Sam: a Tool to Ease the Development of Intelligent Agents

João Faccin, Juei Weng, and Ingrid Nunes

[1]Universidade Federal do Rio Grande do Sul – Porto Alegre – Brazil

{jgfaccin,juei.weng,ingridnunes}@inf.ufrgs.br

Abstract. *Developing intelligent agents is a difficult task. The BDI architecture provides agents with flexible behaviour and our previous work extended this architecture by using learning to improve plan selection. In this paper, we present Sam, a tool that supports BDI agent design and implementation, using a model-driven approach. Sam provides an environment where agents can be modelled according to our proposed meta-model, in which developers focus on domain-specific concepts. Our tool, based on a design model, also generates agent code focusing on the BDI4JADE platform. Our approach hides from developers sophisticated techniques, so that mainstream software developers are able to leverage such techniques without having to learn their technical details. Video:* `https://youtu.be/hkftWZx82EM`

1. Introduction

Software has shifted from standalone applications to distributed and highly interactive systems, evidenced by cloud computing and smart grids. Such complex systems consist of a composition of autonomous software components, with proactive and reactive behaviour, and social ability. In the context of multi-agent systems [Wooldridge 1999], such components are referred to as *agents*, and much work has been done to address the problems that emerge in this scenario. For example, agents must be coordinated, learn in which other agents they can trust, and adapt themselves according to its context. Agent architectures have been proposed to support agent development. One of the most widely used architectures is inspired by human practical reasoning, namely the BDI (*belief-desire-intention*) architecture [Rao and Georgeff 1995]. In this architecture, goals are explicitly represented, and agents have a reasoning cycle that includes the selection of appropriate actions (plans) to achieve goals. Consequently, agent behaviour is flexible in that it can select plans that are suitable to the current context or execute other plans in case of failure.

The BDI architecture is abstract, and there are gaps that must be fulfilled to provide agents with intelligent behaviour. Our previous work [Faccin and Nunes 2015, Faccin 2016] proposed a model-driven approach, which fills one of these gaps with the provision of a learning-based plan selection technique. The key goal was to provide means for developing BDI agents able to learn in which context plans perform best, thus making a wiser plan selection. In this work, we present the Sam tool, which supports the use of our technique by providing an environment to model BDI agents, focusing on domain aspects, and generate agent code. Consequently, we free developers from learning complex learning models or details of the BDI reasoning cycle.

2. Background on BDI Agents and Learning-based Plan Selection

BDI agents are composed of three key components. The first are *beliefs*, which represent its current state. They are referred to as beliefs, rather than attributes, because they may not be accurate, e.g. due to environment perceptions with noise. The second are desires, or *goals*, which represent something the agent wants to achieve. The third are intentions, which are goals that an agent is committed to achieve and already has a *plan* to execute to achieve it. Its reasoning cycle, which is provided by BDI platforms, is the following: (1) beliefs are revised according to perceived events; (2) goals are updated according to current beliefs (goals may have been achieved or no longer desired, or new goals are created); (3) a subset of goals are selected to be achieved (goals may conflict with each other, so an agent may select a subset of them); and (4) a plan from a set of candidates is selected to be executed to achieve each intention. Note that if the selected plan fails, the agent still has the goal, so other options are explored to achieve it. Step 4 is referred to as *plan selection*, and its default implementation randomly selects a plan from those whose context condition matches the current context.

Our plan selection approach is composed of a *meta-model* and a *technique*. The meta-model specifies concepts to model agent preferences, softgoals and plan metadata. Making an analogy to software development, *softgoals* can be seen as non-functional requirements. A simple example is an agent that has the goal of sorting an array, and softgoals indicate other properties to be satisfied, such as time taken and used memory. Plan metadata consist of information to understand which *factors* influence plan *outcomes*, e.g. the number of added items since the last time the array was sorted influences the time taken by a sorting algorithm (each is a plan). Finally, preferences allow specification of softgoal importance to an agent. The technique, based on this information, builds a prediction model based on observations of plan executions. Using this model, it predicts plan outcomes based on the current context. Finally, it calculates plan utility considering agent preferences and estimated outcomes. The plan with the highest utility is selected.

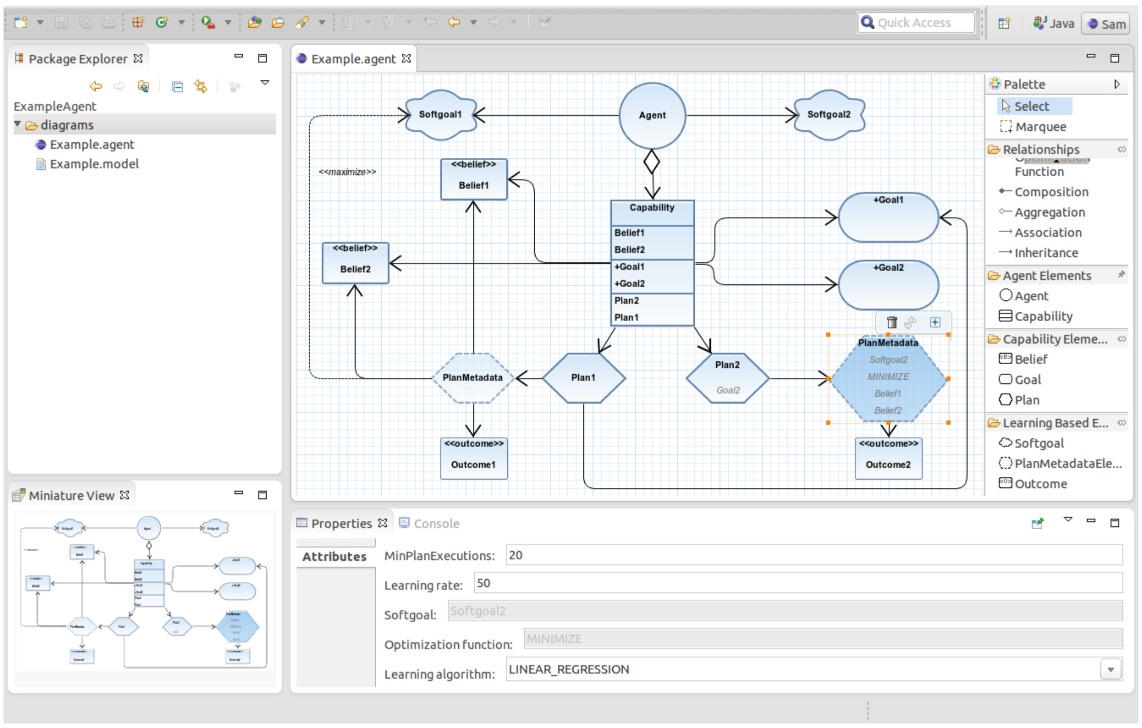
3. The *Sam* Tool

Sam is a development environment that includes a graphical editor and code generator developed focusing on the design and implementation activities of the agent development process. Implemented as an Eclipse[1] plug-in, it allows users to graphically instantiate agent models, also providing features for automatic validation and code generation of these models. In this section, we detail Sam, presenting its main features, the different elements that comprise its user interface and, finally, its architecture.

3.1. Features and User Interface

Our tool provides two key functionalities. The first provides assistance for users to design and model software agents. Therefore, Sam includes a modelling editor to design agents, which is presented in Figure 1. The main element of the Sam user interface is the editor view, which is responsible for displaying the graphical representation of a modelled agent. The information related to a modelled element is stored in diagram and model files, related to graphical and domain-specific information, respectively. Users can navigate these files through the package explorer, located on the left-hand side of the editor view. Below the package explorer, an additional view presents a miniature of the model being presented by the editor view, thus assisting users in visualising large model representations. On the right-hand side of the editor view, there is a creation palette, which contains the elements that can be instantiated in a diagram. These elements, when instantiated, can have their properties changed through a property view, positioned below the

[1]`http://www.eclipse.org/`



MinPlanExecutions: 20

Learning rate: 50

Softgoal: Softgoal2

Optimization function: MINIMIZE

Learning algorithm: LINEAR_REGRESSION

Figure 1. Sam Overview: the Modelling Editor.

editor view. This property view can also be replaced by a console view, which is used to provide specific feedback to users.

The second key functionality supports agent implementation, by generating code according to an instantiated agent model. This code generation feature is automated, and it requires users only to request to generate code. Note that generated code is not limited to simple code skeletons, but includes code to reason about plans. Next, we detail these modelling and code generation features.

3.1.1. Agent Modelling

The modelling feature that Sam provides allows the graphical instantiation of agents following our introduced meta-model. This meta-model defines which elements comprise an agent, and how these elements are related to each other. Our current implementation of the meta-model also includes other extensions of the BDI architecture [Nunes 2014].

In Sam, instantiating a model involves a few steps. Initially, users must create a new diagram file using a specific creation wizard that we provide. Thus, they are able to model a new agent by dragging elements from the creation palette and dropping them into the diagram area presented by the editor view. Graphical representations of the elements composing our meta-model are described as follows. An agent is represented by a circle containing the agent name. Agents, in our meta-model, are an aggregation of capabilities, which are components that modularise a set of beliefs, goals and plans. Capabilities are represented by rectangles with four compartments, which contain its name, and the name of its beliefs, goals, and plans, respectively. A belief is depicted as a rectangle containing a stereotype ≪`belief`≫ and the belief name. Goals are depicted as rounded rectangles while softgoals are represented by cloud-like shapes, both containing the respective element name. Plans and plan metadata elements are represented by hexagons with solid and dashed lines, respectively, also containing the element name. Such plan metadata elements are related to outcomes whose representation is similar to that of beliefs, but with the stereotype ≪`outcome`≫.

Relationships between elements also have their particular representations. Association, aggregation, composition and inheritance relationships are illustrated in the same manner they are in UML class diagrams—solid directional lines with particular arrowheads connecting two elements. Optimisation functions, in turn, are presented as dashed directional lines labelled with the name of the given optimisation function. These functions are used to convert plan outcomes into preference values. A preview of the graphical representation of each available element or relationship is presented in the creation palette. Moreover, it is important to highlight that, to maintain consistency with existing agent methodologies, some representations were imported from those used by Tropos [Bresciani et al. 2004].

Sam provides several usability features, aiming to allow users to create understandable models while improving their experience using our tool. When modelling an agent that aggregates several capabilities, users can create a separate diagram file for each of them. Such capability diagrams can be associated with a particular agent diagram file. Given that our tool is able to identify every component belonging to the same model file, every change performed in a capability diagram reflects on its associated agent diagram. Therefore, users are able to maintain a simplified agent diagram containing only representations of an agent, its capabilities, and their relationships. The representation of capabilities and their elements are, in turn, maintained in their respective capability diagrams. Additionally, users can navigate these related agent and capability diagrams using the so-called *drill-down* feature. This feature simply opens the diagram to which a capability is related when an *Open associated diagram* option is selected in the capability's context menu. Moreover, it is bidirectional, i.e. it is possible to go from an agent to a capability diagram, and from a capability to its agent diagram. Figure 2 illustrates the Sam drill-down feature.

Another feature that contributes to simplifying the visualisation of larger model representations is the *collapse* feature. It allows users to hide/show graphical representations of particular relationships whose source is a plan or plan metadata element. When hidden, the target name of a given relationship is presented inside the source element. When shown, the target name disappears from the source element representation and the relationship shape becomes visible again. Such feature can be activated using the collapse context button, which is shown when the mouse is positioned on a plan or plan metadata element. An example of the collapse feature activated can be observed in Figure 1. Finally, we also provide the validation of relationships between elements, e.g. if a plan is already associated with a goal, the user becomes unable to insert a new relationship with a different goal unless the previous one is removed.

Agent models are stored in two kinds of files: (i) the diagram file, containing details of the graphical representation of the agent model, and (ii) the model file, which stores the model itself. While the former is used to correctly display the model, the latter becomes the input to the code generation process, detailed next.

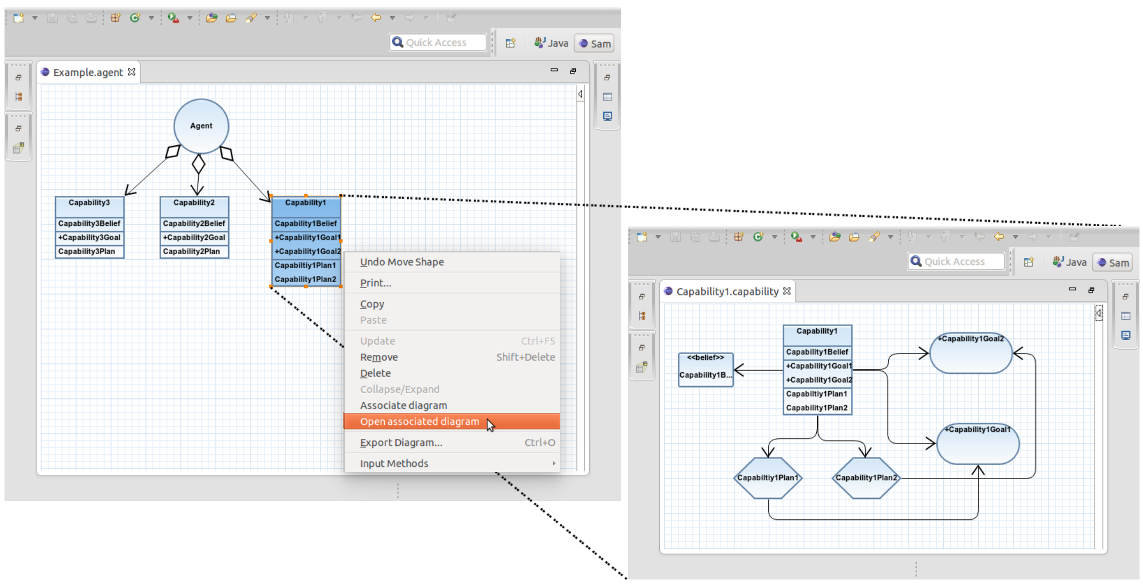
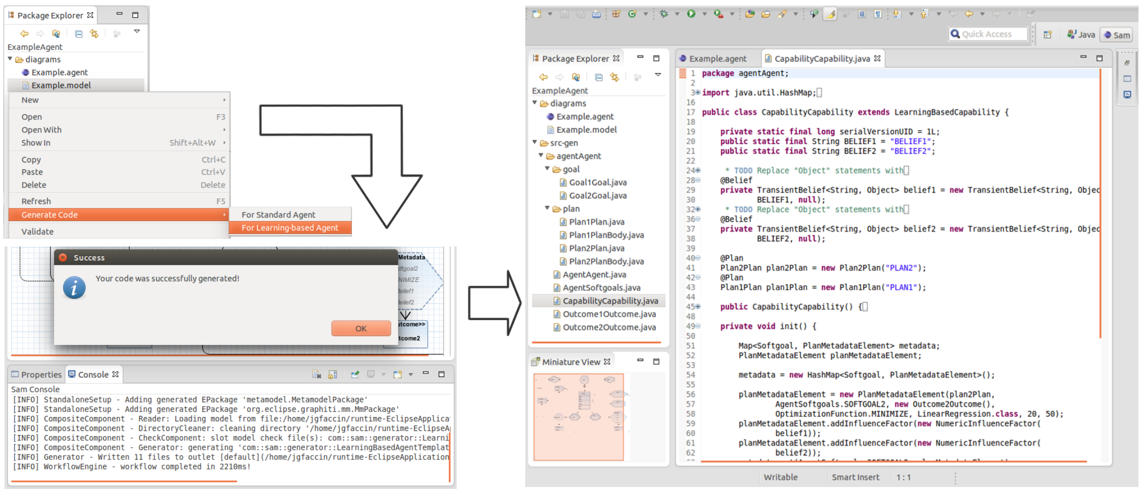


Figure 2. Navigating through Diagrams with the *Open associated diagram* option.

3.1.2. Code Generation

The code generation feature provided by Sam is responsible for creating code from agent models. By taking a model file as input, our tool is able to automatically generate BDI4JADE [Nunes et al. 2011] code corresponding to the agent represented in such model, as well as its entire structure of packages and classes, and required libraries. BDI4JADE is a BDI platform implemented in Java, which was selected because it is concerned with the industrial adoption of the agent technology. Before starting the code generation process, Sam checks the model looking for inconsistencies. Any inconsistency found is reported to users through the console view. There are two classes of model inconsistencies: (i) *regular*, which does not affect code generation; and (ii) *severe*, which immediately interrupts the code generation process. Regular inconsistencies are related to missing information that can be added directly to the code later, while severe inconsistencies refer to missing relationships or elements that are required to be instantiated on the given model, otherwise the generated code would be semantically wrong.

Users can trigger the code generation feature by selecting the *Generate Code* option in the context menu of a model file. Thus, Sam requires users to select one of two available agent profiles: *standard agent* or *learning-based agent*. Each profile corresponds to particular model validation and code generation approaches. The *standard agent* profile refers to typical BDI agents and those implementing capabilities relationships [Nunes 2014], while the *learning-based agent* profile indicates to Sam that it must check and generate code of agents that implement our learning-based approach. Figure 3 illustrates Sam's code generation feature. It is important to highlight that Sam only generates the code it can infer from a model file, e.g. correctly initialising elements and combining them with code that implements our learning-based technique. Therefore, developers should complete some of the agent components, mainly plan bodies, with the domain-specific code that cannot be generated automatically.



Your code was successfully generated!

```
Sam Console
[INFO] StandaloneSetup - Adding generated EPackage 'metamodel.MetamodelPackage'
[INFO] StandaloneSetup - Adding generated EPackage 'org.eclipse.graphiti.mm.MmPackage'
[INFO] CompositeComponent - Reader: Loading model from file:/home/jgfaccin/runtime-EclipseApplica
[INFO] CompositeComponent - DirectoryCleaner: cleaning directory '/home/jgfaccin/runtime-EclipseA
[INFO] CompositeComponent - CheckComponent: slot model check file(s): com::sam::generator::Learni
[INFO] CompositeComponent - Generator: generating 'com::sam::generator::LearningBasedAgentTemplat
[INFO] Generator - Written 11 files to outlet [default](/home/jgfaccin/runtime-EclipseApplication
[INFO] WorkflowEngine - workflow completed in 2210ms!
```

```java
package agentAgent;

import java.util.HashMap;

public class CapabilityCapability extends LearningBasedCapability {

    private static final long serialVersionUID = 1L;
    public static final String BELIEF1 = "BELIEF1";
    public static final String BELIEF2 = "BELIEF2";

     * TODO Replace "Object" statements with
    @Belief
    private TransientBelief<String, Object> belief1 = new TransientBelief<String, Objec
            BELIEF1, null);
     * TODO Replace "Object" statements with
    @Belief
    private TransientBelief<String, Object> belief2 = new TransientBelief<String, Objec
            BELIEF2, null);

    @Plan
    Plan2Plan plan2Plan = new Plan2Plan("PLAN2");
    @Plan
    Plan1Plan plan1Plan = new Plan1Plan("PLAN1");

    public CapabilityCapability() {

    private void init() {

        Map<Softgoal, PlanMetadataElement> metadata;
        PlanMetadataElement planMetadataElement;

        metadata = new HashMap<Softgoal, PlanMetadataElement>();

        planMetadataElement = new PlanMetadataElement(plan2Plan,
                AgentSoftgoals.SOFTGOAL2, new Outcome2Outcome(),
                OptimizationFunction.MINIMIZE, LinearRegression.class, 20, 50);
        planMetadataElement.addInfluenceFactor(new NumericInfluenceFactor(
                belief1));
        planMetadataElement.addInfluenceFactor(new NumericInfluenceFactor(
                belief2));
```

Figure 3. Code Generation Feature.

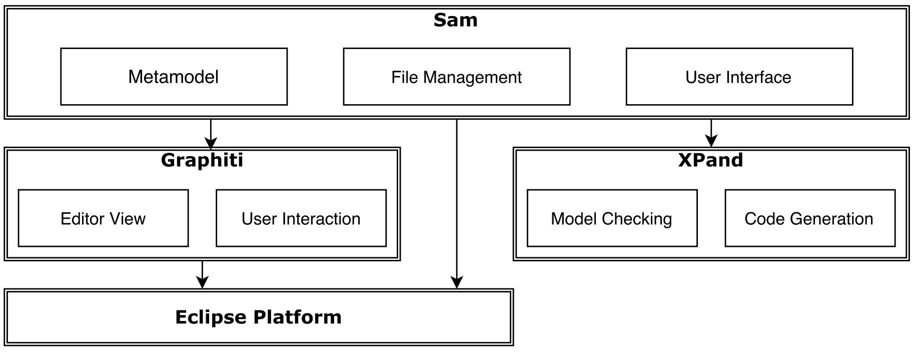


Figure 4. Architecture Model.

3.2. Sam Architecture

Our tool is developed as an Eclipse plug-in built upon the Graphiti[2] framework and the XPand[3] template language. The integration between these different technologies and our tool is presented in Figure 4. Graphiti is particularly suitable for developing graphical editors and viewers. Therefore, it is responsible for creating and displaying the editor view and for handling the user interaction with it. XPand, in turn, is a template language whose functionalities are used specifically on Sam's code generation process. It is responsible for the model validation and code generation. Finally, Sam aggregates the classes related to the meta-model used as the basis for agent modelling and works together with the Eclipse platform to manage diagram and model files and the user interface that does not correspond to the editor view.

4. Evaluation

Previous work that is underlying the Sam tool was evaluated with simulations and empirical studies [Faccin and Nunes 2015, Faccin 2016]. We also evaluated Sam empirically considering three aspects: (i) tool usability, (ii) user satisfaction, and (iii) ease of

[2]http://www.eclipse.org/graphiti/
[3]http://wiki.eclipse.org/Xpand

use. A group of 11 voluntaries was requested to perform two different activities involving the use of our tool. After performing these activities, they were asked to answer a questionnaire—adapted from the USE (Usefulness, Satisfaction, and Ease of Use) questionnaire [Davis 1989, Lund 2001]—composed of 26 seven-point Likert rating scales and two open-ended questions. For each question, voluntaries assigned a score ranging from 1 (strongly disagree) to 7 (strongly agree) to a given statement regarding one of the three aspects evaluated. We also requested voluntaries to provide a list of the three most negative and three most positive aspects they experienced when using our tool. The complete set of questions, as well as obtained answers and additional information on the study participants, are available elsewhere [Faccin 2016].

Results indicate that voluntaries found Sam to be useful (*M* = 6.18, *SD* = 1.08), would recommend it to a friend (*M* = 5.45, *SD* = 1.63) and believe it is easy to remember how to use it (*M* = 5.64, *SD* = 1.12). Although there is no agreement that the tool does everything they would expect it to do (*M* = 4.45, *SD* = 1.51), the majority of voluntaries stated that Sam is somewhat pleasant to use (*M* = 4.82, *SD* = 1.83). Considering answers to the most positive and most negative aspects experienced while using our tool, some voluntaries mentioned that it was easy to get started; lots of useful code is automatically generated, and the tool minimises code handling. However, some voluntaries pointed out some problems. They mentioned that model inconsistencies were not shown or easy to detect, and that a model diagram may become confusing for larger projects.

As this evaluation was performed with a previous version of our tool, we used its results to improve Sam. We were particularly concerned with the issues involving model inconsistency detection and the scalability of the notation used in models. The former was addressed by adding model validation before code generation, as mentioned earlier. The drill-down and collapse features were introduced to deal with the scalability issue.

5. Related Work

There are several approaches that propose the use of graphical editors to model and generate code for agents and multi-agent systems. The DSML4MAS Development Environment (DDE) [Warwas and Hahn 2009] is a tool that supports modelling and code generation of agents developed using a domain-specific modelling language that allows developers to graphically model multi-agent systems with different abstraction levels. Gascueña et al. [Gascueña et al. 2012] present an approach addressing agents based on the Prometheus methodology [Padgham and Winikoff 2004]. For this purpose, they developed the Prometheus Model Editor, allowing developers to graphically model and generate code for Prometheus agents. Pavón et al. [Pavón et al. 2006] present a reformulation of a particular agent development methodology, where they use existing tools provided by such methodology to allow agent modelling and code generation, using different model-to-code transformations to address specific agent platforms.

These initiatives are limited to modelling traditional agent concepts, which do not include techniques that provide agents with intelligent behaviour. Consequently, generated code consists of code skeletons that are similar to code generated from UML class diagrams, and much is left to developers. Our approach, instead, hides sophisticated techniques from developers, so that mainstream software developers are able to leverage such techniques without having to learn their technical details. Sam also provides additional features, such as those to deal with scalability (drill-down and collapse features).

6. Conclusion

Developing software agents is not a trivial task. Frequently, the need for understanding complex concepts and techniques underlying such technology contributes to keeping away users that could potentially benefit from its use. In this paper, we presented Sam, a tool to support the design and implementation of agents with learning capabilities. It provides modelling and code generation features that, when used together, are able to deliver an almost complete BDI4JADE agent code. Sam was not only developed to ease the development process of intelligent agents, but also to promote the adoption of the agent technology in the industry. Our tool will be publicly available with the next stable version of the BDI4JADE platform, licensed under LGPL.

Acknowledgements

This work receives financial support of CNPq, project ref. 442582/2014-5: "*Desenvolvendo Agentes BDI Efetivamente com uma Abordagem Dirigida a Modelos*." Ingrid Nunes thanks for research grants CNPq ref. 303232/2015-3, CAPES ref. 7619-15-4, and Alexander von Humboldt, ref. BRA 1184533 HFSTCAPES-P.

References

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236.

Davis, F. D. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3):319–340.

Faccin, J. (2016). Preference and context-based bdi plan selection using machine learning: from models to code generation. Master's thesis, UFRGS, Porto Alegre. pages 61, 72.

Faccin, J. and Nunes, I. (2015). Bdi-agent plan selection based on prediction of plan outcomes. In *WI-IAT*, volume 2, pages 166–173.

Gascueña, J. M., Navarro, E., and Fernández-Caballero, A. (2012). Model-driven engineering techniques for the development of multi-agent systems. *Engineering Applications of Artificial Intelligence*, 25(1):159–173.

Lund, A. M. (2001). Measuring usability with the use questionnaire. *STC Usability SIG Newsletter: Usability Interface*.

Nunes, I. (2014). *Engineering Multi-Agent Systems: Second International Workshop*, chapter Improving the Design and Modularity of BDI Agents with Capability Relationships, pages 58–80.

Nunes, I., Lucena, C. J. P. D., and Luck, M. (2011). Bdi4jade: a bdi layer on top of jade. In *International Workshop on Programming Multi-Agent Systems*, pages 88–103.

Padgham, L. and Winikoff, M. (2004). *Developing Intelligent Agent Systems: A Practical Guide*. New York, NY, USA.

Pavón, J., Gómez-Sanz, J., and Fuentes, R. (2006). *Model Driven Architecture – Foundations and Applications: Second European Conference*, chapter Model Driven Development of Multi-Agent Systems, pages 284–298. Berlin, Heidelberg.

Rao, A. S. and Georgeff, M. P. (1995). Bdi agents: From theory to practice. In *International Conference of Multi-Agent Systems*, pages 312–319.

Warwas, S. and Hahn, C. (2009). The dsml4mas development environment. In *AAMAS*, volume 2, pages 1379–1380, Richland, SC.

Wooldridge, M. (1999). Intelligent agents. In *Multiagent Systems*, pages 27–77.