Aspect-Oriented Design with Reusable Aspect Models

Jörg Kienzle[1], Wisam Al Abed[1], Franck Fleurey[2], Jean-Marc Jézéquel[3], and Jacques Klein[4]

[1] School of Computer Science, McGill University, Canada
[2] SINTEF IKT, Oslo, Norway
[3] INRIA, Centre Rennes - Bretagne Atlantique, Rennes, France
[4] CRP Gabriel Lippmann and University of Luxembourg, Luxembourg

Abstract. The idea behind *Aspect-Oriented Modeling* (AOM) is to apply aspect-oriented techniques to (software) models with the aim of modularizing crosscutting concerns. This can be done within different modeling notations, at different levels of abstraction, and at different moments during the software development process. This paper demonstrates the applicability of AOM during the software design phase by presenting parts of an aspect-oriented design of a crisis management system. The design solution proposed in this paper is based on the *Reusable Aspect Models (RAM)* approach, which allows a modeler to express the structure and behavior of a complex system using class, state and sequence diagrams encapsulated in several aspect models. The paper describes how the model of the "create mission" functionality of the server backend can be decomposed into 23 inter-dependent aspect models. The presentation of the design is followed by a discussion on the lessons learned from the case study. Next, RAM is compared to 8 other AOM approaches according to 6 criteria: language, concern composition, asymmetric and symmetric composition, maturity, and tool support. To conclude the paper, a discussion section points out the features of RAM that specifically support reuse.

1 Introduction

The idea behind *Aspect-Oriented Modeling* (AOM) is to apply aspect-oriented techniques to (software) models with the aim of modularizing crosscutting concerns. This can be done within different modeling notations, at different levels of abstraction, and at different moments during the software development process.

In [1,2,3] we have proposed *Reusable Aspect Models* (RAM), an aspect-oriented multi-view modeling approach that 1) integrates existing class diagram, sequence diagram and state diagram AOM techniques into one coherent approach; 2) packages aspect models for easy and flexible reuse; 3) supports the creation of complex aspect dependency chains; 4) performs elaborate consistency checks to verify correct aspect composition and reuse; 5) defines a detailed weaving algorithm that resolves aspect dependencies to generate independent aspect models and ultimately the final application model.

So far we have applied RAM only to one big case study: AspectOptima [4,5,6], a product line of transaction middleware implementations consisting of 17 aspects. From previous work we had acquired in-depth knowledge in object-oriented implementations of transaction middleware, and as a result the identification of the crosscutting concerns in transaction middleware was therefore pretty straightforward. Nevertheless, this first experience has provided us with many insights on AOM, and we evolved RAM to be able to deal with complex aspect dependencies and interactions in a way that does not compromise the reusability of the individual aspect models.

In [7] the authors propose a common case study, a *crisis management system* (CMS), to evaluate the strength and weaknesses of different AOM approaches. In this paper we present how we applied RAM to design parts of the functionality of the crisis management system backend according to the requirements listed in [7]. The design solution shown here is useful for software developers who want to learn about aspect-oriented design in general, as well as for developers who want to understand the details of the RAM approach. Please note that the complete models of all aspects involved in this case study can be downloaded from [8].

The outline of the paper is as follows. Section 2 presents a set of selected models of our design of the CMS backend, while at the same time explaining the main concepts of RAM. The design is split into resource management, communication, workflow, logistics, and "base design", each presented in a separate subsection. Section 3 discusses the insights we gained when applying RAM to the CMS. Section 4 presents an in-depth comparison of RAM to 8 other AOM approaches based on 6 criteria. Section 5 comments on how RAM facilitates aspect model reuse, and the last section draws some conclusions.

2 Crisis Management System Design

In this section we present the aspect-oriented design of the backend of the car crash crisis management system (CCCMS) using the aspect-oriented modeling approach *Reusable Aspect Models* (RAM) [1,2,3]. Just like in the original document that describes the CCCMS, we are focusing on the design of the *create mission* functionality that is executed at the CCCMS server backend.

The create mission functionality provided by the backend is triggered in the following context: for every crisis, at least one super observer, an expert in car crashes, is assigned to the scene to observe the emergency situation and identify the tasks necessary to cope with the crisis. These tasks are called *missions*. In [7], the super observer's activities are described in use case 6. The super observer sends the mission requests to the backend (use case 6, step 4), which has to allocate suitable resources (humans, vehicles, etc...) to fulfill the mission (use case 1, step 4). Once appropriate resources have been identified, the involved workers have to be contacted with the mission assignment (use case 3, step 1). The object-oriented design of the create mission functionality is presented in [7] in Fig. 8.

In RAM, *any concern* or *functionality* that is *reusable* is modeled in an *aspect model*. This is different from asymmetric aspect techniques, which usually only encapsulate *crosscutting* concerns within aspects. The RAM way of looking at this is that even if an aspect is only used once in the *same* application, it is (or can be) reused again in other applications. In this case, the structure and behavior models of a reusable aspect cut across the models of the application(s) in which the aspect is reused.

The philosophy in RAM is to decompose aspects that provide high-level functionality into aspects that provide lower-level functionality. For this case study, we decided to split the create mission functionality into 4 separate reusable concerns at a high level of abstraction: *resource management*, *communication, workflow management* and *logistics*, and two CCCMS-specific concerns: *initialization* and *create mission*. Subsection 2.1 presents the aspect models involved in the resource management concern in detail and at the same time explains the basic concepts of RAM. The communication, workflow management and logistics concerns are presented in subsections 2.2, 2.3, 2.4 with less details for space reasons. Finally, subsection 2.5 shows how the initialization and create mission concerns reuse the previous aspects to achieve the desired application functionality, and subsection 2.6 presents a design summary and comments on possible improvements. Please note that the complete models of all aspects involved in this case study can be downloaded from [8].

2.1 Resource Management

Resource management is one of the key concerns of the CCCMS. At all time, the system should be aware of the availability of resources – human resources such as first aid workers, drivers, doctors, etc. as well as vehicles, rescue equipment, etc. The system should also keep track of other relevant information, such as the condition and location of the resources in order to allow the most efficient and effective deployment of the resources in case of a crisis.

At a high level, resource management can be split into *resource search*, i.e. finding an available resource that exhibits a desired capability, and *resource allocation*, i.e., allocating a set of resources to a task. The design of resource allocation is explained first, since it is conceptually simple. At the same time the notation of the *Reusable Aspect Models* modeling approach is introduced.

2.1.1 Allocating Resources

We are going to present the three aspects that make up the design of the resource allocation aspect in a bottom-up way: first the lower-level aspects *ZeroToManyAssociation* and *Allocatable*, and then *ResourceAllocation* itself.

ZeroToManyAssociation

In our *Reusable Aspect Models* (RAM) approach, an aspect model is a special UML package that encapsulates all model elements related to the structure and/or behavior of a concern. The current version of RAM [2] supports aspect models that use class diagrams, state diagrams and sequence diagrams.

Fig. 1 presents the aspect model *ZeroToManyAssociation* that shows the design of a concern that occurs very frequently in applications, and that is also used in resource allocation: an object of some class *A* needs to be associated with

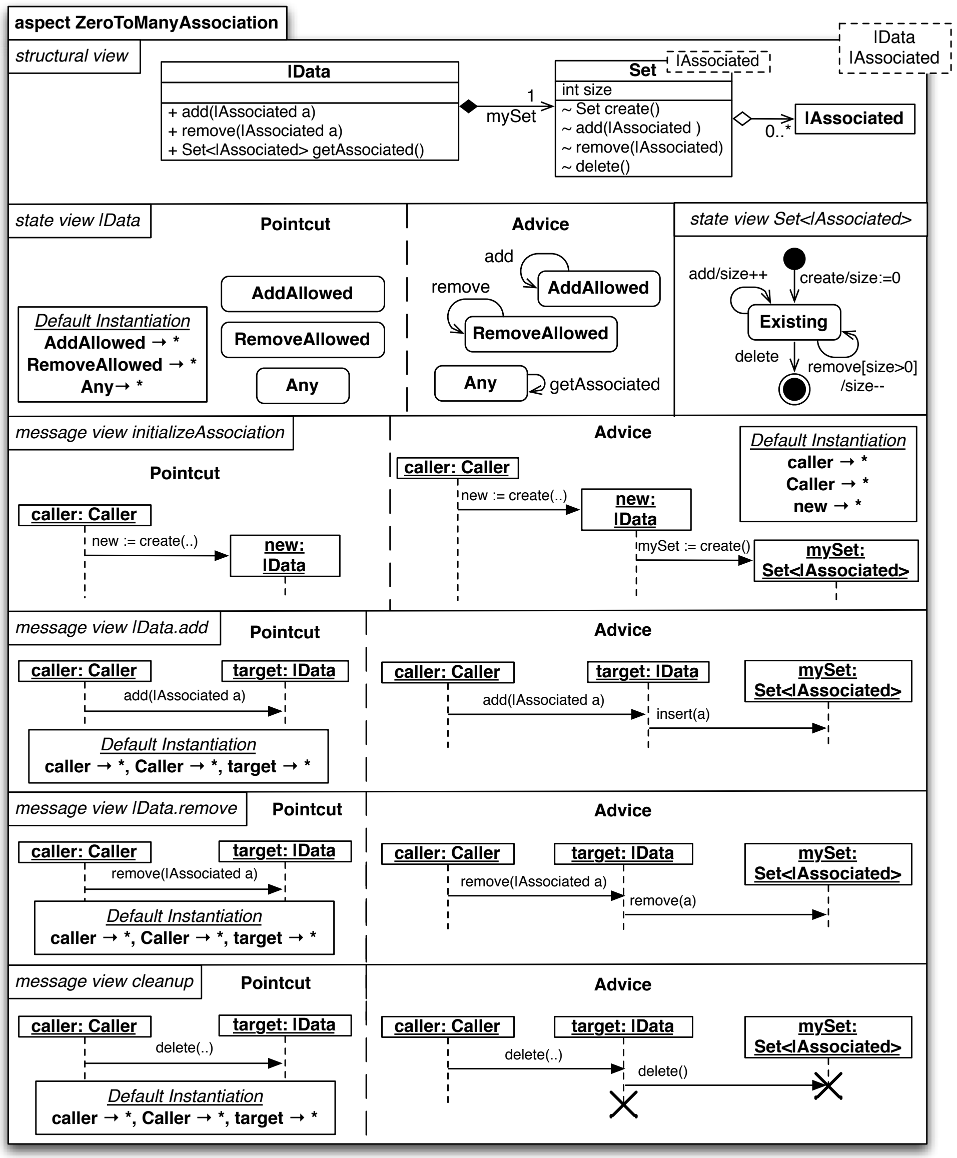


Fig. 1. The *ZeroToManyAssociation* Aspect

many objects of another class B. While in UML this situation can be shown with a standard association that has the multiplicity 0..*, it is usually implemented during the detailed design phase using an intermediate `set` (or `list` or `array`) object contained in the object of class A. To associate an object b with an object a, b is inserted into the set contained in a.

Structural View

The class diagram representing the structure of the design of *ZeroToManyAssociation* is presented in the *structural view* compartment of Fig. 1. It defines three classes: `|Data`, `|Associated` and the template class `Set`.

The `Set` class implements a *set* abstraction: it provides a constructor and destructor, as well as operations to insert elements into and remove elements from the set. It is parametrized with the `|Associated` class, thus creating a "Set of |Associated". Many object-oriented programming language libraries provide such classes, e.g. the generic `Set` class in Java, or the `set` class in the C++ standard template library.

`|Data` and `|Associated` are *partial classes*. A partial class needs to be completed before it can be used in an application. Partial classes, for instance, do not define constructors and destructors, and hence it would be impossible to create instances of the class. All partial classes of an aspect are therefore exported as *mandatory instantiation parameters* of the aspect, and shown as UML template parameters on the top right corner of the aspect package. In order to use the aspect and weave it with a target model, the mandatory instantiation parameters must be mapped to model elements from the target model.

The *public interface* of a RAM aspect is comprised of all the public operations declared by classes inside the aspect. In UML, the public operations are marked with a +. In the *ZeroToManyAssociation* example, only the operations `add`, `remove` and `getAssociated` provided by `|Data` are publicly accessible. The operations of the `Set` class are part of the *intra-aspect interface* of *ZeroToManyAssociation*, i.e. they can only be called from other objects that are part of the aspect. The intra-aspect operations are tagged in RAM using the UML package modifier ~.

Message Views

To provide the functionality related to a concern, the model elements within an aspect model must collaborate at run-time. In RAM, collaboration between objects is shown in the message view compartments using sequence diagrams. A RAM aspect must specify a message view for each public operation that involves message exchanges between objects. If a public operation does not have a corresponding message view, it is assumed that it only modifies or reads the state of the object[5].

The message view *initializeAssociation*, shown in the first message view compartment of the aspect in Fig. 1, shows that whenever a constructor is invoked on an object of the class `|Data` (see *pointcut* sequence diagram), then the con-

[5] This is, for instance, the case for the method `getAssociated`, which simply returns the set `mySet` to the caller

structor also creates an instance of the class `Set` and stores a reference to it in `mySet` (see *advice* sequence diagram). This guarantees that there will never be a `|Data` object without a `Set` object that belongs to it. The *cleanup* message view makes sure that the set is deallocated when the `|Data` object disappears. The message view *add* describes that adding an object of the class `|Associated` is done by inserting a reference to it into the set referenced by `mySet`. Removing objects from the association, described in the message view *remove*, follows the same pattern.

State Views

RAM also allows the modeler to show how the state of an object dictates the messages it accepts in *state views* with the help of state diagrams. For each class in the structural view that defines operations, a corresponding state view has to be specified. The state diagram must contain at least one transition for each operation that the class defines.

In Fig. 1, for example, the *Set<Associated>* state view describes the protocol of the `Set` class. It specifies that after being created, an instance of `Set` accepts calls to the `add` operation, and, if not empty, calls to `remove`, until the instance is destroyed. The *|Data state view* looks different: just like the message views, it has a pointcut state diagram and an advice state diagram. The reason for this is that `|Data` is a *partial class*. It is impossible to specify a complete state diagram, with initial state and end state, for a partial class. It is possible, however, to define states that are relevant with respect to the operations that the partial class offers. The `|Data` class, for example, has 3 states that are important: `AddAllowed`, `RemoveAllowed` and `Any`. They represent the states in which a `|Data` instance accepts calls to `add`, `remove`, and `getAssociated`, respectively.

Instantiation

In order to use the *ZeroToManyAssociation* aspect in a target model, it has to be instantiated by mapping the mandatory instantiation parameters of the aspect to model elements in a target model. In our example, the mandatory instantiation parameters, shown as UML template parameters on the top right corner of the aspect package, are the classes `|Data` and `|Associated`. For instance, to associate a *capability* with zero to many *resources*, a modeler would write the following instantiation: `|Data` → `Resource`, `|Associated` → `Capability`. At run-time, a capability *c* can now be associated with a resource *r* by calling `c.add(r)`.

Allocatable

Another simple low-level aspect of resource management is *Allocatable,* shown in Fig. 2.

Notice how *Allocatable* provides the functionality of being able to tag an `|Allocatable` object as being allocated by calling `allocate`, free it again by calling `deallocate`, and query its state using `isAllocated`. The *Allocatable* aspect is really simple, in the sense that it contains only one class `|Allocatable`. The three public operations of `|Allocatable` form the interface of the aspect. None of the public operations of `|Allocatable` involve object interactions, and

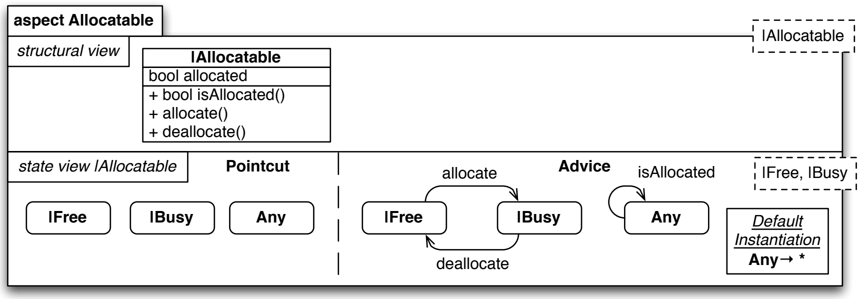


Fig. 2. The *Allocatable* Aspect

therefore this aspect does not contain any message views, nor do its classes need to worry about declaring an intra-aspect interface.

The state view of `|Allocatable` specifies that an instance of the class `|Allocatable` has at least two states, here named `|Free` and `|Busy`, and that the instance must be in the `|Free` state to be able to execute `allocate`, and in the `|Busy` state to execute `deallocate`.

ResourceAllocation

ResourceAllocation, shown in Fig. 3, is an example of a higher-level aspect that depends on the low-level functionality of both *Allocatable* and *ZeroToManyAssociation*. *ResourceAllocation* is in charge of allocating resources to a task, and tagging the resources as being allocated. To this aim, the structural view of *ResourceAllocation* defines the two partial classes `|Resource` and `|Task`. `|Task` provides the public operations `allocateResources`, `deallocateResources`, and `getResources`. The structural view also contains an instantiation directive for the aspects *ZeroToManyAssociation* and *Allocatable*. It maps `|Data` to `|Task` and `|Associated` to `|Resource`, thus reusing the functionality provided by *ZeroToManyAssociation* to associate each `|Task` object with a set of `|Resources`. *ResourceAllocation* also reuses *Allocatable* to be able to mark a resource as being allocated by mapping `|Allocatable` to `|Resource`.

A similar instantiation has to be done in the *|Task state view* in order to specify how the *AddAllowed* and *RemoveAllowed* states of *ZeroToManyAssociation* relate to the states *|NoAllocation* and *|Allocated*.

The *message view allocateResources* also reuses functionality provided by *Allocatable* and *ZeroToManyAssociation*. It demonstrates how behavior of lower-level aspects is reused. The sequence diagram describes that when requested to allocate a set of resources to a task, the task object loops through the set of resources, calling `allocate` (provided by *Allocatable*) for each of them and subsequently `add`ing it (provided by *ZeroToManyAssociation*) to the set of resources associated with the task. Although it would be possible to provide specific instantiation directives to reuse the message views of *ZeroToManyAssociation* and *Allocatable*, it is in this case also possible to use a shortcut. Since the message views in *ZeroToManyAssociation* and *Allocatable* both define default instantiation directives that map `caller`, `Caller` and `target` to any model element in

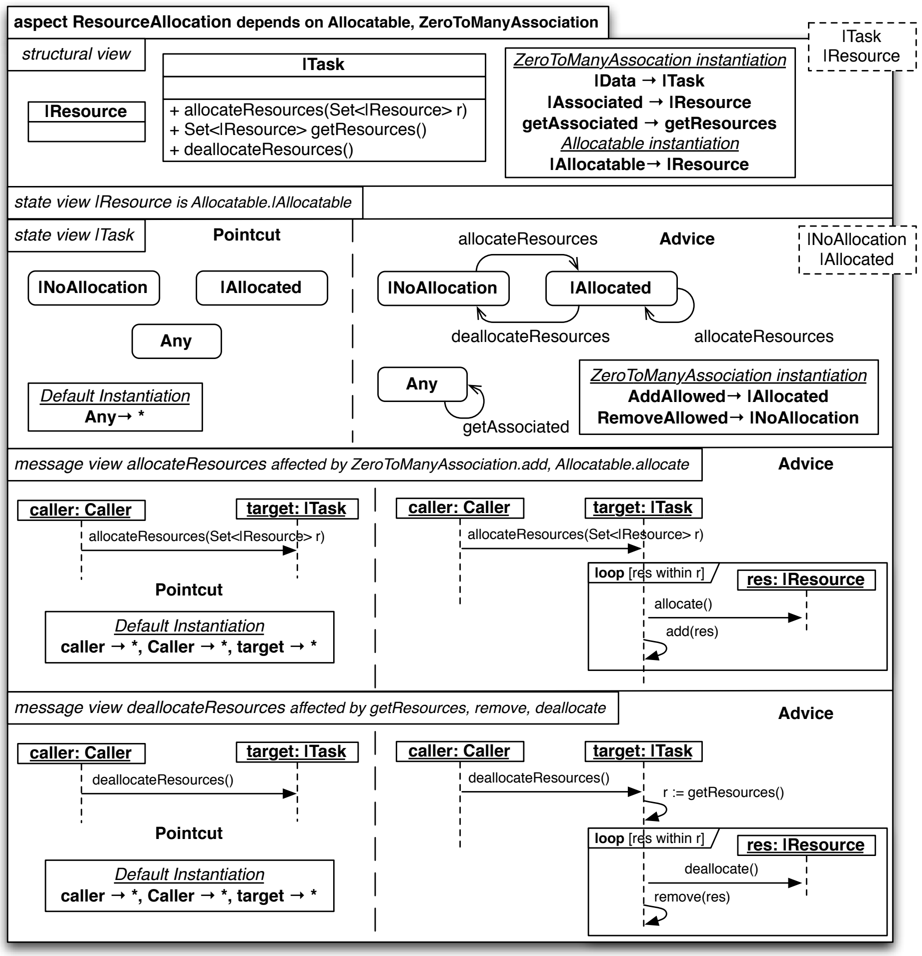


Fig. 3. The *ResourceAllocation* Aspect

the target model (see Fig. 1 and Fig. 2), it is enough to simply state that the message view *allocateResources* is *affected by* the message views *allocate* and *add*. The weaver will know how to combine the models correctly.

The state view `|Task` specifies that while it is possible to incrementally allocate resources to a task by calling `allocateResources` multiple times, a call to `deallocateResources` always frees all of the resources.

Weaving

The weaver in RAM supports aspect hierarchies of arbitrary depth. Given an aspect A that depends on lower-level aspects B and C, the weaver must first create an independent model of A before A can be woven into a base application model or be reused in a higher-level aspect. The independent model of A is an aspect model that contains all the structural entities, states and message exchanges defined in the aspects A depends on, i.e., in B and C. Weaving is performed in pairs. If no specific weave order is specified for B and C, the weaver chooses one randomly, e.g. first weaving B with A, and then C with the result of the previous weaving. Since B and C can themselves depend on other aspects, our weaving algorithm is recursive in nature, processing the dependency graph in depth-first order.

The dependency graph of *ResourceAllocation* is shown in Fig. 4. In order to create an independent aspect model of *ResourceAllocation*, the weaver first weaves the aspect *Allocatable* into *ResourceAllocation* in order to get a model of *ResourceAllocation* that is independent of *Allocatable.* Second, *ZeroToManyAssociation* is woven into the resulting model to finally obtain a model of *ResourceAllocation* that is independent of both *Allocatable* and *ZeroToManyAssociation.*

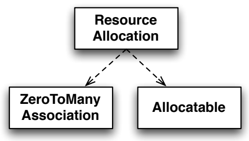


Fig. 4. Resource Allocation Dependencies

Fig. 5 shows the independent model of *ResourceAllocation* after weaving. Note how the structural views have been merged, and how the advice of the */Data state view* added transitions to the advice of the */Task state view* at the states identified by the pointcut of the */Data state view.*

Encapsulation and Information Hiding

In order to make information hiding possible in aspect hierarchies, the RAM aspect weaver *automatically removes the public interface* of the model elements of an aspect when it is woven into a target model by changing the visibility of the model elements from *public* to *package.* This effectively "moves" the operations from the public interface of the aspect to the intra-aspect interface.

The aspect interface of the independent *ResourceAllocation* aspect, just like the dependent one, therefore contains 3 public operations. The public operations of the lower-level aspects had their visibility changed from public to package as they were copied into the woven model. As a result, model elements in higher-level aspects that instantiate *ResourceAllocation* cannot directly use the functionality provided by *Allocatable* and *ZeroToManyAssociation.* Consequently, information hiding principles are maintained even in the woven model and exposure of internal design decisions of *ResourceAllocation* are prevented.

Some times, however, a modeler may want to "re-expose" lower-level operations at the higher level. This situation occurs when an operation of the lower-level aspect directly implements a functionality that the higher-level aspect needs to provide. In the *ResourceAllocation* aspect, for example, it should be possible to query all the resources that are allocated to a task. This functionality is already provided by the operation `getAssociated` in *ZeroToManyAssociation.*

In RAM, the modeler can explicitly "re-expose" lower-level operations by first declaring the re-exposed operation in the appropriate class. The same operation name can be chosen, but a different name can also be used if it better reflects the semantics of the operation in the context of the high-level aspect. Second, when instantiating the lower-level aspect in the high-level aspect, the lower-level operation is simply mapped to the high-level operation. Concretely, if a modeler wishes to "re-expose" `getAssociated` in *ResourceAllocation*, he can add

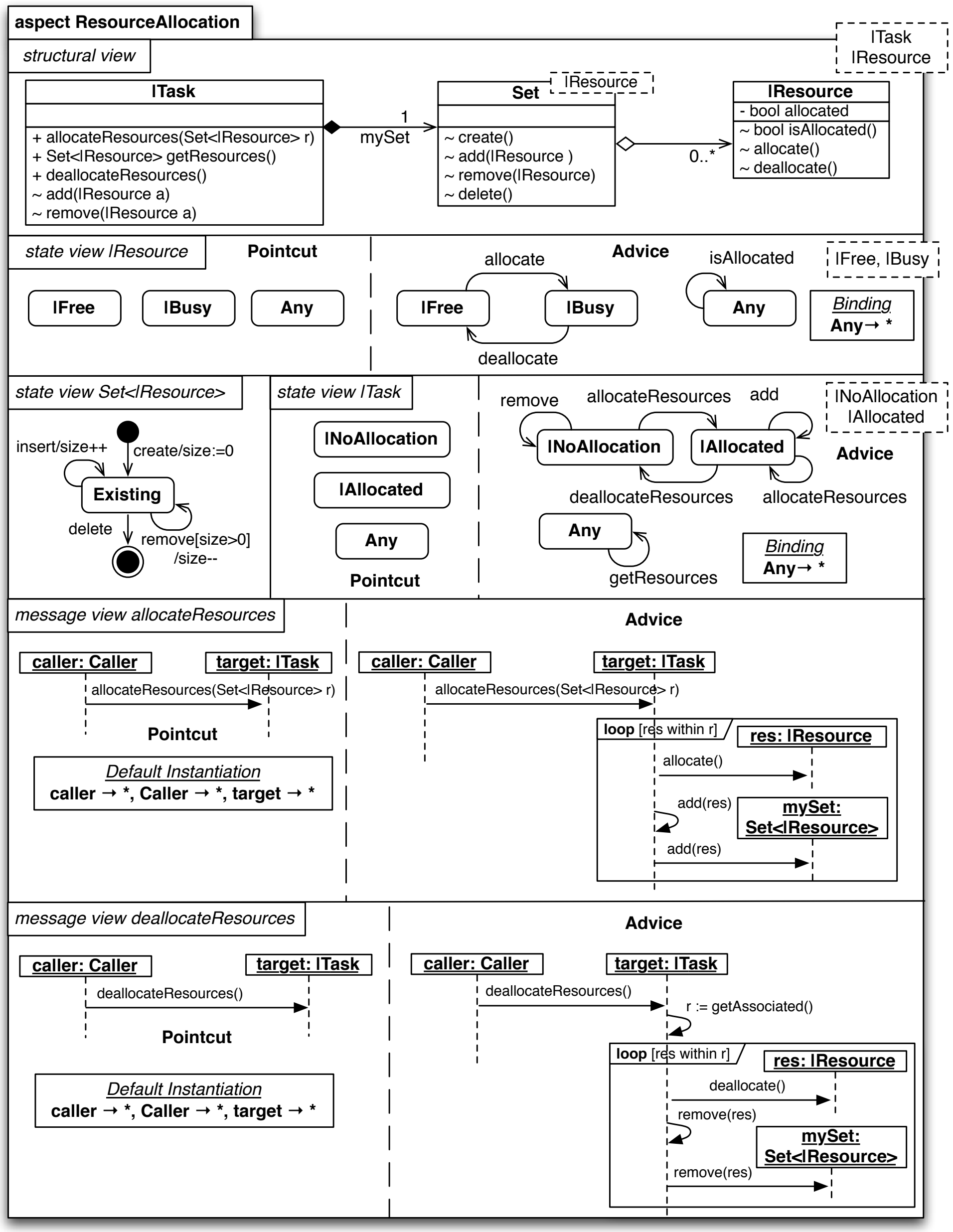


Fig. 5. The Independent *ResourceAllocation* Aspect

a public operation `getResources` in the `|Task` class and map getAssociated to getResources in the instantiation directives as shown in Fig. 3. This tells the weaver to rename the `getAssociated` operation during the weaving process and to not change its visibility modifier. For more details on the weaving algorithm, the reader is referred to [2]. For details on automated encapsulation, the reader is referred to [3].

2.1.2 Finding Resources

In the context of the CCCMS, the resource search concern encapsulates the functionality of finding the most appropriate set of resources that are available and that have the required capabilities to carry out a mission.

Solving the general problem of finding an optimal set of resources that fulfill a given criteria is very hard, and a research area in itself. Since resource search is not the main focus of this paper, we make an assumption that simplifies the problem considerably: a resource has exactly one capability.

With this assumption, fulfilling a request for n resources with capability c can be done simply by finding all available resources that have capability c and choosing n of them. This is the essence of the functionality of the simple *ResourceSearch* aspect shown in Fig. 6.

Resource Search

The structural view of *ResourceSearch* defines three classes: `|Capability`, `|Resource` and `Request`. `|Resource` is a partial class representing resources, `|Capability` is a partial class that represents the expertise, quality or function a resource can have or is able to perform. The specified association between the two partial classes makes sure that every resource has exactly one capability. In order to keep track of all the resources that have a certain capability, *ZeroToManyAssociation* is instantiated.

The *Request* class is central to the *ResourceSearch* aspect. To perform a search, a request must be created, and the desired capabilities and number added to it. The `find` operation performs the actual search for resources, and returns a set of available resources that have the desired capabilities. Internally, the `Request` class is designed using the *Map* aspect. The *Map* aspect is again a concern that occurs very frequently in applications. It basically provides the functionality of a hash table, mapping *key* objects to *value* objects. It therefore represents a design implementing a qualified association in UML. The internal design of the aspect is similar to the *ZeroToManyAssociation* aspect, and hence not shown here for space reasons. The interested reader can look at the model by downloading the set of all CCCMS models from [8].

The *initiateAssociation* and *cleanup* message views make sure that, whenever a resource is created with a capability *c*, the resource is also added to or removed from the set of resources associated with *c*. Finally, the most interesting message view is *find*. The request object first starts by creating a set of resources to store the result of the search. Then it obtains the set of capabilities requested using the `getCapabilities` functionality, which is actually the `getKeys` functionality provided by *Map*. For each requested capability, it queries the number

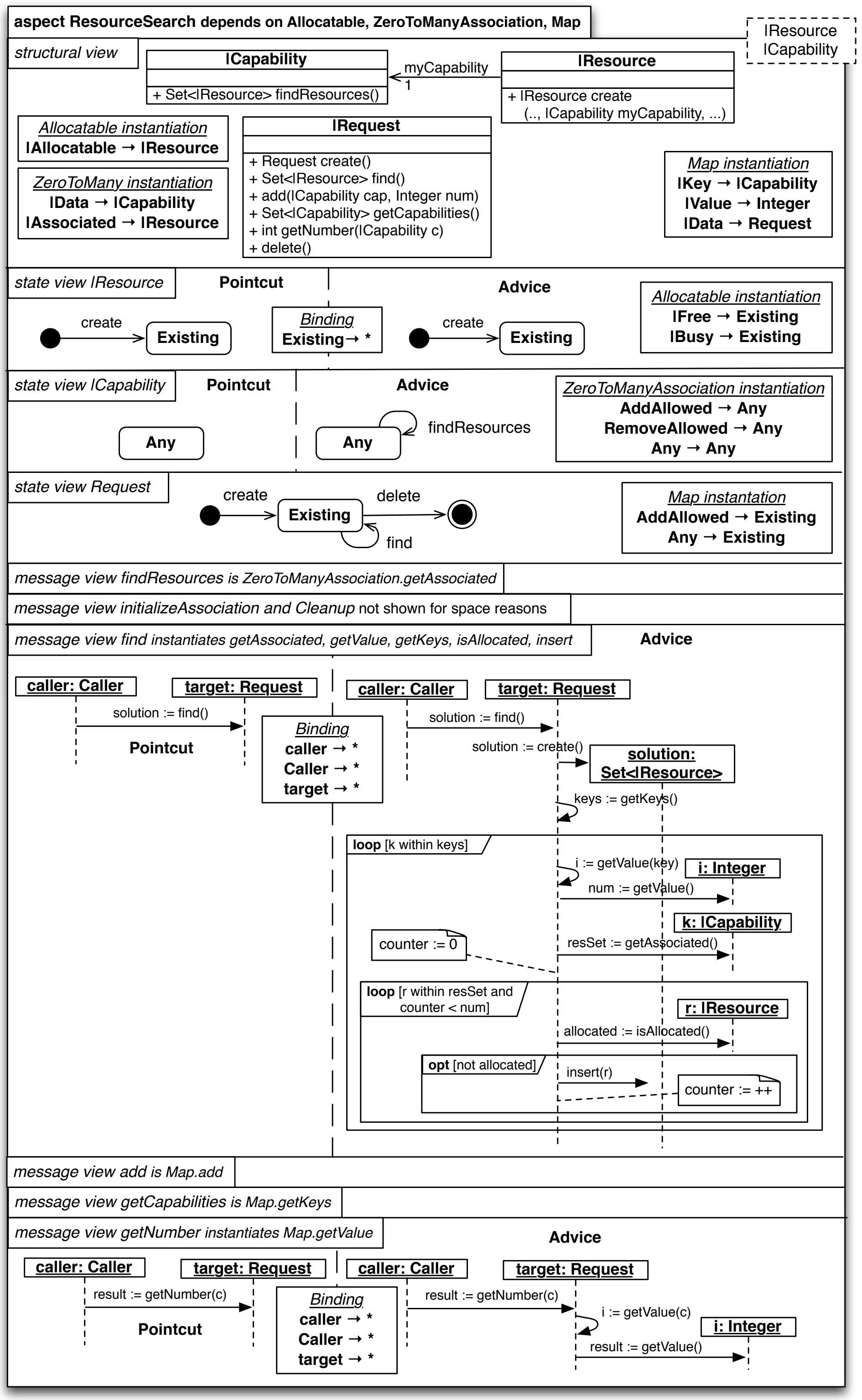


Fig. 6. The *ResourceSearch* Aspect

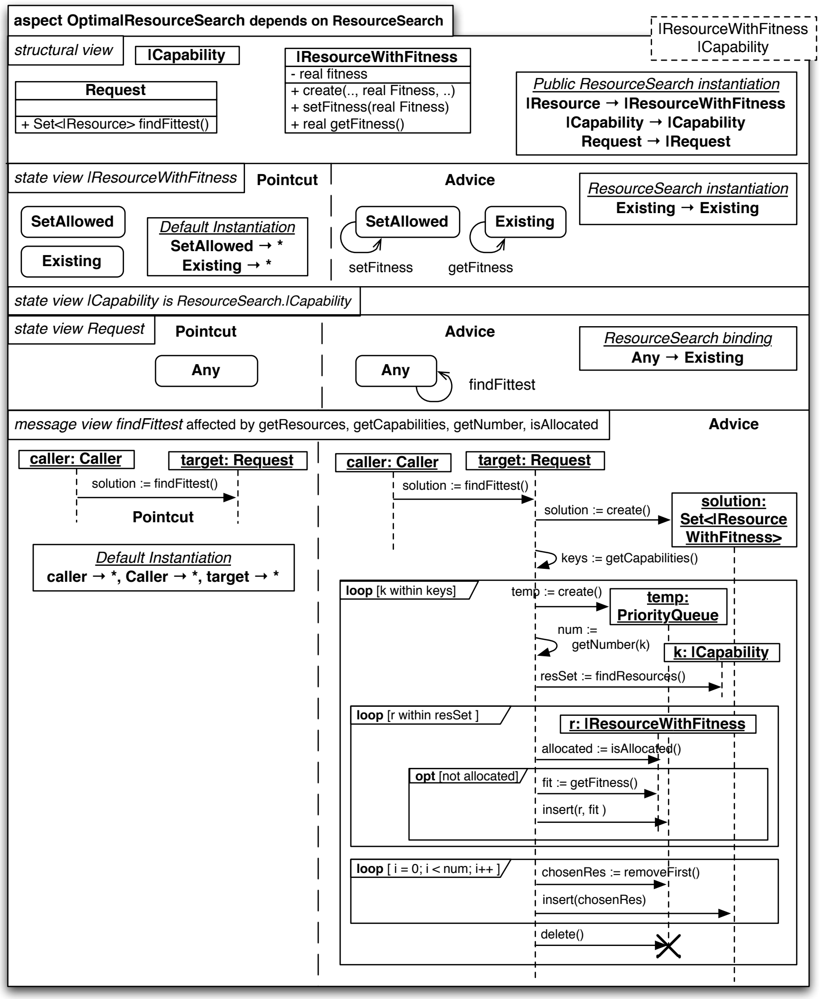

aspect OptimalResourceSearch depends on ResourceSearch
IResourceWithFitness
ICapability
structural view
ICapability
IResourceWithFitness
- real fitness
+ create(.., real Fitness, ..)
+ setFitness(real Fitness)
+ real getFitness()
Request
+ Set<IResource> findFittest()
Public ResourceSearch instantiation
IResource → IResourceWithFitness
ICapability → ICapability
Request → IRequest
state view IResourceWithFitness
Pointcut
Advice
ResourceSearch instantiation
Existing → Existing
SetAllowed
Existing
Default Instantiation
SetAllowed → *
Existing → *
setFitness
getFitness
state view ICapability is ResourceSearch.ICapability
state view Request
Any
findFittest
ResourceSearch binding
Any → Existing
message view findFittest affected by getResources, getCapabilities, getNumber, isAllocated
caller: Caller
target: Request
solution := findFittest()
Default Instantiation
caller → *, Caller → *, target → *
solution := create()
solution: Set<IResource WithFitness>
keys := getCapabilities()
loop [k within keys]
temp := create()
temp: PriorityQueue
num := getNumber(k)
k: ICapability
resSet := findResources()
loop [r within resSet]
r: IResourceWithFitness
allocated := isAllocated()
opt [not allocated]
fit := getFitness()
insert(r, fit)
loop [i = 0; i < num; i++]
chosenRes := removeFirst()
insert(chosenRes)
delete()

Fig. 7. The *OptimalResourceSearch* Aspect

of requested resources of that capability by calling the `getNumber` functionality, which uses the `getValue` functionality provided by *Map*. Then, a set of resources having the requested capability is obtained using `findResources`, which is actually `getAssociated` provided by *ZeroToManyAssociation*. For each resource in this set, `isAllocated` provided by *Allocatable* is called to check if the resource is still available, and if it is, the resource is inserted into the result set. When enough resources of the current capability have been found, the next capability of the request is looked at. This continues until the entire request has been handled.

Optimal Resource Search
The *OptimalResourceSearch* aspect, shown in Fig. 7, illustrates a slightly more

advanced searching algorithm. It associates with each resource a *fitness* value – a real number between 0 and 1– that stores how good a resource is in performing its associated capability. It provides the *findFittest* functionality, that fulfills a request by finding the most adequate available resources. As shown in the message view *findFittest* of Fig. 7, the available resources that have the required capability are first sorted according to their fitness value using a priority queue, and then the top-most resources are chosen to fulfill the request.

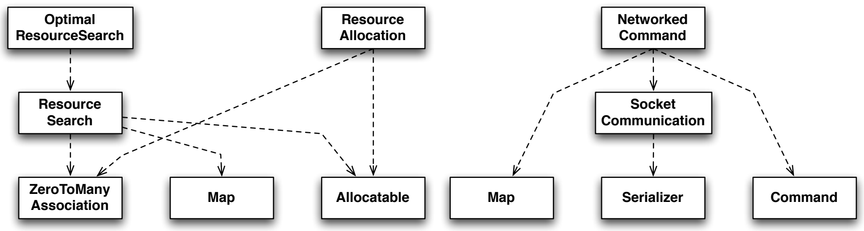


Fig. 8. Dependencies of the Resource Management and Communication Aspects

2.1.3 Dependency Summary

The dependencies between the aspects involved in the resource management design are shown in Fig. 8. It shows clearly that *ZeroToManyAssociation* and *Allocatable* are concerns that crosscut the design of resource management, since both *ResourceSearch* and *ResourceAllocation* depend on them.

2.2 Communication

The main functionality of the communication infrastructure is to allow remote resources, such as first aid workers or vehicles, to communicate with the CCCMS backend. The assumption here is that humans that need to communicate with the backend are carrying a laptop or a PDA (Personal Digital Assistant) that can establish a secure wireless connection (for instance using Virtual Private Network (VPN) technology) to access the CCCMS network. The super observer is an example of a human resource that uses his PDA to communicate new missions to the backend.

Fig. 8 shows a high-level overview of the dependencies among the aspects that handle communication within the CCCMS. The designs of the individual aspects are not shown here for space reasons. The ideas are briefly explained below, and the full models can be downloaded from [8].

The main idea of the design of the communication infrastructure is to send messages and data over the network in form of *commands*. The *Command* aspect implements the *command design pattern* [9], the essence of which is to encapsulate a method call in an instance of a `Command` object. The value of the parameters of the invocation are stored inside fields of the `Command` class. Every command class has an `execute` operation that triggers the execution of the actual command.

The *Serializer* aspect implements the *serializer design pattern* [10], which ensures that the state of any `Serializable` object can be flattened into a stream of bytes in order to be written to a `Backend`. Conversely, it also offers the functionality of recreating the `Serializable` object from a stream of bytes read from a `Backend`.

The *SockedCommunication* aspect implements socket-based communication on top of the *Serializer* aspect. Its design is heavily inspired by how Java does socket-based communication. It defines a `SocketServer` class that provides the functionality of listening for incoming connections using a server thread, and a `Socket` class that can be instantiated by a client to establish a connection with a server, or that is automatically instantiated by the server when a connection is accepted. A `Socket` has an associated `Receiver` and `Sender` object that provide the functionality to *receive* and *send* `|Sendable` objects, respectively.

Finally, the *NetworkedCommand* aspect ties the different aspects together. It defines a `|RemoteCommand` class, which is both a `|Sendable` (provided by *SocketCommunication*) and a `Command` (provided by *Command*). It also defines the `CommandChannel` class that provides the functionality to `send` a remote command to a host identified by a string. At the same time, creating a `CommandChannel` also instantiates a server socket and starts a thread in order to listen for incoming connections. When an incoming connection is established, another thread is instantiated in order to listen for incoming commands and execute them. This functionality is provided by the `CommandListener` class. Finally, in order to not have to establish new connections for each command, the `CommandChannel` object stores all existing connections in a hash table provided by the *Map* aspect.

2.3 Workflow Management

The main functionality of the reusable concern *workflow management* is to allow the modeler to define and later on execute workflows, which control the sequence and conditions that coordinate activities. In the CCCMS requirements document, for example, the use cases define the different activities that constitute the workflow for missions that can be carried out by CCCMS workers.

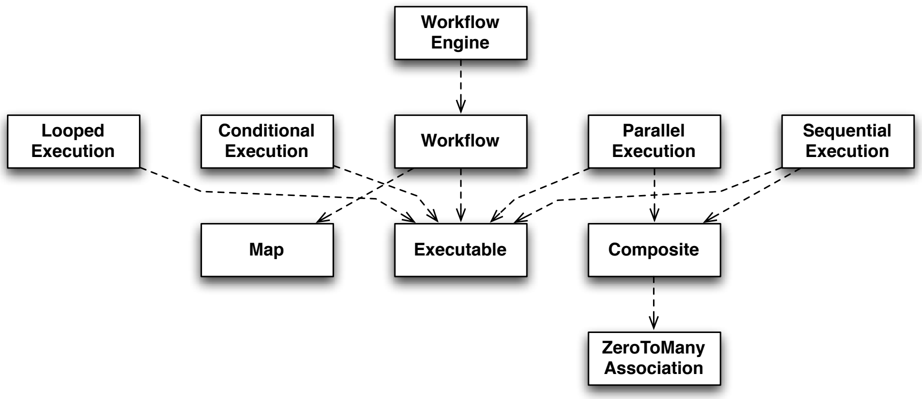


Fig. 9. Dependencies of the *Workflow Management* Aspects

Fig. 9 shows a summary of the dependencies among the aspects that provide workflows to the CCCMS. The central aspect of workflow management is the aspect *Executable,* which only defines a partial class `|Executable` and a partial class `|Context`. An `|Executable` represents an activity within a workflow. The execute operation triggers the execution of the activity. The class also contains an operation *waitForTermination* that suspends the calling thread until the execution of the activity is completed. The method *getOutcome* returns a boolean that communicates if an activity completed successfully or not.

The aspects *SequencialExecution*, *ParallelExecution*, *LoopedExecution*, and *ConditionalExecution* are control structures which allow the modeler to create complex control flows of activities. For instance, the aspect *SequencialExecution* contains an ordered set of `|Executable` instances and provides the functionality of executing them in sequence. It also uses the composite pattern **[9]** by instantiating the *Composite* aspect to pose itself as an `|Executable`*,* and hence inherits the operations of the latter. Another example of control structure is the aspect `ConditionalExecution`. This aspect implements a *"If Then Else"* block, where the `|Executable` in the *"then part"* or the *"else part"* is executed depending on a condition. The condition itself is also an `|Executable`, which is evaluated by executing it and then querying the outcome with the *getOutcome* operation.

The *Workflow* aspect implements basic workflows. This aspect contains a class `Workflow` which is associated with a root activity. This root activity can be any `|Executable`, and hence complex workflows can be built using the control structures presented above. The *Workflow* aspect also defines the `|Variable` class, which an activity can use to store results for following activities. The variables are all linked to the `|Context` in which a workflow executes. The class `|Context` uses the functionality of the *Map* aspect to store the variables indexed by their name.

Finally, the *WorkflowEngine* aspect provides the functionality of executing workflows. The class `WorkflowExecution` links an execution context with a workflow, and also implements the interface *Runnable.* The class `WorkflowEngine` implements a virtual machine which, when requested to do so, launches the execution of a workflow within a given context. It does this by instantiating a new thread, which starts by calling the run operation of `WorkflowExecution`, which in turn starts executing the root activity of the associated workflow.

2.4 Logistics

An orthogonal concern within the CCCMS is *logistics*: a crisis usually affects a certain physical area, and missions to address the crisis situation need to be executed at specific locations, or involve transporting victims, goods or other resources from one location to another. Treating the logistics concern of the CCCMS in its entirety is out of the scope of this paper. However, it is interesting to show how the logistics concern interacts with resource management in general, and resource search in particular.

Locatable

Fig. 10 shows the design of a simple aspect called *Locatable.* It allows a modeler

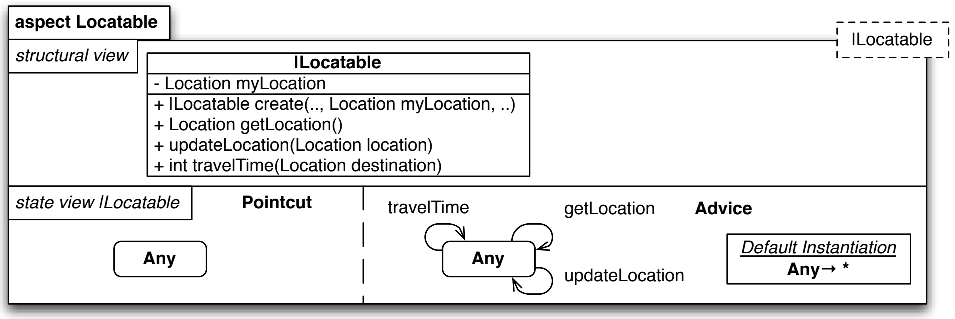


Fig. 10. The *Locatable* Aspect

to augment the state of any class with a physical location. It defines getter and setter operations that can be used to query or to update the location of the object, and a `travelTime` operation that calculates how long it would take the object to reach a certain destination location.

The logistics concern also includes a *Timing* aspect which allows the modeler to store a deadline in the state of an object. The RAM model of the *Timing* aspect is even simpler than *Locatable*, and therefore not shown here for space reasons.

Conflict Resolution Aspects

As mentioned before, the logistics concern has an impact on resource management. For instance, when a resource such as a vehicle is put under control of the CCCMS, its location has to be initialized, and continuously updated whenever the resource changes position. Searching for resources is also affected by logistics. In order to determine if a resource is fit to fulfill a certain mission, it not only has to be capable of performing the task, but it must also be able to reach the location of a time-constrained mission within the deadline.

This kind of situation, i.e., a situation in which the simultaneous application of two or more aspects requires one or several of the aspect's state and behavior to be altered in order to provide a semantically correct functionality, has been called *aspect interference* or *aspect conflicts [11]*.

In RAM, a modeler that detected a conflict between two aspect models A and B can express the adaptations that have to be made to both aspects in order to obtain a semantically correct woven model. In order to keep aspects A and B reusable and independent, the structural and behavioral modifications that are necessary to cope with the aspect conflicts are specified in a separate *conflict resolution aspect model*. The conflict resolution aspect model, of course, depends on the aspects whose conflicts it resolves. The conflicting aspect models, however, do not have to be changed, and hence remain independent from each other. They are therefore still individually reusable.

A conflict resolution aspect model is different from a standard aspect model, because it defines a set of *modification views* instead of standard views. Each modification view contains a *conflict criteria condition* that specifies under which condition the conflict occurs. If the condition is verified, the adaptation defined in the remainder of the view is woven with the structure and behavior of one of the conflicting aspects.

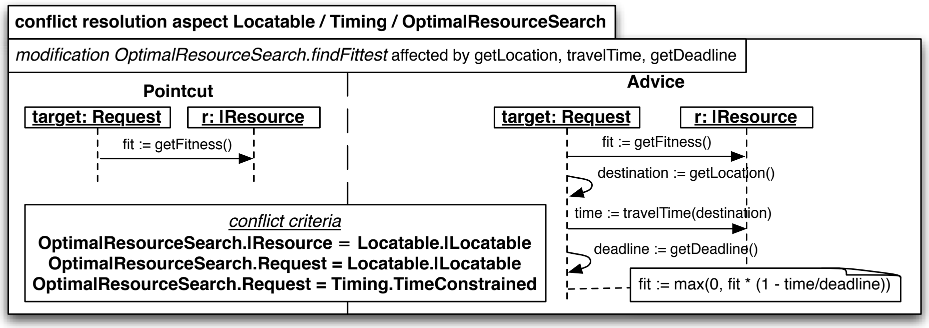

conflict resolution aspect Locatable / Timing / OptimalResourceSearch
modification OptimalResourceSearch.findFittest affected by getLocation, travelTime, getDeadline
Pointcut
target: Request
r: IResource
fit := getFitness()
conflict criteria
OptimalResourceSearch.IResource = Locatable.ILocatable
OptimalResourceSearch.Request = Locatable.ILocatable
OptimalResourceSearch.Request = Timing.TimeConstrained
Advice
target: Request
r: IResource
fit := getFitness()
destination := getLocation()
time := travelTime(destination)
deadline := getDeadline()
fit := max(0, fit * (1 - time/deadline))

Fig. 11. *Locatable / Timing / OptimalSearch* Conflict Resolution

Fig. 11 illustrates this idea. It describes the modifications that have to be applied to the *findFittest* message view of the *OptimalResourceSearch* aspect (shown in Fig. 7) in order to deal with locations and deadlines. The modifications are only applied in the case where the resources are locatable (as defined by the *Locatable* aspect), the request is locatable (because the physical location of the mission is known), and there is a deadline associated with the request (as defined by the `TimeConstrained` class specified in the *Timing* aspect). This condition is specified in the *conflict criteria box* of the modification view.

If the RAM weaver detects that the conflict condition is verified, then the modifications are automatically applied to the *findFittest* message view, i.e. the specified pointcut is matched within the advice sequence diagram of *findFittest*, and all occurrences of the pointcut (in our case there is only 1 match), are replaced by the advice specified in the conflict resolution model. In our case this amounts to decreasing the fitness value returned by the `getFitness` operation according to the time it takes the resource to reach the mission location.

2.5 Base Design

All the aspects described in sections 2.1 to 2.4 are general in the sense that they do not specify any classes or operations that only apply to the CCCMS. Rather, they attempt to capture the structure and behavior of resource management, communication, workflow and logistics in a reusable way, independent of each other, and independent of a specific application or context.

In order to use these aspects to build the design model of the CCCMS, the general aspects have to be carefully mapped to CCCMS-specific model elements. Traditionally, the models that contain the application-specific elements have been called the *base*.

The base model of the CCCMS is huge, and hence we again use the decomposition possibilities offered by RAM to modularize even application-specific concerns into separate aspect models. As a result, the only "special" property of

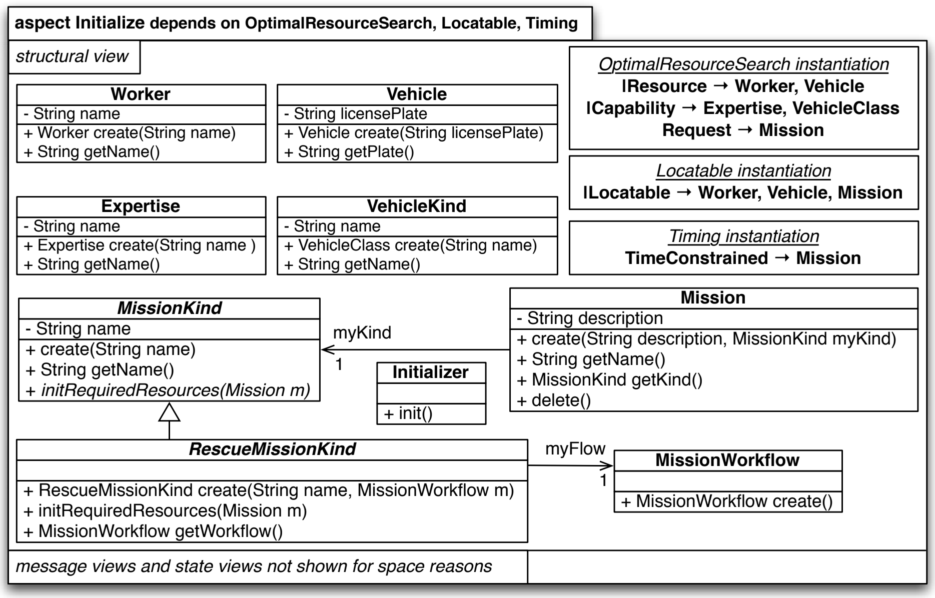

aspect Initialize depends on OptimalResourceSearch, Locatable, Timing
structural view
Worker
- String name
+ Worker create(String name)
+ String getName()
Vehicle
- String licensePlate
+ Vehicle create(String licensePlate)
+ String getPlate()
OptimalResourceSearch instantiation
IResource → Worker, Vehicle
ICapability → Expertise, VehicleClass
Request → Mission
Expertise
- String name
+ Expertise create(String name)
+ String getName()
VehicleKind
- String name
+ VehicleClass create(String name)
+ String getName()
Locatable instantiation
ILocatable → Worker, Vehicle, Mission
Timing instantiation
TimeConstrained → Mission
MissionKind
- String name
+ create(String name)
+ String getName()
+ initRequiredResources(Mission m)
myKind
1
Initializer
+ init()
Mission
- String description
+ create(String description, MissionKind myKind)
+ String getName()
+ MissionKind getKind()
+ delete()
RescueMissionKind
+ RescueMissionKind create(String name, MissionWorkflow m)
+ initRequiredResources(Mission m)
+ MissionWorkflow getWorkflow()
myFlow
1
MissionWorkflow
+ MissionWorkflow create()
message views and state views not shown for space reasons

Fig. 12. The *Initialize* Aspect

a base aspect model is the fact that all of the classes it defines are complete, and hence it does not declare any mandatory instantiation parameters.

As mentioned in the introduction, we are going to concentrate on the functionality of the CCCMS that is triggered when a super observer orders the execution of a new mission to help resolve the crisis. The design of this functionality is provided in the *CreateMission* aspect. In order for *CreateMission* to execute, however, the backend system has to be initialized: several objects need to be created (such as workers and vehicles), associations need to be established (such as between workers and their expertise), and workflows need to be instantiated. The *Initialize* aspect takes care of this.

Initialize

Fig. 12 shows the structural view of *Initialize.* At its heart, it defines the `Mission` and `MissionKind` classes. Every mission has an associated mission kind. `MissionKind` is actually an abstract class, and concrete classes must be defined for each kind of mission. Currently, only one concrete kind is defined: `RescueMissionKind`.

Initialize also defines the `Worker` and `Vehicle` classes, which are two examples of resources, and the `Expertise` and `VehicleKind` classes, which represent capabilities that resources can have. *Locatable* is instantiated to associate a location with workers, vehicles and missions. *Timing* is instantiated in order to associate deadlines with missions. *OptimalResourceSearch* is instantiated to establish a mapping between resources, i.e. workers and vehicles, and capabilities, i.e. expertise and vehicle kinds.

The state views and message views of the *Initialize* aspect are not shown for space reasons. The most important message view is the `init` operation of the `Initializer` class, which performs the actual initialization of all the objects in the system. It creates capabilities, e.g `firstAid`, `superObserver` or `ambulance`, and resources, e.g. `wisam`, who is a superobserver located initially at the crisis control center, `joerg`, a first aid worker located initially at the hospital, or `a1`, an ambulance stationed at the hospital. In addition, it creates a workflow engine, and workflows for each mission as specified in the use cases of the requirements document. For instance, a rescue mission workflow is created by instantiating *SequencialExecution*, and then adding in sequence the individual activities, which would also have to be defined in separate aspect models, as specified in use case UC7 [12].

In a real CCCMS system, initialization should of course not be "hard coded". Usually, information about resources such as workers and vehicles would be stored in a database, and during system start-up, the in-memory data structures would be initialized with data from the database. Even mission workflows could be initialized in such a way, or even dynamically created at run-time using a mission workflow editor. Database access and dynamic mission editing – yet more concerns of the CCCMS – are, however, out of the scope of this paper.

CreateMission
Fig. 13 shows the design of the *CreateMission* aspect. Its design links together the resource management, communication and workflow concerns.

The structural view of *CreateMission* defines a `CreateMissionCommand`, a `CreateMissionReceiver`, a `ReceiverQueue` and a `WaitForCreateMissionStep` class. The following describes how objects of these classes collaborate to achieve the create mission functionality.

On the backend, when a new crisis is created, a `CreateMissionReceiver` object is instantiated and the operation `initialize` is invoked. As shown in the *initialize* message view, this results in instantiating a `ReceiverQueue` object, which is a blocking queue of `Mission` objects thanks to the instantiation of the *BlockingQueue* aspect. The `CreateMissionReceiver` object stores the reference to this queue object in a hash table indexed by crisis number. This functionality is provided by the *Map* aspect. Finally, it creates a `CommandListener` object to listen for incoming commands.

When a super observer is dispatched to observe the crisis, the super observer mission workflow is instantiated and given to the workflow execution engine to execute. At some point, after the super observer reached the crisis location, the workflow execution engine executes the `performStep` operation of a `WaitForCreateMissionStep`. As described in the *performStep* message view, this results in calling the `take` operation of the blocking queue corresponding to the crisis that the super observer is handling. As a result, the workflow execution is suspended, effectively waiting for the super observer to send a create mission command.

When a super observer wants to create a mission, he uses his PDA to instantiate the `CreateMissionCommand` class with the appropriate parameters, and

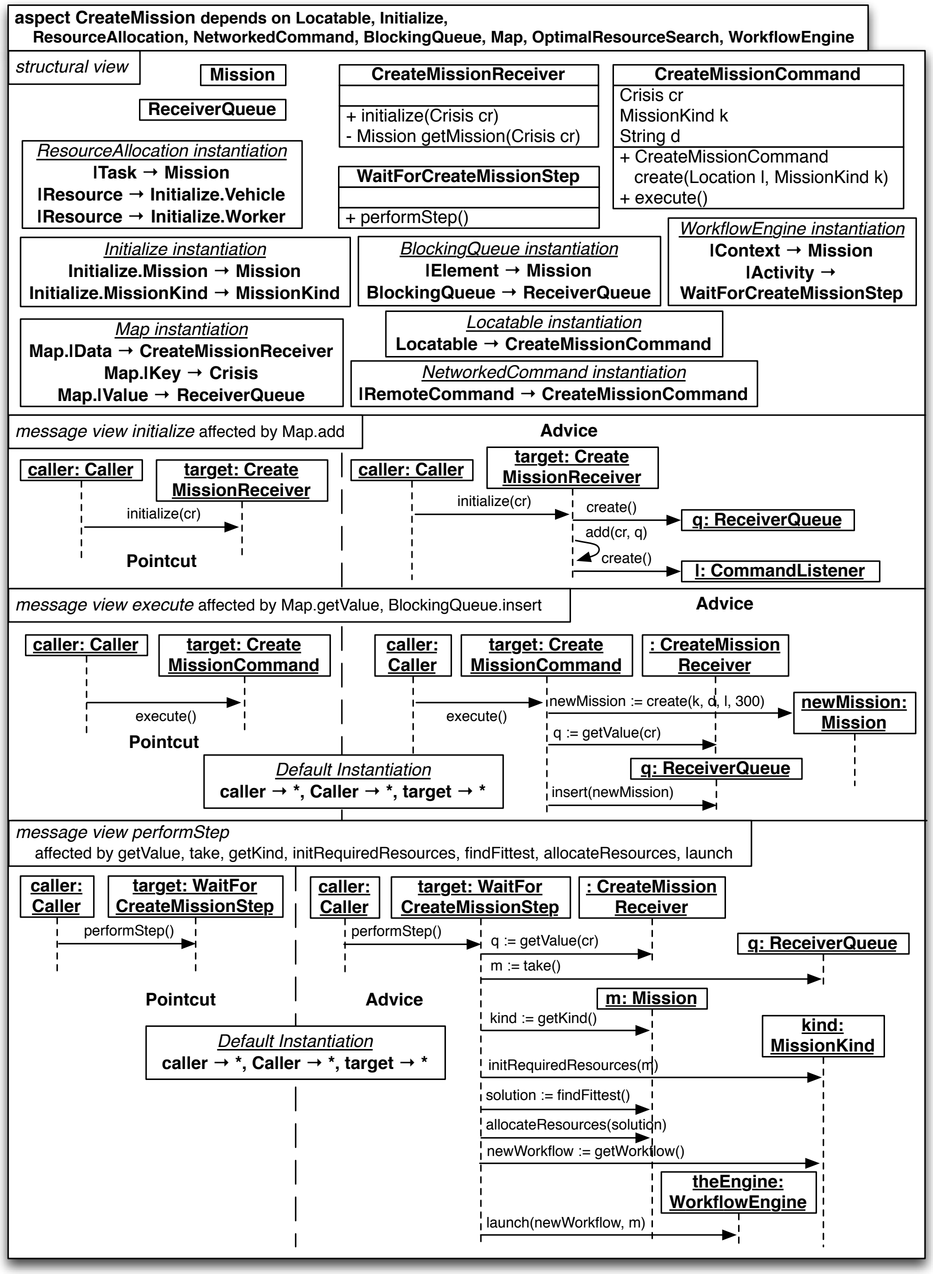

aspect CreateMission depends on Locatable, Initialize, ResourceAllocation, NetworkedCommand, BlockingQueue, Map, OptimalResourceSearch, WorkflowEngine
structural view
Mission
ReceiverQueue
CreateMissionReceiver
+ initialize(Crisis cr)
- Mission getMission(Crisis cr)
CreateMissionCommand
Crisis cr
MissionKind k
String d
+ CreateMissionCommand create(Location l, MissionKind k)
+ execute()
WaitForCreateMissionStep
+ performStep()
ResourceAllocation instantiation
ITask → Mission
IResource → Initialize.Vehicle
IResource → Initialize.Worker
Initialize instantiation
Initialize.Mission → Mission
Initialize.MissionKind → MissionKind
BlockingQueue instantiation
IElement → Mission
BlockingQueue → ReceiverQueue
WorkflowEngine instantiation
IContext → Mission
IActivity → WaitForCreateMissionStep
Map instantiation
Map.IData → CreateMissionReceiver
Map.IKey → Crisis
Map.IValue → ReceiverQueue
Locatable instantiation
Locatable → CreateMissionCommand
NetworkedCommand instantiation
IRemoteCommand → CreateMissionCommand
message view initialize affected by Map.add
caller: Caller
target: Create MissionReceiver
initialize(cr)
Pointcut
Advice
caller: Caller
target: Create MissionReceiver
initialize(cr)
create()
q: ReceiverQueue
add(cr, q)
create()
l: CommandListener
message view execute affected by Map.getValue, BlockingQueue.insert
caller: Caller
target: Create MissionCommand
execute()
Pointcut
Advice
caller: Caller
target: Create MissionCommand
: CreateMission Receiver
execute()
newMission := create(k, d, l, 300)
newMission: Mission
q := getValue(cr)
q: ReceiverQueue
insert(newMission)
Default Instantiation
caller → *, Caller → *, target → *
message view performStep
affected by getValue, take, getKind, initRequiredResources, findFittest, allocateResources, launch
caller: Caller
target: WaitFor CreateMissionStep
performStep()
Pointcut
Advice
caller: Caller
target: WaitFor CreateMissionStep
: CreateMission Receiver
performStep()
q := getValue(cr)
q: ReceiverQueue
m := take()
m: Mission
kind := getKind()
kind: MissionKind
initRequiredResources(m)
solution := findFittest()
allocateResources(solution)
newWorkflow := getWorkflow()
theEngine: WorkflowEngine
launch(newWorkflow, m)
Default Instantiation
caller → *, Caller → *, target → *

Fig. 13. The *CreateMission* Aspect

sends the command to the backend. This functionality is provided by *Networked-Command*. On the backend, the command listener receives the command, and invokes the `execute` operation on it. As shown in the *message view execute*, this results in the instantiation of a `Mission` object, which is subsequently inserted into the blocking queue of the corresponding crisis.

As a result, the workflow execution engine thread is awakened, and it continues the sequence of execution described in the *performStep* message view. It asks the `MissionKind` object associated with the retrieved mission to initialize the required resources for this mission. This operation, provided by the *Initialize* aspect, initializes the mission object, which is also a request, with the capability requests that describe the resources needed to complete the mission. Then, `performStep` invokes the `findFittest` operation provided by *OptimalResource-Search* on the mission object. The found set of resources is then allocated to the mission by using the functionality provided by *ResourceAllocation*. Finally, the workflow that needs to be executed in order to process the new mission is obtained from the `MissionKind` object, and passed to the workflow execution engine to be executed.

2.6 Design Summary and Comments

Fig. 14 summarizes the dependencies among all the aspects that are part of the CCCMS backend design of the create mission functionality. In total, there are 23 aspects. Only 2 aspects, i.e. *CreateMission* and *Initialize*, are application-specific in the sense that they contain model elements that are specific to crisis management systems. All other aspects are CCCMS independent, and could hence be reused in many other applications.

One of the central classes of the CCCMS that is part of many concerns is the `Mission` class. It not only stores the CCCMS-specific mission information, it is also a `Request` of *ResourceSearch*, a `Task` of *ResourceAllocation*, a `Locatable`, a `TimeConstrained`, and a `Context` of *Workflow*. The central message view of the CCCMS design is the `performStep` message view of the `WaitForCreateMissionStep` class of *CreateMission*. After receiving the information about the mission that is to be created from the super observer using *NetworkedCommand*, the best resources are determined using *OptimalResource-Search*, allocated using *ResourceAllocation*, and then the new mission workflow is launched using *WorkflowEngine*.

The woven class diagram that our weaver generates after weaving all dependent aspects into the structural view of *CreateMission* is shown in Fig. 15. The interested reader can compare this design class diagram with the one provided in the case study description document in Fig. 9[6] [7].

2.6.1 Design Improvements

The design presented in this paper is of course only an initial design, and

[6] The main reason why our generated class diagram has more classes than the one provided in the case study document is that ours shows the design of one-to-many associations and qualified associations using *Set*, *List* and *Map* classes.

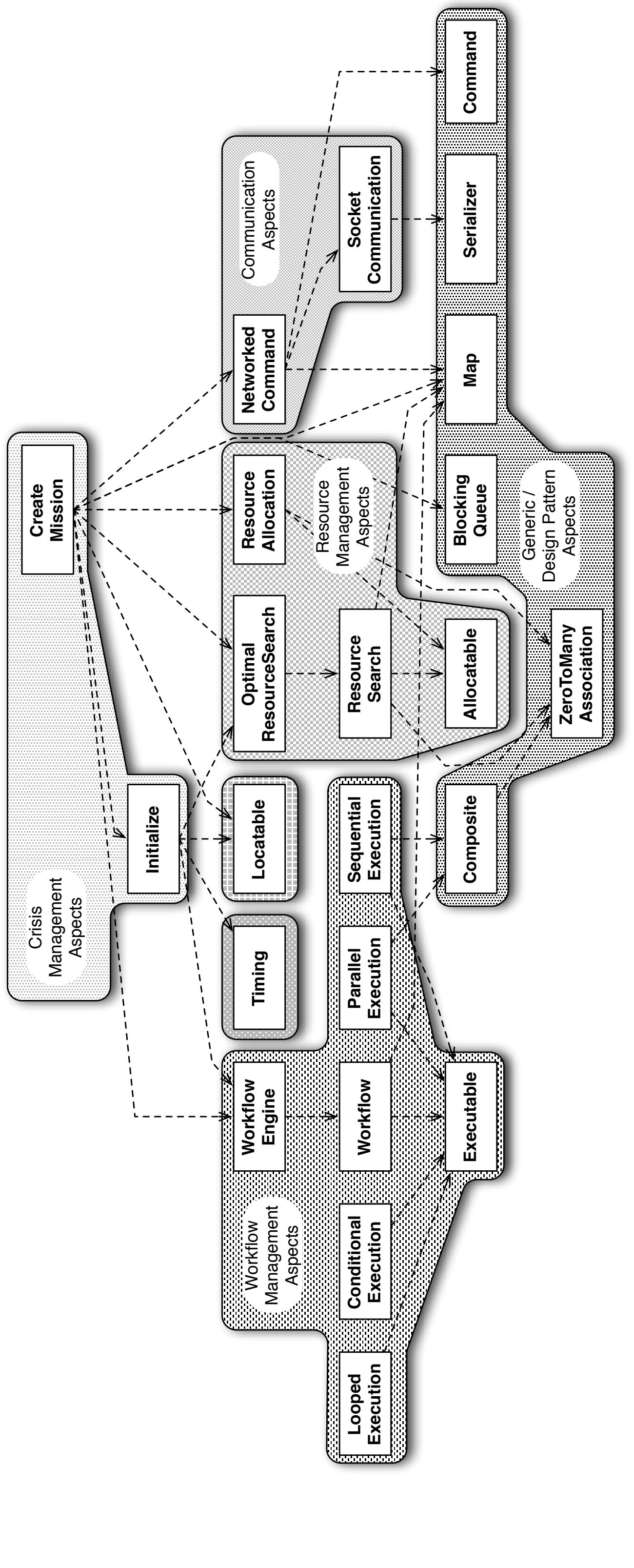


Fig. 14. Design Summary

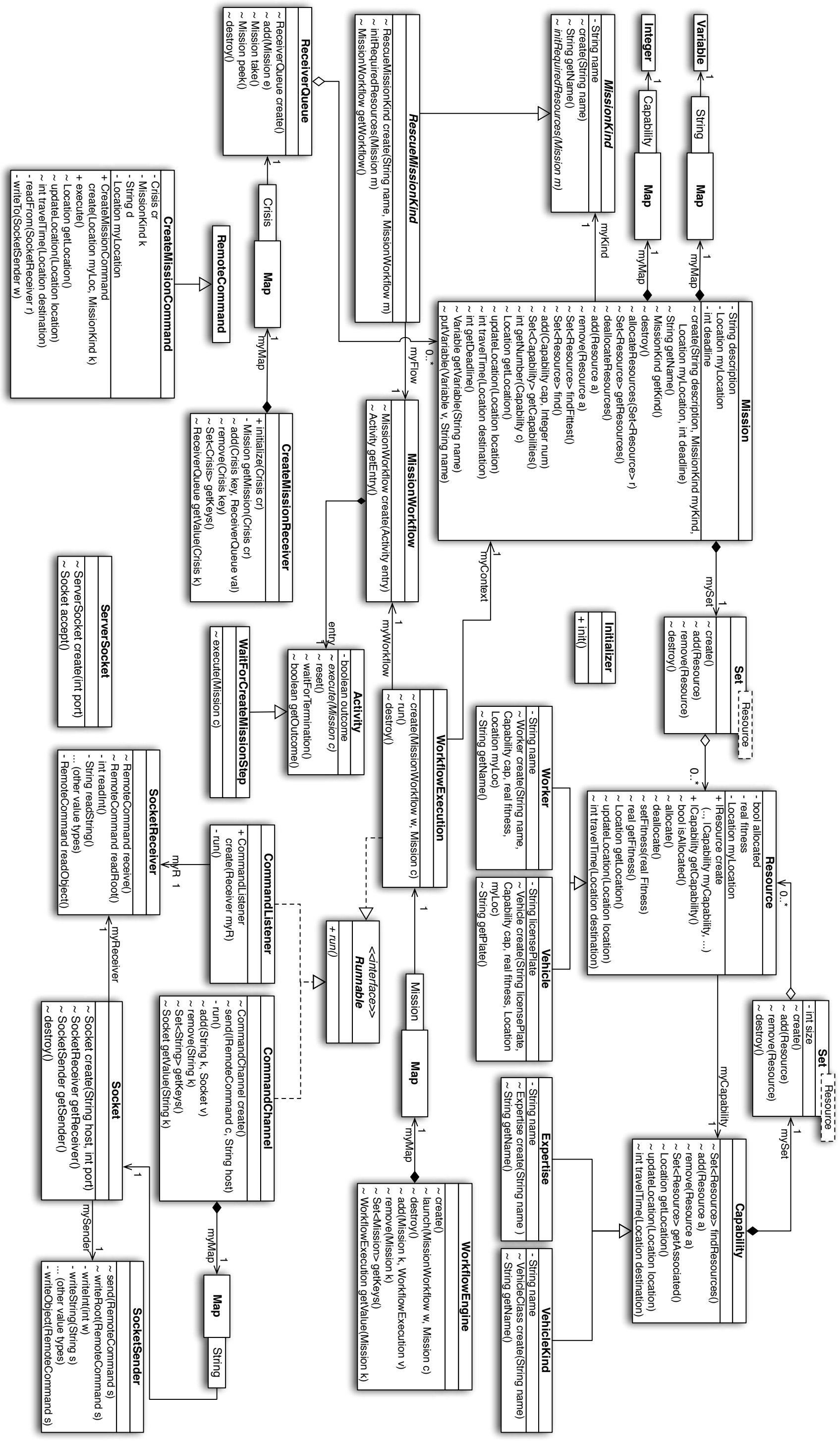

Variable
String
Map
myMap
Integer
Capability
Map
myMap
MissionKind
- String name
~ create(String name)
~ String getName()
~ initRequiredResources(Mission m)
myKind
RescueMissionKind
~ RescueMissionKind create(String name, MissionWorkflow m)
~ initRequiredResources(Mission m)
~ MissionWorkflow getWorkflow()
ReceiverQueue
~ ReceiverQueue create()
~ add(Mission e)
~ Mission take()
~ Mission peek()
~ destroy()
Crisis
Map
myMap
RemoteCommand
CreateMissionCommand
- Crisis cr
- MissionKind k
- String d
- Location myLocation
+ CreateMissionCommand create(Location myLoc, MissionKind k)
+ execute()
~ Location getLocation()
~ updateLocation(Location location)
~ int travelTime(Location destination)
- readFrom(SocketReceiver r)
- writeTo(SocketSender w)
Mission
- String description
- Location myLocation
- int deadline
~ create(String description, MissionKind myKind, Location myLocation, int deadline)
~ String getName()
~ MissionKind getKind()
~ destroy()
~ allocateResources(Set<Resource> r)
~ Set<Resource> getResources()
~ deallocateResources()
~ add(Resource a)
~ remove(Resource a)
~ Set<Resource> findFittest()
~ Set<Resource> find()
~ add(Capability cap, Integer num)
~ Set<Capability> getCapabilities()
~ int getNumber(Capability c)
~ Location getLocation()
~ updateLocation(Location location)
~ int travelTime(Location destination)
~ int getDeadline()
~ Variable getVariable(String name)
~ putVariable(Variable v, String name)
0..*
myFlow
myContext
MissionWorkflow
~ MissionWorkflow create(Activity entry)
~ Activity getEntry()
CreateMissionReceiver
+ initialize(Crisis cr)
- Mission getMission(Crisis cr)
~ add(Crisis key, ReceiverQueue val)
~ remove(Crisis key)
~ Set<Crisis> getKeys()
~ ReceiverQueue getValue(Crisis k)
entry
myWorkflow
Activity
- boolean outcome
~ execute(Mission c)
~ reset()
~ waitForTermination()
~ boolean getOutcome()
WaitForCreateMissionStep
~ execute(Mission c)
ServerSocket
~ ServerSocket create(int port)
~ Socket accept()
Initializer
+ init()
mySet
Set
Resource
0..*
WorkflowExecution
~ create(MissionWorkflow w, Mission c)
~ run()
~ destroy()
Worker
- String name
~ Worker create(String name, Capability cap, real fitness, Location myLoc)
~ String getName()
Vehicle
- String licensePlate
~ Vehicle create(String licensePlate, Capability cap, real fitness, Location myLoc)
~ String getPlate()
Resource
- bool allocated
- real fitness
- Location myLocation
+ IResource create (..., ICapability myCapability, ...)
+ ICapability getCapability()
~ bool isAllocated()
~ allocate()
~ deallocate()
~ real getFitness()
~ setFitness(real Fitness)
~ Location getLocation()
~ updateLocation(Location location)
~ int travelTime(Location destination)
Set
Resource
- int size
~ create()
~ add(Resource)
~ remove(Resource)
~ destroy()
mySet
myCapability
Capability
~ Set<Resource> findResources()
~ add(Resource a)
~ remove(Resource a)
~ Set<Resource> getAssociated()
~ Location getLocation()
~ updateLocation(Location location)
~ int travelTime(Location destination)
<<interface>>
Runnable
+ run()
Mission
Map
myMap
WorkflowEngine
~ create()
~ launch(MissionWorkflow w, Mission c)
~ destroy()
~ add(Mission k, WorkflowExecution v)
~ remove(Mission k)
~ Set<Mission> getKeys()
~ WorkflowExecution getValue(Mission k)
Expertise
- String name
~ Expertise create(String name)
~ String getName()
VehicleKind
- String name
~ VehicleClass create(String name)
~ String getName()
CommandListener
+ CommandListener create(Receiver myR)
- run()
myR
SocketReceiver
~ RemoteCommand receive()
~ RemoteCommand readRoot()
- int readInt()
- String readString()
... (other value types)
- RemoteCommand readObject()
myReceiver
CommandChannel
~ CommandChannel create()
~ send(RemoteCommand c, String host)
- run()
~ add(String k, Socket v)
~ remove(String k)
~ Set<String> getKeys()
~ Socket getValue(String k)
myMap
Map
String
Socket
~ Socket create(String host, int port)
~ SocketReceiver getReceiver()
~ SocketSender getSender()
~ destroy()
mySender
SocketSender
~ send(RemoteCommand s)
~ writeRoot(RemoteCommand s)
- writeInt(int w)
- writeString(String s)
... (other value types)
- writeObject(RemoteCommand s)

Fig. 15. The Woven Structural View of *CreateMission*

could be enhanced significantly. The remainder of this subsection comments on weaknesses of our current design, and presents suggestions for improvement.

Improvement of Current Aspects

The low-level aspects of our design provide minimal functionality, just enough for being useful in the CCCMS context. For example, the *ZeroToManyAssociation* aspect does not provide query operations that can be used to determine if two objects are currently associated or not. Similarly, the *BlockingQueue* aspect does not provide query operations to determine the current length of the queue.

The workflow management concern described in subsection 2.3 is also only a basic design, implemented specifically for the *create mission* functionality. In practice, the complete CCCMS might require more advanced workflow capabilities to efficiently capture the scenarios associated to all kind of missions. For example, the workflow management concern could support the definition of *procedures* in order to share parts of workflows between different types of missions. In practice, support for defining and calling procedures could be added by defining an additional *Procedure* aspect. Other possible extensions include the ability to define explicit parameters for the workflow activities, the support for local variables or advanced synchronization mechanisms (see *http://www.workflow-patterns.com/* for a extensive list of potential workflow features). We believe that most of these features can be captured in RAM aspects in order to enrich the initial version developed for this case study.

Adding New Aspects

In the current CCCMS design, the resources are stored in lists in memory (see *ResourceSearch* aspect). In a realistic CCCMS system, a database management system is required in order to persist all application data such as resources, missions, etc. Implementing this persistence layer would require specific data management aspects for data access, caching and transaction management. It has been shown at the programming language level that aspects can be used to encapsulate database access [13], and it should be possible to model the structure and behavior of such functionality with RAM.

One other important feature of the crisis management system is its connectivity with external services such as police departments, fire departments, hospitals, weather or traffic centers. These connections and the management of events coming from these external systems have not been covered in the design of the *create mission* functionality. A set of RAM aspects should be defined to provide such functionality.

3 Lessons Learned from the Case Study

So far we have not done an in-depth empirical analysis of the practicality of aspect-oriented design in general, and RAM in particular. Working on this case study has nevertheless given us some insight on the usability of our approach. These insights are presented in this section, split into the categories *aspect-oriented design process*, *importance of tool support*, and *support for variability.*

3.1 Aspect-Oriented Design Process

In typical object-oriented design we usually start by designing a small part of the functionality of the system in detail, and then keep on adding functionality iteratively to the existing design, growing our design models gradually. When approaching this case study from an aspect-oriented perspective, we found ourselves starting off with the domain model from the requirements document that contained classes representing the most important CCCMS concepts. We then proceeded by identifying related concepts, and breaking the domain model apart, creating aspects along the way. To each aspect model that now contained some parts of the classes of the domain model we added the detailed design classes that were needed to implement the functionality of the aspect. As a result, our domain model was slowly shrinking, while the number of aspect models were increasing. Whenever possible we would try and reuse already existing aspect models, especially the design pattern aspects that had already been created for other designs.

It was not easy to group related concepts based on the domain model, and then encapsulate them in RAM aspects. The process required a good deal of thought and debate. Several times we had created an aspect model that was later on split into several aspect models, because longer thought had revealed that the aspect was actually providing two distinct functionalities. For instance, initially we had created a *ResourceManagement* aspect that provided operations to search for resources as well as to allocate them. Only while working on the detailed design of *ResourceManagement* we realized that searching and allocating were actually two sub-functionalities that can easily be separated. It also happened that we had designed an aspect model that was later on discovered to not provide a sufficiently distinct functionality from the aspects that it depended on, and as a result we merged it with a higher-level aspect. For example, we had a *ResourceToCapabilityMapping* aspect that ensured that every resource is associated with exactly one capability at creation time, and vice versa. In the end we decided that such an aspect was not adding much functionality beyond what *ZeroToManyAssociation* already provides, and therefore it was merged with *ResourceSearch*.

It was also not always easy to apply good information hiding principles during the CCCMS design. Ideally, a modeler using an aspect A would not have to know about the aspects that A depends on. Our experience shows us that a lot of care needs to be put into the design of the interface of A in order to make this possible.

RAM currently does not provide any algorithm or heuristic on how to discover aspect models during the design, or how simple or complex each aspect model should be, or how to design good aspect model interfaces. The initial hurdle that a modeler has to overcome in order to identify RAM design aspects starting from a requirements document is not small. This struggle is not necessarily a bad thing, though, as it forces the modeler to think deeply about the problem, to look at the design from many angles and perspectives, thus increasing the modeler's understanding of the problem and the designed solution.

Model Scalability The CCCMS is a system of considerable size. However, so far we only concentrated on the design of the create mission functionality. Nevertheless, we decomposed the problem into 23 different aspect models.

Complexity of Individual Aspect Models

Each individual aspect model is small in size. Most models fit entirely on one letter page, some require two pages to print. The structural views of our models contain on average 3 classes. Not surprisingly, *Initialize*, which defines the main CCCMS-specific data structures contains 9 class definitions. The public interfaces of our aspects define on average 8 operations. The state views specify on average 3 states, and the message views depict interaction sequences that involve on average 5 object instances. Again, the `init` operation of the *Initialize* aspect, which takes care of creating and initializing the CCCMS data structures, defines the message view with the most object instances.

Based on our experience, the small size of our aspect models makes them easy to work with. Of course it is easier to understand how a small number of objects work together to achieve a very specific functionality than it is to understand the structure and behavior of the full application. Even if a modeler only looks at the objects that are related to a specific functionality in a non-aspect-oriented design, these objects define state and behavior that are related to many concerns, which is confusing. The model elements inside our aspect models only define the state and behavior relevant to a specific functionality, and hence the modeler is not distracted by unrelated information.

During our design activities, we observed yet another reason why the small size of RAM models is beneficial to a modeler: all information pertaining to a given functionality can be visualized simultaneously on one screen or on one sheet. This allows the modeler to understand the internal workings of an aspect without having to piece information together from multiple windows or sources, which saves considerable time.

Composition Complexity

Our tool helps the modeler to reuse one aspect within the context of another in a consistent way. Complex aspects can depend on many lower-level aspects, e.g., A can depend on B, C and D. Nevertheless, composition is always specified in pairs, e.g. A+B, A+C and A+D. Reasoning about pair-wise composition is relatively simple: the modeler must always only look at two views simultaneously, e.g. the structural view of A and B, in order to specify the desired instantiation or binding directives.

In some rare cases, the order in which the pair-wise compositions are performed matters. To address these situations, RAM allows to specify an ordering for the composition. Conflicts that occur between B, C and D if applied to the same model elements can be resolved by writing conflict resolution aspect models (see section 2.4) that are automatically applied by the weaver when a modeler reuses the conflicting aspects together.

Although lots of effort has been put into limiting composition complexity for the users of RAM, in-depth empirical experiments have to be conducted in the future to determine if modelers are faced with challenges when having to specify compositions of many aspects in real-world sized systems.

Inheritance vs. Merging At several times during the design of the CCCMS, our object-oriented design background made us want to use generalization/specialization relationships (inheritance) to implement a design solution for a particular situation. For instance, we wanted to create an abstract superclass `CCCMSResource`, with subclasses `Worker` and `Vehicle`, because we would like `Worker` and `Vehicle` to share common attributes and operations. Since RAM supports one-to-many mappings during instantiation, the same effect can be achieved simply by mapping both `Worker` and `Vehicle` to `ResourceWithFitness`. Both former classes, as a result, "inherit" the attributes and operations of `ResourceWithFitness`. A similar situation occurred within the workflow management aspects, where we initially had the intention of creating a `ControlStructures` super class to group together our many workflow control structures.

It is not clear to us, for now, if weaving can completely replace inheritance. Further experiments are necessary to find an answer to this question.

3.2 Importance of Tool Support

After having completed a major part of the design of the CCCMS, a new member joined our design team. In order to understand the details of the design of one aspect, he had to also look at and understand the structure and behavior of the instantiated aspect models. Sometimes, he had to dig even deeper, and look at aspects that were two levels below the aspect he wanted to understand in order to get a feeling of how the design was supposed to work. The RAM tool provides essential help in such a situation, because it allows a modeler to create an independent model of any aspect. Since the independent model contains all the model elements of the instantiated aspects, the modeler can look at the "full picture" of the aspect under study, and is able to understand the design in detail.

For example, a modeler trying to understand the structure of *OptimalResourceSearch* does not see any association between `Capability` and `ResourceWithFitness`. Even an inspection of *ResourceSearch* does not reveal that `Capability` is linked to a set of `Resources`. It is only when looking at *ZeroToManyAssociation* that the association between `|Data` and `|Asssociated` appears. In the independent aspect model of *OptimalResourceSearch* generated by our tool, however, the association between the two classes is readily visible.

Having realized the importance of the tool support that a designer can rely on, we are planning in the future to extend our tool with additional features that could improve the usability of RAM. As a first step, we are planning to extend the tool to color-code model elements in an independent aspect model depending on the aspect(s) they originated from. Ultimately, the goal is to implement a tool in which it would be possible to interactively "unfold" instantiated aspects in order to see the structure and behavior they provide in detail.

Correctness Checks The RAM weaver performs extensive consistency checks *within the generated independent aspect model* and within the *final base model.*

Our tool compares, for each object life line in the final generated sequence diagrams, that the ordering in which the object accepts incoming messages corresponds to the ordering of the messages accepted by the woven state diagram. If the state diagram refuses a message, consistency is violated. This signals to the developer that the instantiations and bindings (or the ordering of the instantiations and bindings) of the state and message views contradict each other and have to be revisited.

Although this does not guarantee correctness of the compositions, the probability of detecting erroneous composition directives is significant: the modeler has to specify the same composition from two points of view: the state view and the message view. The only situation in which the tool cannot detect the error is the one where the modeler specifies the *same wrong composition in both views.*

The consistency checks provided by RAM have helped us to detect composition problems during the design of the CCCMS. It is not clear, however, how effective these checks are when models grow even bigger. In-depth empirical experiments have to be conducted in the future to determine if modelers are faced with challenges when having to specify compositions of many aspects in real-world sized systems.

3.3 Support for Variability

Often, a given functionality can be implemented (and hence modeled) in different ways. Likewise, in a product line approach, several applications with similar, but not identical functionality are to be modeled. When applying an aspect-oriented approach, a high-level functionality (or a super-set of features in a product line) can be decomposed into many lower-level functionalities (or many individual features), each one modeled in a separate aspect. When the aspect-oriented approach is applied over several abstraction layers (high–medium–low-level functionalities or features–subfeatures–subsubfeatures), it is necessary to express the rules that govern the correct use of the aspect models. Since RAM encourages the use of multiple abstraction layers, we extended RAM with a software product line approach to handle variabilities, i.e. to model the set of correct configurations of aspects that are available in an aspect framework or product line.

We have not presented the feature model support of RAM in this paper for space reasons. The 23 aspects that constitute the design of *one functionality*, e.g. create mission, of *one specific CCCMS backend* are already sufficiently complex to illustrate the power of RAM. The only optional variant we partially designed and presented is *Logistics*. One could imagine instantiating a CCCMS backend design that does not support keeping track of the location of resources, and does not optimize the allocation of resources based on travel time to the mission location. In the AspectOPTIMA [14] case study, which describes a product line of transaction support systems, the need for variability support becomes apparent. Indeed, the 15 aspect models can be combined in 10 different ways to generate the model of a concrete transaction support product.

While the *designer* of a model of a product line composed of many interdependent and potentially conflicting aspect models has a very difficult task, using the product line model to derive a specific model of a product is very simple. Our tool presents to the *user* of the product line a feature diagram describing all variation points of the product line. Once the user made his choice, our tool weaves the corresponding aspect models together to create the detailed design model for the specific product. For instance, for the current CCCMS design, a user could instruct the tool to include the feature *Logistics*, or to generate a model without the *Logistics* feature enabled. If *Logistics* is not chosen, then the *Initialize* aspect does not instantiate *Locatable* and *Timing*, and as a result the *LocatableTimingOptimalResourceSearch* conflict resolution aspect is not activated, and consequently *OptimalResourceSearch* looks for resources based on capability ratings only.

4 In-Depth Comparison to Related Work

The goal of this section is to present an in-depth comparison of RAM with related work. In [15], a freely available 71 page technical report, Schauerhuber et al. present an overview of 8 different AOM approaches. The surveyed approaches are:

1. The Aspect-Oriented Design Model of Stein et al. [16,17]
   In [17], Stein et al. introduce a way to express various conceptual models of pointcuts (called JPDDs for *Join Point Designation Diagrams*) in aspect-oriented design. Structural and behavioral modeling is achieved by employing for instance class diagrams, state charts, and sequence diagrams. Their objective is not to perform the weaving at the modeling level, but rather to generate code for aspect-oriented programs (such as generating *AspectJ* [18] code) from an aspect-oriented design as shown in [19].
2. The JAC Design Notation of Pawlak et al. [20,21]
   The JAC Design Notation is a lightweight UML 1.x extension that was created to make it possible to model a design that uses the JAC Framework, a middleware (including IDE and modeling support) for the development of J2EE applications that require support for persistence, security, fault tolerance, load balancing and other concerns. The approach only supports class diagrams, and hence modeling of behaviour is not supported. Nevertheless, $\ll$ *pointcut* $\gg$ stereotypes can be used to decorate associations that link an $\ll$ *aspect* $\gg$ class to a base class. Instructions written in a proprietary, textual language are used to statically describe which operations of a base class are advised by operations defined in the aspect class. The main use of the approach being design documentation, model weaving is not supported.
3. Aspect-Oriented Software Development with Use Cases of Jacobson et al. [22]
   The AOSD with Use Cases approach defines a software development process that emphasizes the separation of concerns from requirements elicitation with use cases down to the implementation. High level models specified in the form of *use cases slices* are successively refined and mapped to design models that use class diagrams to represent structure and sequence diagrams to represent behaviour. In the end, the design is mapped to aspect-oriented

code. As a consequence, model weaving is not needed, and hence not supported. The power of the approach is its support for traceability: explicit $\ll$ *trace* $\gg$ dependencies among artifacts produced at various stages of software development are used to link models that pertain to the same concern.

4. Behavioral Aspect Weaving with the Approach of Klein et al. [23,24]
This approach proposes a weaver for scenarios (sequence diagrams or Message Sequence Charts). An aspect is defined as a pair of scenarios. For instance, to weave an aspect sequence diagram into a target sequence diagram, the aspect sequence diagram is composed of a pair of sequence diagrams: one sequence diagram representing the pointcut (specification of the behavior to detect), and the other sequence diagram representing the advice that specifies the expected behavior at the join point. Similar to *AspectJ*, where an aspectual behavior can be inserted *'around'*, '*before*' or '*after*' a join point, an advice in this behavioral weaving approach may extend the matched behavior, replace it with a new behavior, or remove it entirely. The approach defines a two-phased process weaving: 1) a generic detection where the pointcut is used to determine all the join points in the target model and 2) a generic composition mechanism where the advice model is composed with the target model at the join points previously detected. RAM uses this approach to weave the message views of the RAM aspects models.

5. The Motorola WEAVR Approach of Cottenier et al. [25,26]
The Motorola WEAVR approach and tool have been developed in an industrial setting. Behavior is modeled using the Specification and Description Language (SDL), a formalism related to state diagrams. In order to be able to reuse aspects, *mappings* have to be defined (equivalent to our instantiations) that link a reusable aspect to the application-specific context in which it is to be deployed. The WEAVR approach focuses exclusively on SDL, and supports model execution and code generation.

6. The AOSD Profile of Aldawud et al. [27,28]
The AOSD profile is a UML 1.x profile that can be used to model the structure of a concern using class diagrams and the behaviour using state diagrams. The AOSD profile is mostly aimed at modeling an aspect-oriented program, since model-weaving is currently not supported.
Aspects are represented by an $\ll$ *aspect* $\gg$ stereotype, which is derived from the meta-class `Class`. Concurrent state machines are used to specify crosscutting behavior in orthogonal regions. The behaviour defined in different orthogonal regions is combined using event broadcasting. $\ll$ *crosscut* $\gg$ dependencies between aspects and base classes or aspects and aspects dictate the ordering in which events are propagated between the orthogonal regions.

7. The Theme/UML Approach of Clarke et al. [29]
Theme/UML introduces a theme module that can be used to represent a concern at the modeling level. Themes are declaratively complete units of modularization, in which any of the diagrams available in the UML can be used to model one view of the structure and behavior the concern requires to execute. In Theme/UML, class diagrams and sequence diagrams are typically

used to describe the structure and behavior of the concern being modeled. The binding to a base model is done by template parameter instantiation.

8. Aspect-Oriented Architecture Models of France et al. [30,31]
The symmetric model composition technique proposed by France et al. [30,31] supports composition of model elements that present different views of the same concept. This composition technique has been implemented in a tool called *Kompose* [32,**?**]. The model elements to be composed must be of the same syntactic type, that is, they must be instances of the same meta model class. An aspect view may also describe a concept that is not present in a target model, and vice versa. In these cases, the model elements are included in the composed model. The process of identifying model elements to compose is called *element matching*. To support automated element matching, each element type (i.e., the element's meta-model class) is associated with a signature type that determines the uniqueness of elements in the type space: two elements with equivalent signatures represent the same concept and thus are composed. Currently, Kompose focuses mainly on the merging of class diagrams. RAM uses this approach to compose the structural views of the RAM aspect models.

In [15], the authors present how to model the *Observer Design Pattern* using each of these 8 approaches, and show how to apply the observer pattern in the context of a model of a library management system. The authors then proceed to compare the 8 approaches according to 6 criteria: a) language, b) concern composition, c) asymmetric concern composition, d) symmetric concern composition, e) maturity, and f) tool support.

In the following subsections, mimicking the presentation of [15], we show how to model the *Observer Design Pattern* aspect with RAM and how to apply the observer aspect to the library management system. We then proceed to evaluate RAM based on the 6 criteria presented in [15]. Note that for each of the 6 criteria, a table is proposed in [15] to summarize the evaluation of the 8 AOM approaches. In this section, we present the *same tables* for RAM, and point out the advantages or the disadvantages of RAM in comparison to the other approaches. Using this information, the interested reader can compare RAM with the 8 AOM approaches presented in [15]. Note that there are other AOM approaches that RAM could be compared to, such as UML *Package Merge* [33] which defines how the contents of one package are extended by the contents of another package, Whittle and Araujo's approach [34], which represents behavioral aspects with scenarios, and the Whittle and Jayaraman approach called MATA [35].

4.1 Observer Design Pattern and Library Management System

The classic *Observer Design Pattern* [9] is a software design pattern in which an object, called the *subject*, maintains a list of dependents, called *observers*. Whenever the subject's state changes, it notifies all observers by calling one of the their operations. The observer design pattern has been used in many publications to demonstrate different aspect-oriented programming and aspect-oriented modeling techniques.

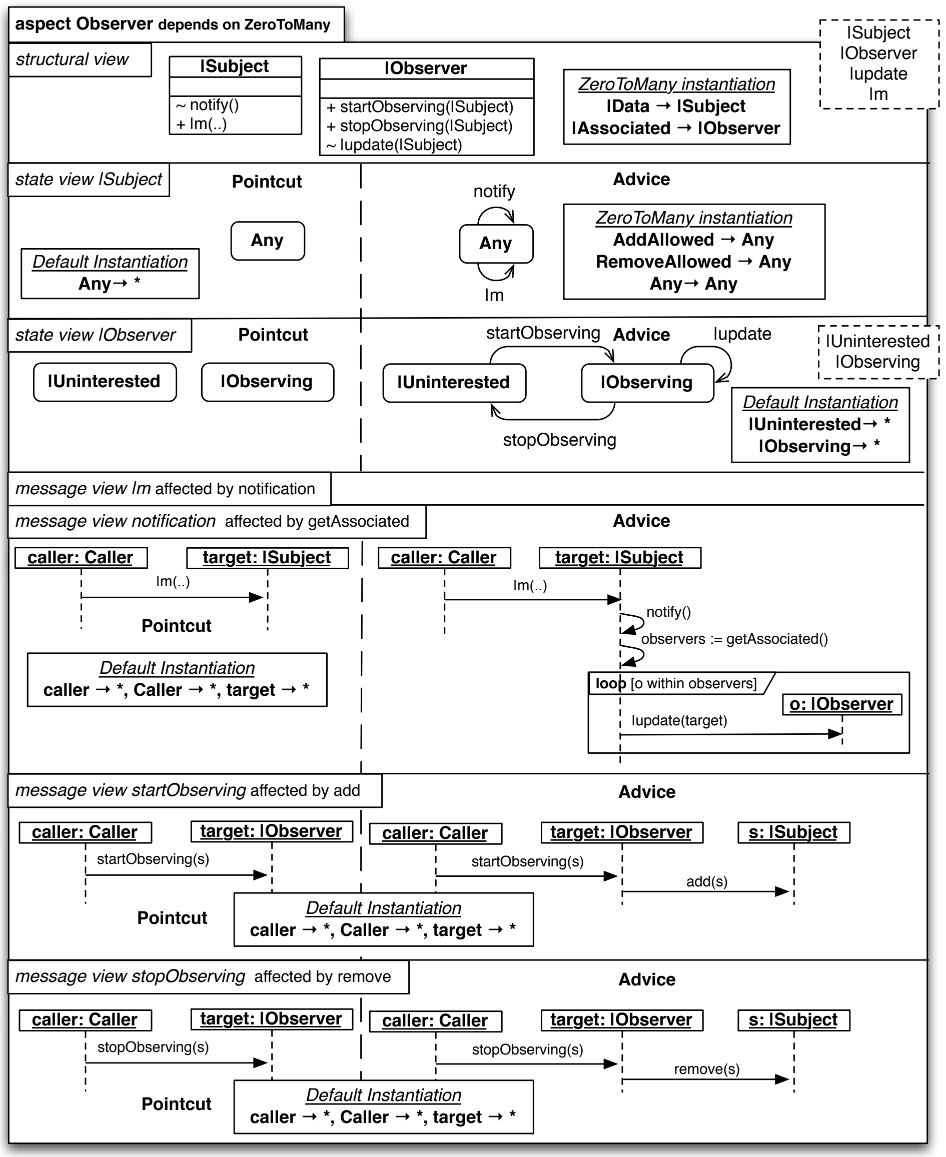


Fig. 16. The *Observer* Design Pattern Aspect modeled with RAM

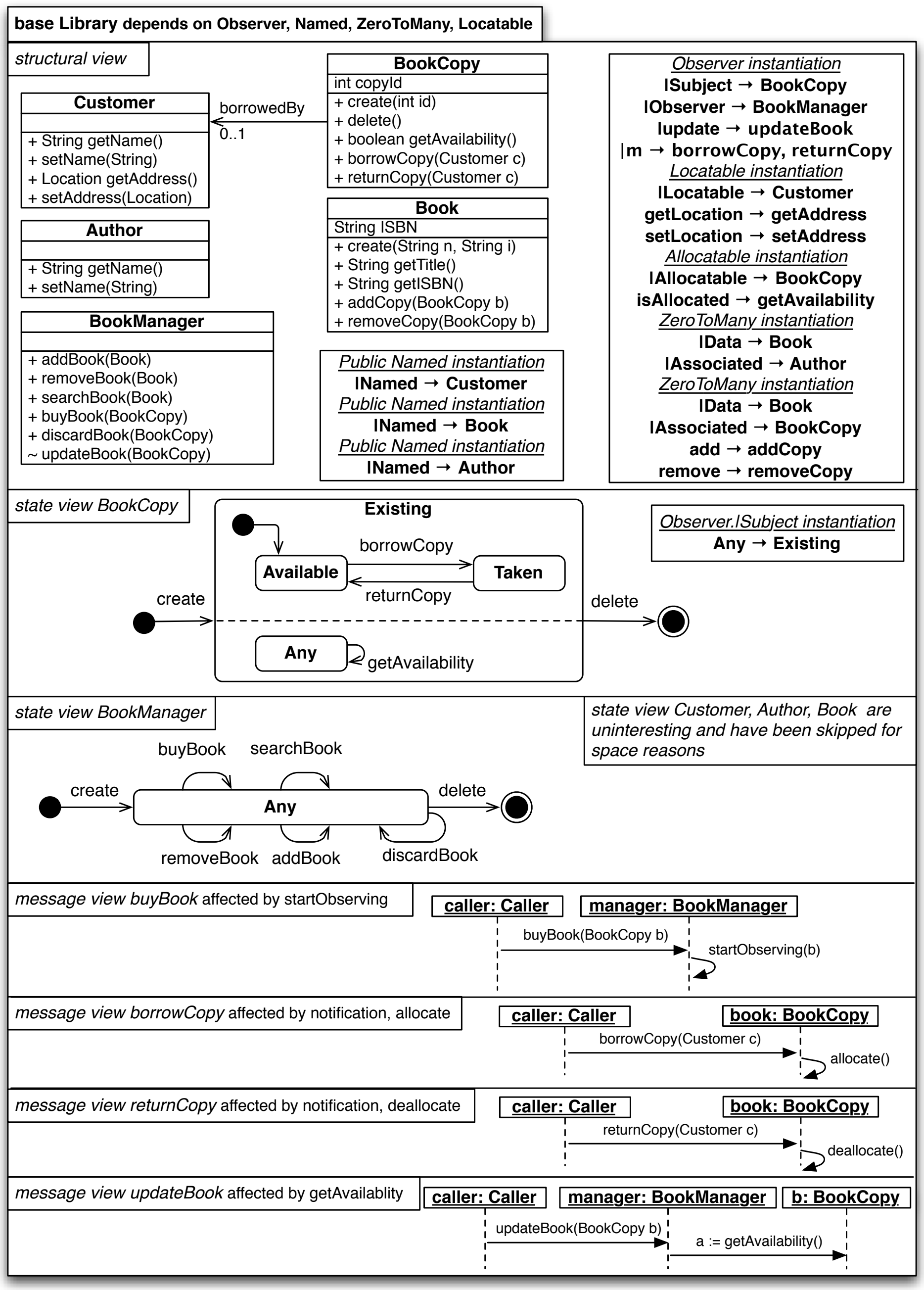
The *Observer* RAM aspect model is shown in Fig. 16. The structural view contains two partial classes, the *|Subject* and the *|Observer* class. The *ZeroToManyAssociation* aspect described in subsection 2.1 in Fig. 1 is reused to associate one subject to many observers. The *Observer* class defines the two public operations `startObserving` and `stopObserving` that allow an observer instance to register, rsp. deregister, with a subject instance. The two corresponding message views show that the operations *add* and *remove*, introduced by *ZeroToManyAssociation*, are used to update the set of observers of a subject. The *Subject* class defines the `|m` operation which represents operations that modify the state of the subject instance. The *message view |m* specifies that every call to `|m` is to be affected by the notification message view. Notification states that a call to `callToBeObserved` should be followed by a call to `notify`, which successively loops through all the registered observers (obtained by calling `getAssociated` provided by *ZeroToManyAssociation*) invoking the `|update` operation. The default instantiation states that *all* calls to instances of the class `|Subject` are to be observed, which means in our case that every call to `|m` is observed. The *state view |Observer* specifies that after invoking *startObserving* on an observer instance, any number of calls to `|update` are allowed to execute, until the *stopObserving* operation is executed.

Fig. 17 shows how the *Observer* aspect is applied in the context of the *Library Management System* model. The observer instantiation directives in the structural view specify that instances of the *BookCopy* class are the subject of observation, and that instances of the *BookManager* class are the observers. The modifying operations are *borrowBook* and *returnCopy*, and the operation that should be called whenever a subject's state is modified is *updateBook*. The structural view also shows that, in order to design the library management system, we were able to reuse many of the aspects that we designed for the CMS. *Allocatable* is used to remember the availability of a book copy. *Locatable* is reused to provide customers with an address. Finally, *ZeroToMany* is used to associate a book with many book copies, as well as to associate a book with many authors. Books, customers and authors also reuse *Named*, an aspect from the AspectOptima framework that associates a name in form of a string with a class.

The *message view buyBook* describes how, whenever a new copy of a book is bought, the book manager registers as an observer of the book copy by invoking *startObserving*. The most interesting message views are *borrowCopy* and *returnCopy*. They are both affected by the *notification message view* of the *Observer* aspect. The result of this is presented in Fig. 18, which illustrates the final message view of *borrowCopy* after our weaver resolved all aspect model dependencies and created an independent base model.

4.2 Language

The first criteria that [15] uses to compare different AOM approaches is the *language* criteria. The evaluation results for RAM for this criteria are presented in Table 1. A RAM aspect is specified with UML 2.x. The concepts of *pointcut* and *advice* are explicitly present in a RAM aspect. Consequently, the UML


base Library depends on Observer, Named, ZeroToMany, Locatable
structural view
Customer
+ String getName()
+ setName(String)
+ Location getAddress()
+ setAddress(Location)
borrowedBy
0..1
BookCopy
int copyId
+ create(int id)
+ delete()
+ boolean getAvailability()
+ borrowCopy(Customer c)
+ returnCopy(Customer c)
Author
+ String getName()
+ setName(String)
Book
String ISBN
+ create(String n, String i)
+ String getTitle()
+ String getISBN()
+ addCopy(BookCopy b)
+ removeCopy(BookCopy b)
BookManager
+ addBook(Book)
+ removeBook(Book)
+ searchBook(Book)
+ buyBook(BookCopy)
+ discardBook(BookCopy)
~ updateBook(BookCopy)
Public Named instantiation
\|Named → Customer
Public Named instantiation
\|Named → Book
Public Named instantiation
\|Named → Author
Observer instantiation
\|Subject → BookCopy
\|Observer → BookManager
\|update → updateBook
\|m → borrowCopy, returnCopy
Locatable instantiation
\|Locatable → Customer
getLocation → getAddress
setLocation → setAddress
Allocatable instantiation
\|Allocatable → BookCopy
isAllocated → getAvailability
ZeroToMany instantiation
\|Data → Book
\|Associated → Author
ZeroToMany instantiation
\|Data → Book
\|Associated → BookCopy
add → addCopy
remove → removeCopy
state view BookCopy
Existing
Available
borrowCopy
Taken
returnCopy
create
delete
Any
getAvailability
Observer.\|Subject instantiation
Any → Existing
state view BookManager
buyBook
searchBook
create
Any
delete
removeBook
addBook
discardBook
state view Customer, Author, Book are uninteresting and have been skipped for space reasons
message view buyBook affected by startObserving
caller: Caller
manager: BookManager
buyBook(BookCopy b)
startObserving(b)
message view borrowCopy affected by notification, allocate
caller: Caller
book: BookCopy
borrowCopy(Customer c)
allocate()
message view returnCopy affected by notification, deallocate
caller: Caller
book: BookCopy
returnCopy(Customer c)
deallocate()
message view updateBook affected by getAvailablity
caller: Caller
manager: BookManager
b: BookCopy
updateBook(BookCopy b)
a := getAvailability()

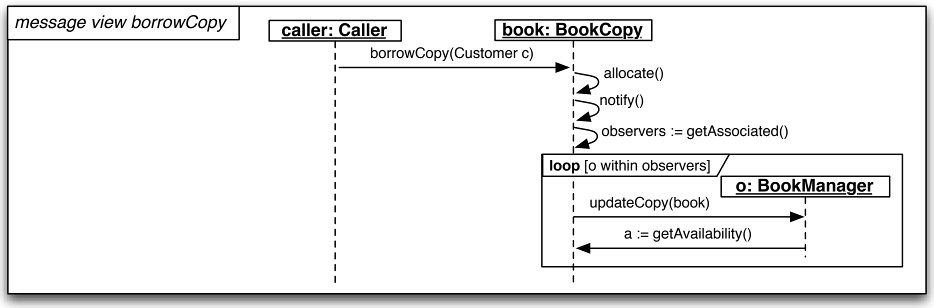


Fig. 18. Woven *borrowCopy* Message View

	Modeling Language (L.L)		Extension Mechanism (L.E)		Platform Innfluences(L.I.)	Diagrams (L.D)		Design Process (L.DP)		Scalability (L.S)		Refinement Mapping (L.R)	Alignment to Phase (L.A)	Traceability (L.T)	
	UML 1.X	UML 2.0	Metamodel	UML Profile		Structural Diagrams (L.D)	Behavioral Diagrams (L.D)	process description	guidlines	high-level modeling elements	proven with examples			internal	external
RAM	🟦	√	√	🟦	🟦	CD	SD, SMD	~	√	√	√	√	D	√	🟦

Table 1. "Language" applied to RAM

metamodel has been extended. A RAM aspect is composed of a class diagram, sequence diagrams and state diagrams. In [2] and in this paper, some guidelines have been stated on how to do aspect-oriented design using RAM, but we don't propose a full process description or a methodology yet. In [15], scalability is defined as "*the ability to cope with small as well as large modeling projects*", and scalability is "*investigated with respect to first, which high-level modeling elements of an approach support scalability, e.g., UML packages, and/or high-level diagram types, and second, if scalability has been proven or not proven in real-world projects or by modeling examples that go beyond the composition of two concern modules*". In this context, by defining the *depends on* relationship between aspects, RAM proposes a way to describe a system with models at different levels of abstraction. The scalability of RAM has been demonstrated by 2 big case studies: AspectOptima and now also the CMS. The RAM aspects are woven according to the instantiation and binding directives expressed explicitly in the models. While this provides internal traceability, RAM currently does not provide external traceability, which is defined in [15] as "*how aspect-oriented design models relate to the full software development life cycle*".

Discussion: In [15], the authors identify several issues related to the 8 AOM approaches studied:

1. "*Behavioral Diagrams are Catching up*": Only half of the studied approaches support behavioral diagrams. RAM stands out in this respect, because it currently supports the use of two kinds of behavioral diagrams (state and sequence diagrams).

2. "*Missing Guidance in the Design Process*": Only the *Theme/UML* and *AOSD with Use Cases* approaches are integrated into the software development process, the other studies approaches did not provide guidance on how the design models are created. RAM does not yet propose a methodology, but the large case studies used to describe RAM have allowed us to elaborate a set of design guidelines.
3. "*Missing Full External Traceability*": Except from *Theme/UML* and *AOSD with Use Cases*, no approach supports external traceability. RAM also does not support external traceability.
4. "*Moderate Scalability*": Only half of the studied approaches provide information hiding means for aspect models. Most studied approaches only showed examples containing a small number of aspects. Only *Theme/UML*, *AOSD with Use Cases*, and the *Motorola WEAVR* approach present examples that weave 3 or more aspects together. RAM clearly provides superior scalability. Each RAM aspect model defines a precise aspect interface, and the weaver performs automatic information hiding when the aspect is reused in some other model. The scalability of RAM has been demonstrated using 2 big case studies, where over 20 aspects were woven together to produce the final application model.

4.3 Concern Composition

	ConcernModule(CC.CM)	Composition Mechanism (CC.M)	Element Symmetry (CC.ES)	Composition Symmetry (CC.CS)	Rule Symmetry (CC.RS)	Effect (CC.E)	Composition Semantics (CC.S)		Composition (CC.C)		Composed Module (CC.CP)	Interaction (CC.I)		Conflict Resolution (CC.CR)		
							detection	composition	static	dynamic		Module	Rule	avoid	detect	resolve
RAM	aspect package (with template) composed of 3 parts. Class Diagram, Aspectual Seq. Diag, Aspectual Statechart.	CMP, PA	√	√	🟦	~	√	√	√	~	Stand. UML	√	🟦	~	√	√

Table 2. "Concern Composition" applied to RAM

The comparison of RAM with the other approaches according to the concern composition criteria is presented in Table 2. In RAM, a concern module is a special UML package that encapsulates all model elements related to the structure and/or behavior of a concern. The current version of RAM [2] supports aspect models that use class diagrams, state diagrams and sequence diagrams. RAM uses two composition mechanisms:

1. A compositor (CMP) to merge the class diagrams. This compositor is the symmetric model composition technique proposed by France et al. [30,31] which has been implemented in Kermeta in a tool called *Kompose* [32].

2. A weaver where the aspects are expressed with a pair Pointcut-Advice (PA) to weave the behavioral diagrams.

By supporting these two composition mechanisms, RAM supports both symmetric (composition *base-base*) and asymmetric (weaving *aspect-base*) paradigms. Currently, the effects of the weaving of a RAM aspect into a base model or into another aspect are not visually represented in the woven model. Note that we are currently working on using the same techniques as presented in [36,37] to automatically add tags (such as *introduced, updated, ...*) on the model elements of the woven model to clearly identify the effect of the weaving of an aspect model, if desired.

The composition approach proposed by RAM for the behavioral diagrams is decomposed into two steps: a step of detection of the join points corresponding to the pointcut model, and a step of composition of the advice model at the level of the join points previously detected. This weaving process is static, in the sense that the weaving is not preformed while the models are executed, but rather during a separate weaving process. However, since the weaving is applied to behavioral models, the weaving of RAM aspects modifies the system run-time behavior. Also, the sequence diagram weaving technique employed by RAM [23] composes aspects based on the semantics of the sequence diagrams as opposed to the syntax. The final model that the RAM weaver produces is a "standard" UML diagram. Finally, RAM allows a modeler to specify the interaction between aspects (using explicit dependencies among aspects), and RAM provides means to the detection and the resolution of aspect conflicts.

Discussion: In [15], the authors identify several issues related to the 8 AOM approaches studied:

1. "*Popularity of Asymmetric Concern Composition*": Schauerhuber et al. note that "*up to now little interest has been shown in evaluating when asymmetric and symmetric approaches have prevailing advantages and shall be employed*". RAM combines symmetric and asymmetric composition approaches. Based on our experience, both paradigms are needed for an AOM approach to be general and reusable.
2. "*Composition often Deferred to Implementation*": [15] notes that composition at the modeling level is only supported by half of the surveyed approaches. RAM aspects can be composed at the modeling level.
3. "*Moderate Support for Modeling Interactions*": Schauerhuber et al. argue that for an unambiguous specification of a system it is necessary to make module interaction explicit. They note that only the Motorola WEAVR approach explicitly specifies dependencies between aspects. In RAM, dependencies between aspects are clearly specified using explicit depends on links among aspect modules. This allows the weaver to automatically resolve indirect dependencies. As a result, aspect reuse is very simple: reusing a high-level aspect does not require the modeler to be aware of the low-level aspects that the high-level aspect depends on.
4. "*Conflict Resolution Based on an Ordering for Composition, Only*": Only the approach of France et al. can detect syntactical conflicts among aspects.

None of the surveyed approaches provide a sophisticated conflict resolution mechanism apart from specifying the composition order. RAM has strong support for conflict detection and resolution as explained in the following paragraphs.

Aspect Model Conflicts When several aspect models are applied within the same product model, *conflicts* can occur. A conflict between aspects A and B refers to the situation in which the correct model expressing the composition of the aspects A and B cannot be obtained by simply weaving A and B into the target model. In these situations, one or both of the aspect models' structure and behavior must be modified to take into account the co-existence of the other aspect model.

RAM does not allow to detect aspect conflicts automatically. However, our weaving tool can warn the modeler that two aspects are potentially conflicting. Conflicts occur only in a certain well-defined conditions: when the two aspects A and B are such that *at least one of the pointcuts of A matches at least one of the elements within the pointcuts or advice of B*. The following two special cases fall into this category:

- When model elements from two aspects are bound to the same base model elements, and one of the aspects removes structure or functionality from the elements it is applied to, and the other aspect expressed the presence of the removed functionality in one of its pointcuts.
- When model elements from two aspects are bound to the same base model elements, and one of the aspects defines behavior that uses a template method call in its pointcut that matches a call in an advice of the other aspect.

By cross-checking each aspect with each other aspect, our tool generates warnings listing the aspects that potentially conflict. However, our tool never reports a conflict between aspects A and B if A directly or indirectly depends on B, even if A's pointcut matches elements of B's pointcuts or advice or some of the generic elements of A are bound to the elements of B. In this case we assume that the modeler who expressed the dependency is aware that both aspects apply to common modeling elements, and has designed A in a way that takes this into account. As a result, our tool only generates conflict warnings for aspect model pairs which are not linked by a dependency relationship.

Once identified, conflicts can be resolved by the modeler by designing a conflict resolution aspect model as explained in section 2.4. The conflict resolution model contains *conflict criteria conditions* that specifies under which conditions the conflict occurs, and modification compartments that model the required adaptations that have to be applied to the conflicting models to resolve the conflict. Whenever all of the aspects that a conflict resolution model depends on are used within a target model, our tool *automatically* checks the conflict criteria specified in the conflict resolution model. If the condition is verified, the adaptations expressed in the modification are *automatically* applied to the target model.

We did not identify many conflicts within the CCCMS. The only conflict is the one between *OptimalResourceSearch*, *Locatable* and *Timing*. The conflict resolution model shown in Fig. 11 demonstrates how *OptimalResourceSearch* is adapted in order to take resource and mission location into account when determining the best match for a given request.

4.4 Asymmetric Concern Composition

	Structural Join Point (AS.SJP)		Behavioral Join Point (AS.BJP)		Explicite JoinPointModel (AS.JPM)	Standardized Pointcut (AS.P)	SimplePointcut (AS.SP)		CompositePointcut (AS.CP)		Quantification Method (AS.QM)			Relative Position (AS.RP)			Abstraction (AS.A)	
	static	dynamic	static	dynamic			graphical	textual	graphical	textual	declarative	imperative	enumeratio	before	around	after	high	low
RAM	√	√	√	🟦	√	√	√	🟦	~	🟦	√	🟦	√	√	√	√	√	√

Aspectual Subject

	Behavioral Advice (AK.BA)	Structural Advice (AK.SB)	Composite Advice (AK.CA)	Abstraction (AK.A)	
				high	low
RAM	√	√	√	√	√

Aspectual Kind

Table 3. "Aspectual Subject" and "Aspectual Kind" applied to RAM

Table 3 shows the evaluation of RAM with respect to the asymmetric concern criteria. The asymmetric concern composition operator of RAM (the sequence diagram weaver [23,24] and GeKo [36,38]) allows a modeler to use any model element in the pointcut, and therefore any model element can be used as a join point. As a result, RAM supports the specification of both static and dynamic structural join points (for instance, *class* and *objects* respectively), but also static behavioral join points (such as *message*). However, RAM does not allow the specification of dynamic behavioral join points, which can only be detected at runtime. The join points in RAM are explicitly and formally defined (see [23,24,36]). The pointcuts are standardized, since a pointcut is an instance of a given metamodel. For instance, for sequence diagram model weaving, a pointcut is a standardized sequence diagram. In RAM, a pointcut is graphical and not textual, and RAM partially supports refinement of pointcuts**.** The quantification method is declarative, but simple enumeration of join points is also possible. Finally, an advice can be added before or after a join point, but it can also replace a join point. The level of abstraction can be *high*, when a user of RAM specifies that an aspect depends on another aspect, but also *low,* because a join point detection mechanism can precisely define where the join points are.

The "Aspectual Kind" table in Fig. 4 shows that in RAM an advice can be behavioral or structural, depending on the nature of the model used. A RAM aspect can express composite advice. The level of abstraction of a RAM aspect can be low (level of the model elements used), but also high because an aspect can depend on other aspects.

Discussion: In [15], the authors identify several issues related to the 8 AOM approaches studied:

1. "*Missing Formal Definition of Join Point Models*": Half of the surveyed approaches only implicitly define the join point model via their pointcut mechanism. The remaining approaches provide a join point model description in terms of natural language. The weavers used in RAM formally define the join point model for class, sequence and state diagrams.
2. "*Modeling Aspectual Subjects at a Low Level of Abstraction*": [15] stresses that while all approaches allow a modeler to express low-level design, only half of the surveyed approaches support the creation of aspect models with a higher level of abstraction. RAM's support for multi-abstraction level modeling is very elaborate: RAM aspects can form complex hierarchies, which allows a RAM aspect providing high-level functionality to depend on low-level functionality provided by other aspects. In addition, aspect models define clear aspect interfaces which hide dependencies on lower-level aspects from the outside world.

For the other points mentioned in [15] in the section discussing the asymmetric concern composition criteria there are no real differences between RAM and the other approaches.

4.5 Symmetric Concern Composition

	Comp. Elements		Match Method (S.MM)			Integration Strategy			Abstraction (S.A)	
	Structural Composeable Elements (S.SCE)	Behavioral Composeable Elements (S.BCE)	match-by-name	match-by-signature	no-match	Merge (S.M)	Overide (S.O)	Bind (S.B)	high	low
RAM	√	🟦	√	√	🟦	√	√	√	√	√

Table 4. "Symmetric Concern Composition" applied to RAM

Asymmetric composition approaches make a distinction between aspect and base model elements, whereas symmetric approaches don't make this distinction [39]. Table 4 shows the evaluation of RAM for symmetric concern composition. To merge class diagrams, RAM uses the symmetric model composition technique proposed by France et al., which has been implemented in Kermeta in a tool called *Kompose* [32]. Consequently, concerning the symmetric composition technique, the features of both the approach proposed by France et al. and RAM are similar, and the comments on the France et al. approach presented in [15] are also valid for RAM.

4.6 Maturity

Table 5 shows the summary of the maturity evaluation of RAM. Until now, RAM has been applied to two big case studies and several small academic examples. The current version of the AspectOptima case study has 18 RAM aspect models and 5 conflict resolution models. The CCCMS model presented in this paper has

	Modeling Examples (M.E)		Applications (M.A)	Publications (M.I)	Topicality (M.T)
	Concerns	Examples			
RAM	>15	2	~	3+4	2009

Table 5. "Maturity" applied to RAM

23 aspect models and 1 conflict resolution model. RAM has been published in the main conference of the domain (AOSD, [2]), plus a workshop and a master thesis [6,1]. Moreover, the AOM weaving techniques on which RAM is based on (France approach, Klein approach), have been largely published in high quality conferences or journals [30,31,23,24]. The most recent publication related to RAM was in 2009 [2].

Discussion: In [15], the authors identify several issues related to the 8 AOM approaches studied:

1. *"Missing Complex Examples"*: In [15], the authors state "*The majority of the surveyed approaches have demonstrated their techniques on the basis of rather trivial examples in which not more than two concerns are composed. In this respect, Jacobson, Cottenier et al., and Clarke et al. set a good example by demonstrating their approaches with non-trivial modeling problems.*" RAM has been demonstrated with the AspectOptima and Crisis Management System cases studies, which are large and non-trivial case studies coming from two different domains.
2. "*Lack of Application in Real-World Projects*": The approach of Pawlak et al. has been applied to 3 industrial projects. The Motorola WEAVR is currently used inside Motorola for software development. So far, RAM has not been applied in the context of a real-world project.

4.7 Tool Support

	Modeling Support (T.M)	Composition Support (T.C)	Code Generation (T.G)
RAM	√	√	🟦

Table 6. "Tool Support" applied to RAM

In order to use aspect-oriented modeling techniques to build real-world size models, tool support is essential. We have implemented a RAM tool prototype within the *Kermeta* [40] environment, an imperative, object-oriented language suitable for the specification of model transformations. Kermeta runs within the Eclipse Modeling Framework, which allows us to use the Eclipse tools to edit, store and visualize models. The RAM prototype is available online[7]. Table 6 summarizes the features our tool currently supports. To weave our class diagrams,

[7] *http://se2c.uni.lu/tiki-index.php?page=AspectOptima+Modeling*

we use the symmetric model composition technique proposed by France et al. [30,31], which has been implemented in Kermeta in a tool called *Kompose* [32]. To weave our behavior models, we use the implementation of a sequence diagram weaver proposed in [23,24].

In order to also support the weaving of state diagrams, we are currently working on replacing the Kompose and sequence diagram weaver with GeKo [38], a generic aspect-oriented model composition and weaving approach that can be used to weave any kind of model with a well-defined meta model.

Discussion: In [15], the authors comment on *"Missing Tool Support for Composition and Code Generation"* for the 8 surveyed approaches. While modeling support in many approaches is implicitly available due to the use of UML's profile mechanism, support for code generation and composition is rare. The approach of Cottenier et al. is the only one that allows for modeling, composition, and code generation. The approaches by France et al. and Klein et al. provide model weaving only. Even if the tool supporting RAM is currently only a prototype, since RAM is based on Kompose (France et al.) and the Sequence Diagram Weaver (Klein et al.), the tool can generate independent aspect models and final application models consisting of class and sequence diagrams. Once the final application model is generated, it can be mapped to code using "standard" mapping techniques. Our prototye tool does not provide this functionality, but since we output the model in XMI, it is possible to load the generated model in a standard UML tool that supports code generation. However, the behavioral models of RAM only specify message exchanges between objects, as well as optional, alternative and looping control flows. As a result, only code skeletons for all classes and operations can be generated.

When working with real-world size models, scalability of the tool itself is of great importance. Since our weaving algorithm performs composition in pairs, we do not face algorithmic challenges when augmenting the number of aspects in a design. For each additional aspect model that is instantiated, only one additional weaving step must be performed. However, the independent aspect models, and of course the final application model, grow as more aspects are added to the system. Currently, our tool requires all models, i.e. the source, the aspect and the target model, to fit into main memory.

The performance of our prototype is currently not optimal. The creation of the independent model of the *CreateMission* aspect takes several minutes. There are many factors that contribute to this slow performance:

1. The version of Kermeta we are running is interpreted. We are hoping to be able to run the prototype with the compiled version soon.
2. We are exporting, in XMI, all intermediately generated independent aspect models. This step speeds up the weaving for aspects that are reused several times, since the independent aspect model is only generated once, but slows down the weaving for aspects that are used only once. Currently, we generate all the models for debugging reasons, but our recursive weaving algorithm could be adapted to only save independent aspect models that are reused again at a later time.

3. We are outputting lots of debugging information into the Eclipse console. This step could be omitted.

5 Discussion on Reuse

A comparison criteria that has been explicitly excluded from the study presented in [15] is *reusability*. The authors argue justifiably that reusability cannot reasonably be measured without empirical studies. Reuse is one of the main strength of RAM, since RAM has been designed to allow the modeler to develop highly reusable aspect models. We have not performed in-depth studies to back up this claim with empirical evidence. However, we successfully reused aspect models within the design of the CCCMS. Using the CCCMS models as example, this section points out the features of RAM that specifically support reuse.

5.1 Encapsulation and Information Hiding

In RAM, a special UML package encapsulates all model elements that define the structure and behavior of an aspect. This aspect package is the unit of reuse. The interface of the aspect consists of the set of public operations defined by the classes in the structural view of the aspect. By hiding design decisions that are likely to change behind a well-designed interface, information hiding principles as defined by [41] can be applied to aspects, sometimes even more effectively as for objects [3].

For instance, in the CCCMS, the *ResourceSearch* aspect shown in Fig. 6 defines an interface that allows a user to create requests, to add the desired capabilities, and to perform a resource search. Internal design decisions, such as the fact that a request is implemented using a map, are hidden from the outside world.

5.2 Reuse Hierarchies

RAM supports the creation of elaborate aspect dependency chains. This makes it possible to model aspects that provide complex functionality by decomposing them into aspects that provide simpler functionality. Vice versa, aspects providing simpler functionality can be reused in several aspects of complex functionality. As a result, scattering and tangling of models can be prevented at all levels of abstraction.

5.3 Consistent Reuse with Tool Assistance

To make reuse possible, it is important that instantiations and bindings observe strict rules: if an aspect A provides a functionality whose design needs a simpler functionality provided by an aspect B, then A depends on B. In this case, and only then, A is allowed to instantiate views of B, or bind A's model elements to model elements defined in B. Circular dependencies are forbidden.

Each aspect model that defines partial classes clearly identifies those classes as mandatory instantiation parameters. To help the modeler when reusing an existing aspect model, our tool (see subsection 3.1) ensures that compatible model elements are provided for all mandatory instantiation parameters when the aspect is instantiated. Flexibility is achieved by allowing any model element to optionally be instantiated or extended through bindings, if desired. Finally, to facilitate the composition process, an aspect model can also define bindings that are used as defaults when an aspect model is instantiated. In the case where A reuses B and the modeler designing A decides to override the bindings specified in B with his own instantiation directives, our tool ensures that the new directives are compatible [2].

Thanks to the above, reusing an aspect model in a consistent way within another model is simple. Indirect dependencies of aspects are hidden from the user of an aspect: when a developer reuses an aspect A by instantiating it, our tool takes care of performing the indirect instantiations and bindings of aspects that A depends on. To fully exploit the benefits of reuse, aspect dependencies are kept unresolved until the aspects are woven with the final model. As a result, if A directly or indirectly depends on B, then a change that is made to B is automatically propagated to A when the final model is created. In case there are any conflicts between aspect models that are reused and that have been identified by the designers of the reused aspects, the tool automatically applies the appropriate conflict resolution aspect models.

Keeping aspect dependencies unresolved also facilitates maintenance: in the case where the design of a low-level aspect is improved, the new design is propagated automatically to all higher-level aspects that depend on the improved aspect when the higher models are re-woven.

5.4 Reuse in the CCCMS

Among the 23 aspect models we created for this case study, all aspects except *Initialization* and *CreateMission* are generally reusable in the sense that they do not contain any CCCMS-specific model elements. They could be reused in any application that requires the functionality that they provide. But even within the CCCMS design, reuse of aspects is happening at several abstraction levels.

Map is an example of a low-level aspect that is reused in many higher-level aspects: it is reused in *ResourceSearch*, where a request is implemented using *Map*; it is reused in *NetworkedCommand*, where a map is created that stores existing channels to communicate with remote hosts for future use; it is reused in *Workflow*, where a map is used to associate a variable to its name; and finally, it is reused in *CreateMission*, where a map is used to find the receiver queue corresponding to a given crisis. Many of the aspects grouped under the name "Generic / Design Pattern Aspects" in Fig. 14 are used frequently in other applications as well. That is the reason why most programming languages provide standard libraries that offer such functionality, especially if the functionality can easily be encapsulated in an object. An indication that our low-level aspect models are reusable is the fact that when designing the CCCMS we were actually able to reuse some of the models designed for our first big case study to which we applied the RAM approach: the AspectOptima case study [5,14]. For instance, *Map* and *ZeroToManyAssociation* are low-level aspects that are also used within the AspectOptima RAM models [2]. *ZeroToManyAssociation* was also reused in the *Observer* aspect model as well as the *Library Management System* model presented in section 4.

Within the CCCMS design, an example of an aspect that is not in the "Generic / Design Pattern Aspects" group that is reused is *Locatable*: it is instantiated several times in *Initialization* to associate a location with workers, vehicles and missions. *Locatable* was also reused in the *Library Management System* to associate an address with a customer.

If we had completed more of the design, we would have probably reused even more higher-level aspects. For example, the resource management aspects could be reused in a CCCMS that supports missions with several scenarios that can lead to their fulfillment. In this case, when a super observer orders the execution of a specific mission, the backend would have to determine the most appropriate scenario for the mission under the current crisis conditions. *ResourceSearch* could be reused to accomplish that task: the resource would be the scenario, and the capability of the resource is the kind of mission that a scenario can deal with.

Finally, the aspects grouped under "Communication Aspects" in Fig. 14 are encountered in many distributed applications. For instance, multi-player turn-based games often use the *NetworkedCommand* design to send player movements to the other players.

6 Conclusion

This paper presented an aspect-oriented design of parts of the crisis management systems case study based on the *Reusable Aspect Models* approach. In total, 23 aspect models forming a hierarchy of inter-dependent aspects were created to model the *create mission* functionality of the crisis management system backend. This experience confirms that the features of RAM allow a modeler to design aspects that provide complex functionality by decomposing them into aspects that provide simpler functionality. Vice versa, aspects providing simpler functionality can be reused in several aspects of complex functionality. As a result, scattering and tangling of models can be prevented at all complexity levels.

The crisis management system case study was specifically proposed in [12] as a case study for aspect-oriented modeling and aspect-oriented software development in general. In addition to demonstrating the power of aspect-oriented modeling during the software design phase, the results presented in this paper can be analyzed in the light of the results obtained using other AOM approaches applied to the crisis management system. Not only will this allow us to compare RAM to other aspect-oriented design approaches more accurately than we did in this paper, it will also give us more insight on how to bridge the gap between aspect-oriented approaches that work at the requirements engineering phase and approaches such as RAM that work at the detailed design phase. Understanding this transition is a key step towards the creation of an aspect-oriented software development process.

7 Acknowledgement

This work was partially supported by the SPLIT project (FNR and CNRS funding), the DiVA project (EU FP7 STREP, contract 215412, http://www.ict-diva.eu/) and the Natural Sciences and Engineering Research Council of Canada.

References

1. Klein, J., Kienzle, J.: Reusable Aspect Models. In: 11th Aspect-Oriented Modeling Workshop, Nashville, TN, USA, Sept. 30th, 2007. (September 2007)
2. Kienzle, J., Abed, W.A., Klein, J.: Aspect-Oriented Multi-View Modeling. In: Proceedings of the 8th International Conference on Aspect-Oriented Software Development - AOSD 2009, March 1 - 6, 2009, ACM Press (March 2009) 87 – 98
3. Abed, W.A., Kienzle, J.: Aspect-Oriented Modeling and Information Hiding. In: 14th Aspect-Oriented Modeling Workshop, Denver, CO, USA, Oct. 4th, 2009. (October 2009) 1–6
4. Kienzle, J., Gélineau, S.: AO Challenge: Implementing the ACID Properties for Transactional Objects. In: AOSD 2006, ACM Press (March 2006) 202 – 213
5. Kienzle, J., Bölükbaşi, G.: AspectOPTIMA: An Aspect-Oriented Framework for the Generation of Transaction Middleware. Technical Report SOCS-TR-2008.4, McGill University, Montreal, Canada (December 2008)
6. Bölükbaşi, G.: Aspectual Decomposition of Transactions. Master's thesis, School of Computer Science, McGill University, Montreal, Canada (June 2007)
7. Kienzle, J., Guelfi, N., Mustafiz, S.: Crisis Management Systems: A Case Study for Aspect-Oriented Modeling. Transactions on Aspect-Oriented Software Development **7** (2010) 1 – 22
8. Kienzle, J.: Reusable Aspect Models of the Crisis Management System: http://www.cs.mcgill.ca/~joerg/SEL/RAM_Case_Study.html (2009)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison Wesley, Reading, MA, USA (1995)
10. Riehle, D., Siberski, W., Bäumer, D., Megert, D., Züllighoven, H.: Serializer. (1997) 293–312
11. Chitchyan, R., Fabry, J., Katz, S., , Rensink, A.: Special Section on Dependencies and Interactions with Aspects. Transactions on Aspect-Oriented Software Development **5** (2009) 133 – 134
12. Kienzle, J., Guelfi, N., Mustafiz, S.: Crisis Management Systems – A Case Study for Aspect-Oriented Modeling. Technical Report SOCS-TR-2009.3, McGill University, Montreal, Canada (February 2009)
13. Rashid, A.: Aspect-Oriented Database Systems. Springer (2004)
14. Kienzle, J., Duala-Ekoko, E., Gélineau, S.: AspectOPTIMA: A Case Study on Aspect Dependencies and Interactions. Transactions on Aspect-Oriented Software Development **5** (2009) 187 – 234
15. Schauerhuber, A., Schwinger, W., Kapsammer, E., Retschitzegger, W., Wimmer, M., Kappel, G.: A survey on aspect-oriented modeling approaches. Technical report, Business Informatics Group, Vienna University of Technology (2006) http://www.wit.at/people/schauerhuber/publications/aomSurvey/.
16. Stein, D., Hanenberg, S., Unland, R.: A UML-based aspect-oriented design notation for aspectJ. In: AOSD. (2002) 106–112

17. Stein, D., Hanenberg, S., Unland, R.: Expressing different conceptual models of join point selections in aspect-oriented design. In: AOSD '06: Proceedings of the 5th international conference on Aspect-Oriented Software Development, New York, NY, USA, ACM Press (2006) 15–26
18. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming, London, UK, Springer-Verlag (2001) 327–353
19. Hanenberg, S., Stein, D., Unland, R.: From aspect-oriented design to aspect-oriented programs: tool-supported translation of jpdds into code. In: AOSD '07. (2007)
20. Pawlak, R., Duchien, L., Florin, G., Legond-Aubry, F., Seinturier, L., Martelli, L.: A uml notation for aspect-oriented software design. In: First Workshop on Aspect-Oriented Modeling with UML (AOSD'02). (The Netherlands, March 2002)
21. Pawlak, R., Seinturier, L., Duchien, L., Martelli, L., Legond-Aubry, F., Florin., G.: Aspect-oriented software development with java aspect components. chapter 16 of aspect-oriented software development. R.E. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, (2005) 343–369
22. Jacobson, I., Ng, P.W.: Aspect-Oriented Software Development with Use Cases. Addison-Wesley (2004)
23. Klein, J., Hélouet, L., Jézéquel, J.M.: Semantic-based weaving of scenarios. In: AOSD '06: Proceedings of the 5th international conference on Aspect-Oriented Software Development, Bonn, Germany, ACM (2006)
24. Klein, J., Fleurey, F., Jézéquel, J.M.: Weaving multiple aspects in sequence diagrams. Transactions on Aspect-Oriented Software Development (TAOSD) **III** (2007) 167–199
25. Cottenier, T., van den Berg, A., Elrad, T.: Joinpoint inference from behavioral specification to implementation. In: ECOOP 2007, Germany, July 30 - August 3. LNCS, Springer (2007) 476–500
26. Cottenier, T., Berg, A.V.D., Elrad, T.: The motoroal weavr: Model weaving in a large industrial context. In: Industry Track of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06), Bonn, Germany, ACM (2006)
27. Aldawud, O., Elrad, T., Bader, A.: Uml profile for aspect-oriented software development. In: 3rd International Workshop on Aspect Oriented Modeling (In conjonction of AOSD'03). (Boston, Massachusetts, March 2003)
28. Elrad, T., Aldawud, O., Bader, A.: Expressing aspects using uml behavioral and structural diagrams. chapter of the book aspect-oriented software development. R.E. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, (2005)
29. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design: The Theme Approach. Number ISBN: 0-321-24674-8. Addison Wesley (2005)
30. France, R., Ray, I., Georg, G., Ghosh, S.: Aspect-oriented approach to early design modelling. IEE Proceedings Software (August 2004) 173–185
31. Reddy, R., Ghosh, S., France, R.B., Straw, G., Bieman, J.M., Song, E., Georg, G.: Directives for composing aspect-oriented design class models. Transactions on Aspect-Oriented Software Development (TAOSD) **I** (2006) 75–105
32. Fleurey, F., Baudry, B., France, R., Ghosh, S.: A generic approach for automatic model composition. In: 11th Workshop on Aspect-Oriented Modeling, AOM at Models'07. (2007)
33. OMG: Uml superstructure, v2.1.1. OMG Document number formal/07-02-05
34. Whittle, J., Araújo, J.: Scenario modelling with aspects. IEE Proceedings - Software **151**(4) (2004) 157–172

35. Whittle, J., Jayaraman, P.: Mata: A tool for aspect-oriented modeling based on graph transformation. In: 11th Workshop on Aspect-Oriented Modeling, AOM at Models'07,. (2007)
36. Morin, B., Klein, J., Kienzle, J., Barais, O., Jézéquel, J.M.: Geko: A generic aspect model weaver. Transactions on Aspect Oriented Software Development (TAOSD), Under Submission (major changes) (2010)
37. Klein, J., Kienzle, J., Morin, B., Jézéquel, J.M.: Aspect model unweaving. In 5795, L., ed.: In 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009), Denver, Colorado, USA (2009) p 514–530
38. Morin, B., Klein, J., Barais, O., Jezequel, J.M.: A generic weaver for supporting product lines. In: Early Aspects Workshop at ICSE, Germany, ACM (2008)
39. Harrison, W.H., Ossher, H.L., Tarr, P.L., Harrison, W.: Asymmetrically vs. symmetrically organized paradigms for software composition. Technical report, Research Report RC22685, IBM Thomas J. Watson Research (2002)
40. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: MODELS/UML'2005. LNCS, Springer
41. Parnas., D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM **15(12)** (1972) 1053–1058