2009

FROM A RESEARCH TO AN INDUSTRY-STRENGTH AGENT PLATFORM: JADEX V2

Alexander Pokahr
University of Hamburg

Lars Braubach
University of Hamburg

FROM A RESEARCH TO AN INDUSTRY-STRENGTH AGENT PLATFORM: JADEX V2

Alexander Pokahr, Lars Braubach[1]

Abstract
Since the beginning of the nineties multi-agent systems have been seen as a promising new software paradigm that is capable to overcome conceptual weaknesses of mainstream object-oriented software solutions. Despite these theoretical advantages, in practice agent software is rarely used and as software paradigm has been widely superseded by the service-oriented architecture. One key reason for the slow adoption of agent-based ideas is that existing agent software in most cases does not provide business-relevant features such as persistency or scalability. Hence, in this paper it is analyzed which essential business requirements exist and a solution agent platform architecture is presented. This architecture has been implemented within the Jadex V2 agent platform, which is a complete overhaul of the V1 architecture.

1. Introduction

In literature agent orientation is often promoted as new software paradigm with which shortcomings of object-oriented solutions can be solved. Nonetheless, in practice only few agent-based systems have been successfully deployed and the paradigm is not well recognized by industry practitioners. One question that will be tackled in this paper is, why this is the case and how this can be changed.

In general, agent orientation brings a new perspective into software engineering and considers an application of being composed by a multitude of individual agents working together to provide the required functionalities. The agent paradigm assumes that an agent is an autonomous entity that acts on its own behalf and communicates with other agents via asynchronous messages. Hence, agent orientation is a paradigm that naturally fits for the development of loosely-coupled *distributed* and *concurrent* systems. On the one hand, distribution is supported by the fact that agents make no assumptions where other agents are located meaning that the location of an agent is transparent for its communication partners. This allows a software developer to distribute agents across system borders as needed without having to change the application logic. On the other hand, concurrency is supported by the agent's autonomy, i.e. each agent can work independently from others and thus the agents of a multi-agent system can in principle be executed in parallel. Therefore, multi-agent systems provide a clear metaphor for building potentially massively parallel applications.

With respect to current and emerging trends the importance of distribution and concurrency is further illustrated. In the following three of the most prevailing trends (cf. e.g. [10]) are sketched:

- Ubiquitous computing is the vision of Weiser [15] and means that computing devices become smaller and smaller and finally vanish into the environment. In a ubiquitous environment a lot

[1] Distributed Systems and Information Systems Group, University of Hamburg, Vogt-Kölln-Str. 30, 22527 Germany.

of independent small devices exist and they often need to build up a heterogeneous service network in order to fulfill user requirements. Hence, ubiquitous computing needs a programming paradigm that takes into account distribution as a core concept.

- *Autonomic computing* [7] puts the administration effort of systems in the focus of attention. The basic idea here is to build self-* (-heal, -optimize, -configure, -protect) systems, which can manage themselves by monitoring and adjusting their individual components. Such behavior can only be achieved when components can act independently from each other and take over different responsibilities. Typically, in an autonomous system for each business component a dedicated management component is responsible for controlling the behavior of the first. Hence, autonomous computing architectures are conceptually agent-based and per-se concurrent.
- *Multicore processors* [2] have reached the consumer market since 2006 and improve processing power by providing more than one processing unit on board of a single chip. Nowadays dual or quadcore processors are common and it seems reasonable that in the near future multicore processors will be available with a huge number of cores. In order to exploit that processing power the software has to be programmed in a way that supports multiple processes or threads. This is feasible using state of the art programming languages. Nevertheless the available concepts for controlling concurrency such as semaphores and monitors are very low-level and lead to error prone code that additionally is hard to program. Therefore, a programming paradigm providing higher-level concurrency concepts is needed.

If agent technology provides solutions for these and other challenges the question arises, why agent orientation has not yet made it to mainstream software engineering projects and so few systems have been deployed. In the following some key reasons are presented:

- *Heterogeneity of the research field* leads to the severe problem that there are no agreed-upon conceptual frameworks. Therefore it exists a multitude of programming languages, methodologies and agent platforms making the choice of the right artifacts a hard one.
- *The lack of interest of industry players* causes too few respectively too expensive industry-grade agent tools being available. One reason for the low number of companies selling agent-oriented solutions is the high research and development effort required before agent products can be sold. Many companies tried to use agent technology in the nineties during the agent hype phase and failed due to the immatureness of the field. As no immediate return on invest was producible the paradigm has been abandoned and the current service-oriented architecture hype lets many companies concentrate on the service wave.
- *The lack of integration with existing mainstream technology and tools* makes agent technology risky to use for companies, because they have to decide whether they want to rely on proven technologies or utilize agent approaches – mostly they cannot have both. But, as agent orientation has orthogonal advantages with respect to existing paradigms, the choice in such a setting is always to use the proven technology.

In the following it will be investigated how these challenges can be addressed under a technical viewpoint, i.e. with respect to adequate infrastructure support. For this purpose in Section 2 the requirements for industry-grade agent software are examined. In Section 3 existing agent solutions are evaluated from the perspective of a business user. Thereafter, in Section 4 the Jadex V2 architecture is presented and it is detailed how the architecture helps to address the issues mentioned. In Section 5 the content is summarized and an outlook to planned future work is given

2. Requirements

Much has been said (cf. [8, 10]) about the advantages of agent technology for building applications in the context of the current and emerging trends as outlined in the introduction. Many of these advantages are closely related to *functional requirements*, such as the ability to adequately react to

changes in a dynamic environment [9] or the necessity of high-level abstractions to support decomposition and organization of large application structures [8]. For a widespread industrial adoption, a technology not only has to provide the functional capacities to solve the specific problems at hand, but also has to answer the question *how well* it embeds into some given business context.

In [3], a criteria catalog was developed that includes, besides functional aspects, especially also *non-functional requirements* that allow assessing how well an artifact under investigation fits to the peculiarities of an application context. These non-functional requirements (also called software quality attributes) are the key-factor when moving from research prototypes to industrial quality software. In the following, the most important criteria are quickly sketched.

- *Usability* refers to the features of an artifact from the viewpoint of a single user or usage context. Relevant criteria in this category are *individualization* and *extensibility*, which state how easily an artifact can be adapted to a concrete usage context. From a developer perspective, *simplicity/intuitivity* and *learnability/familiarity* are important factors that indicate how fast and efficient an artifact can be put into practice.
- *Operating ability* criteria are *performance*, *robustness*, *stability* and *scalability*, which describe how well the artifact behaves when considering varying system loads (performance), presence of failures (robustness), long term behavior (stability) and varying problem sizes (scalability).
- *Pragmatic aspects* are not related to using or operating an artifact, but mostly to the incurred efforts, costs and risks. Concretely, the *maturity* of an artifact has a direct effect on the risk of a software project. *Costs* are usually comprised of initial (purchase) costs and follow-up costs, e.g. for operation/maintenance or for training of personnel. Finally, additional efforts typically arise due to *technical boundary conditions* of an artifact, such as incompatibilities to other existing, required or preferable technology.

It should be noted that a simple "the more the better" rating (e.g., of performance) is not suitable for any concrete software project, because these criteria are highly interrelated. Depending on the business context, suitable trade-offs have to be found, e.g. between simplicity, performance and costs. As a consequence, generic software infrastructures, e.g. for agent based systems, need to be adaptable to business needs.

3. Related Work

As outlined in the last section, industrial uptake of agent technology does not directly follow from the inherent conceptual advantages compared to more conventional software paradigms like objects, components or services. In addition to those conceptual advantages, agent infrastructures need to exhibit *quality attributes* similar to the conventional implementations, such as Java EE application servers. These two aspects can be paraphrased as *"having something to add"* and *"having nothing to remove"*. Agents have something to add to existing solutions, because they, e.g., facilitate more flexibility and quick adaptation to changes and can introduce a higher conceptual level when considering goals ("what should be done?") instead of only actions ("how is something done?"). But agent infrastructures should also have nothing to remove. Especially features, which are well supported by conventional paradigms, like the support for modular software development, integration with existing technologies such as databases or user interface frameworks and high operating ability (scalability, robustness, etc.), could be knock-out criteria, when being absent from agent-based solutions.

Infrastructures for agent development therefore need to balance the two above mentioned aspects. Existing agent infrastructures can be broadly categorized into two different approaches. The first approach focuses on the first aspect ("having something to add"), i.e. this approach aims at building specialized agent-oriented solutions. The second approach tries to "having nothing to remove" by

reusing existing conventional tools and techniques as much as possible and adding agent-oriented concepts and technology only in a selective way.

The primary advantage of building specialized agent-oriented solutions (first approach) is that agent-specific characteristics can be taken fully into account. Therefore, the expressive power of the agent paradigm can be used for building various aspects of an application. The major drawback of this approach is that these specialized solutions usually do not fit well into existing traditional infrastructures. This means that existing features of e.g., applications servers, such as persistence, can not easily be reused in the agent context. As a result, proprietary solutions are implemented to support these features in a more agent-suitable way. An example of this approach is the widely used JADE agent platform (see e.g.[1]), which aims at providing agent-oriented middleware services according to the FIPA specifications [13]. For supporting non-functional requirements, additional components have been developed that e.g. support agent persistence or reliable messaging. Among the advantages of JADE is the simplicity, such that JADE can also be used for rapid proto-typing of agent applications, and the large user base, which results in numerous add-ons of varying quality being available. JADE has limitations with regard to the seamless integration with mainstream technology and poor scalability due to the use of a thread-based concurrency model. Further examples of the first approach are other agent platforms such as 3APL and Jason [1].

The main motivation behind the second approach (reusing existing conventional tools and techniques) is minimizing the gap between agent technology and the software engineering mainstream. In this approach, the seamless deployment of agents or agent-like features into existing object-oriented/component infrastructures is seen as the appropriate way to overcome the barriers that currently hinder the adoption of agent technology in the industry. Besides encouraging industry adoption, a major benefit of this approach is the reduced implementation effort and increase in maturity due to relying on proven industry grade products. Moreover, due to adherence to industry standard APIs, developers can choose from the large body of commercial or open source implementations (e.g. of Java EE application servers), which in turn might offer certain advantages or disadvantages depending on the application context. The drawback of this approach is that the usage of standard technologies imposes certain constraints on which and how agent concepts can be implemented. Depending on the used basis, therefore usually only a subset of existing agent features can be realized without compromising the mainstream integration and as a result, only advantages specific to those agent features can be obtained. Examples of this approach are agent platforms such as Whitesteins LS/TS [14] and the Agentis AdaptivEnterprise Suite. Both focus on integration with a Java EE technology base, but whereas the main aim of the Agentis suite is integrating BDI-style goal orientation, LS/TS puts more weight on agent concurrency and message passing. Among the advantages of LS/TS is the ability to run the same application on different execution environments (Java SE/Java EE) targeted to development vs. production settings and also the availability of different reasoning engines ranging from simple task-based to more high-level goal-oriented agents. Main criticism with regard to LS/TS is the high complexity, which makes it unsuitable for rapid prototyping of agent applications and the fact that the platform is closed and therefore no user community is available. Besides platforms, there are also programming languages like the JACK agent language [1], extending languages like Java to include agent-oriented constructs.

Both approaches have their merits. The first approach is suitable for contexts in which agent advantages have a big payoff (e.g. logistics domains, [12]) and therefore the easy integration with mainstream technologies is not so important. On the other hand, the second approach allows integrating agent-oriented ideas into an existing business IT landscape and is therefore seen as the more promising approach for achieving mainstream industrial uptake of agent technology.

4. Jadex V2 Architecture

The Jadex agent framework (see e.g. [1]) mainly consists of the agent infrastructure, and tools for the development of agent systems. The concrete requirements for Jadex V2 have been revealed by an evaluation of V1 against the criteria catalogue from Section 2. In the following a short review of the evaluation with respect to the non-functional criteria is given.

- *Usability* The usability of Jadex V1 has been evaluated to high due to several reasons. First, Jadex does not introduce a new programming language, but relies completely on a hybrid language approach. This approach distinguishes between structural and behavioral aspects, which are specified in the suitable mainstream languages XML and Java respectively. Second, the BDI concepts are interpreted in an intuitive way, which is near to their folk-psychological meaning and includes an explicit representation of goals. This also allows a natural transition from modeling to implementation via existing goal-oriented methodologies like Tropos or Prometheus. Third, Jadex also fulfills software technical reusability requirements by providing a BDI-based modularization concept called capabilities.
- *Operating ability* The operating ability has been identified as one crucial aspect for industrial exploitation of agent platforms and is hence very well-supported by commercial platforms like JACK or LS/TS. In V1 operating ability has been evaluated to only fair. In order to assess the quality of agent platforms in this respect, several small benchmarks have been conducted on a standard desktop PC as explained in the following. The *performance* has been tested via agent creation resp. termination time. In this respect the creation/termination of agents was at a rate of circa 50 resp. 250 agents/sec. The *scalability* has been measured by the number of agents that can be started on a standard Java VM with normal 64MB heap space. The number of agents was limited to about 200, which is mainly caused by the high 200kb memory footprint of agents. The *robustness* and *stability* are mainly features of the agent platform. Robustness is to some degree ensured by the isolation of agent execution, which prohibits the propagation of errors to the platform layer, i.e. a faulty agent cannot easily cause the platform to fail. The stability has been tested via long-running test-cases, which demonstrate that at the platform as well as the architecture layer no memory-leaks are existent.
- *Pragmatic aspects* This category led to a good evaluation for Jadex V1. The acquisition and installation of Jadex is unproblematic, because it can be directly downloaded from internet and also contains extensive documentation material. Furthermore, the technical boundaries are kept low. This is achieved by a clean separation between the platform and architecture layer and extension points at both layers.

In general, the evaluation showed that the non-functional aspects usability and pragmatics are already covered to a sufficient degree in V1. Hence, in Jadex V2 the objectives are keeping the levels of usability and improvements should especially be targeted to the operating ability area.

The resulting Jadex V2 agent infrastructure is composed of the agent platform and the agent kernel(s) (cf. Figure 1). The responsibility of the agent platform is to ensure the continuous execution of the agents inhabiting the platform. On the contrary, the agent kernel determines the agent architecture used, i.e. it defines which concepts can be used for programming agent behavior. The infrastructure has been designed in such a way that it allows kernels as well as platforms being used interchangeably. On the one hand this means that in combination with a platform different kernel implementations can be used (e.g. BDI and/or rule-based). On the other hand, it is also possible to use a kernel on different kinds of platforms (e.g. Standalone or JADE). The details of the platform and kernel architectures will be explained in the following sections. In this respect it will be especially highlighted how platform and kernel address the requirements from Section 2.

4.1. Platform Architecture

The Jadex platform architecture exhibits two main characteristics:

1. It can execute agents regardless of their internal architecture.
2. It can exploit an arbitrary middleware for reusing available services.

The first aspect is important, because applications differ in the complexity of agents that is required. If e.g. ant algorithms shall be built it is sufficient to use simple reactive agent architectures, while problem solving tasks might require cognitive capabilities and therefore deliberative agent architectures are a better fit. One could even argue that it can have advantages to build parts of the application using different agent architectures according to the complexity of the agents at hand.

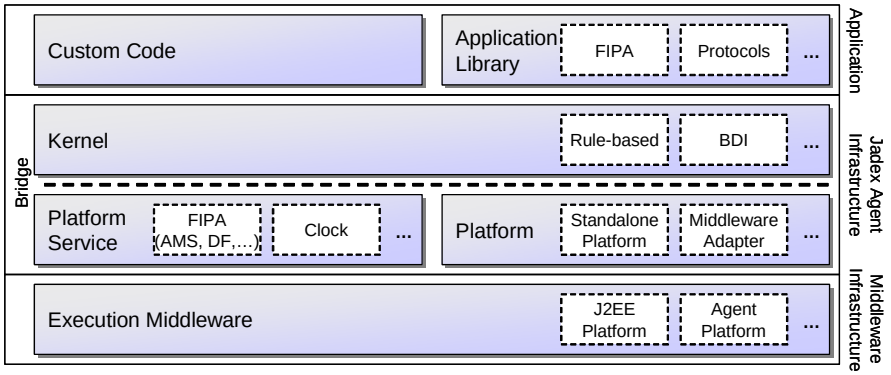

Figure 1: Jadex V2 architecture

In order to execute different agent architectures within a single platform it is necessary to define the responsibilities of the kernel and the platform. A platform has the minimal duties of executing an agent, delivering messages to the agent, notifying the agent at certain time points and indicating that an agent should kill itself. Agent kernels must therefore exhibit the following methods for an agent:

```
public boolean executeAction();
public void messageArrived(IMessageAdapter message);
public void notifyDue();
public void killAgent();
```

The *executeAction()* method has the purpose to execute an agent for a single step, i.e. the agent should perform an action and return the control to the platform in a short period of (CPU) time. If the agent execution exceeds some platform-defined limit, the platform might decide to abort agent execution. The boolean return value indicates if the agent wants to be executed again. If this is not the case it can be reactivated by messages or a timing info. The *messageArrived()* method delivers a message to an agent for processing. It is noteworthy here, that the supplied message has to be an *IMessageAdapter*, which is an interface that hides platform-specific transport details and allows an agent to retrieve all information about the message in a generic way. This means that Jadex does not assume a fixed message structure such as FIPA ACL [13]. Instead, the message adapter provides access to name-value pairs of the underlying message and additionally to the message type. The message type contains meta information about the message and can be used to extract the relevant information from the message. In this way besides FIPA ACL also alternative formats such as Email can be supported. The *notifyDue()* method is necessary for allowing an agent to be reawakened at an agent-defined point in time. For this purpose the agent uses the clock service for registering the desired time point. The service then guarantees to call the *notifyDue()* method on the agent at this time point. Different clock service implementations allow for running the same application in simulation as well as real-time execution mode [12]. Finally, the *killAgent()* method requests the agent to kill itself and starts its takedown process.

For an agent it is necessary to rely on services of the platform adapter. These basic services are especially needed for sending messages and, waking resp. cleaning up the agent, and fetching the platform. A cutout of the adapter interface is shown below:

```
public void sendMessage(IMessageAdapter message);
public void wakeup();
public void cleanupAgent();
public IPlatform getPlatform();
```

Using the *sendMessage()* method an agent can send a message via the platform. The platform is then responsible for all aspects of the transport, which can be either locally or remotely depending on the addresses of the target agents. The *wakeup()* method is needed to ensure that an agent's processing can be activated. It is e.g. called when a message is placed in the agent's inbox or other (kernel-specific) events happen and requests the platform scheduler to call *executeAction()* on the corresponding agent. *CleanupAgent()* is the platform equivalent to *killAgent()* on the agent side and initiates the removal of the agent. An agent can call this method if it wants to be disposed. The platform always has to know when an agent is going to be killed, because it handles the agent's resources and has to release them afterwards. Finally, the *getPlatform()* method gives an agent access to the platform itself and its platform services (cf. Fig. 1). Platform services are a mechanism for building up the platform's functionality in an extensible manner. If the platform should e.g. support the FIPA standards, services for AMS and DF can be added.

Agent platforms in most cases are devised for a specific application scenario, e.g. a lightweight platform for a mobile device versus a heavyweight platform for back office functionalities. The second criterion of being able to reuse existing middleware allows developing different kinds of platforms for Jadex agents and facilitates the integration with proven solutions. Picking up the example it makes sense to build a platform for mobile devices with a low resource footprint restricting its functionalities to a minimum, whereas a platform for back office tasks needs features like high dependability and hence could be build upon a Java EE server.

Regarding the requirements discussed in Section 2 the platform architecture directly contributes to the usability and pragmatics and indirectly also to the operating ability criteria. With respect to the usability it supports individualization and extensibility to a high degree. On the one hand kernels and platforms are rather independent of each other and on the other hand the flexible service-based platform infrastructure supports the adaptation of platforms according the application context. Considering the pragmatic dimension the proposed architecture mainly contributes to the technical boundary conditions, because it facilitates the integration with existing middleware infrastructures. Finally, operating ability is indirectly supported by the customizable service approach of the platform, which allows services being tailored to the concrete demands. E.g. if stability and robustness are crucial, persistent AMS und DF services could be developed.

4.2. Rule Kernel Architecture

To simplify the development of different kernels as part of the Jadex agent infrastructure, a basic rule kernel has been devised. The goal of this kernel is to form a basis for different concrete reasoning kernels that already provides or simplifies the provision of quality attributes. The rule kernel ensures that concrete kernels can be built rather independently from (non-functional) quality attributes thereby focusing on functionality (i.e. describing the behavior of agents). It is realized as a generic rule engine and concrete kernels are specified in terms of a state structure and transition rules that operate on the state (see Figure 2). The declarative specification of the concrete reasoning engine leaves much more room for optimizing the execution in different directions than what would be possible with a procedural implementation.

Basis of the rule kernel is the interpreter, which connects the kernel to the underlying platform by implementing the required Java interface (cf. Section 4.1). The interpreter itself represents a typical rule engine [5] consisting of an agenda, a pattern matcher and a state or working memory. Rules are specified in a CLIPS-based [5] condition language, whereby the action part is assumed to be pure Java. The pattern matcher determines based on the state and a given set of rules at any time the

possible variable assignments to fire some of the rules. The matching rules with corresponding variable assignments are stored as so called activations in the agenda. In each agent execution step, the interpreter fires one activation from the agenda, and informs the pattern matcher about the incurred state changes. The pattern matcher then updates the agenda to reflect newly matched or no longer matched activations. In addition to activations from the agenda, the interpreter has to execute so called external entries, which represent asynchronous occurrences happening parallel to the agent execution, such as messages received from other agents. During each execution step, these entries are copied to the state, leaving details of, e.g., message processing to the concrete kernel rules.

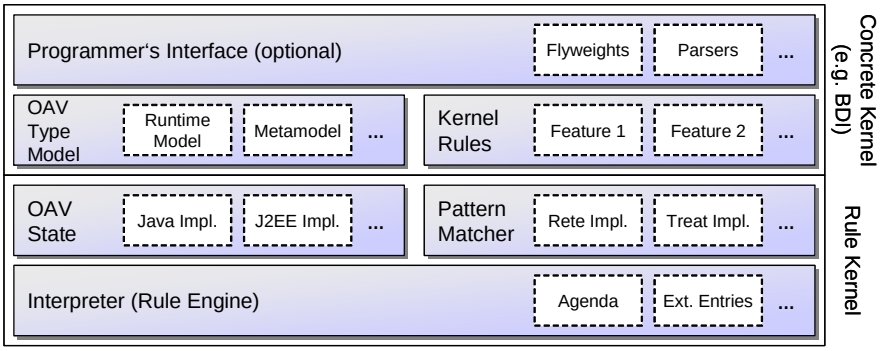

Figure 2: Jadex V2 Rule Kernel Architecture (concretizes kernel layer of Figure 1)

The state uses an OAV (object-attribute-value) triplet representation of data, because this representation can be easily mapped to different implementations (e.g. Java objects, database tables, RDF, etc.) that support different quality attributes. The type model describes the structure of data items that can be stored in the state and is used for data access as well as (optional) runtime consistency checking in the rule kernel. Usually a separation in metamodel and runtime elements is useful, to distinguish between data shared among all agent instances of the same type and runtime data that is private to each agent instance. The kernel rules describe all possible forms of state transition and thereby determine the behavior according to a specific agent architecture. Different implementations of the pattern matcher exhibit different performance characteristics. E.g. the Rete algorithm-based [4] implementation has a higher memory footprint due to caching partial matches, but has superior execution speed compared to less memory consuming implementations, such as TREAT [11]. The manipulation of data based on an OAV model and the specification of behavior in form of rules represent very low-level ways of interfacing with the system. Therefore, concrete kernels will usually offer a higher level programming interface, which hides low-level details. For example, flyweights [6] can provide a Java object like access to the data in the state and parsers can hide rule details behind Java-like expressions.

The rule kernel architecture contributes to achieving desired quality criteria (cf. Section 2) by providing an efficient base layer for concrete kernels. Operating ability criteria such as scalability and performance are achieved using proven rule-based technology such as the Rete algorithm. Individualization is supported by providing different implementations of the components. Therefore, different state representations allow adequate fulfillment of conflicting requirements depending on the situation (e.g. efficiency using in-memory objects, vs. robustness using database storage) and different pattern matcher implementations allow to choose between fast execution and low memory footprint. Also, in the pragmatics area, for technical boundary conditions different implementations can be developed for concrete execution environments, such as a state representation based on EJB technology for Java EE execution requirements.

4.3. BDI Architecture

The Jadex BDI architecture has been described extensively in the literature (e.g. in [1]) and it has been one major objective of Jadex V2 to keep the BDI functionality and the offered programming

interface as similar as possible. Hence, here the Jadex BDI model will only be outlined briefly. In general, BDI agents are programmed with beliefs (subjective knowledge), goals (desired outcomes) and plans (procedural code for achieving goals). Jadex distinguishes the procedural knowledge in plans from the descriptive knowledge and the former is specified in Java classes while the latter is defined in an XML-based ADF (agent definition file). From within Java plans the developer can access agent functionality via API calls, which e.g. allow accessing beliefs or dispatching goals.

In order to keep this programming model the same as in V1 a special layer has to be placed on top of the rule kernel. This layer internally uses the functionalities offered by the rule kernel and adds Java-based access facilities in the style of the flyweight pattern [6]. This means whenever the user calls an BDI-based method (such as *getBelief()*) the new layer will create a flyweight (here Belief-Flyweight), which exposes the same functionality as in V1 (here e.g. *getFact()*) and internally translates calls to OAV state operations.

Internally, the BDI state representation as well as the BDI functionality has been specified in terms of a rule engine implementation. As already denoted the BDI state representation consists of two OAV type models: a metamodel and a runtime model. The metamodel contains elements that can be used for defining Jadex agent types and therefore allow specifying application-specific agent types and their beliefs, plans and goals. In addition, a parser has been implemented that reads/writes XML agent definitions to/from an OAV state. In contrast, the runtime model contains information about agent instances, such as the agent's current belief, goal and plan instances.

In addition to the state representation the BDI functionalities have been rebuilt using production rules. For this purpose the BDI functionalities have been categorized into different groups such as belief handling, event processing, goal processing, goal deliberation, and many more and for each group several rules have been devised. An example rule is illustrated in CLIPS-like notation below:

```
?plan <- (plan (processingstate "ready") (lifecyclestate "body"))
?agent <- (agent (plans contains ?plan))
=> // Java code for plan execution
```

The execute plan body rule states that whenever an agent has a plan that is in processing state "ready" and in lifecycle state "body" an activation will be created. The action side of this rule contains the concrete Java code for plan execution and in this case just resolves the value of the bound *?plan* variable, fetches its plan body and invokes it.

In V2, the existing set of V1 BDI functionalities has been completely rebuilt using the rule-based approach. Because rules have been grouped according to their functionality, it is now possible, to use streamlined BDI rule sets, e.g. when some advanced features, such as goal deliberation are not required. Additionally, new functionalities (e.g. emotions) can much more easily be added to the existing BDI functionality. Besides the BDI kernel it is planned to also develop a reactive kernel and a task-model based kernel. While the reactive kernel will consist of a simple API directly on top of the basic rule kernel (e.g. for ant-like agents), the task-model kernel will not use the rule base functionality but instead provide a simple object-oriented agent programming model (cf. JADE).

4.4. Evaluation

The new Jadex V2 architecture has been devised in a way that makes the underlying rule kernel transparent to a high degree for the agent programmer. Hence, the evaluations for usability and pragmatics are not affected by the changed architecture, whereas the operating ability has been assessed again. In general the V2 operating ability has been evaluated to good, because the performance and scalability could be improved. Regarding the performance benchmark creation and deletion of agents could be improved by an order of magnitude to 500 resp. 2000 agents/s. Also the scalability could be improved. The number of agents that can be started on a standard Java VM with normal 64MB has enhanced to over 2000, each with a footprint of circa 25kb. This shows that the

proposed architecture is able to preserve the advantages of Jadex V1 while the operating ability is now comparable to commercial solutions available.

5. Conclusions and Future Work

In this paper it has been argued that current trends such as ubiquitous computing and multicore processors require advanced *distribution* and *concurrency* concepts that are covered insufficiently by established software paradigms like object-orientation. Agent orientation offers solutions for these problems but is not extensively used in practice. One important reason for this is that the lack of integration with existing technology and tools. In this respect, only if agent systems "have something to add" and "have nothing to remove" they will be considered to be used in commercial settings. Nothing to remove means here that functional and in particular non-functional properties of existing solutions should be kept by agent-based offerings.

To achieve this, the new Jadex V2 framework architecture has been presented. One main characteristic of this architecture is the strict separation between platform (execution environment) and kernel (agent architecture), which allows both being exchanged independently. In this way, on the one hand different agent architectures can be used on the same platform (e.g. BDI versus rule-based) and vice versa one kernel can be used in multiple execution environments (e.g. a mobile- vs. server-based implementation). The platform architecture itself is characterized by a flexible approach using pluggable services, facilitating extensibility and adaptability to the application context. For kernel implementations a rule kernel base has been proposed, which offers a rule mechanism and an OAV state representation. On this (optional) rule kernel concrete kernel implementations such as BDI can be built. Using the rule kernel has advantages especially concerning non-functional aspects, because its usage is independent of its implementation and components can be adapted according to the concrete demands (e.g. for a fast execution the Rete rule matcher can be used).

To further improve robustness and stability, future work will focus on the platform level. A Java EE platform adapter will allow executing agents in the context of application servers, which offer major advantages in the area of non-functional requirements by features such as built-in transactional execution, load management and replication. Furthermore, the standardized deployment procedures of application servers will lower the barrier for adoption and installation of agent-based solutions.

6. References

[1] Bordini, R., Dastani, M., Dix, J., El Fallah Seghrouchni, A., Multi Agent Programming, Springer, 2005.
[2] Borkar, S., Thousand core chips: a technology perspective. In Proc. of (DAC '07), ACM, 2007.
[3] Braubach, L., Pokahr, A., Lamersdorf, W., A Universal Criteria Catalog for Evaluation of Heterogeneous Agent Development Artifacts, in: From Agent Theory to Agent Implementation (AT2AI-6), 2008.
[4] Forgy, C., Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. Artif. Intell. 19(1), 1982.
[5] Friedman-Hill, E., Jess in Action - Rule-based Systems in Java, Manning, 2003.
[6] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns, Addison-Wesley, 1995.
[7] IBM, Autonomic Computing: IBM's Perspective on the State of Information Technology, IBM, 2001.
[8] Jennings, N., An agent-based approach for building complex software systems. Commun. ACM 44, 4, Apr. 2001.
[9] Kirn, S., Herzog, O., Lockemann, P., Spaniol, O. (Eds.), Multiagent Engineering, Springer 2006.
[10] Luck, M., McBurney,P., Shehory, O., Willmott,S., Agent Technology: Computing as Interaction, AgentLink, 2005.
[11] Miranker, D., TREAT: A Better Match Algorithm for AI Production System Matching, Proc. Of. AAAI'87, 1987.
[12] Pokahr, A., Braubach, L., Sudeikat, J., Renz, W., Lamersdorf, W., Simulation and Implementation of Logistics Systems based on Agent Technology, in: Hamburg International Conference of Logistics (HICL'08), 2008.
[13] Poslad, S., Charlton, P., Standardizing agent interoperability: the FIPA approach. In Multi-Agents Systems and Applications, Springer, 2001.
[14] Rimassa, G., Greenwood, D., Kernland, M., The Living Systems Technology Suite. In Proc. ICAS'06, 2006.
[15] Weiser, M., The Computer for the Twenty-First Century, Scientific American, September 1991.