Deployment of Distributed Multi-agent Systems

Lars Braubach[1], Alexander Pokahr[1], Dirk Bade[1],
Karl-Heinz Krempels[2], and Winfried Lamersdorf[1]

[1] University of Hamburg, Dept. of Computer Science,
Distributed and Information Systems,
Vogt-Klln-Str. 30, 22527 Hamburg, Germany
{braubach, pokahr, lamersd}@informatik.uni-hamburg.de
[2] University of Aachen, Dept. of Computer Science, Informatik IV,
Ahornstr. 55, 52074 Aachen, Germany
krempels@informatik.rwth-aachen.de

Abstract. The agent metaphor has shown its usefulness for modelling as well as implementing complex and dynamic applications. Although a number of agent applications has been successfully realised and used, it must be stated that the distribution of commercial off-the-shelf applications is very scarce. For this discontenting situation, at least two reasons can be identified. On the one hand, the development of agent-based applications is difficult suffering from insufficient standards and tools and on the other hand deployment issues are little researched and supported. In this paper, several deployment-related topics are discussed and a vision for the deployment of distributed multi-agent systems is conceived. From the vision, requirements for launching and configuring agent applications are derived. According to these requirements, a platform independent reference model of the proposed deployment infrastructure is presented. The reference model provides the basis for the development of our ASCML (Agent Society Configuration Manager and Launcher) tool, which is currently implemented for the JADE and Jadex multi-agent platforms.

1 Introduction

Multi-agent systems (MAS) are composed of autonomous, interacting, more or less intelligent entities. The agent metaphor has proven to be a promising choice for building complex and adaptive software applications, because it addresses key issues for making complexity manageable already at a conceptual level [1]. Furthermore, agent technology can be seen as a natural successor of the object-oriented paradigm and enriches the world of passive objects with the notion of autonomous actors. Therefore, one would suppose agent applications to be in widespread use in academic as well as in industrial projects. The contrary is the case. Even though many agent applications are developed in various domains [2], most of them are specialised solutions that are deployed in at most one setting. The question arises: Why are agent applications not yet widely distributed?

One reason for this is that the development of MAS is inherently difficult and error prone, because of several intricate issues. First, the development process for

M.-P. Gleizes, A. Omicini, and F. Zambonelli (Eds.): ESAW 2004, LNAI 3451, pp. 261–276, 2005.

building agent applications is in most cases ad-hoc and not based on a generally accepted methodology, like for example the well-known Unified Process for UML [3] in the object-oriented world. For agent systems, no such common ground exists due to different agent architectures and missing standards. In consequence, a methodology has to be chosen independently for each project among several alternatives. This choice is crucial for the project's success and is constrained by domain and implementation aspects [4]. In addition, whatever methodology is selected, the tool support is always relatively poor and does not cover all phases of the development process.

Besides the methodology, the development of agent-based applications is difficult, because the software is distributed and dynamic in nature and demands various new skills and a new way of thinking from the developers. E.g. an object-oriented software engineer cannot easily change to the agent paradigm without considering ontology descriptions and studying the abstract speech-act based agent communication. Additionally, intelligent agents often use mentalistic notions or employ rule-based approaches.

Another important reason for the scarce distribution of commercial off-the-shelf agent applications is that there is currently no support for the deployment of agent applications. In areas such as distributed object systems, systematical guidelines and mechanisms for all activities concerned with deployment issues have been developed. These guidelines ensure that a properly developed distributed application can be packaged into a reusable, maintainable, and configurable piece of software. However, although multi-agent systems composed of autonomous proactively (inter-)acting entities differ considerably from distributed object systems, the issue of appropriate deployment techniques for MAS is not yet very much researched.

The vision of this paper is to specify agent applications at a high-level using constraints to declare what system properties need to be fulfilled for the application to work properly. E.g. one could demand certain services and agent roles to be available, whereby the deployment environment has the task to interpret and supervise these constraints and has to start agent instances accordingly. As a first step towards this high-level deployment for MAS we propose a reference model for the launching of distributed multi-agent applications that are specified by declaring which and how many agent instances shall be instantiated in what order. As part of the reference model a generic meta-model for the specification of agent applications is described, which consists of one layer for the definition of agent types and another one for the ordered composition of agent instances belonging to a certain application scenario. To underline the applicability of the proposed model a prototype implementation is presented.

The rest of the paper is structured as follows. The next section presents some background on deployment in general and deployment of agents in particular. We use our vision to derive requirements for deployment of distributed multi-agent systems in section 3. To meet these requirements, in section 4 a reference model is presented, and it is explained in section 5 how the reference model is

implemented in our Agent Society Configuration Manager and Launcher tool (ASCML). The last section summarises the paper, gives some conclusions, and outlines areas for future work.

2 Background

The Object Management Group (OMG) defines deployment as "the processes between acquisition of software and execution of software". In [5] a general deployment process for distributed systems is specified, which consists of five phases. In the installation step the software is acquired and stored in a local repository that not necessarily needs to be the program's execution location. Next, the software can be functionally configured in the sense that application specific properties are set to certain values. This may result in several different application configurations. Thereafter a deployment plan taking into account the target environments and the software requirements is developed. With the help of this deployment plan, the code placement can be done in the preparation phase. Finally, the application can be launched, which demands the starting and runtime configuration of software at the planned nodes in the target environment.

For agent-based applications, this process is more dynamic and flexible, because the application constituting elements are autonomous agents instead of passive components. Nevertheless, the above-mentioned activities are important for MAS as well and will be discussed with respect to their peculiarities in the following. Concerning the installation step, two distinct kinds of software have to be available. On the one hand, the agent infrastructure, which is responsible for offering the basic agent services like messaging, white and yellow pages service needs to be acquired. For this purpose, normally agent platforms are used. On the other hand, the application specific agent software needs to be accessible, whereby for certain types of applications it may be sufficient to load portions of agent code dynamically. The functional configuration of MAS can be done by adjusting the available agent start-parameters and by fine-tuning the number of agents to be started. E.g. the number of service agents could be used to tune the application's scalability for small and large enterprises accordingly. The planning and preparation steps for MAS involve the decisions about the placement of infrastructure and application code on the environment nodes. Therefore, considerations about possibly mobile agents and dynamic code retrieval have to be taken into account; e.g. movements of agents may require platforms to be installed on network nodes, where no agents are initially running.

The launching of MAS differs to a great extent from starting a component-based application. Component-based applications have a hierarchical structure and are usually launched using a single starting point, which creates the necessary subcomponents. On the contrary, agent-based applications consist of a bundle of autonomous actors that are self-dependent after birth. Hence, to define configurations of agent applications notions conceptually more abstract than single agents are necessary. Minimum for the description of an agent application at a concrete level is that agent instances and dependencies between these in-

stances can be expressed. Nevertheless, specifications that are more abstract are desirable and could support a higher degree of robustness and maintainability.

In [6] several agent platforms are compared with respect to their support for the analysis, design, implementation, and deployment phase. In correspondence to our actual research, it turns out that only very few platforms address issues of deployment at all. Positive exceptions can be found within Agent Academy [7], AgentBuilder [8], ZEUS [9], AgentFactory [10] and BlueJADE [11]. To our knowledge only the Agent Academy and the somewhat outdated AgentBuilder frameworks offer tool support for the specification and launching of agent applications. Both platforms allow the simple designation of parameterised agent instances from formerly defined agent types. These agent instances will be started altogether, when the so defined agent application is launched. ZEUS and AgentFactory utilise tools for the generation of human readable starting scripts that contain a list of ordered commands for instantiating and starting agents. In contrast to the aforementioned tools, BlueJADE is an attempt to integrate an agent framework into an application server treating the platform as manageable service. Hence, it shifts the responsibility of agent management to the application server, which allows starting and stopping individual agents as well as platforms. The conceptual problems of specifying agent applications are not addressed.

One obvious drawback of all solutions found, consists in the missing possibility to define any kinds of dependencies that may constrain the order of agents to start. Additionally, the agent application meta-models are specified only implicitly, rendering the creation of a cross-platform launching tool almost impossible. With respect to our vision it has to be stated that currently available solutions carry out the definition of agent applications merely at the concrete level, what makes it difficult having flexible and scalable applications. By utilising a more abstract approach, agents could be started in response to certain application and environmental demands. This more abstract way of an agent application is also related to organisational approaches [12, 13]. These aim to structure MAS with respect to the organisational settings found in the addressed problem domain. Hence, the motivation for structuring agents is different but the concepts have some similarities and probably will allow a consolidation of both directions.

Directly related to the starting of agent applications is the dynamic application reconfiguration, which either could be done automatically by the configuration environment, or could be done manually by some administration authority. An abstract application specification could be a promising starting point for dynamic configuration mechanisms as well, because application constraints could be supervised and used to trigger reconfiguration actions.

Until now, the extent to which dynamic reconfiguration is supported by agent platforms, is mostly reduced to the allocation of agents to network nodes to cope with varying network loads. E.g. the RECoMa [14] reconfiguration manager of the RETSINA [15] framework was developed to launch agents, reallocate them to other computers, and monitor their runtime states. Some aspects of more advanced configuration mechanisms for agent-based applications have been covered by a preliminary and now deprecated FIPA specification [16], which underlined

the importance of agent dependency specifications, life cycle management, and monitoring mechanisms. The idea of the FIPA Agent Configuration Management work group was to introduce configuration domains in which a designated management agent is responsible for monitoring this domain.

Due to the fact that there are only few agent applications in the market, it is not astonishing that the development of configuration concepts and tools has not gained much interest until to date. Widening the horizon of considered configuration targets from agent-based to distributed component-based applications, it is interesting that agent-based approaches for configuring *component-based* applications can be found. E.g. in [17] a hierarchical agent-based infrastructure for monitoring and configuring distributed applications is proposed.

3 Requirements

Having presented the current state of the art with respect to deployment of multi-agent systems, a lack of concepts, standards, and tools can be identified, in particular for *launching* and dynamic *reconfiguration* of complex agent-based applications. These two aspects of deployment are essential to achieve the vision of specifying agent applications at a high level. In the following, we will discuss the desirable features with respect to the launching of preconfigured multi-agent systems and investigate what is needed to achieve dynamic reconfiguration of agent-based applications.

Before going into details about launching of agent applications, we have to clarify some terms used in the following. Configurations of component-based applications can be defined at two levels: Component level configurations and application level configurations [18]. For agent-based systems, the agent level and the application level can be distinguished. Considering a single agent, a distinction can be made between the static implementation parts and the running processes. When we need to highlight this distinction, we refer to the former as *agent type* and to the latter as *agent instance*. This distinction can also be made at the application level. We use the term *society type* to refer to the static properties of a multi-agent application. A *society type* in our terms is a composition of agent types, supplemented with some (e.g. interaction) constraints. A *society instance* refers to the instantiation of a society, and is composed of single agent instances and concrete dependencies between those instances. The model should be recursive to allow societies to be part of larger societies on the type as well as on the instance level.

3.1 Basic Management Services

To support the launching of distributed multi-agent applications several basic services can be identified. First of all, services are needed for starting and stopping agent and society instances. For invoking these services, at least the following information has to be supplied. The start of an agent instance should be based on a given agent type definition which has to contain a reference to

the agent implementation (e.g. a Java class) and should declare the parameters that can be supplied to an agent of this type. To instantiate an agent, its type definition, the name for the agent instance (according to FIPA) and the assigned values for the parameters have to be supplied. To stop an agent instance only the agent identifier has to be known.

A society instance definition should contain all additional information required to instantiate a multi-agent application based on a society type definition. Therefore, a society instance definition has to contain the concrete agent instances with names and parameter assignments, as well as any dependencies that have to be respected when launching the application. This allows starting a complete society by just referring to the instance definition. To be able to identify a running application, a unique name should be given to each started society. It has to be assured that the agent instances belonging to a society are known, so that a society instance can also be stopped as a whole.

In order to launch *distributed* applications these basic services should be available remotely, therefore issues of security and accounting have to be considered [19]. In addition, it is desirable that only minimal requirements are necessary for the manual configuration of network nodes, which could be achieved by code distribution and a service that allows remotely starting new agent platforms.

The basic services additionally require a launch process management that has to make sure, that the correct agents, societies, and platforms are launched at the correct nodes at the correct times. One can imagine several ways to specify this. At the concrete level, it is possible to directly define the dependencies between agent instances of a society instance. The launch process management can then determine the launch order based on a topological sort of the dependency graph.

Constraints that are more abstract such as dependencies to specific services or roles can be employed to define application characteristics already at the type level (i.e. in agent type or society type definitions). In addition, application specific constraints and network load characteristics can be used to determine the allocation of agent instances to the available network nodes.

3.2 Monitoring and Reconfiguration

Once an application has been launched, the monitoring and reconfiguration of the running societies and agent instances should be supported. On the one hand, an administrator might want to observe a running application and manually add or remove agent instances or reallocate mobile agents to new network nodes. On the other hand a monitoring service should take care of the constraints and dependencies specified in the type and instance definitions and perform appropriate actions when the constraints get violated, e.g. by starting additional service agents to assure a given response time. By detecting failures and relaunching of agents, as well as detecting agents which are no longer needed by any application, the monitoring service can increase the robustness of agent applications. The exact mechanisms available to the monitoring service to alter a running system have to be customized carefully for each application to reflect the varying degree of autonomy for each agent.

To support monitoring and reconfiguration of agents and applications it is necessary to provide the responsible monitoring entities with relevant state information about the monitored entities and vice versa to be able to communicate back reconfiguration commands to the relevant agents. In addition, the reconfiguration of a larger application often requires a coordinated set of reconfigurations against the individual agents that constitute the system. Furthermore, reconfigurations need to assure that the system is in a consistent state after the reconfiguration has been performed [17, 18]. These issues are beyond the scope of this paper and will not be further elaborated.

4 Deployment Reference Model

In the following, we describe our approach towards realising the vision of distributed deployment of multi-agent applications. The approach is based on the idea of specialised service agents that are responsible for launching and managing agents and societies on their platform. These service agents are called ASCML (Agent Society Configuration Manager and Launcher). Fig. 1 depicts the deployment reference model. On each agent platform, at least one ASCML agent will be available to manage the societies on that platform. ASCML agents may respond to remote requests, e.g. from other ASCMLs, in order to start (subordinated) society instances remotely. In the reference model, each society instance will be managed by exactly one ASCML. A society instance is a virtual concept only known to the ASCML agent that started it and has no representation on the agent platform. Therefore, societies may easily span across several platforms, having one root ASCML responsible for the whole society instance and local ASCMLs responsible for different subparts. Agent instances (e.g. generic agents such as a yellow page service) may belong to several society instances at once, and therefore - knowingly or not - may be under control of several ASCML agents.

The reference model is able to capture most of the requirements of the last section. The ASCML agent provides the basic management services for starting and

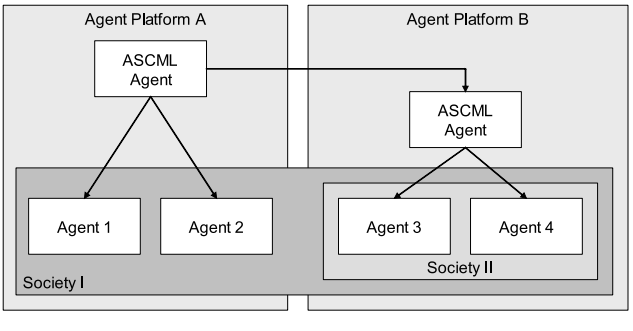

Fig. 1. Deployment reference model

stopping agent and society instances, and is also responsible for launch-process-management as well as monitoring and dynamic reconfiguration. This *external* approach is considered advantageous compared to an *internal* approach where configuration management is built into the single entities [18]. The ASCML is a self-contained component with a standardised interface. Porting the ASCML to different FIPA-compliant agent platforms should be straightforward, making the reference model well suited to achieve deployment capabilities in heterogeneous environments. The reference model does not directly support starting and stopping of remote agent platforms, as an ASCML and a running agent platform have to be present at each network node. To meet this requirement some kind of bootstrapping component would be necessary, which is out of the scope of this paper.

Launching, as well as the planned monitoring and reconfiguration services are based on specifications of agents and societies. To facilitate reusability of specifications a society instance is not defined in one large file, but in two different types of files describing an agent application at different levels. Agent type specifications define self-contained agents at the single-agent level. Society specifications define multi-agent applications by referencing the specifications of included agents and society instances. Both specifications follow an XML schema definition[1] as described in the next two sections.

While we are currently creating the specification files manually, we envisage that graphical user interfaces will be used to compose and configure larger agent applications. Additionally, tools can be developed to crosscheck created specifications for consistency. Once the specifications have been created, the deployment engineer has to take care, that each ASCML agent has access to the specification files for those agents and societies that it has to start on its platform.

4.1 Agent Type Specification

Fig. 2 depicts the structure of an agent type specification. An ASCML agent will read the agent type specification e.g. when it is requested to instantiate an agent of that type. The agent element captures important properties of an agent such as the agent's implementation class and the type, which identifies the required agent platform (e.g. JADE [20]). The single-valued parameters and multi-valued parameter sets represent typed arguments that can be supplied when creating a new instance of the agent. Additionally, it is possible to specify one or more (for parameter sets) default values that are used by the ASCML, when no explicit value is provided for the creation of a specific agent instance. Both kinds of parameters can be further elaborated with additional constraint elements, used for restricting the set of allowed values for the parameter. Furthermore, FIPA-compliant service and agent descriptions [21] can be included in the agent type definition. These allow specifying the services that an instance of this agent type can provide when it is instantiated.

[1] available at http://jadex.sourceforge.net/schemas/

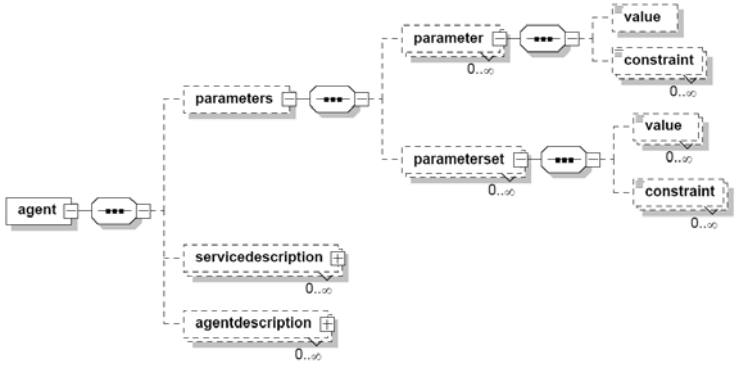

Fig. 2. The agent meta-model

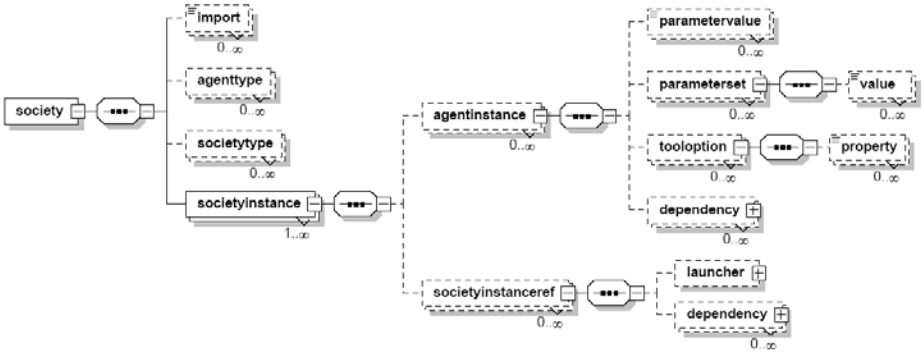

Fig. 3. The agent society meta-model

4.2 Society Type Specification

A society type (see Fig. 3) defines a multi-agent application at the type level. The society contains a declaration part in which all agent types and enclosed subsocieties have to be defined. This declaration part is not necessary from the technical point of view, but it enhances the readability of the application specifications and facilitates model checking by making explicit the available element types. The contained society instances represent different application configurations, whereby each society has to provide at least one default society instance. This society instance will be selected for instantiation when the society needs to be launched without further information available.

A society instance consists of concrete agent instances and subsociety instances that need to be created when the society is started. For the specification of agent instances, at least the mandatory parameter values have to be supplied. Additionally, platform dependant tool options can be specified in a generic way. They can be used to activate tools and e.g. can be utilized to facilitate the de-

bugging process by using agent observation tools such as the Sniffer agent in JADE. Dependency elements are used to establish an implicit ordering of the entities to be started.

In addition to the agents to be created, a society can contain an arbitrary number of subsocieties that can contain further subsocieties as well. This allows a recursive application definition and facilitates the creation of distributed MAS. Each referenced subsociety instance refers to a concrete society instance, which itself belongs to a declared society specification. For the purpose of starting a remote society, a so-called launcher identifier can be declared. This identifier designates the remote ASCML agent responsible for starting the corresponding remote society. In analogy to agent instances, dependencies can be specified for subsocieties as well.

4.3 Dependencies

Dependencies are used to express relationships between elements at the instance level. If one element declares itself dependent from another element this means that the declaring element cannot be started until the referenced element is available. In our model five different types of dependencies can be distinguished: *agent type*, *agent instance*, *society type*, *society instance* and *service dependencies*. An agent type dependency can be used to wait for an arbitrary number of agents of a specified type to be running, while an agent instance dependency exactly refers to a designated agent, identified by its unique name. Both kinds of dependencies also exist for the society element, which means that it is possible to wait for a specified number of societies with a certain type as well as for directly known societies. The last kind of dependency is the most abstract one and allows defining indirect relationships between elements, because the element depends on a service (following FIPA) to be available.

All kinds of dependencies can either be marked active or passive denoting if the ASCML has the duty to actively engage in action when the dependency does not hold. If a dependency is declared *active* the ASCML will try to start missing entities, whereby the mechanism for deciding what instance need to be launched depends on the type of dependency and its parameterisation. In case of a *passive* dependency, the ASCML will wait until the dependency condition holds (e.g. retesting the condition from time to time).

4.4 Example

The following example further explains the meta-model presented in the prior subsections. It relies on a slightly modified version of the *JADE Party* example application provided with the JADE-distribution. In this scenario, guests are invited to a party by an organizer and spread a rumour until it is known by all guests and the party ends (cf. JADE Party Java docs). Hence, the JADE Party consists of two different types of agents, a *Host*- and a *Guest*-agent that make up the basis for the corresponding society type. By defining different settings, e.g. specifying the number of guests taking part in the party, different society instances may be set up.

```
1 <agent name="Guest" package="examples.party" class="GuestAgent" type="JADE"/>
```

```
1 <agent name="Host" package="examples.party" class="HostAgent" type="JADE">
2   <parameters>
3     <parameter name="guestsToWaitFor" type="Integer" optional="false"/>
4   </parameters>
5 </agent>
```

Fig. 4. Guest- and Host-agent type definitions

With the agent type definition all required information for starting an agent of this type is specified (see Fig. 4). The definition of both agent types contains the name, which is used in connection with the package declaration to uniquely identify a model within the ASCML's scope. The class-attribute reflects the agent's implementation class, which is instantiated at the agent's start-up and the type-attribute serves as the agent-platform type identifier (e.g. JADE) and is evaluated by the ASCML to choose among the set of platform-dependent managing-services for starting and stopping agents. Additionally, for the Host agent type one parameter for the number of party guests is specified, in this case obliging the Host not to start the party before at least the specified number of guests has arrived. The parameter is non-optional meaning that a concrete value has to be specified by an agent instance of this type.

Besides the definition of the agent types, an additional definition of the society type, together with a set of society instances is needed (see Fig. 5). It contains

```
01 <society name="BirthdaySociety" package="examples.party">
02
03   [import and declaration of used agenttypes and referenced societies are omitted]
04
05   <societyinstances default="SmallParty">
06
07     <societyinstance name="SmallParty">
08       <agentinstances>
09         <agentinstance name="Birthday Child" type="Host">
10           <parametervalue name="guestsToWaitFor"> 10 </parametervalue>
11         </agentinstance>
12       </agentinstances>
13     <societyinstanceref name="Guests"societytype="BirthdaySociety"societyinstance="SmallGuestpool">
14         <dependency active="false">
15           <agenttype name="Host" quantity="1"/>
16         </dependency>
17         <launcher name="ASCML@remotecomputername:5000/JADE">
18           <address> http://192.168.0.170:5010/acc </address>
19         </launcher>
20       </societyinstanceref>
21     </societyinstance>
22
23     <societyinstance name="SmallGuestpool">
24       <agentinstances>
25         <agentinstance name="Guest No_%N" type="Guest" quantity="10" />
26       </agentinstances>
27     </societyinstance>
28
29   </societyinstances>
30
31 </society>
```

Fig. 5. Definition of the JADE Party-society type

the definition of the *SmallParty* society instance (lines 7-21), which represents the main application and a helper society instance called *SmallGuestpool* (lines 23-27).

One agent instance, named "Birthday Child", is contained within the SmallParty. This instance relies on the agent type Host, indicated by the attribute type (line 9), and therefore has to supply a value for the guestsToWaitFor-parameter (cf. agent type definition). Besides the agent instance also the subsociety *Guests* (line 13-20) is defined as reference to the SmallGuestpool society instance. To make sure the guests do not join the party before the host is ready, a dependency is specified (lines 14-16) forcing the ASCML to first wait for the dependency before going on starting the referenced society instance. Once the dependency is satisfied, the ASCML may try to start the subsociety by sending a request to the launcher (lines 17-19). The launcher, identified by its FIPA-conform name and a set of addresses, has to be an ASCML-agent as well. Assuming this ASCML also has access to the given society instance, it may now start the agent instances contained within the society instance.

The subsociety SmallGuestPool consists of a collection of guests, which are agents of the same type (line 25). For convenience, not every individual agent has to be provided with its own definition. It is sufficient to specify the number of agents contained within the collection by using the quantity-attribute and a naming scheme for enumerating the agent instances.

5 Prototype Realization

The deployment reference model is the basis for the currently developed ASCML prototype. The reference model as described above is platform independent, therefore allowing agent applications not only to be spread across different hosts but also to be composed of agents developed for different platforms. The launcher tool currently exists in two (slightly different) versions, developed for the JADE [20] and Jadex [22, 23] platforms.

5.1 Architecture

The ASCML is subdivided into three co-operating subsystems: the launcher, the repository, and the GUI. To enable subsystems being individually exchanged, modified or enhanced the connection between these components is lightweight based on interfaces and event mechanisms. In the following each of the subsystems is described in more detail and their role within the ASCML's architecture (as depicted in Fig. 6) is highlighted.

The repository-subsystem provides facilities to manage all necessary data used within the ASCML such as agent- and society models, properties and project-management data. The repository is used as an abstract shared data structure and may be accessed by all other subsystems. Furthermore, it is responsible for loading and saving model-objects from and to different data sources, like XML-files or databases. Changes made to the data contained within the repository are acquainted by events to all registered listeners.

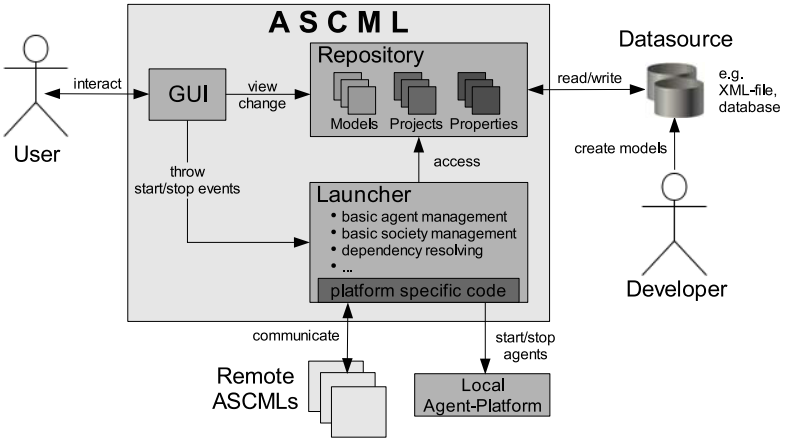

Fig. 6. The ASCML architecture

The GUI-subsystem facilitates the interaction between the user and the underlying subsystems. It provides dialogs to view and change data contained within the repository and allows the user to interact with the launcher to perform actions such as starting and stopping of agent and society instances.

The launcher-subsystem realises the interface between the ASCML and the underlying agent-platform. It is responsible for the basic agent- and society management, which includes starting and stopping of agent instances, delegation of action-requests to remote ASCMLs and resolving dependencies defined by soci-

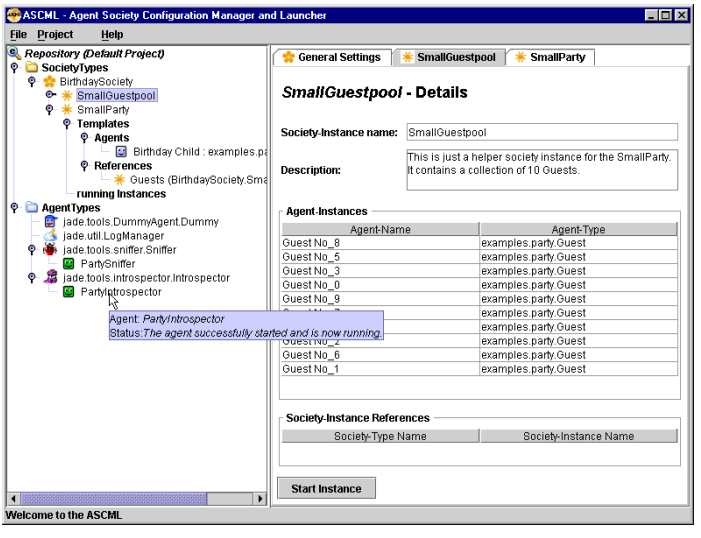

Fig. 7. The ASCML tool screenshot

eties. It encapsulates the logic for communicating with the local agent-platform as well as with remote ASCMLs. Therefore parts of the launcher are platform-dependent, but may easily be exchanged to support different agent-platforms.

5.2 Example Usage

The graphical user interface of our ASCML implementation is depicted in figure 7. On the left hand side, one can see the specification repository tree with some known agent and society types, whereas on the right hand side details of the selected tree element are shown. In this example, the society type called BirthdaySociety and a couple of tool agents are available. In the BirthdaySociety, two different instances (SmallGuestpool and SmallParty) are predefined as ready to run application configurations. In the depicted scenario, two tool agents (sniffer and introspector) already have been started. On the right hand side some details of the SmallGuestpool such as the contained agent instances are presented.

6 Conclusion and Outlook

In this paper, we have argued that deployment techniques are important for the wide-spread and industrial adoption of multi-agent system technology. We have investigated the general requirements and the extent to which existing deployment techniques can be adapted to support the launching and configuration of distributed multi-agent systems.

To address the arising issues we have proposed a reference model that specifies the general launching and configuration infrastructure. The reference model is based on the notions of agents and societies as constituting entities. For the reference model a FIPA-compliant service interface has been designed, which allows (parts of) applications to be started on different hosts and possibly on different platforms. A prototype of the deployment tool (ASCML) has been implemented for the JADE and Jadex frameworks.

Future extensions will be done on two levels. On the conceptual level we will further investigate, which elements and relationships are necessary for the specification of abstract multi-agent applications according to our vision of scalable and adaptive systems. For this purpose, we need to extend our definition of agent societies incorporating more advanced concepts such as roles and constraints, taking into account existing organisational models. The usage of roles promises e.g. to capture the relationships between agents at a more abstract level enabling dependencies to be specified between roles and not only at the agent instance level. The introduction of application constraints will not only leverage the abstraction level of the application specification, but also can be seen as a starting point for dynamic application reconfiguration. This is because the configuration environment could use these constraints to ensure certain properties of the application and engage in appropriate actions whenever this becomes necessary.

On the tool level, the ASCML will be extended to live up to its name by introducing user interfaces for the easy construction of agent-based applications. This will further improve the tool's usability and additionally can be exploited to reduce the number of application specification mistakes. Monitoring capabilities (e.g. observing the lifecycle state of agents) will be added to the tool to facilitate automatic reconfiguration of running applications.

Acknowledgements

This work is partially funded by the DFG German priority research programme SPP 1083: *Intelligent Agents in Real-World Business Applications*.

References

1. Jennings, N.R.: An agent-based approach for building complex software systems. Communications of the ACM **44** (2001) 35–41
2. Jennings, N.R., Wooldridge, M.J.: Agent Technology - Foundations, Applications and Markets. Springer Verlag (1998)
3. Arlow, J., Neustadt, I.: UML and the Unified Process: Practical Object-Oriented Analysis and Design. Addison-Wesley (2002)
4. Sudeikat, J., Braubach, L., Pokahr, A., Lamersdorf, W.: Evaluation of Agent-Oriented Software Methodologies – Examination of the Gap Between Modeling and Platform. In: Proc. of the 5th Int. Workshop on Agent-Oriented Software Engineering (AOSE-2004). (2004)
5. (OMG), O.M.G.: Deployment and Configuration of Component-based Distributed Applications Specification. (2003) `http://www.omg.org/`.
6. Ricordel, P., Demazeau, Y.: From analysis to deployment: A multi-agent platform survey. In: Engineering Societies in the Agents World, Springer-Verlag (2000) 93–105
7. Mitkas, P.A., Kehagias, D., Symeonidis, A.L., Athanasiadis, I.N.: A framework for constructing multi-agent applications and training intelligent agents. In: Proc. of the 4th Int. Workshop on Agent-Oriented Software Engineering (AOSE-2003). (2003) 96–109
8. Systems, R.: AgentBuilder User's Guide. (2000) `http://www.agentbuilder.com/`.
9. Nwana, H., Ndumu, D., Lee, L., Collis, J.: ZEUS: a toolkit and approach for building distributed multi-agent systems. In: Proc. of the 3rd conference on Autonomous Agents, ACM Press (1999) 360–361
10. Collier, R.W.: Agent Factory: A Framework for the Engineering of Agent-Oriented Applications. PhD thesis, University College Dublin (2001)
11. Cowan, D., Griss, M., Burg, B.: BlueJADE - A service for managing software agents. Technical Report HPL-2001-296R1, Hewlett Packard Laboratories (2002)
12. Ferber, J., Gutknecht, O., Michel, F.: From Agents to Organizations: an Organizational View of Multi-Agent Systems. In Giorgini, P., Mller, J., Odell, J., eds.: AOSE. Volume 2935 of Lecture Notes in Computer Science., Springer (2003) 214–230
13. Odell, J.J., Parunak, H.V.D., Fleischer, M.: The role of roles in designing effective agent organizations. In: Software Eng. for Large-Scale MAS, Springer (2003) 27–38

14. Giampapa, J., Juarez-Espinosa, O., Sycara, K.: Configuration Management for Multi-Agent Systems. In: The 5th International Conference on Autonomous Agents (Agents 2001), ACM Press (2001) 230–231
15. Sycara, K., Giampapa, J., Langley, B., Paolucci, M.: The RETSINA MAS, a Case Study. In: Software Engineering for Large-Scale Multi-Agent Systems: Research Issues and Practical Applications. Volume LNCS 2603. Springer-Verlag (2003) 232–250
16. Foundation for Intelligent Physical Agents: FIPA Agent Configuration Management Specification. Document no. FIPA00090 (2001)
17. Castaldi, M., Carzaniga, A., Inverardi, P., Wolf, A.: A Light-weight Infrastructure for Reconfiguring Applications. In Westfechtel, B., van der Hoek, A., eds.: Software Configuration Management, ICSE Workshops SCM 2001 and SCM 2003, Springer (2003)
18. Castaldi, M.: Dynamic Reconfiguration of Component Based Applications. PhD thesis, Department of Computer Science, University of L'Aquila, Italy (2004)
19. Sloman, M.: Management issues for distributed services. In: Proc. of the 2nd Int. Workshop on Services in Distributed and Networked Environments, IEEE (1995) 52–55
20. Bellifemine, F., Rimassa, G., Poggi, A.: JADE – A FIPA-compliant agent framework. In: 4th Int. Conf. on the Practical Applications of Agents and MAS (PAAM-99). (1999)
21. for Intelligent Physical Agents, F.: FIPA Agent Management Specification. Document no. FIPA00023 (2002)
22. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: Implementing a BDI-Infrastructure for JADE Agents. EXP – in search of innovation **3** (2003) 76–85
23. Braubach, L., Pokahr, A., Lamersdorf, W.: Jadex: A Short Overview. In: Net.ObjectDays 2004: AgentExpo. (2004) 76–85