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}@ewi.utwente.nl,
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 the resulting models than traditional techniques. 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.

We propose a technique and tool support for fully automatic detection of conflicts between aspects at the model level; more specifically, our approach works on models defined in UML 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 simulation 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" (CMS) 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 five simulated aspects, the state space contains 9991 different states and 99 different execution paths. Nevertheless, using appropriate pruning methods, the state space only grows polynomially with the number of aspects. In practical cases, the order of the polynomial is very small, e.g., 2 in the case of the simulated CMS; therefore, the automatic analysis scales.

1 Introduction

The goal of Aspect-Oriented Modeling (AOM) [37, 8] 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. Consequently, AOM bears the potential to increase the global understandability of a model, as smaller submodels — the partial views — can be inspected separately.

One downside of the flexible composition mechanisms in AOM is that local understandability is reduced. 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[1] in the composed system by manual inspection. This downside reduces the benefit of an improved modular structure facilitated by AOM, which is a common theme among comments from industry, e.g., in [13]. This disadvantage is especially important when considering that each separate view of the model may evolve in isolation, and each may be maintained by different engineers. Through the separation it becomes harder to ensure that the composed system works together as intended.

For these reasons, we believe that it is important to support the AOM development process by means of tools that detect conflicts in situations when aspects semantically interact with each other. That is, it must be possible to detect *semantic interference among aspects*. Several techniques exist for automatically detecting conflicts between aspects at the implementation level [3,24], but detecting conflicts early at the model level has a number of advantages:

– Models are more abstract than code. Therefore, fixing errors in the design prospectively is cheaper than 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.

Conflict-detection approaches require an abstract semantic model of the overall system. Only a few approaches have been proposed in the past to provide such an abstract model in the design phase and for systems containing aspects. In their case, the abstract model has to be defined separately, i.e., in addition to the design model, for each new system [33, 32] in a specific, limited AOM approach.

To achieve the itemized benefits of model-level conflict detection in an AOM approach without additional efforts for the designer, we propose a technique for automatic, tool-supported detection of semantic interference among aspects at the model level that is independent of the AOM approach used in the design phase. This paper provides tool support that is applicable to any AOM approach extending the UML, which is the majority. Different AOM approaches differ in the way how the UML is extended and in the crosscutting concerns they can successfully modularize.

[1] In the context of this paper, conflicts are understood as undesired execution sequences. This occurs either because methods are wrongly executed or not executed, or because the order of method executions is wrong.

To achieve such tool support, we specialize the graph-based model checker GROOVE [23] to simulate UML-based aspect-oriented models and verify that the execution orders conform with the desired behavior of the software system. The simulation is a foundation of the verification as it generates all the execution sequences supported by the input UML-based aspect-oriented models.

To facilitate the simulation, an operational semantics of the UML is modeled as graph transformations; triggering the relevant transformations allows us to simulate the runtime behavior of the model. To account for aspect-oriented extensions of the UML, we furthermore include graph transformations that allow the implicit inclusion of advice at join points. This mechanism makes our approach independent of the actual pointcut and pointcut-advice bindings whose expressiveness is determined by the used AOM approach.

The output of the simulation is a state space, which is a tree with some merged branches, explicitly showing which software artifacts have executed in which execution sequence. In terms of temporal logic formulas, the user can define requirements for the order of method executions that must always be satisfied in the execution, i.e., invariants of the system. The model checker, which we are using, evaluates these formulas for each execution sequence in the state space in order to identify execution sequences that violate the constraint. That is, our model checker allows us not only to detect that conflicts are possible, but it also shows under which conditions the conflicts take effect.

Conflict detection is mission critical in large software systems like a “Crisis Management System” (CMS) which cannot afford to expose unanticipated behavior. Thus, in this paper, we will demonstrate how to apply our verification approach to the aspect-oriented modeling of the CMS case study. Therefore, we had to concretize the model provided in the common case study; specifically, we have modeled the concerns that relate to the crisis-managing scenarios as aspects.

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 9991 states, 10234 transitions and 99 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 polynomially with the number of aspects. In practical cases, the order of the polynomial is even very small, for example in the simulation of the CMS, the number of states is of the order $O(n^2)$. 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 core runtime behavior of the models. It requires to resolve pointcuts and advice according to the semantics of the actually used AOM approach, and is, thus, applicable to all such approaches.

2. Applying this operational semantics with a model checker, we can automatically detect violations of invariants at the modeling level. In an AOM design, this can be used to detect semantic interferences among aspects. 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 representation.
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. We outline our approach at a very high level in Section 2. In Section 3, we discuss an AOM model for the "Crisis Management System", including an overview of AOM in general and Theme/UML in particular, which we use to show AOM models throughout this paper. Our approach is discussed in detail in Section 4; we start by presenting a motivating example in Section 4.1, followed by a presentation of the graph model used in the simulation and its relation to UML diagrams in Section 4.2; in Section 4.3, we present the means by which our model supports aspect-oriented models. In Sections 4.4 and 4.5, we discuss the operational semantics 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 5. In Section 6, we present related work before we reflect on our approach and conclude in Section 7.

2 An Approach for Graph-Based Model Checking of AOM for Aspect Interference Detection

In this section, we explain how our approach works from the perspective of an application designer, e.g. the engineers designing the Crisis Management System that is the subject of this special issue.

Figure 1 shows the activities involved in using our tooling. The first step relevant to our approach (marked '1' in the diagram) is that the designer creates UML-based models of the system. In this paper, we focus especially on class and sequence diagrams, which model the structure and (partially) the order in which interactions with and within the system should occur. It is important to verify these models in particular, because, as we will show, interference may already occur at this level. Especially in the presence of aspects, this may not be straightforward to "detect" manually. Of course, additional models may also be created in (or before) the design phase, but we do not focus on those in this paper. As shown in figure 1, the class and sequence diagrams may use AOM-specific extensions; in this paper, we have used Theme/UML [12], which

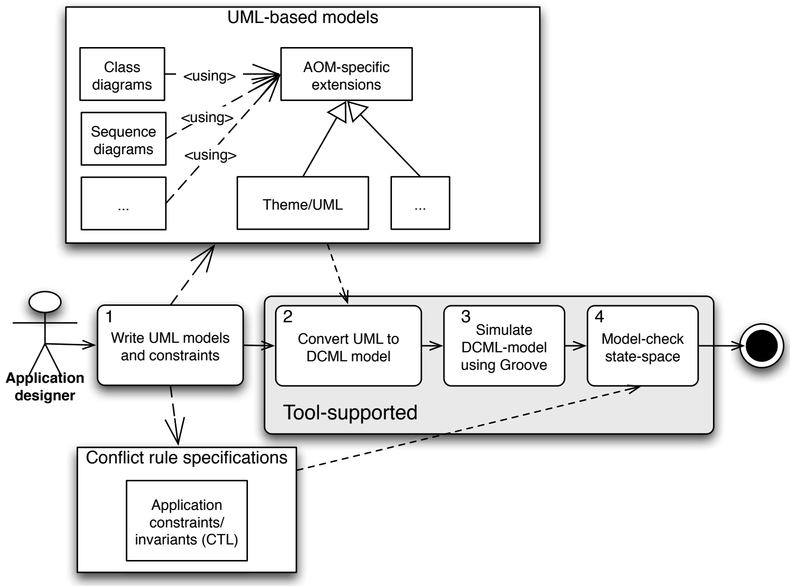

UML-based models
Class diagrams
<using>
AOM-specific extensions
Sequence diagrams
<using>
<using>
...
Theme/UML
...
Application designer
1 Write UML models and constraints
2 Convert UML to DCML model
3 Simulate DCML-model using Groove
4 Model-check state-space
Tool-supported
Conflict rule specifications
Application constraints/ invariants (CTL)

Fig. 1. An overview of our verification approach

is an existing AOM approach (not developed by the authors of this paper). Other AOM extensions may also be used, as long as these can be mapped to the Design Configuration Modeling Language (DCML) model that is used in second step — this will of course involve some effort from the designer of the respective AOM approach. Our design of the Crisis Management System, focusing particularly on class and sequence diagrams, is discussed in detail in section 3.

The next step, marked '2' in the diagram, is to convert the UML-based diagrams to DCML, an existing modeling approach that defines an operational semantics for UML-based models. The conversion of "standard" UML class- and sequence models is described in prior work [9]; to support an AOM extension to UML, it is necessary to augment the conversion step with conversion rules that interpret the aspect-related extensions. This is discussed in subsection 4.3, taking the Theme/UML-based models as an example of how such a mapping can be defined.

Once the UML/AOM models have been converted to DCML, step 3 is to use an existing tool chain to simulate the execution semantics of these models; this is discussed in subsection 4.5. Using the results from the simulation step, the last step (marked '4' in the diagram) is to model check the simulation results with regard to the constraints that have been specified by the application designer in step 1. Section 5 discusses how our entire approach is applied to the Crisis Management System case study.

3 Designing the Crisis Management System Using AOM

3.1 A Brief Overview of AOM Approaches

To date, a substantial number of AOM approaches have been proposed [37, 8]. Many of these modeling approaches define UML extensions that support the modular expression of crosscutting elements [17, 12, 5]. 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.

3.2 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 4, 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 [6, 11, 12]. 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".

3.3 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. Several 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.
- *Resource 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.
- *Real-time monitoring.* To satisfy the requirement of real-time status reports on the availability of resources, mission progress, etc., it is necessary to keep the necessary statistical information up-to-date while the system is running. This way, it is ready to be used at any time without significant delays, such as would be involved in querying a complex data structure of substantial size for various relevant information. To keep such information readily available, real-time monitoring is thus an important concern.

In this paper, we focus especially 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. Later in this section, we give examples of various incremental evolution steps.

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, 2].

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 [4]. 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.

3.4 Crisis Management System Main Class Diagram

Figure 2 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 2 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.

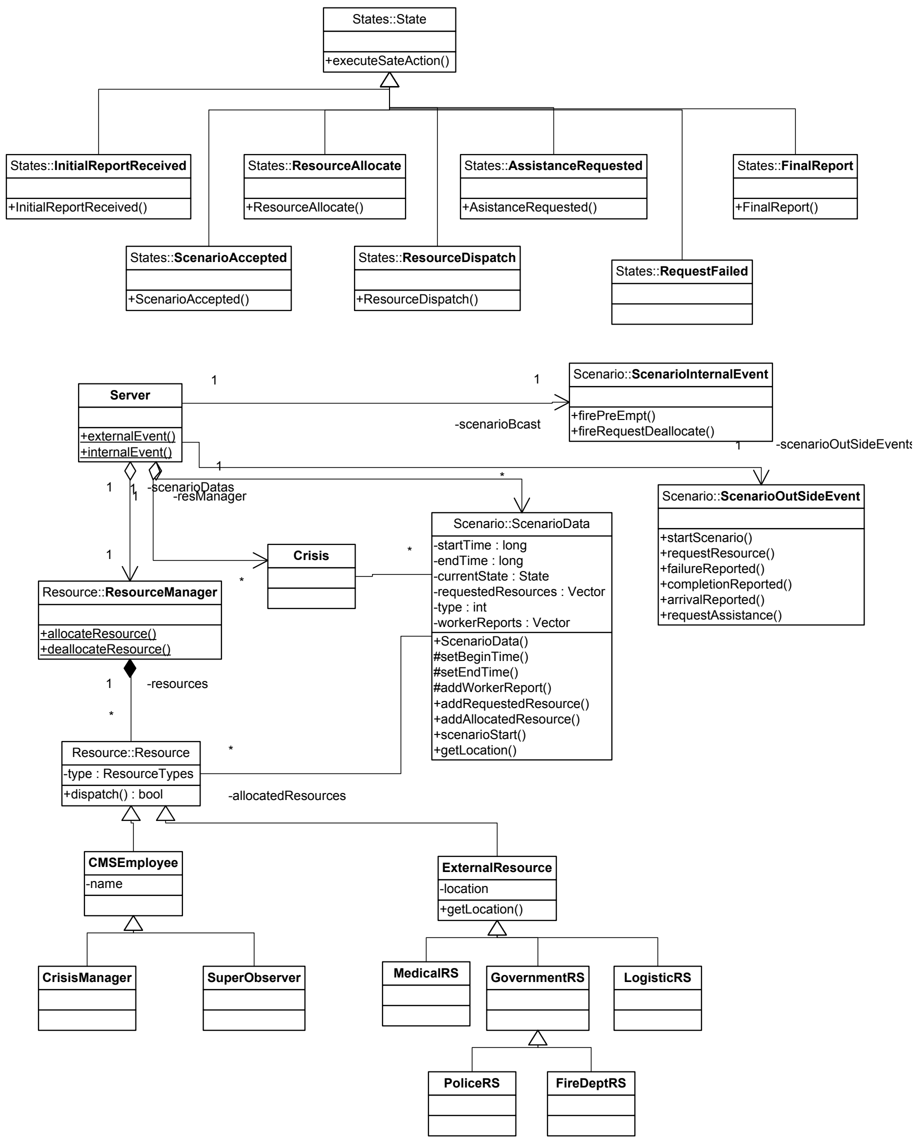


Fig. 2. Class diagram of the Crisis Management System

3.5 Modeling Gathering of Statistical Information as an Aspect

The real-time gathering of statistical information about the system is a cross-cutting concern, as it needs to track state changes all over the system, e.g. to facilitate real-time reporting on mission status. Figure 3 shows the design of such a monitoring aspect using Theme/UML. The pattern class `MonitorStatus` defines the monitoring interface; it is possible to implement different styles of monitoring, such as logging to a file or database (implemented by classes `FileLog` and `DataBaseLog`). The sequence diagram shown in Figure 3(c) shows an implementation that logs to a file. After any bound mission-state-changing operation is invoked, this sequence diagram specifies that the relevant state information should be written to a log file. The binding specification shows how this theme is bound to all the classes in the main package, i.e. all the state implementations shown in the class diagram in figure 2.

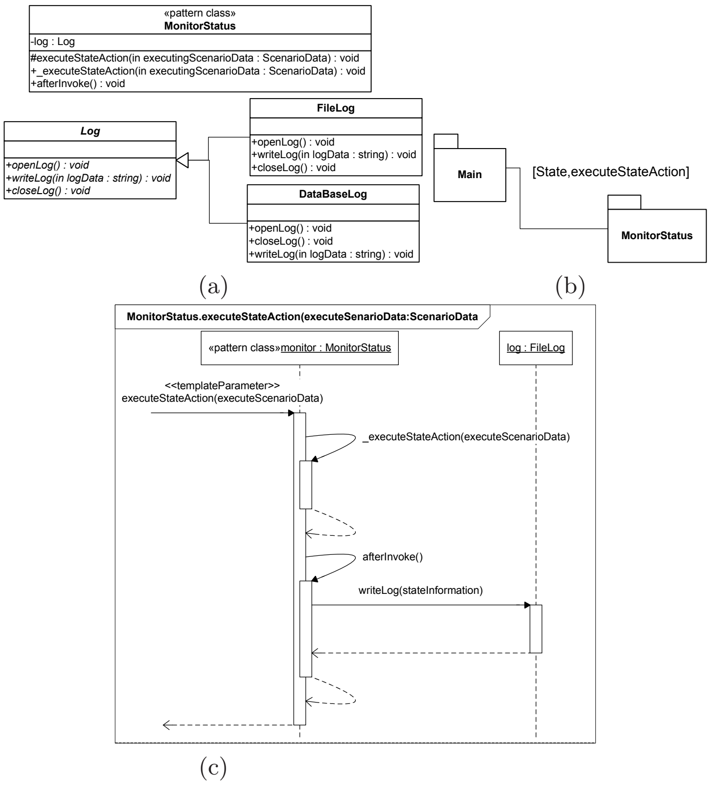

«pattern class»
MonitorStatus
-log : Log
#executeStateAction(in executingScenarioData : ScenarioData) : void
+_executeStateAction(in executingScenarioData : ScenarioData) : void
+afterInvoke() : void
Log
+openLog() : void
+writeLog(in logData : string) : void
+closeLog() : void
FileLog
+openLog() : void
+writeLog(in logData : string) : void
+closeLog() : void
DataBaseLog
+openLog() : void
+closeLog() : void
+writeLog(in logData : string) : void
Main
[State,executeStateAction]
MonitorStatus
(a)
(b)
MonitorStatus.executeStateAction(executeSenarioData:ScenarioData
«pattern class»monitor : MonitorStatus
log : FileLog
<<templateParameter>>
executeStateAction(executeScenarioData)
_executeStateAction(executeScenarioData)
afterInvoke()
writeLog(stateInformation)
(c)

Fig. 3. Monitoring of states using different storing mediums realized through Theme *MonitorState*: a) The classes of this theme. b) The binding specification. c) The sequence diagram for template parameter *_executeStateAction()*.

3.6 Modeling the Car Crash Scenario as an Aspect

We realize the Car Crash scenario as exemplified in the case study, as a distinct aspect shown in figure 4. This figure, which defines the `Car Crash` "theme", consists of two parts. The first part (Figure 4-(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

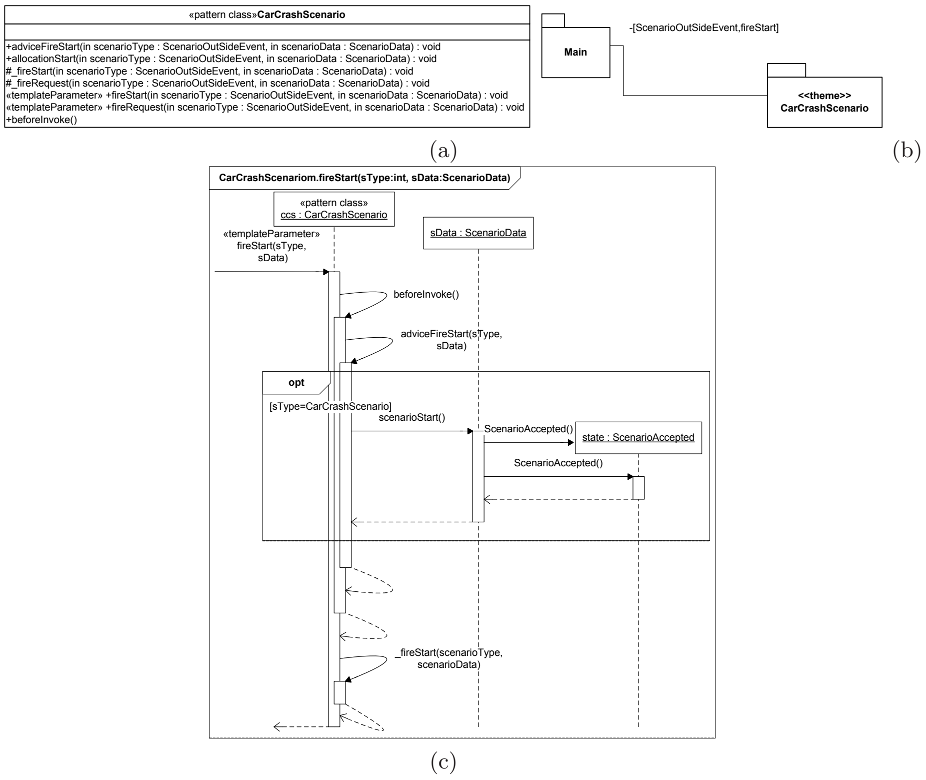
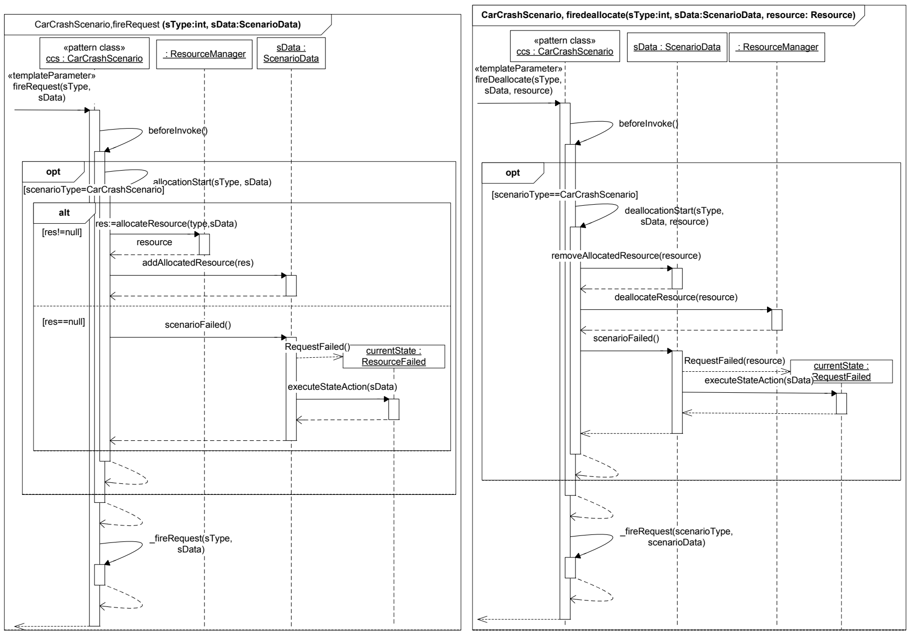

«pattern class»CarCrashScenario
+adviceFireStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
+allocationStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
#_fireStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : 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)
Main
-[ScenarioOutSideEvent,fireStart]
<<theme>> CarCrashScenario
(b)
CarCrashScenariom.fireStart(sType:int, sData:ScenarioData)
«pattern class» ccs : CarCrashScenario
sData : ScenarioData
«templateParameter» fireStart(sType, sData)
beforeInvoke()
adviceFireStart(sType, sData)
opt
[sType=CarCrashScenario]
scenarioStart()
ScenarioAccepted()
state : ScenarioAccepted
ScenarioAccepted()
_fireStart(scenarioType, scenarioData)
(c)

Fig. 4. Car Crash Scenario expressed using Theme/UML: a) The pattern class *CarCrashScenario*. b) The binding specification for the theme Car Crash Scenario. c) The sequence diagram for the template parameter *_fireStart()*.

is invoked. Figure 4-(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=CarCrashScenario", 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.

Figure 5 shows the handling of allocation and deallocation as implemented by the car crash scenario. As shown in figure 5(a), the scenario attempts to allocate resources, and if this fails (the result of allocation is a null-object), it simply switches to state `RequestFailed`. Figure 5(b) shows that resources can also be deallocated, which is only allowed after they have been allocated but not been dispatched yet. In this case, the scenario also moves to state `RequestFailed`.


CarCrashScenario,fireRequest (sType:int, sData:ScenarioData)
«pattern class» ccs : CarCrashScenario
: ResourceManager
sData : ScenarioData
«templateParameter» fireRequest(sType, sData)
beforeInvoke()
opt
[scenarioType=CarCrashScenario]
allocationStart(sType, sData)
alt
[res!=null]
res:=allocateResource(type,sData)
resource
addAllocatedResource(res)
[res==null]
scenarioFailed()
RequestFailed()
currentState : ResourceFailed
executeStateAction(sData)
_fireRequest(sType, sData)
CarCrashScenario, firedeallocate(sType:int, sData:ScenarioData, resource: Resource)
«pattern class» ccs : CarCrashScenario
sData : ScenarioData
: ResourceManager
«templateParameter» fireDeallocate(sType, sData, resource)
beforeInvoke()
opt
[scenarioType==CarCrashScenario]
deallocationStart(sType, sData, resource)
removeAllocatedResource(resource)
deallocateResource(resource)
scenarioFailed()
RequestFailed(resource)
currentState : RequestFailed
executeStateAction(sData)
_fireRequest(scenarioType, scenarioData)

(a) (b)

Fig. 5. Sequence diagrams showing the resource allocation (a) and deallocation (b) of car crash scenario

3.7 Incremental Evolution: Adding New 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 6 defines such a scenario. In principle, the crisis is handled in a similar way as discussed in section 3.6. 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. For the pattern class *PresidentialEmergencyScenario*, the request for pre-emption is shown in Figure 6-(b) with the call to the operation *Server.ScenarioBroadCastEvent()*.

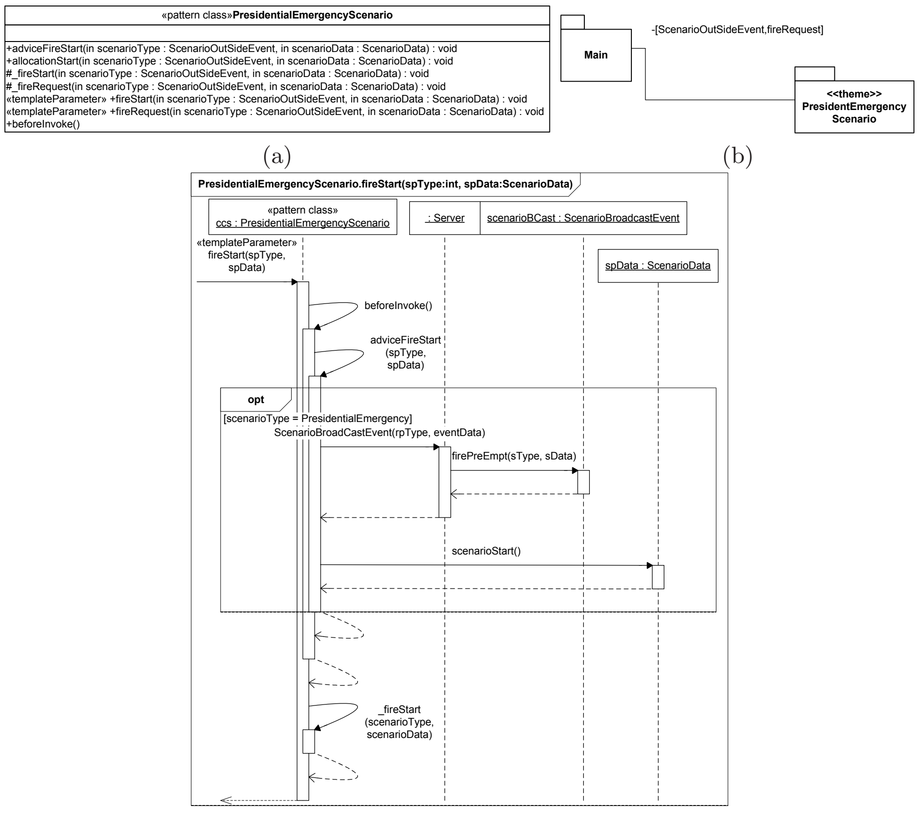

«pattern class»PresidentialEmergencyScenario
+adviceFireStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
+allocationStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : ScenarioData) : void
#_fireStart(in scenarioType : ScenarioOutSideEvent, in scenarioData : 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)
Main
-[ScenarioOutSideEvent,fireRequest]
<<theme>> PresidentEmergency Scenario
(b)
PresidentialEmergencyScenario.fireStart(spType:int, spData:ScenarioData)
«pattern class» ccs : PresidentialEmergencyScenario
: Server
scenarioBCast : ScenarioBroadcastEvent
spData : ScenarioData
«templateParameter» fireStart(spType, spData)
beforeInvoke()
adviceFireStart (spType, spData)
opt
[scenarioType = PresidentialEmergency]
ScenarioBroadCastEvent(rpType, eventData)
firePreEmpt(sType, sData)
scenarioStart()
_fireStart (scenarioType, scenarioData)
(c)

Fig. 6. Presidential Emergency scenario expressed using Theme/UML: a) The pattern class *PresidentialEmergencyScenario*. b) The binding specification for the theme Presidential Emergency Scenario. c) The sequence diagram for the template parameter *fireStart()*.

The sequence diagram in Figure 7 shows how the pattern class *PresidentialEmergencyScenario* allocates resources. This is very similar to the way the resource allocation of the car crash scenario; however, the major difference is that here failures during resource allocation are not handled. There should always be resources available for an resource allocation request by the presidential emergency scenario due to the request for resource pre-emption. Thus, there is no need for failure handling in resource allocation for this scenario.

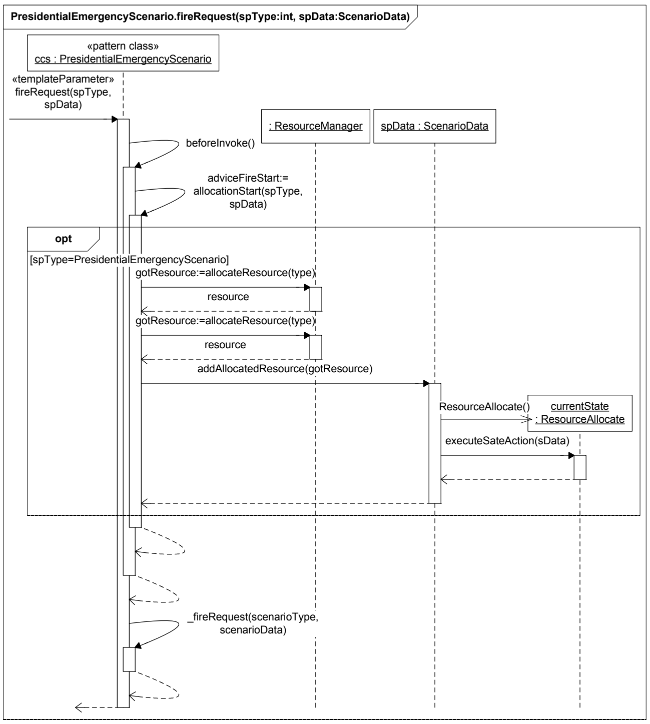

PresidentialEmergencyScenario.fireRequest(spType:int, spData:ScenarioData)
«pattern class»
ccs : PresidentialEmergencyScenario
«templateParameter» fireRequest(spType, spData)
: ResourceManager
spData : ScenarioData
beforeInvoke()
adviceFireStart:= allocationStart(spType, spData)
opt
[spType=PresidentialEmergencyScenario]
gotResource:=allocateResource(type)
resource
gotResource:=allocateResource(type)
resource
addAllocatedResource(gotResource)
ResourceAllocate()
currentState : ResourceAllocate
executeSateAction(sData)
_fireRequest(scenarioType, scenarioData)

Fig. 7. The sequence diagram from theme Presidential Emergency Scenario showing the resource allocation of the pattern class

3.8 Incremental Evolution: Resource Allocation Based on Location

Resources are allocated by calling the method `allocateResource` in class `ResourceManager`, shown in the class diagram of the Crisis Management System 2. The default implementation of this method simply looks for the first available resource of the correct type, and allocates that. This resource allocation strategy is of course very naive. To address this, as an incremental evolution step various resource allocation strategies would be modeled as a "pluggable" aspect that intercepts calls to this method, and implements various different strategies for the allocation of resources. For example, the default implementation just searches for the first unallocated resource of the correct type that is

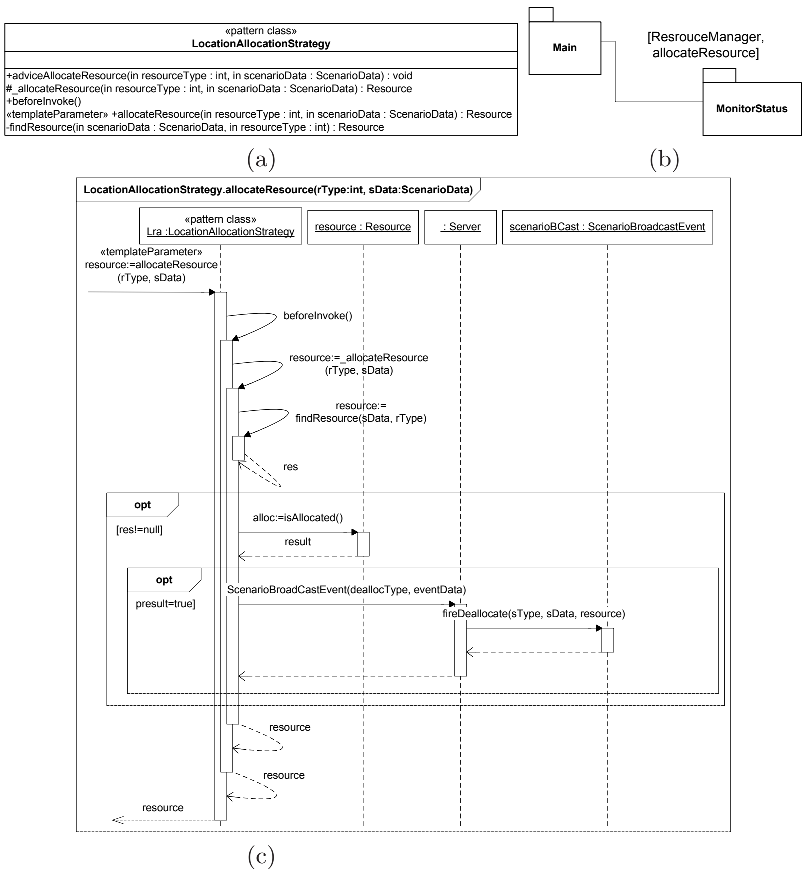

«pattern class»
LocationAllocationStrategy
+adviceAllocateResource(in resourceType : int, in scenarioData : ScenarioData) : void
#_allocateResource(in resourceType : int, in scenarioData : ScenarioData) : Resource
+beforeInvoke()
«templateParameter» +allocateResource(in resourceType : int, in scenarioData : ScenarioData) : Resource
-findResource(in scenarioData : ScenarioData, in resourceType : int) : Resource
(a)
Main
[ResrouceManager, allocateResource]
MonitorStatus
(b)
LocationAllocationStrategy.allocateResource(rType:int, sData:ScenarioData)
«pattern class» Lra :LocationAllocationStrategy
resource : Resource
: Server
scenarioBCast : ScenarioBroadcastEvent
«templateParameter» resource:=allocateResource (rType, sData)
beforeInvoke()
resource:=_allocateResource (rType, sData)
resource:= findResource(sData, rType)
res
opt
[res!=null]
alloc:=isAllocated()
result
opt
presult=true]
ScenarioBroadCastEvent(deallocType, eventData)
fireDeallocate(sType, sData, resource)
resource
resource
resource
(c)

Fig. 8. Location-based resource allocation strategy design using Theme/UML: a) The pattern class *LocationAllocationStrategy*. b) The binding specification. c) The sequence diagram for the template parameter *allocateResource()*.

available, and allocates that. More elaborate management schemes might take resource location, employee scheduling constraints, costs, etc. into account. Figure 8 shows the implementation of a resource allocation strategy that takes the location (proximity to the place of the crisis) of resources into account. It does this by attempting to allocate a resource found within a given maximum distance. If no such resource can be found (i.e., after attempting the allocation, the resource still refers to a "null"-object), an optional part of the sequence diagram in figure 8(c) is executed, which will attempt to obtain resources by first deallocating them from lower priority tasks.

3.9 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 correctly, because it simply moves to a failure state that violates application constraints, as will be shown in the following sections.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.

4 A Graph-Based Approach to Conflict Detection on UML-Based AOMs

UML-based AOM allows the software system to be decomposed into partial views. However, understanding the behavior of the composed system is difficult. Once the structure of the software is designed and the behavior is expressed in terms of execution sequences, the understanding of the overall execution sequences become even harder as one needs to trace (combine) through many sequence diagrams due to pointcuts, polymorphism and conditional executions. For example, one needs to ensure that the composed models do not violate an invariant of the software system and execute in the right sequence. We use graph-based model checking to automate the trace and verification process, 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 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. In our case, the state space contains all the execution sequences supported by the input UML sequence diagrams. The verification is realized by expressing the desired execution sequence as a CTL formula.

Aspects, polymorphism and conditional execution are the main factors altering the execution sequences. The UML diagrams should be simulated as close to the actual execution of an OO software system in order to fully capture the effects of these on the execution. Therefore, we defined a language, called the Design Configuration Modeling Language (DCML), of graph-based models that is an OO-like runtime representation of UML class and sequence diagrams. We modeled graph transformation rules that add OO-like execution semantics 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, a motivating example from the CMS software system is presented. The DCML model is detailed. Subsection 4.3 details how aspects are modeled in DCML. In subsection 4.4, examples of graph transformation rules modeling object- and aspect-oriented execution semantics are presented. Finally, subsection 4.5 explains how execution sequences can be expressed with temporal logic formulas and how these are verified.

4.1 Motivating Example: Ensuring the State Constraints of Scenarios

The design of the CMS follows a state pattern so that each scenario can track its status as detailed in Section 3. The CMS system has the constraint (or invariant) that the crisis scenarios are responsible for executing the state corresponding to the event they received. For example, a scenario receiving the start event *fireStart()* should execute the actions implemented in the method *ScenarioAccepted.executeStateAction()*.

The addition of the presidential emergency scenario and the location-based resource allocation strategy caused the events *firePreEmpt()* and *fireDellocate()* to be used. The old scenarios in the system, like the car crash scenario, did not handle these events; these scenarios may not execute the right state actions, which in turn may cause the software to crash. The designers need to trace through the sequence diagrams and ensure that the old scenarios make the call to the right subclass of the class *State* to prevent such errors. However, the error about the state changes may occur under different compositions of the software system which may be missed by the designer. For example, first presidential emergency scenario resource allocation, then executing the car crash scenario may not cause problem but the reverse order may cause a problem. Because of this, all possible compositions in all possible execution orders should be generated. In our approach, the simulation generates the state space containing all possible execution orders and compositions the design models support and the constraints of the software system in terms of execution orders are verified over this state space.

4.2 Design Configuration Modeling Language

Figure 9 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.

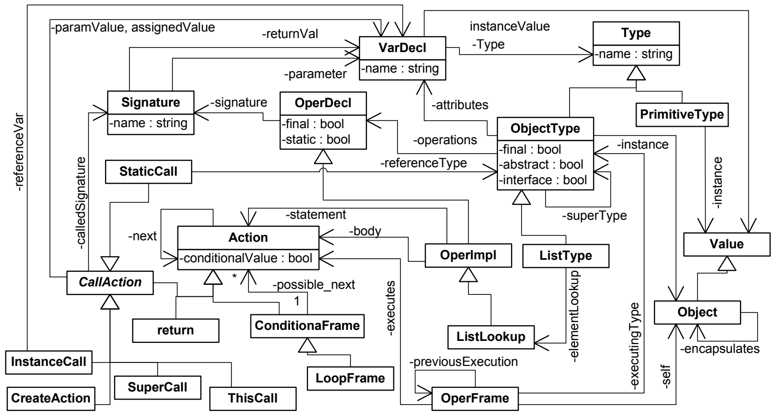


Fig. 9. The DCML meta-model

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 by 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 10-(a) shows a class diagram where the class *State* is generalized by the class *ResourceAllocate*. Figure 10-(b) shows the DCML representation of this class diagram; here, the two object-type nodes represent the classes in the class diagram.

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

Node *VarDecl* – a variable declaration.

Edge *Type* – connects a variable declaration 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.

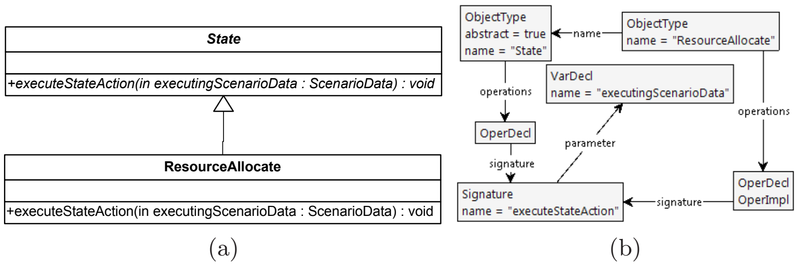

State
+executeStateAction(in executingScenarioData : ScenarioData) : void
ResourceAllocate
+executeStateAction(in executingScenarioData : ScenarioData) : void
(a)
ObjectType
abstract = true
name = "State"
name
ObjectType
name = "ResourceAllocate"
operations
VarDecl
name = "executingScenarioData"
operations
OperDecl
parameter
signature
Signature
name = "executeStateAction"
signature
OperDecl
OperImpl
(b)

Fig. 10. a) An example UML class diagram. b) The DCML model of the class diagram shown in (a).

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 line's of the 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 9), the specializations of the abstract element *Action* represent the actions of sequence diagrams. 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 receives a call.

The sequence diagram presented in Figure 11-(a) shows that in the life line of the operation *addAllocatedResource()* first a call the operation *ResourceAllocate.executeStateAction()* and then a return action is executed. Figure 11-(b) shows the lifeline of the operation *addAllocatedResource()* in DCML. Here, the emphasized node represents the call action of this life line; it is the first action of this lifeline because it is connected to the operation implementation node with an edge labeled *body*. The outgoing edge labeled *calledSignature* from this node shows 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.

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. 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.

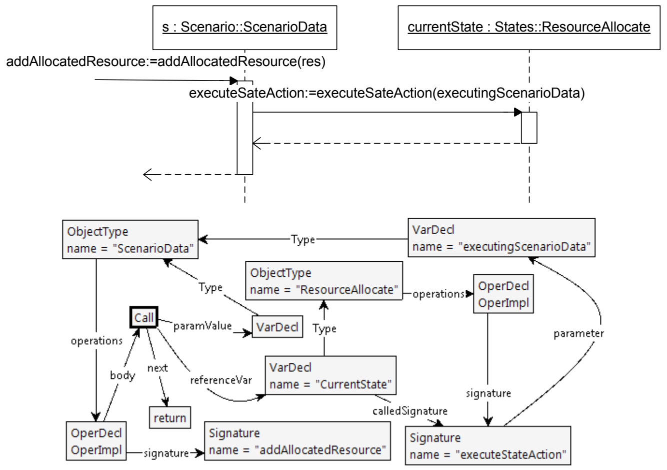

s : Scenario::ScenarioData
currentState : States::ResourceAllocate
addAllocatedResource:=addAllocatedResource(res)
executeSateAction:=executeSateAction(executingScenarioData)
ObjectType name = "ScenarioData"
Type
VarDecl name = "executingScenarioData"
ObjectType name = "ResourceAllocate"
operations
OperDecl OperImpl
Call
paramValue
VarDecl
Type
parameter
operations
body
next
referenceVar
VarDecl name = "CurrentState"
signature
return
calledSignature
OperDecl OperImpl
signature
Signature name = "addAllocatedResource"
Signature name = "executeStateAction"

Fig. 11. a) A sequence diagram showing the actions executed by the operation *ScenarioData.addAllocatedResource()*. b) These actions represented in DCML.

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 implementation, 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 *ConditionalFrame* represents alternative and optional frames.
Node *LoopFrame* represents loop frames.
Edge *possible_next* connects a conditional frame to an action node to represent the frame fragment.
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 than one instance value. During simulation, the values of the variables at the executing frame are resolved with the encapsulated edges.

4.3 Aspects in the Design Configuration Modeling Language

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

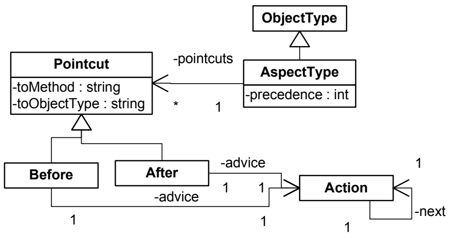


Fig. 12. The meta-model containing elements for representing aspects in DCML models

this specialization, it is possible to specify attributes and operations for aspect types. Figure 12 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. The pointcuts of aspects are represented with nodes connected to aspect-type nodes with edges labeled *pointcuts*. Depending on the advice execution, these nodes are labeled *Before*, for before advices, and *After*, for after advices. Note that a pointcut node can have only one *before* or *after* advice. Thus, two pointcut nodes are required to define before and after advices for the same method. UML-based AOM approaches extend the UML class and sequence diagrams, 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. 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:

- 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 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.
- 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 13 shows the DCML model of the "theme" *CarCrashScenario, _fireAdviceStart*; the Theme/UML diagrams of this "theme" are shown in Figure 4. 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 4-(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*.

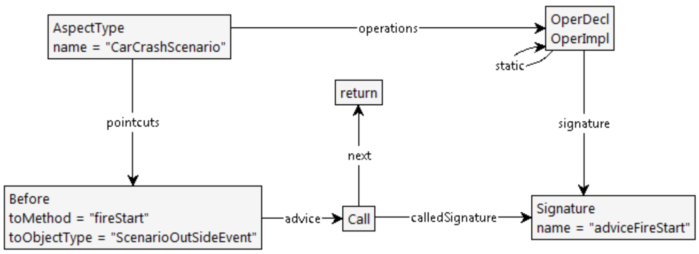


Fig. 13. The DCML model of the "theme" *CarCrashScenario*

4.4 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 OO-like execution semantics for UML sequence diagrams. In addition to these, we modeled execution semantics for before and after pointcuts with *8* transformation rules. For brevity, we summarize how these execution semantics work and detail the transformation rules modeling the semantics for before pointcuts in this section (interested readers can download the graph production system [1] and find detailed explanation of these semantics in [9]).

Before the discussion on the execution semantics, we introduce graph transformations and how they are modeled 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 none of the elements in N occurs. 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 none of the elements in N occurs then the transformation rule is said to match; a rule can have multiple matches. For each match, L is replaced by R which results in the transformed graph H.

In GROOVE, both the left-hand and the right-hand sides 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. The keyword *del* (or the color blue and dashed thin lines) is used for the edges/nodes that are deleted. The keyword *not* (or the color red and dashed bold lines) is used for negative application conditions [18].

OO-like execution Semantics for UML sequence diagrams. We modeled to the following execution semantics for the simulation of the UML sequence diagrams:

Program Counter: When the simulation of an action is complete, the transformation rule modeling the semantics of the program counter advances the simulation to the next action.

Operation Call: The operation call involves finding the receiver object of the call and, then, traversing the inheritance hierarchy to find the latest implementation of the operation. For example, if the object receiving the call implements the called operation, then the inheritance hierarchy is not traversed. If, on the other hand, this object does not implement the operation, the super-type of this object is traversed. The semantics of the operation call are implemented with 5 graph transformation rules (these transformation rules are detailed in the appendix). These rules match when the simulation reaches an instance call action node (i.e. the executes edge of the current operation frame is pointing to a node with the label *InstanceCall*)

This-Call: Because in a this-call the receiver object is the same as the object from which the call is originated, the semantics search for the operation implementation in the object type of the currently executing object. If the operation is not implemented there, the traversal in the inheritance hierarchy starts. This-calls are also implemented with 5 transformation rules and they match to the host graph when the simulation reaches a node representing the self-calls.

Super-Call: Similar to the execution semantics of the this-calls, the receiver object and the object the call is originated from do not change in super-calls. However, the implementation of the operation is searched in the super-type of the currently executing object. If the operation implementation cannot be located at the super-type then the traversal in the inheritance hierarchy starts. The semantics of the super-calls are also implemented with 5 transformation rules. These rules match when the simulation reaches a node representing super-calls.

Static Operation Call: 3 transformation rules are implemented for static operation calls, which match when the simulation reaches an action node representing a static-call. These rules locate the static operation implementation in the class referred by the call.

Parameter Pass: Parameter passing is simulated after the object-type implementing the called operation is located for all kinds of calls. The semantics for parameter pass iterate the list of parameters and copy the values/references to the frame of the called object. Parameter passing is implemented with 3 transformation rules.

Return Value Pass and Assignment: When the simulation reaches a node representing a return action, the semantics of the operation return are executed. These semantics are implemented with 4 transformation rules, where 2 are used for copying the value/reference of the return value to the frame the call is originated from, 1 is used for deleting the operation frame of the returning frame and 1 is used for assigning the return value to the variable specified in the call.

Object Creation: Object creations are simulated with 3 transformation rules, which match to the host graph when the simulation reaches a node representing the create actions. The first transformation rules adds the created object to the frame of the currently executing object and the remaining two are responsible for calling the constructor.

Conditional Execution: The transformation modeling the semantics of conditional execution picks the first action of one of the frame fragments of alternative frames and advances the program counter to that action, when the simulation reaches an conditional frame node. For optional frames, the rule either advances the program counter such that actions within the frame fragment are simulated or the transformation rule for the program counter advances the simulation to the next action (skipping the actions within the frame fragment).

Loops: Loops are realized with 2 transformation rules, which match when the simulation reaches a loop frame node. One of these transformation rules arranges the program counter so that the simulation loops over the actions within the frame fragment. The second transformation rule tests whether the loop is repeated by the user-specified amount and, if so, it terminates to loop. This rule is a generated rule. The user specifies the iteration count for the loop in the sequence diagram (this done within the guard of the loop). During the conversion from UML to DCML, the conversion algorithm generates a copy of this rule for each loop frame in the sequence diagram.

Polymorphism: When an instance call can be received by more than one object, the rule modeling the semantics of polymorphism matches. This rule changes the value of the reference variable to an object that is an instance of one of the type compatible classes. A typical scenario for polymorphism occurs when there are two or more sequence diagrams showing the same call received by different objects.

Execution Semantics of Pointcuts and Before Advices. The design of the CMS requires the scenarios to intercept the entry and exit points of the event methods. Because of this, we modeled the semantics for pointcuts and before/after advices. In this section, we detail the semantics of the pointcuts and before advices; the semantics of the after advices are similar. As discussed before, the pointcuts in DCML specify the name of the operation that is going to be intercepted. The execution semantics of a pointcut evaluate 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.

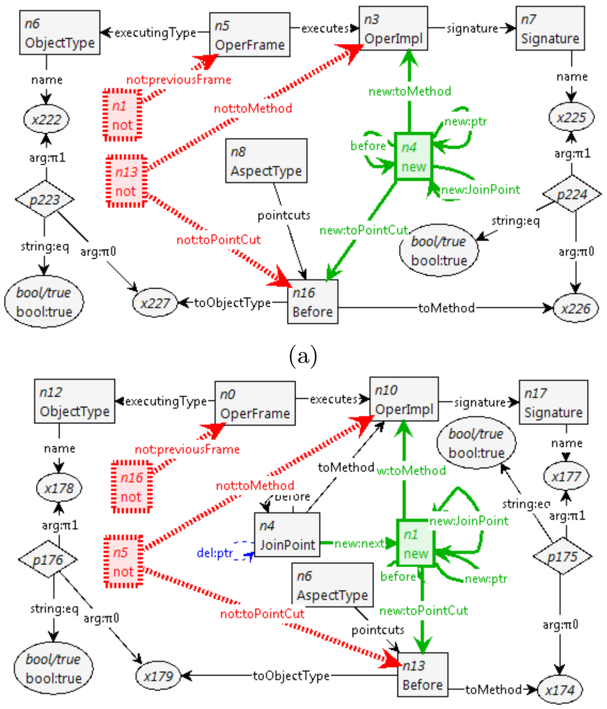


Fig. 14. The transformation rules modeling the semantics of before pointcut

Figure 14 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 14-(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 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 14-(a) matches when the simulation is at an entry point of an operation identified with the node *n8* representing the operation frame that is currently executing and the edge labeled *executes* connecting this operation frame to the operation implementation node *n15*.

– The transformation rule uses attribute operations to evaluate the pointcut at an entry point of an operation. In the transformation rule, the nodes *x274* and *x276* 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 *p272* is a production node; this is the node where the attribute operation is specified. The outgoing edge labeled *string:eq* specifies that 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 *p277* are specified as the name of the operation (node *x274*) and the value of the attribute *toMethod* (node *x276*) in Figure 14-(a). 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 *p277*). 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 14-(b) evaluates whether there are other aspects that can intercept the same operation as the aspect identified by the rule presented in Figure 14-(a). This rule also uses attribute operations to evaluate the pointcut. The main difference between this rule and the rule presented in Figure 14-(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 14-(a) identifies one of these aspects and marks it by adding a join point node. Then, the transformation rule shown in Figure 14-(b) identifies the second aspect and adds another join point node (node *n12*). However, this join point node is connected to another join point node (node *n13*) 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 14-(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 *n12* and deletes it from the previous join point node (node *n13*). Note that the transformation rules shown in Figure 14 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 use attribute operations to compare the precedence value of the aspect-type nodes.

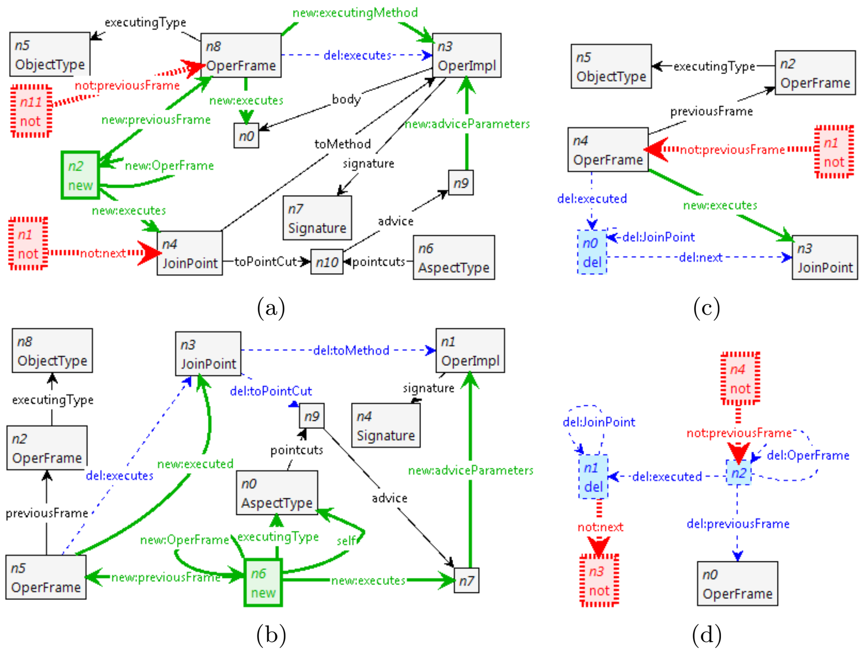


Fig. 15. a) Transformation rule for starting the execution of the join point list. b) The transformation rule that starts the execution of the advice code for a pointcut. c) The transformation rule that advances to the next join point in the join point list. d) The transformation rule that resumes the 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 15. The transformation rule shown in Figure 15-(a) adds a new operation frame, node *n2*, for traversing the join point list. This operation frame is connected to the first join point of the join point list, node *n4*, with edge labeled *executes*; thus, the dispatching of the advices starts from this join point. The operation frame node *n8* 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 15-(b) dispatches the advice code for the current join point. This is the join point node connected to the operation frame node *n3* with the edge labeled *executes*. The dispatching is realized by adding an operation frame node, node *n6*. The self of this new operation frame node is the aspect-type node, node *n0*, where the pointcut is declared. The node *n7* 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 15-(c) 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 15-(d) matches. This rule deletes the operation frame in which the join point list is traversed and resumes the execution of the intercepted operation.

The graph formalism is expressive enough to define more complex pointcuts such as pointcuts based on stack frames. One can extend the approach with a new pointcut model by adding a node type for specifically identifying the pointcut (possibly a sub-class of the node *PointCut*) to the DCML meta model with the matching criteria. Then, the semantics of the interception condition are modeled with transformation rules similar to the transformation rules presented in Figure 14.

4.5 State Space Generated from Simulation

In graph-based model, the simulation of a model generates a state space 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 transit from one state to another. When multiple rules can be applied at a certain state, one edge is created for each rule.

To remind the reader, the state space generated from the simulation of the DCM shows all the execution sequences supported by the input sequence diagrams. Figure 16 shows an excerpt from a GTS demonstrating the simulation of 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, like the transformation rule *executeMethod(operation name, object-type name)*. A rule with a parameterized name specifies a set of node attributes whose values should be output in the transition labeled instead of the parameters. When applied, GROOVE extracts the values from the target graph and replaces the parameters with the extracted value. We designed special transformation rules, called the *informative transformation rules*, that use this mechanism to display information about the operations/aspects that have executed during simulation. For example, the edge between nodes *start* and *S2*, labeled *executeMethod("fireStart", "ScenarioOutSideEvent")*, shows that the simulation is at the entry point of the operation

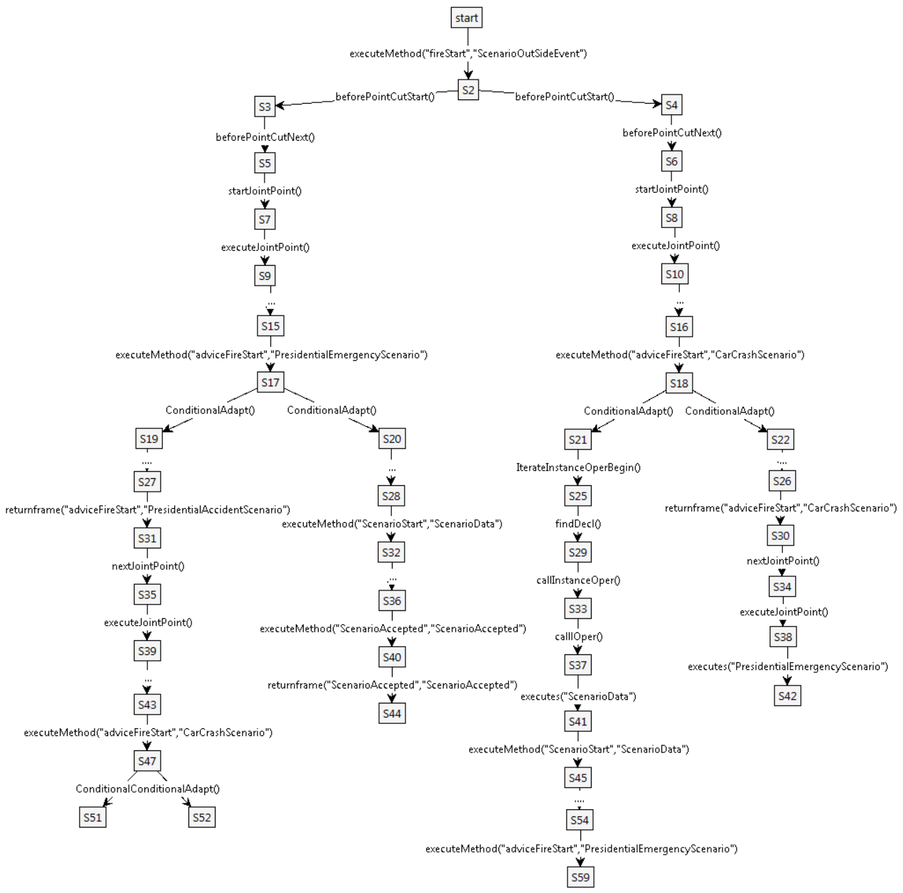

start
executeMethod("fireStart","ScenarioOutSideEvent")
S2
beforePointCutStart()
beforePointCutStart()
S3
S4
beforePointCutNext()
beforePointCutNext()
S5
S6
startJointPoint()
startJointPoint()
S7
S8
executeJointPoint()
executeJointPoint()
S9
S10
...
...
S15
S16
executeMethod("adviceFireStart","PresidentialEmergencyScenario")
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")
callIOper()
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. 16. Excerpt from a graph transition system showing advice execution and a conditional execution in the advice

ScenarioOutSideEvent. fireStart(). Other informative transformation rules are as follows:

– *executes(t)*: shows that class t has received a call.
– *returnframe(m, t)*: the return point of the method m of the class t.
– *PolymorphicReconfiguration(f, t)*: A receiver of the call has been changed from the instance of class f to an instance of the class t. This informative transformation rule matches right before the call is simulated.
– *conditionalExecutes(g)*: shows that the conditional execution has selected the frame fragment with guard g.

In the sample GTS shown in Figure 16, we see that after state *S2* the simulation continues in two branches. Each of these branches starts with a transition labeled *beforePointCutStart*. This label is the name of the transformation rule shown in Figure 14-(a). The theme *CarCrashScenario* and the theme *PresidentialEmergencyScenario* both 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 14-(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 the 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* because of the optional frame in operation *adviceFireStart* of *PresidentEmergencyScenario* (Figure 6-(c)). The semantics of the conditional execution (alternative and optional frames) are modeled with a transformation rule named *ConditionalAdapt* and each application of this rule adds a branch to the GTS that executes one of the frame fragments. In the sample GTS, it can be seen that after state *S20* the actions within the frame fragment are 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*.

4.6 Conflict Detection Using CTL

In order to identify the conflicts or requirement violations, the requirement is expressed as a temporal logic formula. Because the state space generated by the simulation contains all possible behaviors of the system, the verification of the temporal logic formula finds all the states that do not satisfy it. In our case, we are interested in searching for execution sequences that are paths in the state space. Computation Tree Logic (CTL) is a suitable formalism to search for paths in the GTS. A CTL formula is formed with atomic propositions that are ordered with temporal and logical operators (such as *and* ($\wedge$), *or* ($\vee$) and *not* !). There are two types of temporal operators in CTL:

1. Quantifiers over all paths (i.e. branches of the tree): Operator $A(x)$ used for quantifying over all paths and operator $E(x)$ used for quantifying on at least one path.
2. Quantifiers specific to a path (i.e. a branch of the tree): these operators are used to express orders in a path:
   (a) *F(x)*: means that eventually in the subsequent path x has to hold.
   (b) *G(x)*: in the entire subsequent path x has to hold.
   (c) *N(x)*: at the next state x has to hold.
   (d) *(x U y)*: x has to hold until at some state y holds.

There are two approaches to detect a constraint (invariant) violation of the software system in the state space generated: 1) using the quantifiers over all paths express the constraint execution sequence, 2) using the quantifier over a path express the violation of the constraint execution sequence. For the simulation of the UML-based AOMs, the second approach is more suitable for finding the execution sequence that causes a problem in the system as the verification would mark the states after which the violation has occurred. A violation of the constraint can be expressed using the negation. For example, if the constraint requires that after method *a()* eventually the method *b()* should execute, a violation of this constraint is an execution sequence where after method *a()* the method *b()* does not execute; this is expressed in CTL as $EF(a() \wedge !(EF(b())))$. Similarly, if the constraint states that after method *a()* method *b()* never executes, a violation of this constraint is an execution sequence where after *a()* eventually the method *b()* executes, which is expressed in CTL as $EF(a() \wedge (EF(b())))$.

In Section 4.1, the execution constraint of the scenarios on the state pattern is described. One such example of the constraint on the states is that the car crash scenario should go to the scenario accept state, by executing the method *ScenarioAccepted.executeStateAction()* when it receives a start event *fireStart*. The violation of this constraint is the execution sequence where the method *CarCrashScenario. fireStart()* executes and, before it returns, the method *ScenarioAccepted.executeStateAction()* does not execute. Using the informative transformation rules presented in Section 4.5, this execution sequence can be expressed as a CTL formula as follows:

$$\begin{aligned} EF(executeMethod(\text{"}adviceFireStart\text{"}, \text{"}CarCrashScenario\text{"}) \wedge \\ (EF(conditionalExecutes(\text{"}scenarioType = PresidentialEmergencyScenario\text{"}) \wedge \\ !(EF(executeMethod(\text{"}executeStateAction\text{"}, \text{"}ScenarioAccepted\text{"}) \wedge \\ (EF(returnframe(\text{"}adviceFireStart\text{"}, \text{"}CarCrashScenarioScenario\text{"}))))) \wedge \\ (EF(returnframe(\text{"}adviceFireStart\text{"}, \text{"}CarCrashScenarioScenario\text{"})))))) \end{aligned}$$

5 Application of the Approach to the Case Study

In this section, we describe two evolutions of the CMS that introduce semantically interfering aspects. These new aspects depend on a functionality that was not used in the initial version of the CMS software. Thus, it is important to verify that the old aspects (that are interfering with the new ones) obey the invariants of this functionality.

5.1 Evolution of CMS

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 and resource allocation strategies can easily be incorporated into the

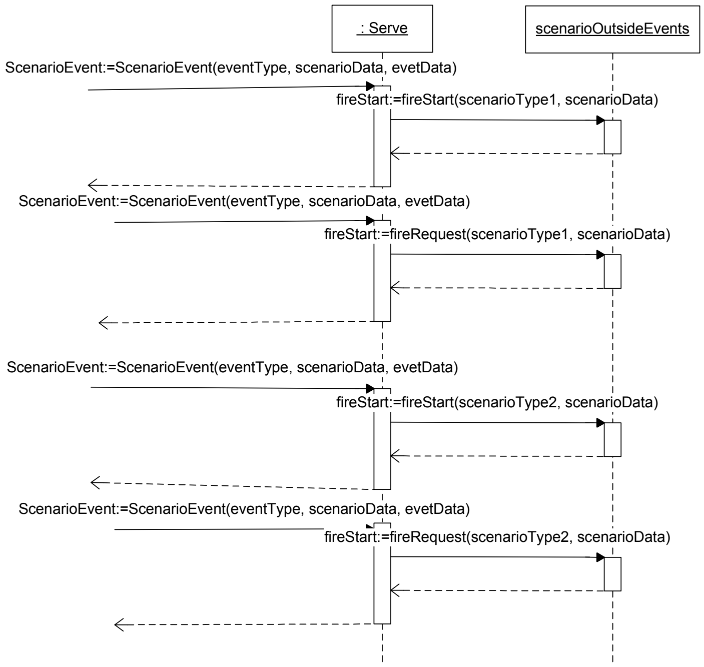

: Serve
scenarioOutsideEvents
ScenarioEvent:=ScenarioEvent(eventType, scenarioData, evetData)
fireStart:=fireStart(scenarioType1, scenarioData)
ScenarioEvent:=ScenarioEvent(eventType, scenarioData, evetData)
fireStart:=fireRequest(scenarioType1, scenarioData)
ScenarioEvent:=ScenarioEvent(eventType, scenarioData,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

Fig. 17. The class *Server* receiving 4 events from the user

system. However, before incorporating these into the system, the validity of the following two conditions should be ensured: 1) the new scenario and the new allocation strategy do 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 fulfill 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. 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.

The initial version of the CMS allocated the first available resource to a crisis (Figure 5). The efficiency can be improved if a resource that is closer to the crisis can be dispatched, even if the resource is already dispatched to another crisis. To address this requirement, the designers introduced a new pattern called *LocationAllocationStrategy* that implements the resource allocation according to the location of the resources (Figure 8. This pattern class makes use of the resource deallocate broadcast event operation *ScenarioBroadcastEvent.fireRequestDeallocate()* to notify a scenario about deallocation. The scenarios are required to implement a “before invoke” operation to listen to this broadcast event. This before invoke operation should re-request deallocated resources.

As shown in Section 3, the newly added strategy and crisis scenario are modeled as new “themes” without modifying the UML diagrams of the initial version of the CMS. These newly added “themes”, however, semantically interfere with the car crash scenario because they depend on the old scenarios to handle requests correctly. In the remaining parts of this section, we will show how these interferences can be simulated and how possible system invariant violations can be captured.

5.2 Simulation of the UML Models of the CMS

The sequence diagram in Figure 17 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", Monitoring, Car Crash Scenario, Presidential Emergency Scenario and the Location-based allocation strategy.

The generated DCML model from these diagrams contains 715 graph elements (nodes and edges). In this DCML model, the pattern classes are mapped to the aspect types *CarCrashScenario*, *PresidentialEmergencyScenario*, *Monitoring* and *LocationAllocationStrategy*. The simulation of this model generated a state space consisting of 47015 states, 47681 transitions; the simulation took 2.38 minutes[2]. 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.

5.3 Verification of the Constraints of the CMS After Evolution

Using the state space generated by the simulation of the UML diagrams, we evaluate the following invariants of the presidential emergency scenario so that introduction of this aspect does not introduce errors:

- **The states of the presidential emergency scenario should match to the outside events it responds to:** The subclasses of the class *State* reflect the states of a crisis scenario. Each subclass is responsible for creating a report about the scenario (e.g. the time the change in the state occurs), which are then logged for statistical purposes. Thus, it is very important for a scenario to instantiate the right subclass of the class *State* for the outside events. For example, the advice operation *PresidentEmergency.adviceFireStart()* for the outside event *fireStart()* should set its state to an instance of the class *InitialReportReceived*. The following CTL formula finds the states where this advice operation does not set its state to an instance of the class *InitialReportReceived*:

$$\begin{aligned} &EF(executeMethod(\text{"}adviceFireStart\text{"}, \text{"}PresidentialEmergencyScenario\text{"}) \wedge \\ &(EF(conditionalExecutes(\text{"}scenarioType = PresidentialEmergencyScenario\text{"}) \wedge \\ &!(EF(executeMethod(\text{"}executeStateAction\text{"}, \text{"}ScenarioAccepted\text{"}) \wedge \\ &(EF(returnframe(\text{"}adviceFireStart\text{"}, \text{"}PresidentialEmergencyScenario\text{"})))))) \wedge \\ &(EF(returnframe(\text{"}adviceFireStart\text{"}, \text{"}PresidentialEmergencyScenario\text{"})))))) \end{aligned}$$

[2] The simulation was executed on a laptop with Core 2 Duo 2.4 GHz CPU 4GB Ram running Windows 7 Ultimate 64-bit and JDK 1.6 Update 6.

This formula evaluates to true for states after which the operation *PresidentialEmergencyScenario.adviceFireStart()* eventually executes and, then, eventually this operation returns before the constructor of the class *InitialReportReceived* is called. Note that the state actions are only executed when the scenario type is executed within the optional frame and with the proposition *conditionalExecutes("scenarioType = PresidentialEmergencyScenario")* the formula searches the path where these actions are executed. The verification did not find any states that satisfy this formula, meaning that the advice operation *adviceFireStart* sets the scenario *PresidentialEmergency* to the correct state. We verified that all the advice operations of the class *PresidentialEmergencyScenario* set the correct states with the same formula by changing the advice operation name and its respective state class name.

– **The states of the presidential emergency scenario should be monitored.** The aspect-type *Monitoring* intercepts the returns from the operation *executeStateAction* of all the sub-classes of the class *State* to log the state information. This behavior should not be harmed by introduction of the aspect-type *PresidentialEmergencyScenario.* If there is an execution sequence where the operation *executeStateAction* does not execute after the advice code of the aspect-type *PresidentialEmergencyScenario*, then the monitoring of this aspect does not behave as it should. With the following CTL formula, the execution sequence where the monitoring of the advice *PresidentialEmergencyScenario. adviceFireStart()* fails can be searched:

$$
\begin{aligned}
&EF(executeMethod(\text{"}adviceFireStart\text{"},\text{"}PresidentialEmergencyScenario\text{"}) \wedge\\
&(EF(conditionalExecutes(\text{"}scenarioType = PresidentialEmergencyScenario\text{"}) \wedge\\
&(EF(returnframe(\text{"}executeStateAction\text{"},\text{"}InitialReportReceived\text{"}) \wedge\\
&!(EF(executeMethod(\text{"}afterInvoke\text{"},\text{"}MonitorStatus\text{"}))) \wedge\\
&(EF(returnframe(\text{"}adviceFireStart\text{"},\text{"}PresidentialEmergencyScenario\text{"}))))))))
\end{aligned}
$$

This formula looks for an execution sequence where the aspect *PresidentialEmergencyScenario* changes its state to *InitialReportReceived* by calling the operation *executeStateAction().* However, the aspect *MonitorStatus* does not execute. The verification did not find any states that satisfy this formula so the monitoring of *adviceFireStart()* works. By changing the operation *adviceFireStart()* and the class *InitialReportReceived* to the other advice operations and their respective states, we verified that the aspect *PresidentialEmergencyScenario* is monitored correctly.

– **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:

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

Note that here *allocationStart()* is an operation called by the advice for the operation *fireRequest()* as shown in Figure 7. The verification was able to find states that satisfied this formula; thus, the invariant was 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 7 does not specify any frame fragments that are executed when resource allocation fails (i.e. when *null* is returned). Thus, the execution of the advice did not complete successfully because it threw 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 that the call to the operation *getType()* fails:

$$EF(executeMethod(\text{"}allocationStart\text{"},\text{"}PresidentialEmergencyScenario\text{"}) \wedge (EF(executeMethod(\text{"}getType\text{"},\text{"}Resource\text{"}) \wedge !(EF(returnframe(\text{"}getType\text{"},\text{"}Resource\text{"})))))$$

– **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:

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

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* do not specify a sequence diagram with a before or after invocation of the operation *firePreEmpt()* because pre-emption was not a requirement of the initial version of the crisis management system. As a result, there is no pointcut for this operation in the aspect-type *CarCrashScenario*.

– **The scenario car crash should deallocate its resources if it receives a request to deallocate.** The location-based resource allocation strategy allocates resources according to their location. Thus, a previously allocated resource may be allocated to another crisis. The deallocation request is sent to the scenarios using the method *ScenarioInternalEvent.fireDeallocate()*. The scenarios should intercept this call and deallocate the resource by calling the method *ResourceManager.deallocateResource()*. The car crash scenario violates this constraint if it does not intercept the event *fireDeallocate()* or does not call the method *deallocateResource()* after intercepting the event, which can be expressed with the following CTL formula:

$$\begin{aligned} &EF(executeMethod(\text{"}fireDeallocate\text{"}, \text{"}ScenarioInternalevent\text{"}) \wedge \\ &!(EF(executeMethod(\text{"}fireDeallocate\text{"}, \text{"}CarCrashScenario\text{"}))) \vee \\ &!(EF(executeMethod(\text{"}fireDeallocate\text{"}, \text{"}CarCrashScenario\text{"}) \wedge \\ &(EF(executeMethod(\text{"}deallocateResource\text{"}, \text{"}ResourceManager\text{"}))))) \\ &(EF(returnframe(\text{"}fireDeallocate\text{"}, \text{"}ScenarioInternalevent\text{"})))) \end{aligned}$$

The verification did not find any states that satisfy this formula, so the car crash scenario does not violate this constraint. The UML diagrams of the "theme" *CarCrashScenario* in Figure 5-(b) show that this scenario intercepts the call to the method *fireDeallocate()* and calls the method *ResourceManager.deallocateResource()*.

– **After receiving the deallocation resource the car crash scenario should request a new resource.** If in any path the aspect-type *CarCrashScenario* does not make a call to the method *ResourceManager.allocateResource()* within the advice for the method *fireDeallocate()*, this invariant is violated. This violation can be expressed in CTL as:

$$\begin{aligned} &EF(executeMethod(\text{"}fireDeallocate\text{"}, \text{"}ScenarioInternalevent\text{"}) \wedge \\ &(EF(executeMethod(\text{"}fireDeallocate\text{"}, \text{"}CarCrashScenario\text{"}) \wedge \\ &(EF(executeMethod(\text{"}allocateResource\text{"}, \text{"}ResourceManager\text{"}) \wedge \\ &(EF(returnframe(\text{"}fireDeallocate\text{"}, \text{"}CarCrashScenario\text{"}) \wedge \\ &(EF(returnframe(\text{"}fireDeallocate\text{"}, \text{"}ScenarioInternalevent\text{"})))))))))) \end{aligned}$$

The verification was able to find states that satisfy this formula, meaning that the invariant is violated. A closer look at the design of the car crash scenario shows that after receiving the deallocation request, it goes to the *request failed* state. The deallocation request in the initial version of the crisis management system was only used to cancel the scenario. However, with the introduction of the location-based resource allocation strategy, the constraint on the deallocation request is changed.

The verification shows that with the evolution 3 constraints of the crisis management system are violated. The main reason for this failure is the incremental evolution: the "theme"s allowed the crisis management system to be evolved without modifying the existing UML models. However, the evolutions do require the UML models of the initial version of the crisis management system to be modified. A designer not evaluating the composed system may clearly miss these errors and introduce bugs to the implementation.

5.4 State Space Size and Methods for Pruning the State Space

The size of the state space depends on the number of actions in the sequence diagrams and the number of alternative execution sequences. Because the number of simulated actions may change in each alternative execution sequence, the exact size of the state space may not be calculated. Nonetheless, an upper bound on the state space size can be provided as: $O(n^k \times c^{nk} \times s)$. Here, n is the number of aspects, k is the number of operations with shared join points, c is the maximum number of alternative/optional frames in the aspects and s is the number of actions in the sequence diagrams. On average, the state space size is much less than this 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 too 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 aspect are 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 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 4.5, 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 this. 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 from the marked path. For example, the conditional paths of the aspects *CarCrashScenario* and *PresidentialEmergencyScenario* are exclusive in that, when one of them executes the actions within the frame fragment the other one immediately 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 are 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 simulated different configurations of the UML models without the reduction mechanisms and with the reduction mechanisms. Tables 1–3 report the state size, the simulation time and the total memory used by the JavaVM in each configuration of the UML models of CMS. Here, the first column shows the number of aspects in the simulated configurations; below the details of the configuration are provided:

- 3 Aspects: in this configuration the UML models of the "theme"s *Monitoring*, *Car Crash Scenario* and *Presidential Emergency Scenario* are simulated
- 4 Aspects: the "theme" *Location Based Resource Allocation* is added.
- 5 Aspects: the "theme" *Fire Scenario* is added. This scenarios sequence diagrams are very similar to the diagrams of the "theme" *Car Crash Scenario.*

When 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 is generated. In this case, the number of states is bounded above by $O(c^{nk} \times s)$.

For the configuration with 5 aspects when exclusive "ifs" are used with aspect precedence, in total, the state space contains 99 branches. This is because in the sequence diagram given in Figure 17, the subsequent invocations to the operations *fireStart()* and *fireResourceAllocate()* used the same value. After the first

Table 1. State space size without reduction mechanism

# of Aspects	# of States	# of Transitions	Simulation time in min.	Memory used in Mb
3	14815	15303	1.1	50
4	111943	113643	6.23	67
5	798600	831499	20.13	134

Table 2. State space size with Precedence as reduction mechanism

# of Aspects	# of States	# of Transitions	Simulation time in min.	Memory used in Mb
3	6993	7251	0.38	46
4	47015	47681	2.38	54
5	106167	108357	7.13	61

Table 3. State space size with Precedence and Exclusive "ifs" as reduction mechanism

# of Aspects	# of States	# of Transitions	Simulation time in min.	Memory used in Mb
3	1633	1697	0.12	43
4	7489	7635	0.46	48
5	9991	10234	0,71	49

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 none of the aspects has 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 to 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 17). 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 for mutual exclusive "ifs" is bounded above by $O(n^k)$. This worst case happens when all 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, in the CMS, four operation invocations are simulated with two distinct values. This reduced the state space size to be bounded above by $O(n^2 \times c_{non-mutual} \times s)$ where $c_{non-mutual}$ is the number of conditions that are not mutually exclusive.

Note that the growth on the state space size on the configuration with 4 aspects is due to the design of the aspect *LocationAllocationStrategy*. This aspect intercepts all the calls to the methods *ResourceManager.allocateResource()* and *ResourceManager.deallocateResource()*. The advice code for the method *allocateResource()* has two optional frames as shown in Figure 8-(c); thus, each interception of this method adds 3 branches to the state space. In the simulation there are in total 3 calls to the method *allocateResource()* which adds 27 branches to the state space that were not in the simulation of the configuration with 3 aspects. However, the state space size does not grow by a factor 27 because some of these branches are isomorphic and are merged during simulation.

6 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 [31], to formalize refactorings [30], 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 also lacks 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 of use-case charts that enable 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 of UML state charts [27] [29]. For example, Kung et al. [27] generate a graph grammar that models the execution semantics for a given state chart. These semantics, however, work only for 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 and 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. 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 et al. 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 meta-model 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 operational 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, [7] 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 occur in models expressed using the Aspect-UML language. To this aim, the static language elements of Aspect-UML are mapped to the specification language Alloy. The operational semantics of Aspect-UML models, however, have to be expressed manually using the dynamic state-based specification language 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 [3], 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.

7 Discussions and Conclusions

In this article, we have analyzed the example CMS case, identified two crisis scenarios as aspects, defined 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 4.4. In the current version, our tool supports UML class and sequence diagrams. A large library of rules is 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 and 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 is 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 3, this pruning method has reduced the generated states more than a factor 9. To further reduce the state space, we have identified the critical operations which cause branching in the state space. As shown in Figure 16, such 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 with a factor over 300; and the state space growth with the number of aspects n is bounded above by $O(n^2)$. 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 beyond 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 metamodel. Secondly, the aspect and pointcut designators of the AOM must be mapped to our pointcut model as specified in Figure 12.

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 interference problems that can be experienced in the current UML-based AOM approaches. As discussed in the article, there may be certain complex cases to which the tool cannot scale. These, however, are considered as exceptional cases rather than routine.

References

1. Gace: Graph-based adaptation, configuration and evolution modeling, `http://trese.cs.utwente.nl/willevolve/`
2. Akşit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: Abstracting object interactions using composition filters. In: Guerraoui, R., Nierstrasz, O., Riveill, M. (eds.) ECOOP-WS 1993. LNCS, vol. 791, pp. 152–184. Springer, Heidelberg (1994)
3. Aksit, M., Rensink, A., Staijen, T.: A Graph-Transformation-Based Simulation Approach for Analysing Aspect Interference on Shared Join Points. In: AOSD 2009: Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development, Charlottesville, Virginia, USA, pp. 39–50. ACM, New York (2009)
4. Altisen, K., Maraninchi, F., Stauch, D.: Aspect-oriented programming for reactive systems: Larissa, a proposal in the synchronous framework. Science of Computer Programming 63(3), 297–320 (2006)
5. Araújo, J., Whittle, J., Kim, D.-K.: Modeling and composing scenario-based requirements with aspects. In: Proc. 12th Int'l Requirements Engineering Conference, pp. 53–62. IEEE, Los Alamitos (September 2004)
6. Baniassad, E., Clarke, S.: Theme: An approach for aspect-oriented analysis and design. In: Proceedings of the 26th International Conference on Software Engineering, pp. 158–167. IEEE Computer Society, Washington (2004)

7. Chitchyan, R., Rashid, A., Rayson, P., Waters, R.: Semantics-based composition for aspect-oriented requirements engineering. In: Proceedings of the 6th international conference on Aspect-oriented software development, pp. 36–48. ACM, New York (2007)
8. Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Alarcon, M.P., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A.: Survey of aspect-oriented analysis and design approaches. Technical Report AOSD-Europe-ULANC-9, AOSD-Europe (May 2005)
9. Ciraci, S.: Graph Based Verification of Software Evolution Requirements. PhD thesis, University of Twente (December 2009)
10. Ciraci, S., van den Broek, P., Aksit, M.: Framework for computer-aided evolution of object-oriented designs. In: COMPSAC, pp. 757–764 (2008)
11. Clarke, S., Walker, R.J.: Composition patterns: An approach to designing reusable aspects. In: Proc. 23rd Int'l Conf. Software Engineering (ICSE), May 2001, pp. 5–14 (2001)
12. Clarke, S., Walker, R.J.: Generic aspect-oriented design with Theme/UML. In: Filman, R.E., Elrad, T., Clarke, S., Akşit, M. (eds.) Aspect-Oriented Software Development, pp. 425–458. Addison-Wesley, Boston (2005)
13. Dürr, P.E.A.: Resource-based Verification for Robust Composition of Aspects. PhD thesis, University of Twente, Enschede (June 2008)
14. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In: Claus, V., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1978. LNCS, vol. 73, pp. 1–69. Springer, Heidelberg (1979)
15. Engels, G., Heckel, R., Sauer, S.: Dynamic Meta Modeling: A Graphical Approach to Operational Semantics of Behavioral Diagrams in UML (1999)
16. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML Activities Using Dynamic Meta Modeling. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 76–90. Springer, Heidelberg (2007)
17. France, R., Ray, I., Georg, G., Ghosh, S.: Aspect-oriented approach to early design modelling. IEE Proceedings Software 151(4), 173–185 (2004)
18. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Inf. 26(3-4), 287–313 (1996)
19. Havinga, W.K., Nagy, I., Bergmans, L.M.J., Akşit, M.: A graph-based approach to modeling and detecting composition conflicts related to introductions. In: de Moor, O. (ed.) Proceedings of International Conference on Aspect Oriented Software Development, AOSD 2007, Vancouver, Canada, March 2007. ACM International Conference Proceedings Series, pp. 85–95. ACM Press, New York (2007)
20. Helm, R., Holland, I., Gangopadhyay, D.: Contracts: specifying behavioral compositions in object-oriented systems. ACM Sigplan Notices 25(10), 169–180 (1990)
21. Jayaraman, P.K., Whittle, J.: Ucsim: A tool for simulating use case scenarios. In: ICSE COMPANION 2007: Companion to the proceedings of the 29th International Conference on Software Engineering, Washington, DC, USA, 2007, pp. 43–44. IEEE Computer Society, Los Alamitos (2007)
22. Kastenberg, H., Kleppe, A.G., Rensink, A.: Defining OO Execution Semantics Using Graph Transformations. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 186–201. Springer, Heidelberg (2006)
23. Kastenberg, H., Rensink, A.: Model Checking Dynamic States in GROOVE. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 299–305. Springer, Heidelberg (2006)
24. Katz, E., Katz, S.: Verifying scenario-based aspect specifications. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, p. 432. Springer, Heidelberg (2005)

25. Katz, S., Rashid, A.: From aspectual requirements to proof obligations for aspect-oriented systems. In: Proceedings 12th IEEE International Requirements Engineering Conference, pp. 48–57 (2004)
26. Kleppe, A., Rensink, A.: 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. Electronic Communications of the EASST, vol. 10, p. 16 (2008)
27. Kong, J., Zhang, K., Dong, J., Xu, D.: Specifying behavioral semantics of UML diagrams through graph transformations. J. Syst. Softw. 82(2), 292–306 (2009)
28. Koppen, C., Störzer, M.: PCDiff: Attacking the fragile pointcut problem. In: Gybels, K., Hanenberg, S., Herrmann, S., Wloka, J. (eds.) European Interactive Workshop on Aspects in Software (EIWAS) (September 2004)
29. Kuske, S.: A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 241–256. Springer, Heidelberg (2001)
30. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Software and Systems Modeling 6(3), 269–285 (2007)
31. Mens, T., van der Straeten, R., D'Hondt, M.: Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 200–214. Springer, Heidelberg (2006)
32. Mostefaoui, F., Vachon, J.: Design-level Detection of Interactions in Aspect-UML models using Alloy. Journal of Object Technology 6, 137–165 (2007)
33. Mostefaoui, F., Vachon, J.: Verification of Aspect-UML models using Alloy. In: Proceedings of the 10th international workshop on Aspect-oriented modeling, pp. 41–48. ACM, New York (2007)
34. Nagy, I., Bergmans, L., Havinga, W., Aksit, M.: Utilizing Design Information in Aspect-Oriented Programming. In: Robert Hirschfeld, A.P., Kowalczyk, R., Weske, M. (eds.) Proceedings of International Conference NetObjectDays, NODe 2005, Erfurt, Germany, September 2005. Lecture Notes in Informatics, vol. P-69, Gesellschaft für Informatik, GI (2005)
35. Rich, C., Wills, L.: Recognizing a program's design: a graph-parsing approach. IEEE Software 7(1), 82–89 (1990)
36. Sampaio, A., Chitchyan, R., Rashid, A., Rayson, P.: EA-Miner: a tool for automating aspect-oriented requirements identification. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp. 352–355. ACM, New York (2005)
37. Schauerhuber, A., Schwinger, W., Kapsammer, E., Retschitzegger, W., Wimmer, M., Kappel, G.: A survey on aspect-oriented modeling approaches. Relatorio tecnico, Vienna University of Technology (2007)
38. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering 10(2), 203–232 (2003)
39. Whittle, J.: Precise specification of use case scenarios. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 170–184. Springer, Heidelberg (2007)
40. Whittle, J., Jayaraman, P.K.: Generating hierarchical state machines from use case charts. In: RE 2006, Washington, DC, USA, 2006, pp. 16–25. IEEE Computer Society, Los Alamitos (2006)
41. Zhao, C., Kong, J., Dong, J., Zhang, K.: Pattern-based design evolution using graph transformation. J. Vis. Lang. Comput. 18(4), 378–398 (2007)

Appendix – DCML Elements and Execution Semantics

In this appendix, all elements of the DCML and the execution semantics for operation dispatch 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 18 repeats the DCML meta model.

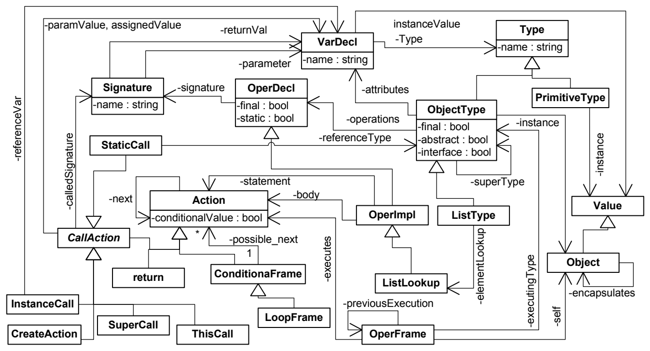


Fig. 18. The DCML meta-model

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 super-type. Figure 19-(a) shows a portion of the class diagram from the CMS with three classes, namely *State*, *ResourceAllocation* and *ScenarioData*. Figure 19-(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.

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 19. In the DCML equivalent of this class diagram

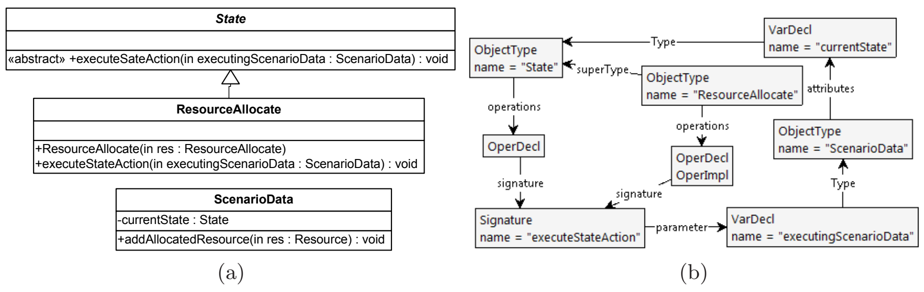

State
«abstract» +executeSateAction(in executingScenarioData : ScenarioData) : void
ResourceAllocate
+ResourceAllocate(in res : ResourceAllocate)
+executeStateAction(in executingScenarioData : ScenarioData) : void
ScenarioData
-currentState : State
+addAllocatedResource(in res : Resource) : void
(a)
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. 19. a) An example UML class diagram. b) The DCML model of the class diagram shown in (a).

in Figure 19, 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 19-(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 19-(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 19-(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.

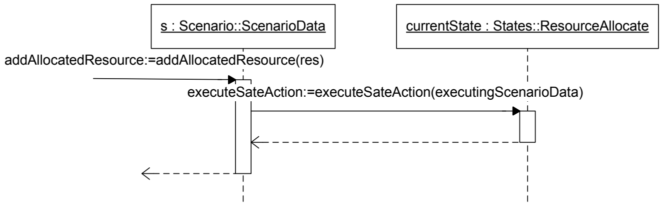

s : Scenario::ScenarioData
currentState : States::ResourceAllocate
addAllocatedResource:=addAllocatedResource(res)
executeSateAction:=executeSateAction(executingScenarioData)

Fig. 20. 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 18), 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 20 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 *addAllocatedResource()* returns. Figure 21 shows the DCML model generated from this sequence diagram (and the class diagram in Figure 19). 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.

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 20 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 21, which represents it, is also labeled *InstanceCall*.

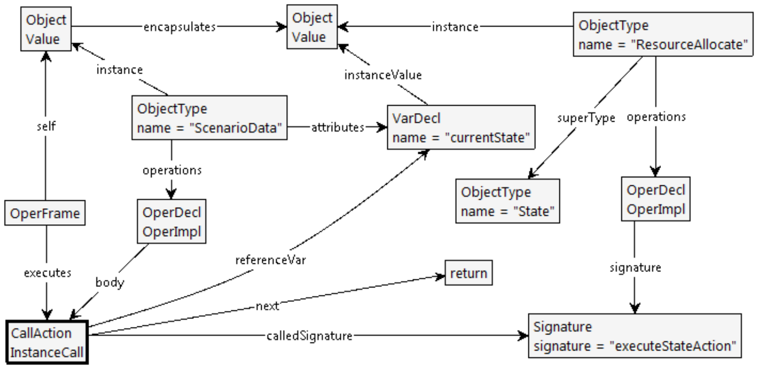


Fig. 21. A snapshot from the simulation of the sequence diagram shown in Figure 20

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 20 is represented as a variable declaration node with the same name in the DCML model of this sequence diagram as shown in Figure 21. 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 19. 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 *instance-Value*). Following the edge labeled *instanceValue* from the variable declaration node named *currentState* in Figure 21, 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.

In UML 2.0 sequence diagrams, conditional execution of the actions is represented by *frames* whose operators are *alt* or *opt* (*alt* stands for alternative and *opt* stands for optional). A frame can have one or more fragments which contain the actions. A fragment with a guard shows that the actions within the fragment are executed when the guard is true. In DCML, these frames are represented with nodes labeled *ConditionalFrames* (conditional frame node). The first action of each fragment is connected to the conditional frame node with an edge labeled *possible_next*. For example, an optional frame would be modeled in DCML, with a conditional frame node that is connected two action nodes: the edge labeled *possible_next* connects the conditional frame node to the first action within the frame and the edge labeled *next* connects it to the first action node that comes after the frame.

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 21, 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 21 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.

Execution Semantics of Operation Dispatch

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 22 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 22-(a) is used for finding the reference type of the call. This is done by finding the object-type whose instance the reference variable of the call action is holding. In this figure, the reference variable is node $n3$ (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 $n9$. When applied, this 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.

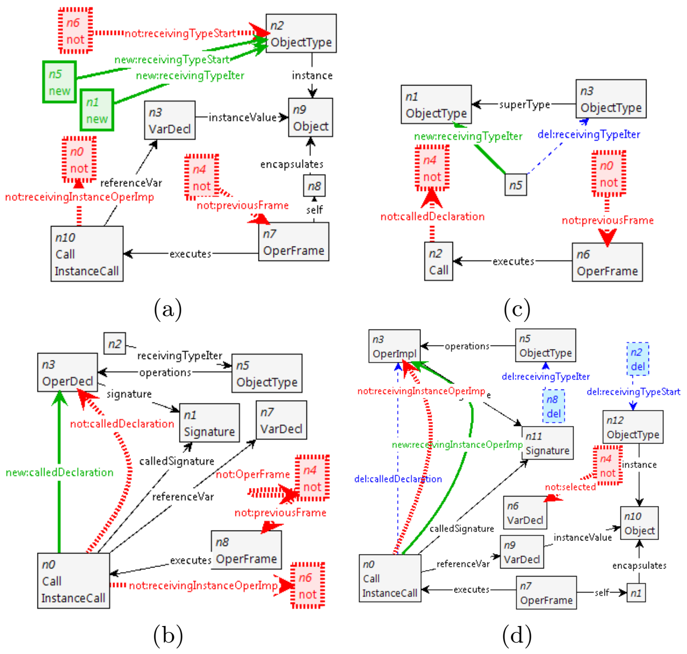


Fig. 22. Graph transformation rules for finding the newest implementation of the called operation: (a) Calculates the target reference type and marks it (b) finds the latest declaration of the operation (c) moves the mark up one level in the inheritance hierarchy, (d) checks whether the latest declaration implements the operation

2.1 The rule in Figure 22-(b) marks the latest declaration of the operation. This declaration is the object-type node with an operation declaration node that has the same signature as the signature called by the action. In the depicted transformation rule, the traversed object-type node is node $n5$ and the called signature is node $n1$. The rule matches when the traversed type has an operation declaration node ($n3$) 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 ($n0$) 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 22-(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.

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 22-(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*.

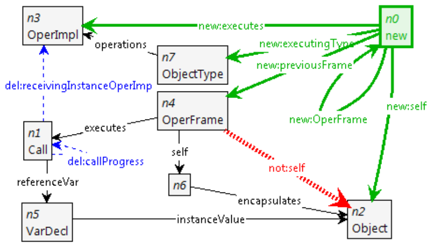


Fig. 23. Graph transformation rule that dispatches the operation after the object-type that implements the operation is discovered by the rules presented in Figure 22

After the object type that implements the operation is discovered, the operation can be dispatched as presented by the graph transformation rule in Figure 23. Here, the dispatching is done by creating a new operation frame node ($n0$), 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 receiving the call; thus, the rule adds the edge labeled *self* between the newly added frame ($n0$) and the object the reference variable holds ($n2$). 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.