A Survey of Agent-oriented Development Tools

Alexander Pokahr and Lars Braubach

Abstract Development tools represent an important additive for the practical realization of software applications, mainly because they help automating development activities and are able to hide complexity from developers. In this chapter, the requirements for tools are generically analyzed by the various tasks that need to be performed in the different development phases. These requirements are the foundation for a detailed investigation of the landscape of available agent-oriented development tools. In order to assess the variety of tools systematically, existing surveys and evaluations have been used to isolate three important categories of tools, which are treated separately: modeling tools, IDEs and phase-specific tools. For each of these categories specific requirements are elaborated, an overview of existing tools is given and one representative tool is presented in more detail.

1 Introduction

The term *tool* is defined in dictionaries as a means used in performing an operation or task. In computing, a (software) tool is therefore a software for

Alexander Pokahr
Distributed Systems and Information Systems Group,
Computer Science Department, University of Hamburg,
Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany,
e-mail: pokahr@informatik.uni-hamburg.de
Lars Braubach
Distributed Systems and Information Systems Group,
Computer Science Department, University of Hamburg,
Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany,
e-mail: braubach@informatik.uni-hamburg.de

developing software or hardware.[1] As the product that is developed with a software tool is itself again a piece of software, we further want to restrict our discussions on tools to so called *development tools* . A development tool is a software that is used by a software developer to produce some result (e.g. a text editor used for editing a source file). Unlike other kinds of software (e.g. libraries or agent platforms), a development tool is only used during the development of a system and not part of the final software product.

The results presented in this chapter are part of a larger survey on agent-oriented development artifacts. Specifically, agent architectures, languages, methodologies, platforms and tools have been researched. For the evaluation of the surveyed representatives, a criteria catalog has been developed, which covers besides functional criteria also non-functional issues such as usability, operating ability and pragmatics. Details of the criteria catalog as well as condensed and summarized survey results can be found in [14]. The criteria catalog will be used in this chapter as a guiding principle for discussing requirements with respect to tool support. More information on the survey is available in [11], covering architectures, methodologies and platforms, and in [47], dealing with languages and tools.

In the next section, the background on software development tools will be presented, thereby highlighting general requirements and providing a model for assessing tool support for the different phases in the software development process. Section 3 deals with agent-oriented tools. In this section, a survey about existing agent-oriented software development tools will be given. Thereafter, in Sections 4 and 5, two important categories of tools – namely modeling tools and integrated development environments (IDEs) – are discussed in detail. In Section 6, tools for individual phases of the development process are presented. A short evaluation of the presented state-of-the-art is given in Section 7. The chapter closes with a summary and conclusion.

2 Background

The following sections discuss which kinds of tools are employed in the different phases of the software development process. For generality and simplicity, a basic and well-known five phase model [6] is used as a foundation instead of a concrete and detailed agent-oriented methodology such as Gaia [73] or Prometheus [45]. The five phase model distinguishes between 1) requirements, 2) design, 3) implementation, 4) testing, and 5) deployment.

First, an overview of common development tasks in each of the phases will be given (Section 2.1). For a structured and systematic discussion, these tasks are then unified according to a generalized classification scheme (Section 2.2). Further, kinds of tools for supporting the tasks as well as criteria

[1] Wiktionary.com: "tool" (5 October 2008), `http://en.wiktionary.org/w/index.php?title=tool&oldid=5262373`

for assessing the quality of tool support are presented in Section 2.3. The following discussions are intentionally kept general (i.e. not specific to agent-oriented software development) to avoid an isolated "agents only" view and for facilitating a comparison of the state of the art of agent tools with respect to the state of the art in software engineering in general.

2.1 Development Tasks During the Software Engineering Process

This section describes the tasks, a developer has to perform during the different phases of the software development process. Usually, these tasks correspond to single steps, which have to be conducted more than once during iterative refinements.

The *requirements* phase is necessary to elaborate the requirements of the software to be. This phase involves talking to end users and customers to identify the needs and wishes. These have to be analyzed for being able to write them down in a precisely defined and unambiguous form. The elaborated requirements also have to be checked for consistency to each other and the requirements specifications have to be validated with the aid of the customers and end users.

After the requirements have been fixed, the *design* of the system can start. The design phase has the goal to develop a blueprint of a system that captures all identified requirements. During the continuous refinements of the design it should be checked for consistency of the design artifacts to each other. Moreover, the design should be validated with respect to the identified requirements, such that problems in the development process can be detected early.

Tasks during the *implementation* phase mainly consist in editing the source code. This includes, besides creating new code fragments, also the task of refactoring, which is a systematic restructuring of the source with the aim of preserving the existing functionality but at the same time e.g. better supporting the integration of planned future functionality. Depending on the level of detail in the design, code generation can be used to produce initial code fragments automatically based on the design information. For iterative development processes it is in this case necessary that changes to the code are also reflected in the original design artifacts, e.g. by using reverse engineering technologies. Another important task during the implementation phase is producing documentation to keep the code base maintainable and understandable. Therefore, decisions should be documented, which are necessary because of a higher abstraction of design artifacts compared to the concrete code. Especially, the concrete interfaces between modules of the system should be described, as in larger projects these modules are often developed by independent developer teams.

In accordance with the V-model [22, 52], the steps of system *testing* mirror the steps of system design in the opposite direction. System design moves from abstract requirements to detailed design-specification and finally concrete code. To validate an implementation, these steps should be taken backwards. Therefore one starts validating concrete implementations of partial functionalities (e.g. by so called unit testing). Validation errors occurring in this step usually can be corrected directly in the code. When the functionality of single components is verified, the correct interplay between these components can be validated (integration testing). This shows, if design specification (e.g. interface definitions) are sufficient to ensure the smooth integration of components. If validation errors occur at this stage, often design decisions have to be revised and implementations adapted accordingly. Finally, a validation of the system as a whole is performed with respect to the initially identified requirements. In so called system tests, the developers can play through the defined use cases. During acceptance tests, the system is evaluated by the real end users and customers. Problems, which are identified in these tests, form the requirements that are used in the next iteration of the development process.

The *deployment* of a software system follows a sequence of several steps (cf. [40]). Because systems are usually not developed from scratch, the first step is to obtain/provide the required additional components. Thereafter, the obtained and newly developed components have to be configured according to the intended usage contexts resulting in a set of application configuration specifications. For each application configuration, a deployment plan needs to be devised. In the preparation step, the code of the application components will be placed at the required target locations. When all components are installed, the application can be started, meaning that for a complex distributed application several components on potentially different network nodes need to be started. Once the application is running, the maintenance phase starts, during which e.g. changes to the application configurations can be made. Such changes may not require performing a new iteration of the development process, if they have been already considered in the system design. Depending on the runtime infrastructure may be possible while the application runs or may require the application to be restarted. For unforeseen changes, the development process has to start again for designing, implementing, testing and deploying changed components according to the new requirements.

Besides these tasks corresponding to the five development process phase, there are other cross-cutting tasks, which have to be conducted during the whole development process. Among such tasks are the provision, management and usage of a repository for holding and providing consistent access to the different versions of the produced specifications and code. Also the coordination among software developers is a task that is required in all phases.

2.2 Classification of Software Engineering Tasks

The aim of the last section was to give an overview of the different tasks and activities that have to be performed during the software development process. This section investigates how these tasks can be supported by software tools. To keep the discussion general it is abstracted away from concrete tools and specific tasks. Instead it is tried to identify the commonalities for tasks that recur in similar forms in different phases. This investigation helps to identify the kind of tool support that is required in general and also sheds some light on the relations between different kinds of tools.

A unification and categorization of the tasks from the last section is illustrated graphically in Figure 1. The five phases of the software development process (*requirements*, *design*, *implementation*, *testing*, *deployment*) are shown from the left to the right. From top to bottom, you can find a classification according to *primary tasks*, *ancillary tasks*, and *cross-cutting tasks* (see left hand side legend).

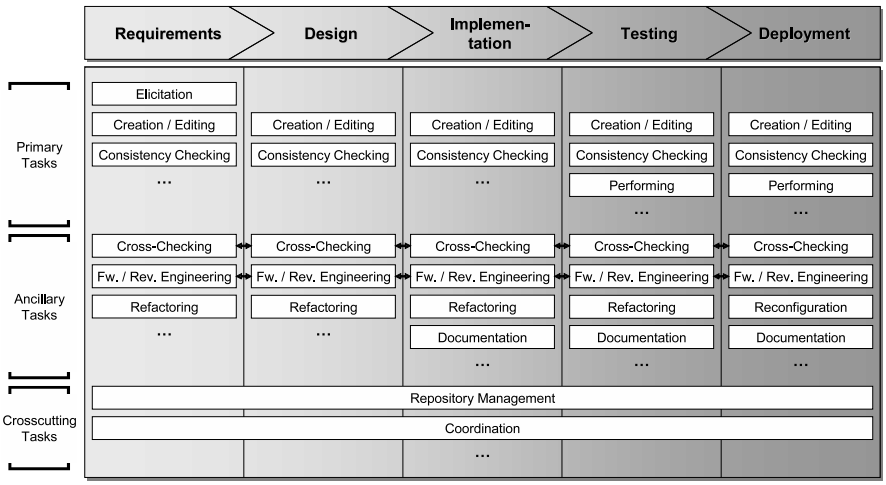

Fig. 1 Tasks in the software engineering process

Primary tasks are those tasks, that form the major part of a phase. Therefore, such tasks should if possible be supported by a single integrated tool to avoid having to switch often between different work environments. This means that, e.g., a design tool should support all tasks of the design phase in an integrated and consistent way.

Supporting tasks, which are optional or required less often compared to primary tasks are termed ancillary tasks. Because these tasks make up only a low portion of the overall development effort, requirements for tool support for these tasks are somewhat reduced. E.g. support for some of these tasks

need not be part of an integrated tool but can also be realized in several independent tools, without causing too much interruption in the workflow of the developer. Nevertheless, an integration of such tools would be beneficial, e.g. in the form of plug-in mechanisms that allow to invoke external tools from an integrated development environment.

Finally, cross-cutting tasks are not associated only to a single phase of the development process. Therefore, tool support for these tasks should be realized separately from any tool that is only intended for a specific phase so as not to require the use of tools from different phases. E.g., repository management support should not be realized solely as part of a design tool, otherwise developers would be required to always have the design tool at hand even in the later development phases. Nevertheless, some integration of the functionality (e.g. using plug-ins) into phase specific tools can be beneficial as long as it does not hinder the consistent usage across all phases.

In the following, the concretely identified generalized tasks as shown inside the Figure 1 will be explained in more detail. In the figure, similar tasks are subsumed under a common name. Therefore, a common primary task for all phases is the *creation and editing* of artifacts, where artifacts depend on the phase (e.g. requirements specifications, design models, application code, test cases, or deployment descriptors). Also, the *consistency checking* refers to the artifacts of the respective phases, e.g. checking different design models for consistency to each other. Some tasks such as *elicitation* of requirements have no counterpart on other phases, as the later phases, the documents from the earlier phases are directly used as input. Similarly, the *performing* of artifacts such as test cases and deployment plans only happens in the last two phases respectively, because in the earlier phases, the produced artifacts (design models or code) directly form the desired result, while in the last two phases, the artifacts are only means to the final goal of a validated resp. installed system.

Unlike the primary task of consistency checking, the ancillary *cross-checking* task refers to checking the consistency between artifacts of different phases (as indicated by small arrows in the figure), mostly checking the newly produced artifacts of the current phase for consistency with the artifacts from the earlier phase(s) to verify that, e.g., design models capture all previously defined requirements. In a similar way, also forward and reverse engineering (*fw. rev. engineering*) has to consider artifacts from different phases. The aim is to automatically create artifacts for one phase out of the information available in artifacts in an earlier (forward engineering) or later phase (reverse engineering). A common form of this task is code generation, which produces an implementation phase artifact (code) based on some design phase artifacts (design models). *Refactoring* is the task of systematically changing a set of artifacts with respect to a common restructuring goal. Systematic means that changes apply to many artifacts at once and special care has to

be taken to ensure the consistency of the artifacts.[2] These tasks are termed ancillary because they are partially optional (fw./rev. engineering and refactoring) or are only require limited amount of effort (cross-checking) compared to the primary tasks. Another ancillary task that requires limited amount of effort is producing *documentation* for the specific phase. Because the major artifacts of the requirements and design phase have documentary character themselves, a separate documentation task is not considered for these phases, in other words, (creating/editing of) documentation is a primary task in these phases.

Cross-cutting tasks are also associated to the artifacts of the different phases. E.g. the aim of *repository management* is to store the artifacts and their changes and to provide access to them, when needed. Besides the artifacts itself, meta-information such as the user, version and time of a change needs to be stored. Also the *coordination* is usually tightly coupled to the creation and editing of artifacts. E.g. it has to be coordination who is allowed to edit which artifact (access management) and who is responsible for creating which artifact (task allocation).

The dots in the categories indicate that the figure is not claimed to be complete with respect to all possible kinds of software development tasks. For example the list of tasks could easily be extended to tasks, which are less focused to direct development tasks, such as e.g. project management or quality assurance.

2.3 Tool Support for Development Tasks

For an effective and efficient software development it is essential that preferably all tasks and activities during the development process are adequately supported by tools. Nowadays, a huge amount of vastly different tools has been developed in research and industry. Grundy and Hosking [30] have given a broad overview over the state of the art in the area of software tools. Their overview considers tools in general (i.e. not specifically focused on agent-oriented application development). Grundy and Hosking identify 18 different kinds of tools (e.g. design tools, IDEs, as well as testing and debugging tools) and describe the phases in the development process, where these tools are used. The considered tools usually support more than a single activity or task inside a phase like, e.g. design tools, which besides the creation and editing of design models also often support consistency checking and/or code generation. Moreover, some tools can be used across different development phases. For example, many IDEs not only address the implementation phase,

[2] Although refactoring is often referred to in the context of code refactoring only, it is also possible to generalize the refactoring idea to other kinds of artifacts. For example, [61, 7] propose refactoring mechanism for UML design models.

but also offer support for testing and debugging as well as sometimes aspects of deployment.

The quality of any tool support can be assessed by considering the degree of support for the different phases and tasks. The support for all tasks and activities in the sense of a complete tool-support for the software development process can be achieved on the one hand by combining a multitude of specialized tools for single tasks or on the other hand by a few powerful tools, each of which addresses a large portion of software development tasks. Besides this functional quality aspects, also non-functional quality criteria, such as usability, operating ability and pragmatic aspects (cf. [14]) should be considered when evaluating or designing tool support. For these criteria, the continuity of the tool support is of primal importance (cf. [51]). In this respect, continuity refers to the seamless working with the same artifacts across different interrelated tasks. This continuity can easily be obtained, if support for related tasks is combined in a larger tool (e.g. design tool or IDE). When related tasks are supported by separate tools, the continuity needs to be achieved by an integration of these tools. A fully integrated tool support also directly improves usability, because it provides a unified view of the development process reducing the learning effort and the potential for errors.

According to Figure 1, integration can be pursued across two axes. On the one hand, integration can consider two tasks from different phases (horizontal integration). For example design and implementation tools, which are both responsible for creating and editing artifacts could be integrated by providing an interface for data interchange. On the other hand, tasks from the same phases can be integrated (vertical integration), like combining support for these tasks inside a common usage interface, e.g. using a plug-in mechanism. Grundy and Hosking [30], differentiate four basic ways of integration: *data integration*, *control integration*, *presentation integration* and *process integration*. Data integration is achieved by the already mentioned data interchange interfaces and can be based on standardized as well as proprietary data formats. Data interchange is essential to allow consistency checking among artifacts created with different tools (especially for cross-checking artifacts from different phases). Control integration allows redirecting commands issued in one tool to another tools. As an example consider a debugger and a source code editor, where the debugger has to tell the editor, which line to show, when the developer issues a program step. Presentation integration has the goal to combine the functionality of different tools in a unified user interface, e.g. simply by invoking the command line tools such as CVS/SVN and present their output or by using sophisticated plug-in facilities that allow to extend also the user interfaces, e.g. of integrated development environments. Finally, process integration focuses on integrating subsequent activities or steps. Therefore, process integration has to combine data, control, and presentation integration and adds knowledge about the development process and the interdependencies of process steps, i.e. process integration automatically

presents to the developer the right tool with the right data for the next required working task.

To summarize the preceding analysis of tools and tool support it is noted that, according to Section 2.2, the quality of tool support and integration is more important for the primary tasks than for ancillary tasks. With respect to the goal of achieving a complete and continuous tool supported development process this means that modeling or design tools as well as IDEs are the most important class of tools, as these tools aim to combine and integrate most of the primary tasks and also many ancillary tasks from the requirements and design as well as implementation testing and deployment phases. Moreover, cross-cutting tasks should be supported by separate tools, which are not bound to a specific development phase. In the following, it will be investigated, how these requirements for design tools, IDEs, and tools for cross-cutting tasks can be met by existing agent-oriented software tools.

3 Agent-oriented Development Tools

The previous sections of this chapter have taken a general viewpoint towards tools. This section investigates, which kind of tools exist in the specific area of agent-oriented software engineering. For this investigation, existing surveys and online resources on agent software are used as a starting point. As some of these surveys have a quite specific focus, the results of them cannot be easily compared to each other. Particularly, some surveys consider quite different kinds of agent software, not limited to pure development tools as defined in the introduction, but often also agent platforms and execution environments. Moreover, it should be distinguished between generic software and software that is targeted to a specific application domain or category. Such category-specific software e.g. supports the creation of virtual characters or the building and execution of simulation experiments. In addition, some of the surveys also include built agent-oriented applications. To give a coherent view of existing agent software, the following analysis presents not only development tools, but also runtime-relevant software like agent platforms and support libraries. Further categories of software are introduced as needed, when they are present in some survey or online resource. Nevertheless, in the subsequent discussions the scope will again be reduced to development tools.

3.1 Analysis of Existing Surveys and Online Resources

The examined surveys differ in the selection of tools as well as in the definition of the investigated categories. Some investigations only define a single category and only study representatives of this category. Other surveys have

the aim to be broader and therefore examine different representatives of categories, which are defined in advance or afterwards.

Among the surveys focused on a single category, Eiter and Mascardi [23] and Bitting et al. [5] only consider environments for developing agent applications. The term multi-agent system development kit (MASDK) is introduced to denote integrated development environments with functionalities similar to object-oriented IDEs, such as eclipse[3] or IntelliJ IDEA[4]. Considering the examined development environments such as AgentBuilder [53], IMPACT [21], JACK [71], and Zeus [39], it can be noted that each of them introduces a new proprietary programming language for agent specification. In contrast, object-oriented IDEs usually support existing languages like C++ and Java. This difference is probably due to the fact, that in the area of agent technology no broad consensus exists about how to implement the agent specific concepts leading to quite different approaches with their own respective advantages and drawbacks. The use of proprietary concepts and languages forces these development environments to also include runtime components such as an agent platform for supporting the execution of the developed agents. Runtime environments resp. platforms for executing agents are the focus of Serenko and Detlor [56], Fonseca et al. [24] as well as Pokahr and Braubach [48, 13]. These surveys consider in addition to platforms as part of a development environment also pure execution environments like JADE [3] and ADK [64]. These platforms do not introduce new proprietary programming languages, but instead rely on existing object-oriented languages such as Java.

The respective aims of the broader surveys are sometimes quite difficult to define. E.g. Mangina [36] considers agent software in general, based on the entries of the AgentLink agent software directory at that time. The survey includes 36 representatives, but partitions them in quite vaguely defined categories such as "development environment" or "support software". Newer reviews of Bordini et al. [9, 8] and Unland et al. [65] consider current software from the area of agent-oriented software engineering categorized, e.g., in languages, platforms, and applications [9].

3.1.1 Agent Software in the AgentLink Software Directory

The most current and comprehensive overview over agent software is the publicly available AgentLink agent software directory.[5] It was initiated in the context of a series of EU-funded research networks. Although AgentLink ended in 2005, the list has still been updated since then.[6] With a total of

[3] http://www.eclipse.org

[4] http://www.jetbrains.com/idea/

[5] http://eprints.agentlink.org/view/type/software.html

[6] Last entry was added on September 10th, 2007.

125 entries[7] it is therefore much more up-to-date than other online resources, which seem to be no longer maintained, such as the UMBC Agent Web software directory (149 entries until 2003)[8] or the even no longer available Université Laval Agents Portal (40 entries until 2006) or MultiAgent.com (35 entries until 2007).

An in-depth analysis of the representatives listed in the AgentLink directory results in the chart shown in Figure 2. For the chart, each of the entries has first been assigned to one of the major groups introduced in the beginning of this section, namely *tools* (left hand side), *runtime* software (middle) and software for a *specific application category* (right). The tools group is subdivided into *IDEs*, *design* tools, as well as *other* tools. In the runtime software group it is further distinguished between complete *platforms* and additional supporting *libraries*, which do not form a platform in their own respect. Category-specific software comprises runtime environments for simulation (*sim.*), applications (*app.*) and miscellaneous agent software (*misc.*), which does not fit into any other category, such as libraries for developing virtual characters.

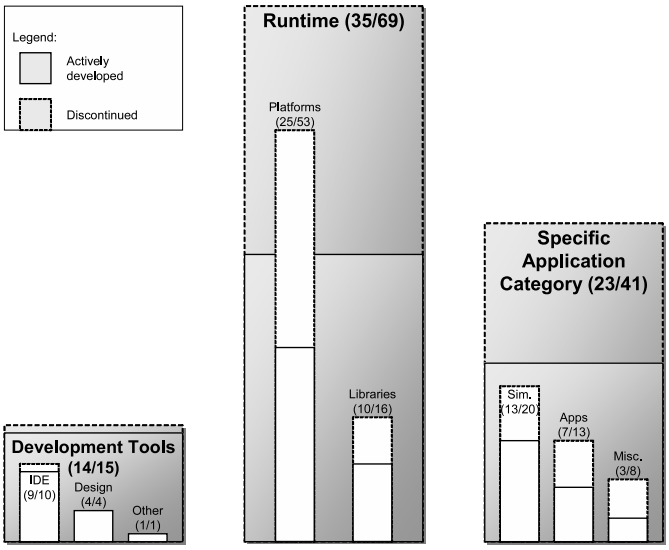

Fig. 2 Analysis based on AgentLink tool directory

For valid conclusions about the current state of the art, it has also been investigated, which of the representatives are still actively developed. A representative is termed inactive, if there has not been a software update or a

[7] Of the actually 128 entries, three have been identified as duplicates.

[8] `http://agents.umbc.edu/Applications_and_Software/Software/index.shtml`

publication about the software in the last two years. The number in braces give the exact number of representatives, whereas the first number is only the active representatives, while the second number is the total number (i.e. active and inactive) representatives.

Considering the distribution of entries into the three major groups, it can be observed, that runtime software is by far the most common group (69 of 125, i.e 55%). The smallest part is tools with only 15 of 125 entries (12%). The reason for this distribution could be that agent programming is still young compared to e.g. object-oriented programming and therefore the main focus of research and development has initially been on the basic infrastructure for execution, and somewhat less on abstract and easy to use development tools. When considering only actively developed representatives, the picture changes in favor of the tools, which now make up 20% (14 of 72) compared to now only 48% for runtime software, while the third category stays around the same level of 32%. This result could be an indication that the available runtime infrastructure has matured in the recent years as now the focus has shifted towards higher level tools.

Concerning the subdivisions in the major groups, it can be noted that platforms are by far the most commonly developed kind of agent software (53 of 125 entries, i.e. 42%). A reason for this might be the already mentioned lack of consensus among agent researchers forcing many research teams to develop a platform on their own instead of reusing existing software. Also, simulation software makes up a considerable portion (16%), which is even higher than the tools group in total and indicates that agent technology is already quite well accepted in the area of simulation. Another noticeable fact is that IDEs are developed twice as much compared to design tools. A possible reason for this could be that the proprietary programming languages often require developing new IDEs while for modeling techniques, which are not specifically agent-oriented, existing tools (e.g. UML tools) can partially be used.

Finally, the reduction of the subcategories to only active representatives is explained. It confirms the picture already present with regard to the major groups, namely, that fewer representatives of runtime software like platforms and libraries remain compared to, e.g. IDEs and development tools. The biggest reduction is in the area of platforms, where more than half of the platforms (54%) are no longer developed. This indicates this area has matured and a convergence to a few widely used platforms, such as JADE, has happened. Additionally, only few of the platforms have been developed in a commercial setting (ca. 10 of 53, i.e. less than 20%) while the majority of IDEs (7 of 10, i.e. 70%) have a commercial background.[9]

[9] This is also due to the fact that many commercial agent platforms like JACK include IDE support and have therefore been assigned to the IDE category.

3.1.2 Tool Kinds for Supporting Agent-oriented Software Development

The goal of the previous sections has been to identify the kinds of tools, which are required and already used for supporting the development of agent-based applications. Therefore, Section 2.1 has presented an overview of software development tasks, which have been generalized and classified in Section 2.2, according to process phases, as well as primary, ancillary and cross-cutting tasks. Moreover, Section 2.3 has discussed how these single tasks should be addressed by integrated (sets of) tools. Finally, Section 3.1.1 has given an overview of tools and other development supporting software in the area of agent technology. This overview identifies the important classes of IDEs and design tools and therefore fits well with the generic analysis from Section 2.3, which identifies modeling tools and IDEs as the basis of a continuous tools support, augmented by additional tools for project management, coordination and special purpose tasks. This leads to the question, for which tasks specific agent-oriented tools are necessary and for which other tasks existing tools e.g. from the object-oriented world would be sufficient. An important criterion for this decision is the kind of artifact, that is manipulated by a tool, i.e. for working with agent-specific artifacts like design models or program code specific agent tools would be advantageous. Cross-cutting tasks (cf. Figure 1) like project or repository management abstract away from concrete artifact types and therefore can be adequately supported by existing tools, such as Microsoft Project[10] or CVS[11].

Therefore the identified modeling tools and IDEs form the most important aspect of a tool-supported agent-oriented software development process to adequately support the requirements and design, as well as implementation, testing and deployment phases. In the following two sections, these two tool kinds will be analyzed in more detail, by discussing the common properties and giving an overview of typical representatives.

4 Modeling Tools

Agent-oriented graphical modeling tools are developed to support the software engineer during modeling tasks and to simplify the transition from an abstract specification to an implemented multi-agent system. Replacing or augmenting existing object-oriented modeling techniques as e.g. available in UML [41], new agent-oriented diagram types are introduced, which allow to specify e.g. interaction protocols or describe internal agent properties at the abstraction level of graphical modeling. A modeling tool realizes the corre-

[10] http://microsoft.com/office/project

[11] http://www.nongnu.org/cvs/

sponding user interface for working with these diagram types. The graphical representation of system properties allows visualizing interdependencies between the elements and improves the developer's understanding of the structure of single agents as well as the system as a whole.

Graphical agent-oriented modeling techniques usually are not self-contained (with the exception of AUML [42] for specifying agent interactions), but rather are embedded into complete software engineering methodologies (see, e.g., [58] or [31] for an overview). Methodologies provide besides modeling techniques also a development process, in which the single techniques are embedded. The development process defines a sequence of steps, which have to be passed through during the realization of a system, and the techniques to be employed in each of the steps [59]. Regarding this aspect, some methodologies are more strict than others, i.e., some restrict single techniques to be only used in some of the steps, while others propagate an iterative refinement of specifications in subsequent steps using the same modeling technique. This strictness can be supported by a tool by offering only those modeling techniques, which correspond to the current process step.

4.1 Requirements for Modeling Tools

This section discusses the specific requirements for modeling tools by referring to the general discussions from Sections 2.2 and 2.3. With respect to development tasks (cf. Figure 1), modeling tools address the design phase as well as (sometimes) the requirements phase. Artifacts of these phases are graphical models and text-based specifications, which can be written in natural language or follow a predefined (formal) scheme (e.g. role schema definitions in GAIA [73]). The main function of a modeling tool is to enable the developer to create and edit these models and specifications. Depending on how strict or formal the models and specifications of the employed technique or methodology are, a tool can also check the consistency of created artifacts and suggest changes for improving the specification (so called design critics [55]).

Among the further tasks in the requirements and design phases is validating specifications or modeling artifacts from different phases with respect to each other. E.g., it can be checked that design documents adequately reflect the scenarios, which have been identified in the requirements phase [2]. Moreover, it is advantageous, if a tool provides the developer with an option to transform artifacts from one phase into artifacts of another phase. For instance, a tool might be able to generate code fragments based on design information (forward engineering) or extract design information out of existing application code (reverse engineering). A drawback of forward or reverse engineering techniques is that after a once generated artifact has been changed manually, forward or reverse engineering cannot be reapplied without loos-

ing the changes, i.e. the so called "post editing problem" [62]. The combined support of forward and reverse engineering, such that changes in one artifact can always be merged into the other without compromising consistency or loosing changes, is called round-trip engineering. Round-trip engineering allows employing forward or reverse engineering techniques also in iterated and agile development processes, where existing implementations are used as a basis for the design of the next iteration. Refactoring techniques are also most useful in agile development processes. Initially only applied to the implementation phase, refactoring ideas have recently also been transferred to graphical modeling [61, 7].

Among the non-functional criteria, especially the group of usability criteria (cf. [14]) requires a specific treatment. To evaluate the usability of graphical modeling tools, the ergonomics of the user interface is of primary importance. In general, this covers properties, such as the suitability for the intended task and controllability (cf. ISO 9241-110 "Ergonomics of human-system interaction - Dialogue principles" [32]). In the specific context of modeling tools, these properties can be refined to concrete requirements. For instance, a tool should relieve the developer by automating tedious tasks like the uniform and clearly arranged placement of diagram elements, but without posing unnecessary restrictions on the user. Moreover, it should be possible to take back actions in case of undesired effects (undo), and often used functionality should be easily accessible without forcing the developer to repeatedly change between mouse and keyboard (e.g. by enabling commands to be issued through hot keys as well as dialog elements).

4.2 Existing Agent-oriented Modeling Tools

The analysis of existing agent-oriented tools from Section 3.1.1 has shown that the choice of tools in the area of agent technology is somewhat limited. Most tools evolved in the context of a specific project or product. Hence, for each specific approach, such as a concrete agent-oriented programming language or development methodology, usually only a single tool (if any) is available, which is highly tailored for this specific approach.

Due to this situation, it seems appropriate to examine tools not in isolation, but to also consider the project or product context. Figure 3 shows current agent-oriented design tools and highlights their interdependencies to other agent-oriented development artifacts. In the figure, design tools are depicted in the highlighted column in the middle. To the left hand side of the design tools, their conceptual foundations are given in the form of development *methodologies*. Methodologies in turn are related to their originating software engineering (*SE*) *approaches*, which, according to [14], are given as agent orientation (*AO*), object orientation (*OO*), knowledge engineering (*KE*), and requirements engineering (*RE*). To the right of the design tool

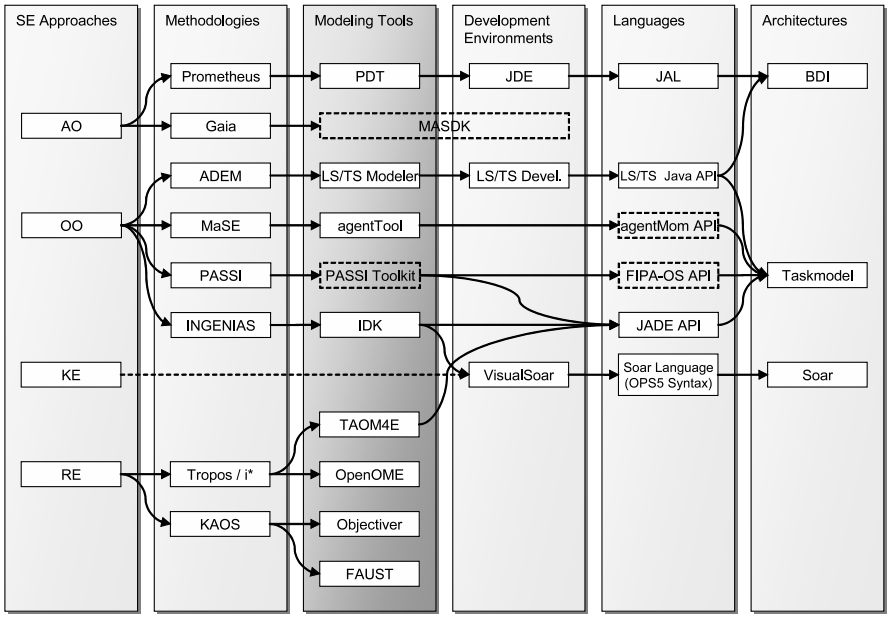

Fig. 3 Modeling tools and relations to other artifacts

column, interdependencies to implementation aspects of the modeled agents can be seen. Those interdependencies go through the *development environments* (i.e. IDEs), which are also covered in this chapter, and programming languages to the behavioral agent architectures, which in turn form the conceptual foundation of the programming languages. For reasons of clarity, artifacts without relations to modeling tools are not shown in this figure, even if they will be covered later with regard to IDEs (e.g. 2APL or AOP languages and associated tools).

All modeling tools shown in the figure have been developed to support a concrete methodology, but not all of them are related to some agent-oriented IDE. The relation to an IDE follows from the fact that a tool is able to generate code of a specific agent programming language or platform, supported by this IDE. For example this is the case for the Prometheus Design Tool (PDT)[12] [63] and the LS/TS Modeler, which are developed for the Prometheus methodology [72] resp. the Agent DEvelopment Methodology (ADEM, [67]), are able to generate code for the language JAL of the JACK agent platform [71] resp. the Java API of the Living Systems Technology Suite LS/TS [54]. The INGENIAS Development Kit (IDK)[13] [27] supports the INGENIAS Development Process IDP [46] and generates code for two

[12] http://www.cs.rmit.edu.au/agents/pdt/

[13] http://ingenias.sourceforge.net/

languages/platforms (JADE [3, 4] and Soar [34]), one of which (Soar) is supported by a specific IDE.

Other tools, such as the PASSI Toolkit[14] and agentTool[15], which have been developed for the PASSI [16] resp. the MaSE methodology [20], are able to create code from the models, but no specific IDEs exist for the target languages. Especially tools related to requirements engineering (OpenOME[16], Objectiver[17], FAUST[18]) are not capable of generating agent-oriented code. This is probably due to the fact that the supported methodologies KAOS [35] and i* [74] as a foundation of Tropos [26] consider agents merely as an abstract modeling concepts and do not target an agent-oriented implementation. The TAOM4e tool[19] is an exception to this case, because unlike OpenOME supporting i*, it supports the Tropos methodology directly and therefore explicitly considers using agent-oriented concepts for the implementation (here by generating JADE code). The Multi Agent System Development Kit MASDK, [28], deserves a special presentation covering the design tool as well as the IDE column. MASDK realizes an approach for graphical programming, which is inspired by the Gaia methodology [73].

4.3 Example Modeling Tool: Prometheus Design Tool (PDT)

The Prometheus Design Tool (PDT) is developed at the RMIT University in Melbourne [44, 43]. It has the objective to support the agent-oriented development according to the Prometheus methodology [72]. The Prometheus methodology consists of three subsequent stages: system specification, architectural design, and detailed design. In each of these stages a developer specifies design artifacts, which is supported by the PDT. In Figure 4 a screenshot of the PDT user interface is shown. The basic working area is split into four main regions. At the upper left pane the three mentioned development stages and the associated design diagrams are listed. This area can be used to select a specific diagram, which is then shown at the right hand side and can be edited there. Each element of a design is also contained in an entity list at the lower left region. For each design element the editor for the type-dependent textual descriptor can be activated and used for adding further details (bottom right area of main window).

---

[14] http://sourceforge.net/projects/ptk

[15] http://macr.cis.ksu.edu/projects/agentTool/agentool.htm

[16] http://www.cs.toronto.edu/km/openome/

[17] http://www.objectiver.com/

[18] http://faust.cetic.be

[19] http://sra.itc.it/tools/taom4e/

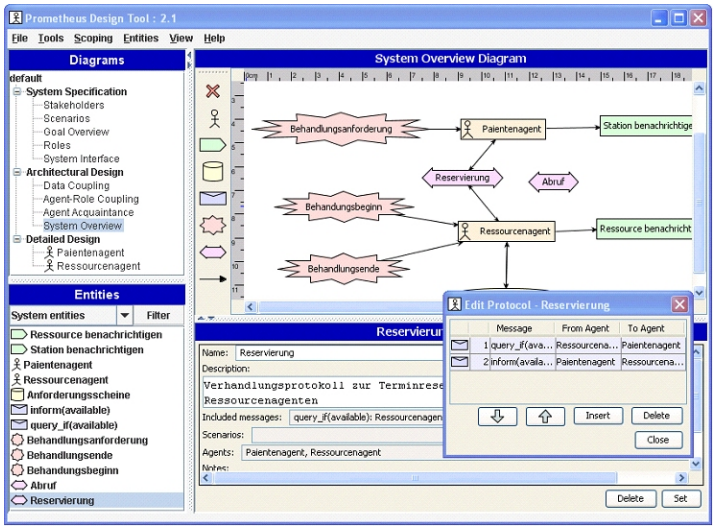

Fig. 4 Overview of PDT

In the system specification phase an analysis of the problem domain is pursued. Main goal in this phase is the specification and elaboration of the analysis overview diagram, which has the purpose to highlight the main use cases and the stakeholders participating in these use cases. This diagram can be further refined to include also the interface of the system described by percepts as inputs and actions as outputs of scenarios. In addition to this analysis overview, a system goal hierarchy can be modeled. As starting point, it is assumed that for each scenario one top-level goal exists, which can be used for a subsequent refinement into lower-level goals.

The next phase is the architectural design, where the internal composition of the system is specified. In this phase it needs to be decided which agent types make up the system and additionally in which way these agent types communicate via protocols. An agent type here is seen as a composition of one or more roles and is guided by data coupling and acquaintance considerations. Once the agent types have been identified, the agent overview diagram can be composed. This diagram is similar to an object-oriented class diagram, because it mainly highlights the agent types and their communication relationships. Also similar to class diagrams, the system overview diagram plays a central role in Prometheus and represents one of the most important artifacts produced by the methodology.

Finally, in the detailed design phase the agent internals are specified in order to equip the agents with the means to achieve their goals and handle their interactions appropriately. For each agent type, represented in an agent overview diagram, a functional decomposition is performed in order to identify its required functionalities. These functionalities are grouped according

to their coherence and consistency into so called capabilities (agent modules). For each capability a capability overview diagram is developed, which shows how a functionality can be realized in terms of belief-desire-intention concepts. For these concepts individual textual descriptors can be devised. Using the detailed design artifacts, the code generation facility of the PDT can be employed for automatically producing JACK agent language code.

PDT provides all standard functionalities of a modeling tool. It allows design diagrams being created and refined and also exported in a graphical format for documentation purposes. In addition, the consistency of the design artifacts is ensured to some degree based on constraints derived from the Prometheus metamodel. According to [44], the tool inter alia avoids references to non-existing entities, giving the same name to two elements, connecting unrelated entities and breaking interface rules. Semantical aspects can be further investigated by the tool, which generates a report indicating possible weaknesses and inconsistencies in the current design. Such a report could e.g. highlight that the model contains a message, which is actually never send by any agent in the design.

PDT also partially addresses the ancillary tasks (cross-checking, refactoring and forward/reverse engineering). The consistency of different artifacts is mainly ensured within one development phase but as specific elements such as percepts and actions are used throughout all phases, also cross-stage consistency is respected. Additionally, the automatic propagation of elements to diagrams of later phases increases the consistency further. Refactoring is not supported by the PDT so far, even though the persistent usage of elements throughout different diagrams helps to make simple operations like renaming of elements work without consistency problems. The PDT offers a code generation module for producing code skeletons directly from the models (forward engineering). If changes in the design are done only within the tool, it will preserve hand-made code changes and hence mitigate the post-editing problem [62]. A reverse engineering for producing design artifacts out of existing code is not yet available and hence no round-trip engineering is possible.

The PDT aims at supporting all relevant modeling activities of a system. A vertical integration, i.e. the integration of further tools for enabling a richer modeling experience, is currently not provided. Concerning the horizontal integration, the PDT has the already mentioned code generation mechanism, which represents a weak form of data integration. The data integration is weak, because it works in one direction only (from PDT -> Code). Furthermore, a PDT eclipse plugin is available, which allows using PDT from eclipse and realizes a control integration.

For the future several extensions are planned. One aspect is the achievement of a complete data integration between the design and code layer. Moreover, the functionalities of the PDT eclipse plugin shall be extended substantially to provide further integration facilities. In this respect, it is aimed at supporting code generation also for other target agent platforms and allow other modeling tools (e.g. UML) to be used directly from PDT, e.g. to model

non-agent related system aspects such as the underlying data model. The horizontal integration shall be further extended in direction of including the test and deployment phases as part of Prometheus and PDT.

5 Integrated Development Environments (IDEs)

IDEs are software applications, which combine different development tools under a unified user interface. The main focus of IDEs are the programming tasks that appear primarily in the implementation phase, but also in the testing and (partially) deployment phases (cf. Sections 2.1 and 2.3). Therefore most IDEs are restricted to these phases. Yet, some IDEs offer graphical modeling features, but these are usually not focused on providing a fully fledged design phase support, but instead target an abstract visual way of programming (e.g. MASDK [28]).

In the area of object-oriented software engineering there are numerous IDEs of different levels of maturity. E.g., among mature IDEs focusing on Java programming, the most widely used ones are eclipse[20], IntelliJ IDEA[21] and NetBeans[22] (cf. Methods & Tools[23] and ComputerWire[24]). IDEA is a commercial product of the company JetBrains, whereas eclipse and NetBeans are freely available Open Source solutions, which are nevertheless initiated and pushed forward by commercial companies (Sun Microsystems in case of NetBeans and a consortium of IBM, Borland, QNX, RedHat, SuSE and others for eclipse).

The following section will discuss desired features of IDEs in general, backed by the analysis from Section 2.2 and the available features of the aforementioned state-of-the art object-oriented IDEs.

5.1 Requirements for IDEs

A simple IDE at least combines an editor for working on source code with a compiler or interpreter, which translates code to a runnable program or directly executes it. Additional important functionalities of an IDE are a debugger, which allows monitoring and control a running program in order to find programming errors, as well as a repository management functionality for dealing with the files associated to a development project.

[20] http://www.eclipse.org/

[21] http://www.jetbrains.com/idea/

[22] http://www.netbeans.org/

[23] http://www.methodsandtools.com/facts/facts.php?nov03

[24] http://www.computerwire.com/industries/research/?pid=8885533F-BE8C-4760-881C-0BBBFECF534E

The central component of an IDE from the viewpoint of the developer is the editor, which is used primarily to create and edit source code but sometimes also other kinds of artifacts, such as deployment descriptors. The editor should offer integrated consistency checks, validating the code while it is typed. To improve the productivity of a developer, many IDEs additionally offer so called auto-completion, i.e. the IDE makes useful suggestions to the developer, how the partial code pieces can be expanded (e.g. variable or method names). These suggestions bear on the one hand on a syntactical understanding of the programming language and on the other hand on the current context, i.e. knowledge of the classes, variables, methods, etc. of the current project, which are accessible from the given code location. Similar knowledge is required for refactoring functionalities, which in the implementation phase also belong to the duties of an editor (e.g. the consistent renaming of methods). Besides text-based source code editors, some IDEs offer other (e.g. visual) description means for specific aspects of a system, such as graphical user interfaces, and transform these descriptions to source code automatically.

To verify progress during programming, the developer continuously has to execute and test those parts of the system, she is working on. Therefore, the IDE on the one hand has to transform source code into an executable program. For larger projects this can include, besides compiling single source files, also additional steps, such as pre- and post-processing as well as creating and assembling complex subcomponents (e.g. libraries). The IDE should enable the developer to specify/alter project specific guidelines for the build process and define all the required steps for constructing the application. Capabilities of an IDE related to the build process therefore also address tasks from the deployment phase. On the other hand, the IDE has to provide a runtime environment, in which partially completed versions of an application can be executed. Using different execution configurations, a developer can select different parts of the application for execution, based on her current situation.

A common task during the programming activity is the process of localizing bugs in a running system. For this purpose, IDEs offer so called debuggers, which allow executing a program in a step-wise manner, while observing the position in the source code as well as current variable values. A central concept of a debugger are breakpoints, i.e. positions that a developer has marked in the source code and at which executing should be interrupted. For compiled programs the debugger therefore has to be enabled to map the internal machine representation of the running program back to the source code. To support this process, program binaries usually are enriched with debugging information during the compilation process. Modern IDEs support in addition to simple breakpoints also semantic breakpoints, which are activated only, when certain conditions or events are detected (e.g. the occurrence of a specific exception type).

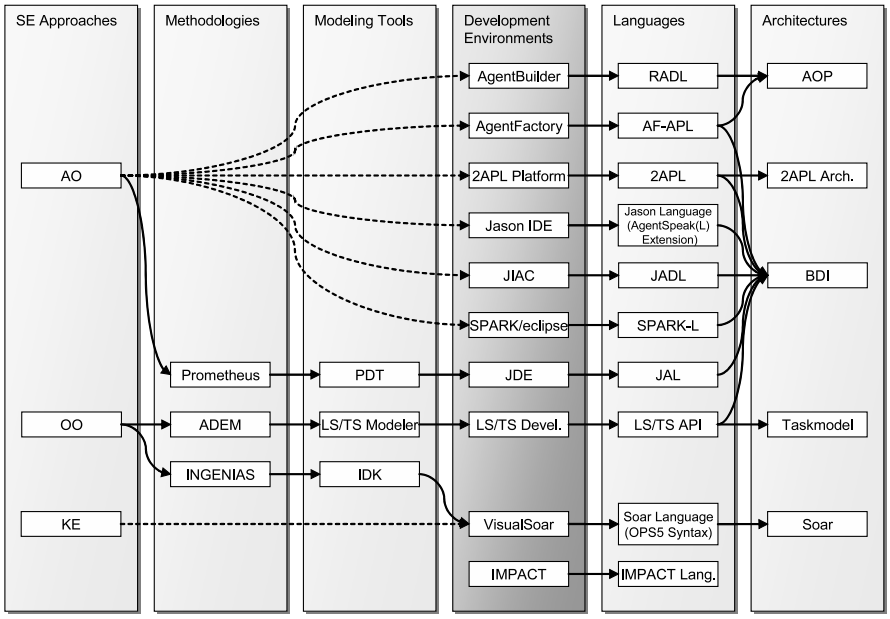

Fig. 5 Agent-oriented IDEs and relations to other artifacts

Besides phase-specific activities in the area of implementation, testing and deployment, many IDEs also support cross-cutting tasks. Especially the repository management or an integration of an existing repository management or versioning system is among the standard features of today's IDEs. The first goal of repository management is grouping all files belonging to a project into a common (e.g. directory) structure, such that the developer can easily grasp the current state of the project. Moreover, versioning features allow retrieving different (earlier) states of single files or the project as a whole, when needed. The integration with an external repository management system like CVS further facilitates a parallel and distributed development in larger project teams.

5.2 Existing Agent-oriented IDEs

For a systematic overview of existing agent-oriented IDEs , the approach from Section 4.2 is picked up. Therefore Figure 5 shows in the style of Figure 3 the interdependencies of existing agent-oriented IDEs with other development artifacts. E.g. the Visual Soar IDE[25] supports the Soar Language, for which the IDK modeling tool can generate code as also already shown in Figure 3.

[25] `http://www.eecs.umich.edu/~soar/sitemaker/projects/visualsoar/`

Extending Figure 3, the AOP and 3APL/2APL architectures have now been included in the figure (see top right), because corresponding IDEs are available. The current 2APL platform[26] [19] supporting the 2APL language provides an IDE-like tool, that offers code editing as well as debugging capabilities. AgentBuilder[27] is a commercial agent platform and toolkit supporting the Reticular Agent Definition Language (RADL), while Agent Factory[28] [crossref for chapter in this book] is an IDE supported framework for the Agent Factory Agent Programming language (AF-APL) and its variations. Both RADL and AF-APL are inspired by the seminal work of Shoham [57] on Agent-oriented Programming (AOP).

Besides the 3APL/2APL and AOP branches, additionally, the IMPACT development environment and corresponding language [21] have been added (see bottom), which are not inspired by a specific agent architecture or methodology. On the other hand, several methodologies (e.g. Tropos and MaSE), which are present in Figure 3, have been removed in Figure 5, because for these there is no continuous tool support available, which includes an IDE.

In the area of IDEs supporting BDI (belief-desire-intention) languages (cf. middle), many IDE/language pairs have been added in addition to the already mentioned JACK development environment and language (JDE, JAL, cf. Section 4.2). The Jason agent interpreter[29] [10] includes an IDE for the Jason agent language, which is a derivative of AgentSpeak(L) [50]. JIAC (Java-based Intelligent Agent Componentware)[30] [25] is a sophisticated tool suite and agent platform, which recently has been made available as open source and uses a BDI-style language called JADL (JIAC Agent Description Language). Finally, for the PRS successor SPARK (SRI Procedural Agent Realization Kit)[31] [38], developed at SRI, an IDE is available, which is realized as an eclipse plugin. Although the figure shows that an LS/TS API for BDI-style agents is available (MARGE - multi-agent reasoning based on goal execution [70]), the LS/TS Developer IDE support is mostly oriented towards the alternative task-model based MDAL API as described below.

---

[26] http://www.cs.uu.nl/2apl/

[27] http://www.agentbuilder.com/

[28] http://www.agentfactory.com/

[29] http://jason.sourceforge.net/

[30] http://www.jiac.de/

[31] http://www.ai.sri.com/~spark/

5.3 Example IDE: LS/TS Developer

The LS/TS Developer is part of the Living Systems Technology Suite (LS/TS) of Whitestein Technologies.[32] As already mentioned in the last section, instead of a new agent language, LS/TS provides several APIs that allow implementing agent applications in Java. The agent concepts and behavioral architecture are realized in the existing framework classes, while the programmer can provide new implementations of API classes that are called by the framework at relevant time points (inversion of control principle). One example of such a class is a user defined message handling component to be executed, when a matching message is received. The basic API of LS/TS is CAL (core agent layer) [68] on top of which the other APIs are built. CAL only provides a very basic autonomous agent that reacts on incoming messages or the passage of time. Besides the autonomous agent, CAL also provides so called Servants, representing passive service, and DAOs (data access objects) for managing potentially persistent data. The MDAL (message dispatching agent logic) [69] extends CAL and introduces mechanisms for selecting specific components (so called message handlers) based on properties of incoming messages. Each message handler is responsible for a sequence of messages (e.g. a negotiation with another agent) and is composed of so called fragments for each single step of the interaction. A so called context factory is responsible for instantiating new message handlers for messages that cannot be assigned to an existing message handler.

The LS/TS Development Suite includes on the one hand the already mentioned LS/TS Modeler (cf. Section 4.2) and on the other hand a set of development tools, which provide views and editors for working with CAL and MDAL elements, carrying handy names such as Developer, Debugger, Tester, and Administrator. Despite this naming, these are not separate tools, but integrated into the eclipse IDE, therefore offering an agent developer an accustomed environment allowing flexible access to the additional agent-oriented development features. The features are grouped into two perspectives that offer a useful predefined layout of the available views that can also be adapted if necessary. In the developer perspective, the agent-specific code of an application can be edited. The administrator perspective contains tools for monitoring and manipulating a running agent application.

Figure 6 shows some features of the developer perspective. In the upper half, there are existing eclipse views for Java programming (package explorer left, Java source editor middle, Java source outline right), which are useful also for editing CAL and MDAL artifacts. The lower area shows extensions like the CAL explorer and MDAL explorer (both left) or message handler diagrams (middle). In the following, the tool support available in the LS/TS Developer will be discussed with respect to the phases implementation, testing, and deployment as well as the corresponding tasks.

---

[32] `http://www.whitestein.com/`

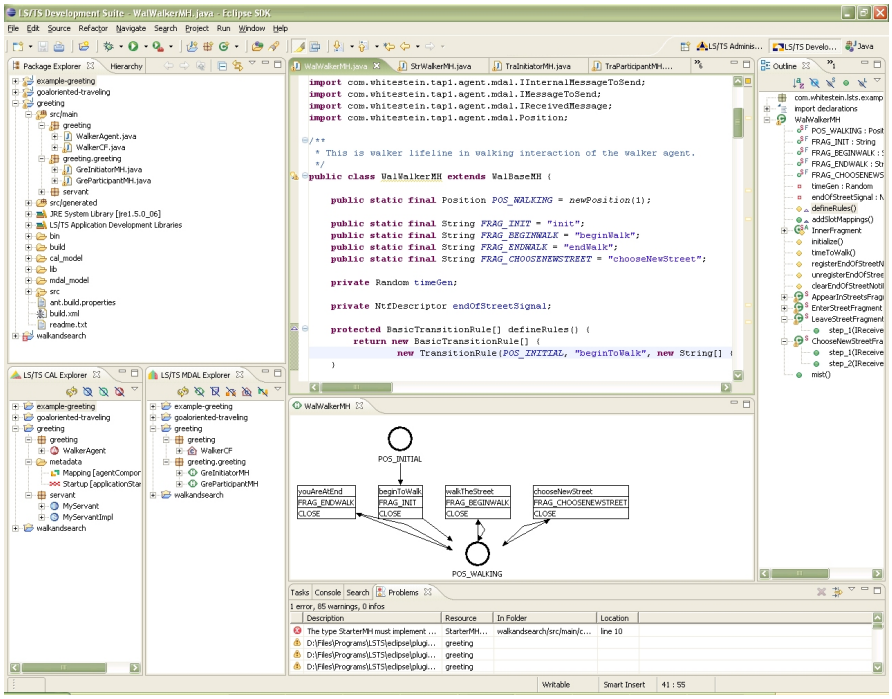

Fig. 6 Developer perspective of the LS/TS Development Suite in eclipse

Relevant implementation artifacts of LS/TS applications are Java files as well as (mostly XML-based) configuration files. Elements of the CAL and MDAL APIs are represented as Java classes that can be created and edited using existing eclipse mechanisms. Additionally, the LS/TS Developer introduces new wizards that simplify the creation of such elements based on predefined templates and can be activated from the CAL/MDAL explorer and partially also directly from the Java code editor. Syntactical consistency of Java classes can also be checked using existing eclipse mechanisms. Dependencies between MDAL elements (e.g. message handlers and contained fragments) are not considered by eclipse, because these are stored in string-based mapping tables in Java code. Besides editing Java code directly, two graphical views are provided. The first (agent diagram) allows observing and manipulating the aforementioned dependencies between context factories, message handlers, and fragments. The agent diagram is extracted from the Java sources and changes to the graphical view, such as adding/removing elements, are written back to the corresponding Java files after the changes have been reviewed and accepted/rejected by the developer in a preview window (roundtrip engineering). The second view (message handler diagram, cf. Figure 6 middle) is also extracted from the Java code and shows the execution flow of a message handler as a Petri-net. Editing of this diagram is not

possible. Refactoring for Java source files is already supported in eclipse and can also be applied to the Java-based CAL and MDAL elements, but may lead to inconsistencies, because references to elements in the aforementioned mapping tables and e.g. in XML-base deployment descriptors will not be considered by eclipse. Similarly, automatic cross-checking between different phases (e.g. implementation classes and deployment descriptors) is currently not supported and therefore has to be performed by hand. No additional support is offered for documentation tasks (e.g. it is not directly possible to export message handler and agent diagrams or include these automatically in generated Javadoc documentation).

For testing and debugging, existing Java mechanisms of eclipse can be reused. Additionally, LS/TS extends Java breakpoints in terms of agent concepts, allowing the developer to focus on specific agent instances or message types. The administrator perspective allows to record messages that are passed between agents and, for the purpose of debugging, display these messages in a sequence diagram like view as well as in a topological view. Creating test cases is supported by a test framework based on the open source jUnit[33] framework. It allows testing parts of agents (e.g. message handlers) and is supported in the IDE through wizards that enable the creation of test cases based on templates. Consistency checking and refactoring, is supported by existing eclipse mechanisms, but, as in the implementation phase, does not respect all dependencies.

Application configurations for deployment are stored in XML files. A mapping file declares available agent types and relates them to the implementing Java classes. A startup file defines the required agent instances for a concrete application configuration by specifying for each agent instance the name, type and optionally parameters. Both descriptors can be edited in XML directly as well as in a specific form based editor that abstracts away from XML syntax and provides all settings in an intuitive manner. To create an application from a specified configuration, a build process can be initiated, that is based on a predefined Ant[34] build file. Using a dialog, settings can be made that specify which configuration files are to be used and if the application should be directly deployed into an existing runtime environment. Once specified, such deployment configurations can then be executed as needed.

Cross-cutting tasks like project and repository management are already supported in part by existing eclipse features or separately available eclipse plugins for e.g. repository management with CVS. For project management, eclipse offers e.g. management of to-do entries and various search features. LS/TS additionally offers access to project files through the CAL and MDAL explorers. Moreover, the MDAL explorer offers special search functionality that simplifies the navigation in the project.

---

[33] `http://www.junit.org/`

[34] `http://ant.apache.org/`

6 Phase-specific Tools

Besides the already discussed modeling tools and IDEs, which generally span several development phases, in this section phase-specific tools will be discussed. Phase-specific tools can substantially support selected development tasks. Nonetheless, these tools are intentionally not meant to be a universal solution for building software. In order to build software the whole development process consisting of all mentioned phases need to be homogeneously tool supported. In this respect, it is of crucial importance that the tools can be integrated with each other and therefore allow a smooth transition forward and back along the different development phases, meaning that artifacts produced in one phase can be also used or refined in another phase. This integration is rather difficult to achieve and requires agreed upon conceptual models as well as standards or at least published specifications. In the area of multi-agent systems, a high heterogeneity on all layers exists rendering a desired fully-integrated tool-support across all development even harder to achieve than for the standard object-oriented paradigm. The following sections will introduce the specific requirements each phase poses towards the possible tool support, name important tool representatives and discuss one of these representatives in detail.

6.1 Requirements Analysis Phase

The artifacts of the requirements analysis phase are graphical models and/or textual specifications of an initially abstract problem. These artifacts represent concretized requirements for the system to be built. At this point in the development process, normally no decision is made about the usage of agent or an alternative technology for the realization of the system. Therefore, the produced artifacts in this stage are generally not agent-specific, which makes it possible to employ existing techniques for the communication between the customers and users on the one side and developers on the other side. Among such techniques, use cases [33] are a widespread approach that allow for capturing the main interaction possibilities of users with the system at a high abstraction level and thus facilitate the customer-developer communication. Another well-known technique is rapid prototyping, which aims at developing software demonstrators with limited functionality very early to be able to get feedback from the customers as soon as possible. As an alternative, also agent-related requirements engineering techniques can be used. Examples especially include the goal-driven approaches i* [74] and KAOS [35], which do not prescribe an agent-oriented implementation of the system, even though the transition to agent systems is conceptually more straight-forward than to traditional approaches.

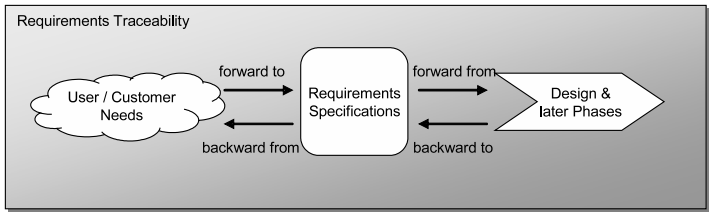

Fig. 7 Requirements traceability (from [47])

One important requisite for tools from this phase consist in the traceability of the produced application requirements, which means that the state of a requirement can be described and can be traced from all development phases. This traceability includes two different aspects (cf. Figure 7). From the viewpoint of a developer, traceability mainly refers to the later development phases, whereby the components responsible for implementing requirements should be locatable (forward tracing) and also the other way around the contribution of components to the requirements should be determinable (backward tracing). Taking up the customer's perspective, it is of importance that changed requirements are adequately communicated to the developers (forward tracing) and also that user groups can be identified, which are responsible for specific requirements (backward tracing). Requirements traceability can be realized utilizing different strategies (cf. [29]). One example are cross-references between requirements and other artifacts, which need to be explicitly specified.

As tools in this phase need not to be agent-oriented, among many different traditional and agent-related tools can be chosen. Overviews of the different tools available in this phase can be e.g. found at the web.[35] If use cases shall be utilized, it is also possible to resort to standard UML case tools.[36] As agent-related requirement tools facilitate an agent-oriented design and implementation, in the following the Objectiver tool for KAOS will be shortly presented.

The commercial Objectiver tool is developed by company Respect-IT.[37] It supports a KAOS-based goal-driven requirements specification, whereby a goal is meant to describe what the system needs to achieve. Goals are described in terms of so called patterns, which define their behavior in temporal logic. In many cases, standard patterns such as "achieve", for making true a specific world state ($\Diamond P$) or "maintain" for permanently preserving a state ($\Box P$) . The initial system goal definitions will subsequently be refined to a goal hierarchy using "why" and "how" questions until the point is reached that the subgoals on the lowest level can be clearly assigned to one of the

[35] http://easyweb.easynet.co.uk/~iany/other/vendors.htm
http://www.volere.co.uk/tools.htm

[36] http://www.objectsbydesign.com/tools/umltools_byCompany.html

[37] http://www.objectiver.com/ http://www.objectiver.com/

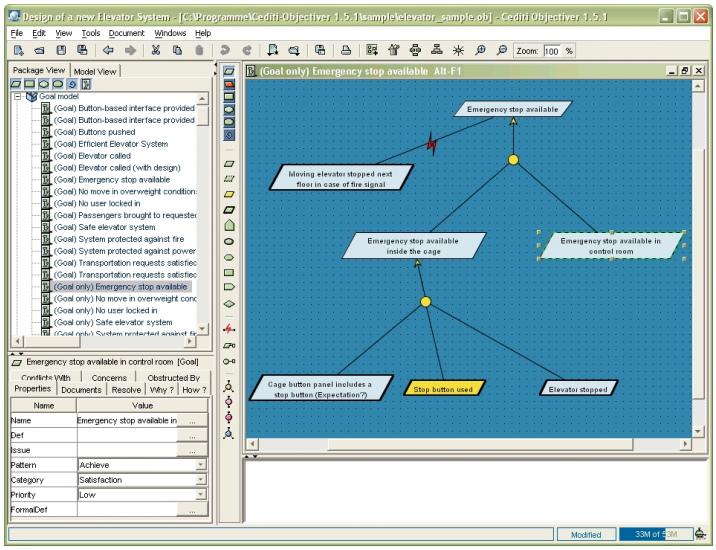

Fig. 8 Objectiver tool

actors. In a second step, besides the goal view, also system responsibilities, data objects and operations are considered and integrated to a holistic system requirements specification. In Figure 8 a screenshot of the Objectiver tool is depicted. In the main area a goal hierarchy is shown, which decomposes the top-level goal "emergency stop available" into several subgoals. Using the form at the bottom left, various entity properties can be edited. An overview of all different diagrams is given via the tree structure above. The tool automatically ensures consistency between different diagrams and can also test the specifications for plausibility and completeness.

6.2 Design Phase

In the design phase graphical and textual specifications for different aspects of the system to be realized are described. In contrast to the requirements analysis phase, agent concepts play an essential role in this stage of the development process. Considering the system as a whole, organizational concepts play an important role and can be used to describe the high-level structures of the system. In this respect, e.g. the AGR-model has been conceived for defining a system in terms of agents, groups, and roles. For the design of concrete system functionalities on the agent level, the internal agent architecture concepts are of primary interest. For example, if intentional agents shall be designed, BDI concepts such as beliefs, goals and plans could be utilized. Besides these agent related concepts, for the description of specific aspects the

agent-based views can be complemented by standard modeling concepts. One prominent area is the description of data model or conversation relationships, which can e.g. be done by using standard UML class and sequence diagrams.

The most important requirement for design tools relates to their integration ability with earlier and later development phases. It should be possible to systematically deal with already defined requirements and connect them to the newly specified design artifacts. Furthermore, the connection from the typically graphical design phase to the following code-centric implementation phase is of vital importance. This connection is difficult to achieve, because the often existing conceptual gap and additionally, the different representation media (diagrams vs. code) have to be bridged adequately.

Existing tools for the design of agent systems mainly have two different origins. First, many agent-oriented tools exist that aim at supporting a specific agent-oriented methodology such as PDT, TAOM4e or agentTool. These kinds of tools have already been discussed in the context of modeling tools (cf. Section 4) and will not be considered further here. Second, a few dedicated agent-oriented modeling tools have been developed to support the agent-oriented design approaches such as AUML and AML (agent modeling notation)[66]. The main contribution of AUML consists in the definition of a standard interaction diagram notation, which has influenced the UML 2.0 standard. Hence, traditional UML case tools can readily be used for modeling agent interactions. AML is already conceived as an agent-specific extension to UML 2.0 and extends it with agent concepts to also support the design of such kinds of software systems. An overview of the new AML diagram is given in Figure 9. In the following the LS/TS modeler, which can be used for creating AML designs, will be presented in more detail.

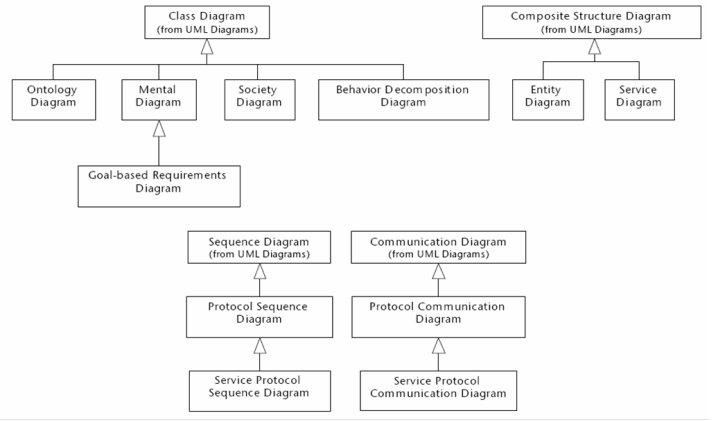

Fig. 9 AML Diagrams (from [66])

The LS/TS Modeler is part of the commercial Living Systems Technology Suite, which is distributed by Whitestein technologies. All new diagram types of AML have been introduced as refinements of existing diagram types using the standardized UML extension mechanisms. This allows normal UML case tools to be used for AML modeling when an AML UML profile for the tool exists. So far, Whitestein has developed an UML profile for the Enterprise Architect UML tool. The modeler supports all new kinds of AML diagrams, which can be categorized into architectural, behavioral and communication diagrams. Architecture diagrams are intended to describe a multi-agent system as a whole, whereas behavioral diagrams relate to the internal agent architecture and finally communication diagrams refine protocol specification facilities. As the tool is based on a standard UML tool it exhibits all necessary modeling features. In addition, the modeler add-in mainly provides code generation mechanism that can be used to create agent skeleton code for the LS/TS platform.

6.3 Implementation Phase

The focus of the implementation phase is on code and corresponding editing tasks. A major concern therefore is the agent-specific code, which depends on the chosen agent architecture and platform. E.g. for a BDI-based platform, the developer will have to write textual specifications of beliefs, goals, and plans in the provided agent language, while for a platform supporting the task model, the activities of an agent and their interrelations have to be coded (e.g. in Java). Regardless of the internal agent architecture, also communication and integration aspects have to be implemented. This includes e.g. Java classes or XML-Schema definitions for representing message content as well as mapping information for persistent data. Often, for the communication and integration issues, traditional implementation means (e.g. XML, JDBC) can be used alternatively to possibly available agent-specific solutions.

Most important requirement for agent-oriented implementation tools is the ability to deal with agent-specific code. In this respect, agent-oriented tools should strive to offer the same level of support that developers are used to in the prevailing object-oriented world. Besides the primary tasks such as creation and editing, therefore also the ancillary tasks should be adequately supported. E.g. in object orientation, code metrics are available that allow to check code consistency not only on the basis of a strict syntactical check, but also provide indications in terms of good or bad design. Moreover, given that some communication and integration issues can be realized using existing techniques, there should be tools, which provide the necessary "glue" to seamlessly operate on the different specification means.

Primary tasks are captured by IDEs for specific agent languages. Adaptable editors, such as jEdit[38] or extensible IDEs like eclipse can be used as a basis for building such IDEs to support features like syntax checks and syntax highlighting. For consistency checks, separately developed tools, such as [60], can be implemented. Moreover, some existing object-oriented metrics or style check programs allow additional metrics or inspections being added (e.g. Eclipse Checkstyle[39]), which allows adding agent-specific consistency checks. To provide the necessary glue between different technologies, often plugins or code generation templates can be used. E.g. the database mapping framework Apache Cayenne[40] supports custom code templates being used and therefore can be adapted to agent-specific requirements and for the ontology editor Protégé several plugins exist that aim at integrating ontology-based knowledge representation with an agent platform. As one example of such a tool, the Beanynizer plugin, which is part of the Jadex BDI agent framework [49], will be shortly described.

The Beanynizer is a plugin to Protégé and allows generating Java classes from an ontology modeled in Protégé. The output format is defined using templates written in the Apache Velocity[41] template language. Besides using custom templates, the Beanynizer includes two ready to use template sets – one for JADE ontology code and one for JavaBeans compliant code. For the JADE template set, the modeled ontology has to be based on a standard FIPA ontology, which includes common agent-related concepts like actions and agent identifiers. Generated classes can be used for developing JADE-based agents communicating, e.g., via the FIPA-SL content representation language. The JavaBeans templates generate platform independent pure Java code, which can e.g. be processed by the Java XML de- and encoding facilities.

6.4 Testing Phase

The aim of the testing phase is to find and correct conceptual as well as technical implementation errors. As these errors are also called "bugs" a common name for this activity in the development process is debugging. According to Dassen and Sprinkhuizen-Kuyper [18] debugging mainly consists of three subsequent steps: noticing the bug, localizing the bug and finally fixing the bug. To find possible bugs in a systematic way, often a testing approach is chosen, which requires that important aspects are captured in test cases. These test cases represent requirements that can be verified against the current implementation. The localization of bugs is still a manual skill that

[38] http://www.jedit.org/

[39] http://eclipse-cs.sourceforge.net/

[40] http://cayenne.apache.org/

[41] http://velocity.apache.org/

requires considerable effort, experience and creativity. It mainly requires the programmer to inspect the source code in detail and possibly use a debugger tool execute the program stepwise resp. stop it at specific breakpoints. As errors may manifest themselves in unpredictable behavior their identification can be a very hard and complex task. Fixing the bug is not directly part of the testing phase but requires a developer to step back to the implementation phase or in case of conceptual problems even to the design phase and correct the identified artifacts. As can be seen from this description, additional artifacts in this phase are only constructed for specifying test cases. The other activities fully operate on existing artifacts, especially on the code level.

Main requirements for tools of this phase consist in a conceptual and technical support for the detection and localization of bugs. For the systematic detection of bugs tools should facilitate the implementation and automated execution of test cases. This should include test cases for different layers such as unit tests for single functionalities, integration tests for larger components and system tests for the validation of system requirements. In addition, it is helpful if the test coverage, i.e. which system aspects are tested to what degree, can be automatically calculated and presented to the developer. An indication of possible bugs can also be produced by software metrics that try to capture the quality of source code.

Tools supporting the testing phase have mainly been developed in the context of object-oriented languages and are often directly integrated into IDEs. In the area of multi-agent systems, only recently the testing topic has gained some attention. Conceptually, multi-agent systems increase the complexity of all activities in this phase, so that a direct transfer of existing solutions is not easily possible. Testing and debugging on the level of the whole multi-agent system entails all the difficulties involved in testing and debugging distributed system (e.g. concurrency and lack of global state). Tools in this area focus on the interactions, i.e. the messages passed between agents and allow monitoring messages (e.g. JADE Sniffer) or testing compliance to specified protocols (e.g. [1]). To address the issues of debugging under consideration of the whole development process, it is also researched how design artifacts can support this phase [crossref chapter debugging from Winikoff]. Support for unit testing at the level of single agents has been devised in the context of tool suites for agent platforms such as JADE , Jadex and LS/TS . Furthermore, nearly all existing agent platforms offer (at least simple) debugging tools, which allow the stepwise execution of agents. In the following Jadex TestCenter tool will be shortly described.

A screenshot of the Jadex TestCenter is shown in Figure 10. Its underlying concepts are based on JUnit, i.e. it is possible to define a set of test cases as test suite (in form of a list at the top right area) and then execute this suite automatically (control area below the list). Here, test agents containing an arbitrary number of test cases can be directly added from the file system view (left area) to the list. The results of the test suite execution are summarized as a colored bar, which is green in case all test cases have been successful and

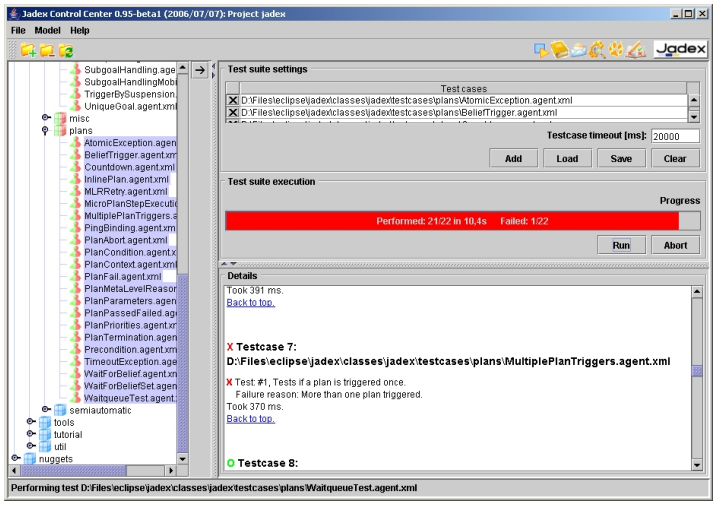

Fig. 10 Jadex TestCenter

red in case at least one test failed. The detailed test results are displayed in a form of a textual report (right bottom), which explains what the individual test cases do, which ones have failed and a possible reason for that failure.

6.5 Deployment Phase

For object-oriented systems the tasks of the deployment phase are clearly specified and also well tool-supported. In this context, the Object Management Group (OMG) has defined deployment as the activity between obtaining and operating a software product. More concretely, in [40] the OMG has specified a general deployment process consisting of five subsequent steps. In the first, so called *installation step*, the software is obtained and stored in a local repository, which must not necessarily be the same location as the destined execution location. In the following *configuration step*, the software is parametrized according to the intended use cases. Hereafter, a deployment plan is devised in the *planning step*. This plan is then used in the *preparation step* to install the desired components on the target platforms. In the final *launching step* the application is started and hence put into operation, which might require further configuring activities at runtime.

Regarding agent-based systems this process is usually more flexible, because the constituents are not passive components, but active autonomous entities [12]. Nonetheless, the aforementioned steps remain important for multi-agent systems as well. In the installation step the execution infrastructure for the agents, i.e. the agent platform and also the application specific

components, e.g. consisting of agent code as well as standard libraries, have to be available. This may not necessarily mean that the application code has to be obtained completely in beforehand. Possibly agent code could also be downloaded on demand at runtime. The functional configuration of the application can be done by defining the number and kinds of agents that should be initially started and by setting their initial parameters to appropriate values. The planning and preparation steps mainly need to take into account at which hosts which infrastructure should be accessible and which agents should be located. In case of mobile agents, the distribution of agents at the different nodes could also be adjusted at runtime, e.g. with respect to non-functional aspects like load balancing. Starting an agent application is quite different from launching a component-based software, because there is no single centralized starting point. Instead a set of (possibly independent or interrelated) actors need to be created in a meaningful way. Hence, in order to specify agent applications it should be abstracted away from single agents and some form of application descriptors should be made available.

Artifacts of the deployment phase are therefore mainly these application and agent descriptors. Tools of this phase have the tasks of supporting the creation and processing of such descriptors, whereby the creation can be associated with the configuration and the processing with the launching step. In addition, deployment tools can also be extended in direction of runtime monitoring facilities.

Tool-based deployment support for agent applications is rather limited today. It has mainly been considered technically in the context of agent platforms and several similar ad-hoc solutions have been provided so far. E.g. in Agent Academy [37], AgentFactory [15], Jason [10], simple application descriptors have been introduced, which at least enable a definition of the parametrized agent instances to start. With the LS/TS Developer also a tool exists, which simplifies the specification of agent applications in a similar way to J2EE deployment descriptors. It can be used to deploy the tool generated application in the agent platform automatically. Similarly, approaches like BlueJADE [17] and jademx[42] try to make agent platforms administrable similar to J2EE server environments, but do not consider the assembly of agent applications. In the following, the ASCML tool, conceived specifically for the deployment of agent applications, will be described.

The ASCML (Agent Society Configuration Manager and Launcher) is based on a generic deployment reference model for agent applications (cf. Figure 11) [12]. This reference model assumes that agent applications (here called societies) are controlled by dedicated manager (ASCML) agents. These agents have the responsibilities to start, supervise, and possible reconfigure the controlled societies. The concept of agent society here is recursively defined, meaning that it can be composed of a hierarchy of agent instances or sub societies possibly distributed across different network nodes. The AS-

[42] `http://jademx.sourceforge.net`

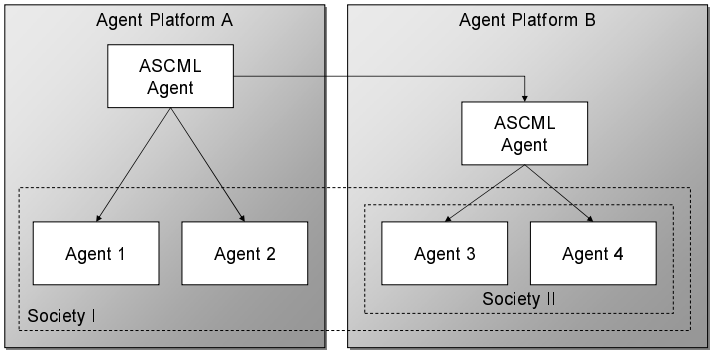

Fig. 11 ASCML refernce model (from [12])

CML tool allows defining agent applications in the form of society and agent descriptors, which are interpreted by the tool at runtime and lead to the instantiation of the specified software runtime configuration. It extends the basic facilities by constraint expressions, which can be used to state, in which cases the application needs to reconfigured, e.g. by restarting specific agents given that a necessary service is not available any longer.

7 Evaluation

The preceding sections have shown that numerous agent-oriented tools have been developed. Besides the phase-specific tools, which only address tasks of one development phase, mainly modeling tools and IDEs have been identified as important tool categories. In this section a coarse evaluation of these modeling tools and IDEs will be presented.[43] The main objective of this evaluation is an assessment of the state of the art of agent-oriented tools in order to highlight the strengths and weaknesses of the current tool landscape. The evaluation is based on the generic task requirements within the different phases of a development process (cf. Section 2.2). Each of the 10 modeling tools and 11 IDEs have been analyzed with respect to the identified tasks of the corresponding phases, i.e. modeling tools have been evaluated against task requirements from the analysis and design phase whereas IDEs have been tested against the task requirements from the implementation, testing and deployment phases. Cross-cutting activities like repository management and development coordination are not agent-specific and have not been evaluated. With regard to those cross-cutting tasks established tool support can be reused, e.g. the CVS (Concurrent Versions System) can be employed for

[43] The phase-specific tools have been excluded from the evaluation due to the low number of representatives in each phase.

version management. In case that agent-oriented tools build on established object oriented IDEs like eclipse, orthogonal support for those features is directly available via plugins for the IDEs. The aggregated results of the evaluation, which intentionally abstract away from the concrete tool representatives, are depicted in Figure 12. It is shown how many tools of each category (modeling tools vs. IDEs) support a given task.

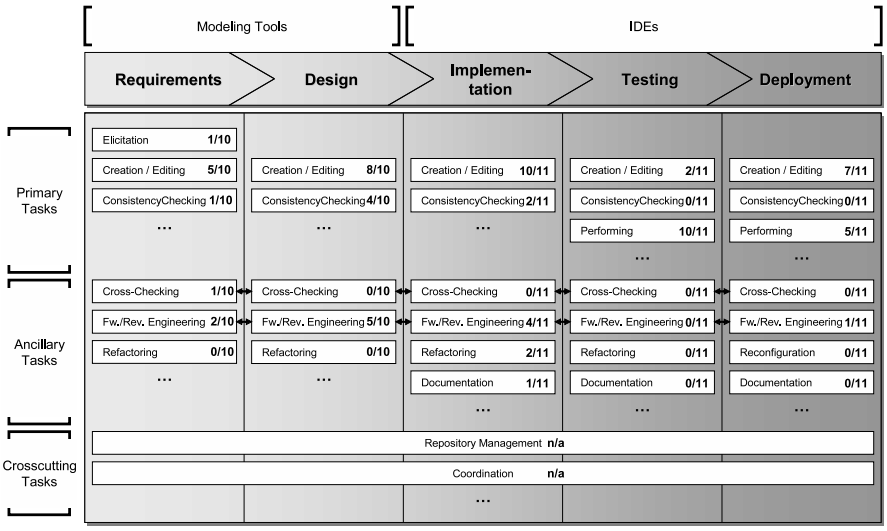

Fig. 12 Tool evaluation

Looking at the modeling tools, it can be seen that, in general, stronger support exists for tasks of the design phase, whereas only a few tools tackle tasks from the preceding requirements phase. The basic features for creating and editing requirements artifacts is supported by 5 of 10 tools. Further primary tasks are rarely supported in the requirements phase, i.e. only one representative handles the initial elicitation of requirements and consistency checking of requirements artifacts. Regarding the ancillary tasks, also only marginal tool support could be revealed. Two representatives allow the generation of design artifacts and one representative tackles cross-checking with generated artifacts. None of the tools addressed refactoring aspects. A similar support structure can be identified also at the design phase. Creating and editing of design artifacts is available by nearly all tools (8/10) and also the consistency checking of artifacts is supported by nearly half of the tools (4/10). When looking at the ancillary tasks, it can be seen that 5/10 tools include forward engineering features. Though, in most cases, only simple code generation facilities are available, which can produce initial code skeletons from design artifacts. Advanced features like reverse or round-trip engineering have not

been introduced in any tool. Cross-checking and refactoring have not been addressed at all.

An agent-oriented IDE would ideally support all tasks from the implementation, testing and deployment phases. Among the IDEs, nearly all representatives (10 of 11) offer functionality for creating/editing implementation artifacts (i.e. agent code) as well as debugging running agent applications (i.e. the performing task in the testing phase). This reveals that tool developers consider programming and debugging agents as the most important tasks of agent developers. On the other hand, the systematic creation of repeatable test cases is only supported by 2 representatives. A considerable amount of support is also available in the deployment phase, for the creation of deployment descriptors (7/11) as well as actually deploying agents to an existing infrastructure (5/11). The fact that deployment features are considered important by tool developers reinforces the significance of agents as a technology for distributed computing. Among the primary tasks, consistency checking is the least supported. Only 2 representatives offer some consistency analysis features for agent programs. Ancillary tasks are also seldom supported. In the implementation phase, only four tools offer code generation features, two support refactoring and one tool allows the generation of documentation. In the testing and deployment phases, ancillary features are mostly not addressed at all. Only one tool offers the forward generation of deployment descriptors from agent models.

Summing up this coarse evaluation of the state of tool support for agent-oriented development it can be noted that at least for the important tasks considerable support is available by most current development tools. This means the the most common development tasks are adequately supported by tools, regardless which specific agent language or methodology is chosen by the developer. On the other hand, no single tool is able to support all tasks. Especially in the area of the (probably less important) ancillary tasks, agent-oriented tools have considerable potential for improvement. As an example: even the most powerful agent-oriented IDEs only support at most 7 of possible 20 tasks in the implementation, testing and design phases. For comparison, a short analysis of state-of-the-art OO IDEs, such as eclipse or IntelliJ IDEA, indicates that these support up to 12-15 tasks out of the box and even more when using additional plugins. One notable feature in this area is refactoring, which becomes more and more important, the larger the developed applications grow. Therefore, improving agent-oriented tool support in this direction could be crucial for adequately supporting larger software projects.

8 Conclusion

This chapter has the purpose the give a systematic overview about the existing agent-oriented development tool landscape. Therefore, first the tasks of sofware development tools have been collected and categorized along the two dimensions: development phases and task importance. It has been identified as crucial that tools of all phases should enable the creation and editing as well as consistency checking of development artifacts. In addition, it is helpful when tools also cope with ancillary tasks like cross-checking, forward/reverse engineering and refactoring and also crosscutting tasks like repository management. Based on the existing surveys three major categories of tools have been identified: modeling tools, IDEs and phase-specific tools. For each of the categories the specific requirements have been described, an overview of existing tools has been given and finally one specific representative has been selected and described in greater detail.

The evaluation of the current agent tool landscape allows some general observations to be made. First, all development phases and all tasks are to some extent tool-supported, meaning that a variety of different tools are available for all possible use cases. Nonetheless, most of these tools suffer from the strong heterogeneity of the multi-agent systems field. This heterogeneity often leads to very specific approaches suitable only for one specific agent approach, e.g. a design tool for BDI agents only. Furthermore, the number of available tools is quite low compared to mainstream object-oriented solutions. In order to further improve the overall tool support it is necessary that at least the following future trends emerge. First, a general agreement on core concepts and a consolidation of the agent platforms would help concentrating on successful development branches. Second, tool support is generally dependent on the industry uptake of agent technology as a whole, because research institutions normally do not have enough resources for providing industry-grade tools, which typically demand high investments in terms of man-month of work (cf. the efforts for building an object-oriented IDE like eclipse).

The survey reveals that tool-support of agent technology is still a bit behind the currently available support in the predominant object-oriented paradigm. Nevertheless, the recent years have shown significant improvements in this respect. The analysis of the AgentLink directory highlights that some convergence has already happened in the area of agent platforms. The increasing maturity of agent platforms allows development efforts to also focus on secondary items like tool support. Moreover, agent-oriented tool support also profits from the fact that object-oriented tools have become more and more flexible and therefore many agent tools are not built from scratch but instead adapt existing object-oriented tools to agent-specific requirements. One reason for object-oriented tools becoming more flexible can be seen in the desire of supporting other post object-oriented technologies like model-driven development and web services. These technologies shift the focus to

abstract modeling or the interaction of system components. Therefore, these technologies are conceptually much closer to the agent paradigm, which may foster an integration and convergence with agent concepts and tools in the future.

References

1. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Compliance Verification of Agent Interaction: a Logic-Based Tool. In: R. Trappl (ed.) Proceeding of the 7th European Meeting on Cybernetics and Systems Research, Track AT2AI-4: From Agent Theory to Agent Implementation (AT2AI 2004), pp. 570–575. Austrian Society for Cybernetic Studies (2004)
2. Bartsch, K., Robey, M., Ivins, J., Lam, C.: Consistency checking between use case scenarios and uml sequence diagrams. In: M. Hamza (ed.) Proceedings of the IASTED International Conference on Software Engineering (SE 2004), pp. 92–103. ACTA Press (2004)
3. Bellifemine, F., Bergenti, F., Caire, G., Poggi, A.: JADE - A Java Agent Development Framework. In: R. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (eds.) Multi-Agent Programming: Languages, Platforms and Applications, pp. 125–147. Springer (2005)
4. Bellifemine, F., Caire, G., Greenwood, D.: Developing Multi-Agent systems with JADE. John Wiley & Sons (2007)
5. Bitting, E., Carter, J., Ghorbani, A.: Multiagent system development kits: An evaluation. In: In Proceedings of the 1st Annual Conference on Communication Networks and Services Research (CNSR 2003), pp. 80–92. CNSR Project (2003)
6. Boehm, B.W.: A spiral model of software development and enhancement. IEEE Engineering Management Review 23 **4**, 69–81 (1995)
7. Boger, M., Sturm, T., Fragemann, P.: Refactoring browser for uml. In: M. Aksit, M. Mezini, R. Unland (eds.) Proceedings of the 4th International Conference on Objects, Components, Architectures, Services, and Applications for a Networked World (Net.ObjectDays 2002), pp. 366–377. Springer (2003)
8. Bordini, R., Braubach, L., Dastani, M., El Fallah Seghrouchni, A., Gomez-Sanz, J., Leite, J., OŽHare, G., Pokahr, A., Ricci, A.: A survey of programming languages and platforms for multi-agent systems. Informatica **30**, 33–44 (2006)
9. Bordini, R., Dastani, M., Dix, J., El Fallah Seghrouchni, A.: Multi-Agent Programming: Languages, Platforms and Applications. Springer (2005)
10. Bordini, R., Hübner, J.F., Vieira, R.: Jason and the Golden Fleece of Agent-Oriented Programming. In: R. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (eds.) Multi-Agent Programming: Languages, Platforms and Applications, pp. 3–37. Springer (2005)
11. Braubach, L.: Architekturen und Methoden zur Entwicklung verteilter agentenorientierter Softwaresysteme. Ph.D. thesis, Universität Hamburg (2007)
12. Braubach, L., Pokahr, A., Bade, D., Krempels, K.H., Lamersdorf, W.: Deployment of Distributed Multi-Agent Systems. In: M.P. Gleizes, A. Omicini, F. Zambonelli (eds.) Proceedings of the 5th International Workshop on Engineering Societies in the Agents World (ESAW 2004), pp. 261–276. Springer (2005)
13. Braubach, L., Pokahr, A., Lamersdorf, W.: Tools and Standards. In: S. Kirn, O. Herzog, P. Lockemann, O. Spaniol (eds.) Multiagent Systems. Intelligent Applications and Flexible Solutions, pp. 503–530. Springer (2006)
14. Braubach, L., Pokahr, A., Lamersdorf, W.: A universal criteria catalog for evaluation of heterogeneous agent development artifacts. International Journal of Agent-Oriented Software Engineering (IJAOSE) (2009). To appear

15. Collier, R.W.: Agent Factory: A Framework for the Engineering of Agent-Oriented Applications. Ph.D. thesis, University College Dublin (2001)
16. Cossentino, M.: From Requirements to Code with the PASSI Methodology. In: B. Henderson-Sellers, P. Giorgini (eds.) Agent-Oriented Methodologies, pp. 79–106. Idea group publishing (2005)
17. Cowan, D., Griss, M., Burg, B.: Bluejade - A service for managing software agents. Tech. Rep. HPL-2001-296R1, Hewlett Packard Laboratories (2002)
18. Dassen, J., Sprinkhuizen-Kuyper, I.: Debugging c and c++ code in a unix environment. The Object Oriented Programming Web (OOPWeb.com) (1999)
19. Dastani, M.: 2apl: a practical agent programming language. International Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS), Special Issue on Computational Logic-based Agents **16**(3), 214–248
20. DeLoach, S., Wood, M., Sparkman, C.: Multiagent systems engineering. International Journal of Software Engineering and Knowledge Engineering **11**(3), 231–258 (2001)
21. Dix, J., Zhang, Y.: IMPACT: Multi-Agent Framework with Declarative Semantics. In: R. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (eds.) Multi-Agent Programming: Languages, Platforms and Applications, pp. 69–94. Springer (2005)
22. Dröschel, W., Wiemers, M.: Das V-Modell 97 - Der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz. Oldenbourg (1999)
23. Eiter, T., Mascardi, V.: Comparing environments for developing software agents. The European Journal on Artificial Intelligence (AI Communications) pp. 169–197 (2002)
24. Fonseca, S.P., Griss, M.L., Letsinger, R.: Agent behavior architectures - A MAS framework comparison. Tech. Rep. HPL-2001-332, Hewlett Packard Laboratories (2002)
25. Fricke, S., Bsufka, K., Keiser, J., Schmidt, T., Sesseler, R., Albayrak, S.: Agent-based telematic services and telecom applications. Commun. ACM **44**(4), 43–48 (2001). DOI http://doi.acm.org/10.1145/367211.367251
26. Giorgini, P., Kolp, M., Mylopoulos, J., Pistore, M.: The Tropos Methodology. In: F. Bergenti, M.P. Gleizes, F. Zambonelli (eds.) Methodologies and Software Engineering For Agent Systems, pp. 89–106. Kluwer Academic Publishers (2004)
27. Gomez-Sanz, J., Pavon, J.: Agent oriented software engineering with ingenias. In: 3rd International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS 2003), pp. 394–403. Springer Verlag (2003)
28. Gorodetsky, V., Karsaev, O., Samoylov, V., Konushy, V., Mankov, E., Malyshev, A.: Multi Agent System Development Kit. In: R. Unland, M. Calisti, M. Klusch (eds.) Software Agent-Based Applications, Platforms and Development Kits, pp. 143–168. Birkhäuser (2005)
29. Gotel, O., Finkelstein, C.: An analysis of the requirements traceability problem. In: Proceedings of the 1st International Conference on Requirements Engineering (ICRE 1994), pp. 94–101. IEEE (1994)
30. Grundy, J., Hosking, J.: Software tools. In: J. Marcin (ed.) The Software Engineering Encyclopedia. Wiley (2001)
31. Henderson-Sellers, B., Giorgini, P. (eds.): Agent-Oriented Methodologies. Idea group publishing (2005)
32. International Organization for Standadization (ISO): Ergonomics of Human-System Interaction-Part 110: Dialogue Principles, ISO 9241-110:2006 edn. (2006)
33. Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley (1992)
34. Laird, J., Rosenbloom, P.: The evolution of the Soar cognitive architecture. In: D. Steier, T. Mitchell (eds.) Mind Matters: A Tribute to Allen Newell, pp. 1–50. Lawrence Erlbaum Associates (1996)
35. van Lamsweerde, A.: Goal-Oriented Requirements Engineering: A Guided Tour. In: Proceedings of the 9th International Joint Conference on Requirements Engineering (RE 2001), pp. 249–263. IEEE Press (2001)

36. Mangina, E.: Review of Software Products for Multi-Agent Systems. Tech. rep., AgentLink (2002). URL http://www.agentlink.org/resources/software-report.html
37. Mitkas, P.A., Kehagias, D., Symeonidis, A.L., Athanasiadis, I.N.: A framework for constructing multi-agent applications and training intelligent agents. In: P. Giorgini, J. Müller, J. Odell (eds.) Proceedings of the 4th International Workshop on Agent-Oriented Software Engineering IV (AOSE 2003), pp. 96–109. Springer (2003)
38. Morley, D., Myers, K.: The spark agent framework. In: Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), pp. 714–721. IEEE Computer Society (2004)
39. Nwana, H., Ndumu, D., Lee, L., Collis, J.: Zeus: a toolkit and approach for building distributed multi-agent systems. In: Proceedings of the 3rd annual conference on Autonomous Agents (AGENTS 1999), pp. 360–361. ACM Press (1999)
40. Object Management Group (OMG): Deployment and Configuration of Component-based Distributed Applications Specification, version 4.0 edn. (2003). URL http://www.omg.org/cgi-bin/doc?formal/06-04-02
41. Object Management Group (OMG): Unified Modeling Language: Superstructure, version 2.0 edn. (2005). URL http://www.omg.org/cgi-bin/doc?formal/05-07-04
42. Odell, J., Parunak, H.V.D., Bauer, B.: Extending UML for Agents. In: G. Wagner, Y. Lesperance, E. Yu (eds.) Proceedings of the 2nd International Bi-Conference Workshop Agent-Oriented Information Systems Workshop (AOIS@AAAI 2000), pp. 3–17 (2000)
43. Padgham, L., Thangarajah, J., Winikoff, M.: Tool support for agent development using the prometheus methodology. In: Proceedings of the 5th International Conference on Quality Software (QSIC 2005), pp. 383–388. IEEE Computer Society (2005)
44. Padgham, L., Thangarajah, J., Winikoff, M.: The prometheus design tool ? a conference management system case study. In: M. Luck, L. Padgham (eds.) Agent Oriented Software Engineering VIII, *LNCS*, vol. 4951, pp. 197–211. Springer (2008). 8th International Workshop, AOSE 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected Papers
45. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. John Wiley & Sons (2004)
46. Pavón, J., Gómez-Sanz, J.: Agent oriented software engineering with ingenias. In: V. Marík, J. Müller, M. Pechoucek (eds.) Multi-Agent Systems and Applications III, 3rd International Central and Eastern European Conference on Multi-Agent Systems, (CEEMAS 2003), pp. 394–403. Springer (2003)
47. Pokahr, A.: Programmiersprachen und Werkzeuge zur Entwicklung verteilter agentenorientierter Softwaresysteme. Ph.D. thesis, Universität Hamburg (2007)
48. Pokahr, A., Braubach, L., Lamersdorf, W.: Agenten: Technologie für den mainstream? In: it - Information Technology, pp. 300–307. Oldenbourg Verlag (2005)
49. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI Reasoning Engine. In: R. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (eds.) Multi-Agent Programming: Languages, Platforms and Applications, pp. 149–174. Springer (2005)
50. Rao, A.: AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In: W.V. de Velde, J. Perram (eds.) Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW 1996), pp. 42–55. Springer (1996)
51. Rausch, A.: Componentware - Methodik des evolutionären Architekturentwurfs. Herbert Utz Verlag (2004)
52. Rausch, A., Broy, M., Bergner, K.: Das V-Modell XT. Grundlagen, Methodik und Anwendungen. Springer (2006)
53. Reticular Systems: AgentBuilder User's Guide, version 1.3 edn. (2000). http://www.agentbuilder.com/

54. Rimassa, G., Greenwood, D., Kernland, M.E.: The Living Systems Technology Suite: An Autonomous Middleware for Autonomic Computing. In: In Proceedings of the International Conference on Autonomic and Autonomous Systems (ICAS 2006) (2006)
55. Robbins, J., Hilbert, D., Redmiles, D.: Software architecture critics in argo. In: Proceedings of the 3rd international conference on Intelligent user interfaces (IUI 1998), pp. 141–144. ACM Press (1998)
56. Serenko, A., Detlor, B.: Agent Toolkits: A General Overview of the Market and an Assessment of Instructor Satisfaction with Utilizing Toolkits in the Classroom. Tech. Rep. Working Paper #455, Michael G. DeGroote School of Business, McMaster University (2002)
57. Shoham, Y.: Agent-oriented programming. Artificial Intelligence **60**(1), 51–92 (1993)
58. Sturm, A., Shehory, O.: A Comparative Evaluation of Agent-Oriented Methodologies. In: F. Bergenti, M.P. Gleizes, F. Zambonelli (eds.) Methodologies and Software Engineering For Agent Systems, pp. 127–149. Kluwer Academic Publishers (2004)
59. Sturm, A., Shehory, O.: A framework for evaluating agent-oriented methodologies. In: P. Giorgini, B. Henderson-Sellers, M. Winikoff (eds.) Agent-Oriented Information Systems (AOIS 2003), pp. 94–109. Springer (2004)
60. Sudeikat, J., Braubach, L., Pokahr, A., Lamersdorf, W., Renz, W.: Validation of bdi agents. In: Proceedings of the 4th International Workshop on Programming Multiagent Systems: languages, frameworks, techniques and tools (ProMAS 2006). Springer (2006). (to appear)
61. Sunyé, G., Pollet, D., Traon, Y.L., Jézéquel, J.M.: Refactoring uml models. In: M. Gogolla, C. Kobryn (eds.) The Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML 2001), pp. 134–148. Springer (2001)
62. Szekely, P.: Retrospective and challenges for model-based interface development. In: F. Bodart, J. Vanderdonckt (eds.) Design, Specification and Verification of Interactive Systems (DSV-IS 1996), pp. 1–27. Springer (1996)
63. Thangarajah, J., Padgham, L., M.Winikoff: Prometheus design tool. In: F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. Singh, M. Wooldridge (eds.) 4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), pp. 127–128. ACM (2005)
64. Tryllian Solutions B.V: The Developer's Guide, release 3.0 edn. (2005). URL `http://www.tryllian.com`
65. Unland, R., Calisti, M., Klusch, M.: Software Agent-Based Applications, Platforms and Development Kits. Birkhäuser (2005)
66. Whitestein Technologies: Agent Modeling Language, Language Specification, Version 0.9 edn. (2004)
67. Whitestein Technologies: Agent-Oriented Development Methodology for LS/TS, A Comprehensive Overview, LS/TS Release 2.0.0 edn. (2006)
68. Whitestein Technologies: Core Agent Layer Concept, LS/TS Release 2.0.0 edn. (2006)
69. Whitestein Technologies: Message Dispatching Agent Logic Concept, LS/TS Release 2.0.0 edn. (2006)
70. Whitestein Technologies: Multi-Agent Reasoning based on Goal-oriented Execution, LS/TS Release 2.0.0 edn. (2006)
71. Winikoff, M.: JACK Intelligent Agents: An Industrial Strength Platform. In: R. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (eds.) Multi-Agent Programming: Languages, Platforms and Applications, pp. 175–193. Springer (2005)
72. Winikoff, M., Padgham, L.: The Prometheus Methodology. In: F. Bergenti, M.P. Gleizes, F. Zambonelli (eds.) Methodologies and Software Engineering For Agent Systems, pp. 217–234. Kluwer Academic Publishers (2004)
73. Wooldridge, M., Jennings, N., Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. Autonomous Agents and Multi-Agent Systems **3**(3), 285–312 (2000)

74. Yu, E.: Towards modelling and reasoning support for early-phase requirements engineering. In: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE 1997), pp. 226–235. IEEE Press (1997)

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the definition; numbers in roman refer to the pages where the entry is used.

2APL, 16, 23

ADEM, 16
ADK, 10
Agent Academy, 35
Agent oriented IDEs, **22**
agent platforms, 11
AgentFactory, 23, 35
AgentLink, 10
agentTool, 30
AML, 31
AOP, 16, 23
ASCML, 35
AUML, 14, 30

BDI, 23

criteria catalog, 2

debugging, **32**
deployment phase, **34**
design phase, **29**
development process, 2
development tasks, 3, 7

eclipse, 20, 32, 38

GAIA, 14

i*, 17, 27
IDEs, 2, **20**, 32, 36
IMPACT, 10, 23
implementation phase, **31**
INGENIAS, 16
integration, 8
  control, 8
  data, 8
  process, 8
IntelliJ IDEA, 20, 38

JACK, 10, 16, 19, 23
JADE, 10, 17, 33
Jadex, 33
Jason, 23, 35
jEdit, 32
JIAC, 23

KAOS, 17, 27

LS/TS, 33
LS/TS Developer, 23, 24, 35
LS/TS Modeler, 16, 24, 31

MASE, 17
methodology, 2, 14
modeling tools, 36

NetBeans, 20

PASSI, 17
PDT, 16, **17**, 30
phase model, 2
Prometheus, 16
Protégé, 32

requirements analysis phase, **27**

SOAR, 17
SPARK, 23

TAOM4e, 17, 30
tasks, 3
  ancillary tasks, 5
  consistency checking, 6
  coordination, 7
  creation and editing, 6
  cross-checking, 6
  crosscutting tasks, 5
  forward engineering, 6
  primary tasks, 5
  refactoring, 6
  repository management, 7
  reverse engineering, 6
testing phase, **32**
tool evaluation, 36
tool survey, 2
tools, 1
  agent oriented, 9
  debugging tools, 7
  design tools, 7, 11
  IDEs, 7, 11
  modelling tools, 13
  software tools, 7
  testing tools, 7
  development tools, 2
traceability, 28

UMBC Agent Web, 11
UML, 30, 31

V-model, 4

ZEUS, 10