Self-describing Agents

Günther Görz, Bernd Ludwig, Peter Reiß, Bernhard Schiemann, Tobias Seutter

Department of Computer Science 8
University Erlangen-Nuremberg
Haberstraße 2
91058 Erlangen
{goerz, bernd.ludwig, reiss, schiemann}@informatik.uni-erlangen.de
tobias.seutter@stud.informatik.uni-erlangen.de

Abstract: Multi-Agent systems follow a highly abstract programming paradigm. Because of this abstraction level, behaviours of the participants are not always clearly reproducible for humans. In this paper, we present a framework that enables agents to do some introspection and describe themselves. Our approach uses Java annotations and JDI, the Java debugging architecture. Programmers enrich the source code with comments for the user; during runtime, explanations are given and inform about what is going on “inside” the agent or whole system. We adopt our framework to the A4 MAS for supply chain management.

1 Introduction

Various programming paradigms have been developed that help software engineers build their programs, but the complexity of today’s software systems makes it impossible for the end user to know about all of the implemented features. A Multi-Agent System (MAS), especially implementing a business system, has the advantage that domain concepts can be transferred into corresponding structures and agents of the MAS. That is why a lot of business system applications use MASs for a wide range of problems, e.g. decision support [Phi05]. The structure of an MAS may, for example, reflect the company structure and/or the entities that collaborate in managing some business project. [Zim06] shows an implementation of this strategy in Supply Chain Event Management (SCEM). User interfaces of MASs for business systems are mainly used to show the results of some – perhaps complex – computation or collaboration of agents. Human experts of the respective fields may intuitively understand what the MAS and the respective agents are doing. Giving additional (background-)information to the user usually is beyond the scope of MAS developers, but is being addressed in current research. E.g., [IH+06] present the ethnographic analysis driven way of building MAS that combines users capabilities and agents. In this paper, we argue for an approach that enriches agents and whole MASs with user oriented information. Our objective is to give the user the possibility to become familiar with such complex and abstract system.

In [RG07], we developed a framework which we call "Java Platform Annotation Processing Architecture (JPAPA)". It enriches software with some self-explaining capability. In the following, we describe the framework and its application to MASs. Although it does not depend on a specific target application (its only limitation is that the application has to be written in Java), we see high potential in using it with agents or entire MASs. We want to give assistance for human users of the MASs in different ways:

- Agents give feedback about their capabilities. They explain what they are made for and how they perform the work they are doing.
- Communication between agents is explained. Users will be informed if messages are sent to other agents and why.
- It is possible to configure filters for explanations. Users can decide which part of the agent software should give additional information.
- Traditional user manuals have several drawbacks: They are often voluminous and it is sometimes hard to find the information about specific tasks, especially tasks related to the current state of the system. In contrast, our system presents the required information just in time. We avoid information overload; information that the user does not need right now is not presented and users are not forced to read long and incomprehensive explanations in user manuals.

Target users of our self-explaining agent system might be

- Domain experts who not only want to see the system running, but also want to know which steps it is actually performing.
- Researchers who want to know something about the application, because they are part of the development team or want to use (some parts of) the agent software in another application.
- People who are testing the program. They are given the ability to collect additional information to give concrete feedback to the programmers.
- "Normal" end users, who not only want to use the software as a black box, but want to know how the system is organized.

Of course, the content of the explanations must be adopted to the intended target user group. JPAPA allows for classifying the documentation and filtering according to the recipient.

This paper is organized as follows: The following two sections describe the historical background of self-explaining systems and how we define self-describing software for user assistance. Section 4 informs about the implementation of JPAPA, which is our general framework for self-explaining software (for the Java programming language). Section 5 connects this framework to AOSE (Agent Oriented Software Engineering) and gives an example of an MAS for supply chain management annotated with JPAPA. In the last section we give a summary.

2 Related Work

Smith [Smi82] was among the first to point out the need for reflective capabilities in programming languages. He developed a LISP system (3-LISP) that incorporated the so called *reflective tower*: programs are executed by an interpreter which itself is a program written in 3-LISP. The current status of the interpreter is visible from "one level above". Thus, the meta-interpreter observes the control flow and gives information about it. Maes [Mae87, MN88] used a similar strategy for the object oriented programming paradigm. Sosic [Sos92] developed a compiler for the C/C++ programming languages. His system also provided introspective and reflective capabilities. Modern programming languages like Java, Python or C# have built-in APIs for reflection with the ability to examine or modify runtime behaviour of applications. The main intent of these systems is to provide mechanisms for debugging purposes, to enable self-optimisation and to generate statistics of the program flow. There is some relationship between our approach and the CLASSIC system [BB+89], where some of the concepts could be marked as "relevant for explanation". If such a concept was used during the execution, the explanation was displayed. But CLASSIC is a representation system for objects, not a programming language. So, to the best of our knowledge, it is a novel approach to use reflective features of a programming language for user assistance.

There are other mechanisms that can be used for additional runtime-information to the end-user. The log4j tool allows for collecting debugging output, filtering and presenting it. One might also imagine a system, where javadoc comments are "abused" for user information, e.g. extracted to a file and connected to the code at runtime. We consider the advantage of our approach in the fact that the user is able to control the program flow (cf. section 4.4). Not until she has fully processed the output, she initiates the next step. The other approaches just mentioned would either result in some kind of log file, and there is no more connection to the system state when reading the entries offline. On the other hand, if the messages were displayed at runtime without being able to suspend the VM, the entries would potentially pass too fast to be processed by the user.

3 Self-describing software for user assistance

Applications are written by programmers for end users to help them to accomplish some task. Nowadays, software systems have become extremely complex, and users can hardly know about all of the features a program might provide. Facing this problem, our main idea is that software should be able to give information about itself – what features are implemented, how the user can trigger some task etc. Using our framework, the programmer puts this information into the source code. The programmer annotates those parts of the software that are important for the end user to understand the program. The way we use them, annotations are somewhat similar to comments in source code – the difference is in the intended recipient of the annotation content: it is not another programmer, but the end user. Another difference is the fact that source code comments are skipped in a pre-compiling step, whereas the content of annotations is visible even at runtime. To the end user, programming details are of no interest. But there are key parts of the software that do something relevant with respect to the business task. We annotate this code and inform the user about what kind of job the software is performing in which manner right now. Used the right way, this makes software comprehensible. As a simple example, a user observing an agent might be wondering why it seems to be stalled. If the user had the information that the agent is waiting for a message of another agent, this would explain agents state and behaviour.

Often, programmers and users have different concepts in mind about the entities and mechanisms required for a certain task. Because of this, putting the right end-user comments at the right parts of the code might be somewhat tricky. As a guideline, we suggest the following points:

- Actions that are related to user interactions, like mouse clicks, are important to explain. Here, we give information about how the program uses the input and processes the data.

- Parts of the software that may take some time to be processed. Because our framework monitors the program flow, we are able to tell the user what is going on “inside” the system.

- In addition, some background information should be presented. Especially parts of the software that match user’s concepts of the task to be done should be annotated.

To summarize, we want to enable software systems to establish some understanding about its functionality and to make the behaviour of such systems become more plausible. Giving additional information about *how* the software performs its work is a task somewhat orthogonal to standard man-machine-communication. All software that is written for the JPAPA framework runs perfectly in a standard Java VM. Starting the software in the JPAPA environment makes it perform its business logic (as it would usually do) with normal user interactions, but additionally the assistance task is performed.

4 Implementation

In contrast to offline help systems, we use reflective capabilities of Java in combination with an adapted debugging mode. This makes it possible to inform the user on time. If the virtual machine (VM) encounters code that was enriched with end-user explanations, a kind of exception is thrown and the information will be presented.

4.1 Java Platform Annotation Processing Architecture (JPAPA)

Our framework is entirely written in the Java programming language and provides mechanisms to explore applications written in Java. The only requirement an application has to fulfil is that it has to be annotated with special Java annotations. The basic principle behind JPAPA is as follows:

- The program's author documents pivotal program elements via Java annotations in the source code.

- During runtime, in addition to the normal control flow, the application is controlled by mechanisms provided by the framework.

- As soon as the byte code of an annotated program element is to be executed, normal execution is suspended and the user is informed about the annotated element. JPAPA presents additional information, the content of which are put from the annotation itself via reflection mechanisms.

The core framework is made up of three components that will be described in detail in the following.

4.2 Java Annotations

Annotations allow developers to supply additional information to their programs. Such metadata can be added to various program elements and during compile time be used to generate additional code or during run time provide additional semantic information. Annotations have no direct impact on the execution of a program. Consequently, concerning the business task, executing the annotated programs with JPAPA will result in the same outcome as without of JPAPA, because the VM that is suspended resumes with unchanged internal state. Of course, one has to keep in mind that this may cause some time delay. Especially in agent oriented programming, we do not consider this to be a severe drawback, because agents should never rely on "real time" answers of other agents. In commonly used agent platforms like FIPA OS or JADE, the type of communication between agents makes it impossible to guarantee time durations anyway. In the context of JPAPA, annotations are used to give the end users information about elements and functionality of the program. Java allows for adding annotations to the following language elements: types (classes and interfaces), packages, fields, methods, constructors and parameters.

The annotation `Doku` used in JPAPA is composed of four fields that can be put together in any order:

```
public @interface Doku {
        String id();                    // unique name
        String version() default "N/A"; // version number
        String doku() default "N/A"; // primary information
        String description() ;//additional information
}
```

The following source code fragment exemplifies an annotated method:

```
@Doku(id="setup()",
        doku="The setup method of the engager agent takes
        care of all important default settings and
        initializes the knowledge base of the engager
        agent. Especially the company the engager agent
        works for is set.")
protected void setup() { ...}
```

4.3 Java Reflection

When the framework during runtime detects a class that contains annotated elements, it obtains the contents of the annotation via the standard Java Reflection API as follows. First the class of an object is queried. By means of this object all required metadata are retrieved. Besides information about the class itself, the implemented interfaces and the superior package, details about constructors, methods and fields together with their respective annotations can be polled. As an example, the following source code fragment shows how the content of a *`Doku`* annotation can be retrieved:

```
if(field.isAnnotationPresent(Doku.class){
        Annotation doc = field.getAnnotation(Doku.class);
        String doku = doc.doku();
        …}
```

4.4 Java Debug Interface

To enable the user to control the execution of the application, JPAPA uses control mechanisms provided by the Java Debug Interface (JDI), which is part of the Java Platform Debug Architecture (JPDA). Similar to a debugger, the executed program is encapsulated in a second Java Virtual Machine that is fully controlled by our framework. Once a class is loaded into the monitored VM, it is inspected via the Reflection API. Every annotated element will be marked with a certain stop request, which causes the monitored VM to suspend execution whenever the corresponding byte code will be executed. When the framework receives a stop request, the user is informed about the causing program element and its annotation. Execution continues as soon as the user resumes the monitored VM.

Figure 1 shows how JPAPA parses the classes and controls the target program.

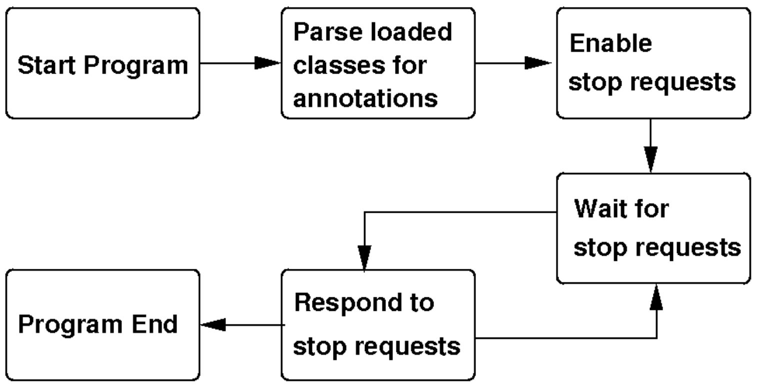


Figure 1: program flow of JPAPA

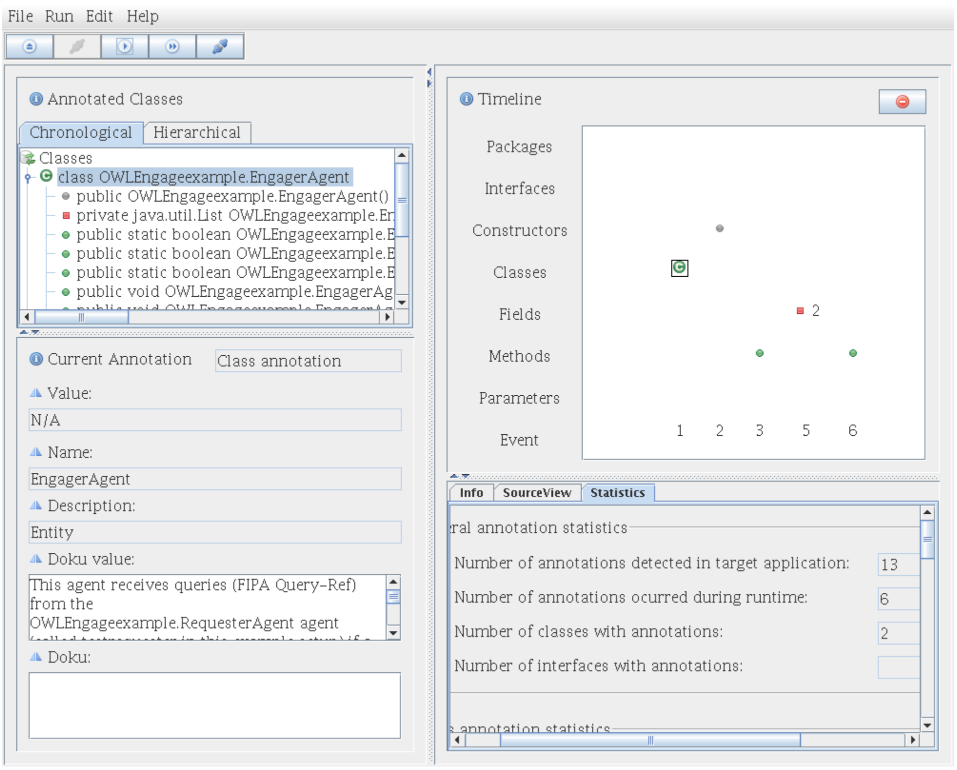
On top of the core API, a client with a graphical user interface (cf. figure 2) has been attached that extends the framework with additional features:

- On the basis of a chart the user can trace what annotations occurred in connection with what values. It is possible to browse the history of events that produced the presentation of documentations. Here, we also give information about which element of the code was annotated. We use the categorisation of the Java programming language and differentiate between annotaed classes, fields, methods, etc. We consider to implement a lightweight view with only one type of event, because not all of the users may understand the differences between the categories that are motivated by the programming language.

- In addition to this visual representation the user can activate an audio representation mode, in which the user will be read to the contents of the annotation.

- An implemented filter enables the user to include certain annotations in the presentation and respectively exclude others. It is possible to filter classes of annotations and single instances. E.g. the user is able to specify that she is not interested in the documentation of a certain method of a class.

- If there are annotations with different target user classes, it is possible to select the type of information one is interested in. For instance, if annotations for novel and advanced users are in the source code, one can choose between these two categories.



Classes
class OWLEngageexample.EngagerAgent
public OWLEngageexample.EngagerAgent()
private java.util.List OWLEngageexample.En
public static boolean OWLEngageexample.E
public static boolean OWLEngageexample.E
public static boolean OWLEngageexample.E
public void OWLEngageexample.EngagerAg

Current Annotation: Class annotation

Value:
N/A

Name:
EngagerAgent

Description:
Entity

Doku value:
This agent receives queries (FIPA Query-Ref) from the OWLEngageexample.RequesterAgent agent

Doku:

ral annotation statistics

```
Number of annotations detected in target application:   13
Number of annotations ocurred during runtime:            6
Number of classes with annotations:                      2
Number of interfaces with annotations:
```

Figure 2: Screenshot of the user interface of JPAPA

5. Self-describing agents in business applications

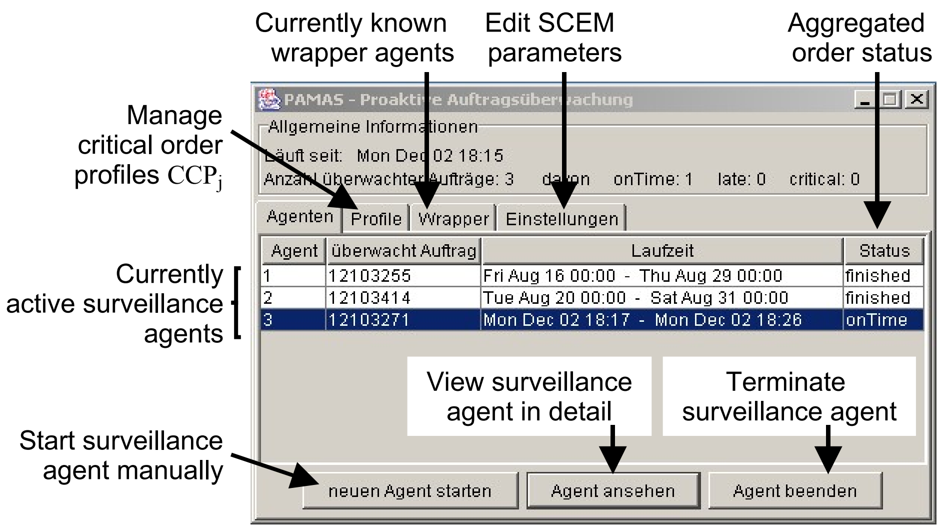
A closer look at figure 2 shows that the example running in JPAPA is already a (simple) MAS. The annotated EngagerAgent is a reimplementation of the well known JADE[1] EngagerRequester example, now built for the JADEOWLCodec[2]. The EngagerAgent manages the employees of a company. In the field "Doku value" (lower left on figure 2), the content of the annotation of the class is displayed. Users of this scenario might be computer scientists trying to learn what the differences between the original JADE example and this ontology driven agent are. Annotations for methods of the agent's behaviours (cf. figure 2 upper right field, row "Methods") would display actual communication issues or the computation details.

[1] http://jade.tilab.com
[2] Downloadable at http://www8.informatik.uni-erlangen.de/en/demosdownloads.html#jadeowlcodec

Now, we transfer this procedure to the A4 MAS described in more detail by [ZW+06]. Agents in this MAS are modelled in companies implementing mainly three different roles:

- Discourse agents provide an iterface to external supply chain partners.
- Coordination agents, which coordinate initialization of monitoring processes and distribute their results. For each enterprise in the supply chain, a single coordination agent assures the initialization of monitoring efforts as well as the management of external status requests in a consistent manner.
- Surveillance agents are responsible for creating and aggregating information about an order by gathering and interpreting SCEM data. For each monitored order of an enterprise a dedicated surveillance agent is triggered by the coordination agent.

Every enterprise modelled in the supply chain is represented by at least one agent out of these agent types. We built this prototype by applying the agent oriented software engineering process introduced by e.g. [WJ06]. Because the self-describing capabilities were not yet developed the time we built up A4, we had to: 1) debug this MAS with traditional methods and 2) design graphical user interfaces (GUI) for the domain experts.


Currently known wrapper agents
Edit SCEM parameters
Aggregated order status
Manage critical order profiles CCP_j
Currently active surveillance agents
View surveillance agent in detail
Terminate surveillance agent
Start surveillance agent manually
PAMAS - Proaktive Auftragsüberwachung
Agenten
Profile
Wrapper
Einstellungen
neuen Agent starten
Agent ansehen
Agent beenden

Allgemeine Informationen

Läuft seit: Mon Dec 02 18:15

Anzahl überwachter Aufträge: 3 davon onTime: 1 late: 0 critical: 0

Agent	überwacht Auftrag	Laufzeit	Status
1	12103255	Fri Aug 16 00:00 - Thu Aug 29 00:00	finished
2	12103414	Tue Aug 20 00:00 - Sat Aug 31 00:00	finished
3	12103271	Mon Dec 02 18:17 - Mon Dec 02 18:26	onTime

Figure 3: Agent GUI from A4

See figure 3 [ZW+06] as an example for such an agent GUI. Business experts normally are not experts in the field of MAS and software systems. That is why it was necessary to develop graphical interfaces to let them get an overview of the running system. A self-describing system has many advantages compared to traditional interfaces:

1. GUIs (and other kinds of hard coded user information) have to be a static part of the implementation; so, every time the software changes, programmers also have to consider adjusting this interface. In contrast, the whole information system of JPAPA is built up at runtime.

2. Without already knowing the number of agents that should inform a user about the state of some computation, a GUI (as an element programmed at design time) is not flexible enough for MASs. The JPAPA framework activates the self-describing capabilities of all agents running at the encapsulated JADE platform at one time. If a single agent or the complexity of its computation provides too much information, the user is able to filter information with the filtering method the JPAPA GUI provides (see Figure 2).

3. JPAPA is configurable; explanations can be filtered at runtime. So, a domain expert interested in a certain aspect of the business logic or in agent communication can easily filter out unnecessary information. Consider the following situation: An employee is told that agent A is responsible to observe a certain step in the supply chain. Perhaps the user wants to know something about what agent A is actually doing to get necessary information, to get into context, to evaluate the data and to compute the decision whether the task is in correct state. In this case, the user enables the self-explaining mode of agent A to get an impression about what is happening behind the scenes.

6. Summary

We presented our approach for user assistance which can be applied to any Java application and adapted it for MAS. We developed a combination of explanation-enriched agent source code and special runtime environment. Using Java annotations, reflection capabilities and debugging tools, we have complete control over program flow. The user gets explanations whenever annotated code is executed; it is possible to browse the history of explanations. We adopted our tool to agents and MASs. Especially for MASs in business systems, we consider that explanations about the internals of the software are extremely useful. Domain experts get the knowledge to reproduce the system behaviour and to understand the application.

Bibliography

[BB+89] Borgida, A.; Brachman, R.; McGuinness, D.; Alperin Resnick, L.: CLASSIC: A structural data model for objects. In: Clifford, J.: Proceedings SIGMOD International Conference on Management of Data, pages 58-67, 1989.

[IS+06] Iqbal, R., Shah, N. H.; Chao, K.; James, A.: A User-Centred Design Approach For Agent Based E-Business Systems icebe, pages 264-270, 2006.

[Mae87] Maes, P.: Concepts and experiments in computational reflection. In *Proc. OOPSLA*, pages 147-155, ACM, 1987.

[MN88] Maes, P.; Nardi, D.: Meta-Level Architectures and Reflection. Amsterdam, 1988.

[Phi05] Phillips-Wren, G.; Jain, L.C.: Intelligent Decision Support Systems in Agent-Mediated Environments In: Volume 115 Frontiers in Artificial Intelligence and Applications, pages 224ff, 2005.

[RG07] Reiß, P.; Görz, G.: SIPaDIM - Assistenz durch selbstbeschreibende Software. In: Koschke, R.; Herzog, O.; Rödiger, K.; Ronthaler, M. (eds.) *Lecture Notes in Informatics P-109,* pages 279-282, Bonn, 2007.

[Smi82] Smith, B. C.: Procedural Reflection in programming Languages. PhD Thesis. Massachusetts Institute of Technology, 1982.

[Sos92] Sosic, R.: The many faces of introspection. PhD thesis. The university of Utah, 1992.

[WJ06] Weiß, G.; Jakob R.: Agentenorientierte Softwareentwicklung: Methoden und Tools (Xpert.press). Springer-Verlag New York, Inc., 2006.

[Zim06] Zimmermann, R.: Agent-based supply network event management. Birkhäuser, Basel 2006.

[ZW+06] Zimmermann, R.; Winkler, S.; Bodendorf, F.: Supply Chain Event Management With Software Agents. In: Lockemann et al. (eds.) *Multiagent Engineering*, pages 157-176, Berlin, 2006