On the Validation of Belief–Desire–Intention Agents

Jan Sudeikat[1,2], Lars Braubach[1], Alexander Pokahr[1], Winfried Lamersdorf[1], and Wolfgang Renz[2]

[1] Distributed Systems and Information Systems,
Computer Science Department, University of Hamburg,
Vogt–Kölln–Str. 30, 22527 Hamburg, Germany
Tel. +49-40-42883-2091
{4sudeika|braubach|pokahr|lamersd}@informatik.uni-hamburg.de

[2] Multimedia Systems Laboratory,
Hamburg University of Applied Sciences,
Berliner Tor 7, 20099 Hamburg, Germany
Tel. +49-40-42875-8304
{sudeikat|wr}@informatik.haw-hamburg.de

Abstract. Testing and Debugging multi-agent systems (MAS) - which are inherently concurrent and distributed – is a challenging task. While complex application scenarios demand intelligent entities with autonomous reasoning capabilities, the applied reasoning mechanisms impair current approaches to validate MAS implementations. Reactive planning systems, namely the well-known *Belief Desire Intention* (BDI) architecture, have been successfully applied to implement these intelligent entities by means of goal directed agents. Despite testing and debugging, used to validate whether implementations behave as intended, are crucial to serious development efforts, only minor attention has been payed to corresponding tool support and testing procedures for BDI–based MAS. In this paper we describe and categorize common bugs in BDI–based MAS implementations and discuss similarities and differences to general software testing procedures. We particularly examine how the reasoning mechanism inside agent implementations can be checked and how static analysis of agent declarations can be used to visualize and check the overall communication structure in closed MAS. We present corresponding tool support, which relies on the definition of crosscutting concerns in BDI agents and enables both approaches to the Jadex Agent Platform.[3]

1 Introduction

Agent-orientation proposes autonomous, proactive entities, so-called agents [1, 2], as an atomic design and development metaphor for software systems. These entities enable a lifelike decomposition of software systems as independent actors, interacting with each other. Besides simple reactive agents [3] have been

[3] Draft version for submission to PROMAS 06

successfully applied in various application domains, the BDI architecture has been established to develop *deliberative* agents [4, 5]. Methodologies and development tools are in active development to support the construction of software systems, utilizing this specific architecture. Implementations of this model use the concrete concepts of *beliefs*, *goals* and *plans*, to design and implement individual agents [6, 7]. Beliefs denote the local knowledge of individual agents, goals describe the agents objectives and plans are the executable means by which agents satisfy their goals. These concepts allow agents to reason pro–actively about which actions to take, i. e. plans to execute.

The autonomous nature of these entities, their complex interactions and their individual memory and reasoning capabilities introduces novel levels of uncertainty [8] to these software systems. While traditional development approaches design the flow of control in a software system, the individual agent knowledge and reasoning capabilities may lead to unexpected individual behaviors, inhibiting predictions of agent actions and interactions. A major challenge for these systems is the validation of internal reasoning processes.

In this respect we discuss currently proposed approaches to test and debug MAS, focusing on approaches for BDI–based agents. We present how assertions can be used in BDI concepts to support testing and debugging of BDI agents. Implementations of BDI agents declare the structural properties of BDI agents. The properties comprise *messages* to be sent and received as well as agent internal event mechanisms. We present how these declarations ca be analysed for validation.

This paper is structured as follows. The next section introduces the BDI architecture. In section 3 testing and validation approaches to MAS are examined and current approaches, particularly concerned with BDI architectures, are discussed. The following section 4 presents our approaches to validate agent implementations. We present a mechanism to execute assertions on BDI concepts (4.1) and two static analysis approaches (4.2). After we exemplify their usage and implementation for the *Jadex* system (section 5), we conclude and give prospects for future work.

2 The BDI Agent Architecture

A successful architecture to develop deliberative agents is the BDI model. Bratman [4] developed a theory of human practical reasoning, which describes rational behavior by the notions *Belief*, *Desire* and *Intention*. Implementations of this model replaced the latter two by the concrete concepts *goals* and *plans*, leading to a formal theory and an executable model [5, 9].

Beliefs represent the local information of agents about both the environment and its internal state. The structure of the beliefs defines a domain dependent abstraction of the actual environment. It can be regarded as the *view–point* of an agent towards its surrounding. The goals represent agent desires, commonly expressed by certain target states inside the beliefs. This general concept enables pro–active agent behaviors. Agents carry out these goals on their own (see [10] for a discussion of goals in BDI systems). Finally, plans are the executable means by which agents achieve their goals. Agents can access a library of plans and deliberate which plans to execute, in order to reach a desired target. This mechanism is also known as *reactive planing*, because the precompiled plans are developed at design time. Single plans are not just a sequence of basic actions, but may also dispatch sub-goals.

Both reactive and pro–active behaviors are enabled by internal reasoning processes, composed of *goal deliberation* [11] and *meta-level reasoning* (problems of this are discussed in [12]). The former is the process to select goals to be pursued by an agent, while the latter one is responsible to select plans for execution in order to satisfy the previously selected goals. To allow appropriate reasoning, the goals and plans are annotated with *conditions*, describing constraints on their applicability.

3 Testing and Debugging Multi–Agent Systems

Engineering approaches need to ensure the quality of developed systems. In this respect *quality* is commonly understood as a set of properties (e.g. [13]) of a (software) product or an activity, which are related to the fulfillment of predefined requirements. In [14] evaluation activities for complex, possibly adaptive and/or distributed software systems have been classified with respect to the amount of expertise required for their application by developers (cf. figure 1) . *Testing* uses

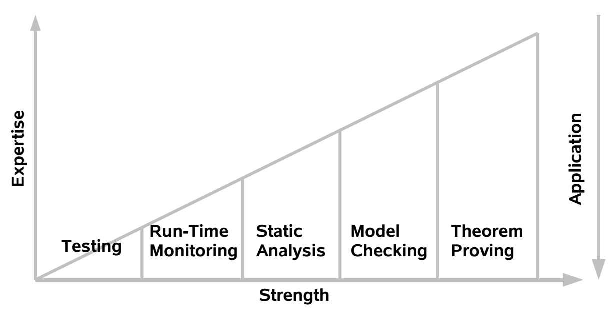

Fig. 1. Categories of evaluation Techniques according to [14]

special programs to simulate input sequences and compare them to specified outputs. *Run–time monitoring* enables analysis of run–time behaviors by observation of applications with specified input-parameters resp. under pre-defined conditions. *Static analysis* examines the structure of source codes without executing it, while *Model Checking* verifies that a system satisfies a specification by examination of all reachable states. Finally, *Theorem Proving* enables formal proofs of correctness.

Both Model Checking and Theorem Proving provide best confidence in source codes but require high cost in both specification effort and computation. While ongoing research [14] is enhancing their applicability, these requirements often impair their application in commercial development settings. These mainly rely on monitoring, static analysis and testing (see for a general discussion of testing approaches for MAS [15]). While we address monitoring of BDI agents in [16], we focus here on the latter two. Figure 2 summarizes the different testing stages

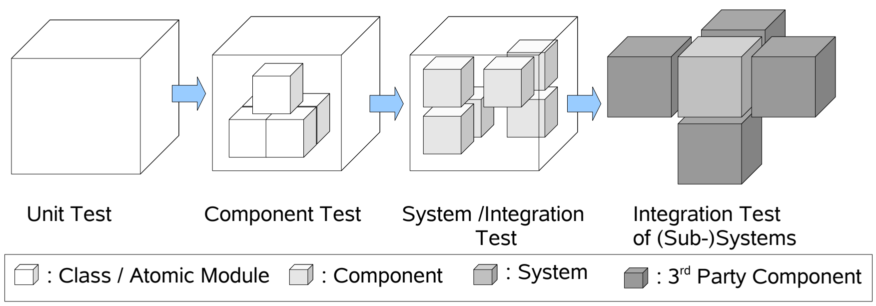

Fig. 2. Typical testing stages for a BDI–based multi–agent system. The System is developed from single modules expressing functionalities to an integrated solution that interfaces 3rd party software packages.

from an single component to an integrated application which is embedded in an IT infrastructure (cf. [17]). While testing is commonly understood as an ongoing activity, iterated and adjusted to different abstraction levels, MAS literature mainly addresses testing procedures for individual agents, namely agent execution and agent communication.

3.1 Testing Agent Execution

Jade testsuite / Whitestein testsuite
[18],[19]

3.2 Testing Agent Communication

The communication between agents is an inherent and foundational property of MAS while the exchanged messages are clearly defined artifacts to be recorded and analysed. Various tool support has been developed to verify the message exchange and the adherence to communication protocols. ACLAnalyser? [20],[21],[22], [23], [24]

3.3 Testing and Debugging the BDI Architecture

A comprehensive testing strategy for BDI agents is outlined in figure 3. According to figure 2 testing procedure should move from basic functionalities, which can be conveniently captured by capabilities [12, 25] to the interplay of these in individual agents. Approaches to verify the interplay of the agents among each other and among the MAS with a surrounding IT infrastructure are expected to differ only slightly.

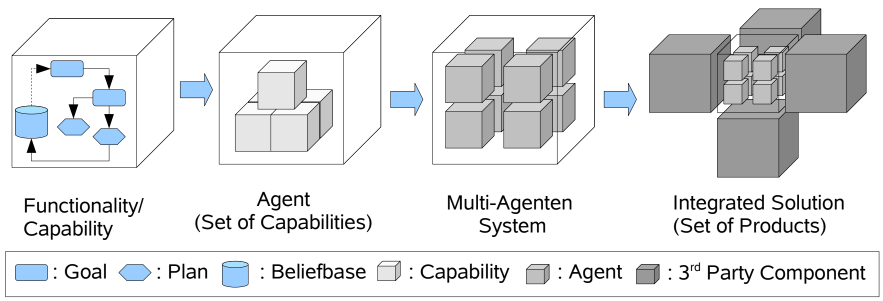

Fig. 3. Typical testing stages in a software project. The System is developed from single functionalities (possibly modularized) to an integrated solution which interfaces 3rd party software packages.

Current approaches to validate BDI agents are concerned with (1) the compliance of agent execution to design artifacts drawn from development methodologies [24], (2) the comprehension of agent behaviors by comparison of models of the expected order of reasoning events [26, 27] and (3) test case generation for BDI–plans, based on coverage criteria.

While the *RMIT* uses design artifacts from the *Prometheus* methodology [28] mainly for validation of communication protocols [20, 24] and agent communication [21], they also define and test for *coverage* and *overlap* of BDI plans. BDI reasoning events have full coverage when the event is expected to have always an applicable plan and an overlap for an event describes that multiple plans may be applicable to handle it [24]. In [24] these criteria are validated by automated introduction of logging code to monitor plan adoption.

In[27] a so–called *Tracer* is visualizing the order of reasoning events in graph structures. In addition, the found sequences of events are compared to models of intended sequences. Differences in the models highlight differences between possible event sequences in actual implementations and the intended agent internals.

In [29] a comprehensive coverage oriented testing methodology has been proposed. The Tool *BDITester* utilizes a subsumption hierarchy of coverage criteria which have been adjusted to plan execution to generate test cases for agent plans.

4 A Practical Approach to the Validation of BDI Reasoning

In the case of BDI agents, proper functioning is based on the processed BDI concepts. These comprise (1) belief consistency, (2) proper goal adoption and consistency and finally (3) correct plan execution. Since agents reason pro–actively about their goal and plan adoption, verification of agent *reasoning mechanisms* is crucial. While developers declare the BDI concepts and annotate conditions to their applicability in order to describe the behavior of the goal directed agents. A comprehensive testing procedure needs to be able to assure that agents will come to the intended conclusions, i. e. adopt appropriate goals and plans. Functional properties only ensure that the subsequent actions are executed properly.

In order to address these issues on BDI agent validation, we present a novel testing and validation approach for BDI agents. It comprises two parts: First, we found that the contributive execution of assertions statements is useful to identify misconceptions in agent code. Secondly, a static analysis of BDI agent declarations has been enhanced to check both the consistency of predefined internal events and messages.

4.1 Assertions in BDI-Concepts

Assertions are typically provided as extensions to programming languages[4]. After we briefly introduce assertions in general, we classify which properties in BDI reasoning can be validated using them in BDI agents. Their usage in the *Jadex* system (cf. section5.1) is exemplified in section 5.

Assertions in Software Engineering Following Hoare [30], an assertion is:

> "... a Boolean formula written in the text of a program, at a place where its evaluation will always be true or at least, that is the intention of the programmer..."

If an assertion statement evaluates to false, the program has entered an inconsistent state. Assertions have their origin in program verification [31, 32] and can be traced back to the founding works of Turing [33], who introduced this concept to specify interfaces between parts of programs. Despite their age, assertions are widely used in the software industry. The *design by contract* principle [13] is closely related to object–oriented development and assertions lend themselves to detect, diagnose and classify violations of these contracts specified as *pre*– and *post–conditions* (e.g. in the *Eiffel* programming language).

Defects in programs can only be identified when testing efforts lead to incorrect output to be observed. This *observability* of software artifacts requires that (1) that an input causes a defect code to be executed, (2) the program data in the succeeding state gets corrupted and finally (3) the corrupted data

[4] e. g. introduced to Java in version 1.4

propagates to an output state leading to incorrect output [34]. Components are commonly tested using *unit–test*[5] frameworks. These facilitate instantiation, automated method calls and comparison of corresponding return values to output specifications. While these tests can usually be used to examine corrupted object states, *encapsulation* and *information hiding* may mask errors in *integration* and *system–level* tests, used to examine the interplay of components and subsystems. In [34], assertions have been proposed as means to increase the likeliness that incorrect outputs occure when defected code is executed.

A Classification of Assertions in BDI Architectures As described in section 3.3, the validation of the BDI–based reasoning process is a major challenge in testing and debugging BDI agents. While only message exchange and external agent actions are observable on the MAS level, it is necessary for developers to gain confidence that the intended goals and plans are adopted during agent execution. While encapsulation and information hiding may be detrimental to state–error propagation in object–orineted systems, the same is true for the event–based and condition centric reasoning cycles in BDI agents.

In this respect assertions can be used to (1) specify and ensure *interfaces* between BDI concepts and (2) ensure *invariant* properties in agent execution. While the conditions which are annotated to BDI goals and plans enable automated reasoning, developers intend specific agent properties and behaviors. Explicit statements of these intentions for all BDI concepts supplement concept properties and highly increase the observability of unintended and/or inconsistent agent states. According to the *design by contract* principle [13], similar contracts – between agent states and BDI concepts – can be specified using assertions.

For *beliefs* the contracted properties are certainly specification of certain subranges of belief values. Since some BDI frameworks allow the storage of generic objects class specific properties may be restricted. When *goals* are adopted the subranges of parameters values my be restricted. Invariants may be specified by assertions in order to highlight when unintended goals are instantiated in specific agent states. These assertions supplement the creation, context and drop conditions for BDI plans and increase observability. Also for plans the value ranges of parameters and return values can be checked.

The annotated conditions also provide additional documentation for the agent code. In section 5 we present an implementations that executes assertions at every state change of the annotated element. Developers need to be aware of negative side effects. First, the execution of assert–statements is intrusive to the agent execution and these statements may have undesired side effects. Extensive processing in these statements will slow down the agents and side effects may impair proper execution.

---

[5] e. g. http://www.junit.org

4.2 Static analysis

BDI–based MAS are composed of a set of agent declarations which define the properties of BDI concepts and further implementation dependent details. Figure 4 gives a canonical overview of such a MAS, composed of n agents. Among the declared details are the *messages* to be sent and received as well as *internal events* which may trigger plan execution (e. g. found in [5, 35, 36]). The consis-

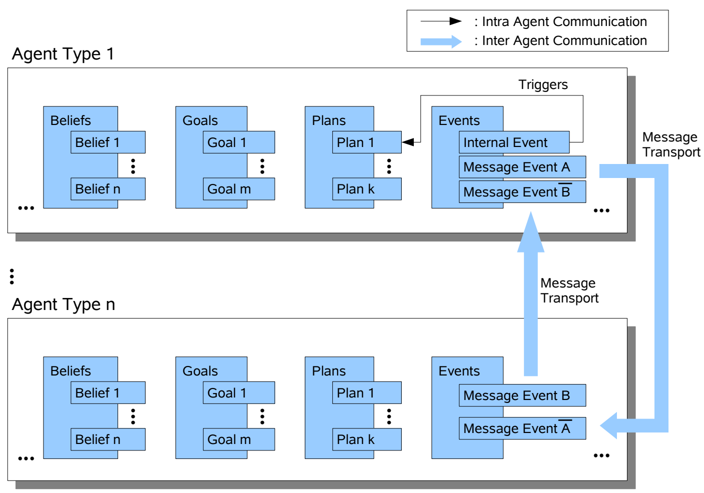

Fig. 4. A canonical view on a Jadex–MAS. One to n agents are declared. Among the properties of BDI concepts and implementation details, *Internal* and *Message Events* are specified. Only matching message events, e. g. A and $\bar{A}$, enable communication.

tency of these two important implementation concepts can be checked in order to validate structural properties and highlight misconceptions. A prototype implementation analyses both kinds of event declarations, the latter ones are ordered in a graph structure to display the possible communications in the MAS under consideration.

The Static Structure of Internal–Events Internal events typically trigger plan adoption as a mean of intra–agent communication (cf. figure 4). Therefore proper specification of these events and their triggering function can be analysed straight forward by iteration and comparison. This allows to validate that all specified events actually trigger plans and vice versa all triggers are correctly declared.

Static Analysis of Message–Events As outlined in figure 4 agent declarations comprise the messages to be sent and received. Therefore a set of agent declarations can be used to analyse the communication structure of a closed MAS. Message declarations comprise FIPA[6] compliant performatives, utilized ontologies, encryption schemes, specific content types, among others. Only matching properties enable proper message exchange. While these declarations only reflect the static possibility of message exchange they can be used to identify messages which can not be send or received.

5 A Case Study – The Marsworld in Jadex

To exemplify the usage of the above described testing and analysis tools, we examine an example MAS from the Jadex–Project. This example scenario has been inspired by a case study in [37], where hierarchical structures of static, predefined roles are examined. In order to allow for cooperative behavior, the system has been generalized as follows. The objective for a group of robots (agents) in the so–called *Marsworld*, is to mine ore on a far distant planet. The mining process is composed of (1) *locating* the ore, (2) *mining* it on the planets surface and (3) *transporting* the mined ore to the home base. Therefore, a collection of three distinct types of agents are released from a home base to a bounded environment. All of them have a sensor range to detect occurrences of ore in the soil an start immediately a searching behavior. Sensed occurrences of ore are reported to the so-called *sentry*-agent. This robot is equipped with a wider sensor range and can verify, whether a suspicious spot actually accommodates ore. When ore is found, the location is forwarded to a randomly selected *production*-agent, equipped with a dedicated mining device. After mining is finished a group of *carry*-agents is ordered to transport ore to the home base (constant number of round trips). In this scenario agent change between two distinct behaviors. They either search for ore or perform a dedicated action, i.e. sense ore, mine ore or transport ore. When the ordered actions have been performed agents continue searching. Details on the game dynamics can be found in [16].

5.1 The Jadex System

The Jadex research project[7] [38],[39], provides the BDI–concepts on top of the well known JADE[8] Agent Platform [40]. A suite of tools facilitate the development, deployment and debugging of Jadex–based MAS. The individual agents consist of two parts. First, they are described by so–called *Agent Description Files* (ADF), which denote the structure of beliefs, goals and plans in XML syntax. Secondly, the activities agents can perform are coded in plans, these are ordinary Java-classes. Plan descriptions in the ADF (so–called *heads*) reference the compiled Java–classes (so–called *body*) and denote the conditions which may lead to plan instantiation (for details cf. [41]).

[6] http://www.fipa.org/
[7] http://vsis-ww.informatik.uni-hamburg.de/projects/jadex
[8] http://jade.tilab.com/

In addition to specifications of the basic BDI concepts, ADF also enforce the description of further implementation details. E. g. it is required that agents declare the messages – so–called *message events* – which can be sent inside of plans. The descriptions of these FIPA compliant messages comprise among other details the used *performative*, *protocol*, *ontology* and restrictions on the *content object* to be delivered.
Jadex plans are also allowed to dispatch so–called *internal events*, which may directly trigger plans in the same agent, forming an agent internal communication mechanism. While these events are declared in the ADF the plan descriptions denote the triggering events.

We developed a command line tool which searches all ADFs from a list of folders, and matches the declared messages against each other. The enabled message exchanges are listed in a detailed report and are compared to the declared messages in order to identify messages which can not be sent or received.

5.2 Checking Consistency Using Assertions

The implemented assertion mechanism executes arbitrary Java statements, that can be annotated to beliefs (including beliefsets), goals and plans in `assertion` tags in agent ADFs. The annotated statements will only be executed when the ADF comprises a references a capability named `jadex.assertion.Assert` (cf. figure 5), therefore allowing to turn assertion execution on and off by simple ADF modification. Assertion statements are expected to evaluate to `true`. When they are violated a detailed warning is generated, specifying the agent and the element where the assertion evaluated to `false`. Figure 6 exemplifies the usage

```
<capabilities>
    <!-- To enable execution of assertion statements. -->
    <capability name="assert" file="jadex.assertion.Assert"/>
    <!-- Include the df capability as dfcap for finding other agents
        and registering a sample service. -->
    <capability name="dfcap" file="jadex.planlib.DF"/>
</capabilities>
```

Fig. 5. Referencing the assertion capability.

of assertions. This code fragment is taken from the *sentry* agent of the marsworld example, which stores reported and found ore locations in a beliefset named *my_targets*. The shown code checks whether the sum of stored values does not exceed the amount of targets in the game environment (defined in a class `Environment`), which can be accessed from within the beliefs of the agent. This assertion mechanism has been implemented as a *crosscutting concern* in the Jadex system. This concept leads to a novel level of modularization in BDI implementations and is therefore briefly discussed.

```
<!-- The seen targets. -->
<beliefset name="my_targets" class="Target">
    <assertion>
        $agent.getBeliefbase().getBeliefSet("my_targets")
        .getFacts().length
            &lt;=
        ((Environment) $agent.getBeliefbase().getBelief("environment")
        .getFact()).cataracts().length
    </assertion>
</beliefset>
```

Fig. 6. An assertion statement added to a *beliefset* description in a jadex ADF. The statement checks the maximum amount of a target set.

Crosscutting Concerns in BDI–Agents In [12] so–called *capabilities* have been proposed to modularize BDI agents. These capabilities comprise beliefs, goals, plans and a set of visibility rules of these elements to the surrounding agent. In development of MAS, they are used to define specific functionalities which can be imported by different agent types.

Aiming towards automated assertion execution, we utilized an enhancement to this modularization concept, which allows to define *crosscutting concerns* in agent implementations. According to the *Separation of Concerns* [42, 43] the functionality of a software system can be decomposed into *core concerns*, which are to be separated into different components or modules and so–called *aspects* which crosscut them. Crosscutting prime examples are inter alia failure recovery and logging.

In this respect capabilities [25, 12, 44] intend to define and modularize core concerns in BDI agents. Agent types can share functionality by inclusion of the same capability. Similar to conventional development efforts — without the notion of aspects — non–functional concerns can be captured in modules and executed by explicit references to elements inside these modules. So–called *co–efficient capabilities* (CC) automate this referencing, by exploitation of the local reasoning mechanisms. We name these capabilities *co–efficient*, because they register for contributive processing on certain BDI reasoning events. While it is possible to to modify the surrounding agent, this mechanism allows crosscutting functionalities, like logging, failure recovery etc., to be automatically triggered, without explicit references in goals or plans. Details of their implementation, a discussion of similarities and differences to object–oriented aspects and their usage for minimum intrusive plan observation can be found in [16].

The above described `jadex.assert.Assert` capability implements a Listening Object to all events originated from belief access, goal or plan adoption. For all state changes of these elements this listening object looks up annotated assertion statements and executes them.

5.3 Internal Event Consistency

A typical declaration of an internal event is shown in figure 7. The event is declared under the name *internal_1* and triggers the execution of a plan called *SimpleExamplePlan3()*. Declared InternalEvents can be dispatched within plans

```
<plans>
    <plan name="simple_3">
        <body>new SimpleExamplePlan3()</body>
        <trigger>
            <internalevent ref="intern_1" />
        </trigger>
    </plan>
</plans>

<events>
    <internalevent name="intern_1" />
</events>
```

Fig. 7. An internal event called *intern_1* triggers the execution of the plan *SimpleExamplePlan3()*

via the Jadex API [41]. We implemented a capability (`jadex.iecheck.IECheck`) which checks on agent startup whether all declared events trigger plans. Despite internal events typically trigger plans in the presented way, it is also possible that plans handle these events directly by a blocking call of `waitForInternalEvent (String type, long timoeout)` or `waitForInternalEvent(String type)`. Therefore our implementation utilizes the novel annotation mechanism of Java 5.0.[9] Developers are expected to annotate the handled events to the plan classes using this meta–data facility (@HandlesInternalEventsString[] types).

5.4 MessageEvent Consistency

Figure 8 exemplifies the declaration of a message event. This message is taken from the sentry agent in the marsworld example. The used performative, transmition language and ontology are declared. In addition, the direction of the message need to be specified. While this example is declared to be sent (direction=`send`), possible values are also `receive` and `send\_receive` [41]. We developed a tool to examine the declared messages in a set of ADFs. A given list of folders will be searched for ADF files and he declared message events are iterated. Message properties and declared directions are compared in order to compute the possible message exchanges. These are reported together with orphaned message events, where no matching sender/receiver is specified.

[9] http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

```
<events>
    <!-- Call producer agent to mine ore at the given location. -->
    <messageevent name="request_producer" type="fipa" direction="send">
        <parameter name="performative" class="String" direction="fixed">
            <value>SFipa.REQUEST</value>
        </parameter>
        <parameter name="language" class="String" direction="fixed">
            <value>SFipa.JAVA_XML</value>
        </parameter>
        <parameter name="ontology" class="String" direction="fixed">
            <value>MarsOntology.ONTOLOGY_NAME</value>
        </parameter>
    </messageevent>
</events>
```

Fig. 8. Declaration of a message event in the *sentry* agent of the *marsworld* example. This message transmits a location to the *production* agent to order the mining of ore.

The found message matches are displayed in a graph structure as exemplified in figure 9. In these graphs agents (bigger, light) and message events (smaller, dark) are denoted as nodes. Messages are connected to the declaring agent via *aggregation* edges (following the well known UML[10] notation), while possible message exchanges are represented by blue arrows. Since all messages are displayed, i. e. are not structured in the protocols involved, we display the MAS in a three dimensional space to allow efficient layout. Graph representations are generated to be displayed with the *Wilmascope*[11] tool. This operating system independent graph visualization tool visualizes XML representations of graphs and allows users to set various rendering options to control a force directed layout [45]. Therefore users can move freely along the virtual graph and adjust the graph layout to their needs. Figure 9 (right hand side) displays the static communication structure of the marsworld example. *Carry* and *production* agents can report ore locations (`inform_target`) to the sentry agent. Sentry agents can order mining by the *production* agent via (`request_producer`) and *producer* agents in turn can order the transportation of ore by carry agents (`request_carry`). The left hand side shows the same MAS with a mistake in the declaration of the `request_producer` message. The force directed layout highlights that sentry and producer agents have no mean to communicate with each other and that two message events are orphaned.

6 Conclusions

In this paper we highlighted that the validation of the BDI–based reasoning process is a major challenge in testing and debugging BDI agents. We proposed assertions for this purpose, exemplified their usage and outlined a crosscutting

[10] http://www.omg.org
[11] http://wilmascope.org

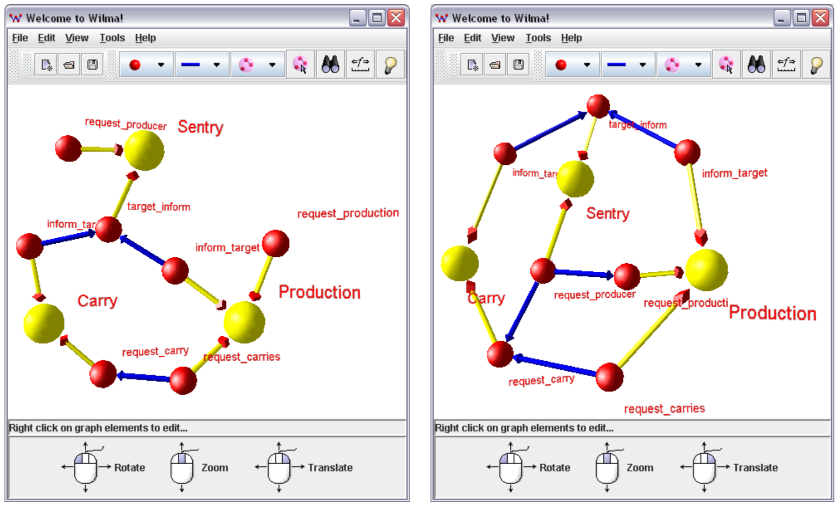

Fig. 9. The communication–network of the Marsworld example. Bigger vertices (light colored) denote Agents while the smaller ones (dark colored) represent MessageEvents in the agent ADFs. While the right hand side displays the intended communication structure, the force directed layout on the left side highlights that the *sentry* agent can not contact the *producer*.

implementation. In addition, static analysis of a set of agent declarations allowed to verify the consistency of internal events and message declarations. Finally, the overall communication structure of MAS have been visualized as three dimensional graphs.

While assertions have their origin in program verification [31, 32], they may also be used for formal proofs of correctness for proper goal and plan adoption. examination of common bugs and debugging strategies for BDI agents may inspire the methodic usage of assertions, i. e. a structured process to derive assertions from agent declarations, which has not been examined here. In addition it needs to be taken care that violations of these are in fact reflecting inconsistent agent states and the assertions have no side effects.

The presented visualization approach is preliminary and we plan to enhance display and user interaction in order to show the dynamic properties of message exchange and plan execution (cf. [16]).

References

1. Odell, J.: Objects and agents compared. Journal of Object Technology **1** (2002)
2. Franklin, S., Graesser, A.: Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In: Intelligent Agents III. Agent Theories, Architectures and Languages (ATAL'96). Volume 1193., Berlin, Germany, Springer-Verlag (1996)
3. Brooks, R.A.: Elephants don't play chess. Robotics and Autonomous Systems **6** (1990) 3–15

4. Bratman, M.: Intentions, Plans, and Practical Reason. Harvard Univ. Press. (1987)
5. Rao, A.S., Georgeff, M.P.: BDI-agents: from theory to practice. In: Proceedings of the First Intl. Conference on Multiagent Systems. (1995)
6. Georgeff, M.P., Lansky, A.L.: Reactive reasoning and planning: an experiment with a mobile robot. In: Proc. of AAAI 87, Seattle, Washington (1987) 677–682
7. Pokahr, A., Braubach, L., Lamersdorf, W.: A flexible bdi architecture supporting extensibility. In: The 2005 IEEE/WIC/ACM Int. Conf. on IAT-2005. (2005)
8. Jennings, N.R.: Building complex, distributed systems: the case for an agent-based approach. Comms. of the ACM **44 (4)** (2001) 35–41
9. Rao, A.S.: Agentspeak(l): Bdi agents speak out in a logical computable language. In: MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away, Springer-Verlag New York, Inc. (1996) 42–55
10. Braubach, L., Pokahr, A., Lamersdorf, W., Moldt, D.: Goal representation for bdi agent systems. In: Proc. of PROMAS'04. (2004)
11. Pokahr, A., Braubach, L., Lamersdorf, W.: A bdi architecture for goal deliberation. In: AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems, New York, NY, USA, ACM Press (2005) 1295–1296
12. Busetta, P., Howden, N., Rönnquist, R., Hodgson, A.: Structuring bdi agents in functional clusters. In: ATAL '99, Springer-Verlag (2000) 277–289
13. Meyer, B.: Object Oriented Software Construction. Prentice Hall (1997)
14. Menzies, T., Pecheur, C.: Verification and validation and artificial intelligence. In Zelkowitz, M., ed.: Advances in Computers. Volume 65., Elsevier (2005)
15. Timm, I.J., Scholz, T., Frstenau, H.: IV From Testing to Theorem Proving. In: Multiagent Systems Intelligent Applications and Flexible Solutions. to be published by Springer (2006)
16. Sudeikat, J., Renz, W.: Monitoring group behavior in goaldirected agents using coefficient plan observation. In: Submitted to the 7th International Workshop on Agent-Oriented Software Engineering (AOSE-2006). (2006)
17. Hailpern, B., Santhanam, P.: Software debugging, testing, and verification. IBM Systems Journal **41** (2002) 4–12
18. Liedekerke, M.H.V., Avouris, N.M.: Debugging multi-agent systems. Information and Software Technology Journal **37** (1995) 103–112
19. Ndumu, D.T., Nwana, H.S., Lee, L.C., Collis, J.C.: Visualising and debugging distributed multi-agent systems. In: Proc. of AGENTS '99. (1999) 326–333
20. Poutakidis, D., Padgham, L., Winikoff, M.: Debugging multi-agent systems using design artifacts: the case of interaction protocols. In: Proc. of AAMAS '02. (2002)
21. Poutakidis, D., Padgham, L., Winikoff, M.: An exploration of bugs and debugging in multi-agent systems. In: Proceedings of the 14th International Symposium on Methodologies for Intelligent Systems (ISMIS 2003). (2003)
22. Satyanarayanan, M., Steere, D.C., Kudo, M., Mashburn, H.: Transparent logging as a technique for debugging complex distributed systems. In: 5th European SIGOPS Workshop, on "Models and Paradigms for Distributed Systems Structuring", Mont Saint-Michel (France), IRISA, INRIA-Rennes (1992)
23. Flater, D.W.: Debugging agent interactions: a case study. In: Proceedings of the 2001 ACM Symposium on Applied Computing (SAC), ACM (2001) 107–114
24. Padgham, L., Winikoff, M., Poutakidis, D.: Adding debugging support to the prometheus methodology. Engin. Applications of Art. Intel. **18** (2005) 173–190
25. Braubach, L., Pokahr, A., Lamersdorf, W.: Extending the capability concept for flexible bdi agent modularization. In: Proc. of PROMAS-2005. (2005)

26. Lam, D.N., Barber, K.S.: Automated interpretation of agent behavior. In: Workshop for Agent-Oriented Information Systems (AOIS-2005). (2005)
27. Lam, D.N., Barber, K.S.: Comprehending agent software. In: Proc. of the 4th int. joint conf. on autonomous agents and multiagent systems (AAMAS '05). (2005)
28. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. Number ISBN 0-470-86120-7. John Wiley and Sons (2004)
29. Low, C.K., Chen, T.Y., Rönnquist, R.: Automated test case generation for bdi agents. Autonomous Agents and Multi-Agent Systems **2** (1999) 311–332
30. Hoare, C.A.R.: Assertions: a personal perspective. Software pioneers: contributions to software engineering (2002) 356–366
31. Floyd, R.: Assigning meaning to programs. Mathematical Aspects of Computer Science **XIX American Mathematical Society** (1967) 19–32
32. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12(10)** (1969) 576–580,583
33. Turing, A.M.: Checking a large routine. In: Report on a Conference on High Speed Automatic Calculating Machines, Cambridge University Mathematical Lab. (1949)
34. Voas, J.: How assertions can be increase test effectiveness. IEEE Software **March/April** (1997) 118–122
35. Busetta, P., R/"onnquist, R., Hodgson, A., Lucas, A.: Jack - intelligent agents – components for intelligent agents in java. Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia (1998) http://www.agent-software.com.
36. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A bdi reasoning engine. In R. Bordini, M. Dastani, J.D., Seghrouchni, A.E.F., eds.: Multi-Agent Programming, Springer Science+Business Media Inc., USA (2005) 149–174 Book chapter.
37. Ferber, J.: Multi-Agent Systems. Addison Wesley (1999)
38. Braubach, L., Pokahr, A., Lamersdorf, W.: Jadex: A short overview. In: Main Conference Net.ObjectDays 2004. (2004) 195–207
39. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: Implementing a bdi-infrastructure for jade agents. EXP - in search of innovation (Special Issue on JADE) **3** (2003) 76–85
40. Bellifemine, F., Rimassa, G., Poggi, A.: Jade a fipa-compliant agent framework. In: In 4th International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-99). (1999)
41. Pokahr, A., Braubach, L., Walczak, A.: Jadex User Guide. Distributed Systems Group University Hamburg. 0.941 edn. (2005)
42. Dijkstra, E.: A Discipline of Programming. Prentice-Hall (1976)
43. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15** (1972) 1053–1058
44. Padgham, L., Lambrix, P.: Agent capabilities: Extending bdi theory. In: Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence. (2000) 68–73
45. Dwyer, T.: 3d uml using force directed layout. In: Proceedings of the Australian Symposium on Information Visualization. (2001) 77–85