Jadex Active Components Framework

BDI Agents for Disaster Rescue Coordination

Lars BRAUBACH [a], Alexander POKAHR [a]
[a] *Distributed and Information Systems Group*
Department Informatik, Universität Hamburg
Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany
{braubach|pokahr}@informatik.uni-hamburg.de

AbstractIn this chapter the Jadex framework is presented that aims at supporting the construction of distributed and concurrent applications. Main conceptual entity in Jadex is an active component that combines properties of agents with software components in order to strengthen the software technical means for developing applications. The conceptual foundations of Jadex as well the advantages of active components will be described in a practical way by using a disaster management scenario as a running example. Concretely, the notions of virtual environment, active components as well as BDI agents are introduced to show how a simulation scenario can be built, in which different kinds of rescue forces are coordinated to handle disasters in a cooperative way.

Keywords. virtual environments, simulation, service component architecture (sca), active components, software agents, BDI architecture

1. Motivation

The development of distributed applications is intricate due to a number of inherent characteristics of those systems. One fundamental reason for the increased complexity is that separated network nodes lead to implicit concurrency and additionally require message based communication mechanisms. Implications of concurrency and inter-address space communcation are many new error sources, some of which are extremely difficult to handle and also cannot completely be masked by distribution transparency.

In order to simplify the construction of distributed systems the Jadex framework proposes conceptual abstractions for dealing with distribution and concurrency by combining ideas from agent and software component orientation. Core concept is a so called *active component*, which is similar to an agent as it represents an autonomous entity that has control about its state and execution. Similar to a component it is seen as a *service provider* and *consumer*, which may interact with other components by using their public services. Thus, on the one hand active components are a first-class and natural abstraction for concurrency because they are executed independently of each other. On the other hand the notion of provided and required services for active components establishes a foundation for systematic software architectures with clear inter-component dependencies. Additionally, the service based interrelationships between components facilitate the

composition of basic components to composite components and foster modularity and reusability of system parts.

On basis of these foundations Jadex offers an open source middleware software solution, which consists of a runtime infrastructure as well as an extensive tool suite. The runtime infrastructure is a platform that allows applications to be simulated as well as executed without requiring code changes in components when switching execution modes [10]. The platform is capable of running different component types, e.g. complex BDI (belief-desire-intention) reasoning agents or BPMN (business process modeling notation) workflows. Each component type is characterized by its *internal architecture*, which determines the programming abstractions for the component, e.g. beliefs, plans and goals in case of a BDI agent. The execution logic of such an internal architecture is realized in a *kernel*, so that new component types can easily be added by just providing a new kernel for the new type. The tool suite consists of development and runtime tools, whereby development tools are kernel specific so that e.g. BDI agents can be built relying on a standard integrated development environment (IDE) but for BPMN workflows a new graphical modeling tool is provided. Runtime tools mainly serve management and debugging purposes including e.g. a starter tool for starting and stopping applications and a debugger tool that can be used for executing components in a stepwise manner and introspect their state.

This chapter will introduce the conceptual foundations of Jadex and explain these concepts relying on a consistent example scenario. Therefore, in the following Section 2 a specific disaster management scenario is introduced. The system design then first tackles the environment modeling aspects in Section 3 and highlights how the virtual example environment can be described. Thereafter, in Section 4 active components and component services are introduced and will be used to model the main scenario component types as well as their service interrelationships. In Section 5 details about the Jadex BDI agent architecture are explained and it is shown how complex rescue force coordination can be managed using a BDI coordinator agent. Finally, a conclusion and an outlook is given in Section 6.

2. Disaster Management Scenario

Figure 1 shows the AML agent diagram of the disaster scenario. The presented disaster scenario targets the coordination between disaster *Rescue Forces* such as *Fire Brigades* and *Ambulances*. Distributed *Stations* exist for hospitals as well as fire departments, where each of them has its own fleet of vehicles (*Rescue Force*s). Different types of vehicles are needed at different *Disaster* sites that may unexpectedly occur in the environment. Main task of the planned disaster management system consists in coordinating the rescue forces for handling disasters in an effective and efficient manner. Hence a *Commander Agent* is responsible for handling disasters by assembling and controlling a *Rescue Team*. Such a rescue team may consist of an arbitrary number of *Rescuers*, i.e. ambulances and fire brigades.

2.1. Scenario Details

For simulation purposes several simplifications of the scenario sketched above have been introduced. It is assumed that only the following kinds of disasters can occur:

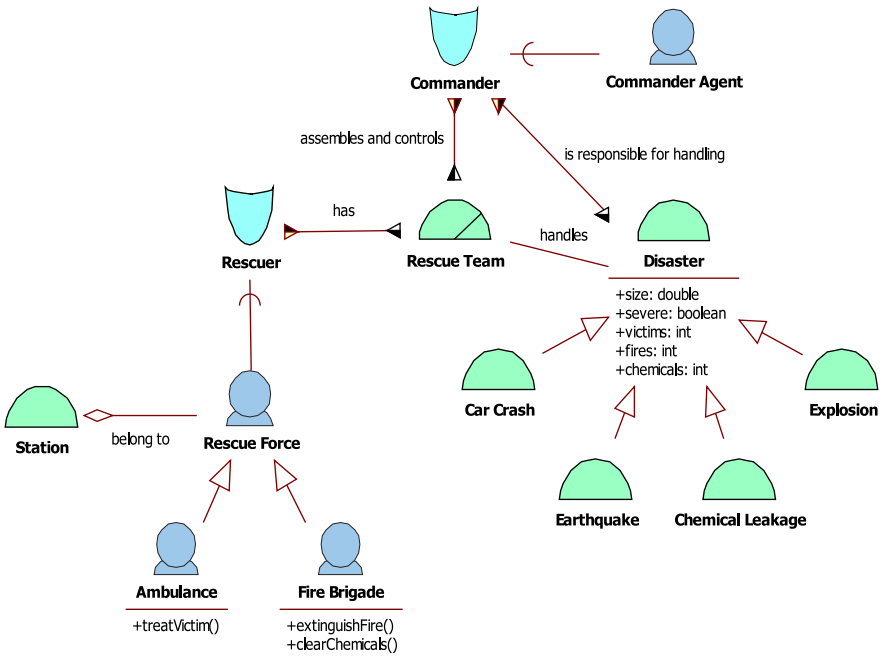

Figure 1. AML agent diagram for disaster management components

Car Crashes, *Earthquakes*, *Chemical Leakages* and *Explosions*. Each disaster instance is characterized by a set of common properties, namely the number of victims, fires and chemicals. Additionally, it has a specific size and is classified as normal disaster or severe disaster, whereby severe disasters have priority over normal disasters with respect to their resolution, i.e. if a severe disaster occurs and not enough rescue forces are available, they may be discharged from the current task and assigned a new one related to the severe disaster. In order to handle a disaster, fire brigades and ambulances are used. A fire brigade can clear chemicals as well as extinguish fires and ambulances have the capability to transport victims one by one to a nearby hospital. Handling disasters is a complex coordination problem as it has to be determined which forces should be sent to which disasters at what time, possibly needing to disrupt ongoing tasks in case of new severe problems. In the simplified scenario used here it is assumed that the number of victims, fires and chemicals is an important factor for disaster resolution. Even if one force of each kind can in principle handle a disaster this would be far from optimal with respect to the amount of time that is needed. The resolution of a disaster is done much faster, when more than one rescue force is used but with the limitation that no speedup is gained in case the number of forces for a given task exceeds its current number of victims, fires or chemicals (e.g. if three fires exist, three fire bridges will extinguish them three times faster than one fire brigade but four brigades will be as fast as three). A further constraint of disaster resolution is that for safety reasons all chemicals have to be cleared before ambulances can start treating victims at the disaster site.

3. Environment Support

Jadex applications are often placed in complex and highly dynamic distributed environments like the disaster management scenario described above. Such applications are composed of independently executing active components that interact with each other and with the surrounding environment. As a result, the behavior of these applications becomes hard to design and predict using traditional software engineering approaches. Instead, simulation approaches can be used for analyzing system behavior under specific conditions and also for benchmarking different behavioral strategies against each other. Thus, for testing purposes, applications need to be executed in specifically designed virtual environments. Once, an implementation has been thoroughly tested using simulations, it can be deployed in the real environment.

To support this common use case of developing both a simulation and a deployment version of an application, the Jadex framework features an environment support ("EnvSupport" for short). EnvSupport is a set of APIs, framework classes and tools to facilitate the development of simulation applications and achieves the following design objectives:

- Clean separation of environment and active component implementations
- Easy building and configuration of virtual environments for testing
- Provision of tools, e.g. for observing running simulations or collecting and analyzing data

The clean separation of the environment from other application components fosters an easy transition to a deployed system, because the component implementations do not need to be changed in the process. The EnvSupport framework further provides many ready to use classes for typical environments like continuous or grid-based 2D virtual worlds. Thus test environments for applications can quickly be constructed from existing framework classes. Environments are described in declarative XML files that simplify configuration and thus allow quickly changing environment parameters for testing applications in different scenarios. Moreover, tools are provided that allow observing the current state of 2D environments in graphical views. To this end, environment configurations may include hints for graphical representation of environment elements (e.g. geometric shapes and textures), which allows fine-tuning the graphical representation of the environment. Also, the data produced during simulation runs can be collected, saved to files and/or rendered in charts for analysis. In the following, the features of EnvSupport are shortly introduced. A more detailed introduction can be found in [5].

3.1. EnvSupport Realization and Usage

The basic building blocks of the EnvSupport framework are depicted in Figure 2. The environment itself is modeled as a so called *Space* (Figure 2 left). Jadex applications may contain an arbitrary number of spaces, which can represent besides virtual environments also other applications structures like organizational models for e.g. assigning roles to application components (cf. [8] for more details on spaces). The environment space contains *Domain* constructs (left upper area) as well as *Interaction* constructs (left lower area). The domain constructs allow defining environment objects (*Space Objects*), e.g. ambulances, victims, disasters etc., which together represent the current state of the

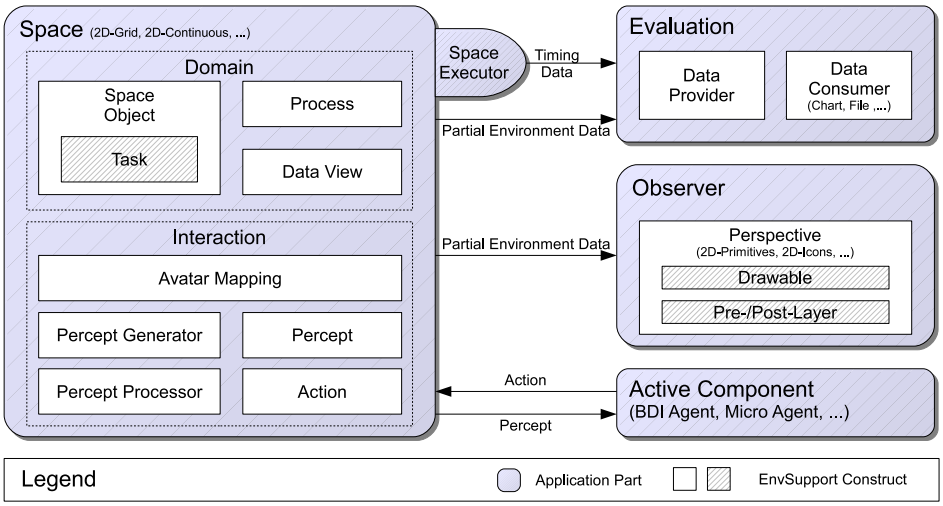

Figure 2. Building blocks of the EnvSupport framework

environment. An object may have one or more associated *Tasks* for defining some currently ongoing behavior of the object (e.g. movement of an ambulance). In addition to tasks, *Processes* allow representing global behavior of the environment (e.g. random occurrence of new disasters). The *Space Executor* of the environment is responsible for executing the tasks and processes based on a specific execution model (e.g. round-based or continuous). For evaluation and visualization purposes it is often helpful to restrict the set of considered environment objects (e.g. focusing on ambulances and ignoring fire brigades). For this purpose *Data Views* can be defined to select a specific subset of environment objects based on declarative queries.

Interaction constructs (left lower area of Figure 2) establish the relation between environment objects and application components. The *Avatar Mapping* defines a one-to-one relation between active components and space objects, i.e. allowing a component to be represented in the environment as a space object. The avatar mapping serves two purposes. First, the existence of components and space objects can be linked to each other. In this respect, the developer can specify if corresponding space objects should be automatically created and destroyed for each created or destroyed component and vice versa. Secondly, the avatar mapping influences the percepts and actions that are available to a component as explained in the following. *Percepts* represent information about changes in the environment that are passed to interested components. Each percept type declares the type(s) of space objects that can cause the percept and the type(s) of components that can observe the percept. *Percept Generators* are responsible for creating percepts based on the declared percept types. Custom percept generators can be implemented to serve specific application requirements, but common use cases are also provided in predefined framework classes, such as a default percept generator that automatically creates percepts for all environment changes inside a definable visual field of an avatar. As components might be implemented using one of the different kernels provided by Jadex (e.g. BDI reasoning agents vs. simple micro agents), *Percept Processors* are used to feed the generated percepts flexibly into the application logic of the components. Again, custom implementations can be provided as needed, yet common use cases are readily available,

like automatically updating definable beliefs of BDI agents according to specific percepts. Finally, components may schedule *Actions* in the environment. Actions are implemented as Java classes that may apply arbitrary changes to the environment state (e.g. changing properties of space objects, creating/destroying objects, etc.). The *Space Executor* executes the scheduled actions along with the tasks and processes of space objects as described above. Moreover, the executor invokes the percept generators and percept processors for propagating environment changes to application components.

A developer can choose to collect data during simulation runs using the *Evaluation* facilities of EnvSupport (upper right of Figure 2). *Data Providers* allow defining the concrete data items to be collected as well as specifying collection intervals and aggregation functions. *Data Consumers* use the collected data for e.g. rendering charts while the simulation is running and/or writing collected data to disk for later analysis. To get a visual feedback of a running simulation, the *Observer* (Figure 2, right) produces a continuously updated 2D view of the current environment state. The developer can define so called *Perspectives* that are visual representations of the environment. A perspective mainly consists of *Drawables*, which assign visualizations to space objects. The drawable for a space object can be composed from arbitrary many drawing primitives (e.g. geometric shapes and external images). Drawing primitives can be further parametrized using properties of the underlying space object, such that the state of a space object can be used to influence its visual appearance. A perspective can further include *Pre-* and *Post-Layers* to add further visual elements (e.g. a map or a grid) that do not correspond to specific space objects.

3.2. Scenario Environment

For the disaster management application the Jadex EnvSupport was used to build an environment for testing the behavior of the commander agents. This environment can be configured in various aspects, e.g. regarding the number and location of rescue force stations as well as the numbers of available vehicles, frequency and size of disasters, etc. The environment further allows visually following the operations of the rescue forces on a map and inspecting statistical data about the efficiency of the system in graphical charts.

3.2.1. Scenario Domain

Figure 3 shows the XML-based definition of the domain elements of the disaster management scenario. The environment is situated in a continuous 2D area of size 1x1 for simplicity (line 1). If the application should be tested for a concrete city map, the scale can be adapted to better match the real dimensions. The available types of space objects are defined in the *objecttypes* section (lines 2-17). Each type definition contains the supported properties of the object as well as optionally the default value for each property. E.g. the *disaster* type is defined in lines (3-10) and has properties as modeled in Figure 1 (*severe*, *size* etc.). The specific disaster subtypes ('Car Crash', ...) are captured in the *type* property (line 4) to avoid having to define identical space object types for each disaster subtype. All space objects automatically have a property for their location, and often space objects do not need any further properties, like the *firestation* type (line 11). The *firebrigade* definition (lines 12-15) shows the use of a default value. Each fire brigade has a speed property, which defaults to 0.05 (line 13). Thus all fire brigade instances in

```
1  <e:envspacetype name="2dspace" class="ContinuousSpace2D" width="1"height="1">
2    <e:objecttypes>
3      <e:objecttype name="disaster">
4        <e:property name="type" class="String"/>
5        <e:property name="severe" class="boolean"/>
6        <e:property name="size" class="int"/>
7        <e:property name="victims" class="int"/>
8        <e:property name="fire" class="int"/>
9        <e:property name="chemicals" class="int"/>
10     </e:objecttype>
11     <e:objecttype name="firestation"/>
12     <e:objecttype name="firebrigade">
13       <e:property name="speed" class="double">0.05</e:property>
14       <e:property name="state" class="String"/>
15     </e:objecttype>
16     ...
17   </e:objecttypes>
18   <e:tasktypes>
19     <e:tasktype name="move" class="MoveTask" />
20     <e:tasktype name="extinguish_fire" class="ExtinguishFireTask" />
21     ...
22   </e:tasktypes>
23   <e:processtypes>
24     <e:processtype name="create" class="DefaultObjectCreationProcess">
25       <e:property name="type">"disaster"</e:property>
26       <e:property name="timerate" dynamic="true">
27         DisasterType.getExponentialSample(30000)
28       </e:property>
29       <e:property name="properties" dynamic="true">
30         DisasterType.generateDisaster()
31       </e:property>
32     </e:processtype>
33   </e:processtypes>
34   <e:avatarmappings>
35     <e:avatarmapping objecttype="firebrigade" componenttype="FireBrigade"
36       createavatar="false" createcomponent="true"/>
37     ...
38   </e:avatarmappings>
39 </e:envspacetype>
```

Figure 3. Domain elements of disaster management environment

the simulation will move with this default speed, unless the speed property is specifically set to a different value for some instance.

The behavior of the space objects is captured in tasks. In the disaster management scenario only the vehicles (ambulances and fire brigades) exhibit individual behavior. The tasks are defined in the *tasktypes* section (lines 18-22). The *move* task (line 19) handles movement of a vehicle according to its speed and a chosen destination. The task is implemented in a Java class as explained later. Fire brigade objects can further perform the *extinguish_fire* task (line 20), which continuously reduces the amount of fire of a nearby disaster object. In case all fires have been extinguished and also no chemicals and victims are present the task removes the resolved disaster object from the space. Further similar tasks (e.g. clear chemicals and treat victims) are omitted for brevity. The global behavior of the disaster management environment is described using processes (lines 23-33). Here, a single process is defined that randomly creates new disaster objects in the environment. The process implementation is the generic framework class *DefaultObjectCreationProcess* (line 24), that can be used to create arbitrary kinds of objects using a configurable object *type* (line 25), *timerate* (line 26-28), and object *properties* (lines 29-31). In the scenario, the time between two disasters and the disaster properties are

```
1  public class MoveTask extends AbstractTask
2  {
3    public void execute(IEnvironmentSpace space, ISpaceObject obj,
4       long progress, IClockService clock)
5    {
6      IVector2 destination = (IVector2)getProperty("destination");
7      IVector2 loc = (IVector2)obj.getProperty(Space2D.PROPERTY_POSITION);
8      double speed = ((Number)obj.getProperty("speed")).doubleValue();
9      IVector2 direction = destination.copy().subtract(loc).normalize();
10     double dist = ((Space2D)space).getDistance(loc,destination).getAsDouble();
11     double maxdist = progress*speed*0.001;
12     IVector2 newloc = dist<=maxdist ? destination
13       : direction.multiply(maxdist).add(loc);
14     ((Space2D)space).setPosition(obj.getId(), newloc);
15     if(newloc==destination)
16       setFinished(space, obj, true);
17   }
18 }
```

Figure 4. Implementation of the move task

randomly generated using static methods of the helper class *DisasterType* (lines 27 and 30). The timerate is drawn from an exponential distribution with an average of 30000 milliseconds, while the disaster properties are based on specific probabilities, e.g. for disaster type and corresponding numbers of victims, etc.

Finally, in the *avatarmappings* section (lines 34-38), the space objects of the vehicles (fire brigade and ambulance) are mapped to concrete component types. The mapping definition for the fire brigade (lines 35-36) shows that an application component of type *FireBrigade* should be created for each *firebrigade* space object as specified by the *createcomponent* attribute. Therefore, in the scenario configuration as explained later, one can simply add or remove fire brigade objects that subsequentially lead to automatic creation of corresponding application components.

Figure 4 shows the Java class implementing the move task. The class extends the framework class *AbstractTask* (line 1) and overrides the *execute()* method (lines 3-17), which is repeatedly called by the space executor until the task is marked as finished. First, the target *destination* value of the task instance as well as the *loc*(ation) and *speed* of the space object are retrieved (lines 6-8). Based on these values, the *direction* from the current location to the destination is calculated (line 9) as well as the *dist*(ance) to the destination (line 10). The *maxdist* value (line 11) represents the maximal distance the vehicle could have moved in the available time, incorporating the *progress* of time since the move task was last executed. The new location *newloc* is calculated by multiplying the direction vector with the maximal movement distance, unless the vehicle already reaches the destination with less movement (lines 12-13). Finally, the new location is set as a property of the vehicle (line 14) and if the destination is reached, the task is marked as finished (lines 15-16).

3.2.2. Scenario Visualization

The last section has shown how to define the data and behavior of the disaster management environment. If only statistical data of simulation runs is required for analyzing the application performance, there is no need for a visualization at all. Yet, immediate visual feedback of running simulations is usually an indispensable help during building

```
1  <e:perspective name="icons" class="Perspective2D" opengl="true">
2    <e:drawable objecttype="disaster" width="0.08" height="0.08">
3      <e:property name="drawsize" dynamic="true">
4        new Vector2Double($object.getProperty("size").intValue()*0.005)
5      </e:property>
6      <e:ellipse layer="1" size="drawsize" abssize="true" color="#FAFA1E99">
7        <e:drawcondition>!$object.getProperty("severe")</e:drawcondition>
8      </e:ellipse>
9      <e:ellipse layer="1" size="drawsize" abssize="true" color="#FA1E1E99">
10       <e:drawcondition>$object.getProperty("severe")</e:drawcondition>
11     </e:ellipse>
12     <e:texturedrectangle layer="2" height="1" width="1"
13        imagepath="images/carcrash.png">
14       <e:drawcondition>
15         $object.getProperty("type").equals("Car Crash")
16       </e:drawcondition>
17     </e:texturedrectangle>
18     ...
19     <e:text layer="3" x="0.04" y="-0.02" size="6" font="Arial"
20        text="victims: $victims$\nfire: $fire$\nchemicals: $chemicals$"
21        abssize="true" align="left" color="black"/>
22   </e:drawable>
23   <e:drawable objecttype="firestation" width="0.1" height="0.1">
24     <e:texturedrectangle layer="4" height="1" width="1"
25        imagepath="images/firestation.png"/>
26   </e:drawable>
27   <e:drawable objecttype="firebrigade" width="0.05" height="0.05">
28     <e:texturedrectangle layer="3" height="1" width="1"
29        imagepath="images/firebrigade.png"/>
30     <e:texturedrectangle layer="3" height="0.4" width="0.4"
31        imagepath="images/beacon.png" x="-0.2" y="-0.35">
32       <e:drawcondition>
33         "moving_to_disaster".equals($object.getProperty("state"))
34       </e:drawcondition>
35     </e:texturedrectangle>
36     ...
37   </e:drawable>
38   ...
39   <e:prelayers>
40     <e:tiledlayer width="1" height="1" imagepath="images/map.png" />
41   </e:prelayers>
42 </e:perspective>
```

Figure 5. A perspective for the disaster management environment

and debugging of the application, as well as for appropriately configuring the simulation to resemble realistic behavior.

As shown in Figure 5, a visual perspective for the disaster management scenario can be quickly defined. One only has to define a drawable for each space object that should be visible in the perspective. Here, drawables are defined for disasters (lines 2-22), fire stations (lines 23-26) and fire brigades (lines 27-37). Similar drawables are defined for ambulances and hospitals (omitted for brevity). If the appearance of a space object is static (i.e. does not depend on the properties of the object), the drawable definition is usually quite simple. E.g. the fire station is represented by an icon (*texturedrectangle*) loaded from the external image *firestation.png* (lines 24-25). The drawable for the disaster is more complex, as it calculates a drawing size based on the size property of the disaster object (lines 3-5) and uses this size to draw a circle (*ellipse*) representing the disaster area (lines 6-11). Further, the disaster drawable chooses from a set of different icons, based on the disaster type. E.g. the icon for a car crash is defined with a corresponding *drawcondition* to match the type property (lines 12-17). Finally, the visual disaster rep-

```
1  <e:dataproviders>
2    <e:dataprovider name="statistics">
3      <e:source name="$fire" objecttype="disaster" aggregate="true">
4        $object.fire
5      </e:source>
6      ...
7      <e:data name="time">$time</e:data>
8      <e:data name="fire">SFunction.sum($fire)</e:data>
9      ...
10   </e:dataprovider>
11 </e:dataproviders>
12 <e:dataconsumers>
13   <e:dataconsumer name="statistics_chart" class="XYChartDataConsumer">
14     <e:property name="dataprovider">"statistics"</e:property>
15     <e:property name="title">"Disaster Statistics"</e:property>
16     <e:property name="maxitemcount">500</e:property>
17     <e:property name="legend">true</e:property>
18     <e:property name="seriesname_0">"Fire"</e:property>
19     <e:property name="valuex_0">"time"</e:property>
20     <e:property name="valuey_0">"fire"</e:property>
21     ...
22   </e:dataconsumer>
23 </e:dataconsumers>
```

Figure 6. Evaluation settings for the disaster management environment

resentation includes text fragments denoting the current numbers of fires, chemicals and victims (lines 19-21). Similarly to the disaster drawable, the visual representation of the fire brigade adapts itself to the properties of the fire brigade object, e.g. by displaying a beacon, when the fire brigade moves towards a disaster (lines 30-35). Besides the individual space objects, the environment itself is also visually represented by using a so called *prelayer* (lines 39-41), which in this case displays a map of an area. A screenshot how the perspective looks like during a simulation run is shown in Section 5.2.3 in Figure 20 (right).

3.2.3. Scenario Evaluation

As described in Section 3.1, an application description may include an evaluation section to specify, which data should be collected during simulation and how this data should be presented. Figure 6 shows the evaluation settings for the disaster management application. The collection of data is specified using *dataproviders* (lines 1-11) while the presentation of data is defined as *dataconsumers* (lines 12-23). The output of a data provider is similar to a relational database table, i.e. the *data* entries (lines 7 and 8) represent the columns of the produced table, and for each simulation time point a row is added to this table. The input of the data entries is based on *sources* for fire (lines 3-5), victims and chemicals (not shown), as well as predefined values like the current simulation time. Thus the data provider has one column for the current simulation time (line 7) for the current number of fires as a sum over the fires of all current disasters (line 8) and for the sums of victims as well as chemicals (not shown).

The application uses a chart data consumer (line 13) to present the collected data. The chart is based on data from the previously defined data provider (line 14) and has some properties to influence the visual appearance (lines 15-17), i.e. displaying a title and a legend and restricting the amount of plotted information to the last 500 data rows. A chart data consumer can plot multiple data series at once, each of which requires a

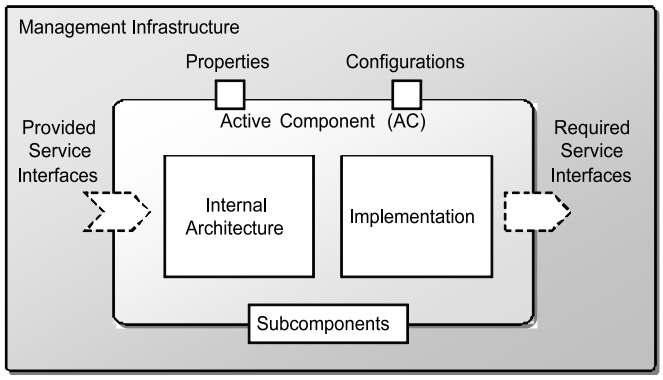

Figure 7. Active component conceptual view

name, as well as inputs for a X and Y values. E.g. a series is specified for displaying information about fires (line 18) having the current time as X value (line 19) and the current number of fires as Y value (line 19). The resulting chart is shown in Section 5.2.3 in Figure 20 (middle).

4. Active Components and Services

In Figure 7 the concepts of an active component are shown. This view is similar to the definition of a component in the service component architecture (SCA) [6] with some substantial differences. One main aspect of an active component that is shared with nearly all existing component models is the explicit definition of *provided and required services.* On the one hand this makes the functional component dependencies explicit (required services) and on the other hand it also clearly states what can be functionally expected from the component (provided services). Describing components with required and provided service is a necessary precondition for building modular and reusable software applications as component dependencies become visible and therefore manageable on an architectural level. Hence, adding the notion of services to agents facilitates the construction of complex applications in a sound software technical way.

The active component model is also hierarchical meaning that composite components can be constructed from basic components by wiring required and provided services. A composite component may thus be a parent of an arbitrary number of *subcomponents*. A component can be configured from the outside using *properties* and *configurations.* Properties are a way to set specific argument values individually and configurations represent a set of predefined argument values that can be referred to via a user given name. The most obvious difference of an active component with regard to other component definitions is that it is an autonomously executing entity similar to an agent. Its behavior control is determined by an *internal architecture*, which constitutes the available conceptual abstractions for programming the autonomous behavior of an active component.

```
a) + <methodname>(<param>[0..*]): void
b) + <methodname>(): <type>
c) + <methodname>(<param>[0..*]): <futuretype>
```

Figure 8. Allowed service method signature types

4.1. Services

The publicly available functionality of an active component is defined by an arbitrary number of provided services. A service is defined via an interface specification, which allows object oriented access to the service functionality and further allows locating services of a given type. In addition to the interfaces active components also contain the concrete service implementations that realize the underlying domain logic. Typically, service implementations belong to the component and are executed decoupled from the caller on the component thread, but services can also process requests directly so that the enclosing active component is not involved at all.

The active components paradigm imposes an important constraint on service interface specifications as it is mandatory that no interaction between active components ever blocks in order to already conceptually avoid deadlocks. As direct consequence it is required that all method signatures are asynchronous, i.e. the service caller should never be blocked when invoking a service. To meet this objective method signatures can be defined in three different ways: a) with no return value, i.e. void, b) as special case also with a constant return value or c) with a future return value. These different kinds of method signatures are schematically shown in Fig. 8. The first case allows asynchronous invocation as the caller does not need to wait for a result. The second case only applies to methods, which always return the same value (typically only methods without arguments). Here, the constant return value can be cached in advance and immediately returned on invocation without blocking the caller. In the general case a method provides a return value that cannot be known in advance. In order to achieve asynchronous calls with non void return values, the third variant with a future return value can be used.

A future represents the result of an asynchronous computation in the sense that a method call immediately returns the future to the caller, but the computation result may be provided later. Normally, futures realize a *wait-by-necessity* scheme what means that a future blocks the caller in the moment when it needs to access the result of the call and it has not yet been provided. This deferred waiting does not prevent deadlocks so that the future concept has been extended using a callback mechanism. Instead of directly fetching the return value the caller can add a result listener that is notified as soon as the value is set by the callee. This allows avoiding blocking calls completely and conceptually prevents technical deadlocks.[1]

[1] This does not mean that components cannot 'deadlock' on the application layer by endlessly waiting on each other, but the technical deadlock avoidance also in this case guarantees that such a 'deadlocked' component remains responsive and can handle further service or message requests by executing corresponding domain logic.

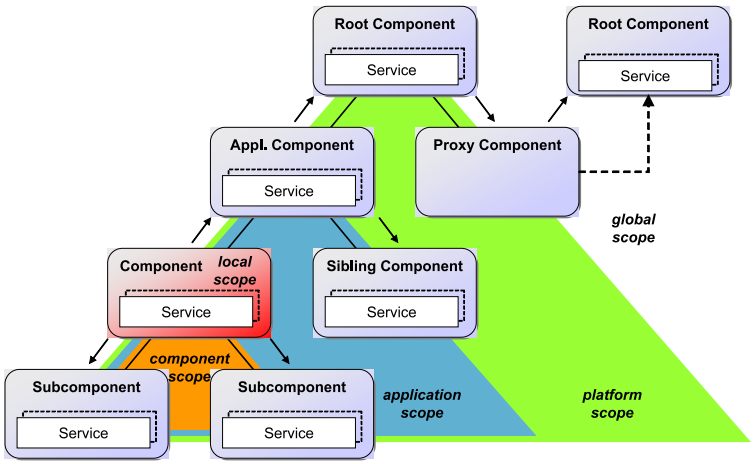

Figure 9. Search scopes

4.2. Composition

Active components make use of primitive and composite components. A composite component can be composed of arbitrary many subcomponents that are either primitive or composite components again. The composition of components is done using required and provided services, which can be interconnected. Depending on this specification a required service can be bound to a service instance of a directly known component but also to a service instance that will be searched at runtime and selected according to specified service requirements. It is also possible to include subcomponents without service relationships in the composite component. The reason is that components are active and may perform autonomous computations without being controlled by the composite element. Thus, in certain cases it makes sense having subcomponents that cannot be accessed via services from the outside as part of a composite component.

As active components are meant to support the construction of complex distributed systems service bindings can typically not assumed to be static so that especially dynamic composition aspects gain importance. Dynamic composition relies on service search, which is commonly supported by centralized registries that can be used to store and search service entries, e.g. in case of web services UDDI registries can be used. Although such an approach can be used for active components as well, we propose a decentralized mechanism without central service repository. The mechanism is based on the observation that all active components are possible service providers and their hierarchical composition structure can be exploited for searching. The rationale behind this assumption is that proximity is often an important factor for estimating the service usefulness, i.e. the nearer a service is the more relevant it probably is.

Figure 9 shows five different scopes, which can be used to control the search. If *local scope* is used only declared services of the component itself will be considered. In contrast, using *component scope* includes also subcomponents and *application scope* further extends the search to all components of the same application. In some cases this is not sufficient so that the search scope can be further expanded to include all services of the platform (*platform scope*) or even all connected remote platforms (*global scope*).

```
provided service =
  interface:Interface implementation:BasicService [direct:boolean]

required service =
  interface:Interface name:String [multiple:boolean] [dynamic:boolean] [scope:String]
```

Figure 10. Provided and required service definitions

Having described how services can be located dynamically now the composition approach can be explained in more detail. The specification properties for provided and required services are listed in Figure 10. It can be seen that a provided service consists of an *interface* as well as a service *implementation* of type *BasicService*, which is a predefined framework class that has to be extended. Additionally, the boolean *direct* flag can be used to state that service calls should not be executed on the enclosing active component thread but directly on the caller thread. Per default all calls are automatically executed as part of the enclosing active component so that service implementations can safely access component internals without consistency risks caused by concurrent thread accesses.

A required service is also characterized by its interface. Furthermore, it has a component widely visible *name*, which can be used to fetch a service implementation using the *getRequiredService(name)* framework method. As it is a common use case that several service instances of the same type are needed the *multiple* declaration can be used. In this case it is obligatory to fetch the services via *getRequiredServices(name)*. Service binding is performed according to the *dynamic* and *scope* properties. Is a required service declared to be dynamic it will not be bound at all but a fresh search is performed on each access. The scope properties allow to constrain the search to several different areas as introduced above, i.e. when scope is set to application the search will not exceed the bounds of the application components.

4.3. Application Description

Jadex applications are described using an XML descriptor file. Basically, this file allows declaring component types, environment spaces as well as application instances. Definitions of component types include a logical component type name as well as the file name of the component implementation. In this way application descriptions are independent of the concrete component types and it becomes possible to set up heterogeneous applications consisting of different kinds of components, e.g. BDI agents and BPMN workflows. As introduced in Section 3, Jadex also allows different environment types being used in concert with components. A space type is also defined by a logical name and an implementation class but typically allows further space elements being declared. These elements are defined relying on a space type dependent XML schema that is included as separate namespace in the application descriptor (e.g. 'e:' is used in the example as identifier for the EnvSupport namespace).

In addition to type related information also concrete application instances can be defined. In general, an application instance may consist of an arbitrary number of space and component instances. Both are defined by referring to the respective logical type name from the application model and may declare an instance name. Space instance def-

```
1  public interface ITreatVictimsService extends IService
2  {
3    public IFuture treatVictims(ISpaceObject disaster);
4    public IFuture abort();
5  }
6
7  public interface IExtinguishFireService extends IService
8  {
9    public IFuture extinguishFire(ISpaceObject disaster);
10   public IFuture abort();
11 }
12
13 public interface IClearChemicalsService extends IService
14 {
15   public IFuture clearChemicals(ISpaceObject disaster);
16   public IFuture abort();
17 }
```

Figure 11. Disaster management service definitions

initions are space type dependent. In case of the EnvSupport space it is possible to create instances of element types of the model, e.g. space objects, processes as well as data providers and consumers. For a component instance, arguments, a start configuration, and a number can be specified. Arguments can be employed to pass values to component instances, whereby the instance name is just the name for the component created. The start configuration allows creating a component with a predefined setting and the number states how many components of the same type will be initialized.

4.4. Scenario Architecture

The basic scenario design consists of agent types for the different rescue forces as well as for the commander, which has the task of resolving disasters by coordinating rescue forces. This coordination is based on the component services of the rescue forces, which reflect their respective capabilities (cf. Figure 11). An ambulance offers an interface *ITreatVictimsService* (lines 1-5) that can be used to instruct the rescue force to start treating injured people at a specific disaster site. For this purpose the interface offers the *treatVictims()* method, which takes the disaster space object as parameter and returns a future which indicates when patient treatment has finished and the ambulance can be assigned to another task. In order to support also the reassignment of units an additional *abort()* method is available, which tells an ambulance to stop working on the current disaster site. Again, the rescue force indicates its availability via the future return value. A fire brigade exposes two interfaces. One called *IExtinguishFireService* (lines 7-11), which can be used to instruct a brigade to extinguish fires at a specific disaster site and another one called *IClearChemicalsService* (lines 13-17) for working on chemical problems. Both interfaces are syntactically similar to the first one and also offer an abort method for task cancellation. It has to be noted that each rescue force can work on one task at the same time only, even if it offers more than one service. In case a unit is instructed to work on a task while it is busy it is expected to signal an exception to the caller via the future return value.

The disaster management application descriptor is shown in Figure 12. It brings together the different parts of the application. On the one hand it can be seen that the appli-

```
1  <applicationtype ... name="DisasterManagement" package="disastermanagement">
2    <spacetypes>
3      <e:envspacetype name="2dspace" class="ContinuousSpace2D" ...
4    </spacetypes>
5
6    <componenttypes>
7      <componenttype name="FireBrigade" filename="FireBrigade.agent.xml"/>
8      <componenttype name="Commander" filename="Commander.agent.xml"/>
9      <componenttype name="Ambulance" filename="Ambulance.agent.xml"/>
10   </componenttypes>
11
12   <applications>
13     <application name="small">
14       <spaces>
15         <e:envspace name="my2dspace" type="2dspace">
16           <e:objects>
17             <e:object type="firestation">
18               <e:property name="position">new Vector2Double(0.8,0.4)</e:property>
19             </e:object>
20             <e:object type="firebrigade" number="10">
21               <e:property name="position">new Vector2Double(0.8,0.4)</e:property>
22             </e:object>
23
24             <e:object type="hospital">
25               <e:property name="position">new Vector2Double(0.3,0.3)</e:property>
26             </e:object>
27             <e:object type="ambulance" number="10">
28               <e:property name="position">new Vector2Double(0.3,0.3)</e:property>
29             </e:object>
30           </e:objects>
31
32           <e:processes>
33             <e:process type="create"/>
34           </e:processes>
35
36           <e:dataproviders> ... </e:dataproviders>
37           <e:dataconsumers> ... </e:dataconsumers>
38         </e:envspace>
39       </spaces>
40
41       <components>
42         <component type="Commander"/>
43       </components>
44     </application>
45   </applications>
46 </applicationtype>
```

Figure 12. Application descriptor of the disaster management

cation is itself understood as a composite component, which has its own type name and package (line 1). Next, the space type definition is given (lines 2-4). For brevity reasons this definition is only indicated here and in the real descriptor it encompasses all information about domain and perspective aspects that have been presented in the previous section. Hereafter, the component types are declared (lines 6-10). In this case three different component types are used, namely the *FireBrigade*, *Ambulance* and *Commander* types. The last part of the application descriptor (lines 12-45) shows the definition of an application instance named *'small'* (line 13). This application defines a space instance called *'my2dspace'* based on the *'2dspace'* space type. The space creates several space objects (lines 16-30), a space process (lines 32-34), as well as data providers and consumers (lines 36-37). Space objects include a fire station (lines 17-19), ten fire brigades (lines 20-22), as well as a hospital (line 24-26) with ten ambulances. It can also be seen

that initially the rescue forces are colocated with their respective home bases, i.e. fire station or hospital. The process (line 33) is responsible for generating disasters in the environments at a specific rate and refers to the *'create'* process type defined already in the space type domain elements (cf. Figure 3). Data provider and consumer specifications (lines 36-37) as already presented in Figure 6 complete the space instance definition.

In the last part of the application descriptor, initially created components can be specified. It can be seen that here only a commander agent is declared (lines 42). This is sufficient because the rescue force agents are automatically started when their corresponding space objects thanks to the avatar mapping that ties both sides together.

5. BDI Agents

In Jadex, active components can be implemented according to many different internal architectures. An internal architecture and its realization in a Jadex kernel represent a consistent set of concepts and constructs for implementing the autonomous behavior of concrete active components (e.g. a fire brigade). One prominent internal architecture supported by Jadex is the belief-desire-intention (BDI) model. BDI has been initially conceived as a philosophical model explaining human rational decision making [1]. It is a folk-psychological model that describes how humans *perceive* their own decision making processes, instead of describing how the human brain actually works. The advantage of the model is its intuitive usage, as developers can implement behavior very similar to how they would plan their own everyday activities.

The decision making process described by models like BDI is called *practical reasoning*, as it determines actions to be taken and is opposed to *theoretical reasoning*, which is not directed towards concrete actions. The practical reasoning process of the BDI model is two-staged and consists of a *goal deliberation* and a *means-end reasoning* phase. In the first phase it is decided, which goals an agent should pursue. The latter phase decides how a chosen goal can be accomplished by executing suitable plans [13]. Simplified versions of the BDI model continuously have been applied to agent programming, e.g. in the procedural reasoning system (PRS) [11] and its successors like dMARS [4] and JACK [12].

5.1. The Jadex BDI Kernel

The Jadex BDI kernel is inspired by earlier PRS systems, but introduces several significant extensions. Unlike other systems, the Jadex BDI kernel explicitly supports the goal deliberation phase. Goal deliberation is realized using the so called *EasyDeliberation* strategy [9], which allows specifying inhibition arcs describing which active goals inhibit others based on their importance. Inhibition arcs are used at runtime to determine a consistent set of goals to be pursued. For each of these goals, then the means-end reasoning process is triggered. Means-end reasoning is realized similarly to other PRS systems by goal and plan declarations. The basic idea is that plans encode the procedural knowledge how to accomplish certain tasks and actions, while goals declaratively describe the reasons why tasks and activities should be carried out. The advantage of the approach is that the desired result of agent behavior can be specified separately as a goal and when certain plans fail during execution, the agent can check other options for reaching the

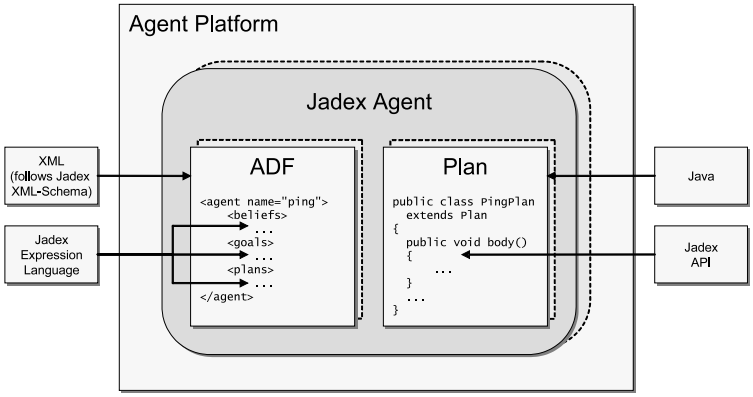

Figure 13. BDI agent specification

respective goal. Thus, the means-end reasoning process collects all plans that are in principle applicable for a goal and checks whether they can be used in the current context. Then the most promising plan is chosen and executed and if it does not accomplish the goal means-end reasoning will try out other plans until the goal is reached or no more plans are available and the goal is considered as failed.

In Jadex, goals are represented explicitly having a state and a lifecycle (see [3]). Goals remain part of the mental state of an agent until they are considered finished or explicitly dropped. Jadex supports four separate kinds of goals that refine the basic goal lifecycle in different ways. A *perform goal* is related to action execution and is considered finished, when at least one plan has been executed for the goal. *Achieve goals* have the purpose to establish some world state, which ist specified as a target condition. An achieve goal is finished, when the target condition is fulfilled, regardless how many plans (if any) had to be executed for the goal. Similarly, a *query* goal describes some information need and is finished, when the required information is available (e.g. as agent belief), regardless if the information had to be obtained by executing plans or was readily available in the agents beliefs. Finally, the *maintain* goal observes a specific world state specified as maintain condition and only becomes active, when this condition is violated. The goal remains active until the condition becomes valid again, e.g. due to the execution of some plan, which was selected for the goal. Afterwards the goal changes back to the inactive state, but keeps monitoring the maintain condition. Thus by default, a maintain goal is never finished and is only removed from the agent's mental state, when it is explicitly dropped.

The implementation of active components using the BDI kernel follows a hybrid language approach. An agent type is defined in a so called agent definition file (ADF), which contains declarative information about the agent structure consisting e.g. of beliefs, goals and plans (see Figure 13, left). The ADF is described in a specific XML dialect, which besides the structural information also supports Java like expressions e.g. for goal conditions and initial belief values. The actual behavior of the agent is captured in the procedural plans, which can be implemented as normal Java classes that inherit from the framework class *jadex.bdi.runtime.Plan* (see Figure 13, right). The BDI functionalities for e.g. creating goals or accessing beliefs are available in the plan classes via an application programming interface (API).

```
<agent name="Ambulance" package="disastermanagement">
  <capabilities>
    <capability name="move" file="Movement" />
  </capabilities>
  <beliefs>
    <beliefref name="env"><concrete ref="move.env"/></beliefref>
    <beliefref name="self"><concrete ref="move.self"/></beliefref>
    <beliefref name="home"><concrete ref="move.home"/></beliefref>
  </beliefs>
  <goals>
    <achievegoalref name="move"><concrete ref="move.move"/></achievegoalref>
    <achievegoal name="treat_victims">
      <parameter name="disaster" class="ISpaceObject"/>
    </achievegoal>
  </goals>
  <plans>
    <plan name="treat_victim_plan">
      <parameter name="disaster" class="ISpaceObject">
        <goalmapping ref="treat_victims.disaster"/>
      </parameter>
      <body class="TreatVictimPlan"/>
      <trigger>
        <goal ref="treat_victims"/>
      </trigger>
    </plan>
  </plans>
  <services>
    <providedservice class="ITreatVictimsService">
      new TreatVictimsService($scope)
    </providedservice>
  </services>
</agent>
```

Figure 14. Agent definition file of the ambulance agent

5.2. Scenario Agent Type Descriptions

In this section, the implementation of the different BDI agent types of the disaster management application is shortly sketched. First, the ambulance agent is presented as representative for the rescue force agents. Afterwards, the implementation of the commander agent is explained. Finally, a short overview of the complete disaster management application is given.

5.2.1. Rescue Force Agents

The rescue force agents (fire brigade and ambulance) are implemented very similarly. Both are based on a separate module called *move capability* that handles interaction with the environment, i.e. the move capability provides access to the avatar of the agent and includes a goal to move the avatar to some target location in the environment. For each service (treat victims, extinguish fire, clear chemicals), the respective rescue force BDI agent defines a separate goal type. The connection between the agent goal and the service interface is realized by service implementation classes that dispatch a corresponding goal for each service request. In response to these goals, the agents execute appropriate plans, i.e. for each service, a plan class is implemented capturing the desired agent behavior.

The combination of agent goal, service implementation and plan class is explained using the ambulance vehicle and the treat victims service as an example. Figure 14 shows the BDI agent definition file of the ambulance component. The agent definition includes

```
1  public class TreatVictimsService extends BasicService
2     implements ITreatVictimsService {
3    protected ICapability agent;
4    public TreatVictimsService(ICapability agent) {
5      super(agent.getServiceProvider().getId(),ITreatVictimsService.class,null);
6      this.agent = agent;
7    }
8    public IFuture treatVictims(final ISpaceObject disaster) {
9      final Future ret = new Future();
10     if(agent.getGoalbase().getGoals("treat_victims").length>0) {
11       ret.setException(new IllegalStateException("Ambulance busy."));
12     }
13     else {
14       final IGoal tv = (IGoal)agent.getGoalbase().createGoal("treat_victims");
15       tv.getParameter("disaster").setValue(disaster);
16       tv.addGoalListener(new IGoalListener() {
17         public void goalFinished(AgentEvent ae) {
18           if(tv.isSucceeded())
19             ret.setResult(null);
20           else
21             ret.setException(tv.getException());
22         }
23       });
24       agent.getGoalbase().dispatchTopLevelGoal(tv);
25     }
26     return ret;
27   }
28   public IFuture abort() {
29     final Future ret = new Future();
30     ISpaceObject self = (ISpaceObject)agent.getBeliefbase()
31        .getBelief("self").getFact();
32     if((Boolean)self.getProperty("patient")) {
33       ret.setException(new IllegalStateException("Patient on board."));
34     }
35     else {
36       IGoal[] goals = (IGoal[])agent.getGoalbase().getGoals("treat_victims");
37       for(int i=0; i<goals.length; i++) {
38         goals[i].drop();
39       }
40       ret.setResult(null);
41     }
42     return ret;
43   }
44 }
```

Figure 15. Implementation of treat victims service

the *move* capability (lines 2-4) and imports some beliefs and goals of the capability. The *env* belief (line 6) provides access to the environment space including the agent avatar, which is stored in the belief *self* (line 7). The move capability further provides the *home* location of the vehicle (line 8) and the achieve goal *move* for moving the avatar (line 11). In addition to the goals and beliefs of the included capability, the ambulance agent defines a new achieve goal *treat_victims* (lines 12-14), which includes a *disaster* parameter for the disaster object (line 13). The *plans* section (lines 16-26) declares a *treat_victims_plan* (line 17) for handling *treat_victims* goals, as specified by the plan *trigger* (lines 22-24). The plan carries over the *disaster* parameter of the goal (lines 18-20) and is implemented in the class *TreatVictimPlan* (line 21). To expose the treat victims service, the agent definition contains a *providedservice* declaration (lines 28-30), which specifies the service interface class (line 28) as well as the service implementation as a constructor invocation expression (line 29).

```
1  public class TreatVictimPlan extends Plan {
2    public void body() {
3      Space2D space = (Space2D)getBeliefbase().getBelief("env").getFact();
4      ISpaceObject self = (ISpaceObject)getBeliefbase()
5         .getBelief("self").getFact();
6      ISpaceObject disaster = (ISpaceObject)getParameter("disaster").getValue();
7
8      // Step 1: Move to disaster location
9      self.setProperty("state", "moving_to_disaster");
10     IVector2 targetpos = DisasterType.getVictimLocation(disaster);
11     IGoal move = createGoal("move");
12     move.getParameter("destination").setValue(targetpos);
13     dispatchSubgoalAndWait(move);
14
15     // Step 2: Treat victim.
16     self.setProperty("state", "treating_victim");
17     Map props = new HashMap();
18     props.put(TreatVictimTask.PROPERTY_DISASTER, disaster);
19     Object taskid=space.createObjectTask("treat_victim", props, self.getId());
20     SyncResultListener res = new SyncResultListener();
21     space.addTaskListener(taskid, self.getId(), res);
22     res.waitForResult();
23
24     // Step 3: Move to hospital
25     ...
26
27     // Step 4: Deliver patient.
28     ...
29   }
30 }
```

Figure 16. Implementation of treat victims plan

The Java class of the service implementation is shown in Figure 15. The class extends the framework class *BasicService* (line 1) and implements the *ITreatVictimsService* interface shown in Section 4.4 (line 2). The constructor (line 4) is used in the ambulance agent definition file shown before and keeps a reference to the *agent* object (line 6), which can be used to access the BDI internals like beliefs and goals from Java code. The remainder of the service implementation contains the two methods defined in the ITreatVictimsService interface. The *treatVictims()* method (lines 8-27) first checks, if there exists a *treat_victims* goal in the goalbase of the agent (line 10). As the ambulance cannot deal with two disasters at once, an exception is generated as the result of the service invocation (line 11), if another active *treat_victims* goal is present. Otherwise a new *treat_victims* goal is created (line 14), initialized with the disaster object (line 15) and dispatched as a new top level goal of the agent (line 24). The goal listener (lines 16-23) is asynchronously notified of the result of the goal execution and passes the success (line 19) or failure (line 21) to the future object, which has been provided as a result of the service invocation (line 26). Similarly, the *abort()* method (lines 28-43) first checks, if the ambulance currently transports a patient (lines 30-32), in which case the current action cannot be aborted and an exception is generated (line 33). Otherwise all *treat_victims* goals of the ambulance are dropped (lines 36-39) and the abort request succeeds (line 40).

Thus, as a result of the service invocation a *treat_victims* goal is activated and in response to this goal, the *TreatVictimPlan* as shown in Figure 16 is selected and executed by the ambulance agent. The plan extends the framework class *Plan* (line 1) and overrides the *body()* method (lines 2-29). In the beginning (lines 3-6), some variables are initialized

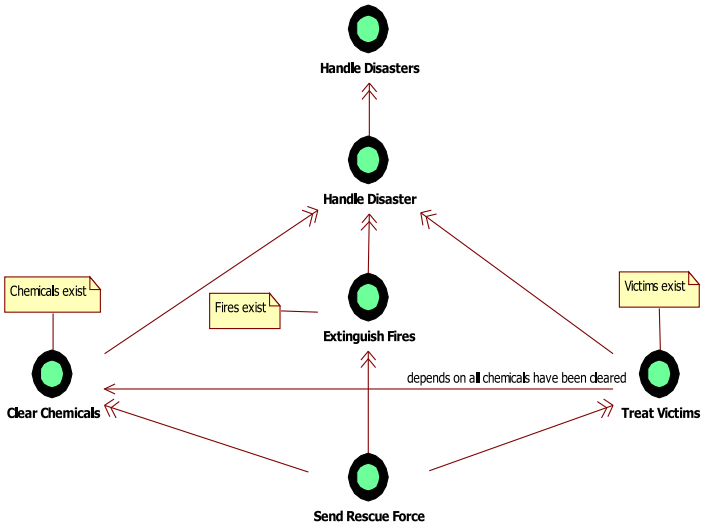

Figure 17. AML mental model for commander goals

from the beliefs and the parameters of the plan including the environment *space*, the avatar *self*, the *home* location and the *disaster* object. The first step of the plan is to move to the disaster location (lines 8-13). This is done by querying the location of a victim from the disaster object (line 10) and dispatching a *move* goal with the victim location as target destination (lines 11-13). The plan then automatically waits until the goal is achieved and the victim location has been reached. Afterwards, the victim is treated and loaded in to the ambulance (lines 15-22). This activity is performed as a *treat_victim* task (lines 17-19) of the ambulance avatar in the environment. The plan waits until the task is finished, using a *SyncResultListener* (lines 20-22), which blocks the plan, but keeps the ambulance component itself responsive to other requests. The other two steps of the plan are similar to the first two steps: After the patient has been loaded into the ambulance, the avatar moves back to the hospital using a move goal (cf. step 1) and delivers the patient in the hospital using another environment task (cf. step 2).

Following the behavior described above, the ambulance continuously performs activities for treating victims as requested through its service interface. The service is requested by the commander agent as described in the next section.

5.2.2. Commander Agent

The general behavior of the commander agent has been described as AML mental model, which is depicted in Figure 17. It shows that the commander has the top-level goal to resolve all occurring disasters (*Handle Disasters*). This overall goal is decomposed to the *Handle Disaster* goal, which represents the commanders objective to tackle a specific disaster, i.e. for each disaster instance a corresponding goal will exist in the commander agent. A disaster is handled by three subgoals. The first one, *Clear Chemicals*, has the task to ensure that all chemicals will eventually be removed from the disaster site. Similarily, the *Extinguish Fires* and *Treat Victims* goals represent the respective desires

to extinguish all fires and transport all injured victims to a hospital. It can also be seen that the goals are only materialized in case it is necessary, i.e. there are chemicals, fires or victims at the considered disaster site. Furthermore, the *Treat Victims* goal depends on the *Clear Chemicals* goal to be successfully finished so that ambulances can enter the target area without contamination risks. Each of the three subgoals is realized using the *Send Rescue Force* goal, which allows instructing a rescue force to work at a selected disaster site.

In Figure 18 a cutout of the commander agent's agent definition file is shown. In addition to the goal types introduced above it contains belief and plan descriptions as well as required services. In the beliefs section (lines 2-10) the environment, the known disaster sites and currently working rescue forces are represented. The *environment* belief (lines 3-5) is initialized with the environment space that is accessible via the parent component acting as application context. Disasters and busy rescue forces are described using belief sets. In case of *disasters* (lines 6-8) the belief set is automatically kept up to date with the current situation by using an update rate, which reevaluates the facts every second (1000 ms). The disasters are retrieved by fetching space objects of type *disaster* from the environment. The belief set for *busy_entities* (line 9) does not initially contain any facts. Instead, plans will use this belief set to store rescue forces that have been assigned tasks by the commander and are currently working on these tasks.

The described goal model is mapped directly to the goals section (lines 11-43). An exception is the top-level goal *handle_disasters*, which needs not to be explicitly modelled as goal entity in the commander ADF. The reason is that its main purpose consists in creating individual *handle_disaster* goals (lines 12-28) for each incidence. This can be achieved in a declarative manner by using a creation condition (lines 17-19), whereby each space object of type disaster is bound to a variable *$disaster*. In order to make this variable permanently available in the goal instance, it is assigned as value to a goal parameter *disaster* (lines 13-15). Additionally using the unique tag (line 16) ensures that every disaster is represented by exactly one *handle_disaster* goal. The goal is considered to be achieved when the target condition (lines 25-27) is fulfilled, i.e. when the disaster object is not contained in the belief set *disasters* any longer. Finally, the requirement of treating severe disasters with higher priority than normal incidents is realized using goal deliberation settings (lines 20-24). Here, an instance level inhibition relationship is used. All other known and non severe goals (expressed using the implicit variable *$ref* pointing to another goal) are inhibited when the considered goal instance is severe (*$goal* refers to the current goal). The other goal types are much simpler. For brevity reasons only *clear_chemicals* and *send_rescueforce* are shown. The first one (lines 29-37) has a parameter *disaster* for storing the disaster space object. As this goal is created from within a plan, the parameter value is set procedurally and needs not to be specified on type level. The target condition (line 36) declares that the goal is achieved when the number of chemicals in the considered disaster site is zero. Additionally, the goal uses a deliberation definition (lines 32-34) to inhibit the *treat_victim* goal for the corresponding disaster. In this way the *treat_victim* goal is suppressed as long as the fire brigades are dealing with chemicals and the *clear_chemicals* goal is active. The omitted *treat_victims* and *extinguish_fires* goals are structurally very similar with a corresponding target condition but no deliberation settings. Finally, the *send_rescueforce* goal is a procedural goal without target condition (lines 38-42). It has two parameters, one for the disaster site and the other for the rescue force.

```
1  <agent name="Commander" package="disastermanagement.commander">
2    <beliefs>
3      <belief name="environment" class="ContinuousSpace2D">
4        <fact>$scope.getParent().getSpace("my2dspace")</fact>
5      </belief>
6      <beliefset name="disasters" class="ISpaceObject" updaterate="1000">
7        <facts>$beliefbase.environment.getSpaceObjectsByType("disaster")</facts>
8      </beliefset>
9      <beliefset name="busy_entities" class="Object"/>
10   </beliefs>
11   <goals>
12     <achievegoal name="handle_disaster" exclude="never">
13       <parameter name="disaster" class="ISpaceObject">
14         <value>$disaster</value>
15       </parameter>
16       <unique/>
17       <creationcondition>
18         ISpaceObject $disaster && $disaster.getType().equals("disaster")
19       </creationcondition>
20       <deliberation>
21         <inhibits ref="handle_disaster">
22           $goal.disaster.severe && $ref.disaster.severe==false
23         </inhibits>
24       </deliberation>
25       <targetcondition>
26         !Arrays.asList($beliefbase.disasters).contains($goal.disaster)
27       </targetcondition>
28     </achievegoal>
29     <achievegoal name="clear_chemicals" exclude="never">
30       <parameter name="disaster" class="ISpaceObject"/>
31       <deliberation>
32         <inhibits ref="treat_victims">
33           $goal.disaster==$ref.disaster
34         </inhibits>
35       </deliberation>
36       <targetcondition>$goal.disaster.chemicals==0</targetcondition>
37     </achievegoal>
38     <achievegoal name="send_rescueforce">
39       <parameter name="disaster" class="ISpaceObject"/>
40       <parameter name="rescueforce" class="IService"/>
41       <targetcondition>$goal.disaster.fire==0</targetcondition>
42     </achievegoal>
43   </goals>
44   <plans>
45     <plan name="handle_disaster_plan">
46       <parameter name="disaster" class="ISpaceObject"/>
47       <body class="HandleDisasterPlan"/>
48       <trigger><goal ref="handle_disaster"/></trigger>
49     </plan>
50   </plans>
51   <services>
52     <requiredservice name="tvs"class="ITreatVictimsService"multiple="true"/>
53     <requiredservice name="efs"class="IExtinguishFireService"multiple="true"/>
54     <requiredservice name="ccs"class="IClearChemicalsService"multiple="true"/>
55   </services>
56 </agent>
```

Figure 18. Agent definition file of the commander agent

For each of the described goal types plans exist in the complete commander agent definition file. Due to brevity only the *handle_disaster_plan* (lines 45-49) is shown and explained here. It can be seen that the plan reacts on *handle_disaster* goals (lines 48) and has a parameter for storing the *disaster* site (lines 46). The value of this parameter is automatically mapped from the corresponding goal parameter. The execution logic of

```
1  public class HandleDisasterPlan extends Plan {
2    public void body() {
3      ISpaceObject disaster = (ISpaceObject)getParameter("disaster").getValue();
4
5      IGoal cc = createGoal("clear_chemicals");
6      cc.getParameter("disaster").setValue(disaster);
7      dispatchSubgoal(cc);
8
9      IGoal ef = createGoal("extinguish_fires");
10     ef.getParameter("disaster").setValue(disaster);
11     dispatchSubgoal(ef);
12
13     IGoal tv = createGoal("treat_victims");
14     tv.getParameter("disaster").setValue(disaster);
15     dispatchSubgoal(tv);
16
17     waitForGoal(cc);
18     waitForGoal(ef);
19     waitForGoal(tv);
20   }
21 }
```

Figure 19. Handle disaster plan body

the plan is contained in an external Java class file for the plan body (line 47), which is described in detail below.

The last part of the commander ADF contains the services section (lines 51-55), which may contain provided and required services of the agent. In case of the commander only required services are specified for the three kinds of offered rescue force tasks (lines 52-54). All of these required services define a *name*, a service *class*, i.e. its interface type as well as a *multiple* attribute. As all required services are multiple they will be bound to all available rescue force services of a given type.

For illustrating further how a disaster is handled by the commander, the plan body of the *HandleDisasterPlan* is shown in Figure 19. In the first part of the plan body (line 3) the disaster is extracted from a plan parameter. In the next part of the plan body subgoals are created for resolving the disaster. A *clear_chemicals* subgoal is created, initialized with parameter values and dispatched (lines 5-7). The same is done for fires using the *extinguish_fires* (lines 9-11) and *treat_victims* (lines 13-15) subgoals. The first two subgoals are active at the same time and may be pursued in parallel depending on how many fire brigades are available. The *treat_victims* subgoal is not activated until all chemicals have been cleared. This is realized using the declarative deliberation settings introduced above. After all chemicals are cleared, the commander deals concurrently with extinguishing the remaining fires (if any) and treating victims. The plan finally waits for all subgoals being accomplished (lines 17-19), because otherwise the plan would immediately finish and automatically abort possibly open subgoals.

5.2.3. Application overview

Now that all aspects of the disaster management application have been presented, Figure 20 shows a screenshot of the running application. The backmost window is the Jadex control center (JCC), in which you can see the model (upper left) and runtime view (lower left) of the Jadex platform. In the model view, the *DisasterManagement.application.xml* has been selected for starting. The runtime view shows the structure of the running appli-

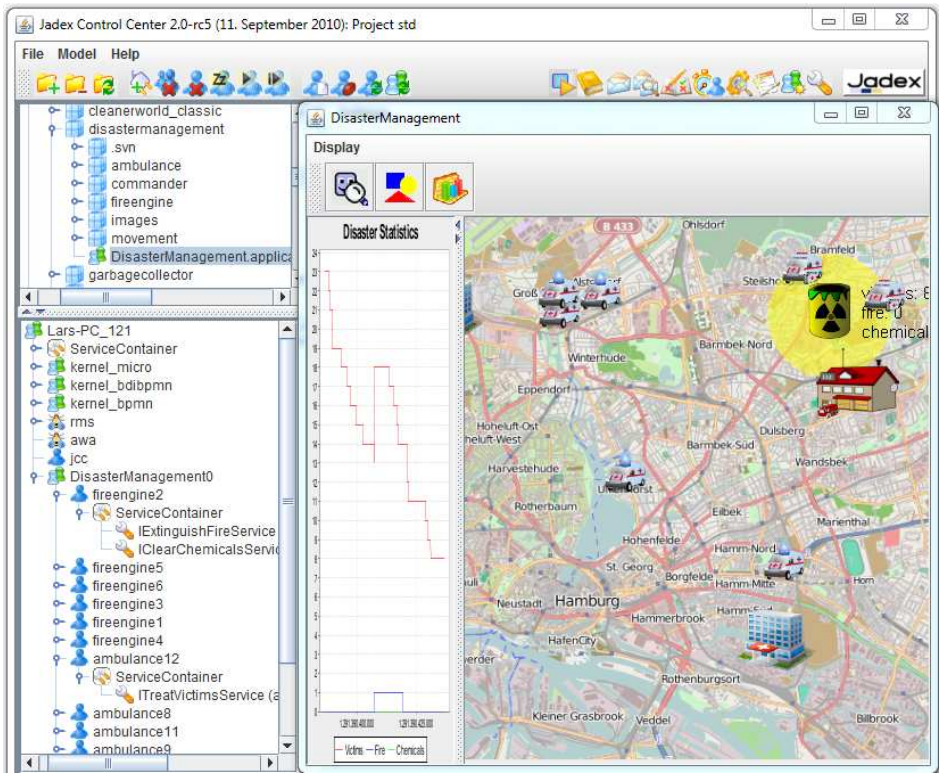

Figure 20. Screenshot of the disaster management application

cation. You can see a number of *fireengine* and *ambulance* components, some of which have been unfolded to show the provided services. Thus the runtime view reflects the application architecture as presented in section 4. The visualization of the application is shown in the frontmost window. To the right, the map of the environment displayed, containing the visual representations of the vehicles, stations and disaster objects. Left from the map, the evaluation is included. It displays the history of the victims, fire, and chemicals values as time series charts. Both aspects are defined using the EnvSupport framework presented in Section 3. The evaluation charts are further used to rate the performance of the coordination strategies of the commander agents, instructing the ambulances and fire brigades, as described in this section.

6. Conclusion

This chapter presented the Jadex framework and its conceptual underpinnings. Main building blocks that have been introduced are environments, active components and services as well as the BDI agent model. In general, Jadex supports various kinds of environments in order to be usable for building simulations as well as real world applications. In case of simulations, virtual environments are of vital importance for enabling rapid prototyping. Jadex supports virtual 2D environments via a specific EnvSupport space, which offers a complete description model for domain objects including environment

processes as well as customizable visualizations via perspectives. Active components are not themselves part of the space, but act on the space by issuing actions or by controlling space objects, such as their avatars.

The second key aspect of Jadex is the concept of active components. An active component is seen as an agent that may act as a service provider and consumer. For this purpose an active component can explicitly define provided and required services. This allows composite components being built from other ones by connecting service ports. As active components are typically used in dynamic environments, in which e.g. service providers vanish or newly appear, dynamic service binding is of special importance. Dynamic binding is based on service search, which is handled in a completely decentralized manner by traversing the component hierarchy. Scopes have been introduced to constrain the areas that should be included in the search, e.g. application scope includes only service providers of the current application and global scope also includes service providers from remote platforms.

As third topic the BDI model of agency and the Jadex BDI architecture have been introduced. Jadex supports the full practical reasoning cycle including goal deliberation as well as means-end reasoning. The first is responsible for deciding which of the existing goals are currently pursued and the latter has the task to find means for realizing a specific goal by applying suitable plans. BDI agents are programmed using a hybrid language approach, in which declarative agent type information is separated from procedural plan knowledge. In the XML based agent definition file (ADF) the beliefs, goals and plans of an agent type are defined, whereas Java classes are used for encoding the plan bodies.

The interworkings of these buildings blocks have been further explained by an example from the disaster management area, in which commander agents are responsible for handling disasters by coordinating different rescue forces such as ambulances and fire brigades. The scenario has been realized as a simulation, whereby the environment has been defined as space that represents rescue units as well as stations and disaster sites. Furthermore, the environment automatically generates disasters using an environment process. The space definition also contains a perspective, which allows a visualization of the environment at runtime. The application logic has been put into agent types for the commander as well as for the different rescue forces. The commander instructs the rescue forces using their exposed services for treating victims, clearing chemicals and extinguishing fires. The internal decision logic of the agents has been realized using the BDI approach.

The Jadex active component framework encompasses several other interesting features that have not been presented in the context of this chapter. One aspect concerns further platform kernels realizing other active component types. Most importantly, kernels have been developed also for executing workflow descriptions based on BPMN [7] and on a newly developed goal oriented process modeling notation called GPMN [2]. Furthermore, a comprehensive tool suite has been built around Jadex, on the one hand supporting the construction of specific active components and on the other hand allowing management and debugging at runtime. Jadex is an open source framework that is hosted at SourceForge. The complete disaster management example including all sources presented in this paper is contained in the Jadex distribution since Jadex V2RC6, available from `http://jadex.sourceforge.net`, where also further documentation on Jadex can be found.

References

[1] M. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, 1987.

[2] L. Braubach, A. Pokahr, K. Jander, W. Lamersdorf, and B. Burmeister. Go4flex: Goal-oriented process modelling. In *Proceedings of the 4th International Symposium on Intelligent Distributed Computing (IDC 2010)*. Springer, 2010.

[3] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal Representation for BDI Agent Systems. In *Proc. of (ProMAS 2004)*, pages 44–65. Springer, 2005.

[4] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dmars. In M. Singh, A. Rao, and M. Wooldridge, editors, *Proceedings of the 4th International Workshop Intelligent Agents IV, Agent Theories, Architectures, and Languages (ATAL 1997)*, pages 155–176, 1998.

[5] K. Jander, L. Braubach, and A. Pokahr. Envsupport: A framework for developing virtual environments. In *Seventh International Workshop From Agent Theory to Agent Implementation (AT2AI-7)*. Austrian Society for Cybernetic Studies, 2010.

[6] J. Marino and M. Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 1st edition, 2009.

[7] Object Management Group (OMG). *Business Process Modeling Notation (BPMN) Specification*, version 1.1 edition, February 2008.

[8] A. Pokahr and L. Braubach. The notions of application, spaces and agents — new concepts for constructing agent applications. In M. Schumann, L. Kolbe, M. Breitner, and A. Frerichs, editors, *Multikonferenz Wirtschaftsinformatik 2010*, pages 159–160. Universitätsverlag Göttingen, 2010.

[9] A. Pokahr, L. Braubach, and W. Lamersdorf. A goal deliberation strategy for bdi agent systems. In T. Eymann, F. Klügl, W. Lamersdorf, M. Klusch, and M. Huhns, editors, *Proceedings of the 3rd German conference on Multi-Agent System TEchnologieS (MATES-2005)*. Springer, 2005.

[10] A. Pokahr, L. Braubach, J. Sudeikat, W. Renz, and W. Lamersdorf. Simulation and implementation of logistics systems based on agent technology. In T. Blecker, W. Kersten, and C. Gertz, editors, *Hamburg International Conference on Logistics (HICL'08): Logistics Networks and Nodes*, pages 291–308. Erich Schmidt Verlag, 2008.

[11] A. Rao and M. Georgeff. BDI Agents: from theory to practice. In V. Lesser, editor, *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS 1995)*, pages 312–319. MIT Press, 1995.

[12] M. Winikoff. JACK Intelligent Agents: An Industrial Strength Platform. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 175–193. Springer, 2005.

[13] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2001.