conforming to the sequence diagrams and, in the remaining sections of this paper, we describe how the approach proposed in this paper can capture such conformance errors.

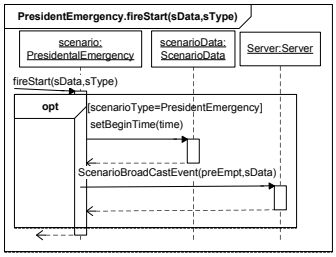

Fig. 3. Sequence diagram showing the scenario presidential emergency handling the event *fireStart*.

3 Related Work

In the literature, the conformance checking of
UML class diagram models with the implementation has been addressed in several ways. Code generation techniques [4] directly generate program skeletons (class declarations and lists of method headers) that can then be expanded to full systems and are guaranteed to satisfy the structural requirements of the class diagram. Mappings can be used to connect formal model elements to UML model elements using predicate logic [5], and runtime state observation can be used to check for consistent use [6]. These techniques only consider structural UML diagrams (class or object diagrams in particular) for conformance matching and do not address the conformance of the behavior/interactions specified using UML sequence diagrams.

A partial solution that does restrict the behavior is provided by adding class invariants or other assertions to the class diagram, using the OCL (Object Constraint Language) notation. Such assertions can then be verified for the implementation, either using static formal methods, or using well-known run-time verification approaches to check assertions about the state of the system. Unfortunately, such invariants are again not always provided in industrial uses of UML, and of course do not treat liveness or required sequences of actions.

Some approaches such as [7, 8] aim at checking the behavioral correctness of software by utilizing state diagrams as their specification language. The problem with state diagram is the level of granularity depends on the employed language/tool for modelling. Thus UML state diagrams can only model the behavior within an object, and not inter-object behavior or message flow. Other languages such as Stateflow [9] focus on the modelling of the abstract behavior of software, without providing constructs to model objects and interactions within and among objects that realize the behavior. Therefore, it may be difficult, if not impossible, to use Stateflow for more fined-grained models of the software.

To overcome the shortcomings of state diagrams, [10–12] make use of UML sequence diagrams as their specification language to verify security policies and web-service interactions, respectively. Although a sequence diagram can include polymorphic calls, calls made by multiple threads of execution, or activate other sequence diagrams, those approaches cannot verify such advanced features of sequence diagrams. For example, in our case study, we would not be able to verify that the execution of *PresidentialEmergencyScenario* in one thread results in the release of resources that are acquired by another thread executing *CarCrashScenario*. The polymorphic nature of *fireStart* in the example is also beyond the capabilities of those tools.

There are several approaches [13–16] that make use of formalisms such as temporal logics and regular expressions for their specification language. One may try to express design conformance criteria as predicates in these formalisms and verify the implementation of software against these predicates. However, there are two difficulties. First, it is likely that most industrial developers are not expert in these formalisms, and would show resistance to using them [17]. Second, instead of reusing design models, separate specifications are used for the verification, which implies that the specifications must also be updated as the software evolves. In the development of complex software with strict time-to-market requirements, it is already a challenge to keep the design documents up-to-date [18], and having more documents (i.e. verification specifications) to update is a time-consuming and error-prone activity.

There are other works such as [19] which extend the message sequence charts of UML as their specification language. However, we believe that such customized versions of UML are not widely used in industrial environments.

4 Requirements and Overview of the Solution

In view of the shortcomings of existing techniques, we identify the following requirements for a conformance checking system based on class and sequence diagrams:
1) The ability to distinguish between polymorphic calls. As Figures 2 and 3 show,

the invocations of *fireStart* and *firePrempt* on *Scenario* objects can be received by different scenarios. Here, according to the object on which the polymorphic methods are invoked, the conformance checking system must be able to match the execution trace with the corresponding sequence diagram.
2) The ability to check the conformance of sequences that span multiple sequence diagrams. For example, a sequence diagram may use a reference frame to include or activate another sequence diagram, as seen in the *PresidentialEmergency* diagram.
3) The ability to support multiple activations. In the implementation of our motivating example, multiple sources (running as different processes and threads) may trigger an event by calling the method *ScenarioOutSideEvent*. Here, the conformance checking system must be able to distinguish between the execution trace of each thread, and must check each trace against the corresponding sequence diagram.

In this paper, we describe an implemented conformance checking system that addresses the requirements above. To utilize sequence diagrams as the specification, we provide a simulator which constructs an execution tree from the sequence diagrams. The execution trees are later on used as specification for the conformance verification. The verification is done in an off-line manner, after the execution of the software terminates, which implies that the execution traces of software must be profiled for the off-line analysis. We make use of aspect-oriented programming to generate profiling aspects (i.e. observers) and insert the aspects into the implemented software code.

Figure 4 shows the three-phase architecture of our approach. In the compilation phase, a developer specifies the sequence and class diagrams, and the XML representation of the diagrams are input to the tools *UML to Graph Convertor* and *Translator*. The tool *UML to Graph Convertor* converts class and sequence diagrams to their equivalent graph representation. The generated graphs are input to the simulator called GROOVE [20], to which we added contains detailed graph transformation rules that closely mimic the actual OO execution of the operations described in the sequence diagrams. These are used to generate an execution tree of all possible executions that conform to the sequence diagrams (that we call a *simulation*).

In the right-hand side, for each specified sequence diagram, the tool *Translator* generates the *Profiling* aspect in the aspect-oriented language Compose* [21]. *Translator* checks the static structure of the software to extract a list of methods defined in the code. In addition, it receives information about the so-called activator method (i.e. the first method that is invoked in a sequence diagram) from each sequence diagram, and generates the *Profiling* aspects. The aspects log information about the methods that are invoked during the execution of an activator method. Since multiple invocations to an activator method may exist, the *Profiling* aspects distinguish among the invocations by associating a unique identifier, called *ActivationID*, to each of them. Consequently, separate log files are generated for each invocation of the activator method.

If during the execution of an activator method, a remote method is invoked in another process, the corresponding *ActivationID* must be passed to the target process and *ActivationID* must be preserved until the call returns to the caller process. This facilitates logging the information about all the local and remote invocations during the execution of an activator method. The *Profiling* aspects are input to the Compose* compiler which generates the executable codes for the aspects and inserts them in the software

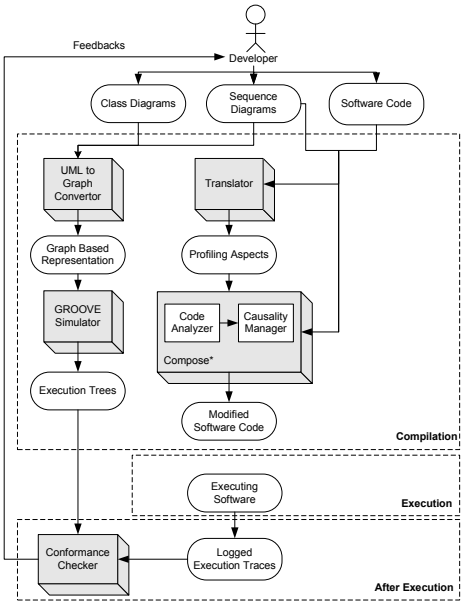

Fig. 4. The overall architecture of our solution

code. The module *Code Analyzer* within the Compose* compiler checks the software code to detect whether there is an inter-process communication (i.e. remote method invocation) within the context of an activator method. The analysis is done based on the available static information about the remote method invocations, for example, sockets or Java-RMI method invocations. If there is such an invocation, the module *Causality Manager* modifies the invocation in both caller and callee sides with one more parameter holding *ActivationID*. Both *Code Analyzer* and *Causality Manager* receive the name of activator methods as input from the sequence diagrams.

After the execution terminates, the tool *Conformance Checker* verifies the log file against the simulation and provides feedback to the developer. The feedback includes both where and in which sequence diagram a deviation is found, and the sequence of calls that led to the deviation including the inconsistent method call or other response.

Although in this work we focus on off-line conformance checking, it is relatively straightforward to accomplish an online checker as well. Here, we need to implement the tool *Conformance Checker* as another aspect which receives the required information from the *Profiling* aspects and checks the observed information against the already-generated execution trees.

As we will explain in the following sections, the sequence diagrams and generated execution models and states are independent from the implementation language of the software. Moreover, the Compose* language is also a language- and platform-independent aspect-oriented language; and the Compose* compiler can compile aspects for various language environments such as Java, .Net and C. This increases the applicability of our approach to software developed in various or even multiple languages.

5 From Class and Sequence Diagrams to Graph Transition System

The simulation of the class and sequence diagrams models are realized with graph-based state-space generation [20] by *i)* Defining a model for representing an OO-like runtime for the UML models with graphs, and *ii)* Modeling generic execution and exception handling semantics with graph-transformation rules over this model. The main reason for adopting graph-based state-space generation is that UML models can be transformed to graph models in a relatively straightforward manner. Also, the user is not requested to provide any other specifications than the UML class and sequence diagrams.

The simulation of the models resembles the execution of an OO software system. For this, we use Design Configuration Models (DCMs), whose meta-model is defined with Design Configuration Modeling Language (DCML), which includes a call stack, operation frames and program counters in addition to the UML elements. These models are represented as graphs since we defined the OO-like execution semantics with graph transformation rules.The DCMs are not defined to be a full semantic representation of OO software. They only include elements that can be modeled with class and sequence diagram models. A DCM is generated from one class diagram and at least one sequence diagram. We programmed a proof-of-concept converter for ArgoUML [22] and the interested readers are referred to [23] for a detailed description of the UML-to-DCML conversion.

The simulation starts from a user specified method we refer to as the *activator method*. It generates a state-space, called the Graph Transition System (GTS), showing all possible execution sequences that can be achieved by the invoke of the activator method. The state-space is a tree where each path from root to a leaf node is an execution sequence. The simulation is realized with a graph-production system, consisting of *57* graph transformation rules that model OO-like execution semantics for UML class and sequence diagrams (these rules can be downloaded from [24]). With these rules the following actions of the sequence diagrams is simulated:

Follow in the activation bars – DCML contains a program counter, which shows the action to be simulated. This program counter is advanced in the activation bar once the action simulation completes.

Call invocation – Allows the simulation of the call instances, self calls, super calls and calls to static methods. For example, the dispatch of a call to an instance method involves finding the receiver object, and then, traversing the inheritance hierarchy to find the latest implementation of the operation. If the object receiving the call implements the called operation, then the inheritance hierarchy is not traversed. If, on the other hand, the object does not implement the operation, the super-type of this object is traversed. After the method implementation is located, an operation frame for the method is created. The program counter of this operation frame points to the first action of the method (i.e. the first action in the activation bar). The newly created frame also contains a pointer (in the form of an edge) to the operation frame from which the call is made; in this way, a call stack is simulated. The semantics of the call to an instance is implemented with 5 graph transformation rules. These rules match when the program counter is a call action node whose receiver is an instance.

Asynchronous call invocation– These are presently allowed only as activator messages

that initiate a sequence diagram. For asynchronous calls that are activator messages, before the call is invoked, a transformation rule increase the attribute *activationCount* of the operation frame. In this way, multiple invocations of the activator method can be distinguished in the state-space.
Parameter passing – is realized by going through the parameters specified in a call action and adding the necessary graph-edges so that the values/object of the parameters are accessible to the dispatched method: for *in* parameters these graph-edges simulate call-by-value and for *out*/*in-out* parameters they simulate call-by-reference.
Create operations – simulation of these creates a new object which represents the classifier receiving the create action in the sequence diagram.
Return from a method – is simulated in two steps when the program counter points to a return action. In the first step, if the returning method has a return value, it is copied to the previous (calling) operation frame. In the second step, a transformation rule "pops" the frame of the returning method from the call stack.
Return value assignments – After the pop of the operation frame, if a return value is copied to the top of the call stack, this return value is assigned to a variable specified in the call action. The assignment is also simulated in two steps: first the type compatibility of the returned value and the assigned variable is checked. Then, if this check succeeds, the variable gets the return value of a method.
Conditional execution – When the program counter is at an alternative frame with N frame fragments, the simulation continues in N branches: in each branch the actions within one of these fragments are simulated. For an optional frame, two branches are created, one activating the operational frame, and one ignoring it.
Loops – The simulation of a loop frame, requires the user to specify the desired number of iterations. Semantics of loops are modeled with 2 transformation rules, which match when the simulation reaches a loop frame node. One of these transformation rules arranges the program counter so that the simulation loops over the actions within the loop fragment. The second transformation rule tests whether the loop is repeated by the user-specified amount and, if so, it terminates the loop.

Figure 5 illustrates an excerpt of the GTS from the simulation of the sequence diagrams shown in Figures 1- 3. The sequence diagram $S1$ shows the operation frame and the program counter at state $S1$: the program counter is at the beginning of the activation bar of the classifier *Server* showing that the call *ScenarioOutSideEvent()* has just been dispatched. From state $S1$ the simulation moves to state $S2$ with the transition *nextcall*, which is the name of the graph transformation rule responsible for incrementing the program counter. Since at $S1$ the call *ScenarioOutSideEvent()* is dispatched, this rule matches and moves the simulation to the beginning of the next action. The sequence diagram $S2$ shows the program counter at state $S2$.

The call *fireStart* is a polymorphic call and can be received by the instance of classes that implement the interface *IScenarioOutSideEvent*. There are two such instances in the input sequence diagrams and, so, there are two outgoing transitions from the state $S2$. When the program counter is at a polymorphic call, the transformation rule *PolymorphicReceiver* has a match for each possible receiver. Each match picks one of the receivers and the application of the match arranges the picked instance as the receiver of the call. This multiple matching causes branching. After the arrangement of the receiver

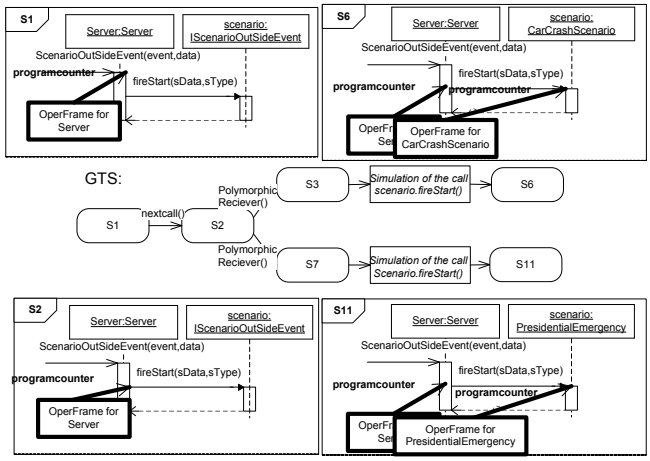

Fig. 5. An excerpt from the GTS of the simulation of the sequence diagrams Server.ScenarioOutSideEvent(), CarCrashScenario.fireStart(), and PresidentialEmergency.fireStart() depicted in Figures 1, 2 and 3.

instance, the call is simulated. The sequence diagrams $S6$ and $S11$ corresponding to the states with the same name, show the operation frame after the call dispatched: at state $S6$ the call *fireStart* is received by the instance of the class *CarCrashScenario* and at state $S11$, it is received by the instance of the class *PresidentialEmergency*. Even though these receivers are in different sequence diagrams, the execution sequences where they respond to a call from the users (i.e. the call *ScenarioOutSideEvent()*) is generated.

The GTS also includes transitions that display which methods begin/end executing. These transitions are added by *parameterized transformation rules*; i.e, a rule specifies a set of node attributes that should be output instead of the parameters. For example, a label *executeMethod(activationCount, ClassifierID, ClassName, MethodName)* is added by the transformation rule *executeMethod* which matches when the program counter is at the beginning of an activation bar.

6 Runtime Observation

During the execution of software, upon the invocation of the activator method, the corresponding profiling aspect is activated, and consequently the runtime transitive effect of the activator method is logged. The output of the logger is the *observed execution sequence*, which is a a state machine where each state has at most one transition. The transitions are of the form $< action >$ *(activationID, ObjectID, ClassName, MethodName)*. Here, the action can be *executeMethod* when the logger observers the start of a method or *returnMethod* when the logger observes the end of a method. To facilitate the conformance checking, the class name, object unique identifier ($ObjectID$), method name and arguments of the methods are logged. As we explained in Section 4, we distinguish between multiple invocations of an activator method by assigning a unique identifier called *activationID* to each invocation, and we log the execution trace initiated from each invocation in a separate logfile. Note that each logfile must eventually finish (be closed) in order to check for deviations. This can be guaranteed even for

executions that do not perform expected events by, for example, building in an "error" operation that closes a log file whenever a time bound has passed with no activity in that log.

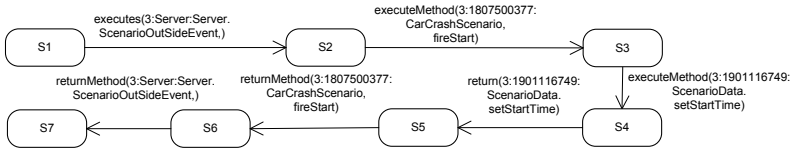

Fig. 6. An example observed execution sequence.

In Figure 6 an example output of the logger is shown. Here, after the activator method *Server.ScenarioOutSideEvent()* starts execution, the car crash scenario executes and returns. The logging stops with the return of the activator method.

7 Verifying Runtime Observation with GTS

The verification is realized by tracing the GTS with the transitions of an observed execution sequence. However, before the tracing starts the GTS is converted to a nondeterministic automata, we refer to as an *abstract execution*. This automata is generated in the following steps: 1) all transitions except the ones added by the informative transformation rules are removed. 2) The states where a different invocation of the activator method occurs (i.e. states with different *activationCount*s) are connected to the start state with λ transitions. 3) a self transition labeled *, a wild card transition, is added to each state except the start state. The semantics of the wildcard transition is specified as follows:
Let ℓ_i be the labels of the all the outgoing transitions from state S_i and let U be the union of all transition labels from an observed execution sequence. The wildcard transition for S_i are all the input whose label belongs to $U - \ell_i$.
The wildcard transition allows us to abstract away from the observed execution sequence to the level of the sequence diagram: as a sequence diagram shows the sequence of important calls, the observed execution may contain calls that are not modeled in it. During the verification the wildcard transitions allows us to map these calls that are not in the sequence diagrams to *don't cares*. Figure 7 illustrates the abstract execution for the GTS presented in Figure 5. Here, there is only one λ transition because there is one invocation of the activator method *Server.ScenarioOutSideEvent()*.

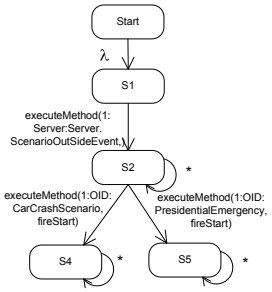

Fig. 7. The abstract execution automaton for the example GTS in Figure 5.

The verification algorithm applies the transitions in the order they are seen from the observed execution sequence to the abstract execution, taking wildcard (irrelevant) operations into account. After applying these transitions, it checks if a final state or

a state with a different *activationCount* in the abstract execution is reached. If such a state is not reached, then there are calls missing in the observed execution sequence which are in the sequence diagram. An important part of the verification is the binding of the identifiers. The classifier and the activation identifiers from the abstract execution are treated as variables, which are bound to actual values from the observed execution sequence. At a transition T_a of the abstract execution, if the method and the class names match to the next transition T_o from observed execution but the activation/classifier identifiers (*activationCount*, *classifierID*) are not bound, then the activation/classifier identifiers of T_a are set to the values of these identifiers at T_o.

We programmed an extension to GROOVE, that uses the output of the runtime observer and verifies it against the GTS generated from the simulation of UML models. Here, the verification step is repeated for each log file.

8 Case Study: Crisis Management System

In Section 2, we described an example inconsistency between the sequence diagrams of the CMS and an implementation, where the scenario car crash does not handle the request to preempt its resources correctly. Here, we show how our approach can detect this inconsistency. For simulation, we used the sequence diagrams showing the handling of the event *fireAllocate()* for the classes *CarCrashScenario* and *PresidentialEmergency* scenario, in addition to the class and sequence diagrams presented in Figures 1, 2 and 3. Each of these two additional sequence diagrams have 3 call actions (and 3 return actions); the DCM generated from these diagrams contains 36 actions. With these diagrams we simulated following scenario: the user interface making 4 outside events; these events are all different invocations of the activator method, and so, they are shown as asynchronous calls. The simulation of these diagrams generated 20075 states and 21007 transitions and completed in 2 minutes using 37Mb memory (with 2.2GHz Core Duo2 laptop running JRE 1.6_11). The simulation generated this many states because the GTS contains every possible execution of the sequence diagrams. For example, an invocation of the method *ScenarioOutSideEvent()* generates 16 branches: 2 branches for the frame fragments of the alternative frame, 2 more branches due to the polymorphic call in each frame fragment adding 4 branches. In each of these 4 branches, another 4 branches is added due the loop frame. The number of transitions are higher then the number of states because of the build-in isomorphism detection mechanism of GROOVE. During simulation, GROOVE detects the isomorphic states in different branches of the GTS and merges them reducing the size of the GTS.

We implemented a prototype CMS in Java using the models presented in Section 2. We also added a sample user interface where the outside events can be sent to the server: the user interface and the class *Server* run in different threads. The implementation of the scenarios and server only consists of the calls presented in the models of Section 2 with the following exceptions: **i)** Upon receiving an event the scenarios call the method *ScenarioStatistics.addStatistic()*; this call is added to test the wildcard transitions. **ii)** To conform with the motivating example the method *ResourceManager.requestDeallocate()* used for releasing resources is not called by the car crash scenario upon receiving the preemption event. We ran this prototype with 2 user inter-

face threads, where one user interface sends two start events and the other sends two allocate events. This run output 4 observed execution sequences, one for each invocation of the method *ScenarioOutSideEvent()*.

The 4 log files are then transfered into the GROOVE to verify the execution sequences. The verification of these state machines took 25 seconds, which includes the time for abstraction execution generation. For the resource allocation request sent by the second user interface thread to the presidential emergency scenario, the verification displayed the mismatch *executeMethod(2, PID, ResourceManager, requestDeallocate)*. This states that the car crash scenario did not in practice call the method to deallocate the resources; however, in the GTS from the sequence diagram this method is called.

This case study shows that it is possible check the UML-to-code conformance for sequence diagrams with polymorphic calls and for execution sequences spanning multiple sequence diagrams. Moreover, the runtime observer is able to trace sequences that originated from different sources and overlap.

9 Conclusion

Sequence and class diagrams can provide constraints that go beyond the assert statements commonly used to provide input for runtime monitoring and verification. This is especially true when the diagrams are intended to describe all possible types of usage for the system. Moreover, these diagrams are often readily available, when used as part of the design process of systems. No new notation has to be mastered, or kept updated as the system evolves, since most development processes anyway require updating the design for purposes of documentation and maintenance.

It may be argued that the conformance checking can be realized without generating all possible sequences (as is done here), where the conformance checker traces the sequence diagrams directly according to the logged execution. For situations where the number of logs to be checked is small, this is less costly than generating all possible executions in advance. However, direct tracing may become too expensive when the number of logs to be checked is large. Software systems, like CMS, are usually deployed at multiple sites and at each site many logs are sampled for consistency checking. Our approach is designed for such situations.

As future work, we are going to apply the approach to industrial software systems. In our previous studies, we have applied a similar simulation to the UML models of an industrial software system from the health care domain [23], where all possible uses of the software system are specified with sequence diagrams. We observed that the simulation generated execution sequences that are not explicitly modeled but are possible due to polymorphism. These execution sequences contained errors, showing the importance of considering the polymorphism in simulation. The application also showed that GROOVE and the simulation can scale to the industrial context.

Acknowledgements

This paper was written when Shmuel Katz visited the University of Twente for 6 months. This visit was partially supported by the NWO Grant no: 040.11.140.

References

1. UML: Unified modeling language. (http://www.uml.org)
2. Damm, W., Harel, D.: Lscs: Breathing life into message sequence charts. Formal Methods in System Design **19** (2001) 45 – 80
3. Kienzle, J., Guelfi, N., Mustafiz, S.: Crisis Management Systems: A Case Study for Aspect-Oriented Modeling. Trans. on Aspect-Oriented Software Development **7** (2010) 1 – 22
4. Harrison, W., Barton, C., Raghavachari, M.: Mapping uml designs to java. In: OOPSLA '00, ACM (2000) 178–187
5. Massoni, T., Gheyi, R., Borba, P.: A framework for establishing formal conformance between object models and object-oriented programs. Elec. Notes Theor. Comp. Sci. **195** (2008) 189–209
6. Crane, M.L., Dingel, J.: Runtime conformance checking of objects using alloy. In: RV'03, Springer-Verlag (2003) 2–21
7. Shing, M.T., Drusinsky, D.: Architectural design, behavior modeling and run-time verification of network embedded systems. In: Reliable Systems on Unreliable Networked Platforms, Springer Berlin / Heidelberg (2007) 281–303
8. Drusinsky, D.: Semantics and runtime monitoring of tlcharts: Statechart automata with temporal logic conditioned transitions. In: RV '04, Springer-Verlag (2005) 3–21
9. Stateflow. (http://www.mathworks.com/products/stateflow/)
10. Seehusen, F., Stolen, K.: A transformational approach to facilitate monitoring of high-level policies. In: Workshop on Policies for Distributed Systems and Networks. (2008) 70–73
11. Simmonds, J., Chechik, M., Nejati, S., Litani, E., O'Farrell, B.: Property patterns for runtime monitoring of web service conversations. In: RV '08, Springer-Verlag (2008) 137–157
12. Kiviluoma, K., Koskinen, J., Mikkonen, T.: Run-time monitoring of architecturally significant behaviors using behavioral profiles and aspects. In: ISSTA '06, ACM (2006) 181–190
13. Malakuti, S., Bockisch, C., Aksit, M.: Applying the composition filter model for runtime verification of multiple-language software. In: ISSRE '09, IEEE (2009) 31–40
14. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: a run-time assurance approach for java programs. Formal methods in system design **24** (2004)
15. Chen, F., Rosu, G.: Mop: an efficient and generic runtime verification framework. In: OOPSLA '07, ACM (2007)
16. Havelund, K., Rosu, G.: An overview of the runtime verification tool java pathexplorer. Formal methods in system design **24** (2004) 189–215
17. Hatcliff, J., Dwyer, M.B.: Using the bandera tool set to model-check properties of concurrent java software. In: CONCUR '01. (2001) 39–58
18. Gulesir, G.: Evolvable Behavior Specifications Using Context-Sensitive Wildcards. PhD thesis, University of Twente (2008)
19. Cook, T., Drusinsky, D., Shing, M.: Specification, validation and run-time monitoring of soa based system-of-systems temporal behaviors. In: ICSSE '07. (2007) 16–18
20. Kastenberg, H., Rensink, A.: Model checking dynamic states in groove. In: SPIN'06. Volume 3925., Berlin, Springer-Verlag (2006) 299–305
21. de Roo, A., Hendriks, M., Havinga, W., Durr, P., Bergmans, L.: Compose*: a language- and platform-independent aspect compiler for composition filters. In: WASDeTT '08. (2008)
22. ArgoUML. (http://argouml.tigris.org)
23. Ciraci, S.: Graph Based Verification of Software Evolution Requirements. PhD thesis, Univ. of Twente, Enschede (2009) CTIT Ph.D. thesis series no. 09-162.
24. GrACE: Graph-based adaptation, configuration and evolution modeling. (http://trese.cs.utwente.nl/willevolve/)