A graph-based aspect interference detection approach for UML-based aspect-oriented models

Selim Ciraci, Wilke Havinga, Mehmet Aksit, Christoph Bockisch and Pim van den Broek

University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
{s.ciraci, w.havinga, m.aksit, c.m.bockisch, p.m.vandenbroek}@ewi.utwente.nl

Abstract. Aspect Oriented Modeling (AOM) techniques facilitate separate modeling of concerns and allow for a more flexible composition of these than traditional modeling technique. While this improves the understandability of each submodel, in order to reason about the behavior of the composed system and to detect conflicts among submodels, automated tool support is required.
Current techniques for conflict detection among aspects generally have at least one of the following weaknesses. They require to manually model the abstract semantics for each system; or they derive the system semantics from code assuming one specific aspect-oriented language. Defining an extra semantics model for verification bears the risk of inconsistencies between the actual and the verified design; verifying only at implementation level hinders fixing errors in earlier phases.
We propose a technique for fully automatic detection of conflicts between aspects at the model level; more specifically, our approach works on UML models with an extension for modeling pointcuts and advice. As back-end we use a graph-based model checker, for which we have defined an operational semantics of UML diagrams, pointcuts and advice. In order to simulate the system, we automatically derive a graph model from the diagrams. The result is another graph, which represents all possible program executions, and which can be verified against a declarative specification of invariants.
To demonstrate our approach, we discuss a UML-based AOM model of the "Crisis Management System" and a possible design and evolution scenario. The complexity of the system makes conflicts among composed aspects hard to detect: already in the case of two simulated aspects, the state space contains 623 different states and 9 different execution paths. Nevertheless, in case the right pruning methods are used, the state-space only grows linearly with the number of aspects; therefore, the automatic analysis scales.

1 Introduction

Aspect Oriented Modeling (AOM) techniques [37, 9] facilitate separate modeling of a project's concerns and allow flexible composition of these without being restricted by hierarchical composition schemes. If used properly, AOM may reduce complexity, and increase flexibility and reuse of models. Most AOM approaches extend a modelling language established for the proramming paradigm on which the targeted aspect-oriented systems are based. Thereby, the AOM approaches differ in the way how the base modelling language is extended and in the crosscutting concerns they can successfully modularize. Nevertheless, they share the goal to provide different partial views on the same system, which may overlap after composition. This improves the global understandability of a model, as smaller submodels – the partial views – can be inspected separately; furthermore, separate submodels can evolve in isolation.

One downside of the flexible composition mechanisms in AOM is that the local understandability is reduced [13]. Detailed understanding of single facets of the behavior in the composed system is difficult because the details of the composition are, on purpose, hidden from the beholder. Therefore, it is difficult to ensure the absence of conflicts in the composed system by manual inspection. Conflicts in the *behavior* when crosscutting parts of the system interact are especially hard to detect, such conflicts are called *semantic interference among aspects*. Conflict detection is mission-critical in large software systems like a "Crisis Management System" (CMS) which cannot afford to expose unanticipated behavior.

Several techniques exist for automatically detecting conflicts between aspects. Most of them either require an abstract semantic model of the overall system that has to be defined specifically for each new system [33, 32]; or they work at the implementation level and assume a particular aspect-oriented programming language [4, 24]. Only a few aspect-interference detection approaches have been proposed that work at the modeling level [33, 32]. But these only work on a specific limited AOM approach and require a manual definition of the semantics of the models.

Performing aspect-interference at the model level has the benefit that models are independent from the programming language used later to implement the model. In contrast, programming-language-based interference detection generally requires reimplementation of the detection technique for each language. Furthermore, early detection of conflicts at the model level has a number of advantages:

- Models are more abstract then code. Therefore, fixing errors in the design prospectively is cheaper then fixing errors in the code.
- When errors are recognized and fixed at the model level, one source of the code's deviation from the model is eliminated. Thus, model versioning and consistency enforcement activities can be avoided.
- In the case of model-based code generation, aspect-interference detection at the code level may even become unnecessary.
- As the AOM model is independent of the technique and language later used to implement the system, model-based conflict detection is also independent of these concerns.

For these reasons, we propose a technique for automatic detection of semantic interference among aspects at the model level; more specifically, our approach works on UML models with an extension for modeling pointcuts and advice. Our aspect-oriented extension requires that pointcuts, advice and their bindings are already resolved; this opens the opportunity to map different modeling styles to our generic model. We discuss such a mapping for the Theme/UML [12] AOM approach.

As our back-end is a graph-based model checker (namely GROOVE [23]), first the UML-based AOM model is transformed to a graph-based representation; already this step is automated. Second, the runtime behavior of the model is simulated at graph level; an operational semantics of the UML and our aspect-oriented extension is modeled as graph transformations, and triggering the relevant operations allows us to simulate the runtime behavior of the model. Finally, the simulated model is verified against the invariants of the system and each composed scenario, which are defined declaratively as computation tree logic (CTL) expressions. Our model checker allows us not only to detect that conflicts are possible, but it also shows under which conditions the conflicts take effect.

To demonstrate our approach, we first discuss how UML-based AOM techniques can be applied in modeling the "Crisis Management System" (CMS). As an example, the concerns that relate to the crisis-managing scenarios are proposed to be modeled as aspects; this design exposes a reduced complexity and eases the evolution of scenarios. Since defining and implementing all scenarios at the same time is too rigid and not realistic, this design has a further benefit: it allows incremental modeling and introduction of crisis-management strategies. Whenever necessary, new scenarios should be defined by specialized experts and be introduced into the system incrementally. Since detection of inconsistencies requires joint analysis of the composed scenarios which may be developed by independent experts at different times, manual detection is extremely difficult if not impossible.

With our graph-based model checker, we have been able to detect semantic interference among two independently developed scenarios. The size of the simulated state space, namely 623 states, 652 transitions and 9 execution paths, shows that tool support is crucial to verify the behavior of a system like the CMS. The size of the state space of our simulation grows linearly with the number of aspects. Therefore, the simulation also scales to reasonably large systems.

The contributions of this paper are threefold:

1. For a graph-based model checker, we have defined graph-transformation rules that constitute an operational semantics for aspect-oriented models. The semantics is an accurate representation of the runtime behavior of the models.
2. Applying this operational semantics with a model checker, we can automatically detect semantic interferences among aspects at the modeling level. As input, our tool only requires a declarative description of invariants and a graph-based representation of the AOM model, which can be derived in an automated way from UML models; different aspect-oriented extensions to UML can be mapped to this graph-based representaion.
3. We show that our approach is applicable to large-scale software by the example of the "Crisis Management System". Therefore, we discuss a possible

design and evolution scenario of this system where aspects conflict in non-obvious ways. With our approach, it is possible to detect these interferences at the model level.

The structure of this paper is as follows. In Section 2 we discuss an AOM model for the "Crisis Management System", including an overview of AOM in general and Themen/UML in particular, which we use to show AOM models throughout this paper. Our approach is discussed in detail in Section 3; we start by presenting the graph model used in the simulation and its relation to UML diagrams in Section 3.1, in Section 3.2 we present the means by which our model supports aspect-oriented models. In Sections 3.3 and 3.4 we discuss the operational semantcs of our model and the verification of invariants in the simulated state space. We discuss the application of our approach to the CMS case study and present performance figures in Section 4. In Section 5 we present related work before we reflect on our approach and conclude in Section 6.

2 Designing the Crisis Management System using AOM

2.1 A brief overview of AOM approaches

To date, a substantial number of AOM approaches have been proposed [37, 9]. Many of these modeling approaches define UML extensions that support the modular expression of crosscutting elements [17, 12, 6]. Although each modeling approach facilitates the expression of crosscutting behavior in different ways, these approaches share many common characteristics. Typically, a user first identifies crosscutting concerns in the system under design. This can be done manually or supported by tools, such as EA-miner [36]. Then, the system is designed using a mix of regular UML-based models (such as class diagrams, sequence diagrams, etc.) and aspect-specific extensions that model crosscutting (structural or behavioral) elements. In this paper, we focus on the use of such UML-based approaches.

The goal of AOM approaches is to improve the modularity of software designs, and this is commonly supported by allowing the specification of different (partial) views on the same system, which may overlap after composition. This improves the potential for separate views of the system to evolve in isolation, i.e., without affecting multiple models in several places. Unfortunately, the specification of multiple separated views on the system as supported by AOM approaches also has a disadvantage: given that each separate view of the model may evolve in isolation, and each may be maintained by different engineers, it becomes harder to ensure that the composed system works together as intended. If crosscutting parts of the system interact in undesired ways, this is called semantic interference among aspects.

The benefit of an improved modular structure facilitated by AOM becomes less important, if the same approach makes it harder to establish that the composed system will behave as intended, which is a common theme among comments from industry, e.g., in [13]. For this reason, we believe that it is important to support the AOM development process by means of tools that detect (semantic) interference among design artifacts. As UML is used as the basis for many AOM approaches, this paper focuses on the automated, tool-supported detection of semantic interference among aspects at the UML design level. The next subsection briefly introduces Theme/UML, the specific approach that we use in this paper.

2.2 On the use of Theme/UML

For the purpose of designing the Crisis Management System we chose to use Theme/UML [12], a representative member of the UML-based approaches mentioned above. The use of specific AOM approaches is however not the focus of this paper, as the problem of semantic interference among aspects is inherent to all AOM approaches. In section 3 we discuss the requirements under which AOM-specific UML extensions can be mapped to our approach in general, and the mapping for Theme/UML in specific. Since the mapping typically appears to affect only a small part of the modeling approach (e.g. mapping the specific ways in which pointcuts and advice are modeled), we expect this to be possible with reasonable effort.

For a detailed description, please refer to publications about Theme/UML [12, 7, 11]. Here, we only discuss the main principles of using Theme/UML, as needed to follow the discussion in this paper. Seen from a user perspective, Theme/UML adds two important features to UML models: it allows (1) the separation of structural concerns, and (2) the expression of crosscutting behavior.

Structural elements (such as – potentially partial – class definitions) can be separated into "themes", which can then be composed into a coherent system by means of a composition specification. For example, a simple composition specification might simply merge the partial classes (defined in several themes) based on their names. This way, Theme/UML supports the modular expression of crosscutting structure.

Crosscutting behavior can be expressed by allowing the use of "template parameters", such as class parameters or method parameters in various models – most importantly, in sequence diagrams. A "template parameter" may be bound by a composition specification to zero or more actual classes or methods, for example indicating that a particular sequence of events should be initiated whenever one of the bound parameter methods is invoked. In this sense, such composition ("parameter binding") specifications can be considered a "pointcut", whereas a sequence of events that is specified (using a sequence diagram) to follow the invocation of such a bound parameter can be considered an "advice".

2.3 Concern identification

There are many alternative ways to define a modular structure for the Crisis Management System described in the case study. Typically, the choice for particular design alternatives would be driven by evaluating relevant trade-offs against the characteristics that are deemed most important by the various stakeholders. For example, the convenience of a given design may be judged with respect to optimized performance, ease of configuration or use, enforcement of security, etc. In this subsection, we describe one possible design, which will be used as an example throughout this paper. As the design as such is not our main focus, we do not describe all the trade-offs made to reach this design in detail, however. The two concerns that are relevant to the CMS are:

- *Coordination.* To facilitate dealing with crisis situations in an efficient manner, an automated system should actively support the coordination of mission- and resource assignment. By *coordination*, we mean the support for standardized scenarios – which may vary based on the type and severity of a crisis – for requesting resources, defining missions, dealing with reports and requests for assistance from workers, etc.
- *Allocation.* To support the handling of crises with a limited amount of resources, the system should support means to optimally allocate available resources. Since optimal strategies may depend on (e.g. national) policies as well as circumstances (number of concurrent crises, scarcity of certain resources), the system should support multiple allocation strategies, as well as pre-emption of resources in low-resource situations, if appropriate.

In this paper, we focus primarily on the concern of coordination support, as "supporting coordination of crises resolution processes" is a primary requirement for this system (as defined in the case study, section 2). For the CMS to properly facilitate this, it should support standardized *scenarios* that deal with recurring types of crises, such as the Car Crash Scenario described in the case study. This way, a scenario prescribes the actions that should be taken to remedy a specific crisis.

Since the definition of such reusable scenarios is likely to be the most unstable part of the CMS, it makes sense to explicitly modularize scenarios. For example, scenarios will be subject to change based on national policies. In addition, new scenarios may be introduced at a later stage, and scenarios may also be extended in an incremental way to incorporate knowledge acquired in its previous applications.

Within the context of the CMS, a scenario can be seen as a reactive controlling process, because it gathers all kinds of information from its environment and reacts to these "events". As to how such a system should be designed, several publications state that it is best to separate the *coordination of behavior* from the *behavior* itself [20, 3].

In this sense, the coordination of a scenario can be seen as a crosscutting concern: several scenarios may need to react to the same event, while conversely, one scenario may depend on multiple information-gathering (and event-generating) modules. In the context of reactive systems, the notion that coordination of behavior can be seen as a crosscutting concern has been identified before [5]. Since this is the case, the coordination of a scenario can be modeled as an aspect that intercepts events that are of importance to the scenario, and invokes the intended actions prescribed by the scenario. This way, the coordination of behavior is properly separated from the behavior itself, as well as decoupled from the different information-gathering and event-generating modules. In the next subsection, we show how the Car Crash scenario can be modeled following this principle.

2.4 Modeling the Car Crash scenario as an aspect

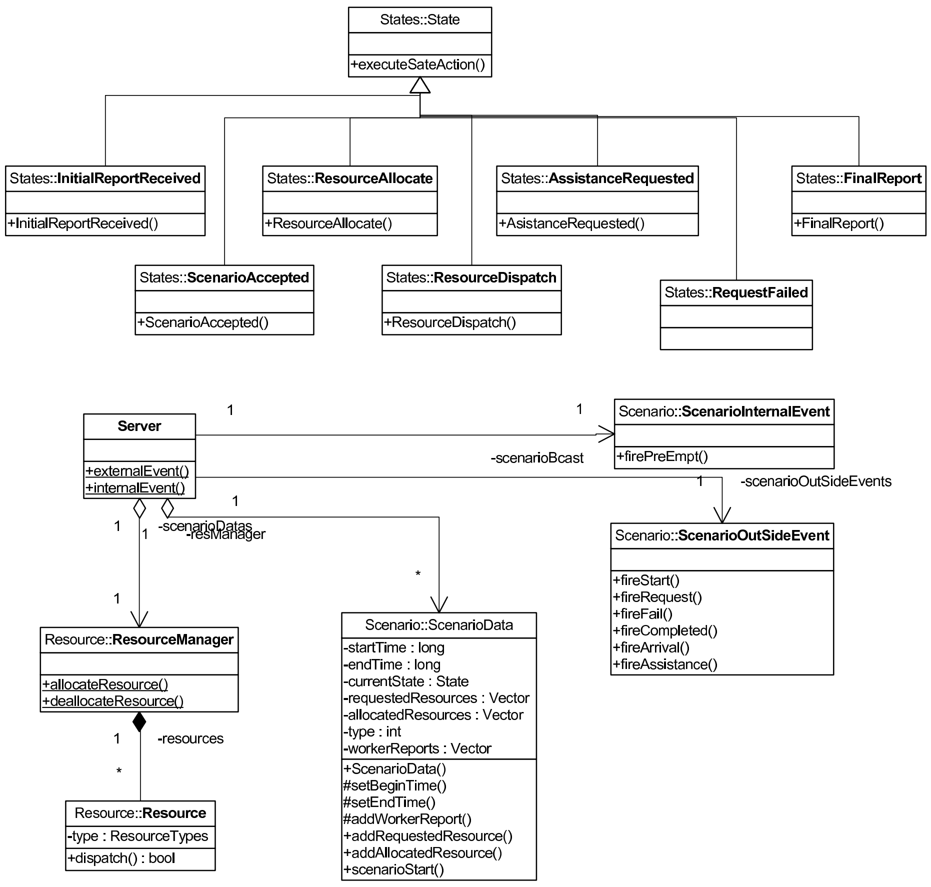

Fig. 1. Class diagram of the Crisis Management System

Figure 1 shows the important structural elements of our design. The structure at the top of the diagram shows a collection of classes modeling a hierarchy of *states*. Potentially reusable actions are modeled as states, each of which implements a specific part of desired system behavior. For example, an instance of class ResourceAllocate defines behavior that checks whether a requested resource is available, and if so, allocates it to a scenario. ResourceDispatch implements instructing a specific resource (i.e., assigning it a mission to carry out). Note that this part of the structure does not implement a complete state machine; the *coordination* of these states, i.e. defining transitions between them as the result of particular events, is implemented by aspects that model specific scenarios.

The bottom half of figure 1 shows the structure of other relevant system components. The class Server has an interface to receive external as well as internal events. External events originate from the environment and are generated by a client (not modeled here) through a user interface. Internal events are signaled by the system itself, for example, if it runs low on resources of a specific kind. Furthermore, the server keeps track of resource allocations through a class ResourceManager, as well as a list of common data for each crisis scenario, as found in the domain model specified in the case study.

The implementation of the Car Crash scenario, as exemplified in the case study, can then be realized as shown in figure 2. This figure, which defines the Car Crash "theme", consists of two parts. The first part (Figure 2-(a)) is the definition of the pattern class CarCrashScenario, which keeps track of the current state the system is in and which may also define behavior that is specific to the low-level implementation of this scenario. The second part is a sequence diagram, that defines how to react to selected events.

Some Theme/UML-specific extensions are visible in this sequence diagram: the sequence diagram is *parameterized*: it refers to a template parameter and the template class CarCrashScenario, both of which should be bound with other "themes" by means of a composition specification. The meaning of the template parameter in the sequence diagram is that the actions specified by the sequence diagram will be executed whenever a method bound to that parameter is invoked. Figure 2-(c) shows the binding specification for the theme Car Crash Scenario, where the pattern class CarCrashScenario is bound to the class *ScenarioOutSideEvent* and the template pattern *_fireStart()* is bound to the actual operation *fireStart()*.

The meaning of the sequence diagram is that it responds (only) to events that indicate a scenario should be started, specified by the invocation of fireStart. The remainder of the sequence diagram forms the advice, which is executed at each fireStart join point (invocation): if the scenario-type specified by the event is indeed a CarCrash scenario (indicated by the guard condition "scenarioType=1", in the diagram), it changes the state of the scenario to "scenario accepted", and invokes the actions defined by that state. Of course, each event has exactly one scenario-type, and in that sense the guard condition based on the scenario-type can be considered to be mutually exclusive with such guard conditions that may be defined by other scenarios.

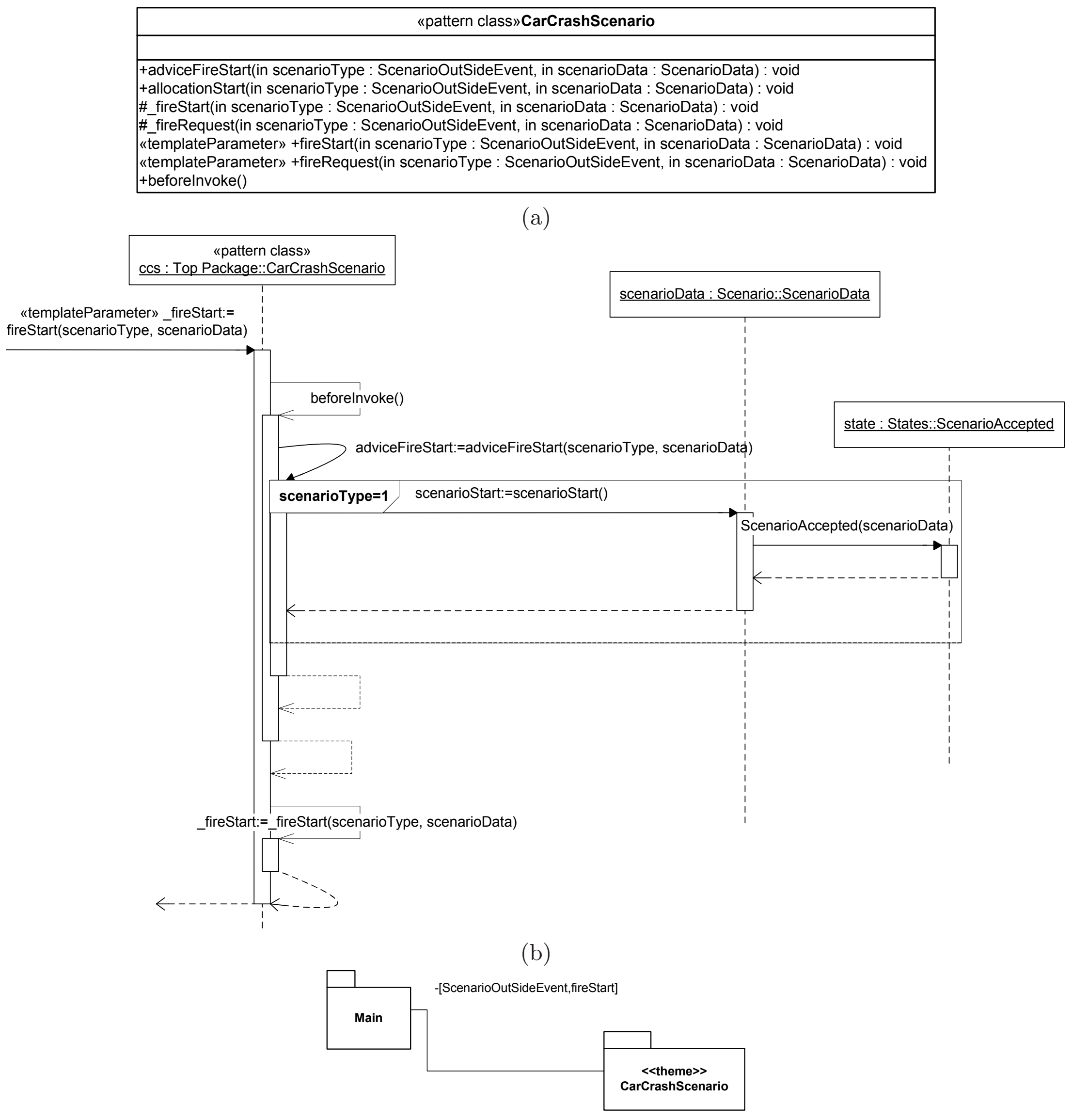
«pattern class»CarCrashScenario
+adviceFireStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
+allocationStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
#_fireStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
#_fireRequest(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
«templateParameter» +fireStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
«templateParameter» +fireRequest(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
+beforeInvoke()
(a)
«pattern class»
ccs : Top Package::CarCrashScenario
scenarioData : Scenario::ScenarioData
«templateParameter» _fireStart:=
fireStart(scenarioType, scenarioData)
beforeInvoke()
state : States::ScenarioAccepted
adviceFireStart:=adviceFireStart(scenarioType, scenarioData)
scenarioType=1
scenarioStart:=scenarioStart()
ScenarioAccepted(scenarioData)
_fireStart:=_fireStart(scenarioType, scenarioData)
(b)
Main
-[ScenarioOutSideEvent,fireStart]
<<theme>>
CarCrashScenario
(c)

Fig. 2. Car Crash Scenario expressed using Theme/UML a) the pattern class *CarCrashScenario*. b) the sequence diagram of for the template parameter *_fireStart()*. c) the binding specification for the theme Car Crash Scenario.

2.5 Incremental evolution: adding scenarios

It is to be expected that, over time, new scenarios will be added to the CMS, and existing ones might evolve as well. Here, we briefly discuss an additional scenario: suppose that there is an accident that involves the president of the nation. In such a case, since the number of resources is limited and may already be assigned to other crises, it may be required to pre-empt resources assigned to low-priority crises. Diagram 3 defines such a scenario. In principle, the crisis is handled in a similar way as discussed in section 2.4. However, if insufficient resources are available, an internal system event is generated, asking all running crises to pre-empt necessary resources, if appropriate. The higher-priority scenario should then be able to allocate those resources.

2.6 Aspect Interference in the crisis management system

In the sections above, we have defined a system that allows the modular specification and evolution of multiple scenarios and separates the coordination specification from modules that implement low-level behavior. However, since these scenarios may be developed independently of each other, by different actors, and since scenarios may also evolve over time, it could easily occur that multiple scenarios interfere with each other.

In the example above, the Presidential Crisis scenario sends an event that asks other scenarios to pre-empt non-critical (to them) resources. However, the Car Crash scenario as defined earlier does not take this into account, and thus does not specify how to respond to such an event. While the small size of our example case makes this conflict relatively easy to discover, interactions among scenarios are more complex and less obvious when more realistically sized projects are considered. Furthermore, the potential for inconsistencies or unintended interactions increases as the system evolves.

Since multiple scenarios may need to react to one event, and each scenario is interested in multiple kinds of events, it becomes very hard to keep track of all potential interactions manually. For this reason, the help of automated tools that can detect (potential) interference is necessary. In the next section, we discuss an approach that facilitates this.

3 A graph-based approach to detecting semantic interference at the design level

UML sequence diagrams include means for modeling the runtime relation between objects and aspects. However, complex software systems usually have many sequence diagrams, making it very hard to manually trace and reason about the runtime behavior. For example, one may need to trace all the possible receivers of a call to reason about the runtime behavior due to polymorphism, or all applicable advice. With model-checking the semantics of the call can be simulated and all the possible receivers and applicable advice are automatically generated.

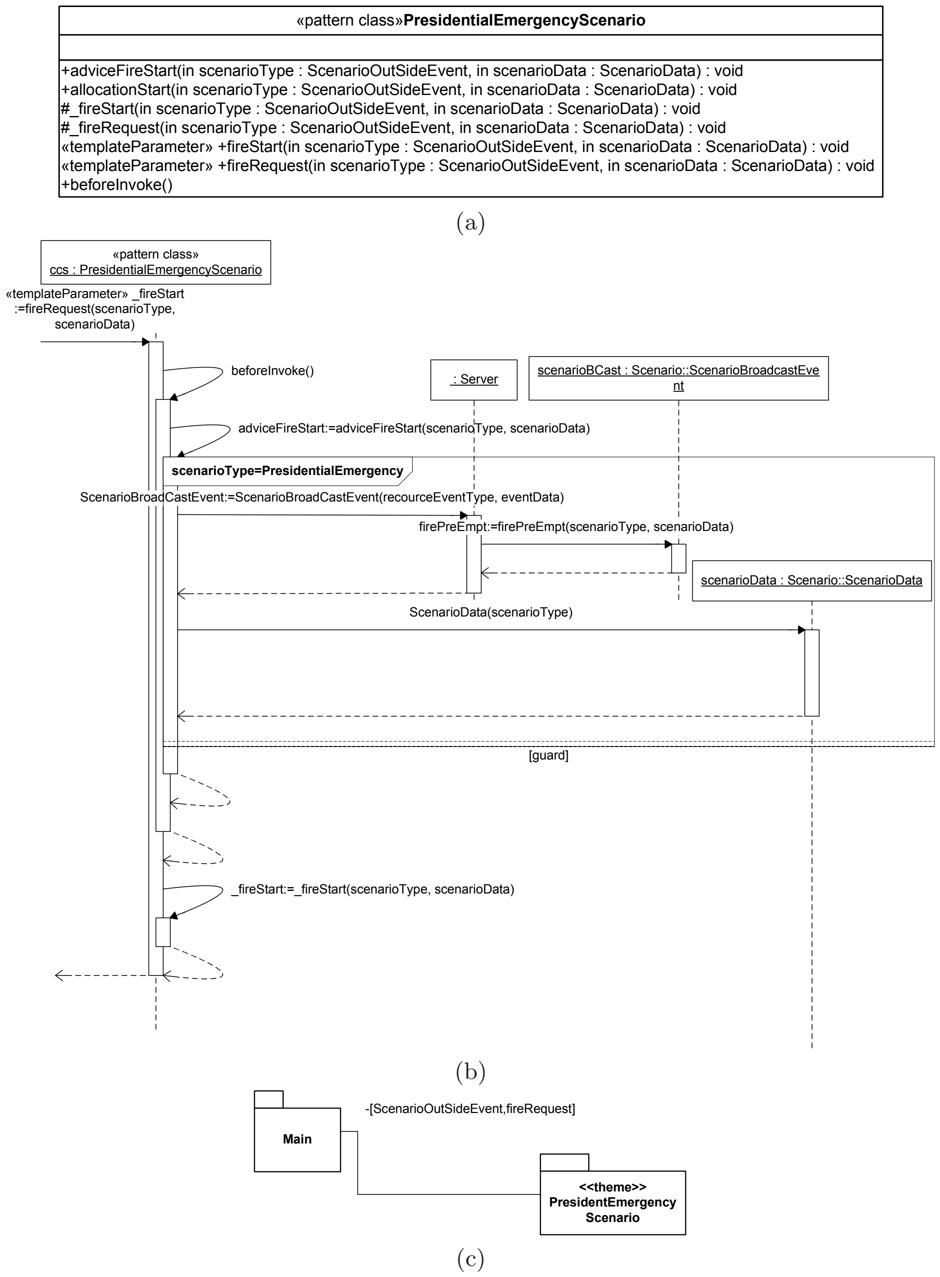
«pattern class»PresidentialEmergencyScenario
+adviceFireStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
+allocationStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
#_fireStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
#_fireRequest(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
«templateParameter» +fireStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
«templateParameter» +fireRequest(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
+beforeInvoke()
(a)
«pattern class»
ccs : PresidentialEmergencyScenario
«templateParameter» _fireStart :=fireRequest(scenarioType, scenarioData)
beforeInvoke()
: Server
scenarioBCast : Scenario::ScenarioBroadcastEvent
adviceFireStart:=adviceFireStart(scenarioType, scenarioData)
scenarioType=PresidentialEmergency
ScenarioBroadCastEvent:=ScenarioBroadCastEvent(recourceEventType, eventData)
firePreEmpt:=firePreEmpt(scenarioType, scenarioData)
scenarioData : Scenario::ScenarioData
ScenarioData(scenarioType)
[guard]
_fireStart:=_fireStart(scenarioType, scenarioData)
(b)
Main
-[ScenarioOutSideEvent,fireRequest]
<<theme>>
PresidentEmergency
Scenario
(c)

Fig. 3. Presidential Emergency scenario expressed using Theme/UML a) the pattern class *PresidentialEmergencyScenario*. b) the sequence diagram for the template parameter *_fireStart()*. c) the binding specification for the theme Presidential Emergency Scenario.

To detect semantic interference at the level of UML models, we use graph-based model-checking, in terms of the GROOVE graph production system. In graph-based model-checking, the behavior of the system is modeled as graph-transformation rules and runtime states of a system are modeled as a graphs. Here, applying a transformation rule results in one or more graphs that represent different states of the system [23]. The graph-production tool automatically applies the transformation rules, which *simulates* the behavior of the modeled system. The simulation generates a state-space (with transitions) showing the possible states the system can reach. The requirements of the system are expressed as temporal logic formulas which are verified over the generated state-space.

Although UML sequence diagrams include much of the information related to the simulation, they lack elements like operation frames that are crucial to achieve a simulation close to actual object-oriented (OO) execution. Therefore, we defined a graph-based model that is an OO-like runtime representation of UML sequence diagrams. We modeled graph-transformation rules that add OO execution semantics, like polymorphism, to the UML sequence diagrams so that the runtime relation can be simulated with graph-based model checking. Similarly, we also modeled the effects of aspect weaving as performed by an aspect-oriented runtime system.

This section details the application of graph-based model-checking to semantic interference detection. In the next subsection, the graph-based model for representing UML sequence diagrams is explained. Subsection 3.2 details how aspects are modeled with this graph-based model. In subsection 3.3 examples of graph-transformation rules modeling object- and aspect-oriented execution semantics are presented. Finally, subsection 3.4 explains how execution sequences can be expressed with temporal logic formulas and how these are verified.

3.1 Design Configuration Modeling Language

UML sequence diagrams depict execution sequences in order to provide an overview of the interactions between objects in software systems. Due to mechanisms such as conditional executions and polymorphism, a software system may support executions other than the ones depicted by the sequence diagrams. These hidden interactions may introduce bugs to the software when the sequence diagrams are implemented. In order to prevent the introduction of these bugs to the software system, there should be a way to reason about the executions supported by the diagrams. This reasoning requires the sequence diagrams to be simulated as close as possible to the actual execution of object-oriented software – as we will show in next subsection, this carries over to aspect-orientation. However, sequence diagrams do not include model elements like execution frames that allow an OO-like execution simulation.

The graph-based Design Configuration Modeling Language (DCML) includes these elements and allows one to model an OO software runtime for UML sequence diagrams. In our approach, the DCML models (DCMs) are generated from one UML class diagram and at least one sequence diagram. We added the capability of exporting UML class and sequence diagrams as DCML models to the open-source UML editor ArgoUML [1].

Figure 4 shows the meta-model of DCML. In this figure *Var* stands for variable, *Decl* stands for declaration, *Impl* stands for implementation and *Oper* stands for operation (DCML uses the term "operation"; one could also read "method"). The DCML meta-model has two parts, the structure part and the dynamic part. Both parts and their elements are quickly presented in the following. For a more detailed discussion with additional examples, we refer to the appendix, which describes the DCML elements in detail.

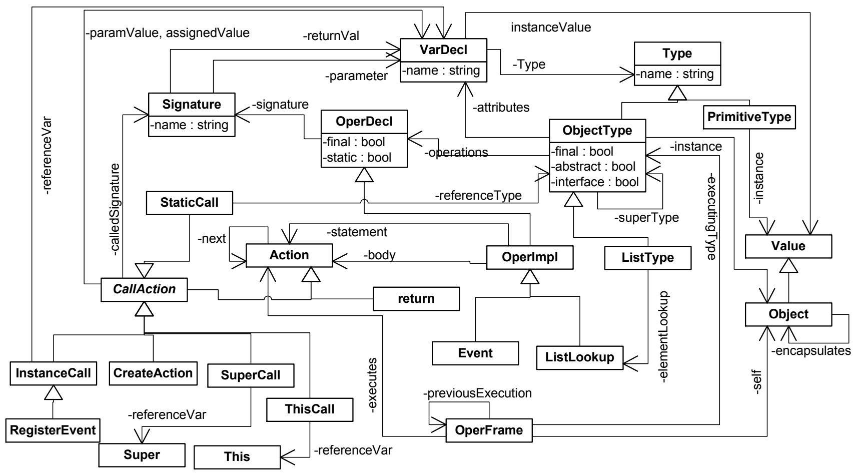

Fig. 4. The DCML meta-model

Structure part of DCML. The structure part covers the elements of the meta-model for modeling the classes, the interfaces and the relations between these. This part is generated from the class diagram. Because classes and interfaces are types at runtime, they are represented with nodes labeled *ObjectType* (object-type nodes). If the object-type node is representing an interface, then the attribute interface is set to true. The equivalent of the generalization relation is the edge labeled super-type. Figure 5-(a) shows a portion of the class diagram from the CMS with three classes, namely *State*, *ResourceAllocation* and *ScenarioData*. Figure 5-(b) shows the DCML representation of this class diagram; here, the three object-type nodes represent the classes in the class diagram. For example, the object-type node with the attribute name *ResourceAllocation* (i.e. the name of object-type node) is the class with the same name represented in DCML. The class *ResourceAllocate* generalizes the class *State* in the class diagram. This is shown in DCML with the edge labeled *SuperType* connecting the object-type nodes representing these classes.

Further kinds of nodes, edges, and specializations in the structure part are:

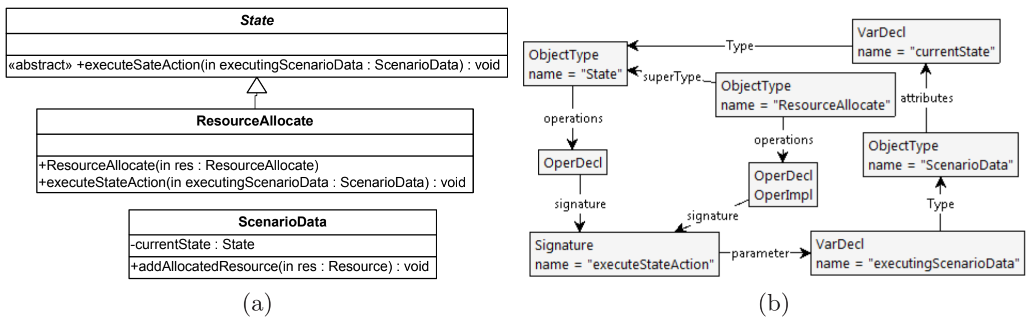
State
«abstract» +executeSateAction(in executingScenarioData : ScenarioData) : void
ResourceAllocate
+ResourceAllocate(in res : ResourceAllocate)
+executeStateAction(in executingScenarioData : ScenarioData) : void
ScenarioData
-currentState : State
+addAllocatedResource(in res : Resource) : void
(a)
ObjectType
name = "State"
Type
VarDecl
name = "currentState"
superType
ObjectType
name = "ResourceAllocate"
attributes
operations
OperDecl
operations
OperDecl
OperImpl
ObjectType
name = "ScenarioData"
signature
signature
Type
Signature
name = "executeStateAction"
parameter
VarDecl
name = "executingScenarioData"
(b)

Fig. 5. a) An example UML class diagram. b) The DCML model of the class diagram shown in (a)

Node *VarDecl* – a variable declaration.
Edge *Type* – connects a variable declration node with a type node to model the variable's type.
Edge *attributes* – connects an object-type node with variable-declaration nodes which represent the type's attributes.
Node *OperDecl* – an operation declaration.
Specialization *OperImpl* – added to *OperDecl* nodes for operation with implementation.
Edge *operations* – connects an object-type node with operation-declaration nodes which represent the type's operations.
Node *Signature* – a unique operation signature.
Edge *parameter* – connects a signature with the variable declarations that represent the signature's parameters.
Edge *returnType* – connects a signature with a type node that represent the signature's return type, if it has one.

Dynamic Part of DCML. The dynamic part, which is generated from the sequence diagrams, covers the elements for modeling the objects, the values and the life-lines operations. A life-line in a sequence diagram shows the actions the object executes when it receives a call. In the DCML meta-model (Figure 4), the specializations of the abstract element *Action* represent the actions of sequence diagrams. For example, the nodes labeled *CallAction* represent call actions and the nodes labeled *ReturnAction* represent return actions. An action node can be connected to another action node by an edge labeled *next*; in this way, the order between the actions of a life-line is represented in DCML. The first action of a life-line is connected to an operation implementation node by an edge labeled *body* in DCML to show that these actions are executed when this operation received a call.

The sequence diagram presented in Figure 6 shows the life-line of the operation *addAllocatedResource()*. The first action executed in this life-line is a call action. This action is followed by a return action where the operation *addAllocatedResouce()* returns. Figure 7 shows the DCML model generated from this sequence diagram (and the class diagram in Figure 5). In this figure, the emphasized node represents the call action belonging to the life-line of the operation *addAllocatedResource*. Because in the sequence diagram this call action is the first action executed in the life-line of the operation *addAllocatedResource()*, the emphasized node is connected to the operation implementation node representing the operation *addAllocatedResource()* by an edge labeled *body*. This call action node is connected to the signature node named *executeStateAction* by an edge labeled *calledSignature* to show that the call action is to the signature *executeStateAction*. Following the outgoing edge labeled *next* from the call action node, it can be seen that the call action is succeeded by a return action.

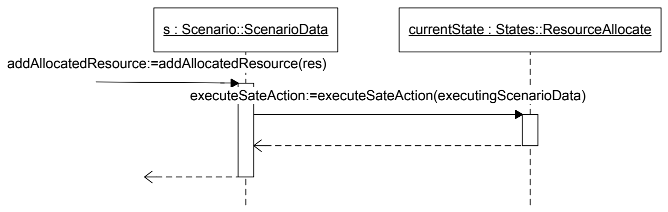
s : Scenario::ScenarioData
currentState : States::ResourceAllocate
addAllocatedResource:=addAllocatedResource(res)
executeSateAction:=executeSateAction(executingScenarioData)

Fig. 6. A sequence diagram showing the actions executed by the operation *ScenarioData.addAllocatedResource()*

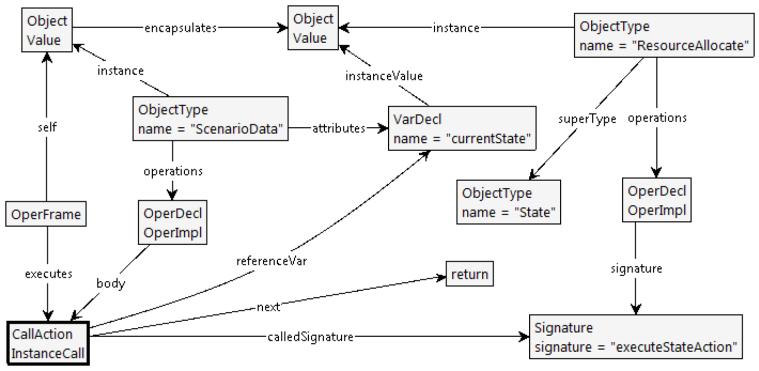

Fig. 7. A snapshot from the simulation of the sequence diagram shown in Figure 6

The frame of an executing operation is represented by nodes labeled *OperFrame* in DCML. These nodes are used to identify, during simulation, the object that is currently executing, the scope of the executing object, the type that contains the called method and the statement that is being executed. When

UML diagrams are converted to DCML models, the conversion algorithm automatically adds the operation frame node which marks the first action of the sequence diagram as the action that is being executed. Thus, the simulation starts executing from that action.

Further kinds of nodes, edges, and specializations in the dynamic part are:

Specialization *InstanceCall*, *CreateOper*, *SuperCall*, *ThisCall*, *StaticCall* – added to *CallAction* nodes to distinguish between calls to instances, object creation, calls to the super implementatoin, self calls and calls to static operations.
Edge *referenceVar* – connects a call-action node with a variable declaration node to show that the operation is invoked on this value.
Node *Value* – represents a value.
Specialization *Object* added to *Value* nodes to specify that the value is an object.
Edge *instanceValue* – connects a variable-declaration node with a value node which represents the variable's value.
Edge *self* – connects an operation-frame node with object node representing the active object during the execution of the frame.

A DCML model can be generated from more than one sequence diagram and, thus, a variable can have more then one instance value. During simulation, the values of the variables at the executing frame are resolved with the encapsulated edges.

3.2 Aspects in the Design Configuration Modeling Language

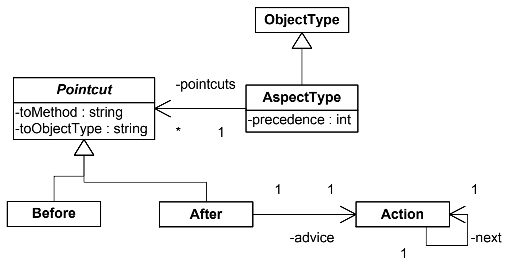

Fig. 8. The meta-model containing elements for representing aspects in DCML models

DCML treats aspects as a specialization of the object-types called aspect-types which are shown as nodes labeled *AspectType* (aspect-type nodes). Because of this specialization, it is possible to specify attributes and operations for aspect-types. Figure 8 shows elements used for representing aspects in DCML models. It is possible to give precedence to aspect-types in DCML; if a precedence value is given to an aspect, then the integer attribute *precedence* is set to the given precedence value. Two kinds of pointcuts can be modeled in DCML: before pointcuts (nodes labeled *Before*) and after pointcuts (nodes labeled *After*). A pointcut node connected by an edge labeled *pointcuts* to an aspect-type node shows that the pointcut is declared in the scope of that aspect-type. The pointcut specification consists of the name of the class and the name of the operation which is to be intercepted; the two string attributes of the pointcut nodes hold these names. The action node that is connected to a pointcut node by an edge labeled *advice* represents the beginning of the advice actions.

UML-based AOM approaches extend the UML meta-model, thus, in order to use our approach for conflict detection in an AOM model, a mapping from these extensions to the DCML elements must be provided. Thereby, it is necessary that the aspect-oriented extensions can be captured by the DCML elements shown in Figure 8. These aspect-oriented elements are simple primitives and potentially require the AOM front-end to resolve parts of the model. For Theme/UML, which we choose to model the design of the CMS, the mapping to DCML is realized as shown in the itemization below. Theme/UML could be mapped to DCML without the need of resolving parts of its models.

– The pattern classes are represented as aspect-types. Currently, the execution semantics we modeled do not support aspect instances; thus, the operations and the attributes declared in template classes are converted to static operations and attributes of the aspect-types.
– The operation *beforeInvoke()* of the template operations is represented as before pointcut. Similarly, the operation *afterInvoke()* is represented as after pointcut. These pointcuts are added to the aspect-type node representing the template class; that is, they are connected to the aspect-type node representing the template class by the edge labeled *pointcuts*.
– The life-line of the operations *beforeInvoke()* and *afterInvoke()* is represented as the advice of a before or after pointcut. That is, the node representing the first action executed in the life-line of these operations is connected to the pointcut node with an edge labeled *advice*.
– The names in the binding specification are converted to the values of the attributes *toMethod* and *toClass* of a pointcut node. We also support to match names against patterns rather than just comparing them for equality.

Figure 9 shows the DCML model of the "theme" *CarCrashScenario, _fireAdviceStart*; the Theme/UML diagrams of this "theme" is shown in Figure 2. Here, the pattern class *CarCrashScenario* is represented by the aspect-type node that has the same name. Looking at the sequence diagram of this theme, it can be seen that the operation *beforeInvoke()* is called after the invocation of the template method *_fireAdviceStart()*; so, a before pointcut is added to the aspect-type *CarCrashScenario*. In the life-line of the operation *beforeInvoke()* first a call action is executed, then the operation returns. In the DCML model of this theme, the call action node and the return action node (labeled *return*) represent these actions. Because the call action is executed first in the life-line, the before pointcut node is connected to the call action node by the edge labeled *advice*.

The specification presented in Figure 2-(c) states that the "theme" *CarCrashScenario* should be bound to the class *ScenarioOutSideEvent* and to the method *fireStart()*. Following this, the attribute *toObjectType* of the before pointcut node is set to *ScenarioOutSideEvent* and the attribute *toMethod* of the same node is set to *fireStart*.

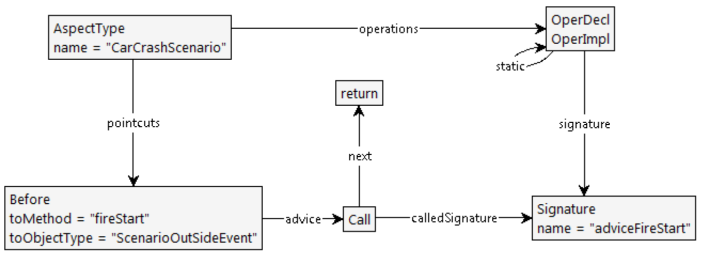

Fig. 9. The DCML model of the "theme" *CarCrashScenario*

3.3 Execution Semantics via Graph Transformations

A DCM is simulated by automatically triggering the appropriate graph-transformation rules that represent the OO and aspect-oriented execution semantics of the UML models. We formed a graph-production system (a collection of graph transformation rules [23]), consisting of *57* graph transformation rules that model the following execution semantics for UML models: operation dispatch, parameter passing, returning a value, object creation, before and after pointcuts. Due to space limitations, we only detail the transformation rules modeling the semantics for operation dispatch and before pointcuts in this section (interested readers can download the graph production system [2]). In the following, we present an overview on graph transformations and how they are modeled in GROOVE.

Graph Transformations in GROOVE A graph transformation rule has a left-hand side, L, a right-hand side, R and a set of negative application conditions N. The rule transforms a source graph G to a target graph H by searching for an occurrence of L in G where N does not occur. In order to say L occurs in G all the nodes and edges in L should also be found in G [14]. When L of a transformation rule occurs in G where N does not occur then the transformation rule is said to match; a rule can have multiple matches. For each match, L is replaced with R which results in the transformed graph H.

In GROOVE, both the left-hand and the right-hand side of a graph transformation rule are represented in the same graph. The modifications the rule applies to the host graph are specified using keywords:

– The keyword *new* (or the color green and solid bold lines) is used for the edges/nodes that are added. These nodes are not in the left-hand side of the rule but are in the right-hand side of the rule.
– The keyword *del* (or the color blue and dashed thin lines) is used for the edges/nodes that are deleted. These nodes are in the left-hand side of the transformation rule but are not in the right-hand side.
– The keyword *not* (or the color red and dashed bold lines) is used for negative application conditions [18]; these edges/nodes should not exist in the part of the host graph where the left-hand side of the transformation rule exists.
– All other edges/nodes are both in the left-hand side and right-hand side of the transformation rule.

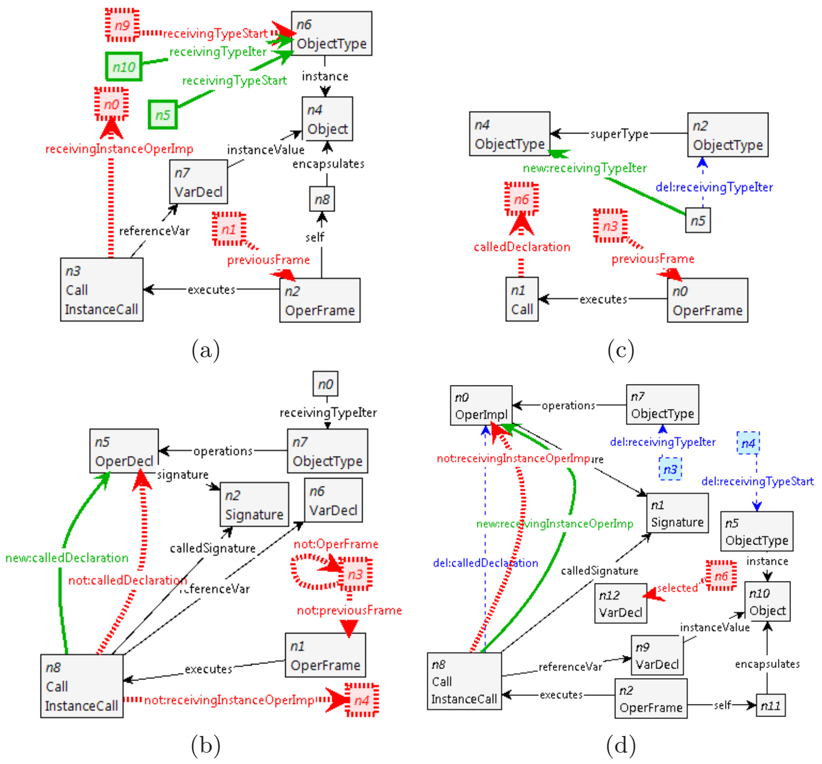

Fig. 10. Graph transformation rules for finding the newest implementation of the called operation: (a) calculates the target reference type and marks it (b) finds the latest declaration of the operation (c) moves the mark up one level in the inheritance hierarchy, (d) checks whether the latest declaration implements the operation.

Execution Semantics for Operation Dispatch A call action requires certain type-checks to be enforced at compile-time. UML editors also employ similar checks so that the call is made to a compatible type. We assume that these static checks are enforced and the call action is valid.

The execution semantics for operation dispatch consists of finding the latest implementation of the operation in the inheritance hierarchy and passing the arguments that are executed in the following manner: 1) calculating the type of the object the reference variable is holding; that is, the reference type of the call 2) starting from the reference type traversing the inheritance hierarchy upwards until an object-type that declares the operation is found 3) passing the arguments 4) checking that the latest declaration implements the operation. Figure 10 presents the 4 transformation rules that realize steps 1, 2 and 4 of the operation dispatch. For brevity, step 3 is not detailed further. Below we describe how these steps are realized by the four transformation rules of the figure:

1. The rule in Figure 10-(a) is used for finding the reference type of the call. In sequence diagrams, the reference variables of the call actions are only variable declarations. So, it is sufficient to find the object-type whose instance this reference variable is holding to identify the reference type. In this figure, the reference variable is node $n7$ (i.e. the variable declaration node that is connected to the call node with an edge labeled *referenceVar*) and the object it is holding is node $n0$. For this rule to match, the reference variable's value at the current operation frame should be an object and this object should be connected to an object-type node with an edge labeled *instance*. If, for example, the reference variable does not hold an object, then the call cannot continue. The transformation rule adds two nodes and edges. From these, the edge labeled *receivingTypeStart* marks the object-type from which the traversal in the inheritance hierarchy starts. The edge labeled *receivingTypeIter* marks the object-type that is traversed. Since the reference type of the call is the type the traversal starts from and since it is the first type to be traversed, these edges are connected to the object-type node that is the reference type of the call.

2.1 The rule in Figure 10-(b) marks the latest declaration of the operation. If the traversed object-type contains an operation declaration node that has the same signature as the called signature then this operation declaration is the latest declaration of the operation. In the depicted transformation rule, the traversed object-type node is node $n7$ and the called signature is node $n2$. The rule matches when the traversed type has an operation declaration node ($n5$) that is connected to the same signature node as the called signature. The rule marks the declaration by adding an edge labeled *calledDeclaration* between the call node ($n8$) and the operation declaration node.

2.2 If the traversed object-type (i.e. the object-type where the edge labeled *receivingTypeIter* is pointing to) does not have the operation declaration then its super-type should be traversed. The transformation rule in Figure 10-(c) deletes the edge labeled *receivingTypeIter* and adds another edge with the same label pointing to the super-type of the traversed object-type. This rule has a lower priority then the rule presented in the previous step, therefore, they do not match at the same time. The rule in the previous step deletes the edge labeled *receivingTypeIter* if the traversed object-type has the operation declaration. That label is required to match this rule, therefore, the traversal stops in this case.

4. After finding the operation declaration and preparing the arguments, the operation can be dispatched. However, before dispatching, we must be sure that the operation is implemented. The transformation rule in Figure 10-(d) matches when the operation declaration node marked in step 2 is also an operation implementation node (i.e. that is also labeled *OperImpl*). When this rule matches, it marks the operation implementation to be ready for dispatch by adding the edge labeled *receivingInstanceOperImpl*. The previous rule could also be modeled in a way that the traversal would search for the operation implementation. However, we made this a separate transformation rule because at run-time parameter passing is done after the operation is located and before the operation is dispatched.

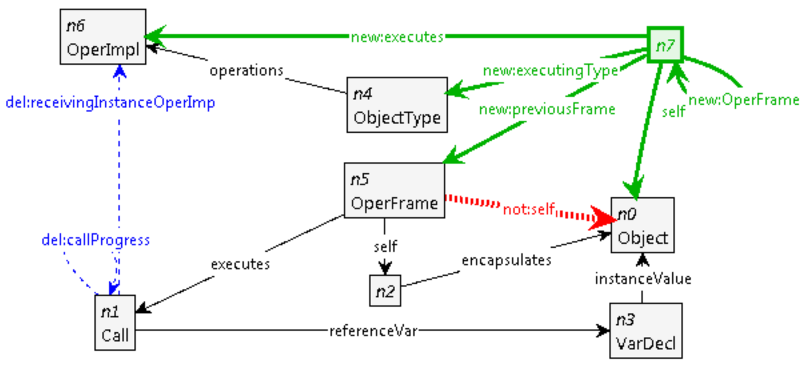

Fig. 11. Graph transformation rule that dispatches the operation after the object-type that implements the operation is discovered by the rules presented in Figure 10.

After the object-type that implements the operation is discovered, the operation can be dispatched as presented by the graph transformation rule in Figure 11. Here, the dispatching is done by creating a new operation frame node ($n7$), that is connected to the dispatched operation implementation (the node labeled *OperImpl*) with an edge labeled executes. The *self* of the new frame is the object on which the operation is called; thus, the rule adds the edge labeled *self* between the newly added frame ($n7$) and the object the reference variable holds ($n0$). The executing type of the new frame is the object-type that implements the operation ($n4$). The frame from which the call is initiated from is connected to the new frame with an edge labeled *previousFrame*. With this edge, the frame that will be returned when the execution of the called operation finishes is marked.

Execution Semantics for Before Pointcut We modeled an execution pointcut model where the aspects intercept the entry and the exit points of the operations. As discussed before, the pointcuts in DCML specify the name of the operation that is going to be intercepted. The execution semantics of a before pointcut evaluates whether the simulation is at the entry point of the operation specified in the pointcut. If this evaluation yields true, then the advice code of the pointcut is executed before the operation. Similarly, the semantics for the after pointcut, evaluate whether the simulation has finished executing the operation specified by the pointcut. The advice code of these pointcuts is executed just after the return action of the intercepted operation.

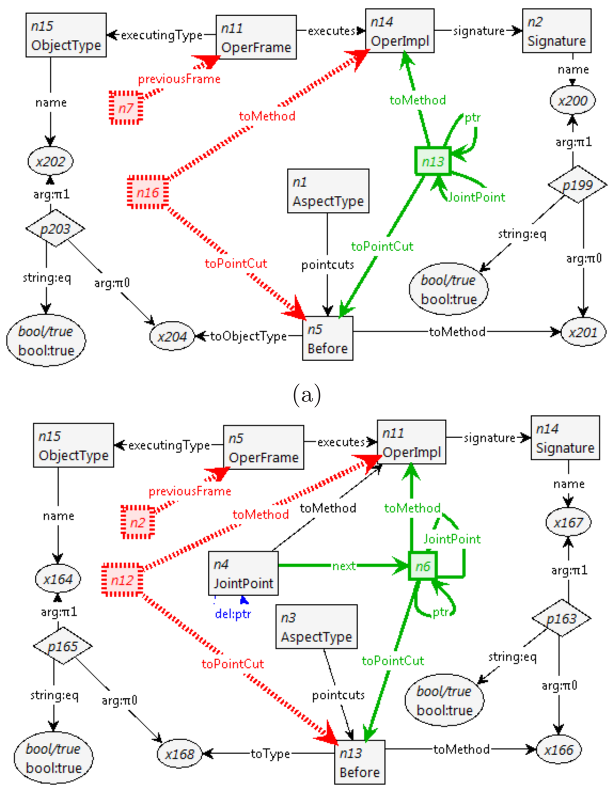

Fig. 12. The transformation rule modeling the semantics of before pointcut

Figure 12 shows the two transformation rules that identify the join points for before pointcuts. At the entry point of an operation, the transformation rule shown in Figure 12-(a) evaluates whether there is a before pointcut that can intercept this operation. If there is, then it marks the intercepted operation and intercepting pointcut with a node labeled *Joinpoint* (joint point node), node *n8*. To evaluate whether a before pointcut can intercept the operation, the rule checks for two conditions: 1) the execution should be at an entry point of an operation 2) The name of this operation and the name of the object-type should match to the names specified in the pointcut. These conditions are realized by the transformation rule as follows:

- The entry point of an operation, in DCML, is the point where the program counter, the edge labeled *executes*, connects the operation frame node to an operation implementation node. The left-hand side of the transformation rule shown in Figure 12-(a), includes the nodes *n2*, *n9* and the edge labeled *executes*. The node *n2* represents the operation frame that is currently executing. This node is connected to the operation implementation node *n9* with the edge labeled *executes*. These nodes and the edge represent the entry point of an operation. Since, they belong to the left-hand side of the rule, the rule only matches when the simulation is at the entry point of an operation.
- The transformation rule uses attribute operations to evaluate the pointcut at an entry point of an operation. In the transformation rule, the nodes *x200* and *x201* represent the name of the operation and the value of the attribute *toMethod* respectively. These nodes are generic value nodes and they can match to any value of the attribute. The node *p199* is a production node; this is the node where the attribute operation is specified. The outgoing edge labeled *string:eq* specifies the attribute operation is string comparison. This edge is connected to the attribute node holding the value *bool:true*; this states that the two string arguments of the operation should be equal for the rule to match. The outgoing edges from a production node whose labels start with *arg* are used for specifying the arguments of the attribute operation. The arguments of the production node *p199* are specified as the name of the operation (node *x200*) and the value of the attribute *toMethod* (node *x201*) with these edges. In this way, the rule evaluates whether the name of the operation to be executed is the same as the name in the pointcut specification. The evaluation of the name of the object-type is also realized using the string comparison attribute operation (this attribute operation is specified in the production node *p203*). Thus, the transformation rule only matches when the string values of the pointcut specification are equal to the object-type and operation that is to be executed.

The transformation rule shown in Figure 12-(b) evaluates whether there are other aspects that can intercept the same operation as the aspect identified by the rule presented in Figure 12-(a). This rule also uses attribute operations to evaluate the pointcut. The main difference between this rule and the rule presented in Figure 12-(a) is that this rule forms a list of join points. Assume that there are two aspects that can intercept the currently executing operation. The transformation rule shown in Figure 12-(a) identifies one of these aspects and marks it by adding a join point node. Then, the transformation rule shown in Figure 12-(b) identifies the second aspect and adds another join point node (node *n6*). However, this join point node is connected to another join point node (node *n4*) by an edge labeled *next*. In this way, a list with two join points is formed. The beginning of the join point list is always the join point added by the transformation rule of Figure 12-(a). This rule extends the join point list by adding items to the end of the list. The edge labeled *ptr* is used for marking the last item in the join point list. The rule connects the edge labeled *next* to a join point node that has the self-edge labeled *ptr*. Since the newly added join point is now the last item on the pointcut list, the rule adds the self-edge labeled *ptr* to the new join point node (node *n6* and deletes it from the previous join point node (node *n4*). Note that, due to prioritization of the rules, when the transformation rule shown in Figure 12-(b) matches, the rule in Figure 12-(a) cannot match.

Note that the transformation rules shown in Figure 12 work on aspects that do not specify a precedence. The execution semantics for aspects with precedence are handled by another two transformation rules. These two rules also attribute operations to compare the precedence value of the aspect-type nodes.

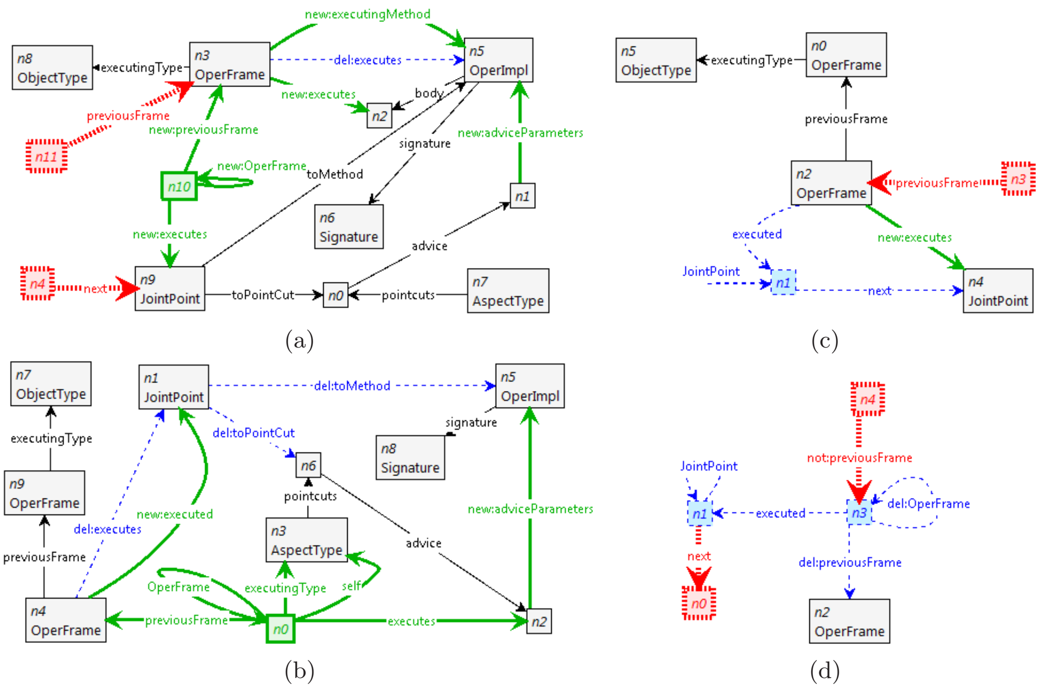

Fig. 13. a) Transformation rule for starting the execution of the join point list. b) The transformation rule that starts the execution of the advice code for a pointcut. c) The transformation rule that advances to the next join point in the join point list. d) The transformation rule that resumes the execution of the intercepted operation.

After the join points are identified and the join point list is formed, the join point list is traversed and the advice code of the pointcuts is executed.

The execution semantics of these are modeled in the 4 transformation rules shown in figure 13. The transformation rule shown in Figure 13-(a) adds a new operation frame, node *n10*, for traversing the join point list. This operation frame is connected to the first join point of the join point list, node *n9*, with edge labeled *executes*; thus, the dispatching of the advices start form this join point. The operation frame node *n3* is the frame of the operation that is intercepted by the aspects. The edge labeled *previousFrame* connects the new operation frame to this node so that when the traversal of the join point list is finished the intercepted operation can resume its execution.

The transformation rule shown Figure 13-(b) dispatches the advice code for the current join point. This is the join point node connected to the operation frame node *n4* with the edge labeled *executes*. The dispatching is realized by adding an operation frame node, node *n0*. The self of this new operation frame node is the aspect-type node, node *n3*, where the pointcut is declared. The node *n2* represents the first action of the advice. The new operation frame is connected to this node with the edge labeled *executes*; thus, the execution of the advice code starts from this action. The edge labeled *previousFrame* is connected to frame where the join point dispatched the advice. When the advice code returns, the execution is given back to this operation frame; so the traversal of the join point list resumes.

When the advice code returns, the transformation rule shown in Figure 13 matches. This rule advances to the next join point in the list. When there are no more items to be traversed in the join point list, the transformation rule in Figure 13 matches. This rule deletes the operation frame in which the join point list is traversed and resumes the execution of the intercepted operation.

3.4 Verification of Execution Sequences

When simulating the execution of a DCM, GROOVE generates the space of states in which the program can be as well as all possible sequences of these states. Next GROOVE verifies if specified constraints are violated by at least one execution sequence; in this case, it has detected a conflict. To generate the execution sequences, it iteratively applies the graph-transformation rules constituting the operational semantics, as discussed in the previous subsection, to the DCM of the design. The result of this simulation is represented as a so-called *graph transformation system* (GTS) [23]. A GTS is, again, a graph, where the nodes represent distinct runtime states and the directed edges represent graph-transformation rules that were applied to transition from one state to the other. When multiple rules can be applied at a certain state, one edge is created for each rule.

Figure 14 shows an excerpt from a GTS demonstrating the simulation of a UML models of the CMS with the aspects *CarCrashScenario* and *PresidentialAccidentScenario*. The execution starts in the state labeled *start*. The labels of the transitions are the names of the applied graph transformation rules. It is important to note that some of the labels are parameterized, in particular, these are *executes(object-type name)*, *executeMethod(operation name, object-type name)* and *returnframe(operation name, object-type name)*. A rule with a parameterized name, specifies a set of node attributes whose values should be outputted in the transition labeled instead of the parameters. When applied, GROOVE extracts the values of from the target graph and replaces the parameters with the extracted value. We use this mechanism to learn about the operations/aspects that have executed during simulation. For example, the edge between nodes *start* and *S2* is labeled *executeMethod("fireStart", "ScenarioOutSideEvent")* shows that the simulation is at the entry point of the operation *ScenarioOutSideEvent. fireStart()*.

In the sample GTS, we see that after state *S2* the simulation continues in two branches. Each of these branches start with a transition labeled *beforePointCutStart*. This label is the name of the transformation rule shown in Figure 12-(a). The design of the CMS contains two aspects and both of these aspects specify a before pointcut to the operation *fireStart()*. So the transformation rule *beforePointCutStart* can match at two different places; one for aspect *CarCrashScenario* and the other one for aspect *PresidentialEmergencyScenario*. Application of this rule to one of these aspects, adds a join point node specific for that aspect. This in turn causes the branching in the GTS. In fact, when more than one aspect specifies a pointcut to the same operation, each application of the rule *beforePointCutStart* to one of these aspects adds a branch to the GTS.

Because there are two aspects with a pointcut to the same operation, the transition labeled *beforePointCutStart* is followed by the transition labeled *beforePointCutNext* once in each branch. The label *beforePointCutNext* is the name of the transformation rule shown in Figure 12-(b) and as discussed before it adds join points to the end of the join point list. Here, this rule matches once in each branch because after the application of the transformation rule *beforePointCutStart* there is another aspect that has a pointcut to the operation *fireStart()*.

It can be seen from the transition between state *S15* and *S17* that at the left branch first the advice of the aspect *PresidentialEmergencyScenario* is executed. Thus, the transformation rule *beforePointCutStart* has matched to this aspect in this branch and the join point node marking this aspect is the first item in the join point list. Following this, the next item in the join point list should be the join point of the aspect *CarCrashScenario*. This can be confirmed at the sample GTS where after advice code of the aspect *PresidentialEmergencyScenario* returns (i.e. after the transition labeled *returnframe("adviceFireStart", "PresidentialEmergencyScenario")*) there is a transition labeled *executeMethod("fireStart", "CarCrashScenario")* between the states *S43* and *S47* in the left branch.

The left branch is further divided into two branches after the state *S17*. As can be seen from the sequence diagram shown in Figure 3-(c), the execution of the operation *adviceFireStart* has two paths depending on the condition on the parameter *scenarioType*. If this parameter is equal to *PresidentEmergencyScenario*, then the actions within the frame fragment is executed. On the other hand, if the parameter is not equal to *PresidentEmergencyScenario* then the operation returns (i.e. the actions within the frame fragment is not executed). The transformation rule named *ConditionalAdapt* matches when there is conditional paths

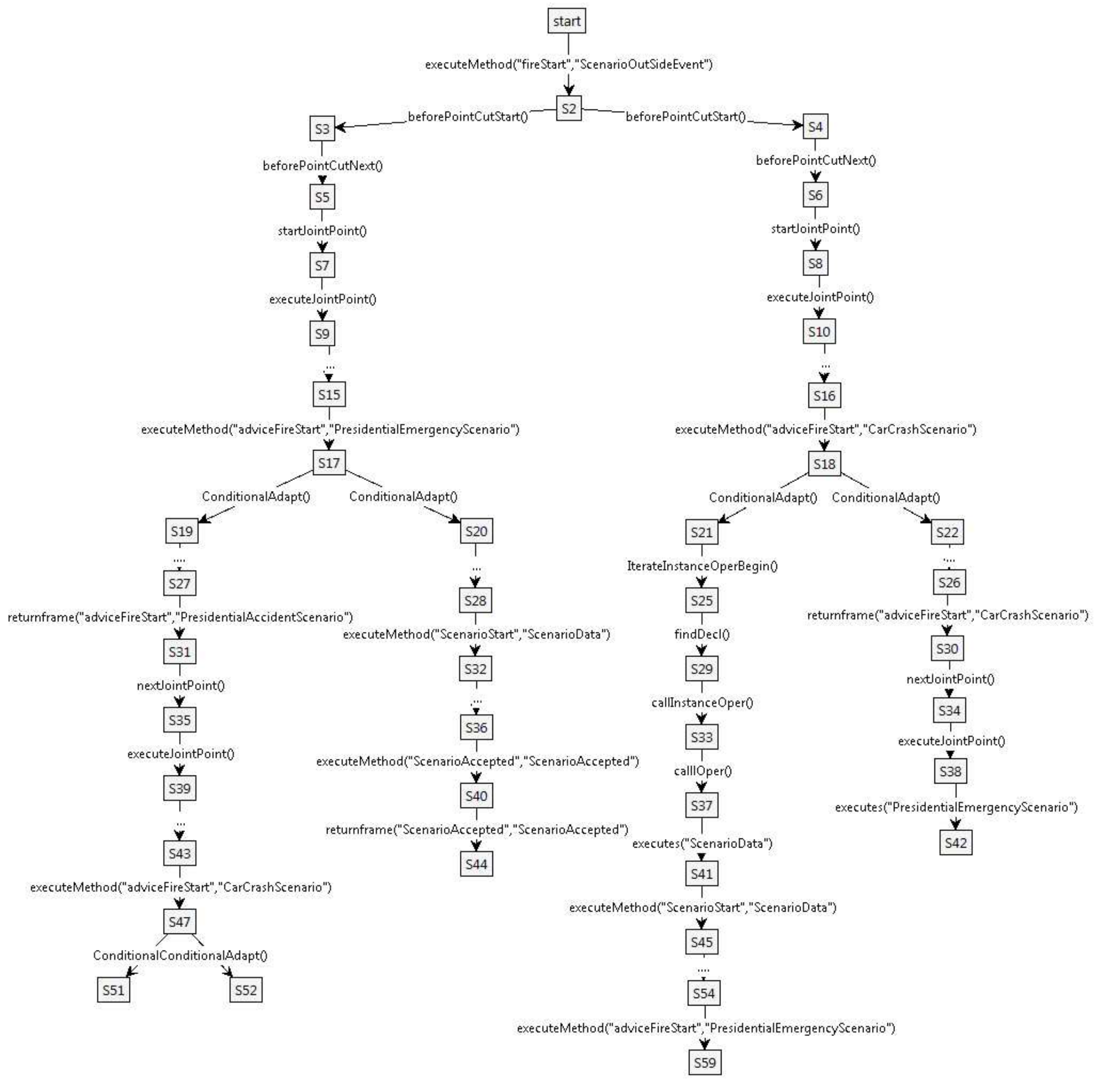
start
executeMethod("fireStart","ScenarioOutSideEvent")
S2
beforePointCutStart()
beforePointCutStart()
S3
S4
beforePointCutNext()
beforePointCutNext()
S5
S6
startJointPoint()
startJointPoint()
S7
S8
executeJointPoint()
executeJointPoint()
S9
S10
...
...
S15
S16
executeMethod("adviceFireStart","PresidentialEmergencyScenario")
executeMethod("adviceFireStart","CarCrashScenario")
S17
S18
ConditionalAdapt()
ConditionalAdapt()
ConditionalAdapt()
ConditionalAdapt()
S19
S20
S21
S22
....
...
IterateInstanceOperBegin()
...
S27
S28
S25
S26
returnframe("adviceFireStart","PresidentialAccidentScenario")
executeMethod("ScenarioStart","ScenarioData")
findDecl()
returnframe("adviceFireStart","CarCrashScenario")
S31
S32
S29
S30
nextJointPoint()
callInstanceOper()
nextJointPoint()
S35
...
S36
S33
S34
executeJointPoint()
executeMethod("ScenarioAccepted","ScenarioAccepted")
callOper()
executeJointPoint()
S39
S40
S37
S38
...
returnframe("ScenarioAccepted","ScenarioAccepted")
executes("ScenarioData")
executes("PresidentialEmergencyScenario")
S43
S44
S41
S42
executeMethod("adviceFireStart","CarCrashScenario")
executeMethod("ScenarioStart","ScenarioData")
S47
S45
ConditionalConditionalAdapt()
...
S51
S52
S54
executeMethod("adviceFireStart","PresidentialEmergencyScenario")
S59

Fig. 14. Excerpt from a graph transition system showing advice execution and a conditional exeuction in the advice.

in the execution. The application of this rule picks one of the conditional paths as the execution path; each application of this rule adds a branch to the GTS. So, it can be seen that after state *S20* the actions within the frame fragment is executed because the operation *ScenarioData.ScenarioStart()* is executed in this branch. The branch after the state *S19* constitutes to the execution path where the parameter *scenarioType* is not equal to *PresidentEmergencyScenario*.

Since the GTS contains all possible execution sequences, in order to identify conflicts, it must be checked whether one path of the GTS violates an invariant. Computation Tree Logic (CTL) is a suitable formalism to specify constraints that have to be satisfied by every path in the GTS, i.e., which must be invariant during all possible executions of the system. If there is at least one path that violates the invariant, the verification fails. GROOVE implements an algorithm to evaluate GTS against CTL formulae.

In a CTL formula, the transition labels from the GTS and their parameters are used as operands. Together with the CTL operators, execution sequences can be specified which are required to occur. For example, the operator *EF* specifies a label which eventually has to occur in a successive state. The CTL formula below specifies that an edge with the label *executeMethod("adviceFireStart", "CarCrashScenario")* must eventually be followed by an edge *returnframe("adviceFireStart", "CarCrashScenario")*. In natural language, this means than after the advice *CarCrashScenario.adviceFireStart* has started executing it must eventually terminate normally. In the GTS shown in Figure 14, this formula is only satisfied in the right-hand execution sequence, therefore, the overall system violates the invariant.

$$EF(executeMethod("adviceFireStart","CarCrashScenario") \wedge \\ (EF(returnframe("adviceFireStart","CarCrashScenario")))$$

4 Application of our approach to the case study

While adding a new scenario, it is important that the modified parts do not violate the invariants of the software system. In order to prevent resulting errors, whether the invariants are violated or not must be verified. Especially, great attention should be paid to verifying invariants of mission-critical systems like a CMS because errors could have catastrophic effects.

The CMS software can be deployed at different crisis domains. Not every type of crisis in a domain can be known when the software is deployed. Due to this, CMS software is designed to be extendable such that new crisis management scenarios can easily be incorporated into the system. However, before incorporating a new scenario into the system, the validity of the following two conditions should be ensured: 1) the new scenario does not violate the invariants of the CMS software 2) the old scenarios respect the invariants of the CMS so that they do not cause problems with the new scenario. For verifying the second condition one has to consider all possible interactions between different crisis scenarios.

Assume that initially the stakeholders wanted to deploy the CMS software to manage only car accidents. To fulfil this deployment, the pattern class *CarCrashScenario* was designed, implemented as an aspect with the same name and shipped with the CMS software to the stakeholders. As the stakeholders gained experience with car accidents, they noticed that some car accidents have a high priority and the crisis resources should be first allocated to these accidents. One such accident is the presidential accident, which always dispatches an ambulance to the crisis scene. Thus, if all the ambulances are allocated by other car crash scenarios, one of the crash scenarios should pre-empt the resources it has allocated.

To manage presidential accidents, a new pattern class called *PresidentialEmergencyScenario* is added to the design of the CMS. The CMS already supports prioritization of the crisis scenarios and pre-emption of resources: by calling the operation *ScenarioBroadcastEvent.firePreEmpt()* a scenario may request other scenarios to release the resources. This feature is used by the newly added scenario; as can be seen from the sequence diagram shown in Figure 3-(b), the pattern class *PresidentialEmergencyScenario* calls the operation *Server. ScenarioBroadCastEvent()* to request pre-emption when the operation *fireStart()* is invoked. The sequence diagram in Figure 16 shows how the pattern class *PresidentialEmergencyScenario* allocates resources. This design does not handle failures during resource allocation because it assumes all other crisis scenarios in the system have correctly pre-emptied the required resources. In an environment where pre-emption is required, the pattern classes of the scenarios are required to implement a "before invoke" operation for the operation *firePreEmpt().* The "before invoke" operation realizes how the pre-emption is realized in the crisis scenario.

However, pre-emption was not a requirement when the CMS is initially deployed and the aspect *CarCrashScenario* does not handle the request to pre-empt resources. As a result, one of calls to allocate a resource in the presidential emergency scenario fails and returns *null.* When the pattern class *PresidentialEmergencyScenario* is implemented as an aspect by following the design presented in Figure 16, it would crash because a reaction on a failed resource allocation is not specified.

In the reminder of this section, we describe how the graph-based semantic interference detection is used to verify the pre-emption requirement of the presidential emergency scenario based on the UML models of CMS (i.e. before the aspect *PresidentialAccidentScenario* is implemented).

4.1 Simulation of the UML models of the CMS

The sequence diagram in Figure 15 shows the class *Server* receiving 4 events from the users. Two of these events request a new crisis scenario to be initialized and the other two require the newly initialized scenarios to allocate the resources. To apply graph-based model checking, we used this sequence diagram, the class diagram of the CMS software (these two diagrams constitute the "theme" main), the UML diagrams of the "theme" Car Crash Scenario and the UML diagrams

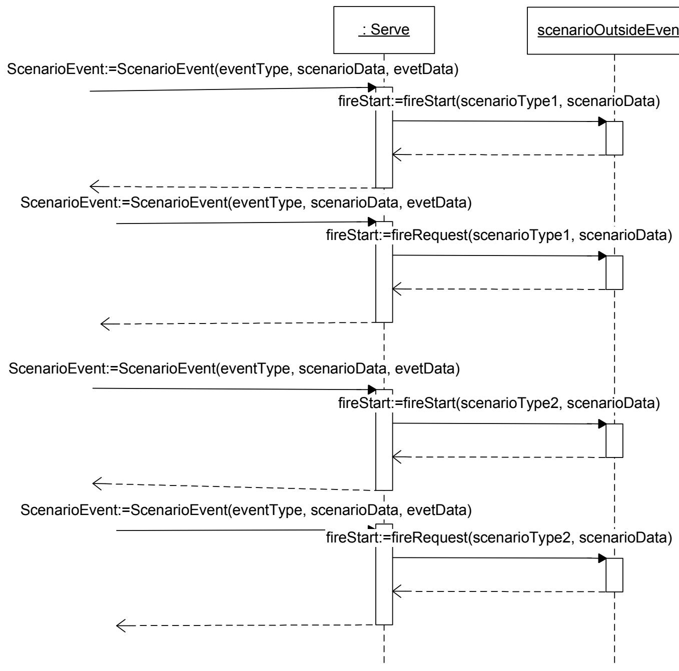
: Serve
scenarioOutsideEvents
ScenarioEvent:=ScenarioEvent(eventType, scenarioData, evetData)
fireStart:=fireStart(scenarioType1, scenarioData)
ScenarioEvent:=ScenarioEvent(eventType, scenarioData, evetData)
fireStart:=fireRequest(scenarioType1, scenarioData)
ScenarioEvent:=ScenarioEvent(eventType, scenarioData, evetData)
fireStart:=fireStart(scenarioType2, scenarioData)
ScenarioEvent:=ScenarioEvent(eventType, scenarioData, evetData)
fireStart:=fireRequest(scenarioType2, scenarioData)

Fig. 15. The class *Server* receiving 4 events from the user

of the "theme" Presidential Emergency Scenario. Some of the sequence diagrams of the "theme"s are not shown in this paper for ease of understanding; however, interested readers can download the full diagrams from [2]. Table 1 presents the details on the number of actions of the sequence diagrams of the CMS.

The generated DCML model from these diagrams contains 490 graphs elements (nodes and edges). In this DCML model, the pattern classes are mapped to the aspect-types *CarCrashScenario* and *PresidentialEmergencyScenario*. Both of these aspects have 3 before pointcuts because there are 3 "before invokes" operations in each theme. The simulation of this model generated a state-space consisting of 7803 states, 8075 transitions; the simulation took 18.54 seconds[1]. Since every possible order of the aspects is a different branch, each invoke of the operations *fireStart* and *fireResourceAllocate* adds two branches to the GTS. In addition to this, the conditional paths also add two branches for each advice invocation. Furthermore, the GTS contains one node for each simulation step of the non-aspect-oriented semantics.

[1] The simulation was exeuted on a laptop with Core 2 Duo 2.4 GHz CPU 4GB Ram running Windows Vista Ultimate 64-bit and JDK 1.6 Update 6.

Table 1. The details of the UML models of the CMS used for the simulation

Theme	#Classes	#Actions in the sequence diagram	#Binding Spec.
Main	14	8	0
Car Crash Scenario _fireStart	1	13	1
Car Crash Scenario _fireResourceAllocate	1	16	1
Presidential Scenario _fireStart	1	17	1
Presidential Scenario _fireResourceAllocate	1	21	1
Presidential Scenario _firePreEmpt	1	4	1

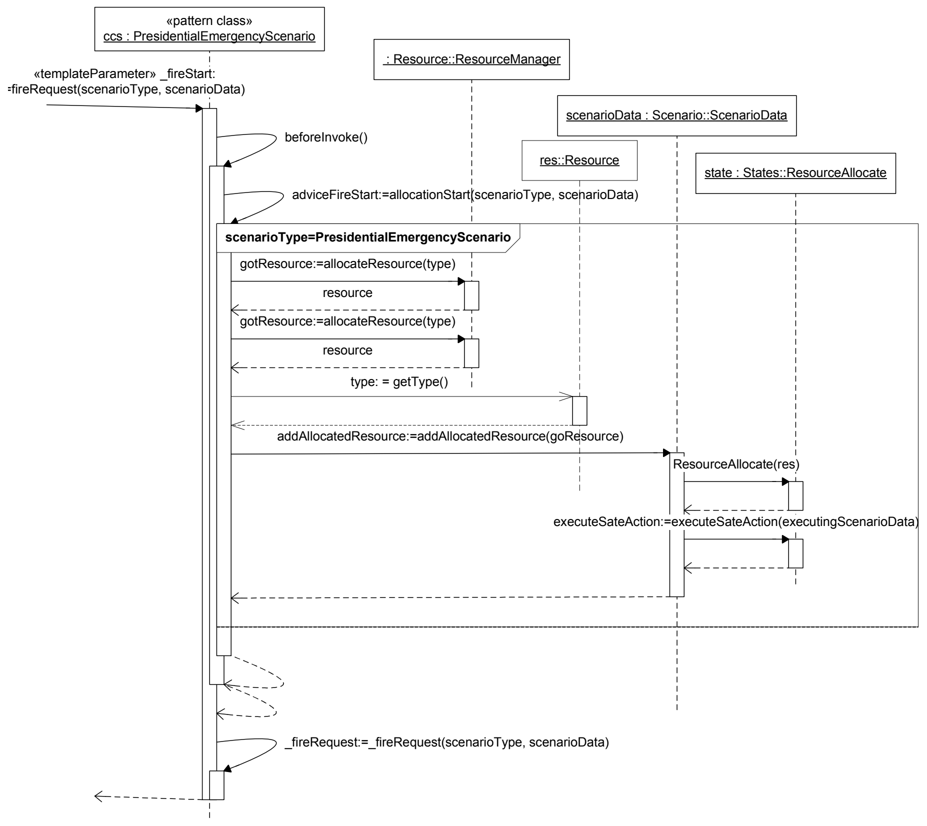
«pattern class»
ccs : PresidentialEmergencyScenario
: Resource::ResourceManager
scenarioData : Scenario::ScenarioData
res::Resource
state : States::ResourceAllocate
«templateParameter» _fireStart:
=fireRequest(scenarioType, scenarioData)
beforeInvoke()
adviceFireStart:=allocationStart(scenarioType, scenarioData)
scenarioType=PresidentialEmergencyScenario
gotResource:=allocateResource(type)
resource
gotResource:=allocateResource(type)
resource
type: = getType()
addAllocatedResource:=addAllocatedResource(goResource)
ResourceAllocate(res)
executeSateAction:=executeSateAction(executingScenarioData)
_fireRequest:=_fireRequest(scenarioType, scenarioData)

Fig. 16. The sequence diagram from theme Presidential Emergency Scenario showing the resource allocation of the pattern class

4.2 Methods for Pruning the State-Space

The number of states generated by the simulation in the worst case is bounded above by $O(n^k \times c^{nk})$, where n is the number of aspects, k is the number of operation with shared join points and c is the maximum number of alternative executions in the aspects. On average, this number is much less then the provided upper bound because GROOVE detects isomorphic states in branches and merges them into one branch. Nevertheless, if there are many aspects on shared join points, the state-space becomes to large. Therefore, we offer the use of the following methods for pruning the state-space:

– **Simulating with reduced number of aspects:** By simulating a subset of the aspects that are important for the verification, the size of the state-space may be reduced greatly. However, the important aspects for an invariant are selected manually and, as a result, an aspect that causes interference may be left-out.
– **Simulating with reduced number of operations with shared join points:** The operations which are intercepted by more than one aspects is likely to cause interference problems. However, not all of these methods need to be simulated to verify an invariant. If one can select a subset of these operations important to the invariant, then one of the dominating factors in the size of the generated state-space would be reduced. This method shares the same disadvantage of the previous method, as manual selection may miss an operation that violates the invariant.
– **Using aspect precedence:** For certain invariants, the execution of order of the aspects may not be important. Thus, the state-space can be reduced by specifying the precedence of the aspects. For example, this is applicable for scenarios of the CMS because scenarios are mission critical and extra attention must be paid in verifying whether they obey the invariants.
– **Identifying mutually exclusive conditions:** As discussed in Section 3.4, the transformation rule ConditionalAdapt just picks one of the conditional paths. When the conditions are mutually exclusive, semantically impossible executions may be generated due to this. The size of the state-space may be reduced by specifying which conditional frame fragments are mutually exclusive in the sequence diagrams. Using the stereotype <<exclusive>>, the designers can specify that these conditional paths as exclusive. When the simulation reaches an exclusive conditional path, it picks one of the execution paths but adds edges describing the picked execution path to the value of the variable declaration the condition is taken upon. Thus, exclusive conditional statements that take conditions upon the same value, are only allowed to pick the path that is different then the marked path. For example, the conditional paths of the aspects *CarCrashScenario* and *PresidentialEmergencyScenario* are exclusive in that, when one of them is executes the actions within the frame fragment the other one returns (skips the actions in the frame fragment). Without mutually exclusive conditional paths, the simulation may generate a branch where actions within the frame fragment

is executed for both aspects (i.e. an execution sequence where the parameter *scenarioType* is equal to *CarCrashScenario* and *PresidentialScenario*. This execution sequence is omitted when the condition on the parameter *scenarioType* is specified as mutually exclusive.

To test the effects of these state space reduction mechanisms on the design of the CMS, we run the simulation without the reduction mechanisms and with the reduction mechanisms. We also added crisis scenarios, whose executions are very similar to the car crash scenario, to see how the simulation scales. In Table 2 the size of the generated state-space and the simulation time of each run is shown. It can be seen that without any reduction method, as the number of scenarios increases, the size of the state-space grows significantly.

When the aspect precedence is used, the size of the state-space reduces by a factor of 9 compared to the size of the state-space without any reduction methods. Because with precedence, only one execution order for aspects are generated. In this case, the number of states is bounded above by $O(c^{nk})$; where c is the maximum number of alternative execution paths in the aspects, n is the number of aspects and k is the number of shared operations at join point.

For four scenarios, when exclusive "ifs" are used with aspect precedence, in total, the state-space contains at 25 branches. This is because in the sequence diagram given in Figure 15, the subsequent invocations to the operations *fireStart()* and *fireResourceAllocate()* used the same value. After the first invocation of the operation *fireStart()*, 5 branches are generated with exclusive "ifs" (in 4 of these branches one of the aspects executed the actions within the frame fragment, and in the fifth one none of the aspects have executed this code). The first invocation of the operation *fireResourceAllocate()* does not cause branches because the parameter *eventType* is the same as the parameter passed the operation *fireStart()*. After this, the second invocation of the operation *fireStart()* causes another 5 branches. This time exclusive "ifs" caused branches because the variable *eventType2* is passed and no path decision is made according to the value of this variable (Figure 15). Similar to the first invocation, the second invocation to the operation *fireResourceAllocate()* does not generate any branches because the variable *eventType2* is passed to this operation. Thus, the number of states in the worst case exclusive "ifs" is bonded above by $O(n^k)$. This worst case happens when all of the k operations are invoked with a distinct value. However, for practical cases the number of states is much less because not every operation invocation uses a distinct value. For example, with CMS for operation invocations are simulated with two distinct values. This reduced the state-space size to be bounded above by $O(n^2)$.

4.3 Verification of Invariants of the Presidential Emergency Scenario

As discussed before, the invariants are verified by checking whether there is a path in the GTS that violates the invariant. Such a check is realized by expressing an execution sequence that violates the invariant as a CTL formula. If the CTL

Table 2. The effects of the state-space reduction mechanisms

# of Scenarios	No Reduction Mechanism			Aspect Precedence			Precedence and Exclusive "ifs"		
	# of States	# of Transition	Simulation time in s.	# of States	# of Transition	Simulation time in s.	# of States	# of Transition	Simulation time in s.
2	7803	8075	18.54	3735	3881	9.29	623	652	2.22
3	99464	98689	368.68	14711	15367	38.12	1258	1324	2.96
4	525940	544100	1059.56	59149	62079	171.07	1824	1928	5.541

evaluation algorithm finds states that conform to the formula (i.e. states where the CTL formula evaluates to true) then the invariant is violated. Using this, we evaluate the following three invariants of the presidential emergency scenario:

– **The scenario presidential emergency should not release its resources when it receives the request for pre-emption.** A scenario releases a resource by calling the operation *ResourceManager.deallocateResource()*. Thus, if the operation *deallocatedResource()* is called from the aspect-type *PresidentialEmergencyScenario* within the advice for the operation *firePreEmpt*, then the invariant is violated. This execution sequence can be expressed with the following CTL formula

$$EF(executeMethod(\text{"}adviceFirePreEmpt\text{"}, \text{"}PresidentialEmergencyScenario\text{"}) \wedge (EF(executeMethod(\text{"}deallocateResource\text{"}, \text{"}ResourceManager\text{"}) \wedge (EF(returnframe(\text{"}adviceFirePreEmpt\text{"}, \text{"}PresidentialEmergencyScenario\text{"}))))))$$

This formula looks for a path where before the advice code at the aspect *PresidentialEmergencyScenario* for the operation *firePreEmpt* returns, the operation the *deallocateResource()* executes. Here, the labeled *returnframe("adviceFirePreEmpt", "PresidentialEmergencyScenario")* designates the exit point of an operation. The verification did not find any states that satisfy this formula. So, we can conclude that the invariant is not violated. This invariant is not violated because in the "theme" presidential emergency scenario the designers specified a sequence diagram, which shows the "before invoke" for the operation *firePreEmpt()* only returns.

– **The resource allocation for the scenario presidential emergency should always complete.** This invariant is violated if the advice for the method *fireResourceAllocate()* of the aspect-type *PresidentialEmergencyScenario* does not complete successfully. The transitions labeled *returnframe (operation name, object-type)* only occur when the operation successfully returns. Thus, with the following CTL formula we can search an execution sequence where the advice starts executing but does not return.

$$EF(executeMethod(\text{"}allocationStart\text{"}, \text{"}PresidentialEmergencyScenario\text{"}) \wedge$$

$$!(EF(returnframe("allocationStart", "PresidentialEmergencyScenario")))$$

Note that here *allocationStart()* is an operation called by the advice for the operation *fireRequest* as shown in Figure 16. The verification was able find states that satisfied this formula; thus, the invariant is violated. The advice can fail when the resource manager runs out of resources. When there is not any resource to allocate the operation *allocateResource()* returns *null*. The sequence diagram shown in Figure 16 does not specify any frame fragments that is executed when resource allocation fails (i.e. when *null* is returned). Thus, the execution of the advice did not complete successfully because it thrown a null pointer exception. We can confirm this because after receiving the resources the advice calls *ScenarioData* to log the resources. With a *null* resource, the call to *Resource.getType()* fails. We used the following CTL formula to confirm the call to the operation *getType* fails:

$$\begin{aligned}&EF(executeMethod("allocationStart", "PresidentialEmergencyScenario") \wedge\\ &\quad (EF(executeMethod("getType", "Resource") \wedge\\ &\quad !(EF(returnframe("getType", "Resource"))))))\end{aligned}$$

– **The scenario car crash should deallocate its resources if the resource is not already dispatched.** In any of the paths if the aspect-type *CarCrashScenario* does not execute after the operation *firePreEmpt()*, then this invariant is violated. We can express this path as follows:

$$\begin{aligned}&EF(executeMethod("firePreEmpt", "ScenarioInternalEvent") \wedge\\ &\quad (EF(executes("CarCrashScenario") \wedge\\ &\quad (EF(returnframe("firePreEmpt", "ScenarioInternalEvent"))))))\end{aligned}$$

The verification found states that satisfy this formula which means that pre-emption is not handled by the aspect-type *CarCrashScenario*. The UML diagrams for the "theme" *CarCrashScenario* does not specify a sequence diagram with a before or after invoke to the operation *firePreEmpt*. As a result, there is not a pointcut for this operation in the aspect-type *CarCrashScenario*.

5 Related work

In the following, we discuss the related work from 4 different perspectives:

Representing UML models In the literature, graph-based approaches are used to formally define the semantics of UML class- and object diagrams [26], to detect inconsistencies in UML diagrams caused by evolution [30], to formalize refactorings [31], to recover design information [35] and to correctly evolve design patterns [41]. These approaches provide a graph-based model for object-oriented systems focusing on the static structure; in contrast, DCML is tailored more to model the dynamic structure of the software. In our previous studies [10], we used graph transformation rules to correctly evolve UML models by following the constraints of software structures like design patterns (e.g. we defined transformation rules to add a strategy using the strategy pattern). In contrast to DCML, the graph based model used in this study is static, captures the class diagram and one sequence diagram. It lacks also the model elements that are used for execution semantics.

Execution Semantics for UML models Although the meta-model of UML is documented and the modeling language is widely known, the lack of formal semantics makes it hard to reason about the models. In the literature, formal semantics for different types of UML diagrams are proposed. Whittle [39] provides a formal semantics to use-case charts that enables the specification of use-case scenarios. Use cases capture the software system's behavior from the stakeholders' view, and use-case charts model them as three-level diagrams. The first level is an activity diagram where the nodes are the use cases; the second level is also an activity diagram where the nodes are scenarios of a use case in the previous level; and the third level is constituted by interaction diagrams of the scenarios of the second level. Because the use-case charts formally specify the scenarios, they can be simulated. Therefore, a hierarchical state machine synthesis algorithm is proposed [40] and the tool UCSIM that executes this algorithm and simulates the generated state machines has been developed [21].

Graph transformations have been used to specify formal execution semantics to UML state-charts [27] [29]. For example, Kung et al. [27] generate the graph grammar that models the execution semantics for a given state-chart. These semantics, however, work only on providing verification/visualization for a single state-chart.

Dynamic meta modeling is also proposed as a way to add operational semantics to the UML diagrams [15]. In this approach, the meta model of the UML class diagram is extended with a dynamic meta model that uses the collaboration diagram notations. The state-chart diagrams specify the behavior of the system; for example, in order to trigger a transition, a method has to be called which in turn can trigger another event in the state chart of the called method. Using graph transformations the operational semantics such as state transitions or method call triggers are modeled. Using a state space generator such as GROOVE and these graph transformations [16], it is possible to simulate the behavior of the software system and generate the state space of the system. Then, the requirements of the system can be verified in the generated state space.

The main difference between the approaches presented in this section and our semantics is that we provide semantics that are close to actual aspect-oriented software execution. Moreover, the semantics we provide are generic can be applied to any sequence diagram.

Object-oriented verification Programming languages generally have a well-defined syntax, but their execution semantics are often informally specified. To formalize the execution semantics of Object-oriented programs, Kastenberg et al. model execution semantics of the TAAL language (a simplified version of Java) as graph transformations [22]. Here, the idea is that a program in the TAAL language can be compiled into a graph model and simulated using graph transformation rules. By using graph-based model checking, the properties of the execution can be verified. In our approach, we apply graph-based model checking to UML models by modeling the execution semantics of UML, in particular of sequence diagrams (which define valid/desired sequences of actions) in combination with class diagrams. The semantics defined by Kastenberg are specific for TAAL and cannot suitably be adapted to represent UML-based models, without major effort. As compared to this, we have defined an (automated) mapping from the UML metamodel to DCML. In addition, their approach does not support aspect-based functionality (such as explicit mappings for pointcuts, join points, etc.).

Visser et al. propose to apply model checking to main stream programming languages, such as Java [38]. They propose a system (Java Path Finder) to model check programs expressed in Java Bytecode. Their approach is applied to programs at the implementation level; this is an explicit design choice. As compared to this, we want to detect interference at the UML design level, while also supporting the use of aspects at this level.

Aspect-oriented verification A few approaches exist that aim to prove the correctness (usually with regard to specific properties) of programs or models in the presence of aspects. For example, Katz et al. propose an approach that checks whether a given *program*, which may include crosscutting definitions (such as pointcut-advice constructs), conforms to one or more scenarios [24]. In this approach, the system and aspects must be specified as a set of scenarios in a formal language. The verification is carried out by detecting the join points of the aspectual scenarios and finding out if there are undesired interferences at these points. Our approach, on the other hand, does not require the user to specify anything about the program beyond UML models; interference is detected based on the operation semantics of these models.

An approach related to the above facilitates the automated derivation of proof obligations from requirements models that are specified using an aspect-oriented requirements engineering approach [25]. For example, the approach generates temporal logic formulas that have to be satisfied (proven) in later design stages. This allows to check the consistency between early and later design stages. As compared to this, the approach presented in this paper verifies the consistency of several models at the same modeling level (i.e. at the concrete design stage). As such, the approaches could be used to complement each other.

At the level of aspect-oriented requirements engineering, [8] proposes an approach that uses *semantic annotations* to make aspect composition specifications less fragile. The work is related to ours, as these specifications later drive the generation of models; it is however not aimed at (model-)checking the correctness of the resulting system with regard to the intended semantics of the composed system. Rather it supports the user to actually compose the intended elements in the first place (cf. the fragile pointcut problem [28]). In this paper we did not focus on addressing problems related to fragile pointcuts; we do however discuss this issue (at the program level) in other work [34, 19].

The authors of [33, 32] aim at detecting the possible semantic interferences that can manifest in models expressed using the Aspect-UML language. To this aim, the static language elements of Aspect-UML are mapped to the specification language of Alloy. The operational semantics of Aspect-UML models, however, have to be expressed manually using the dynamic state-based specification language of Alloy. There are at least three differences between our approach and the one proposed in [33]. Firstly, in our approach, we derive the formal representation of models based on the static and dynamic semantics of the UML, and on a graph based generic pointcut model. As such, our approach is less AOM specific. Secondly, we derive the operational semantics of models from the UML models directly; the modeler does not need to define the dynamic behavior of models in the specific language of the verifier. Thirdly, Alloy is designed intentionally as a limited model-checker that mainly adopts the small scope hypothesis; that is, the errors are searched only within a limited scope. The GROOVE model checker, however, incorporates various verification tools including a possibility for a full state-space exploration.

In [4], Aksit et al. introduce an approach for the detection of interference between aspects that is also based on graph transformations. There are two major differences with the work presented in this paper: first, the paper focuses on modeling the operational semantics of aspect-specific behavior (i.e., advice code) within the Composition Filters approach. Thus, the approach works at the program level and focuses on (only) the aspect-related part of the program. Second, the paper focuses on detecting interference at shared join points, i.e., it detects situations where the behavior of the program differs based on the order in which advices at shared join points are executed. As compared to this, the work presented in our paper does not focus on detecting interference at (only) shared join points, and takes the semantics of the entire (UML-based) model into account.

6 Discussions and Conclusions

In this article, we have analyzed the example CMS case, identified two crisis scenarios as aspects, define a model using Theme/UML, and verified its correctness at the modeling level with respect to a possible semantic interference among the scenarios. We will now evaluate our approach from the following three perspectives:

Expressivity: The expressiveness of the tool is determined by the base modeling language UML, and the adopted join point model as described in section 3.3. In the current version, our tool supports UML class and sequence diagrams. A large set of library of rules are defined for the static and operational semantics of UML. The join point model currently supports before and after pointcut specifications. Since our model is based on a general purpose graph-verification system (GROOVE), further extensions and tailoring should be possible within the limits of the selected approach. Our system is also supported by UML to graph translators, pruning techniques, model checkers. The sophistication of the graph model and the operational semantics, the simulation and the complexity of the model-checking process are hidden from the user. The user supplies UML-based AOM models to the system, and has to use CTL formulas to specify the invariants. The user is warned if the state space becomes too large, so that the available pruning methods can be investigated. Our system was able to detect the UML-level semantic errors among the aspects of the CMS model. Of course, the capability of our approach is by definition limited to the expressiveness of the selected UML models and the join point model.

Scalability: In the CMS, scenarios are represented as aspects. The space and time complexity of the detection algorithm depends on the simultaneously active aspects that interfere with each other. In most aspect-oriented applications, the number of potentially conflicting aspects are expected to be low. However, depending on its context of usage, the CMS system should be able to handle, say 10-20 scenarios simultaneously. Within the context of the example case, without taking any measures, the tool is capable of handling approximately four scenarios. Since this is rather limited, we have decided to use the available pruning methods. We first applied a precedence order among the scenarios and investigated if such an ordering caused any artificial restriction. This did not cause any problem because, the invariants that are used in the verification process as shown in section 4.3 are independent of the aspect execution order. As shown in Table 2, this pruning method has reduced the generated states more than 9 times. To further reduce the state space, we have identified the critical operations which cause branching in the state space. As shown in Figure 14, such as an information is easily obtainable from the tool. We have then tagged the branches that should be mutually exclusive. As a result, the state space is reduced more than 300 times. Concluding, from this practical experience, we observe that our approach is applicable to verifying models without pruning up to a limited set of interfering aspects. Nevertheless, as in the case of the CMS, the pruning process may reduce the complexity of the verification dramatically. From the perspective of the tool, a model which consists of a high-level of interfering aspects and that cannot be pruned, is considered either ill-designed and therefore must be re-factored, or is out of the capability of the tool. We assume this will be an exception in practice.

Applicability to AOM tools: Our tool is only applicable to UML-based AOM models, which can be translated to our internal representation. First of all, this requires mapping UML-specific parts of the AOM to our meta-model. Secondly, the aspect and pointcut designators of the AOM must be mapped to our pointcut model as specified in Figure 8.

Based on these evaluations, we can conclude that the tool proves its applicability to the CMS example case. We think that the tool is capable of handling a large category of aspect-interface problems that can be experienced in the current UML-based AOM approaches. As discussed in the article, there may be certain complex cases which the tool cannot scale. These, however, are considered as rather exceptional cases rather than the routine.

References

1. ArgoUML [online] http://argouml.tigris.org.
2. Gace: Graph-based adaptation, configuration and evolution modeling [online] http://trese.cs.utwente.nl/willevolve/.
3. M. Akşit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object-interactions using composition-filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Processing*, pages 152–184. Springer-Verlag Lecture Notes in Computer Science, 1993.
4. M. Aksit, A. Rensink, and T. Staijen. A Graph-Transformation-Based Simulation Approach for Analysing Aspect Interference on Shared Join Points. In *AOSD '09: Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development, Charlottesville, Virginia, USA*, pages 39–50, New York, 2009. ACM.
5. K. Altisen, F. Maraninchi, and D. Stauch. Aspect-oriented programming for reactive systems: Larissa, a proposal in the synchronous framework. *Science of Computer Programming*, 63(3):297–320, 2006.
6. J. Araújo, J. Whittle, and D.-K. Kim. Modeling and composing scenario-based requirements with aspects. In *Proc. 12th Int'l Requirements Engineering Conference*, pages 53–62. IEEE, Sept. 2004.
7. E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the 26th International Conference on Software Engineering*, pages 158–167. IEEE Computer Society Washington, DC, USA, 2004.
8. R. Chitchyan, A. Rashid, P. Rayson, and R. Waters. Semantics-based composition for aspect-oriented requirements engineering. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 36–48. ACM New York, NY, USA, 2007.
9. R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tekinerdogan, and A. J. Siobhán Clarke and. Survey of aspect-oriented analysis and design approaches. Technical Report AOSD-Europe-ULANC-9, AOSD-Europe, May 2005.
10. S. Ciraci, P. van den Broek, and M. Aksit. Framework for computer-aided evolution of object-oriented designs. *COMPSAC*, pages 757–764, 2008.
11. S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *Proc. 23rd Int'l Conf. Software Engineering (ICSE)*, pages 5–14, May 2001.
12. S. Clarke and R. J. Walker. Generic aspect-oriented design with Theme/UML. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 425–458. Addison-Wesley, Boston, 2005.

13. P. E. A. Durr. *Resource-based Verification for Robust Composition of Aspects*. PhD thesis, University of Twente, Enschede, June 2008.
14. H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69, 1979.
15. G. Engels, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to operational semantics of behavioral diagrams in uml. 1999.
16. G. Engels, C. Soltenborn, and H. Wehrheim. Analysis of uml activities using dynamic meta modeling. In *FMOODS'07*, LNCS, pages 76–90. Springer-Verlag, 2007.
17. R. France, I. Ray, G. Georg, and S. Ghosh. Aspect-oriented approach to early design modelling. *IEE Proceedings Software*, 151(4):173– 185, Aug. 2004.
18. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundam. Inf.*, 26(3-4):287–313, 1996.
19. W. K. Havinga, I. Nagy, L. M. J. Bergmans, and M. A. sit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In O. de Moor, editor, *Proceedings of International Conference on Aspect Oriented Software Development, AOSD 2007, Vancouver, Canada*, ACM International Conference Proceedings Series, pages 85–95, New York, March 2007. ACM Press.
20. R. Helm, I. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. *ACM Sigplan Notices*, 25(10):169–180, 1990.
21. P. K. Jayaraman and J. Whittle. Ucsim: A tool for simulating use case scenarios. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 43–44, Washington, DC, USA, 2007. IEEE Computer Society.
22. H. Kastenberg, A. G. Kleppe, and A. Rensink. Defining oo execution semantics using graph transformations. In *8th IFIP*, volume 4037 of *LNCS*, pages 186–201, 2006.
23. H. Kastenberg and A. Rensink. Model checking dynamic states in groove. In *SPIN'06*, volume 3925, pages 299–305, Berlin, 2006. Springer-Verlag.
24. E. Katz and S. Katz. Verifying scenario-based aspect specifications. *Lecture notes in computer science*, 3582:432, 2005.
25. S. Katz and A. Rashid. From aspectual requirements to proof obligations for aspect-oriented systems. In *12th IEEE International Requirements Engineering Conference, 2004. Proceedings*, pages 48–57, 2004.
26. A. Kleppe and A. Rensink. On a graph-based semantics for uml class and object diagrams. In *Proceedings of the 7th International Workshop on Graph Transformation and Visual Modeling Techniques*, volume 10 of *Electronic Communications of the EASST*, page 16, 2008.
27. J. Kong, K. Zhang, J. Dong, and D. Xu. Specifying behavioral semantics of uml diagrams through graph transformations. *J. Syst. Softw.*, 82(2):292–306, 2009.
28. C. Koppen and M. Störzer. PCDiff: Attacking the fragile pointcut problem. In K. Gybels, S. Hanenberg, S. Herrmann, and J. Wloka, editors, *European Interactive Workshop on Aspects in Software (EIWAS)*, Sept. 2004.
29. S. Kuske. A formal semantics of uml state machines based on structured graph transformation. In *UML'01*, pages 241–256, London, UK, 2001. Springer-Verlag.
30. T. Mens, R. V. D. Straeten, and M. DíHondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In *Model Driven Eng. Lang. and Sys.*, volume 4199/2006, pages 200–214, 2006.
31. T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, 2007.

32. F. Mostefaoui and J. Vachon. Design-level Detection of Interactions in Aspect-UML models using Alloy. *Journal of Object Technology*, 6:137–165, 2007.
33. F. Mostefaoui and J. Vachon. Verification of Aspect-UML models using Alloy. In *Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 41–48. ACM New York, NY, USA, 2007.
34. I. Nagy, L. Bergmans, W. Havinga, and M. Aksit. Utilizing design information in aspect-oriented programming. In A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, editors, *Proceedings of International Conference NetObjectDays, NODe2005*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany, Sep 2005. Gesellschaft für Informatik (GI).
35. C. Rich and L. Wills. Recognizing a program's design: a graph-parsing approach. *Software, IEEE*, 7(1):82–89, Jan 1990.
36. A. Sampaio, R. Chitchyan, A. Rashid, and P. Rayson. EA-Miner: a tool for automating aspect-oriented requirements identification. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 352–355. ACM New York, NY, USA, 2005.
37. A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, and G. Kappel. A survey on aspect-oriented modeling approaches. *Relatorio tecnico, Vienna University of Technology*, 2007.
38. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.
39. J. Whittle. Precise specification of use case scenarios. In *FASE'07*, volume 44 of *LNCS*, pages 170–184. Springer-Verlag, 2007.
40. J. Whittle and P. K. Jayaraman. Generating hierarchical state machines from use case charts. In *RE '06*, pages 16–25, Washington, DC, USA, 2006. IEEE Computer Society.
41. C. Zhao and et al. Pattern-based design evolution using graph transformation. *J. Vis. Lang. Comput.*, 18(4):378–398, 2007.

Appendix – DCML elements

In this appendix, all elements of the DCML and examples for them are discussed in detail. For convenience, we have copied the figures used in this discussion to the appendix which are already used throughout the paper. Figure 17 repeats the DCML meta model.

Structural Part of DCML

The structure part covers the elements of the meta-model for modeling the classes, the interfaces and the relations between these. This part is generated from the class diagram. Because classes and interfaces are types at runtime, they are represented with nodes labeled *ObjectType* (object-type nodes). If the object-type node is representing an interface, then the attribute interface is set to true. The equivalent of the generalization relation is the edge labeled supertype. Figure 18-(a) shows a portion of the class diagram from the CMS with three classes, namely *State*, *ResourceAllocation* and *ScenarioData*. Figure 18-(b) shows the DCML representation of this class diagram; here, the three object-type

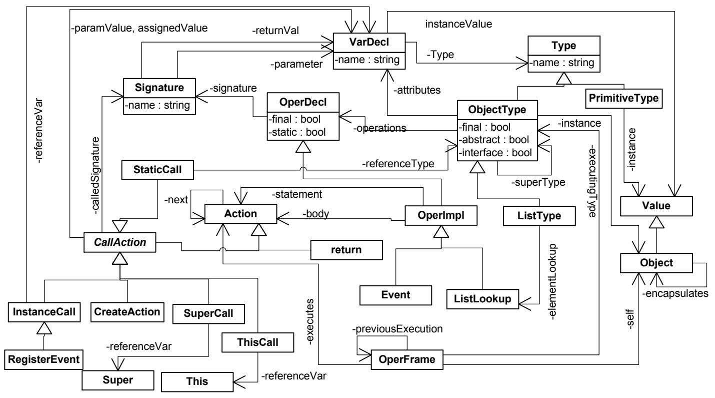

Fig. 17. The DCML meta-model

nodes represent the classes in the class diagram. For example, the object-type node with the attribute name *ResourceAllocation* (i.e. the name of object-type node) is the class with the same name represented in DCML. The class *ResourceAllocate* generalizes the class *State* in the class diagram. This is shown in DCML with the edge labeled *SuperType* connecting the object-type nodes representing these classes.

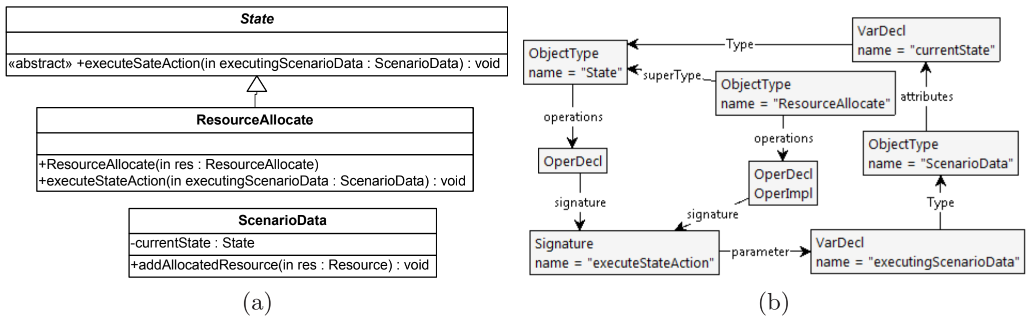
State
«abstract» +executeSateAction(in executingScenarioData : ScenarioData) : void
ResourceAllocate
+ResourceAllocate(in res : ResourceAllocate)
+executeStateAction(in executingScenarioData : ScenarioData) : void
ScenarioData
-currentState : State
+addAllocatedResource(in res : Resource) : void
(a)
ObjectType
name = "State"
Type
VarDecl
name = "currentState"
superType
ObjectType
name = "ResourceAllocate"
attributes
operations
OperDecl
OperDecl
OperImpl
ObjectType
name = "ScenarioData"
signature
Signature
name = "executeStateAction"
parameter
VarDecl
name = "executingScenarioData"
(b)

Fig. 18. a) An example UML class diagram. b) The DCML model of the class diagram shown in (a)

The nodes labeled *VarDecl* represent the variable declarations (variable declaration node); the type of the variable is modeled by the edge labeled *Type*, connecting the variable declaration node to a type node. An object-type node connected to a variable declaration node with an edge labeled *attributes* represents that the variable is an attribute of the object-type. For example, the class *ScenarioData* has the attribute *currentState* and the type of this attribute is the class *State* as shown in Figure 18. In the DCML equivalent of this class diagram in Figure 18, this is also shown: the object-type node named *ScenarioData* and the variable declaration node named *currentState* are connected by the edge labeled *attributes*.

DCML separates the operation signatures from the operation declarations. The operation declaration nodes (nodes labeled *OperDecl*) are used for representing the abstract operations and operations without implementations. The object-type node connected to an operation declaration node with an edge labeled *operations* represents the object-type which declares the operation. The implemented operations, on the other hand, are represented by nodes labeled both *OperImpl* and *OperDecl* (operation implementation nodes). In Figure 18-(a), the class *State* has an abstract operation; thus, the object-type node *State* is connected to an operation declaration node in the DCML model of this class diagram (Figure 18-(b)).

Each unique signature in the class diagram is represented by signature nodes (nodes labeled *Signature*). The parameters of a signature are represented by variable declaration nodes connected to the signature node with an edge labeled *parameter* and the return type of the signature is represented by connecting the signature node to a type node. In Figure 18-(a), there are two operations with the same signature named *executeStateAction* that take one parameter of type *ScenarioData* and do not return a value. In the DCML model of this class diagram, this signature is represented by the signature node named *executeStateAction*. Note that the operation declaration node of the object-type *State* and the operation implementation node of the object-type *ResourceAllocate* are both connected to this signature node by an edge labeled *signature*. This shows that in the object-type *State* an operation with a signature named *executeStateAction* is declared and in the sub-type *ResourceAllocate* this operation is implemented. In this manner, operation overriding is modeled by connecting the operation implementation node of a sub-type to a signature node to which an operation implementation node of the super-type is connected.

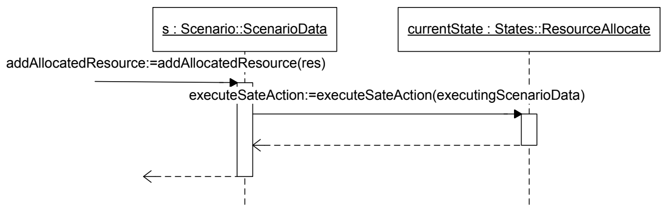
s : Scenario::ScenarioData
currentState : States::ResourceAllocate
addAllocatedResource:=addAllocatedResource(res)
executeSateAction:=executeSateAction(executingScenarioData)

Fig. 19. A sequence diagram showing the actions executed by the operation *ScenarioData.addAllocatedResource()*

Dynamic Part of DCML

The dynamic part, which is generated from the sequence diagrams, covers the elements for modeling the objects, the values and the life-lines operations. A life-line in a sequence diagram shows the actions the object executes when it receives a call. In the DCML meta-model (Figure 17), the specializations of the abstract element *Action* represents the actions of sequence diagrams. For example, the nodes labeled *CallAction* represent call actions and the nodes labeled *return* represent return actions. An action node can be connected to another action node by an edge labeled *next*; in this way, the order between the actions of a life-line is represented in DCML. The first action of a life-line is connected to an operation implementation node by an edge labeled *body* in DCML to show that these actions are executed when this operation received a call. The sequence diagram presented in Figure 19 shows the life-line of the operation *addAllocatedResource()*. The first action executed in this life-line is a call action. This action is followed by a return action where the operation *addAllocatedResouce()* returns. Figure 20 shows the DCML model generated from this sequence diagram (and the class diagram in Figure 18). In this figure, the emphasized node represents the call action belonging to the life-line of the operation *addAllocatedResource*. Because in the sequence diagram this call action is the first action executed in the life-line of the operation *addAllocatedResource()*, the emphasized node is connected to the operation implementation node representing the operation *addAllocatedResource()* by an edge labeled *body*. This call action node is connected to the signature node named *executeStateAction* by an edge labeled *calledSignature* to show that the call action is to the signature *executeStateAction*. Following the outgoing edge labeled *next* from the call action node, it can be seen that the call action is succeeded by a return action.

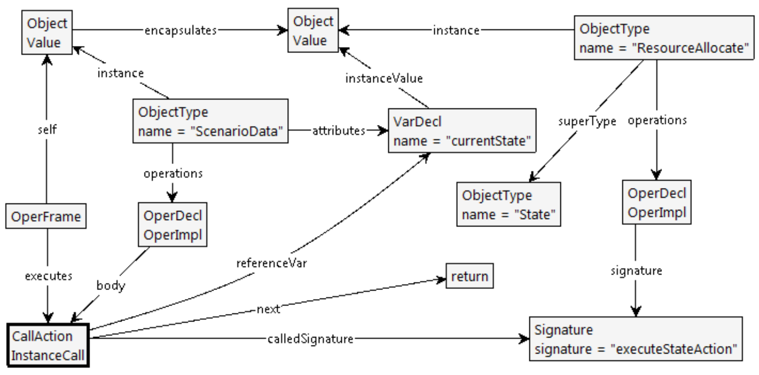

Fig. 20. A snapshot from the simulation of the sequence diagram shown in Figure 19

In DCML, call actions have *5* specializations representing different kinds calls: the calls to instances (*InstanceCall*), create actions (*CreateOper*), super operation calls (*SuperCall*), self calls (*ThisCall*) and static operation calls (*StaticCall*). The call action to the operation *ResourceAllocate.executeStateAction* in the sequence diagram shown in Figure 19 is an instance call because this call is received by an instance labeled *currentState* of the class *ResourceAllocate*. Because this call action is an instance call, the emphasized node in Figure 20, which represents it, is also labeled *InstanceCall*.

The classifier names are represented as variables which hold the objects in DCML because DCML only supports communication between objects through encapsulation. So, the classifier *currentState* in the sequence diagram of Figure 19 is represented as a variable declaration node with the same name in the DCML model of this sequence diagram as shown in Figure 20. The type of this variable is set the object-type named *State* because the class *ScenarioData* has an attribute named *currentState* whose type is the class *State* as shown in Figure 18. If the class *ScenarioData* did not contain such an attribute, then the type of the variable *currentState* would be set to *ResourceAllocate*. Note that the emphasized call action node is connected to this variable declaration node by an edge labeled *referenceVar* to show that the call references the value of this variable.

The values of the variables are represented by connecting the variable declaration nodes to value nodes (nodes labeled *Value* with edges labeled *instanceValue*). Following the edge labeled *instanceValue* from the variable declaration node named *currentState* in Figure 20, it can be seen that the variable is holding an object. This object is an instance of the class *ResourceAllocate*; this is represented by the edge labeled *instanceValue* connecting the object-type node named *ResourceAllocate*. The object node representing an instance of the class *ScenarioData* is connected to the object node representing an instance of the class *ResourceAllocate* by an edge labeled *encapsulates*. This means that in the scope of this instance of the class *ScenarioData*, the variable *currentState* holds an instance of the class *ResourceAllocate*. A DCML model can be generated from more than one sequence diagram and, thus, a variable can have more then one instance value. During simulation, the values of the variables at the executing frame are resolved by the encapsulated edges.

The frame of an executing operation is represented by nodes labeled *OperFrame* in DCML. These nodes are used to identify, during simulation, the object that is currently executing, the scope of the executing object, the type that contains the called operation and the statement that is being executed. The self of an operation frame is represented in DCML by connecting the operation frame node to an object node by an edge labeled *self*. In figure 20, for example, the self of the operation frame is an instance of the class *ScenarioData*. The action that is currently executing is represented by the edge labeled *executes*; for the DCML model in Figure 20 the currently executing action is an instance call. When UML diagrams are converted to DCML models, the conversion algorithm automatically adds the operation frame node which marks the first action of the sequence diagram as the action that is being executed. Thus, the simulation starts executing from that action.