Introduction

The Jadex agent framework supports the definition and management of agent applications composed of different agents and additional components. Agent applications can be defined in an XML-based specification file (*.application.xml). Agent instances can also be started within an already running application and also access other agents or components of this so called application context. Inside the platform, applications can be managed (i.e. started and stopped) as a whole. The application definition is intentionally agent type agnostic, meaning that it is possible to describe agent applications composed of different agent kinds (e.g. bdi and micro agents). This also means that no details about agents are specified within an application descriptor.

The application features have been introduced in Jadex 2.0. It is still supported to develop agents, which are not associated to a specific application. The agent application features described in this guide are thus optional and it is not required to use them for developing your own applications. Nevertheless, most agent applications composed of more than a single agent will profit from these features and it is highly recommended to take the little bit of extra effort for writing an application descriptor in favor of a better and simpler application design.

Defining Simple Agent Applications

This section describes how to define simple agent applications. Subsequent sections cover how to add non-agent components (called spaces).

Application Types

The root element of the application descriptor XML schema is shown in Figure 2.1. As you can see, an application descriptor contains a single application type, which may contain agent types as well as space types. Agent types are defined by specifying a name that is used to refer to the type in the document and a filename, that refers to the agent implementation. Space types will be covered later. Based on the defined agent types and space types, application configurations (<applications>) can be specified. Additionally, imports can be declared to avoid having to use fully qualified class names throughout the descriptor.

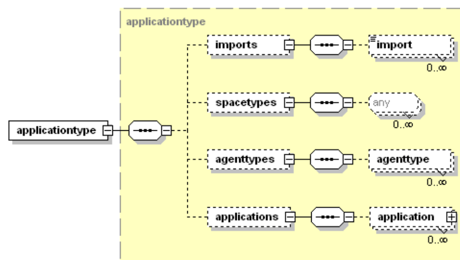

Figure 2.1: XML schema for applicationtype elements

Figure 2.2 gives an example of the elements in the application descriptor, discussed so far. First, some comments about file/XML issues. Names of application definition files should end with '.application.xml' to be found by Jadex. Moreover, you can (and should) use an XML comment at the start of the document to describe your application. The comment will be displayed by tools, such as the JCC, when loading the application. In the comment, you may use HTML codes to adapt the text presentation. The namespace declaration 'xmlns="http:// jadex.sourceforge.net/jadex-application"', while not strictly required, should also be used, as it allows tools to check the conformance of your document to the schema definition (editors, such as eclipse, e.g. can provide context-help and auto-completion based on the schema definition). The schema location definition is useful, because it allows your editor to load the schema definition file from the Jadex web space. Alternatively, you may also configure your editor, to use a local .xsd file (e.g. 'XML Catalog' settings in eclipse).

```
<?xml version=

"1.0" encoding=
"UTF-8"?>


<!--
  <H3> The Ping Application

</H3>
  ...
-->

<applicationtype xmlns=

"http://jadex.sourceforge.net/jadex-application"
  xmlns:xsi=

"http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=

"http://jadex.sourceforge.net/jadex-application
                      http://jadex.sourceforge.net/jadex-application-2.0.xsd"
  name=
```

```
"Ping" package=
"jadex.bdi.examples.ping">



<agenttypes>


<agenttype name=

"Ping" filename=
"jadex/bdi/examples/ping/Ping.agent.xml"/>



<agenttype name=

"Pinging" filename=
"jadex.bdi.examples.ping.Pinging"/>



</agenttypes>



<applications>
    ...


</applications>

</applicationtype>
```

Figure 2.2: Basic application descriptor, part 1 (taken from ping example)

With regard to the application type itself, you have to specify a name, which should correspond to the file name (e.g. use 'Ping' for file 'Ping.application.xml') and also provide the package with respect to the directory structure, the file is located in (e.g. jadex.bdi.examples.ping for jadex/bdi/examples/ping). The package attribute can be omitted, if the application file is located in the root directory of the classpath (i.e. in the default package).

The <agenttypes> section contains the types of all agents that participate in the application. Each entry is a mapping from a type name that can be freely chosen to the implemenation given in form of a file name (e.g. jadex/bdi/examples/ping/Ping.agent.xml) or logical name (e.g. jadex.bdi.examples.ping.Pinging). The type name is used inside the application instances section (described below), but can also be used to start new agents at runtime (e.g. from the code of some agent). Therefore, the agenttypes section provides a convenient way to easily exchange agent implementations without having to alter any other code of the application.

Application Instances (Configurations)

The XML structure of the application element is shown in Figure 2.3. It represents an instance or configuration of the application type and defines the spaces (i.e. space type instances) and agents (agent type instances) together with their configuration settings (e.g. arguments for agents).

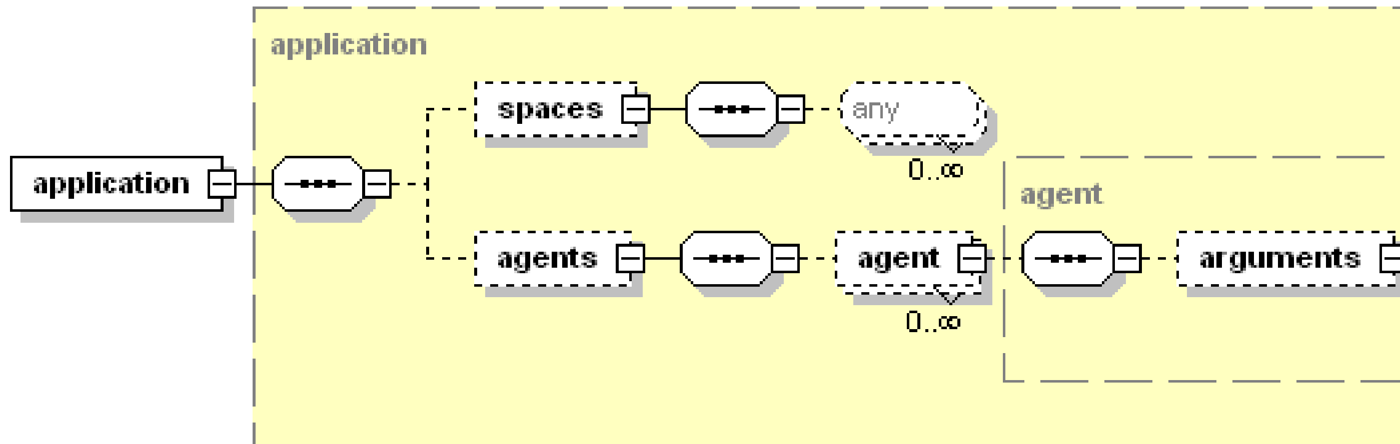

Figure 2.3: XML schema for application elements

An example of an <applications> section is given in Figure 2.4. Here, three different application configurations are defined ("Simple Pinging", "Multi Pinging" and "Fast Pinging"). The name of the application configuration is required and is used to identifty, which configuration to instantiate, when starting an application. The first application ("Simple Pinging" in the example) is used as default configuration, when no specific configuration is selected.

```
<applications>

<application name=

"Simple Pinging">


<agents>

<agent type=

"Ping" name=
"Ping" />


<agent type=

"Pinging" name=
"Pinging" />


</agents>

</application>


<application name=

"Multi Pinging">


<agents>

<agent type=

"Ping" name=
"Ping"/>
```

```
<agent type=

"Pinging" number=
"2"/>



</agents>


</application>



<application name=

"Fast Pinging">



<agents>


<agent type=

"Ping" name=
"Ping"/>



<agent type=

"Pinging" name=
"FastPinging">



<arguments>


<argument name=

"ping_delay">

            500


</argument>


</arguments>


</agent>


</agents>


</application>

</applications>
```

Figure 2.4: Basic application descriptor, part 2 (taken from ping example)

The <agents> section of each application specifies, which agents to start. The available attributes are given in Figure 2.5. The type is the only required attribute. The arguments for each agent are specified in a separate <arguments> section. Each argument is a name/value-pair, where the name is given as attribute, while the value is specified as content of the <argument> element. The value can be an arbitrary Java expression and may refer to the platform using $platform.

- **name:** The name for the agent instance (if omitted, a name will be generated).

- **type:** The type of agent to be created (as specified in the agenttypes section).
- **configuration:** The configuration of the agent (as specified in the agent's ADF).
- **start:** Per default, each created agent is also started (i.e. will immidiately start to execute any actions). Set the start flag to false, if the agent should be started later manually.
- **number:** Allows the create multiple agent instances at once. If omitted only one agent is created.
- **master:** A master agent ist required for the application, thus, if a master agent is killed, the application will be closed down. Default is false.

Figure 2.5: Attributes of the <agent> element

Defining and Using Application Spaces

A space is a structure that contains application specific data and components, which are independent of a single agent. Therefore a space provides a convenient way of sharing resources among agents without using purely message-based communication.

Spaces also can be seen as an extension point of the agent platform as spaces offer application functionality, independent of the agent kernel (e.g BDI or Micro agents can participate in the same space). Two examples of such functionality are currently included in the Jadex distribution. A simulation of 2D environments (environment space) and coordination based on an agent-group-role model (AGR space). The environment space is quite sophisticated and therefore described in a separate guide on its own . The AGR space implementation is currently only a basic proof of concept. Due to its simplicity, the AGR space is used as an example throughout the rest of this chapter.

Technically, a concrete kind of space (e.g. environment or AGR) consists of an XML schema defining elements to be included in the <spacetypes> and <spaces> sections of the application descriptor, as well as Java classes, which provide an interface to the implemented functionality. If you are interested in building your own kind of space, you should look at the simple AGR space as a starting point.

Space Type Definition

Space-related definition are included in two places in the application descriptor. In the <spacetypes> section the type definitions are contained, which describe the static structure and properties of a space. In the <spaces> section of the <application> element, space instances are described, that contain concrete space settings for the enclosing application configuration.

To be able to refer to the elements from the XML schema of the specific space kind, it is useful to define a separate XML namespace prefix for each of the space kinds, used in the application descriptor. E.g., in Figure 3.1, the 'agr' prefix is declared for the http://jadex.sourceforge.net/jadex-agrspace namespace, which contains the elements of the AGR space.

```
<applicationtype xmlns=

"http://jadex.sourceforge.net/jadex-application"
  xmlns:agr=

"http://jadex.sourceforge.net/jadex-agrspace"
  xmlns:xsi=

"http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=

"http://jadex.sourceforge.net/jadex-application
                      http://jadex.sourceforge.net/jadex-application-2.0.xsd
                      http://jadex.sourceforge.net/jadex-agrspace
                      http://jadex.sourceforge.net/jadex-agrspace-2.0.xsd"
  name=

"MarsWorldAGR" package=
"jadex.bdi.examples.marsworld_classic">



<spacetypes>


<agr:agrspacetype name=

"marsagrspace">
```

```
<agr:grouptype name=

"marsteam">


<agr:roles>


<agr:role name=

"sentry" min=
"1" max=
"1" />


<agr:role name=

"producer" min=
"1" />


<agr:role name=

"carrier" min=
"1" />


</agr:roles>


</agr:grouptype>


</agr:agrspacetype>


</spacetypes>


<agenttypes>
    ...


</agenttypes>


<applications>
    ...


</applications>

</applicationtype>
```

Figure 3.1: Example space type definition (from marsworld example)

The marsworld example above defines a single space type 'marsagrspace', which is of the kind 'agr:agrspacetype'. The details of this definition are outside the scope of the generic application descriptor XML schema and therefore all refer to the AGR schema namespace using the 'agr' prefix. Here, a simple group structure is defined, composed of the sentry, producer and carrier role with corresponding min and max values for the number of required/allowed agents playing each role.

Space Instance Definition

As mentioned above, a space instance is part of a specifc application configuration in the <applications> section and describes the concrete space settings for this application configuration. A space instance in an application usually has to refer the corresponding space type as described in the <spacetypes> section. Space instance elements usually provide ways to relate the application agents to the space. E.g., in an AGR space, you can specify, which kind of agent plays which role(s).

```
<applications>

<application name=

"1 Sentry / 2 Producers / 3 Carriers">

<spaces>

<agr:agrspace name=

"myagrspace" type=
"marsagrspace">

<agr:group name=

"mymarsteam" type=
"marsteam">

<agr:position role=

"sentry" agenttype=
"Sentry" />

<agr:position role=

"producer" agenttype=
"Producer" />

<agr:position role=

"carrier" agenttype=
"Carrier" />

</agr:group>

</agr:agrspace>

</spaces>

<agents>

<agent type=

"Environment"/>

<agent type=

"Sentry"/>
```

```
<agent type=

"Producer" number=
"2"/>



<agent type=

"Carrier" number=
"3"/>



</agents>


</application>

</applications>
```

Figure 3.2: Example space (instance) definition (from marsworld example)

Figure 3.2 shows, how an AGR space instance is defined. The 'myagrspace' space is declared as an instance of the 'marsagrspace' space type declared previously. A single group instance is defined, which uses <position> tags, to establish a mapping between agent types and roles. Whenever an agent of a specific type (e.g. 'Producer') is started it is automatically assigned the corresponding role (e.g. 'producer').

Accessing a Space From Inside Your Agents

The previous sections have explained how to declare space types and space instances in the application.xml file. Usually you will also want to access a space from the code of your agent. Therefore a space instance is represented as a Java object providing a set of methods that is specific to the space kind (e.g., an AGR space object offers other methods than an environment space object). To get hold of the space object, an agent can use its application context as shown below. Accessing the application context differs for the different agent kernels. Please refer to the concrete kernel documentations for more details. Below, two examples are given for BDI agent and Micro agents, respectively.

To access a space from the plan of a BDI agent, use 'getScope().getApplicationContext()' to get the application context. From the external access, you can also directly get the application context. From the application context, you can lookup a specific space instance, by providing the name as defined in the application.xml. E.g., in Figure 3.3, the 'myagrspace' space instance of obtained and casted to the specific implementation class of the space kind (AGRSpace). Now the agent can use the AGR to find all agents, which currently play the 'carrier' role in the 'mymarsteam' group. The resulting agent identifiers can then be used to send messages to agents of the requested role (not shown).

```
IApplicationContext ac = getScope().getApplicationContext();
AGRSpace agrspace = (AGRSpace)ac.getSpace(

"myagrspace");
Group group = agrspace.getGroup(

"mymarsteam");
IAgentIdentifier[] carriers = group.getAgentsForRole(

"carrier");
```

Figure 3.3: Usage of the AGR space API (ProductionPlanAGR.java from marsworld example)

Using the 'getApplicationContext()' method, spaces can be accessed similarily from Micro agents as well. In Figure 3.4 below is a cutout from the heatbugs example (in this case using an environment space).

```
IApplicationContext app = getApplicationContext();
Grid2D grid = (Grid2D)app.getSpace(

"mygc2dspace");
ISpaceObject avatar = grid.getAvatar(getAgentIdentifier());
```

Figure 3.4: Accessing a space from inside a Micro agent (from heatbugs example)

Application Management

This chapter describes how applications can be managed using the Starter perspective of the Jadex Control Center (JCC).

Starting and Stopping Applications

Applications can be started and stopped in a similar way like agents. The model explorer will show application.xml files that are found in the included directories (see Figure 4.1, top-left). When an application.xml is selected, the start options will be shown to the right. You can choose one of the application configurations as defined in the <applications> section of the XML. Additionally, you have to specify a name for the to be created application instance (the type name is used as default). Because you can not have two application instances of the same name at once, you have to change the name, when you want to start an application twice. In the figure, two application instances of the garbage collector example have been started, named 'GarbageCollector app1' and 'GarbageCollector app2'.

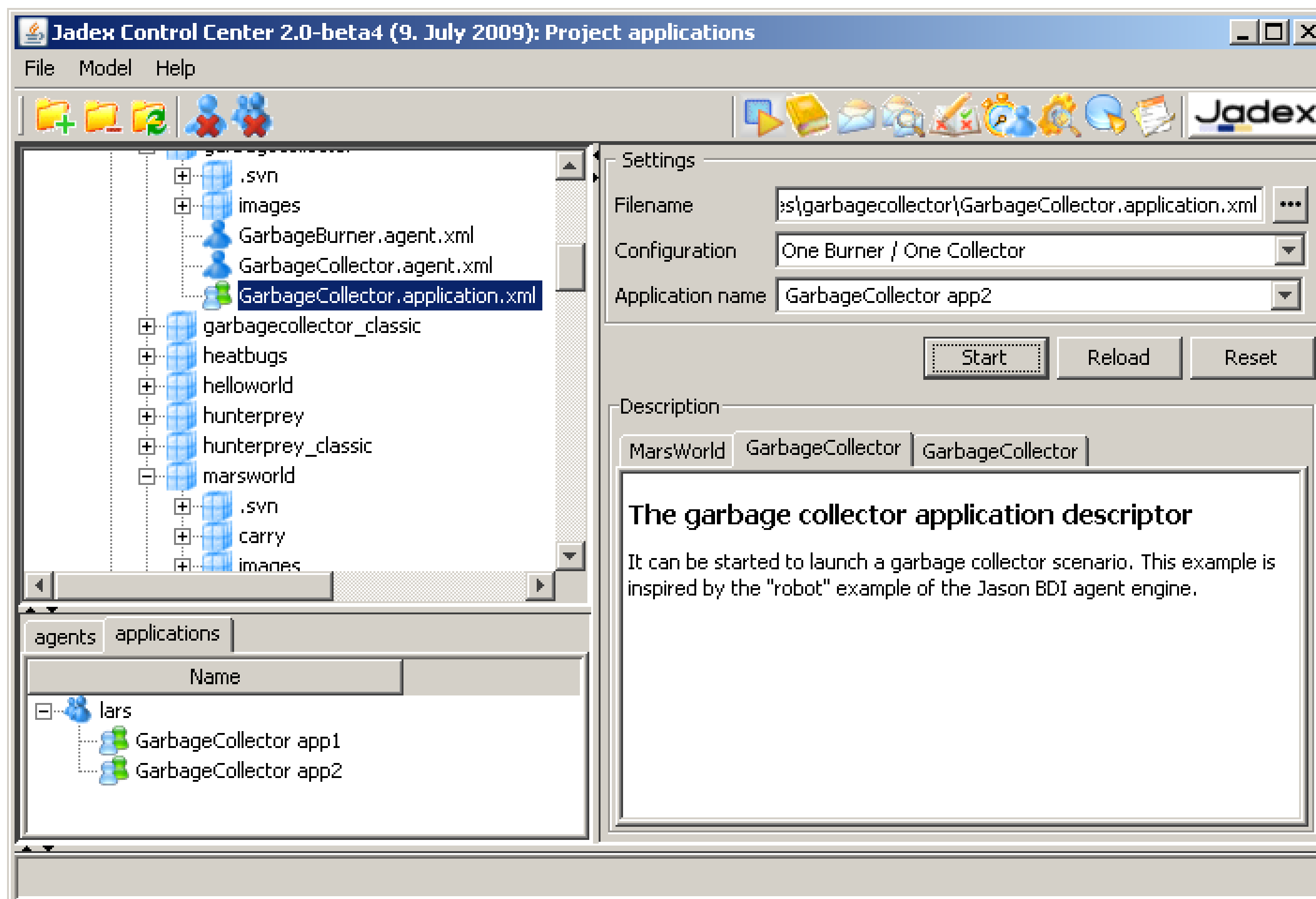

Figure 4.1: Application management in the JCC

The running applications are shown at the left bottom of the JCC window. There is a tab panel that allows to switch between the 'agents' view and the 'applications' view. To stop an application, right-click on it in the applications view and choose 'Kill application'. The application instance and all its agents will then be removed from the platform.

Starting and Stopping Agents of an Application

When an application instance is already running, you can still add and remove agents from this application. In the dialog for starting agents, a new option for the application is available ('Application name', Figure 4.2, top-right). The drop down list allows you to choose among the currently running applications. If you leave the name blank, the agent will be started outside any application. Otherwise, the agent is added to the running application. If the agent type is not included in the <agenttypes> section, an error will occur and the agent will not be started.

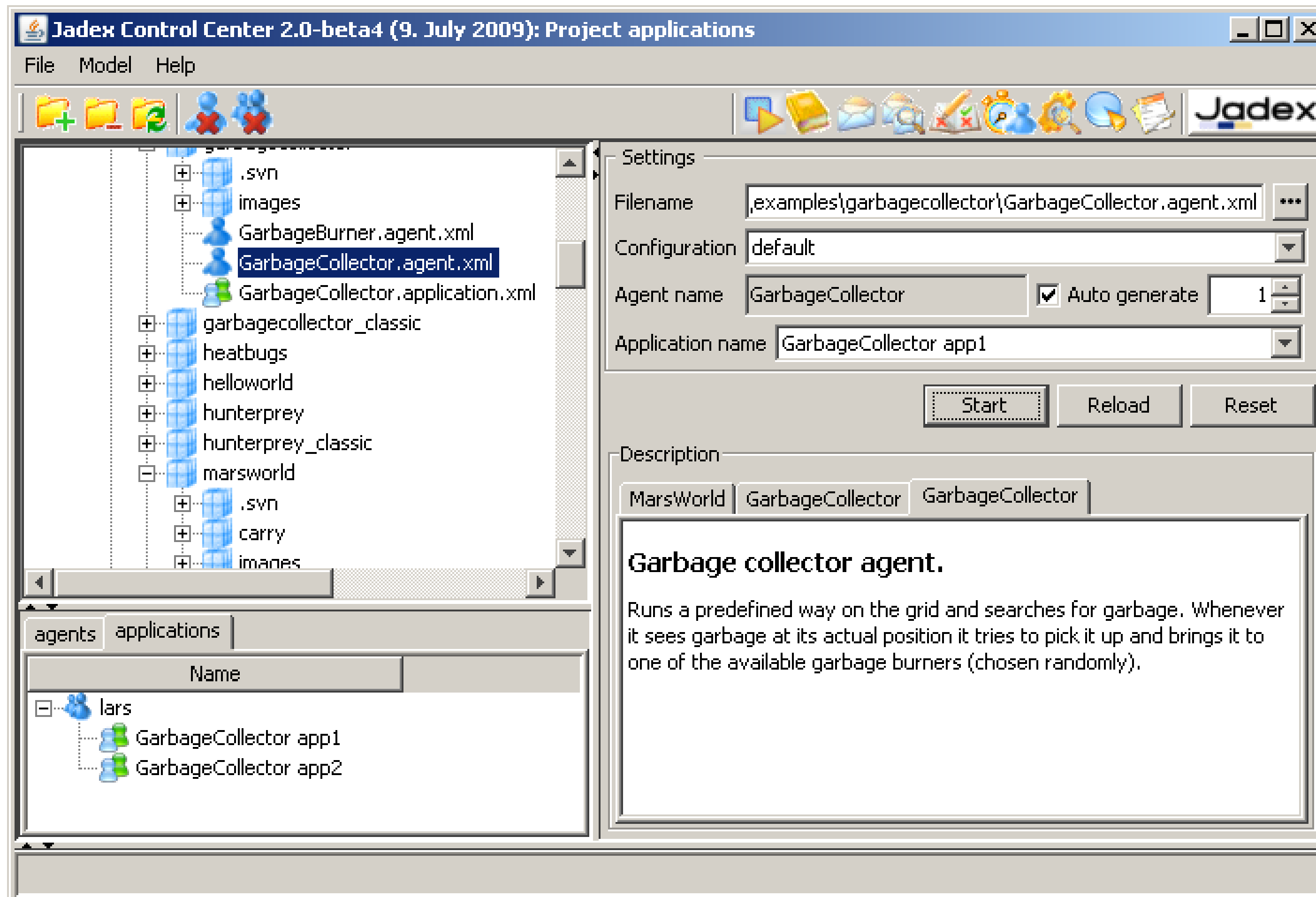

Figure 4.2: Starting an agent as part of an already running application

When an agent is killed, it will be automatically removed from its application. So, if you wish to remove an agent from an application, just right-click on the agent instance in the 'agents' tab (bottom-left) and choose 'Kill agent'.