UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JOÃO G. FACCIN

Preference and Context-based BDI Plan Selection using Machine Learning: from Models to Code Generation

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. PhD. Ingrid Nunes

Porto Alegre
March 2016

CIP – CATALOGING-IN-PUBLICATION

Faccin, João G.

Preference and Context-based BDI Plan Selection using Machine Learning: from Models to Code Generation / João G. Faccin. – Porto Alegre: PPGC da UFRGS, 2016.

95 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2016. Advisor: Ingrid Nunes.

1. Software agents. 2. BDI architecture. 3. Plan selection process. 4. Agent-oriented software engineering. I. Nunes, Ingrid. II. Título.

"We can only see a short distance ahead,
but we can see plenty there that needs to be done."
— Alan Turing

ACKNOWLEDGEMENTS

I would like to thank my supervisor Prof. Ingrid Nunes for her guidance through the development of this dissertation and for serving as a reference of the researcher I want to be. Without her support and motivation this work would not be possible.

Thanks to the members of the examining committee, Prof. Diana Adamatti, Prof. Felipe Meneguzzi and Prof. Álvaro Moreira for taking the time of reading my dissertation and providing valuable feedback.

I would also like to thank my friends and colleagues from the Federal University of Rio Grande do Sul (UFRGS), for their help, support and friendship. A special thank you to my lab colleagues Vanius, Jhonny, Fernando and Vinícius, for their suggestions, ideas and our endless conversations.

My sincere gratitude to my family and friends, who always believed in me and supported me in my choices. Thanks to my parents, Cláudia and João, for all the love, and for teaching me to always pursue my dreams. To my brother and sister, Carlos and Ana, thanks for always being with me when I needed. To Nalim, my girlfriend, thanks for the comprehension and affection, and for being my safe haven when I needed to reassure me.

To those whose name I did not mention, but helped me developing this work in any way, thank you all. Finally, I must acknowledge that this research would not have been possible without the financial assistance of CNPq and CAPES and the infrastructure provided by the Federal University of Rio Grande do Sul.

ABSTRACT

Agent technology arises as a solution that provides flexibility and robustness to deal with dynamic and complex domains. Such flexibility can be achieved by the adoption of existing agent-based approaches, such as the BDI architecture, which provides agents with the mental attitudes of beliefs, desires and intentions. This architecture is highly customisable, leaving gaps to be fulfilled in particular applications. One of these gaps is the plan selection algorithm that is responsible for selecting a plan to be executed by an agent to achieve a goal, having an important influence on the overall agent performance. Most existing approaches require considerable effort for customisation and adjustment to be used in particular applications. In this dissertation, we propose a plan selection approach that is able to learn plans that provide possibly best outcomes, based on current context and agent's preferences. Our approach is composed of a meta-model, which must be instantiated to specify plan metadata, and a technique that uses such metadata to learn and predict plan outcomes. We evaluated our approach experimentally, and results indicate it is effective. Additionally, we provide a tool to support the development process of software agents based on our work. This tool allows developers to model and generate source code for BDI agents with learning capabilities. A user study was performed to assess the improvements of a tool-supported BDI-agent-based development method, and evidences suggest that our tool can help developers that are not experts or are unfamiliar with the agent technology.

Keywords: Software agents. BDI architecture. plan selection process. agent-oriented software engineering.

Seleção de Planos BDI Baseada em Contexto e Preferências usando Aprendizado de Máquina: dos Modelos à Geração de Código

RESUMO

A tecnologia de agentes surge como uma solução que fornece flexibilidade e robustez para lidar com domínios dinâmicos e complexos. Tal flexibilidade pode ser alcançada através da adoção de abordagens já existentes baseadas em agentes, como a arquitetura BDI, que provê agentes com características mentais de crenças, desejos e intenções. Essa arquitetura é altamente personalizável, deixando lacunas a serem preenchidas de acordo com aplicações específicas. Uma dessas lacunas é o algoritmo de seleção de planos, responsável por selecionar um plano para ser executado pelo agente buscando atingir um objetivo, e tendo grande influência no desempenho geral do agente. Grande parte das abordagens existentes requerem considerável esforço para personalização e ajuste a fim de serem utilizadas em aplicações específicas. Nessa dissertação, propomos uma abordagem para seleção de planos apta a aprender quais planos possivelmente terão os melhores resultados, baseando-se no contexto atual e nas preferências do agente. Nossa abordagem é composta por um meta-modelo, que deve ser instanciado a fim de especificar metadados de planos, e uma técnica que usa tais metadados para aprender e predizer resultados da execução destes planos. Avaliamos nossa abordagem experimentalmente e os resultados indicam que ela é efetiva. Adicionalmente, fornecemos uma ferramenta para apoiar o processo de desenvolvimento de agentes de software baseados em nosso trabalho. Essa ferramenta permite que desenvolvedores modelem e gerem código-fonte para agentes BDI com capacidades de aprendizado. Um estudo com usuários foi realizado para avaliar os benefícios de um método de desenvolvimento baseado em agentes BDI auxiliado por ferramenta. Evidências sugerem que nossa ferramenta pode auxiliar desenvolvedores que não sejam especialistas ou que não estejam familiarizados com a tecnologia de agentes.

Palavras-chave: Agentes de software, arquitetura BDI, processo de seleção de planos, engenharia de software orientada a agentes.

LIST OF ABBREVIATIONS AND ACRONYMS

BDI	Belief-Desire-Intention
ILP	Inductive Logical Programming
BUL	Bottom Up Learning
CL	Concurrent Learning
ACL	Aggressive Concurrent Learning
CPS	Contextual Planning System
ESum	Excluding Sum
EMax	Excluding Max
MAUT	Multi-Attribute Utility Theory
API	Application Programming Interface
GQM	Goal-Question-Metric
USE	Usefulness, Satisfaction, and Ease of use
IDE	Integrated Development Environment
XML	eXtensible Markup Language

List of Figures

List of Tables

Contents

1 INTRODUCTION

Recently, the interaction between people, systems and tools evolved in a way that it became remarkable. More than growing in scale, this evolution also involves an increase in complexity and have affected many knowledge areas. One of these areas is Computer Science, which includes the development of new technologies to deal with this scenario. Software agents are a promising approach to deal with complex and dynamic problems. This concept of agent is not new and, although it has received more attention especially in the last two decades, relates to the initial definitions of Artificial Intelligence, which not only aims to understand, but also build intelligent entities (RUSSELL; NORVIG, 2003).

Software agents can be defined as computational entities with autonomous, proactive and reactive characteristics. It means that an agent is capable of perceiving the environment it is inserted (through sensors) and acting on this environment (through actuators) or reacting to changes in it, aiming to achieve their own goals, without the need for human intervention.

It is common to find dynamic scenarios dealing with interactions between different entities in industry; however, software agents are not frequently used as an approach to address these domains. Currently, agent-based solutions developed and in use cover a small range of sectors in industry, being used for specific applications. We can find successful cases in military, logistics and insurance management sectors, as well as examples related to production chains (BELECHEANU et al., 2006).

Several methodologies were proposed to support the design and implementation of software agents. The belief-desire-intention (BDI) architecture (RAO; GEORGEFF, 1995) is the focus of various studies and can be highlighted as one of these approaches. This architecture is based on the BDI model (BRATMAN, 1987) to provide to an agent the mental attitudes of beliefs, desires and intentions, which determine agent behaviour. The BDI architecture gives to developers robustness to deal with different domains in which a flexible and intelligent behaviour is necessary. For this purpose, this architecture is composed of a set of modules that are connected in a reasoning cycle, which leaves many gaps to be fulfilled in particular applications. One of the gaps is the plan selection process, which is the process that selects, from a set of predefined suitable plans, the best plan to achieve a given goal according to a given criteria. Without this plan selection process, one of the suitable plans is selected randomly.

Thus, to leverage the power provided by the BDI architecture, it is necessary, among

others factors, an appropriate plan selection algorithm. Thereby, several studies have been done in the context of plan selection process in BDI agents (GUERRA-HERNáNDEZ; FALLAH-SEGHROUCHNI; SOLDANO, 2004; VISSER; THANGARAJAH; HARLAND, 2011; SINGH et al., ; NUNES; LUCK, 2014). Many techniques have been developed to improve this process, ranging from the use of machine learning techniques to the adoption of concepts of preferences satisfaction and utility values. However, most of these approaches require considerable effort for customisation and adjustment to be used in particular applications, such as the specification of utility values and probabilities, thus calling for specific knowledge about the domain they are going to be used and consequent assistance from an expert in the given domain. This intense need for domain-specific knowledge is one of the obstacles that the agent technology faces in industry regarding its adoption in large-scale applications.

Therefore, more than providing theoretical and conceptual background, we aim to demonstrate that software agents have practical applicability and can be effectively used in industry. Without practical implementations, the use of all techniques is limited to a small number of specialists inside the academia, preventing their use by the development community and moving their benefits away from potential users.

Given this context, we noticed an opportunity to explore concepts and techniques related to the plan selection process of software agents built using the BDI architecture. Therefore, we present the research problem addressed by this dissertation and the limitations of existing work in Section 1.1. We describe our proposed solutions and present an overview of the contributions of this work in Section 1.2. Finally, Section 1.3 details the structure of the remainder of this dissertation.

1.1 Problem Statement and Limitations of Related Work

As introduced earlier, the BDI architecture stands out as a well-known and used approach for developing software agents. Several development languages and platforms emerged to allow the implementation of this architecture, such as JACK (BUSETTA et al., 1999), Jason (BORDINI; HüBNER; WOOLDRIDGE, 2007), BDI4JADE (NUNES; LUCENA; LUCK, 2011), among others.

A typical BDI agent has a plan library, which consists of a collection of plans indexed by goals or events and defined at design time, representing the standard behaviour of the agent in a given context. Each of these plans has a set of context conditions, which

defines under which state of the environment this plan can achieve a particular goal. Thus, during its reasoning cycle, the agent selects, from a set of plans whose context conditions holds considering the current state of the environment, a plan that is expected to perform successfully.

However, in most implementations, this plan selection is performed randomly by default, disregarding aspects such as the possibility that a plan has to fail in particular circumstances or the uncertainty that surrounds the environment's behaviour. Moreover, most of these implementations do not consider agent's preferences on how to achieve a given goal. Based on this issue, we state our research question below.

Research Question

How can a BDI agent perform the plan selection process considering the current context and the uncertainty related to the outcomes of a plan while also considering its own preferences?

Several approaches emerged aiming to improve the plan selection process, bringing solutions ranging from learning-based plan selection to taking into account agent's preferences. We list the limitations of these existing work next.

There is a lack of consideration regarding satisfaction of agent's preferences. Frequently, when trying to accomplish a goal, we have preferences on how we want to achieve this objective. Most of the existing work does not allow the expression of these preferences in an agent environment. When they do, they require developers or users to explicitly provide information that can be considered unrealistic for real applications such as probabilities, which are hard to elicit, may evolve over time, and may be context-dependent. In a real world application, providing this information becomes an expensive task, which is also affected by the following limitation.

Most existing work does not consider how the context can influence the execution of a plan. A plan able to achieve a given goal in a particular context may not have the same performance in two different executions. It can perform successfully in one attempt and leads to a failure situation in another. Moreover, even if it has the same result, the way the plan contributes to satisfying a preference may change due to unobservable factors. Thus, this limitation relates directly to the existence of uncertainty in an agent environment. There is work that considers uncertainty in a certain degree; however, they do not address the role played by context elements in this uncertainty and

how they can impact a plan's outcome. Therefore, we can improve the results from a plan's execution when concerning about these factors during the plan selection process.

There is a lack of work addressing tool-support for developing agents implementing plan selection approaches. Providing theoretical foundation is an initial step towards the development of BDI agents able to perform an effective plan selection. However, it is also necessary to demonstrate the feasibility of such theory to be used in real world applications. Most of the existing work do not focus on this issue, being limited to present their theoretical contributions. Therefore, providing means of easing the development of agents following a given approach can contribute to promote the use of the agent technology.

1.2 Proposed Solution and Overview of Contributions

Given our aim of improving the plan selection process of BDI agents considering preferences and the context that affects the execution of a plan, we proposed an approach that involves representing these elements and using this representation to perform the plan selection. Therefore, we proposed the study and development of a meta-model that provides means of expressing the metadata representing the relationship between elements of the environment and a plan's outcome. More than just success or failure, a plan can have different outcomes when we consider the way it executes — e.g. it can spend more or less of a given resource during its execution. Moreover, this meta-model also defines the relationship between metadata elements and the structures that comprise the BDI architecture, such as plans, beliefs, goals, etc.

We also propose a technique that uses the information from our meta-model to perform plan selection. This technique involves a plan selection algorithm and the use of machine learning techniques. Our approach monitors the metadata provided and builds prediction models for each plan using a predefined learning algorithm. Therefore, these prediction models provide the possible value that every plan's outcome has in a given context. The plan selection algorithm assigns a utility value for each plan using the information from each prediction model, then selects the plan with the highest utility value. Moreover, the outcomes of a plan's execution are recorded and stored to update and improve the prediction models when necessary.

Another concern of our work is to promote the adoption of software agent technologies in real world applications. Thus, we proposed the practical implementation of our meta-

model and technique as an extension of the BDI4JADE framework (NUNES; LUCENA; LUCK, 2011). We chose this platform for two particular reasons. Firstly, it already encompasses some elements of our meta-model e.g. preferences and softgoals, contributing to the easy extension of the framework. Secondly, this platform is purely Java-based, which makes agents implemented with it able to be extended and integrated with different technologies currently in use, matching with our objective of promoting the utilisation of the agent technology in industry.

Moreover, a developer familiar with the BDI4JADE platform only needs to learn about our meta-model to create an agent with learning characteristics, without being concerned with how the plan selection technique is implemented. Given that our implementation encapsulates specific knowledge about the plan selection approach, its use becomes transparent to the final user.

However, modelling and developing an agent is still a complex task that requires a great effort from developers, especially concerning the need for adjustments and customisations for specific domains. Considering this, we propose the development of a solution that allows a simplified way to implement software agents, from the design to the coding stage. This solution — a graphical tool that called *Sam* — allows a developer to graphically model and instantly generate source code for an agent based on our extension of the BDI4JADE framework. *Sam* also provides an entry point for the first contact with the agent technology, with novice developers being able to produce a useful agent in few steps.

In summary, the main contributions of this dissertation are:

1. **a meta-model**, which allows modelling software agents with learning capabilities;
2. **a technique for plan selection**, which uses the information from the meta-model to select the plan expected to better satisfy agent's preferences;
3. **the implementation of the proposed meta-model and technique** as an extension of the BDI4JADE framework; and
4. **a tool to support the development process of agents based on our work**, allowing the modelling and generation of source code for such agents.

1.3 Outline

The remainder of this dissertation is organised as follows. Chapter 2 presents a theoretical background, detailing the concepts of software agents and the BDI architecture.

It also discusses related work, which uses different techniques aiming to improve the plan selection process of BDI agents. Thus, Chapter 3 introduces our approach, defining the meta-model and technique developed to provide to an agent a learning capability able to satisfy its preferences. From these definitions, Chapter 4 presents the practical implementation of our meta-model and technique as an extension of the BDI4JADE framework, as well as *Sam*, the graphical tool we developed to support modelling and generating source code for BDI agents. Moreover, we evaluated our technique and tool by performing a simulation and a user study, respectively. Chapter 5 describes these experiments as well as presents and analyses their results. Finally, conclusions and future work are presented in Chapter 6.

2 BACKGROUND AND RELATED WORK

In this chapter, we present the theoretical background that guided our research. We also discuss existing work related to the plan selection process in BDI agents. Section 2.1 presents a brief overview of software agents, its definition and main concepts. In Section 2.2 we introduce the main concepts, components and benefits of the BDI architecture, as well as its reasoning cycle. In Chapter 2.3 we show work that adopts machine learning techniques in the plan selection process. Chapter 2.4 highlights the main techniques integrating preference satisfaction to the plan selection process, as well as its relation to the use of utility values in this process. Finally, in Chapter 2.5 we discuss and relate the work described in previous sections.

2.1 Software Agents

The concept of *agent* is not new, although it has received more attention especially in the last two decades (RUSSELL; NORVIG, 2003). An intelligent software agent can be defined as a computational entity with *autonomous*, *proactive* and *reactive* characteristics, which is situated in some environment. The concept of agent autonomy relates to the total control over its internal states and behaviour. The proactive and reactive characteristics allow the agent to act in anticipation of future goals and respond in timely fashion to environmental changes, respectively. This kind of behaviour is possible due to the existence of sensors that allow the agent to experience the environment and actuators that allow this agent to act on it.

However, in most cases, a single agent is not sufficient to address complex and dynamic domains, given that these domains typically involve several elements and entities. Therefore, an intelligent agent must also present *social ability*, being able to interact with other agents to achieve its goals. For this interaction successfully occur, it is necessary that agents cooperate, coordinate and negotiate with each other, much as people do.

A stronger notion of agency adds to the characteristics previously mentioned the existence of mental components, such as beliefs, desires, intentions, wishes, and so on to explain an agent's behaviour. Next section presents an architecture to support the development of agents with such mental components.

2.2 BDI Architecture

Several approaches have been developed aiming to deal with the implementation of software agents. A known and widely used architecture to develop these software agents is the BDI architecture. It is the base for a lot of languages and platforms to agent development, like JACK (BUSETTA et al., 1999), Jason (BORDINI; HüBNER; WOOLDRIDGE, 2007), BDI4JADE (NUNES; LUCENA; LUCK, 2011), among others.

The BDI architecture was proposed by Rao and Georgeff (1995), with its base in concepts of intentional systems (DENNETT, 1987) and practical reasoning (BRATMAN, 1987). This architecture provides to an agent the mental attitudes of beliefs, desires and intentions, which collectively define agent behaviour. The behaviour of an agent built upon the BDI architecture arises from the commitment to achieve a goal.

In this context, we define the role of each of these mental attitudes. Beliefs represent the information about the world. They can be updated by the perception of the agent or by its intervention in the environment. Desires, which are alternatively referred to as goals, can be represented as the states of the world the agent wants to reach. When an agent has a commitment to achieve one of its desires, we say that it has the intention to accomplish some task. Thus, the intention is the course of actions an agent has committed to perform.

When seen as data structures, these mental attitudes form the basis of BDI architecture. Additionally, there is another element to point out: the plan library. Plan library is a set of plans defined by the developer at design time. A plan contains a series of actions to be executed. This plan is commonly composed of: (i) an invocation condition that defines which event the plan will handle, e.g., a goal to achieve; (ii) a precondition, i.e., a set of conditions that establishes under which context this plan will be executed; and (iii) a plan-body containing the course of actions to be performed, i.e., actions that affect the environment, subgoals to be achieved, etc.

Figure 2.1 shows the relationship between the three main structures and some functions defined by the BDI architecture (RAO; GEORGEFF, 1995). When integrated into the reasoning cycle and added to other customisable modules, these structures provide flexibility and robustness that are characteristics of this architecture. This reasoning cycle can be described as the following sequence of steps, where the agent:

1. perceives the environment where it is inserted and, from this perception and/or internal actions, updates its beliefs and desires; and also lists new events to manage;

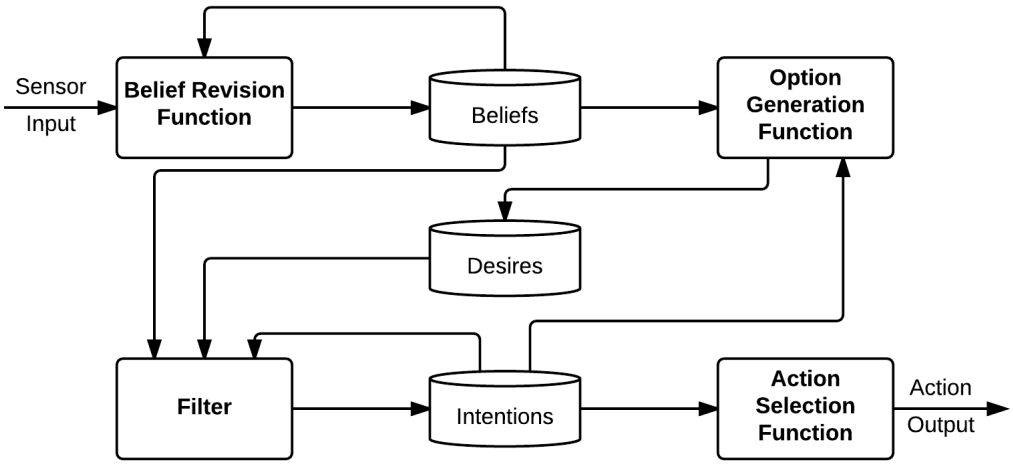


Figure 2.1: A Typical BDI Architecture.
Source: Wooldridge (1999)

2. selects an event and identifies a set of relevant plans, i.e., a set of plans able to handle this selected event;
3. selects, from the set of relevant plans, a set of plans whose execution conditions hold the current state of the environment;
4. chooses a plan from the set of executable plans using a selected algorithm;
5. creates an instance of the plan to be executed and pushes it onto an existing or new collection of intentions; and
6. selects a collection of intentions, executes the current step of the plan on top of it, and restarts the cycle.

This reasoning cycle is the main responsible for the architecture's ability to adapt in a dynamic environment. It makes the BDI architecture especially suitable when there is a need for an almost immediate response to deal with an environment in constant change. Moreover, it allows this responsiveness by constantly receiving updates about the state of the environment, and thus, plans are selected and performed just when necessary, based on current beliefs.

The plan selection process constitutes a key element for the proper functioning of the architecture. Thus, the adoption of a plan selection algorithm that selects a plan that best matches the current context and not simply verifies its applicability can significantly affect the performance of the agent.

From the three main components previously explained (the invocation condition, the precondition and the plan body), just two of them effectively work in the plan selection

process. When an event (or goal) is selected, every plan where the invocation condition unifies with selected event is identified as relevant, i.e., a set of plans that can handle this event is created. From this set of relevant plans, the executable ones are selected, i.e., those whose precondition hold true given current beliefs.

Then, from this set of executable plans, an algorithm defined by the developer selects a plan to add to the intentions structure. This algorithm can choose a plan randomly, following a predefined condition, or any other method (and this is one of the points where the performance can be improved). When selected, a plan instance is allocated in the structure of intentions. If this plan derives from an internal event, i.e., with its origin in another plan, this instance is set on top of a collection that already exists in the structure; else a new structure is created.

However, the selected plan may not achieve the expected result when executed i.e. a successful execution, thus leading to a failure situation. When a failure situation occurs, the agent starts a process of failure recovery, executing a backtracking for selecting another plan from the set of executable plans.

The plan selection process noticeably influences the whole system performance. An inadequate plan selection process can lead to a drop in agents performance, e.g., for never selecting applicable plans in specific situations, or for frequently choosing plans that will result in a failure situation and, thus, spending resources and time to recover from this failure situation.

Thus, many techniques have been developed with the aim of improving the plan selection process. We have approaches ranging from the use of machine learning techniques to preference satisfaction techniques. So, these approaches and techniques used to improve plan selection process are described and discussed in following sections.

2.3 Learning-Based Plan Selection

According to Russell and Norvig (2003), the first mention of learning in intelligent machines emerged with Turing (1950). Considering the possibility of hand-programming his intelligent machines, Alan Turing imagined that there would be an efficient method to do it. So, he proposed creating machines able to learn and which, subsequently, could be taught how to accomplish their tasks.

After that, machine learning concepts have been widely used in several areas. This popularity is given mainly by the benefits offered by this technology. With learning

capabilities, an intelligent agent can operate in unknown environments and raise its performance to a level that would not be reached using just its initial knowledge.

An agent able to learn can be divided into four conceptual elements: (i) a learning element, responsible for performing enhancements by executing a learning algorithm; (ii) a performance element, responsible for selecting actions to be performed; (iii) a critic element, responsible for providing feedback; and (iv) a problem generator element, responsible for suggesting exploring actions, leading to new and informative experiences (RUSSELL; NORVIG, 2003). Based on this conceptual division, Guerra-Hernández, Fallah-Seghrouchni and Soldano (2004) described one of the first attempts to integrate learning into the BDI architecture's plan selection process. For assuming the entire agent as the performance element, they considered the context conditions of a plan as the element to be learned, i.e., their agent is able to learn when to use their plans. After learning, the agent also suggests modifications in its definitions of plans preconditions, aiming to apply its acquired knowledge and improve its performance.

In this context, they describe the use of Inductive Logic Programming (ILP) as a tool to endow agents with learning skills. Specifically, the use of Induction of Logical Decision Trees as a technique to implement learning in their architecture is a significant contribution from Guerra-Hernández, Fallah-Seghrouchni and Soldano (2004). The use of this approach was proposed because the decision tree representation (corresponding to a disjunction of conjunctions) is equivalent to the one utilized in the definition of plan's context conditions.

This concept of Inductive Logic Programming was also used by Phung, Winikoff and Padgham (2005) with Alkemy, which is a symbolic inductive learner that uses ILP to produce a decision tree. Their approach uses Alkemy to generate decision trees for suggesting a suitable plan for firefighting. The suggestion is used just after an estimation of the accuracy of the tree. It is done by using a threshold that is achieved by the authors in many ways. If the accuracy is "good enough", i.e., above this threshold, then the plan suggested is executed; else, the agent starts to explore.

This brings us the known dilemma of *exploration* versus *exploitation*. In this problem the agent needs to choose what is a better decision: to use the knowledge achieved by previous executions of plans leading to an expected result (exploitation); or to explore new possibilities for executing plans never tried in a given context, aiming to get better results than those that its knowledge base suggests, and thus improving its knowledge (exploration).

However, none of these presented approaches have looked at the problem of how an agent learns to select the most appropriate plan amongst those applicable in a given context. They focused on learning the context in which a plan leads to a successful state. To address this limitation Nguyen and Wobcke (2006) proposed an approach focused on dialog adaptation, i.e., how to properly display information to users and interact with them, in their mobile device (called Smart Personal Assistant). As well as in the work of Phung, Winikoff and Padgham (2005), this approach uses Alkemy as a component of its learning method.

A group of researchers have been developing a series of studies (AIRIAU et al., 2008; AIRIAU et al., 2009; SINGH et al., ; SINGH; SARDINA; PADGHAM, 2010; SINGH et al., 2011) where they use decision trees linked to each plan as a resource to learn context conditions where this plan is applicable and leads to a successful state. However, the main concerns of their approaches are to define when a suggestion from a learning algorithm is reliable, and when to feed the learning algorithm with instances to train.

Many approaches proposed by this particular research group relies on a structure built based on plans and goals, namely goal-plan tree. This goal-plan tree is presented by Airiau et al. (2008) as a representation of the agent plan library and is used as a resource for subsequent work. In the goal-plan tree, a goal is a node which has child nodes corresponding to plans that handle this objective. These plans may have subgoals that, consequently, have plans to deal with them, and so on. This approach was proposed because it also suits to the process of failure recovery of the BDI reasoning cycle. Figure 2.2 shows an example of a goal-plan tree, where G_n corresponds to a (sub)goal and P_n to a plan.

The work of Airiau et al. (2008) also mentions the idea of building the decision tree online, i.e., instead of learning the behaviour from a static data set, it would be done accumulating experiences when the system was already deployed. However, while recognizing the existence of incremental algorithms to do it, it is not applied in this work.

Airiau et al. (2009) deal with the problem of when to incorporate information about a plan execution in the corresponding plan's decision tree by defining two approaches: the bottom-up learning (BUL) and the concurrent learning (CL) methods. The difference between these two approaches is in the way they consider a failure execution to be recorded. The BUL approach will only record a failure execution if the plan is stable enough, i.e., when a minimum number of recorded instances of the plan is reached, instead of always assimilating all data as CL does. This concept of stable plans clearly relates to the idea

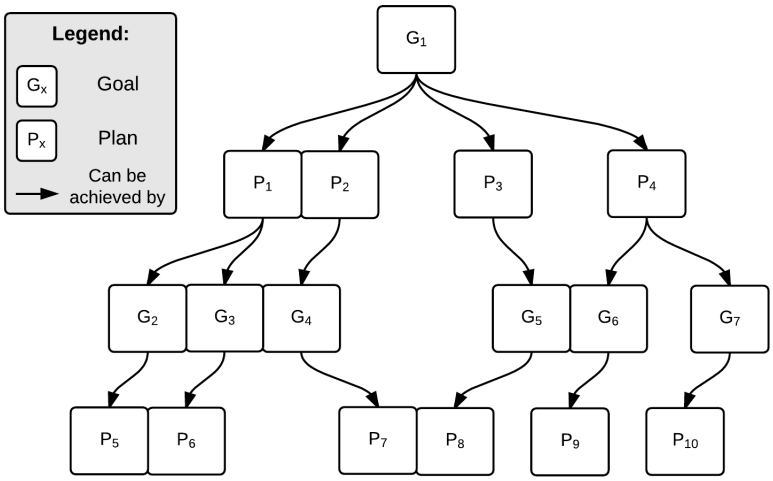


Figure 2.2: A Goal-Plan Tree.

of a "good enough" accuracy from Phung, Winikoff and Padgham (2005).

Additionally, the authors define the concept of the stable decision tree as a tree where its suggestion reflects the reality. This definition is implemented through an algorithm that considers a tree stable when all plans that compose this tree are also considered stable.

The work of Singh et al. () continues previous studies for extending the BUL and CL (in this work called ACL - Aggressive Concurrent Learning) methods with a coverage-based confidence measure. This measure is based on the definition of coverage as a value indicating how explored a plan is in a given context. Additionally, they defined an applicability threshold to balance the exploration and exploitation of plans. Singh, Sardina and Padgham (2010) improved this study for allowing learning for parameterised goals and recursion, both of which are characteristics necessary in real applications. A modified confidence measure was defined, incorporating a decay factor to its calculation aiming to adapt previous work to this improvement.

Finally, Singh et al. (2011) end by allowing an agent to adapt to changes in its environment. Instead of a value that just increases to indicate that a plan is achieving full confidence, this approach uses a dynamic confidence measure from which its value can be decreased, indicating a confidence loss. It helps agents to detect when a learned behaviour becomes less effective in the environment, allowing them to re-learn.

Recently, work from Chaouche et al. (2015) also used machine learning techniques to improve the plan selection process. However, instead of learning plans or contexts, they aimed to learn the outcome of single actions (e.g. success or failure, duration, etc.).

Thereby, they use the knowledge obtained from this learning on a planning process that combines these actions to provide a plan able to achieve a given goal. This planning process involves what they called the Contextual Planning System (CPS), which is responsible for generating all possible execution paths for the actions of a plan.

Moreover, Chaouche et al. (2015) used an approach similar to the Q-learning algorithm (WATKINS; DAYAN, 1992), to learn a quality value for each action considering that known values become deprecated over time. It allows the agent to give more importance to the most recent experiences, considering them more informative than older ones. Thus, the agent selects the execution path that can accomplish a given goal while respecting a specific strategy.

As seen in this section, many approaches have exploited machine learning techniques in the context of the plan selection process in BDI agents. However, most of them have not taken into account an important concept that may influence the plan selection process: agent preferences. Approaches that take them into account are presented in next section.

2.4 Preference-based Plan Selection

An agent developed upon the BDI architecture can select the most suitable plan in its plan library to achieve a given goal. However, sometimes there are situations which we expect this goal to be accomplished in a particular way. A way to inform the agent about this expectation is specifying preferences over plans or plan instances, i.e., a preference among plans which are operationally equivalent (achieve the same goal), and among different values for the variables of a single plan. Nevertheless, there is no much work considering the use of preferences on the plan selection process.

One of the first attempts to use preferences in plan selection process was the work made by Nguyen and Wobcke (2006), where they propose a solution incorporating user preferences and learning for dialog adaptation in a Smart Personal Assistant, as explained in the previous section. Their approach aims to predict when a plan is applicable according to user preferences. When there is more than one applicable plan, the agent asks the user to clarify his preferences over these plans, learning it, and selecting the most preferred plan. In this way, there are no previously specified preferences, as they are informed and refined due to the use of the Smart Personal Assistant. Finally, preferences are defined just over different plans, instead of also considering plan instances.

A more complex approach is proposed by Visser, Thangarajah and Harland (2011)

whose work shows a solution where preferences are defined over possible values of goal properties (or attributes), as well as over plan usage of resources to achieve a given goal. To allow this kind of specification an adapted version of the preference language *LPP* is used. This approach considers the agent plan library as a goal-plan tree (like most of the approaches presented in Section 2.3) in which nodes are annotated with information about goal properties and usage of resources. This information is then propagated to upper levels of the hierarchy at compile time. In this way, a node can have summaries of properties and resources propagated from child-nodes, as well as summaries of properties and resources directly specified by the programmer.

In the plan selection process this technique proposes to express numerically how well a plan satisfies the specified preferences. This numerical value is inferred by calculating a score for each plan given a goal, a set of preference formulas assigned to this plan and a set of relevant metadata. The score is calculated by evaluating each plan's preference formula from a set of formulas that are related to the given goal, giving a value to them (where a lower value means more preferred), and summing these values. Then, plans are sorted in a non-increasing order of preference satisfaction, and this sort establishes the order in which plans will be attempted. Some limitations of this work include the fact that conflicts between preferences specifications as well as interactions among them are not addressed.

Another study addressing the use of preferences in the plan selection process was conducted by Padgham and Singh (2013). A primary concern of this work is to address the fact that, in common BDI agent implementations, preconditions which define the applicability of a plan are overconstrained. Overconstraining these preconditions makes a plan unusable under conditions in which it would be potentially useful. Also, this approach wants to allow end-users to know which plan an agent will select and why it will choose that option. So, the definition of preference proposed in this work tries to favour a subsequent interpretation of why a plan was selected over another.

Like the previous proposal (VISSER; THANGARAJAH; HARLAND, 2011), this approach seeks to assign a quantitative value to each plan instance. This value assignment uses a preference specification that allows the calculation of the plan value based on current state of the agent environment as well as on the attributes of a particular plan instance. However, there are cases in which several preference specifications hold true to a particular plan instance, leading a plan to have different preference values. Thus, there is a need for defining a function to aggregate this diverse preference values appropriately,

aiming to determine a single plan value that will serve as a reference for sorting the set of applicable plans.

Then, two aggregation functions were defined: *ESum* and *EMax*. The *ESum* (for *Excluding Sum*) function works excluding more general, entailed descriptions from the set of preference formulas that are true for the particular plan instance, summing values of remaining functions, and then assigning this final measure to the plan's instance value. The second defined aggregation function, called *EMax*, uses the same exclusion criteria as the *ESum* function, but instead of summing the values of remaining formulas, it selects the higher measure and assigns this as the plan's instance value.

An issue of this approach is that the plan sorting is made from values specified at design time for a developer. In a large set of plans or instances this becomes a task which demands a great effort from the developer, who must also ensure that the final ordering would be the desired one.

A similar need for providing a considerable amount of information can be found in the work of Nunes and Luck (2014), which shows an approach that aims to improve the efficiency of the plan selection process considering preferences over softgoals. Based on the BDI architecture, this approach considers softgoals as secondary goals, which can be more or less satisfied with a plan execution, e.g. achieve a goal with more or less safety, rapidness, resource saving, etc. Thus, at design time, developers must inform the initial values related to agent's preferences for each of the defined softgoals. This value must be between 0 and 1 (including the limits), where 0 indicates the lowest and 1 indicates the highest possible preference.

Moreover, for each softgoal, in each plan, values related to the plan contribution for satisfying a particular softgoal are informed, together with the probability of the plan to achieve this contribution. Thus, as well as the preference value, these plan contribution values and contribution probability values are informed at design time and must be between 0 and 1 (inclusive), where values close to 1 indicate a greater contribution and a greater probability that this contribution occurs, respectively. This approach uses all these provided data along with an algorithm based on the Multi-Attribute Utility Theory (MAUT), aiming to calculate a utility value that is assigned to each plan. This utility value is then used as criteria to sort the set of applicable plans in a way that the plan with a greater utility value will be attempted.

Chaouche et al. (2015) use a similar approach to deal with agent preferences over the performance of plans. In this work a preference value is also expressed as a balance

proportion (a value between 0% and 100%); however, values of plan contributions do not need to be explicitly provided such as in the previous approach, as they are learned from past plan executions. This work presents another similarity to the work from Nunes and Luck (2014) when we consider that the concept of softgoal used in the former relates to the agent's strategies in the latter. Therefore, the execution path selected to be performed is that which best satisfies a given strategy.

In addition, there is the work from Dam and Winikoff (2008) that uses the concept of a cost-value to perform the plan selection in BDI agents. This approach is part of a framework proposed as a way of fixing violations of consistency constraints in software. Dam and Winikoff (2008) defined the notion of cost as an abstract value calculated counting the number of primitive actions on a given repair plan. To each primitive action, the developer must assign a numerical cost, used to compare different action types. They also defined an algorithm to calculate the overall plan cost considering the main cost of a plan (composed of the values of its primitive actions and its subgoals) and a scope cost of repairing all violated constraints (consisting of the minimum value of the set of plans that can handle each of these constraints). Then, in a given state, the plan selection algorithm calculates the cost of the set of applicable plans and select the one with minimum cost. Therefore, we can also consider this technique as performing a preference-based plan selection where the higher preference is assigned to minimise the cost of a plan execution.

However, an issue of this approach is that it needs to perform a lookahead technique to discover the cost of a plan. It resembles to a planning problem that needs to determine a full course of action and reflects the philosophical difference between planning and BDI execution.

As we can see, there are several ways in which we can deal with the plan selection process. Thus, next section discusses strengths and weaknesses of these different approaches.

2.5 Discussion

After presenting work whose aim is to improve the plan selection process through several methodologies, we discuss the effort made by these previous studies. Firstly, this general overview of related work allows us to clearly identify two main sets of approaches that deal with the problem of improvement of the plan selection process in BDI agents: one using machine learning techniques, and other using agent preferences.

The first set of approaches can be subdivided considering its learning target. Some of

the approaches are concerned with learning in which context a plan is applicable (PHUNG; WINIKOFF; PADGHAM, 2005; GUERRA-HERNáNDEZ; FALLAH-SEGHROUCHNI; SOLDANO, 2004). In complex environments, writing correct preconditions may be a difficult task. So, these approaches can save some effort lessening the burden of domain modellers to encode perfect plan preconditions for this plan being applicable. Then, these preconditions can be generally (or also no-) specified and refined by learning.

Moreover, there are approaches that are focused on learning, from the set of applicable plans, which is the most suitable in a given context (NGUYEN; WOBCKE, 2006; AIRIAU et al., 2008; AIRIAU et al., 2009; SINGH et al., ; SINGH; SARDINA; PADGHAM, 2010; SINGH et al., 2011), i.e., from the learning process, they basically establish the order in which the plans will be attempted in a given context. This kind of approach still needs the developer to provide information about preconditions and may impact in the usability in larger domains with hundreds or thousands of plans. Clearly, a better approach would provide a way in which developers may suppress certain information that is strongly linked to the need for a specific domain knowledge, like preconditions. Furthermore, determining overconstrained preconditions can prevent a plan from being executed in situations where it would be potentially useful.

The third kind of learning target is the one presented by Chaouche et al. (2015), whose focus is to learn the outcome of primitive actions. The concern of this work is a bit different because it aims to use the knowledge acquired to build better plans through a planning process. In this context, this approach ends providing a lookahead mechanism to discover the utility of a plan. This technique can be seen as a planning strategy into a plan. This planning problem reflects the philosophical difference between planning and BDI execution, where the latter makes a choice at a time, considering its environment state while the former previously defines an entire course of action to follow.

Another detail that stands out is that there is just one approach (CHAOUCHE et al., 2015) using a different learning technique than decision trees to support its plan selection process. There is a significant amount of learning algorithms and models that can potentially improve results of the learning process in particular situations or domains, such as support vector machines (SMOLA; SCHOELKOPF, 1998), linear regression (GALTON, 1894), etc. However, as mentioned above, other learning techniques were not explored at all. Furthermore, when we consider that the decision trees from current approaches are built offline, a problem is identified if these approaches are used in dynamic environments. The only approaches that handle this issue is the one of Singh et al. (2011), which

allows an agent of re-learn its behaviour when perceiving a loss of effectiveness in its plan selection; and that from Chaouche et al. (2015), which uses a technique of reinforcement learning to continually improve its plan selection.

Moreover, not all of these solutions deal with the problem of uncertainty when learning. Those approaches that handle this issue adopt a method to perform the plan selection just when a specified confidence degree is reached, i.e., the choice of a plan is made just if there is enough information to trust in this choice.

Particularly, from this set of approaches that consider a confidence degree for plan selection, the work of Airiau et al. (2008), Airiau et al. (2009) and Singh et al. (), Singh, Sardina and Padgham (2010), Singh et al. (2011) can also be related to the process of planning. Their stated problem resembles to a planning problem that needs to determine a full course of action when we consider that these approaches need to know the entire structure of the goal-plan tree aiming to define a current confidence value of a plan to perform a suitable plan selection.

Furthermore, there are situations in which we want to specify to the agent a particular way in which a task may be accomplished and a good way to do it is declaring preferences. However, there is limited research work considering preferences in plan selection process as we can see in this related work overview.

The existing approaches present different techniques to represent preferences in the domain of the plan selection process. However, some issues need to be highlighted, such as the lack of consistency checks in the work of Visser, Thangarajah and Harland (2011), allowing the developer to specify conflicting preferences. Also, interactions between preferences are not explored, such as when satisfying a preference in a part of an execution can make impossible other preference to be satisfied. Padgham and Singh (2013) avoid this issue when they use logical expressions to specify preferences.

However, both of these approaches face the problem of needing to define and annotate preferences for every property and attribute of a given plan or goal, what becomes an expensive task in a large scenario. Nunes and Luck (2014), using the concept of *softgoals*, limit this number of preference definitions. However, they reduce the reliability of their approach when they give to developers the task of define the contribution of each plan for each softgoal based on fixed probabilities, regardless of the context. Chaouche et al. (2015) use a concept similar to softgoals to define preferences requiring developers to provide preferences over agent's strategies. However, they do not demand the expression of plan contributions, given that they are learned from past plan executions.

As we can see from this discussion and the previous chapters, considerable effort has been made addressing the improvement of plan selection process of BDI agents. However, despite all the effort made, they require much information to be provided and thus cannot be trivially used by mainstream software developers. All of the presented studies focus on a particular improvement. So, there is still an opportunity to develop and apply new concepts, or also merge these already existing concepts in a new one. Next section summarizes and concludes this chapter.

2.6 Final Remarks

The BDI architecture is a robust architecture proposed to deal with a domain composed of dynamics and complex applications that provides a flexible and intelligent behaviour. One of the main reasons for the flexibility and robustness of this approach is the plan selection process inherent of this architecture. This plan selection process is highly customisable so, many techniques have been developed to improve it.

Thus, this overview introduced developed work whose aim was to improve the plan selection process. These approaches use different techniques to endow BDI agents with the ability to learn how to behave best in a given situation, or how to select a plan that best satisfies agent preferences, applying concepts of machine learning and preference specification. There is a single approach that uses a different technique, based on the cost of a plan to perform the selection; however, it can be considered as a preference-based strategy where there is a high preference to reduce costs.

Based on the analysis of existing work, we concluded that there is still an opportunity to explore techniques and concepts related to the plan selection process in BDI agents, such as merging learning and preference specification in a single approach. Moreover, a concern that must be taken into account is the practical use of these approaches, reducing the amount of information required to be provided as input. This could make the approaches easier to be adopted by mainstream software developers, thus promoting the adoption of agent technology in large-scale in industry.

3 LEARNING AND PREFERENCE-BASED PLAN SELECTION

To achieve the full potential of the BDI architecture, it is necessary to specify an adequate plan selection algorithm. As presented in the previous chapter, several approaches have been proposed in this context, e.g. (GUERRA-HERNáNDEZ; FALLAH-SEGHROUCHNI; SOLDANO, 2004; VISSER; THANGARAJAH; HARLAND, 2011). Some of them focus on learning the context in which a plan possibly fails, thus increasing the chance that only plans that will succeed are executed. Others aim to identify the plan that best matches agent's preferences. A recent approach (NUNES; LUCK, 2014) not only considers agent's preferences, but also the uncertainty of possible plan outcomes regarding secondary goals (or *softgoals* (BRESCIANI et al., 2004)), such as minimisation of execution time, that are important to an agent. Nunes and Luck (2014) claimed that their approach requires little information about the domain to develop specific applications; however, it requires the specification of probabilities of each possible plan outcome, which are hard to elicit, may evolve over time, and may be context-dependent. Consequently, their required input may be considered *unrealistic* for real applications.

We present an approach that eliminates the need for the specification of such detailed information. In our approach, agents learn which are the expected plan outcomes according to particular contexts. Based on this, agents are able to *predict plan outcomes* based on the current context, information which is used to select a plan to achieve a goal, taking into account preferences that are expressed over agent's *softgoals*. Our approach comprises a meta-model, which must be instantiated to specify plan metadata, and a technique that uses such metadata to learn and predict plan outcomes, in order to select plans. Our approach thus extends existing work mainly by providing means of learning the required input of the plan selection algorithm, and simplifies agent design by requiring simple information as part of an agent design built based on our meta-model, thus hiding technique details usually unknown by mainstream software developers. This is a concern that directed our research and is essential to make the industrial adoption of the agent technology practically feasible, but is often left unaddressed by agent community.

Therefore, in Section 3.1, we briefly introduce concept definitions proposed by Nunes and Luck, which are adopted in our work. We also describe an example used throughout this chapter. Finally, we present our approach in Section 3.2, detailing the developed meta-model and technique.

3.1 Background and Running Example

The problem addressed by Nunes and Luck (2014) consists of choosing the best plan to achieve a goal, from a set of candidate plans. All these candidates are able to achieve the goal, but they have different side effects, which are associated with agent softgoals. This concept of softgoal was introduced by Bresciani et al. (2004) and its definition is presented below. In the work of Nunes and Luck (2014), their ultimate goal is to maximise agent satisfaction, considering agent's preferences over softgoals. Our goal is similar but, as said above, we do not require the specification of information that is difficult, if not impossible, to be provided. Given that our work is based on this existing work, we present next three definitions of these authors that we use.

Definition 1 (Softgoal) *An agent softgoal $sg \in SG$ is a broad agent objective, which cannot be achieved by a plan, but can achieve different degrees of satisfaction due to the effect produced by agent's actions that are part of plans.*

Definition 2 (Preferences) *Pref is a total function in $SG \rightarrow [0, 1]$ is a function that maps a softgoal $sg \in SG$ to a value indicating the preferences for softgoals. A preference is a value $pref_{val} \in [0, 1]$ that indicates the importance of a softgoal $sg \in SG$, 0 and 1 being the lowest and highest preference, respectively. Moreover $\sum_{sg \in SG} Pref(sg) = 1$.*

Definition 3 (Agent) *An agent is a tuple $\langle B, G, SG, P, Pref \rangle$ where B is a set of beliefs, G is a set of goals, SG is a set of softgoals, P is a set of plans, and $Pref$ is a preference function.*

We exemplify the concepts defined above in the scenario described next, which is in the context of logistic companies. This scenario also serves as a motivation for our work, and will be used to evaluate our approach. Logistic companies have, in the core of their business model, the action of *delivering* customer's *loads* from one place to another. Aiming to maximise its profits, a company with several *modes of transportation* must choose one, which better fits a given scenario.

A logistic company C has four options (or plans) to transport loads, which are delivering loads (i) by airplane (*AirplanePlan*); (ii) by ship (*ShipPlan*); (iii) by train (*TrainPlan*); and (iv) by truck (*TruckPlan*). An agent A is able to coordinate C, and must choose one of these available plans for each delivery request. Though all these plans can achieve the goal of *transporting a load from place x to place y* (for short, $transport(x, y)$), the choice

must take into account other factors that influence the decision. First, the rapidness to accomplish the goal may differ, e.g., transporting a load by airplane tends to be faster than by ship, if x is far from y. However, an airport can be closed due to bad weather condition, turning the other options more suitable in this context. Second, the cost of transportation varies across the different transportation modes, being influenced by a number of factors, such as the fuel consumption, vehicle's maintenance condition, among others. Third, it is possible that the load integrity is compromised, when vehicles are exposed to particular contexts. For example, when an airplane goes through turbulence, or when a truck drives through a road in poor condition.

In this scenario, all plans achieve the goal $transport(x, y)$. But this goal can be achieved in different amounts of time, with different costs, and be associated with different levels of reliability (related to load losses and damages) — these are plan outcomes. These are associated with different agent concerns, more specifically: maximise performance ($maxPerformance$), minimise costs ($minCosts$), and maximise reliability ($maxReliability$). These are softgoals, which are defined in Definition 1. Note that these softgoals are often conflicting, i.e. they cannot be maximised at the same time. Preferences (Definition 2), in turn, are the trade-off relationship between these softgoals. They express how much important each softgoal is. For instance, our agent needs to deliver a fragile load with extreme urgency. It may prefer to attempt a plan that offers a faster transport with highest reliability. Preferences that represent this scenario could be $maxPerformance = 0.4$, $maxReliability = 0.4$, and, $minCost = 0.2$. That is, the first two softgoals are equally important, and both are more important than the latter.

In summary, agent A is thus an entity able to execute plans in order to achieve its goal. It has the goal $transport(x, y)$; the plans, preferences and softgoals described above; in addition to beliefs, which comprise its knowledge. These are the five elements that comprise the definition of an agent, given in Definition 3. Considering this scenario, a question remains: how can agent A choose a plan, without requiring us to explicitly provide probabilities of plan outcomes?

3.2 Learning and Plan Selection

Next we present an overview of our approach to address the scenario described above. This approach has two main components, which are a meta-model and a technique to learn and predict plan outcomes, which are also detailed.

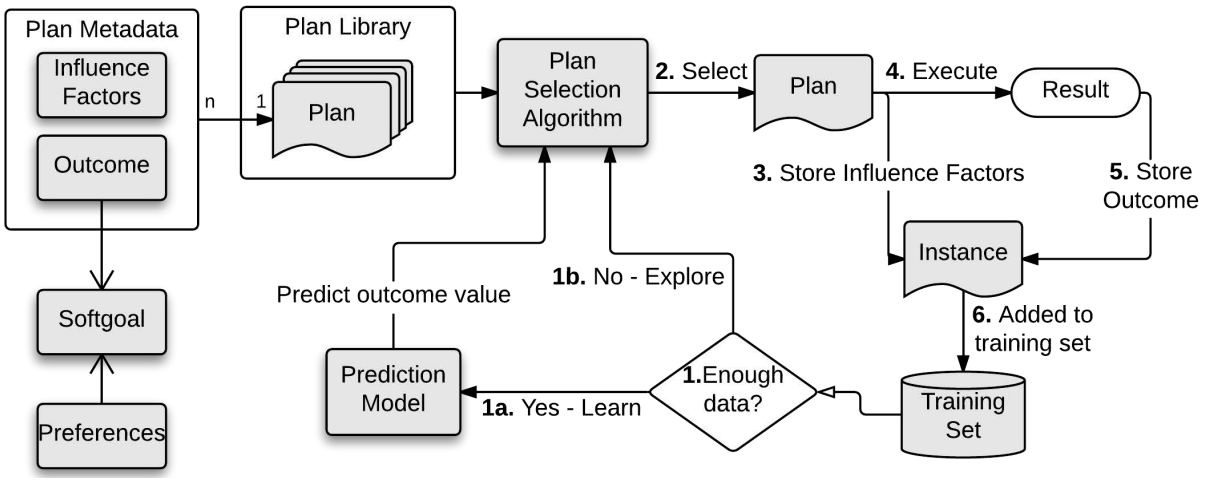


Figure 3.1: Approach Overview.

3.2.1 Approach Overview

In order to learn plan outcomes, our approach takes as input the factors that influence plan outcomes, e.g. the weather conditions may influence the time taken to deliver a load. Therefore, the key information that must be provided while modelling specific applications is this relationship between these factors and plan outcomes. Once this information is learned, it can be used to predict plan outcomes for a given context, and then the plan with the best outcomes is selected. We must highlight that, to provide the information required here, there is still a need for a domain-specific knowledge; however, the knowledge required is much less specialised.

Figure 3.1 shows the steps of our approach. Plans must be specified with additional metadata, which must specify the influence factors and outcomes for each plan. Moreover, such outcomes must be associated with softgoals; for example, load integrity can be associated with the *maxReliability* softgoal. In the BDI reasoning cycle, after agents generate goals to achieve, select those they are committed to achieve, and select a plan to achieve them; they execute a plan. When this happens, our approach records the current state of influence factors of the selected plan and, after the plan execution, it records plan outcomes. This allows us to collect the data needed to predict outcomes in the future.

When there is enough data collected, agents use them to build a prediction model. When a plan must be selected, plan outcomes are predicted using these data and current context. These expected outcomes are then transformed into utilities, associated with the satisfaction of softgoals. Next, utilities are combined using agent's preferences and, finally, the plan with the best utility (i.e. which maximises preference satisfaction) is selected. The execution of this selected plan is recorded as well, adding new data to improve the

prediction model. We specify the concepts associated with our approach in a meta-model described next, and then describe how we build and use our prediction model.

3.2.2 Meta-model

We briefly described above the main concepts of our meta-model. These concepts are those that need to be instantiated in specific applications that use our approach. Our meta-model specifies concepts to allow the representation of: (i) plan outcomes, and factors that influence them; (ii) a function that transforms plan outcomes into utilities; (iii) association of such utilities with softgoals; and (iv) preferences over softgoals. We represent the last with the function introduced in Definition 2. We begin by defining what we want to observe from plan executions, i.e. outcomes, as shown next.

Definition 4 (Outcome) *An outcome $o \in O$ is a measurement that can be taken during and/or after a plan execution. It is associated with a domain. Measuring an outcome from a plan execution gives us a* value *within this domain.*

As previously introduced, such outcomes are directly or indirectly affected by factors, that is, outcomes can be seen as an (unknown) function of these factors that influence them. Factors give the context information that a plan executes, which is a state that, in the BDI architecture, is given by beliefs. Consequently, in our meta-model, factors are mapped to agent beliefs. Influence factors are thus defined as follows.

Definition 5 (Influence Factor) *An influence factor $if \in IF$ is context variable, which is mapped to belief $b \in B$, and the value of which influences one or more plan outcomes $o \in O$.*

For instance, the traffic condition can influence the time that the *TruckPlan* takes to achieve its goal and, also, a vehicle in maintenance condition can affect the integrity of the load. In these cases, traffic condition and maintenance condition are influence factors (whose values are given by agent beliefs), and time and integrity are outcomes, respectively. An outcome is thus just value; therefore, we must provide means of specifying how such values satisfy agent softgoals. Currently, we assume that outcomes are numeric values, and agents have one of two objectives: either maximise or minimise an outcome value, i.e. the higher (in case of maximise) or the lower (in case of minimise) the outcome value, the more satisfied a softgoal. This is given by the definition below.

Definition 6 (Optimisation Function) *An optimisation function of* $\in OF$*, where* $OF = \{min, max\}$*, specifies whether the value of an outcome* $o \in O$ *must be minimised or maximised to better satisfy a softgoal* $sg \in SG$*.*

For example, the optimisation function should be *min* for the outcome *time* and softgoal *maxPerformance*, meaning that the lower the time, the more satisfied the *maxPerformance* softgoal. Now, we put all these defined concepts together in the definitions of a plan and plan metadata element.

Definition 7 (Plan) *A plan* $p \in P$ *is a tuple*

$$\langle Ctx, G', Body, MD \rangle$$

where Ctx *is the context conditions that specify when* p *is applicable,* G' *is a set of goals that can be achieved by this plan,* $Body$ *is a sequence of actions performed when the plan is executed, and* MD *is the set of plan metadata elements.*

Definition 8 (Plan Metadata Element) *A plan metadata element* $md \in MD$ *is a partial function*

$$md : SG \nrightarrow O \times \mathbb{P}(IF) \times OF \times Alg$$

which specifies an outcome, a set of influence factors an optimisation function and a machine learning algorithm/model, for a particular softgoal.

This completes the description of our meta-model. Next we describe how to use the information provided by this meta-model to improve the plan selection process.

3.2.3 Learning and Plan Selection

Our meta-model must be instantiated in particular applications to provide the input needed by our plan selection technique. This section shows how such inputs are used to build a prediction model, which is used to select plans. We assume that influence factors and outcomes are provided at development time, and are fixed at runtime; while softgoals, preferences over them, and optimisation functions can be provided at runtime, and evolve over time, but can be initialised at development time.

Given that our aim is to build a prediction model based on recorded plan executions, we first need an initial learning stage, where a minimal amount of data is collected to extract

Table 3.1: Example of Recorded Plan Executions.

	Influence Factors			**Outcome**
Execution	**Truck Conditions**	**Traffic Conditions**	**Road Conditions**	**Time Taken**
1	0.591	0.903	0.096	2.35
2	0.858	0.987	0.419	2.5
3	0.495	0.430	0.677	1.725
...	...	...	...	...

useful information. During this stage, plans may be selected randomly, for example. A threshold must be specified by developers to indicate when a prediction model must be built to select plans. For example, a minimum number of plan executions to achieve a particular goal may be provided.

As result of recording plan executions, we store collected information in the form of a table, for each plan, so as to form a *dataset*. An example of this table is shown in Table 3.1, which is associated with the *TruckPlan*. The first column of this table (*Execution*) is an identifier of the recorded plan execution, the second, third and fourth columns are influence factors, and the last column is an outcome (in hours). In this example, there is only one outcome, but it is possible to have others, associated with different softgoals. If there were other outcomes, influence factors of all of them would be recorded and be columns in this table. As previously mentioned, we are currently considering only numeric outcomes, so time is a continuous real number in this case. Influence factors, in this example, are given as a value between 0 and 1, which are the worst and best conditions, respectively. However, influence factors may be associated with any domain.

Influence factors are associated with beliefs, and collecting them means consulting the current value associated with beliefs in the agent belief base, which is easy. Outcomes, in turn, are expected to be collected from external sources through sensors, such as speedometers, timers, and so on. Therefore, in our approach, outcome is an abstract concept, and specific applications must provide means of measuring outcomes during or after plan execution. Some outcomes just need to be measured in the end of the plan execution (e.g. load integrity), but others require some preprocessing (e.g. measure the start and end times, and calculate the difference). Therefore, the outcome class of our implementation provides methods to start and end the measurement of an outcome (see Chapter 4), which are invoked by our plan selection strategy.

Given this collected data, we are now able to use existing machine learning algorithms to predict outcome values. The nature of our problem consists of understanding the relationship between a dependent variable (an outcome) and one or more independent variables (influence factors). Thus, any machine learning model able to perform regression can be used in our approach, e.g. linear regression (GALTON, 1894), support vector machines (SMOLA; SCHOELKOPF, 1998) and multilayer perceptron (RUMELHART; HINTON; WILLIAMS, 1986). Our approach is currently using linear regression, but better performance may be achieved when a learning algorithm is chosen taking into account the specificities of what is being learned. The predicted outcome value v_o using linear regression is thus given by $pred_{lr}(dataset(p), o)$, where $dataset(p)$ is a collected dataset for plan p, and o is an outcome.

When enough data is available, a prediction model is thus built for each outcome, because each has its particular influence factors. Then, in order to select a plan, we use such models to predict outcomes according to the current context. After learning plan outcomes based on the current context, there are two challenges left that need to be addressed: (i) how to convert predicted outcomes to an agent utility; and (ii) how to combine these utilities and select a plan.

Predicted plan outcomes are values of the outcome domain. The conversion of this outcome domain to a utility value, $\mathbb{T}(v_o, o)$, ranging from 0 (worst) to 1 (best), is made using the optimisation function and domain boundaries (i.e. maximum and minimum values). Such domain boundaries are used to normalise the value to a scale ranging from 0 to 1, which are the minimum and maximum domain values, respectively, if the optimisation function is *max*. If the optimisation function is *min*, domain boundaries are used in the opposite way. For example, the outcome value 2.35 hours for the time outcome is converted to 0.530, because the domain boundaries are 0 and 5, and time is associated with the *min* optimisation function. If no domain boundaries are available, we use the minimum and maximum values across datasets.

The selected plan is that with the *highest utility*. In order to combine utilities of individual outcomes, we use a *weighted sum* model, given that agent's preferences can be used as weights that establish the trade-off relationship among softgoals, and each of which is associated with an outcome. This sum is also possible as all utilities of individual outcomes are in the same scale. Consequently, the plan selected to achieve a goal g is the plan that satisfies the following equation from all candidate plans to achieve g, where o_{sg}

is the outcome associated with the softgoal sg.

$$\arg\max \sum_{sg \in SG} Pref(sg) \cdot \mathbb{T}(pred_{lr}(dataset(p), o_{sg}), o_{sg})$$

3.3 Final Remarks

An plan selection process able to understand the influence of context on plan outcomes is essential to achieve the full potential of the BDI architecture. Existing work dealing with this process requires input that may be considered unrealistic for real applications. Therefore, we presented an overview of our approach, which extends current work by providing means of learning the required input of the plan selection algorithm. Moreover, we described the main concepts that comprise our meta-model and need to be instantiated to use our approach. Finally, we detailed a technique that uses our instantiated meta-model for selecting plans.

In this chapter, we provided the conceptual solution for providing agents with learning characteristics. Next, we present its practical implementation.

4 BDI4JADE EXTENSION AND THE *SAM* TOOL

One of the main objectives of this work is to provide not only theoretical foundation but also demonstrate that our approach has practical applicability. As a consequence, we contribute to the promotion of the use of this agent technology in industry. Therefore, we implemented the meta-model and technique presented in Chapter 3 as an extension of the BDI4JADE platform (NUNES; LUCENA; LUCK, 2011). We also developed a tool to support the model-driven development of agents endowed with learning capabilities. Thus, Section 4.1 details the implementation of our approach, while Section 4.2 presents the *Sam* tool.

4.1 Implementation

This section presents an extended BDI4JADE platform to allow developers to practically use our technique of learning-based plan selection. BDI4JADE[1] is a framework that implements a BDI layer on top of JADE[2]. It is purely Java-based, which makes agents implemented with it able to be extended and be integrated with different technologies currently in use, such as AspectJ[3], Hibernate[4], among others.

Figure 4.1 shows main components of our implementation in a class diagram. This implementation was used to run the simulation presented in the evaluation chapter. Classes in dark grey are those already existing in BDI4JADE, while the remaining classes are those implemented by us. Classes in light grey are those that are transparent to developers, while the unfilled ones are those they will explicitly manipulate. Moreover, some classes, namely `SoftGoal`, `NamedSoftGoal`, and `SoftGoalPreferences`, were reused from the extension available in the BDI4JADE website (NUNES; LUCK, 2014).

We encapsulated the plan selection technique in the `selectPlan()` method of the `LearningBasedPlanSelectionStrategy` class. Initially, plans are selected randomly, on an exploration process that aims to build the initial knowledge about plan outcomes. Every plan will be selected and executed several times until they reach a minimum number of executions specified by the developer.

This process of collecting new data from plan executions involves two main classes:

[1] <http://inf.ufrgs.br/prosoft/bdi4jade>
[2] <http://jade.tilab.com/>
[3] <https://eclipse.org/aspectj/>
[4] <http://hibernate.org/>

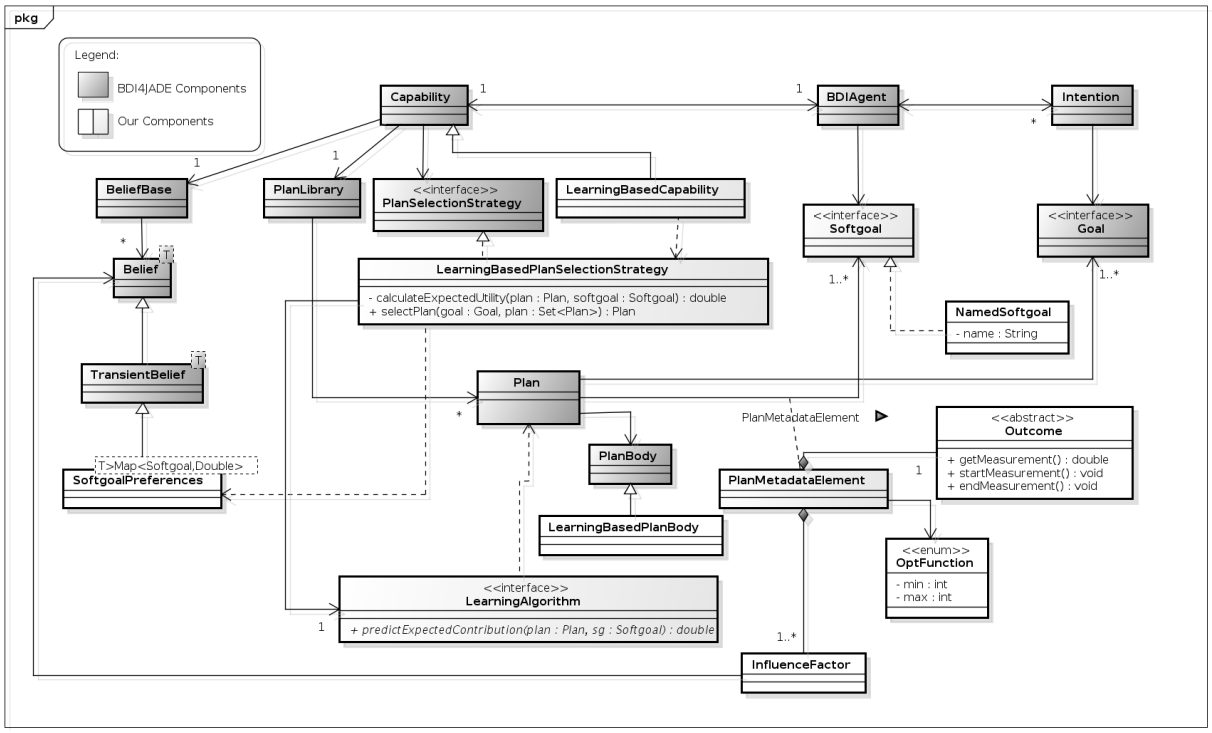


Figure 4.1: Meta-model as a BDI4JADE Extension.

the `InfluenceFactor` and the `Outcome` classes. The first provides the current status of the agent beliefs, while the second informs the impact of a plan execution in a given data related to a particular softgoal. Given that influence factors can be seen as abstractions of beliefs, the `InfluenceFactor` class is implemented as a map to these elements. Thus, our plan selection strategy can use the information from these influence factors without the need for accessing the agent's belief base.

Outcomes, in turn, must provide external data. As explained in Chapter 3, they are expected to collect data from sensors, such as thermometers, speedometers, and so on. Therefore, in our implementation `Outcome` is an abstract class, which developers must extend. Some outcomes require to be measured only at the end of the plan execution (e.g. a resource consumption) while others need also to do some preprocessing (e.g. to measure the start and end times, and calculate the difference). Thus, we provided methods to start and end the measurement of an outcome (`startMeasurement()` and `endMeasurement()`), which are invoked by our plan selection strategy right before and right after a plan is executed, respectively.

Therefore, when a plan is selected, our algorithm collects, for each softgoal, the current state of the influence factors. The measurement of outcomes occurs right after the plan is executed. Collected information are stored in a particular file, which relates to the plan and softgoal that this data refers to. Thus, for each softgoal, a plan has a specific file.

When a plan reaches the minimum number of executions specified by the developer, its collected information is used in the plan selection process. As we defined in our technique, the agent must be able to predict the values of the outcomes of each plan in a given context to define a contribution value to them. In our implementation, we use Weka[5] to build a prediction model for each outcome of a plan. Weka is a tool that provides a collection of machine learning algorithms to be used in data mining tasks. It provides an API that allowed us to use it directly in our code.

Developers can define the machine learning algorithm that will be used to build a prediction model. The only restrictions are that this algorithm must be able to deal with regression problems (given that the outcomes are expressed numerically), and must exist on the set of algorithms provided by Weka. From the values provided by prediction models and the preferences defined for each existing softgoal, our plan selection algorithm can assign a utility value to a plan. Then, the algorithm selects the plan with the highest expected utility. Moreover, each prediction model created is updated when a plan reaches a specific number of executions defined by developers. We referred to this number as the *learning gap*.

Now, we show how to use the extended BDI4JADE implementation to provide an agent with learning capabilities. Most of the implementation of our technique become invisible to developers. The main task they need to accomplish is to correctly relate influence factors and outcomes to a plan metadata element, and then relate these elements to a particular plan. Moreover, developers must provide specific code for outcomes, implementing its access to external devices or sensors that provide required data.

Initially, an agent is created as a standard BDI4JADE agent and a capability extending the `LearningBasedCapability` class must be added to it. The `LearningBasedCapability` is responsible for defining which plan selection strategy the agent will use, in our case the `LearningBasedPlanSelectionStrategy`. Every relationship between elements comprising our meta-model is made into this capability.

Outcomes, in turn, are created as extensions of the `Outcome` class. Moreover, every plan has its plan body, which must be an extension of the `LearningBasedPlanBody` class, allowing us to trigger the methods that perform outcome measurements.

Figure 4.2 shows a code snippet from a project where the agent `ExampleAgent` is implemented. In this project, `ExampleAgent` has the capability `ExampleCapability`, which allows this agent to accomplish the goal `ExampleGoal`. Two plans can deal with

[5]<http://www.cs.waikato.ac.nz/ml/weka/>

```
21    @Belief
22    private TransientBelief<String, Double> belief01 = new TransientBelief<String, Double>(
23            "BELIEF01", 0.0);
24    @Belief
25    private TransientBelief<String, Double> belief02 = new TransientBelief<String, Double>(
26            "BELIEF02", 0.0);
27    @Belief
28    private TransientBelief<String, Double> belief03 = new TransientBelief<String, Double>(
29            "BELIEF03", 0.0);
30
31    @Plan
32    ExamplePlan01 plan01 = new ExamplePlan01();
33    @Plan
34    ExamplePlan02 plan02 = new ExamplePlan02();
35
36    public ExampleCapability() {
37        init();
38    }
39
40    private void init() {
41
42        Map<Softgoal, PlanMetadataElement> metadata = new HashMap<Softgoal, PlanMetadataElement>();
43
44        PlanMetadataElement planMetadataElement = new PlanMetadataElement(plan01, Softgoals.MINCOST,
45                new Outcome01(), OptimizationFunction.MINIMIZE,
46                LinearRegression.class, 50, 100);
47        planMetadataElement.addInfluenceFactor(new NumericInfluenceFactor(belief01));
48        planMetadataElement.addInfluenceFactor(new NumericInfluenceFactor(belief02));
49        metadata.put(Softgoals.MINCOST, planMetadataElement);
50
51        planMetadataElement = new PlanMetadataElement(plan01, Softgoals.MAXPERFORMANCE,
52                new Outcome02(), OptimizationFunction.MAXIMIZE,
53                MultilayerPerceptron.class, 50, 100);
54        planMetadataElement.addInfluenceFactor(new NumericInfluenceFactor(belief02));
55        planMetadataElement.addInfluenceFactor(new NumericInfluenceFactor(belief03));
56        metadata.put(Softgoals.MAXPERFORMANCE, planMetadataElement);
57
58        plan01.putMetadata(PlanMetadataElement.METADATA_NAME, metadata);
```

Figure 4.2: Code Using the Extended BDI4JADE Implementation.

this objective, namely `ExamplePlan01` and `ExamplePlan02`. Moreover, when pursuing its goal, `ExampleAgent` has two softgoals it wants to satisfy: *minCost* and *maxPerformance*. They are affected by `Outcome01` and `Outcome02`, respectively, which are influenced by some factors related to agent's beliefs.

Lines 44–48 show how an instance of `Outcome01` and its influence factors are related to a plan metadata element. The constructor of the `PlanMetadataElement` class receives the plan to which the given outcome corresponds, the softgoal that this outcome relates to as well as the outcome itself. It also requires developers to provide the optimisation function value (here implemented as an enumeration comprising the values *MINIMISE* and *MAXIMISE*), a reference to the learning algorithm class chosen, the minimum number of executions and the learning gap.

Therefore, developers must inform influence factors that affect the given outcome. Lines 47 and 48 show the addition of two influence factors, which are mapped to beliefs `belief01` and `belief02`, respectively. Finally, the plan metadata element created is inserted into a general metadata set (line 49) which, in turn, is assigned to a particular

belief that comprises all metadata of this plan(line 58).

This coding process is repeated for every softgoal in each plan, as we can observe in lines 51–56, where a plan metadata element corresponding to `ExamplePlan01` and the *maxPerformance* softgoal is created and added to the general metadata set. Next, we present *Sam*, a tool developed to support the development of agents using our implementation.

4.2 *Sam*: a tool to support the development of software agents

To create BDI agent models from the meta-model defined in Chapter 3, we introduce a notation that represents the different elements comprising our approach. In the centre of Figure 4.3, we present how each component is graphically represented. Agents and capabilities are represented by circles with solid and dashed lines, respectively, containing its names. A belief is depicted as a rectangle containing a stereotype `«belief»` and the belief's name. Outcomes have the same representation of beliefs, but with the stereotype `«outcome»` instead. Goals are represented as rounded rectangles containing a label that displays the goal's name, while a softgoal is represented by a shape that resembles a cloud containing the softgoal's name. Finally, plans and plan metadata elements are depicted as a hexagon with solid and dashed lines, respectively.

Relationships between the components of our meta-model are also presented. We considered the existence of two types of relationships: containment and association. A containment relationship occurs, as its name says, between an element that contains another, e.g. an agent that has a belief. Association relationships, in turn, refers to an element that relates to another without containing it, e.g. a plan that relates to a goal. These relationships are illustrated as directional lines connecting a source and a target elements, with an open arrowhead near the target. Containment relationships are characterised by the use of a solid line, while association relationships are depicted as a dashed line connecting the elements. Moreover, we observe that optimisation functions are represented by a labelled association relationship, which shows the name of the given optimisation function.

This introduced notation is then used to create diagrams that provide a complete view of a BDI agent. Figure 4.4 presents the `ExampleAgent` introduced in Section 4.1 modelled through a diagram. As can be seen, plan metadata elements, in addition to a label displaying a generic name, present also a list of labels, which represent the softgoal,

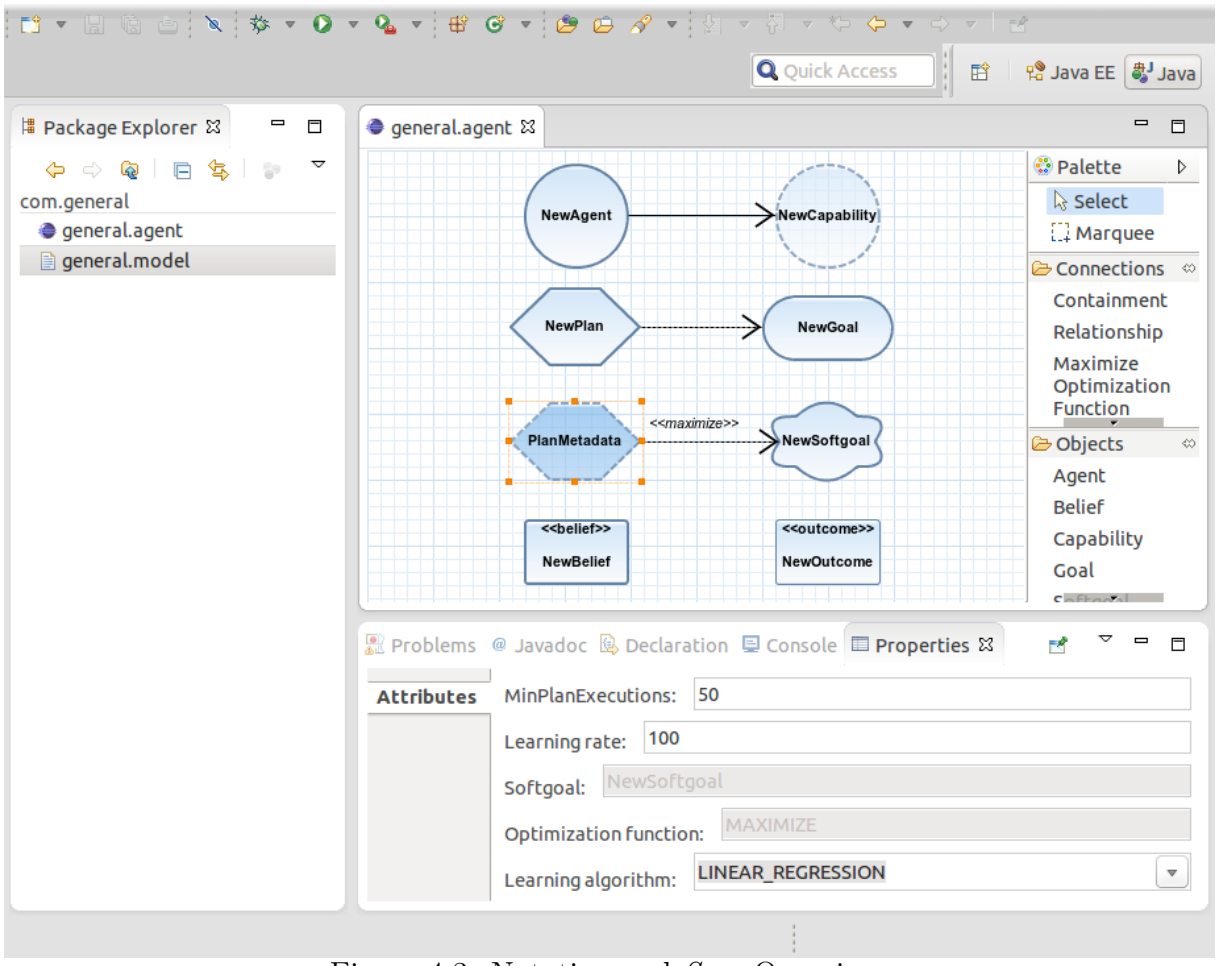


Figure 4.3: Notation and *Sam* Overview.

optimisation function and influence factors related to the given plan metadata element. Plans, in turn, present their name and the name of the goal it relates.

We not only specify the notation to use our approach to model BDI agents but also provide its implementation as a tool, namely *Sam*, to support agent development. Figure 4.3 also shows the general appearance of our tool. *Sam* is a graphical editor developed to provide a tool-supported way of modelling and generating code for agents with learning capabilities. It is implemented as a plug-in for Eclipse[6] and is built upon a framework called Graphiti[7].

We briefly describe the environment the tool provides. The main element of *Sam*'s user interface is the editor view. This editor displays the graphical representation of a modelled agent. The information related to a modelled element is stored in diagram and model files, related to graphic and domain-specific information, respectively. A user can navigate these files through the package explorer, located on the left-hand side of the editor view.

[6] <http://www.eclipse.org>
[7] <http://www.eclipse.org/graphiti/>

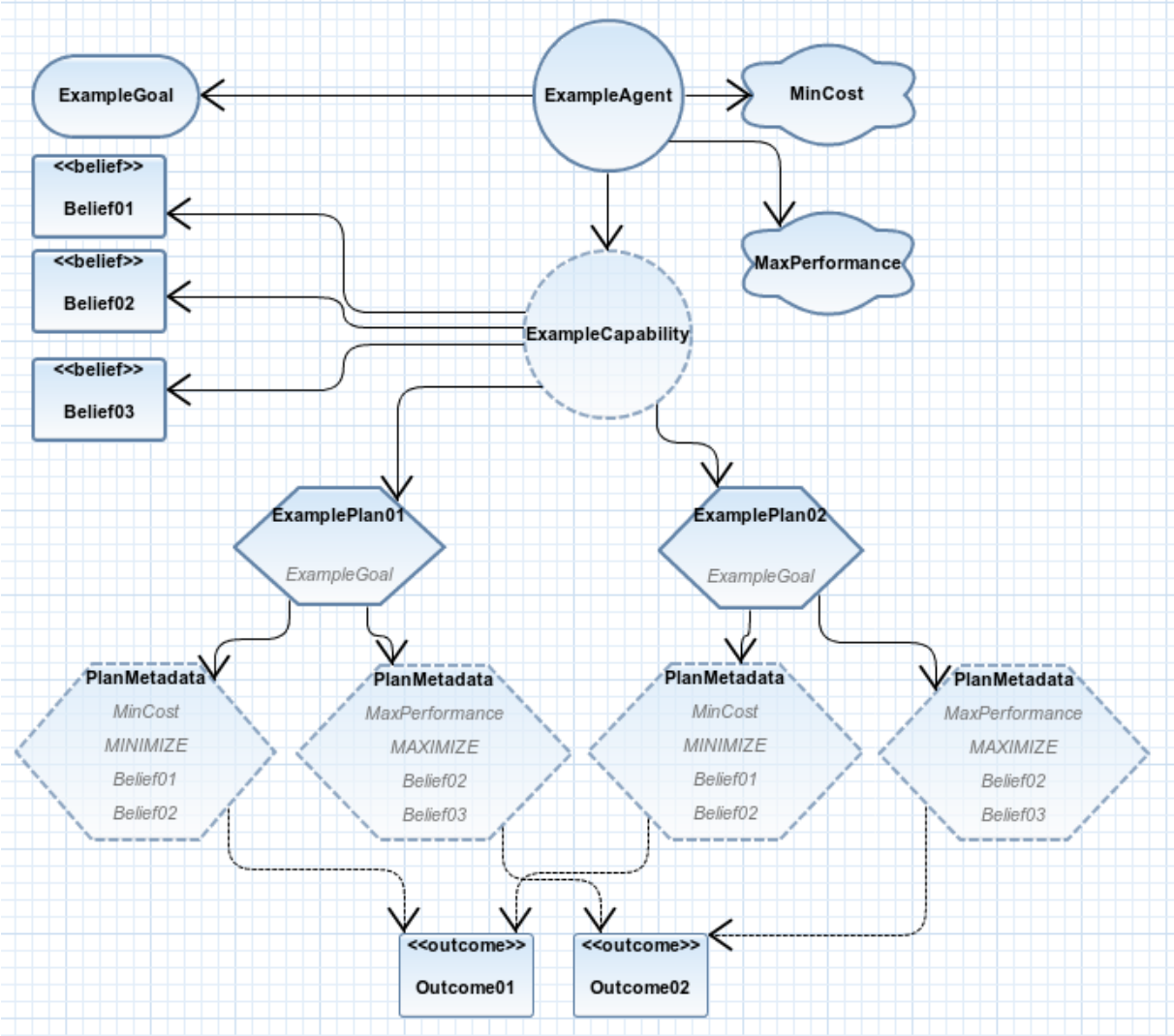


Figure 4.4: `ExampleAgent` Modelled in *Sam*.

On the right-hand side of the editor view stands a creation palette, which contains the elements that can be instantiated in an agent diagram.These elements, when instantiated, can have their properties changed through a property view, positioned below the editor view. Moreover, some specific information required to our plan selection strategy works are informed through this property view, e.g. preferences for softgoals and the minimum number of executions, learning gap and the machine learning algorithm for plan metadata elements.

Sam has also features to improve diagram readability. It allows the user to choose between showing the goal's name inside a plan element or explicitly displaying a connection between these elements. This same feature, which we referred to as *collapse feature*, is applied for a plan metadata element, which can display its connections to beliefs and softgoals or show this information inside its representation.

Moreover, more than just modelling, *Sam* allows generating code for agents based on

the implementation previously presented. After creating a diagram with a valid model from our meta-model, users can access the model file through the package explorer, open a context menu and select the option *Generate code*. This triggers a process that generates the complete structure of packages and classes, including relationships represented on the model. Users have to provide only specific code for plan bodies and outcomes, given that this code is domain-specific. This code-generation feature was implemented using a template language called XPand[8]. Figure 4.5 depicts the different technologies integrated to develop our tool.

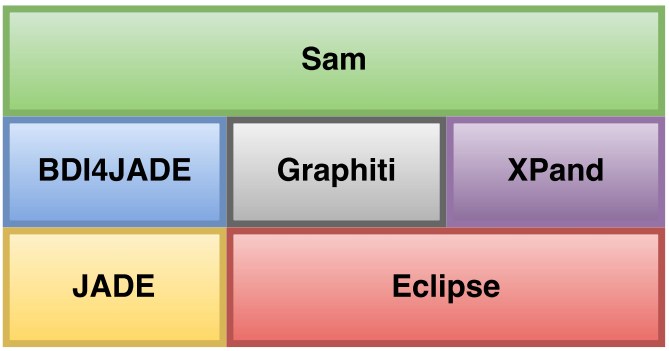


Figure 4.5: Integration of the Technologies Used to Develop *Sam*.

In short, *Sam* works as a tool for the design and implementation of software agents endowed with learning capabilities. We must highlight that the notation selected for representing some of the BDI components in our tool is based on the Tropos notation (BRESCIANI et al., 2004), in order to be consistent with existing BDI concepts. Moreover, this tool still being improved and features such as model checking are under development.

4.3 Final Remarks

More than provide theoretical foundations, we aim to demonstrate that our approach has practical applicability. Therefore, we presented our meta-model and technique implemented as an extension of the BDI4JADE platform, detailing its operation, providing technical information about its development and introducing how developers can use it to create an agent able to learn and select plans according to its preferences. Moreover, we introduced *Sam*, a tool we created to support the development process of agents based on our implementation. We described the notation used to represent elements and relationships in diagrams created with *Sam*, as well as provided an overview of our tool. The implementation and tool described in this chapter are evaluated next.

[8] <http://wiki.eclipse.org/Xpand>

5 EVALUATION

Now that we presented our meta-model with its concepts and how they are used to endow an agent with a learning capability to perform an improved plan selection, as well as an implementation of this technique and a tool to support the agent development, we evaluate these contributions. The technique is evaluated with an empirical approach in Section 5.1. Section 5.2 describes a user study performed to assess the impact of the *Sam* tool and its underlying approach in agent development.

5.1 Evaluation of the Plan-selection Approach

Although much work considers the use of learning techniques in the plan selection process, they all differ from our approach in the subject of learning. Furthermore, some of them focus on identifying applicable plans, i.e. learning when plans fail (AIRIAU et al., 2008; SINGH et al., ; SINGH et al., 2011), while in our approach we select the best of a set of plans, which are applicable to the current context (which is given). In our approach, other plans are still candidates in case this purported best plan fails. Therefore, because the goal of these approaches differs from ours, it is not possible to compare it with them. The most similar approach to ours is that of Nunes and Luck (2014), but their approach requires a different and more complicated input — actually, our approach learns the input that they require to be explicitly provided, and be fixed at runtime. We then compare our approach to a random plan selection. The procedure of our evaluation is presented in Section 5.1.1, and its results and discussion are presented in Sections 5.1.2 and 5.1.3, respectively.

5.1.1 Procedure

To evaluate our approach we implemented the scenario presented in Chapter 3. This scenario involves an agent A with the goal $transport(x, y)$, which means transporting a load from place x to y. As introduced before, to achieve this goal our agent has four possible plans: *AirplanePlan*, *ShipPlan*, *TrainPlan*, and *TruckPlan*. Moreover, the agent has three softgoals to consider, namely *maxPerformance*, *minCosts*, and *maxReliability*. In addition, each plan is associated with its metadata, specifying influence factors, an

outcome, and an optimisation function, for each softgoal. For example, the *TruckPlan* has the influence factors *Truck Conditions*, *Traffic Conditions* and *Road Conditions* and the outcome *Time Taken*, and is associated with the *min* optimisation function, with respect to the *maxPerformance* softgoal. Similar metadata are specified for each softgoal in each plan. Table 5.1 presents these details.

Our experiment consists of measuring the agent satisfaction (i.e. obtained utility) produced using different plan selection strategies. For this purpose we ran a simulation, in which we performed the following steps in each iteration: (i) randomly generate preferences for each softgoal; (ii) randomly instantiate a current context, and agent beliefs are adapted accordingly; (iii) predict outcomes for each plan; (iv) select a plan using a plan selector (our algorithm or randomly); and (v) measure and store the satisfaction of agent's preferences of the plan execution.

Satisfaction of agent's preferences is measured by transforming the results of a plan execution (outcome values) to utility. The result of a plan execution (e.g. actual time taken) was generated randomly, using a normal distribution parameterised with arbitrary average and standard deviation. These average and standard deviation were generated using a function based on the values of the influence factors of the outcome considered. In order to ensure unbiased results, we ran our simulation with different functions, and they all performed similarly. Furthermore, the threshold for building the prediction model and using it was set to 50 plan executions. This prediction model was updated with different update-rates (i.e. the prediction model is rebuilt after a specified number of plan executions), and results for each are reported next.

5.1.2 Results and Analysis

After running 5000 iterations of steps listed above, we compared the accumulated satisfaction for each plan selector. Besides a random plan selector, we used plan selectors using our approach, updating the prediction model after each plan execution (*LB-1*), 100 plan executions (*LB-100*), 500 plan executions (*LB-500*), and 1000 plan executions (*LB-1000*). The average satisfaction (*M*), standard deviation (*SD*), minimum (*Min*), maximum (*Max*) and accumulated satisfaction (*Cum Sat*) for each plan selector are presented in Table 5.2, with highest values highlighted in bold and lowest ones in italics; while Figure 5.1 shows the box plot of the agent satisfaction obtained for each plan selector. We also compared the accumulated satisfaction of each plan selector, which is depicted

Table 5.1: Plans and Respective Metadata

Plan	Softgoal	Influence Factor	Outcome
AirplanePlan	minCosts	AirplaneConditions	FuelConsumption
		WeatherConditions	
		Distance	
	maxPerformance	AirplaneConditions	TimeTaken
		WeatherConditions	
		AirportConditions	
	maxReliability	AirplaneConditions	LoadIntegrity
		AccidentProbability	
		ChanceOfTheft	
ShipPlan	minCosts	ShipConditions	FuelConsumption
		WeatherConditions	
		Distance	
	maxPerformance	ShipConditions	TimeTaken
		WeatherConditions	
		SeaConditions	
		HarborConditions	
	maxReliability	ShipConditions	LoadIntegrity
		AccidentProbability	
		ChanceOfTheft	
TrainPlan	minCosts	TrainConditions	FuelConsumption
		WeatherConditions	
		Distance	
	maxPerformance	TrainConditions	TimeTaken
		TrafficConditions	
		RailroadConditions	
	maxReliability	TrainConditions	LoadIntegrity
		AccidentProbability	
		ChanceOfTheft	
TruckPlan	minCosts	TruckConditions	FuelConsumption
		TrafficConditions	
		Distance	
	maxPerformance	TruckConditions	TimeTaken
		TrafficConditions	
		RoadConditions	
	maxReliability	TruckConditions	LoadIntegrity
		AccidentProbability	
		ChanceOfTheft	

Table 5.2: Satisfaction by Plan Selector ($n = 5000$).

Plan Selector	**M**	**SD**	**Min**	**Max**	**Cum Sat**
LB-1	0.6812	0.091	0.219	0.961	3406.04
LB-100	0.6815	0.091	**0.262**	0.968	3407.99
LB-500	0.680	0.092	0.214	**0.982**	3402.68
LB-1000	**0.6816**	*0.090*	0.138	0.969	**3408.00**
RAN	*0.597*	**0.114**	*0.102*	*0.941*	*2988.98*

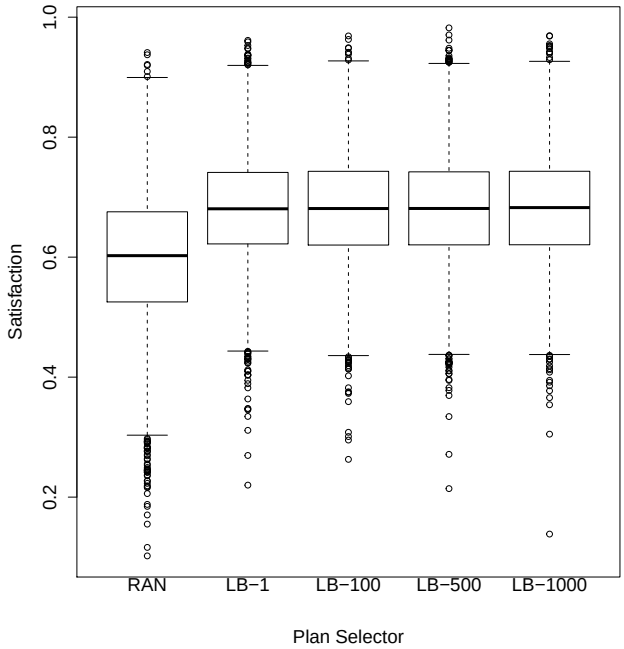


Figure 5.1: Analysis of Satisfaction by Plan Selector.

in Figure 5.2.

Considering the best satisfaction value as 1.0, the obtained satisfaction may seen not extremely high (the best average satisfaction is 0.6816); however, we need to highlight that the *uncertainty* associated with plan executions is still playing its role, preventing results for being always accurately predicted. For example, bad weather means that there is a high probability of delay, but it is not guaranteed that there will actually be delay. Moreover, the plan selection process involves a choice where *trade-off* must be resolved, that is, some plans are better with respect to certain softgoals and others are better with respect to other softgoals — e.g. resolve a trade-off between transportation time and cost. Consequently, no plan leads to the best satisfaction value, and 0.6816 can be considered a high value.

Analysing our results, it can be observed that our learning-based plan selector always

performs better than the random plan selector, independently from the selected update-rate. We highlight that, although it is not possible to see in the presented charts, the results for all plan selectors are similar in the first iterations, given that there is a need for obtaining an initial dataset for building prediction models. LB-1000 has slightly better results than those obtained with LB-1, LB-100 and LB-500. However, these results do not differ significantly among them (see Figure 5.1). The *small* differences between the plan selectors using our approach with different update-rates are expected, because after building a model with enough data to learn an adequate model, the behaviour of all plan selectors tend to be the same. However, if there were an event in the system that would impact in the relationship between influence factors and outcomes, plan selectors with lower update-rates would faster react to these changes.

A one-way ANOVA was used to test for preference differences among satisfaction of each plan selector. Satisfaction of plan selectors differed *significantly* across the five selectors, $F(4, 24995) = 747.8$, $p \ll .05$. Post-hoc Tukey's HSD (TUKEY, 1949) tests showed that comparisons between all learning-based plan selectors and the random plan selector were significantly different at .05 level of significance. With respect to performance, all simulations using plan selectors based on our approach execute in around 10 seconds, except LB-1, which executes in approximately 6 minutes. This performance from LB-1 was expected, given the need for frequently updating prediction models.

Some may argue the random plan selector is not a reliable baseline for evaluating our approach. However, as argued before, we cannot make a direct comparison with other approaches since they focus on learning in which context plans fail. Nunes and Luck's approach is the most similar to ours, but requires an input often unavailable for real applications (their baseline is also a random plan selector). Consequently, given the novelty of our approach, it was impossible to evaluate it against existing approaches.

5.1.3 Discussion

Our simulations indicate that our approach can significantly increase agent satisfaction by selecting plans that will likely satisfy more agent's preferences. Based on our evaluation, we discuss important issues. Although we do not constrain the value types of influence factors, we are currently considering only numeric continuous values. We may notice that, in order to use discrete values for an influence factor, we should know in advance all possible values it can have. In addition, we give the same importance to all

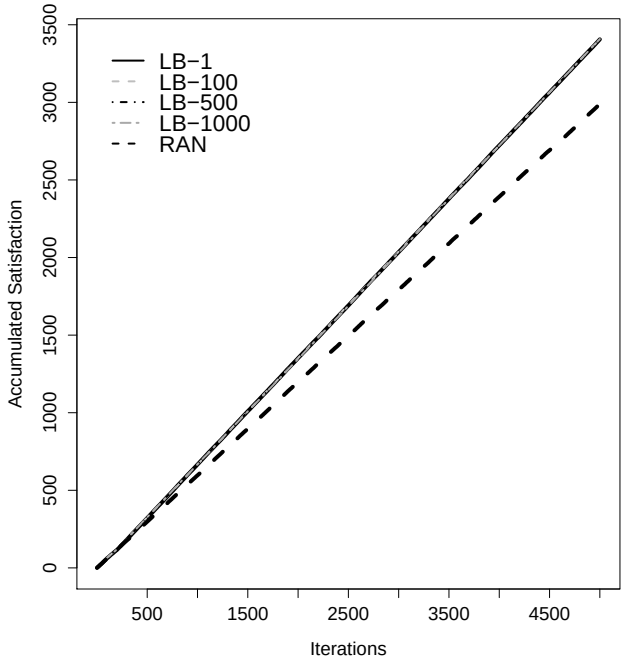


Figure 5.2: Accumulated Satisfaction by Plan Selector.

records in our dataset, but it may be interesting to give more weight to recently collected records, mainly in dynamic environments.

One may say that online learning, such as Q-learning, is a more adequate solution to our problem. Note, however, that here we make the reward independent from the plan outcome, and there are different plan outcomes for each plan. In this way, using the same prediction model, we are able to deal with agent's preferences and optimisation functions that evolve over time. We are aware that our present form of transforming plan outcomes to agent utility is simple (i.e. minimise or maximise values), but it is already effective and applicable to many scenarios. It is part of our future work to explore other forms of transformation functions, leveraging preference elicitation approaches (KEENEY; RAIFFA, 1976). Next section presents and evaluation of our tool.

5.2 Evaluation of Support to Agent-based Development

With the design and implementation of the *Sam* tool, we aimed to achieve two key goals: (i) reduce the effort required to develop BDI agents; and (ii) improve the development quality. Thereby, we conducted a user study to verify the benefits provided by *Sam* in the development process of an agent, specifically concerning project understandability and development time. We made a comparison between our tool and using source code

only. Next, Section 5.2.1 presents the study settings detailing the procedure followed in the experiment, while Section 5.2.2 shows the obtained results and analysis.

5.2.1 Study Settings

Two main activities comprise this user study, each of them regarding one specific concern — project understandability and development time. We assigned two groups of volunteers to perform these activities with different resources. Both groups used BDI4JADE extended with our meta-model and technique; however, only one of them used *Sam*. From these activities, we extracted metrics that helped us to measure the benefits of our tool. Also, we evaluated the *Sam*'s usability by requesting participants that were assisted by the tool to answer a usability questionnaire. Now we detail each aspect of this user study.

5.2.1.1 Goal and Research Questions

To model and develop this experiment we used the Goal-Question-Metric (GQM) approach (BASILI; CALDIERA; ROMBACH, 1994). The GQM is a traditional method of software metrics planning. Its purpose is to identify *metrics* that effectively contribute to answer specific *questions*, which in turn are related to a *goal*. Following this approach, we define the statement bellow as the goal for our user study.

> *To assess the improvements provided by a tool-supported BDI-agent-based development method, evaluate the effectiveness of the use of the* Sam *tool to understand and evolve BDI-agent-based systems from the perspective of the researcher in the context of graduate and undergraduate students in Computer Science.*

There are several ways in which a tool can aid the software development process, e.g. saving time, improving code quality, reducing costs, etc. considering this we limited the scope of our experiment in the context of two main benefits: the improvement of project understandability and coding effectiveness. The first one concerns how well developers can understand an existing project, i.e., how well they can correctly identify different components and relationships throughout the code. The second benefit, in our context, refers to improvements in code quality and time to implement such code. Thus, considering this context and aiming to achieve the goal stated above, we defined our research questions as follows.

? Research Questions

RQ1 *Does the use of* Sam *tool facilitate understandability of existing BDI-agent-code?*

RQ2 *Does the use of* Sam *improve the evolution of existing BDI-agent-code?*

Based on these questions, we can define the most appropriate metrics to be collected. To answer **RQ1**, we defined two metrics in the context of an existing agent project. They are: (**M1.1**) the number of correctly answered questions about the code; and (**M1.2**) the time taken to correctly answer these questions. In addition, three metrics are used to answer **RQ2**. These metrics are: (**M2.1**) time taken to evolve the code according to a particular task; (**M2.2**) number of compilation errors of modified code; and (**M2.3**) number of logical errors of the modified code. Details of each metric are presented as follows. Figure 5.3 depicts the relationships among our research questions and metrics.

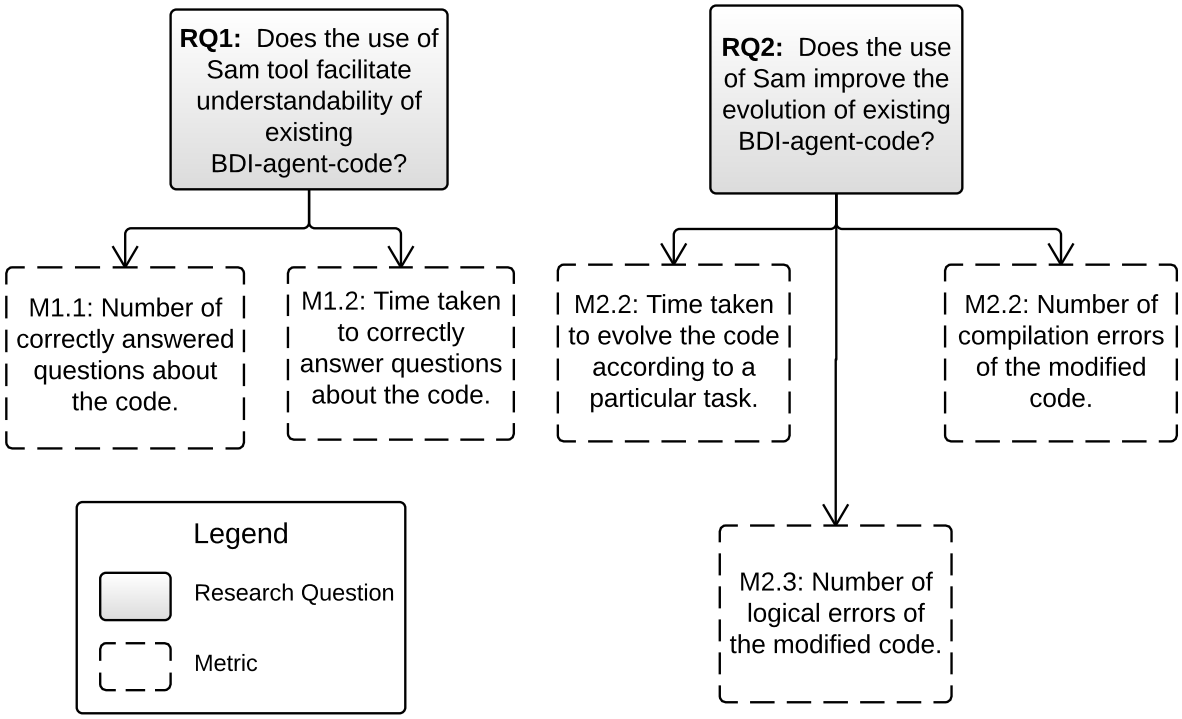


Figure 5.3: Research Question and Metrics Relationship.

5.2.1.2 Procedure

To evaluate our tool, we performed a five-step study. In summary, we asked participants to answer a questionnaire regarding their experience in areas related to our experiment, then submitted they to an introductory class, giving an overview of different concepts needed as background to perform our study. After, participants performed

activities regarding code understanding and evolution, with or without the assistance of our tool. Participants who were assisted also answered a usability questionnaire. These steps are detailed next.

Step 1: Experience Survey. Participants were requested to answer a questionnaire regarding their previous knowledge in programming, the Java programming language, software agents and the BDI architecture. Thus, from their answers, we classified each participant as beginner, intermediary or expert. In each question we assigned a weight for existing alternatives. This weight relates to the level of knowledge expressed by a given alternative and is proportional to the amount of alternatives in the question. For instance, a question with three different alternatives would have weights 0.0, 0.5 and 1.0 for each option, respectively, given that the alternatives are sorted in ascending order of expressed knowledge. Therefore, we calculated the arithmetic average from answers and assigned a final score to each participant. Participants with a score $<= 0.35$ were classified as beginners, while those with score > 0.65 were classified as experts. Participants whose score was in the interval between beginners and experts were classified as intermediaries. The complete questionnaire is presented in Appendix A and the demographic data from participants is presented in Section 5.2.1.5.

Step 2: Introductory Class. We split participants in two groups and submitted them to an introductory class, which aimed to ensure that all participants had the same basic knowledge regarding concepts and approaches addressed by this experiment. Therefore, they had contact with concepts such as software agents, the BDI architecture, our meta-model and technique as well as the extended BDI4JADE implementation; thus, being able to recognise influence factors, plan metadata elements and other related elements. Moreover, one of the groups had a a brief overview of using *Sam* for modelling and generating source code for agents based in our approach. This group was able to use *Sam* during the entire experiment while the other one had access to source code only.

Step 3: Understanding Activity. This activity consists of a questionnaire about an implemented target system. The purpose of this questionnaire is to provide metrics to answer the first research question (**RQ1**), thus verifying the existence of any difference concerning code understandability when analysing a project from its source code, or assisted by our tool. Both groups performed this activity being able to access the material provided in the introductory class. We recorded the time taken for each participant to answer each question as well as their responses, and extracted the corresponding metrics (**M1.1** and **M1.2**).

Step 4: Code Evolution Activity. The second activity is an exercise of code evolution. In this activity we used an initial implementation of a target system as the base project, which must be evolved in certain aspects, aiming to provide metrics to answer the second research question (**RQ2**). Again, both groups were able to use the material from the introductory class. For each participant, we recorded the evolved project and the time taken to develop the task, from where we extracted metrics **M2.1**, **M2.2** and **M2.3**.

Step 5: Usability, Satisfaction and Ease of Use Questionnaire. Finally, we asked the group that performed the previous activities using *Sam* to answer a questionnaire comprising questions regarding tool's usability, satisfaction provided to the user and ease of use. This questionnaire allows us to identify aspects that can be improved to deliver an instrument that provides a better experience to developers.

We present the target systems implemented to support the understanding and code evolution activities in Section 5.2.1.3. Details of the questionnaires used in steps 3 and 5, and the exercise of code evolution performed in step 4 are presented in Section 5.2.1.4.

5.2.1.3 Target Systems

In order to support the execution of this study, we implemented two different target systems. The first one, which is referred to as *transportation system*, was used in the third step of our experiment, and is similar to that presented in the evaluation shown in Section 5.1. In this target system an agent T has the goal of $transport(x, y)$, i.e. transporting a load from its origin in x to its destination in y. To achieve this goal agent T can choose among three available plans: *AirplanePlan*, *TruckPlan* or *ShipPlan*. Each plan has its specific metadata, which provides information about influence factors and outcomes, and how they relate to each agent's softgoal, namely *maxPerformance*, *minCosts* and *maxReliability*. Figure 5.4 shows this system modelled in *Sam*.

The second implementation represents a *sorting system*, whose objective is to sort elements from a given array. An agent A with the goal $sort(x)$ is, thus, responsible for managing this system and selecting the best way of sorting the array x. Agent A can initially choose among two plans, each of them representing one particular sorting algorithm: *InsertionSortPlan* and *SelectionSortPlan*, representing the insertion sort and selection sort algorithms, respectively. This target system is the base for the code evolution activity. Figure 5.5 depicts the sorting system modelled with *Sam*. Observe that in that model we refer to agent A's goal as *SortArray*. The same occurs in Figure 5.4 where agent T's goal is presented as *Transport* instead of $transport(x, y)$.

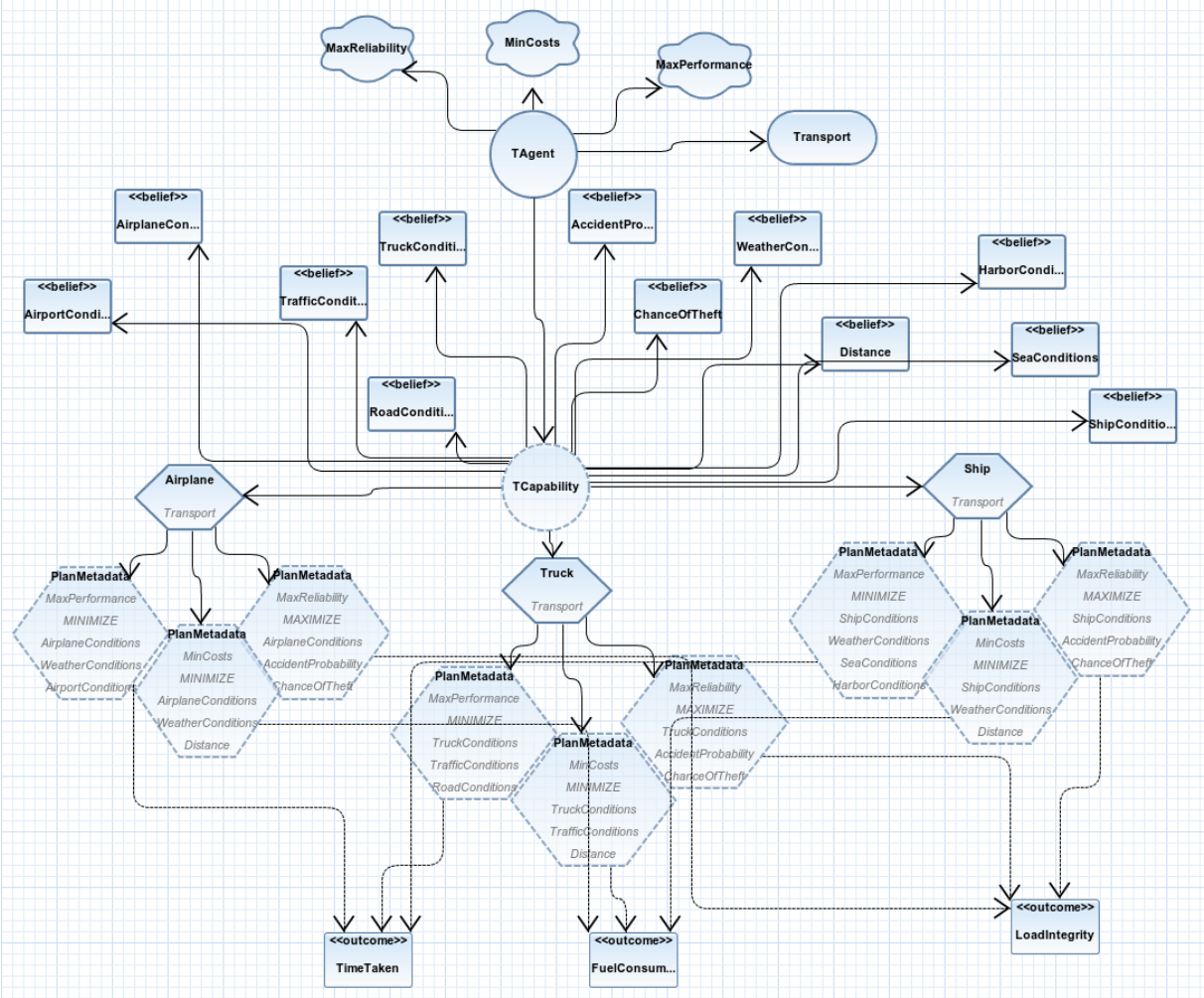


Figure 5.4: The Transportation System Modelled in *Sam*.

5.2.1.4 Questionnaires and Exercise

The questionnaire associated with the understanding activity consists of 12 questions about the project implemented regarding the transportation system. These questions aim to ensure how well a developer understands the project and are directly associated with the elements related to our approach, i.e., outcomes, influence factors, optimisation functions, and their relationships. There are four different types of questions, each of them asking about specific relationships and following a particular template. These question templates are presented next.

- Which outcome is associated with the S softgoal in the P plan?
- Which is the optimisation function used in the P plan, with respect to the S softgoal?
- Which beliefs are influence factors of the O outcome in the P plan?
- Which influence factors and outcome are associated with the S softgoal in the P plan?

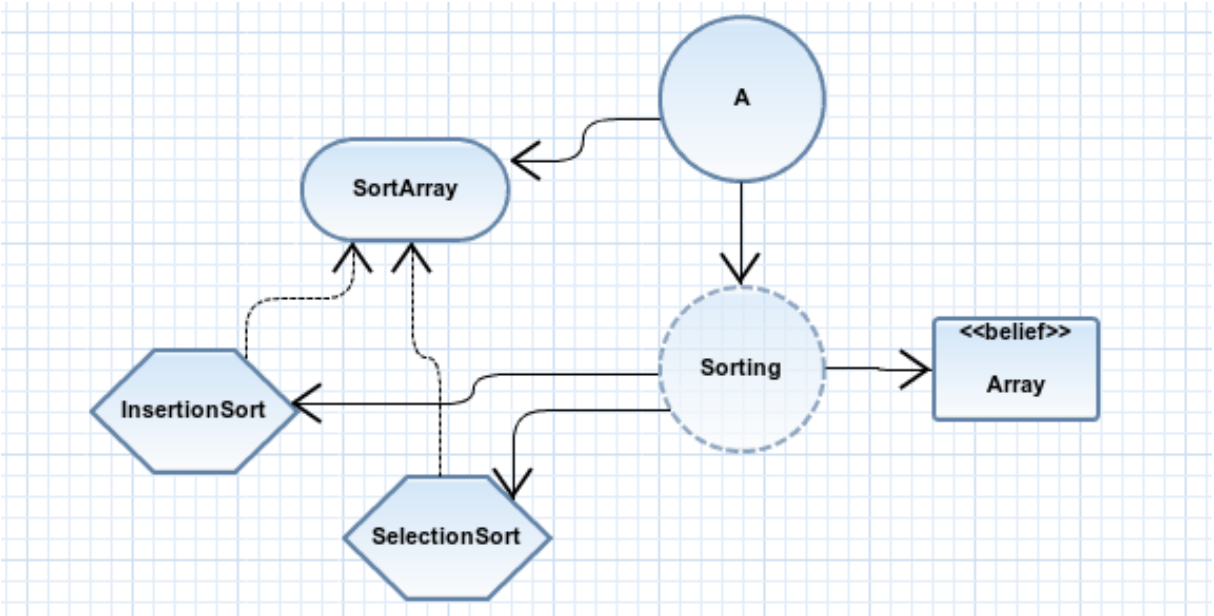


Figure 5.5: The Sorting System Modelled in *Sam*.

From these templates we derived our questions, replacing S, P and O by different softgoals, plans and outcomes, respectively, of the transportation system. A concrete question is, for instance, "*Which outcome is associated with the maxPerformance softgoal in the AirplanePlan plan?*" The questions were interleaved in a way that participants do not answer the same type of question in a sequence, thus, requiring them to learn how to interpret a model or code to give the correct answer. The questionnaire used with its explanation text and the complete list of questions is presented in Appendix A.

The code evolution activity considers the ability of a developer to evolve an agent-based system and how it can be improved when assisted by a suitable tool. This exercise provided to the participants an initial Java project that implements the sorting system already presented. From this project they should evolve the initial code, providing an agent with the learning capability described in our approach.

Moreover, we requested them to add some elements present in our meta-model, which would support the learning capability. For instance, participants were asked to add the *maxPerformance* and the *minNumberOfSwaps* softgoals to the agent. Thus, we asked them to set the metadata for each plan, creating and relating elements according to a given specification. Tables 5.3 and 5.4 presents the metadata that must be added to *InsertionSortPlan* and *SelectionSortPlan* respectively. The complete explanation text of this exercise is also shown in Appendix A.

Finally, the participants who performed the understanding and code evolution activities assisted by *Sam* were asked to answer a final questionnaire. This questionnaire is an

Table 5.3: Metadata for the Insertion Sort Plan

Influence Factors	**Outcome**	**Optimisation Function**	**Softgoal**
ArraySize, Additions, Removals	CPUTime	Minimise	maxPerformance
ArraySize, Additions, Removals	NumberOfSwaps	Minimise	minNumberOfSwaps

Table 5.4: Metadata for the Selection Sort Plan

Influence Factors	**Outcome**	**Optimisation Function**	**Softgoal**
ArraySize, Additions, Removals	CPUTime	Minimise	maxPerformance
ArraySize, Additions, Removals	NumberOfSwaps	Minimise	minNumberOfSwaps

adaptation of the USE (Usefulness, Satisfaction, and Ease of use) questionnaire (DAVIS, 1989; LUND, 2001) and is composed of 26 seven-point Likert rating scales and two open-ended questions. For each question the participant must assign a score ranging from 1 (strongly disagree) to 7 (strongly agree) to a given statement regarding one of the three dimensions addressed by the USE questionnaire. Moreover, we requested participants 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 is presented in Appendix A. Next, we describe our group of participants.

5.2.1.5 Participants

The participants of our study were all volunteers, graduate and undergraduate students in Computer Science. This user study initiated with a total of 26 participants, from which 7 were not present for step 3 of our study. From those remaining, one participant was not present for step 4. Therefore, 19 participants performed the understanding activity, while 18 were present for the code evolution activity. We included answers from all participants who performed a given activity in results presented in Section 5.2.2.

Table 5.5 presents the demographic characteristics of the participants that completed the study, while Table 5.6 shows how participants were distributed into groups according to their classification as beginners, intermediaries and experts.

Table 5.5: Demographic Characteristics.

Characteristic		**N**	**%**
Education	Undergraduate student	5	27.8
	Master Student	11	61.1
	PhD Student	2	11.1
Programming Experience (in years)	< 1 year	0	0.0
	1 – 2 years	1	5.6
	2 – 5 years	9	50.0
	5 – 10 years	3	16.7
	> 10 years	5	27.8
Experience with Java programming language	0 - None	0	0.0
	1	0	0.0
	2 - Minimal experience	1	5.6
	3	0	0.0
	4 - Some experience	4	22.2
	5	3	16.7
	6 - Substantial experience	5	27.8
	7	3	16.7
	8 - Extensive experience	2	11.1
Knowledge about software agents	0 - Fundamental awareness	5	27.8
	1	5	27.8
	2 - Novice	5	27.8
	3	2	11.1
	4 - Intermediary	0	0.0
	5	0	0.0
	6 - Advanced	1	5.6
	7	0	0.0
	8 - Expert	0	0.0
Knowledge about the BDI architecture	0 - Fundamental awareness	12	66.7
	1	2	5.6
	2 - Novice	3	16.7
	3	2	11.1
	4 - Intermediary	0	0.0
	5	0	0.0
	6 - Advanced	0	0.0
	7	0	0.0
	8 - Expert	0	0.0

Table 5.6: Distribution of Participants

	Before Dropouts		**After Dropouts**	
	Tool	**Code only**	**Tool**	**Code only**
Beginner	6	5	5	4
Intermediary	6	5	5	3
Expert	0	1	0	1

5.2.2 Results and Analysis

Based on the data collected while executing the steps of our procedure, we analysed them to answer our research questions. First, we compare the performance of both groups regarding the understanding activity, which provided metrics to answer research question **RQ1**. Then, we analyse data collected from the code evolution activity aiming to answer research question **RQ2**. We also discuss the results from the USE questionnaire, which was the last step of our experiment. The results and analysis from these activities are presented next.

5.2.2.1 RQ1: Does the use of Sam *tool facilitate understandability of existing BDI-agent-code?*

In order to answer our first research question, we analysed metrics **M1.1** and **M1.2** collected from the understanding activity. Metric **M1.1**, which corresponds to the number of correctly answered questions about the code, was analysed from two perspectives: (i) consideration of partially correct answers; and (ii) consideration of completely correct answers only. The first perspective considers the fact that each question from the questionnaire associated with this activity has a specific answer composed of different elements. Therefore, if at least one of these elements is correctly mentioned in an answer, it is possible to consider this answer as being partially correct. The second perspective, in turn, assumes that an error in an answer invalidates the entire answer. It relates to the idea that one mistake in a project can lead to a software bug or a misconception of the entire system.

Therefore, we assigned a score for each participant considering both perspectives. Regarding partially correct answers, we calculated this score by dividing the number of correctly provided elements by the sum of all elements in an answer – hits and misses. We considered a miss a wrong element in the answer or the absence of a correct one. For instance, in a question whose correct answer is *AirplaneConditions, WeatherConditions and AirportConditions* and the participant answered *AirplaneConditions, WeatherConditions and ChanceOfTheft*, we considered two correct elements and two misses. Thus, the score assigned to this participant in this question would be equals to *2 / (2 + 2) = 0.5*. Table 5.7 shows the scores for each question as well as the general score (GS) from participants of both groups. T_n and C_n represent participants of the group using tool and participants of the group using source code only, respectively. This table also presents a

summary with average (M) and standard deviation (SD) for each group. Moreover, we highlight correct and partially correct answers by colouring its background.

Analysing the scores for each question, we notice that developers assisted by our tool tend to have a better understanding of a project when confronting it for the first time. The difference presented between the averages in Q1 supports this statement. Although the difference between groups decreases as participants answer more questions and become familiar to the project, the general score of the group using *Sam* is still higher than that of the group using source code only. Moreover, the increasing scores for questions of the same type, e.g. the sets Q1, Q5, Q9 and Q2, Q6, Q10, show that participants learn the project and the tool/code they are using and handling. This behaviour, which was already expected, also explains the decreasing in differences between groups scores.

We used a Shapiro-Wilk test to check for normality in partially correct scores and results indicate that our samples do not follow a normal distribution ($p < .05$). Therefore, a Mann-Whitney U test was conducted to determine whether there was a difference in total scores considering partially correct answers from participants assisted by tool and participants using code only to perform this activity. Results of that analysis indicated that there was not a difference statistically significant, $z = -0.7569$, $p > .05$.

Regarding the perspective of analysing these results considering completely correct answers, participants received an score of 1 for a correct answer, and 0 otherwise. Table 5.8 presents the scores for each participant as well as a summary with average (M) and standard deviation (SD) of each group. We also highlight correct answers.

From this perspective, we observe that the group assisted by our tool still presenting a better performance regarding its initial comprehension of the given project, while the difference between averages becomes more apparent in some questions. We also notice the same learning behaviour presented previously; however, it is remarkable the sudden growth of the learning curve presented by participants using *Sam*. We must highlight the set of questions Q3, Q7 and Q11, where the average score increases from 0.55 to 1.00 (almost duplicates) and remains the same, i.e. the maximum, in the last question of this type. These results indicate that participants using our tool are able to learn a project easier than those using source code only, and the errors made in the initial question are possibly due to the understanding of how the tool works.

Figures 5.6a and 5.6b summarises scores from participants of each group considering both perspectives, using a box diagram. Analysing the correctness of provided answers, we observed that although the median of general scores of participants of the code group is

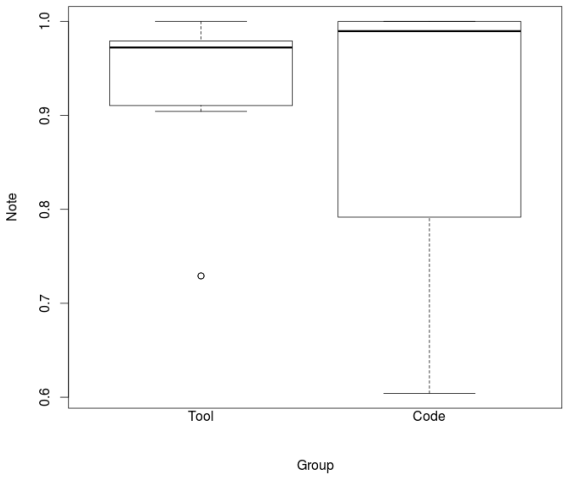
Table 5.7: Scores by Participant (Partially Correct Answers)

Participant	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**Q12**	**GS**
T1	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.92
T2	1.00	1.00	0.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.75	0.92
T3	1.00	1.00	0.67	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97
T4	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.75	0.98
T5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T6	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.75	0.98
T7	1.00	1.00	0.67	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97
T8	1.00	0.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00	0.00	1.00	0.75	0.73
T9	1.00	1.00	0.60	0.75	1.00	1.00	1.00	0.75	1.00	1.00	1.00	0.75	0.90
T10	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T11	0.00	0.00	0.50	0.50	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.75	0.73
M	**0.91**	**0.73**	**0.80**	**0.93**	**1.00**	**0.91**	**1.00**	**0.98**	**1.00**	**0.91**	**1.00**	**0.86**	**0.92**
SD	**0.30**	**0.47**	**0.25**	**0.16**	**0.00**	**0.30**	**0.00**	**0.08**	**0.00**	**0.30**	**0.00**	**0.13**	**0.10**
C1	0.00	0.00	0.50	0.75	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	0.60
C2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.75	1.00	1.00	1.00	1.00	0.98
C3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
C4	0.00	1.00	0.50	0.75	0.00	1.00	1.00	0.75	0.00	1.00	1.00	0.75	0.65
C5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
C6	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
C7	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
C8	0.50	1.00	1.00	0.75	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.94
M	**0.69**	**0.88**	**0.88**	**0.91**	**0.75**	**0.88**	**1.00**	**0.94**	**0.88**	**1.00**	**1.00**	**0.97**	**0.90**
SD	**0.46**	**0.35**	**0.23**	**0.13**	**0.46**	**0.35**	**0.00**	**0.12**	**0.35**	**0.00**	**0.00**	**0.09**	**0.17**

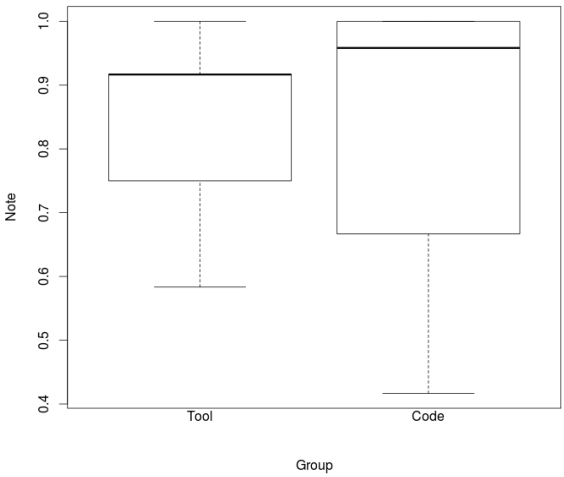
Table 5.8: Scores by Participant (Completely Correct Answers)

Participant	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**Q12**	**GS**
T1	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.92
T2	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.83
T3	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.92
T4	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.92
T5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T6	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.92
T7	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.92
T8	1.00	0.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00	0.00	1.00	0.00	0.67
T9	1.00	1.00	0.00	0.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00	0.00	0.67
T10	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T11	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.58
M	**0.91**	**0.73**	**0.55**	**0.82**	**1.00**	**0.91**	**1.00**	**0.91**	**1.00**	**0.91**	**1.00**	**0.45**	**0.85**
SD	**0.30**	**0.47**	**0.52**	**0.40**	**0.00**	**0.30**	**0.00**	**0.30**	**0.00**	**0.30**	**0.00**	**0.52**	**0.14**
C1	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	0.50
C2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	0.92
C3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
C4	0.00	1.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	1.00	0.00	0.42
C5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
C6	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
C7	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
C8	0.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.83
M	**0.63**	**0.88**	**0.75**	**0.63**	**0.75**	**0.88**	**1.00**	**0.75**	**0.88**	**1.00**	**1.00**	**0.88**	**0.83**
SD	**0.52**	**0.35**	**0.46**	**0.52**	**0.46**	**0.35**	**0.00**	**0.46**	**0.35**	**0.00**	**0.00**	**0.35**	**0.24**

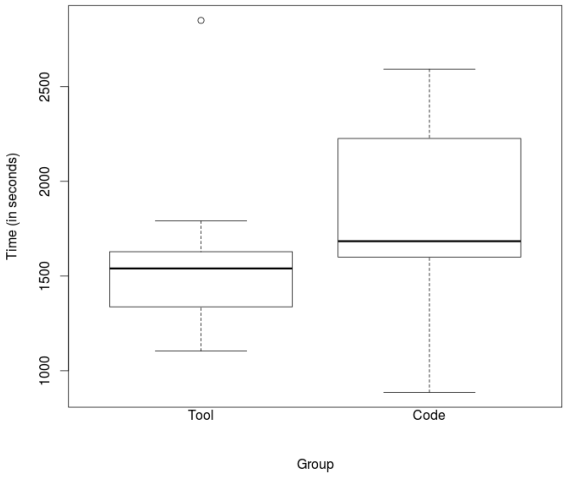
higher with respect to understanding, the variance of this group is also much higher – note that the average of the code group is lower. We further analysed why this larger variance occurred, and we concluded that experienced developers were able to correctly answer questions, while beginners made many mistakes. With our tool, on the other hand, most of the participants achieved similar good results. Consequently, there is evidence that our tool can help project understanding when developers are not experts or are unfamiliar with the technology.



(a) Partially Correct Answers



(b) Completely Correct Answers



(c) Analysis of Total Time by Group

Figure 5.6: Collected Measurements by Group

We used a Shapiro-Wilk test to check for normality in completely correct scores and results indicate that our samples do not follow a normal distribution ($p < .05$). Therefore, a Mann-Whitney U test was conducted to determine whether there was a difference in

total scores considering completely correct answers from participants assisted by tool and participants using code only in an understanding activity. Results of that analysis indicated that there was not a difference statistically significant, $z = -0.5119$, $p > .05$.

We also analysed results collected from metric **M1.2**, which corresponds to the time taken to correctly answer the questions from the understanding activity. Table 5.9 shows the time taken for each participant to answer each question, as well as their total time to finish this activity. Similarly, participants from the group assisted by our tool are represented by T_n, while participants using source code only are represented by C_n. However, this table presents the time to answer a question even if the answer is wrong. Thus, we highlighted the time taken to correctly answer a question with a light grey background. Times for partially correct questions are presented with a dark grey background, while cells containing times of wrong answers remain with a white background. Therefore, we present a summary of the time taken for each group to *correctly* answer a question, showing average (M) and standard deviation (SD) for each question. Averages and standard deviations for total time were omitted, given that participants may have a distinct number of correct answered questions, which makes impracticable to compare these values.

Analysing the results provided by this metric, we notice that our tool also allows participants to correctly understand a project faster than those using source code only. This benefit is observed in the first questions, with a considerable difference between groups. However, this difference disappears over time, which we consider an effect of the participant's learning process.

Moreover, regarding the total time to perform the understanding activity, we observe a tendency of participants assisted by *Sam* to do it faster than those from the code group. We can notice this tendency observing Figure 5.6c, which shows an analysis of the total time taken (in seconds) from participants. More than presenting a lower median time, the group using our tool presents a much lower variance. An analysis of this difference between groups indicates that also beginners are able to quickly understand how to use and read the information provided by *Sam*, which results in fast and correct responses.

We performed a Shapiro-Wilk test to check for normality in total time and results indicate that our samples follow a normal distribution ($p > .05$). Therefore, an independent-samples t-test was conducted to compare total time taken for participants from both groups. There was not a significant difference in the times for participants assisted by tool ($M = 1589.6$, $SD = 464.9$) and using code only ($M = 1736.2$, $SD = 528.2$); $t(12) = 0.5786$, $p > 0.05$.

Table 5.9: Times by Participant in Understanding Activity

P	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Total
T1	04:01	07:54	03:08	04:07	00:41	00:37	01:05	01:19	00:23	00:16	00:54	01:15	25:40
T2	05:31	01:40	02:51	01:10	00:50	00:47	00:45	01:28	00:39	01:03	00:52	00:48	18:24
T3	12:00	02:46	01:55	05:07	00:45	01:09	00:42	01:40	00:36	01:22	00:34	01:15	29:51
T4	15:29	04:42	01:42	01:01	00:28	00:23	00:29	01:16	00:17	00:13	00:34	00:40	27:15
T5	04:13	01:08	08:31	02:01	01:30	00:32	04:20	01:25	00:33	00:17	00:53	01:37	27:02
T6	04:56	01:51	07:35	03:10	02:02	00:45	01:24	00:55	00:32	00:23	00:57	01:28	25:57
T7	04:45	01:54	02:39	03:07	00:31	00:30	01:27	01:20	00:47	00:26	01:12	01:16	19:54
T8	08:20	01:21	02:03	01:18	00:41	00:49	03:17	01:30	00:28	01:07	00:44	01:03	22:39
T9	08:13	06:02	03:23	01:37	00:29	01:14	01:38	00:41	00:23	00:19	00:39	00:40	25:18
T10	11:19	01:09	02:07	01:47	00:21	00:27	00:32	00:56	00:19	01:02	01:04	00:52	21:56
T11	09:13	08:02	10:17	04:56	00:45	01:53	02:21	03:22	01:13	00:43	02:12	02:33	47:30
M	07:53	02:39	04:11	02:32	00:49	00:50	01:38	01:31	00:34	00:37	00:58	01:15	–
SD	03:56	01:47	03:03	01:26	00:30	00:28	01:14	00:41	00:16	00:25	00:27	00:16	–
C1	29:13	04:55	00:21	03:29	00:35	00:46	00:37	00:42	00:26	00:36	00:41	00:53	43:12
C2	17:24	01:43	08:30	02:59	00:29	00:22	00:58	00:43	00:39	00:28	00:46	01:04	36:03
C3	10:27	01:48	07:35	01:34	00:38	00:43	00:45	01:16	00:45	00:46	00:53	01:15	28:25
C4	12:43	01:23	07:06	00:30	00:54	00:41	02:30	00:24	00:20	00:14	00:24	00:26	27:34
C5	12:54	01:59	03:35	02:50	00:38	00:43	00:58	00:56	00:19	00:19	00:31	00:44	26:26
C6	04:36	02:28	04:49	01:34	00:19	00:38	00:49	01:13	01:05	00:31	00:44	01:08	16:54
C7	07:28	00:49	00:57	00:56	00:34	00:29	00:48	00:51	00:22	00:17	00:35	00:41	14:46
C8	11:09	10:46	06:58	01:04	01:34	00:32	00:56	01:12	01:04	00:33	00:29	01:52	38:10
M	10:34	02:59	04:54	01:59	00:42	00:35	01:03	01:01	00:40	00:28	00:38	01:05	–
SD	04:56	03:28	03:12	00:54	00:27	00:08	00:36	00:14	00:19	00:11	00:10	00:24	–

Table 5.10: Summary of Metrics From the Project Evolution Activity

	Tool-assisted		**Code-only**	
	M	**SD**	**M**	**SD**
M2.1	00:37:58	00:17:52	00:42:39	00:17:14
M2.2	01.50	03.24	00.00	00.00
M2.3 – Quantity	14.20	04.73	06.00	02.72
M2.3 – Type	05.00	02.45	03.12	01.46

In summary, evidence suggests that the use of *Sam* tool facilitates the understanding of existing BDI-agent-code from both perspectives: correctness and time. Participants assisted by our tool were able to correctly identify elements and their relationships in a target system implementing our approach, with a better performance regarding time than those using source code only.

5.2.2.2 RQ2: Does the use of Sam *improve the evolution of existing BDI-agent-code?*

After performing the code evolution activity, which consists of evolving an initial project based on the sorting system presented before, we analyse the results. Table 5.10 summarises the results of metrics collected from this activity, presenting averages (M) and standard deviation (SD) for each of them.

Analysing metric **M2.1**, which represents the time taken to perform this activity, we observe that participants assisted by *Sam* were faster than those using source code only. On average, the former spent almost 38 minutes to evolve the sorting system, while the latter used approximately 43 minutes. This difference makes explicit a benefit of evolving a model instead of code. The use of models allows developers to focus on the specific task they are performing, without being distracted by implementation details of a given language. Therefore, once one learns how to correctly model an agent in *Sam*, it becomes trivial to have a project working.

Metric **M2.2** provides the number of compilation errors on an evolved project. In our context, a compilation error relates to a code snippet that does not correspond to the grammar of a given programming language, thus resulting in an error in the compiler.

Considering the results, participants working with code only performed better than those assisted by our tool. We must highlight the lack of errors from the group using source code only. We assume that this result occurs because participants from this group work directly with the code, easily perceiving the appearing of this kind of error and

correcting it immediately because of the used IDE, which automatically compiles the code and highlight compilation errors. Moreover, the compilation errors that appeared on projects from the group assisted by *Sam* were not located in the code generated by our tool, but in code snippets modified by the participants.

We also measured the number of logical errors that emerged in evolved projects (**M2.3**). A logical error relates to a code snippet that is correct from the perspective of a given programming language grammar, but presents logical faults such as the lack of objects or wrong relationships between elements. We analysed this metric from two perspectives, regarding the total number of occurrences and the number of different types of logical errors in a project. Considering the total number of occurrences, results show that participants assisted by our tool made more mistakes while evolving the sorting system than those working with code only.

However, drawing a conclusion from this total number of occurrences of logical errors can be misleading, given that making a mistake can mask the existence of others. Consider the following scenario regarding the code evolution activity. A participant correctly relates two plan metadata elements to a plan; however, in these two plan metadata elements this participant does not assign any value defining a minimum number of executions and a learning gap to them. In this case, considering the total number of occurrences, we assume that this participant made 4 logical errors (two for each plan metadata element). If a second participant performing the same activity relates only one of the correct plan metadata elements to the plan and made the same logical errors as the first participant, her final logical error count would be 3: one for not relating a plan metadata element and two for forgetting the minimum number of executions and learning gap values. However, if this second participant had correctly added a second plan metadata element to the plan, she probably would make the same mistakes she made in the first plan metadata element. Therefore, her final error count would be 4, as well as the first participant.

Considering the given example, we also analysed the number of different types of logical errors in evolved projects. We can note that the difference between groups decreases; however, participants working with code only still present better performance than those assisted by *Sam*. This difference observed can be related to the expertise of participants from this group, with some of them presenting more than ten years of programming experience (including beginners). Given that development experience is helpful when dealing only with the code, these participants managed to adequately evolve the implemented system.

Analysing the types of errors found in projects from participants working with code only, we observe the predominance of modelling errors, such as missing elements or wrong relationships. Not adding softgoals and preferences to an agent and not adding the whole set of metadata correctly to the agent's belief base were errors also observed. Moreover, there were several cases in which the classes that implement our approach were not correctly extended.

Projects from the group supported by our tool present different types of logical errors. Most of them relates to code snippets automatically generated by our tool, which should have been modified by participants (i.e. initialising beliefs and defining values for minimum plan executions and learning gap), while others relate to modelling mistakes. We can observe that most errors found in this group directly relate to the experience of participants using our tool and may be occurred due to the limited tool usability and limited tutorial time. Therefore, with an adequate background, the number of errors is expected to decrease over time.

In a nutshell, there is evidence indicating that the use of *Sam* tool can improve the evolution of existing BDI-agent-code when we consider the time to perform this evolution. From the perspective of correctness, participants working with source code only had a better performance. However, we believe that with an extended tutorial and a longer time to handle and use the tool before performing this activity, participants using *Sam* would have a performance similar or better than that from participants using code only. This belief is supported by the fact that most errors found in their projects were not originated by the wrong use of our tool, but from a lack of knowledge on adaptations that should be made in the generated code.

5.2.2.3 USE Questionnaire

Finally, the participants that performed the previous activities with the support of *Sam* were requested to answer a series of questions regarding three aspects: (i) tool usability; (ii) satisfaction; and (iii) ease of use. Table 5.11 presents the average (M), standard deviation (SD), minimum and maximum values for each question.

Analysing these values brings us some interesting information. The highest averages of each dimension show that the participants found *Sam* to be useful (6.18), would recommend it to a friend (5.45) and think that it is easy to remember how to use it (5.64). However, as the lowest averages shows, there is no agreement that the tool does everything they would expect it to do (4.45). An important observation is the consensus of the need

Table 5.11: USE Results.

Question	**M**	**SD**	**Min**	**Max**
It helps me be more effective.	5.18	1.54	3.00	7.00
It helps me be more productive.	5.27	1.49	3.00	7.00
It is useful.	6.18	1.08	4.00	7.00
It makes the things I want to accomplish easier to get done.	5.36	1.36	3.00	7.00
It saves me time when I use it.	5.09	1.81	1.00	7.00
It does everything I would expect it to do.	4.45	1.51	2.00	7.00
It is easy to use.	5.36	1.57	2.00	7.00
It is simple to use.	5.45	1.37	3.00	7.00
It is user friendly.	5.36	1.20	3.00	7.00
It requires the fewest steps possible to accomplish what I want to do with it.	4.45	1.57	1.00	7.00
It is flexible.	4.91	1.22	3.00	7.00
Using it is effortless.	4.64	1.80	2.00	7.00
I can use it without written instructions.	3.91	1.87	1.00	6.00
I don't notice any inconsistencies as I use it.	4.45	2.16	1.00	7.00
Both occasional and regular users would like it.	4.73	1.79	2.00	7.00
I can recover from mistakes quickly and easily.	4.54	2.25	1.00	7.00
I can use it successfully every time.	4.45	1.69	1.00	7.00
I learned to use it quickly.	4.64	2.42	1.00	7.00
I easily remember how to use it.	5.64	1.12	3.00	7.00
It is easy to learn to use it.	5.00	1.55	2.00	7.00
I quickly became skillful with it.	4.54	1.63	2.00	6.00
I am satisfied with it.	5.18	1.54	2.00	7.00
I would recommend it to a friend.	5.45	1.63	2.00	7.00
It is fun to use.	5.18	1.40	3.00	7.00
It works the way I want it to work.	4.91	1.64	2.00	7.00
It is pleasant to use.	4.82	1.83	2.00	7.00

for written instructions to allow the correct use of *Sam* (3.91). However, the majority of users still thinking that the tool is somewhat pleasant to use (4.82).

We also asked participants to list the most positive and negative aspects while using our tool. Some mentioned that it was easy to get started, and lots of useful code is generated automatically, with the tool minimising code handling. However, some participants noticed problems, which we already expected. They listed that model inconsistencies are not shown or easy to detect. It occurs due to the lack of a model checking module, which would allow *Sam* to verify a model before generating its code and warning if any violation is found. This lack of consistency checks also relates to the types of errors found in projects from this group, which can be prevented if adequately addressed. Moreover, some participants think that a model diagram may become confusing for bigger projects. We addressed this issue by providing ways of suppressing graphical connections between elements; however, as a participant pointed out, this feature can be improved by allowing a user to handle these connections.

5.2.2.4 Discussion

Our user study indicates that *Sam*, which is based on our approach, has the potential to improve the development process of BDI-agents. Evidence shows that our tool may reduce the amount of time taken from developers to initially recognise elements and their relationships in a given domain. In addition, results indicate that the time taken to correctly understand a project may be reduced when using our tool.

Considering the code evolution activity, our experiment demonstrates that the use of *Sam* allows developers to be faster when evolving a system than they would be using code only. We observe that this benefit, as well as those mentioned before, come mainly from the use of graphical models to design and generate source code, which allow developers to work without focusing on implementation particularities.

However, participants assisted by our tool do not presented a great performance regarding correctness of the evolved project. There was a considerable larger amount of errors in projects from this group of participants than in those from the group using code only. However, we must highlight that most of these errors were not originated by the wrong use of our tool, but from a lack of knowledge on adaptations that should be made by participants in the generated code. Therefore, we identified the need for providing a detailed tutorial that, together with a longer use of our tool, should be able to address this correctness issue. Moreover, we pointed out that verifying model inconsistencies would

also allow developers using *Sam* to perform better. This is an issue that we were aware and aim to address providing our tool with a model checking feature.

Finally, participants gave a feedback about the usability, satisfaction and ease of use of *Sam* tool, providing valuable insights of improvements that can be performed to ease de development process of software agents with learning capabilities. In this way, we provided not only a tool for implementing software agents but also means of designing and documenting such agents.

5.3 Final Remarks

In this chapter we presented the experiment and user study conducted to evaluate our approach for plan selection and the impact of using *Sam* to support the development process of agents, respectively. In Section 5.1, we described the empirical experiment where we compared our technique for plan selection with a random selection. Then, we presented and analysed results, which demonstrated that our approach is effective. In Section 5.2, we detailed the user study performed, describing its steps and participants. Results were presented and discussed, and indicated that our tool has the potential to improve the development process of BDI-agents. Finally, in next chapter we provide a conclusion describing our main contributions and opportunities for future work.

6 CONCLUSION AND FUTURE WORK

The BDI architecture is a robust solution proposed to deal with dynamic and complex domains, addressed by applications that need to provide flexible and intelligent behaviour. One of the main reasons for the flexibility and robustness of this approach is the plan selection process, part of the BDI reasoning cycle. This plan selection process is highly customisable, and many techniques have been developed to improve it.

In this dissertation, we proposed an approach that improves the plan selection process by the use of machine learning techniques. Our approach allows agents to learn the plans that possibly will perform best considering the current context and agent's preferences over softgoals. Our approach is twofold. First, we introduced a meta-model that defines concepts that allow the representation of the information needed for the plan selection, which must be provided by specific applications. Second, we proposed a technique that selects plans based on agent's preferences and the information provided by the meta-model instances. The latter is used to predict plan outcomes according to the current context. Therefore, an agent built with our approach is capable of learning to select a plan that best satisfies agent's preferences using information provided by previous plan executions. Moreover, learning allows agents to cope with dynamic environments that evolve over time. Our empirical evaluation showed that our approach is effective even when the prediction model is not frequently updated.

We also aimed to provide an approach feasible to be used in real world applications. Therefore, we implemented the meta-model and technique proposed as an extension of the BDI4JADE framework. This implementation is purely Java-based, which makes agents implemented with it able to be extended and integrated with several technologies currently in use. Additionally, we developed *Sam*, a tool to support the development process of software agents that implement our technique. It allows developers to graphically model and generate source code of BDI agents with learning capabilities. A user study was performed to assess the improvements of a tool-supported BDI-agend-based method, and evidence suggests that our tool may help developers that are not experts or are unfamiliar with the agent technology, mainly with respect to application understanding. We also evaluated how users perceive our tool regarding usability, satisfaction and ease of use. Results indicated that the tool is easy to use, although features like model checking would prevent common errors to be made. These results will serve as a basis to improve *Sam* in future work. Next, we detail contributions and discuss future work.

6.1 Contributions

Given the results presented in this dissertation, we list below our main contributions.

Meta-model for Learning-based Plan Selection. In Chapter 3 we presented a meta-model defining the information needed to model an agent able to select plans based on its preferences and the learning of outcomes from previous plan executions. This meta-model extends existing work by learning information that is required to be explicitly provided by it. Thus, simplifying the design process of an agent by hiding technical details usually unknown by mainstream software developers.

Technique to Select Plans. We proposed a technique, presented in Chapter 3, that uses the information that our meta-model provides for selecting plans accordingly. The novelty of our technique consists in learning how the environment in which an agent is inserted influences the execution of plans and, consequently, their outcomes. Using this approach, we were able to predict expected contributions of plans regarding agent's preferences, and select the plan that possibly will perform best when satisfying these preferences, as shown in a simulation performed to evaluate our approach in Chapter 5. The proposed technique and meta-model, as well as the results of our simulation, were published elsewhere (FACCIN; NUNES, 2015).

Implementation of a Technique for Plan Selection. One of the goals of our study is to demonstrate the practical applicability of the agent technology. Therefore, in Chapter 4 we presented an implementation of our proposed meta-model and technique as an extension of the BDI4JADE framework. It uses the Weka API to provide agents with learning capabilities. From this implementation, developers can create agents using the Java programming language, thus being able to extend or integrate existing technologies.

Tool to Support the Development of Software Agents. In Chapter 4, we provided a tool to support the development process of software agents based on our approach. This tool, which is referred to as *Sam*, allows users to graphically model an agent following a particular notation that represents elements and relationships from our meta-model. Additionally, a source-code generation feature was implemented, being able to generate the entire structure of packages and classes needed, including relationships represented in the model. Users must provide only specific code for plan bodies and outcomes, given that this code is domain-specific.

6.2 Future Work

The contributions provided by this dissertation are a step towards the development of an intelligent plan selection process of BDI agents. However, our work still has limitations, which leave many possibilities to be explored in future work. We discuss them next.

Expression of Different Optimisation Functions. In our work, we defined the use of optimisation functions to specify how an outcome affects the satisfaction of a softgoal and how outcome values must be transformed to utilities. We limited our study to using only two of these optimisation functions, namely *minimise* and *maximise*, which are linear functions. However, there are situations these functions are not adequate. For instance, when the utility of an outcome increases as the outcome value increases but starts to decrease if this value exceeds a certain limit. In fact, allowing the expression of different optimisation functions can make our approach adequate in more domains.

Assign Weights to Recorded Observations According to Their Recency. Our technique considers every data collected from plan executions as having the same informative power. However, we must consider that older information tends to become less informative and, from a given point, useless. Therefore, assigning weights to collected data in a way that recent information becomes more valuable than older could improve the adaptive characteristic of our technique.

Exploring Different Learning Algorithms and Models. We mentioned in Chapter 4 that any machine learning model able to perform numeric regression could be used in our technique. However, we only adopted the linear regression model to perform our simulation. Exploring existing learning algorithms and models, and how they fit in different scenarios, would be a valuable contribution to our work.

Improvement of our tool. From results collected in our user study, we identified features that are necessary to improve our tool usability as well as to ease the development process of software agents. For instance, the development of a model checking extension that allows our tool to validate an agent model before generating its source-code; thus, informing the user if this model violates any constraint and possibly suggesting means of correcting this violation. Also, we can investigate how to improve diagrams visualisation, given that the current way models are displayed may become difficult to read for bigger projects.

In summary, we provided an approach composed of a meta-model and technique, which can select a plan that is expected to better satisfy agent's preferences over softgoals. We also presented an implementation of this technique, as well as a tool to support the development of software agents based on our work. Our main objective is to make this approach easier to be adopted by mainstream software developers, thus promoting the large-scale adoption of agent technology in industry.

REFERENCES

AIRIAU, S. et al. Incorporating learning in BDI agents. In: ADAPTIVE LEARNING AGENTS AND MULTI-AGENT SYSTEMS WORKSHOP (ALAMAS+ALAG-08). **Proceedings...** Estoril, Portugal, 2008.

AIRIAU, S. et al. Enhancing adaptation in BDI agents using learning techniques. **International Journal of Agent Technologies and Systems (IJATS)**, IGI Global, v. 1, n. 2, p. 1–18, jan 2009.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. In: **Encyclopedia of Software Engineering**. [S.l.]: Wiley, 1994.

BELECHEANU, R. et al. Commercial applications of agents: lessons, experiences and challenges. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS. **Proceedings...** 2006. (AAMAS'06), p. 1549–1555. Available from Internet: <http://doi.acm.org/10.1145/1160633.1160932>. Accessed in: 2016-01-07.

BORDINI, R. H.; HüBNER, J. F.; WOOLDRIDGE, M. **Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)**. [S.l.]: John Wiley & Sons, 2007.

BRATMAN, M. **Intention, plans, and practical reason**. Cambridge, MA: Harvard University Press, 1987.

BRESCIANI, P. et al. Tropos: An agent-oriented software development methodology. **Autonomous Agents and Multi-Agent Systems**, Kluwer Academic Publishers, Hingham, MA, USA, v. 8, n. 3, p. 203–236, 2004.

BUSETTA, P. et al. JACK intelligent agents: Components for intelligent agents in Java. **AgentLink Newsletter**, v. 2, jan. 1999.

CHAOUCHE, A.-C. et al. Spatio-temporal guidance for ambient agents. In: INTERNATIONAL CONFERENCE ON CONTROL SYSTEMS AND COMPUTER SCIENCE (CSCS). **Proceedings...** 2015. p. 719–726. Available from Internet: <http://dx.doi.org/10.1109/CSCS.2015.79>. Accessed in: 2016-01-07.

DAM, K. H.; WINIKOFF, M. Cost-based bdi plan selection for change propagation. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS. **Proceedings...** Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2008. (AAMAS'08), p. 217–224.

DAVIS, F. D. Perceived usefulness, perceived ease of use, and user acceptance of information technology. **MIS Quarterly**, Society for Information Management and The Management Information Systems Research Center, Minneapolis, MN, USA, v. 13, n. 3, p. 319–340, sep 1989. Available from Internet: <http://dx.doi.org/10.2307/249008>. Accessed in: 2016-01-07.

DENNETT, D. **The Intentional Stance**. [S.l.]: MIT Press, 1987. (A Bradford book).

FACCIN, J.; NUNES, I. Bdi-agent plan selection based on prediction of plan outcomes. In: INTERNATIONAL CONFERENCE ON WEB INTELLIGENCE AND INTELLIGENT AGENT TECHNOLOGY (WI-IAT). **Proceedings...** 2015. v. 2, p. 166–173. Available from Internet: <http://dx.doi.org/10.1109/WI-IAT.2015.58>. Accessed in: 2016-01-07.

GALTON, F. **Natural inheritance**. [S.l.]: Macmillan and Company, 1894. 282 p.

GUERRA-HERNáNDEZ, A.; FALLAH-SEGHROUCHNI, A. E.; SOLDANO, H. Learning in bdi multi-agent systems. In: COMPUTATIONAL LOGIC IN MULTI-AGENT SYSTEMS (CLIMA). **Proceedings...** [S.l.]: Springer, 2004. (Lecture Notes in Computer Science, v. 3259), p. 218–233.

KEENEY, R. L.; RAIFFA, H. **Decisions with Multiple Objectives: Preferences and Value Tradeoffs**. New York: John Wiley & Sons, Inc, 1976.

LUND, A. M. Measuring usability with the use questionnaire. **STC Usability SIG Newsletter: Usability Interface**, oct 2001.

NGUYEN, A.; WOBCKE, W. An adaptive plan-based dialogue agent: integrating learning into a bdi architecture. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS. **Proceedings...** ACM, 2006. (AAMAS'06), p. 786–788. Available from Internet: <http://doi.acm.org/10.1145/1160633.1160771>. Accessed in: 2016-01-07.

NUNES, I.; LUCENA, C. J. P. D.; LUCK, M. Bdi4jade: a bdi layer on top of jade. In: INTERNATIONAL WORKSHOP ON PROGRAMMING MULTI-AGENT SYSTEMS. **Proceedings...** [S.l.], 2011. (ProMAS 2011), p. 88–103.

NUNES, I.; LUCK, M. Softgoal-based plan selection in model-driven bdi agents. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS. **Proceedings...** Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2014. (AAMAS'14), p. 749–756.

PADGHAM, L.; SINGH, D. Situational preferences for bdi plans. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS. **Proceedings...** Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2013. (AAMAS'13), p. 1013–1020.

PHUNG, T.; WINIKOFF, M.; PADGHAM, L. Learning within the bdi framework: An empirical analysis. In: INTERNATIONAL CONFERENCE ON KNOWLEDGE-BASED INTELLIGENT INFORMATION AND ENGINEERING SYSTEMS. **Proceedings...** Berlin, Heidelberg: Springer-Verlag, 2005. (KES'05), p. 282–288. Available from Internet: <http://dx.doi.org/10.1007/11553939_41>. Accessed in: 2016-01-07.

RAO, A. S.; GEORGEFF, M. P. Bdi agents: From theory to practice. In: INTERNATIONAL CONFERENCE ON MULTI-AGENT SYSTEMS (ICMAS). **Proceedings...** [S.l.], 1995. p. 312–319.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. In: RUMELHART, D. E.; MCCLELLAND, J. L.; GROUP, C. P. R. (Ed.). Cambridge, MA, USA: MIT

Press, 1986. chp. Learning Internal Representations by Error Propagation, p. 318–362. Available from Internet: <http://dl.acm.org/citation.cfm?id=104279.104293>. Accessed in: 2016-01-07.

RUSSELL, S. J.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. 2. ed. [S.l.]: Pearson Education, 2003.

SINGH, D.; SARDINA, S.; PADGHAM, L. Extending BDI plan selection to incorporate learning from experience. **Journal of Robotics and Autonomous Systems**, v. 58, p. 1067–1075, 2010.

SINGH, D. et al. Integrating learning into a BDI agent for environments with changing dynamics. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE (IJCAI). **Proceedings...** [S.l.]: AAAI Press, 2011. p. 2525–2530.

SINGH, D. et al. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS. **Proceedings...** [S.l.]: IFAAMAS. (AAMAS'10), p. 325–332.

SMOLA, A.; SCHOELKOPF, B. **A tutorial on support vector regression**. [S.l.], 1998. NeuroCOLT2 Technical Report NC2-TR-1998-030.

TUKEY, J. W. Comparing individual means in the analysis of variance. **Biometrics**, International Biometric Society, v. 5, n. 2, p. pp. 99–114, 1949. Available from Internet: <http://www.jstor.org/stable/3001913>. Accessed in: 2016-01-07.

TURING, A. M. Computing machinery and intelligence. **Mind**, Oxford University Press on behalf of the Mind Association, v. 59, n. 236, p. 433–460, 1950.

VISSER, S.; THANGARAJAH, J.; HARLAND, J. Reasoning about preferences in intelligent agent systems. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE. **Proceedings...** AAAI Press, 2011. p. 426–431. Available from Internet: <http://www.ijcai.org/Proceedings/11/Papers/079.pdf>. Accessed in: 2016-01-07.

WATKINS, C. J. C. H.; DAYAN, P. Q-learning. **Machine Learning**, v. 8, n. 3, p. 279–292, 1992. Available from Internet: <http://dx.doi.org/10.1007/BF00992698>. Accessed in: 2016-01-07.

WOOLDRIDGE, M. Intelligent agents. In: WEISS, G. (Ed.). **Multiagent Systems**. [S.l.]: The MIT Press, 1999. p. 27–77.

APPENDIX A QUESTIONNAIRES AND TASK

This appendix presents the explanation texts and questions used in our user study.

Experience Survey

Name:

Student ID:

1. I am a(n)...
   ( ) ... Undergraduate student ( ) ... Master student ( ) ... PhD student.
2. For how many years have you been programming?
   ( ) < 1 year
   ( ) 1 year – 2 years
   ( ) 2 years – 5 years
   ( ) 5 years – 10 years
   ( ) > 10 years
3. In a scale from 0 to 8, how do you classify your experience with the Java programming language?
   ( ) 0 – None
   ( ) 1
   ( ) 2 – Minimal experience (maybe compiled a couple of example programs)
   ( ) 3
   ( ) 4 – Some experience (wrote a handful of small programs)
   ( ) 5
   ( ) 6 – Substantial experience (wrote some small to medium-sized programs)
   ( ) 7
   ( ) 8 – Extensive experience (wrote at least several complex programs)
4. In a scale from 0 to 8, how do you classify your knowledge about software agents?
   ( ) 0 – Fundamental awareness (basic knowledge)
   ( ) 1
   ( ) 2 –Novice (limited experience)
   ( ) 3
   ( ) 4 – Intermediary (practical application)

( ) 5
( ) 6 – Advanced (applied theory)
( ) 7
( ) 8 – Expert (recognized authority)

5. In a scale from 0 to 8, how do you classify your knowledge about the BDI architecture?
( ) 0 – Fundamental awareness (basic knowledge)
( ) 1
( ) 2 –Novice (limited experience)
( ) 3
( ) 4 – Intermediary (practical application)
( ) 5
( ) 6 – Advanced (applied theory)
( ) 7
( ) 8 – Expert (recognized authority)

Understanding Questionnaire

Based on the code/model given in the project `com.transportation`, answer the following questions:

1. Which outcome is associated with the MinCosts softgoal in the airplane plan?
2. Which is the optimisation function used in the ship plan, with respect to the MaxPerformance softgoal?
3. Which beliefs are influence factors of the LoadIntegrity outcome in the airplane plan?
4. Which influence factors and outcome are associated with the MaxPerformance softgoal in the truck plan?
5. Which outcome is associated with the MaxPerformance softgoal in the ship plan?
6. Which is the optimisation function used in the truck plan, with respect to the MinCosts softgoal?
7. Which beliefs are influence factors of the TimeTaken outcome in the airplane plan?
8. Which influence factors and outcome are associated with the MinCosts softgoal in the ship plan?

9. Which outcome is associated with the MaxReliability softgoal in the truck plan?
10. Which is the optimisation function used in airplane plan, with respect to the MaxReliability softgoal?
11. Which beliefs are influence factors of the FuelConsumption outcome in the truck plan?
12. Which influence factors and outcome are associated with the MaxReliability softgoal in the airplane plan?

Code Evolution Exercise

An agent A is responsible for sorting given arrays of elements. When these arrays are sorted by agent A, it means that it achieved its goal named SortArray. In order to achieve this goal, agent A has two different plans, which correspond to two different sorting algorithms: SelectionSortPlan and InsertionSortPlan. However, these plans are selected randomly in order to accomplish the specified task. Now, we want to evolve this system allowing agent A to select a suitable plan according to its preferences over softgoals. This plan selection process must also consider (i) the current state of the environment in which agent A is located and (ii) the experience from previous plan executions.

Then, in order to evolve this system you must accomplish the following tasks.

1. Provide agent A with the **learning-based plan selection algorithm**.
2. Add the following softgoals to agent A: **maxPerformance** and **minNumberOfSwaps**.
3. Set the metadata of the two existing plans, according to the following specification.

InsertionSortPlan

Influence Factors	**Outcome**	**Optimisation Function**	**Softgoal**
ArraySize, Additions, Removals	CPUTime	Minimize	maxPerformance
ArraySize, Additions, Removals	NumberOfSwaps	Minimize	minNumberOfSwaps

SelectionSortPlan

Influence Factors	**Outcome**	**Optimisation Function**	**Softgoal**
ArraySize, Additions, Removals	CPUTime	Minimise	maxPerformance
ArraySize, Additions, Removals	NumberOfSwaps	Minimise	minNumberOfSwaps

USE Questionnaire

Considering your experience using *Sam*, rate the following statements with a value from 1 (strongly disagree) to 7 (strongly agree).

	1	**2**	**3**	**4**	**5**	**6**	**7**
It helps me be more effective.	○	○	○	○	○	○	○
It helps me be more productive.	○	○	○	○	○	○	○
It is useful.	○	○	○	○	○	○	○
It makes the things I want to accomplish easier to get done.	○	○	○	○	○	○	○
It saves me time when I use it.	○	○	○	○	○	○	○
It does everything I would expect it to do.	○	○	○	○	○	○	○
It is easy to use.	○	○	○	○	○	○	○
It is simple to use.	○	○	○	○	○	○	○
It is user friendly.	○	○	○	○	○	○	○
It requires the fewest steps possible to accomplish what I want to do with it.	○	○	○	○	○	○	○
It is flexible.	○	○	○	○	○	○	○
Using it is effortless.	○	○	○	○	○	○	○
I can use it without written instructions.	○	○	○	○	○	○	○
I don't notice any inconsistencies as I use it.	○	○	○	○	○	○	○
Both occasional and regular users would like it.	○	○	○	○	○	○	○
I can recover from mistakes quickly and easily.	○	○	○	○	○	○	○
I can use it successfully every time.	○	○	○	○	○	○	○
I learned to use it quickly.	○	○	○	○	○	○	○
I easily remember how to use it.	○	○	○	○	○	○	○
It is easy to learn to use it.	○	○	○	○	○	○	○
I quickly became skillful with it.	○	○	○	○	○	○	○
I am satisfied with it.	○	○	○	○	○	○	○
I would recommend it to a friend.	○	○	○	○	○	○	○
It is fun to use.	○	○	○	○	○	○	○
It works the way I want it to work.	○	○	○	○	○	○	○
It is pleasant to use.	○	○	○	○	○	○	○

List the most negative aspects:

List the most positive aspects:

APPENDIX B RESUMO ESTENDIDO

Seleção de Planos BDI Baseada em Contexto e Preferências usando Aprendizado de Máquina: dos Modelos à Geração de Código

O crescimento da interação entre pessoas, sistemas e ferramentas tornou-se notável principalmente nos últimos anos. Esse crescimento não se dá apenas em termos de escala, mas também de complexidade. Estes fatores demandam que novas tecnologias sejam desenvolvidas a fim de lidar com essas interações de maneira adequada. Uma dessas tecnologias é a tecnologia de agentes. Ela se baseia no conceito de agentes de software, que podem ser entendidos como entidades computacionais com características autônomas, proativas e reativas, situadas em determinado ambiente. Isso significa que um agente é capaz de agir e reagir a estímulos do ambiente buscando atingir determinado objetivo, sem a necessidade de intervenção humana.

Diversas metodologias surgiram buscando auxiliar o desenvolvimento de agentes de software. Uma destas abordagens é a arquitetura BDI, que provê agentes com atitudes mentais de crenças, desejos e intenções, que definem o comportamento destes agentes. Estas atitudes, quando combinadas a outros módulos dentro de um ciclo de raciocínio, deixam diversas lacunas a serem preenchidas em aplicações específicas, entre elas o processo de seleção de planos. Este processo consiste na seleção de um plano dentre um conjunto de planos pré-definidos, adequados a atingir determinado objetivo. Nas implementações típicas de agentes BDI, a ausência de uma definição específica de processo de seleção de planos faz com que planos sejam selecionados de maneira aleatória. Porém, geralmente consideramos diversos fatores ao realizarmos uma escolha, como nossa situação e nossas preferências. Um modo de melhorar o processo de seleção de planos é permitir que agentes realizem este tipo de consideração.

Neste contexto, diversas abordagens tem sido propostas, abrangendo desde o aprendizado de contextos em que planos tendem a falhar (desta forma selecionando apenas planos cuja execução leve a uma situação de sucesso), até a seleção de planos que melhor combinem com as preferências dos agentes. Porém, grande parte destas abordagens não considera a incerteza que envolve a execução de um plano. Isto é, um plano que normalmente atinja seu objetivo pode falhar e originar uma situação de insucesso. As abordagens que fazem este tipo de consideração acabam dependendo de informações que são difíceis, ou até mesmo impossíveis, de serem elicitadas em tempo de desenvolvimento.

Baseados nestes fatores, estabelecemos a seguinte questão de pesquisa: *como um agente BDI pode realizar a seleção de planos considerando seu contexto e a incerteza relacionada aos resultados da execução de um plano de modo que também considere suas preferências?*

Buscando responder a esta questão, desenvolvemos uma abordagem que, basicamente, permite que agentes aprendam o resultado de execuções de planos em determinados contextos e utilizem esta informação para prever os resultados destes mesmos planos em contextos distintos. Este conhecimento é, então, utilizado no processo de seleção de planos, considerando ainda as preferências dos agentes sobre softgoals. Buscando demonstrar a aplicabilidade prática deste processo de seleção melhorado, desenvolvemos uma implementação e uma ferramenta de suporte ao desenvolvimento de agentes baseadas na técnica desenvolvida. Assim, essa abordagem é composta basicamente por (i)um meta-modelo que, quando instanciado, fornece informações específicas para (ii) uma técnica que fará uso dessas informações no processo de seleção de planos; e (iii) um método de desenvolvimento de agentes de software com apoio ferramental.

Um agente BDI típico é composto por um conjunto de crenças, um conjunto de objetivos, e um conjunto de planos que permitirão que o agente atinja estes objetivos. Porém, alguns outros fatores devem ser considerados no processo de seleção de planos. Nosso meta-modelo busca representar estes fatores, especificamente:

Softgoal É um objetivo secundário, que não pode ser totalmente alcançado por um plano, mas pode ser mais ou menos satisfeito de acordo com as ações executadas por esse plano. Um exemplo de softgoal é maximizar a performance para a obtenção de determinado objetivo.

Preferências São valores, entre 0 e 1, expressos sobre cada um dos softgoals de um agente. Estas preferências representando um trade-off entre estes softgoals, e indicam a importância de determinado softgoal para o agente. Quanto maior a importância dada a um softgoal, maior o valor de preferência definido.

Resultados São valores que podem ser mensurados durante e/ou após a execução de um plano. Estes resultados se relacionam a um domínio específico e fornecem um valor dentro deste domínio. Eles definem a relação entre um plano e os softgoals do agente. Considerando um softgoal de maximizar performance, o resultado de um plano relacionado a este softgoal pode ser o tempo dispendido para atingir determinado objetivo.

Fatores de Influência São variáveis de acordo com o contexto, mapeados diretamente às crenças do agente, e podendo afetar um ou mais resultados.

Função de Otimização Define o modo como um resultado afeta um softgoal. Esta função estabelece se o valor de um resultado deve ser maximizado ou minimizado para melhor satisfazer as preferências do agente sobre o softgoal considerado.

Elemento de Metadado do Plano É responsável por agregar à um plano um resultado, seus fatores de influência, uma função de otimização e seu respectivo softgoal.

Esse meta-modelo instanciado em aplicações específicas fornece os dados necessários para a técnica de seleção de planos criada. Dado que o objetivo desta técnica é criar um modelo preditivo adequado para prever resultados de planos em diferentes contextos, um estágio inicial de aprendizado é necessário. Neste estágio, planos são selecionados de maneira randômica e informações específicas sobre sua execução são coletadas e registradas em uma estrutura semelhante à apresentada na Tabela 3.1, onde cada linha representa uma execução de um plano, e as colunas representam fatores de influência sobre o resultado representado na última coluna. Este estágio de aprendizado acontece até que todos os planos atinjam um limite mínimo de execuções definido pelo desenvolvedor.

Terminado este processo de coleta de informações, modelos preditivos são criados para cada plano. Dado que a natureza deste problema é entender o relacionamento entre uma variável dependente (o resultado) e uma ou mais variáveis independentes (os fatores de influência), qualquer algoritmo capaz de realizar regressão numérica pode, em teoria, ser utilizado. Neste trabalho apenas a técnica de regressão linear foi utilizada, dada a sua popularidade e capacidade de abranger diversos domínios.

Criados os modelos preditivos, durante o processo de seleção de planos, o agente irá considerar o contexto atual para prever os valores dos resultados de cada plano. Estes valores serão transformados em valores de utilidade de acordo com as funções de otimização definidas, serão combinados, e o plano com o maior valor de utilidade predita será selecionado. O contexto em que o plano foi selecionado e os valores de seus resultados após a execução são então registrados e alimentarão uma base de dados que, posteriormente, servirá para atualizar o modelo preditivo, seguindo uma taxa de atualização definida pelo desenvolvedor.

Com a finalidade de validar o meta-modelo e a técnica propostos, implementamos um cenário onde uma empresa C fornece o serviço de entregar cargas de um lugar x a um lugar y. Para isso ela dispõe de quatro meios de transporte distintos - avião, caminhão, navio e trem. Um agente A é responsável por gerenciar os serviços desta companhia, e, buscando garantir a qualidade dos serviços oferecidos, possui três softgoals distintos - *maximizar a integridade* da carga entregue, *minimizar o custo* da entrega desta carga e *maximizar*

a performance da entrega. O plano de transporte usando caminhão se relaciona com o softgoal maximizar performance através do resultado tempo dispendido. Este resultado é influenciado pelas condições do veículo, condições da rodovia e condições do tráfego. Da mesma forma, todos os planos possuem elementos de metadados que os relacionam a todos os softgoals do agente A.

A partir desta implementação, realizamos um experimento buscando mensurar a satisfação obtida pelo agente A utilizando diferentes estratégias de seleção de planos. Neste experimento valores de preferência para cada softgoal eram gerados de maneira randômica. Valores para o contexto também eram gerados randomicamente e as crenças atualizados de acordo. Planos eram selecionados de acordo com a estratégia definida e a satisfação obtida com a execução do plano selecionado era registrada. Foram utilizadas uma estratégia de seleção randômica (RAN) e a estratégia baseada em aprendizado, com o número mínimo de execuções dos planos definido em 50 e diferentes valores de atualização do modelo preditivo. Foram utilizadas estratégias com atualização do modelo preditivo a cada 1 (LB-1), 100 (LB-100), 500 (LB-500) e 1000 (LB-1000) execuções de plano. Foram executadas 5000 mil iterações dos passos descritos acima para cada estratégia de seleção de planos e um resumo dos resultados é apresentado na Tabela 5.2.

É possível perceber que todas as estratégias baseadas em aprendizagem possuem uma performance melhor do que a seleção randômica, com todas elas obtendo um valor aproximado de satisfação de 0.68 em média. Apesar de parecer um valor pequeno à primeira vista em comparação a satisfação máxima que pode ser obtida (1.00), é necessário considerar que a incerteza continua cumprindo seu papel, impedindo que resultados sejam preditos com total precisão. Além disso, a seleção dos planos envolve a resolução de um trade-off entre softgoals, visto que alguns planos satisfazem melhor determinados softgoals, enquanto outros planos satisfazem melhor outros softgoals. Dessa forma, nenhum plano é capaz de atingir a satisfação máxima. Partindo destas considerações, 0.68 pode ser considerado um valor alto.

A pouca diferença de resultado entre as diferentes estratégias baseadas em aprendizagem já era esperada, visto que a partir do momento em que um modelo preditivo válido é construído, as estratégias tendem a se comportar de maneira semelhante. Entretanto, é esperado que, caso algum evento altere a relação entre um resultado e seus fatores de influência, estratégias que atualizem seu modelo preditivo de modo mais frequente reajam de maneira mais rápida.

Também analisamos a satisfação acumulada ao longo das 5000 iterações e como é pos-

sível observar na Figura 5.2, estratégias baseadas em aprendizado apresentam um melhor benefício a longo prazo. Apesar de não ser possível notar neste gráfico, inicialmente todas as cinco estratégias possuem um desempenho similar, visto que as estratégias baseadas em aprendizagem realizam uma seleção randômica de planos durante seu período de coleta de informações. Estes dados nos mostram que, neste tipo de cenário, a seleção baseada em aprendizado é efetiva mesmo quando o modelo preditivo não é frequentemente atualizado.

Além de prover fundamentação teórica, o objetivo deste trabalho é demonstrar a aplicabilidade prática do meta-modelo e técnica propostos. Dessa forma, desenvolvemos uma implementação e uma ferramenta que dá suporte ao desenvolvimento dirigido a modelos de agentes com características de aprendizado.

A implementação foi realizada como uma extensão do BDI4JADE[1], um framework que implementa uma camada BDI sobre o JADE[2]. O BDI4JADE não fornece uma nova linguagem de programação nem depende de linguagens específicas de domínio baseadas em XML. Ele é puramente baseado em Java, e desta forma permite que agentes construídos a partir deste framework sejam estendidos ou integrados a tecnologias já existentes, como AspectJ[3] ou Hibernate[4] por exemplo.

A Figura 4.1 apresenta o diagrama de classes da implementação realizada. As classes em cinza escuro são aquelas já existentes no framework, enquanto as demais são aquelas implementadas pelo autor. Classes em cinza claro são transparentes ao desenvolvedor, enquanto as em branco são aquelas classes que devem ser explicitamente manipuladas no processo de desenvolvimento de um agente. Cabe destacar que toda a técnica de seleção de planos fica encapsulada na classe `LearningBasedPlanSelectionStrategy`, desta forma, é apenas requisitado ao desenvolvedor que forneça informações necessárias solicitadas pelo meta-modelo. Um exemplo parcial de implementação pode ser observado no trecho de código apresentado na Figura 4.2, onde dois elementos de metadados de plano são criados, agregando resultados, fatores de influência, funções de otimização e outras informações, à um plano.

Além desta implementação, desenvolvemos uma ferramenta que auxilia no processo de desenvolvimento de agentes com capacidade de aprendizado. Esta ferramenta, chamada *Sam*, foi desenvolvida como um plug-in para o Eclipse, e permite que o desenvolvedor modele um agente de forma gráfica e gere código Java baseado no modelo instanciado.

[1] <http://inf.ufrgs.br/prosoft/bdi4jade>
[2] <http://jade.tilab.com/>
[3] <https://eclipse.org/aspectj/>
[4] <http://hibernate.org/>

A Figura 4.3 apresenta um panorama geral da ferramenta. Ao centro encontra-se a área de modelagem, apresentando alguns elementos do meta-modelo e suas relações. É importante destacar que a notação desenvolvida foi baseada na notação do Tropos, buscando, dessa forma, manter uma consistência entre notações já existentes. Representações de capacidades, elementos de metadados do plano, crenças e resultados, além de suas relações, foram definidas pelo autor. Do lado direito da tela encontra-se a paleta de elementos do meta-modelo que podem ser instanciados em um diagrama, e do lado esquerdo, o explorador de pacotes, que permite ao desenvolvedor navegar entre os arquivos do modelo e do diagrama. Na parte inferior, uma aba de propriedades é exibida ao selecionar-se um elemento do diagrama. Esta aba permite a visualização e edição de informações referentes ao elemento selecionado.

Buscando avaliar os benefícios da utilização da nossa ferramenta no processo de desenvolvimento de agentes, realizamos um estudo com usuários, onde o *Sam* foi comparado ao uso exclusivo de código para realizar duas atividades distintas relacionadas ao entendimento e evolução de projetos de agentes já implementados. Para modelar este estudo utilizamos a metodologia GQM, originando um objetivo que define este experimento. Deste objetivo, duas questões de pesquisa foram derivadas, especificamente: (i) *o uso da ferramenta Sam facilita o entendimento de código BDI existente?* e (ii) *o uso da ferramenta Sam melhora o processo de evolução de código BDI existente?* Para cada questão de pesquisa foram definidas métricas que serão detalhadas posteriormente.

Este estudo contou inicialmente com 26 participantes, porém, apenas 18 deles realizaram todas as etapas do experimento, que foi composto por cinco etapas. Inicialmente os participantes responderam a um questionário referente à sua experiência relativa a tópicos deste estudo. A partir das respostas fornecidas, eles foram classificados como novatos, intermediários ou especialistas. Os participantes foram, então, divididos em dois grupos e receberam uma aula introdutória, buscando garantir que todos os participantes possuíssem um mesmo conhecimento mínimo a respeito dos tópicos abordados no experimento. Um dos grupos recebeu um rápido tutorial sobre o uso da ferramenta *Sam*. O terceiro passo consistiu em uma atividade de entendimento, onde os participantes deveriam responder questões referentes à um projeto de agente já implementado, com um dos grupos utilizando a ferramenta e o outro apenas código. Esta atividade forneceu as métricas necessárias para responder a primeira questão de pesquisa. O quarto passo foi uma atividade de evolução de código, onde foi solicitado aos participantes que evoluíssem um projeto de agente já implementado, com um grupo utilizando a ferramenta *Sam* e

outro apenas código. Este passo forneceu métricas para responder a segunda questão de pesquisa. O último passo foi realizado apenas pelo grupo que utilizou o *Sam* nas duas atividades anteriores, onde os participantes forneceram feedback sobre a usabilidade, satisfação e facilidade de uso da ferramenta.

A primeira métrica obtida pela atividade de entendimento diz respeito à questões respondidas de maneira correta. Esta métrica foi analisada sob duas perspectivas: considerando respostas parcialmente corretas e considerando apenas respostas certas e erradas. Observando os resultados apresentados, é possível perceber que participantes utilizando o *Sam* possuem um maior entendimento quando confrontados com um projeto pela primeira vez. Ainda, é possível observar que participantes de ambos os grupos aprendem tanto o projeto quanto a ferramenta que estão utilizando.

Sob a perspectiva de questões completamente corretas, o grupo utilizando nossa ferramenta continua com um melhor desempenho inicial e o mesmo comportamento de aprendizado é observado. Porém, sob esta perspectiva, notamos o súbito crescimento na curva de aprendizado dos participantes utilizando a ferramenta. Isso indica que participantes utilizando o *Sam* são capazes de aprender um projeto de maneira mais fácil do que aqueles utilizando apenas código.

Considerando apenas os escores gerais de cada grupo e observando a Figura 5.6 é possível notar que o grupo utilizando apenas código possui uma mediana maior, entretanto a variância deste grupo também é muito maior. Analisando estes resultados, verificamos que, neste grupo, participantes com maior experiência foram capazes de responder às questões adequadamente, enquanto novatos cometeram vários erros. No grupo utilizando a ferramenta, porém, todos os participantes obtiveram resultados bons e similares. Cabe lembrar que a média do grupo usando a ferramenta foi maior em ambas as perspectivas.

A segunda métrica fornecida nesta atividade foi o tempo para responder às questões de forma correta. O grupo utilizando o *Sam* foi capaz de entender inicialmente o projeto de forma mais rápida do que o outro grupo. Verificamos ainda que o grupo utilizando o Sam teve menor variância e mediana se comparado ao outro grupo considerando o tempo total para a realização da atividade. Desta forma, os resultados indicam que o uso da ferramenta *Sam* facilita o entendimento de um projeto considerando tanto tempo quanto corretude.

Buscando responder a segunda questão de pesquisa, extraímos da atividade de evolução as seguintes métricas: tempo para a realização da tarefa, número de erros de compilação no código evoluído e número de erros lógicos no código evoluído. Esta última sendo anal-

isados sob duas perspectivas, a de número total de ocorrências de erros e de número de tipos diferentes de erros encontrados.

Considerando o tempo dispendido na realização da atividade, notamos que o grupo utilizando a ferramenta foi mais rápido. Este resultado explicita o benefício de evoluir um modelo ao invés de um código. O uso de modelos permite que o desenvolvedor foque na tarefa a ser desenvolvida, sem se preocupar com distrações específicas da linguagem de programação utilizada.

O grupo que trabalhou diretamente com o código obteve melhores resultados considerando tanto erros de compilação quanto erros lógicos. A partir destes dados nota-se a necessidade de que usuários sejam expostos a um maior tempo de manipulação da ferramenta e orientação a fim de que possam obter resultados semelhantes ou até mesmo melhores do que se utilizassem apenas código. Os resultados, sugerem que o uso do *Sam* pode melhorar a evolução de projetos se considerarmos o tempo para realizar esta evolução. Por fim, os participantes que fizeram uso do *Sam* no desenvolvimento das tarefas consideraram-no útil e recomendariam o seu uso a amigos.

A arquitetura BDI é uma solução robusta para lidar com domínios dinâmicos e complexos e um dos responsáveis por essa robustez é o processo de seleção de planos. Neste trabalho desenvolvemos uma abordagem que melhora este processo fazendo uso de técnicas de aprendizado de máquina e considerando fatores como contexto e preferências. Esta abordagem se mostrou efetiva no cenário apresentado. Ainda, procuramos demonstrar a viabilidade prática desta abordagem em aplicações reais através de sua implementação e do desenvolvimento de uma ferramenta que permite a modelagem e geração de códigos de agentes capazes de aprender. Resultados sugerem que essa ferramenta pode auxiliar desenvolvedores não tão experientes ou que não estejam familiarizados com a tecnologia de agentes. Considerando-se os resultados apresentados, listamos a seguir as principais contribuições desta dissertação.

Meta-modelo para Seleção de Planos Baseada em Aprendizagem. Nesta dissertação, apresentamos um meta-modelo definindo a informação necessária para a modelagem de um agente capaz de selecionar planos baseando-se em suas preferências e o aprendizado dos resultados de suas execuções prévias. Este meta-modelo estende um trabalho já existente, aprendendo informações que devem ser explicitamente fornecidas naquela abordagem. Essa aprendizagem simplifica o processo de projeto de um agente, encapsulando detalhes técnicos que são usualmente desconhecidos por desenvolvedores de software.

Técnica de Seleção de Planos. Nós propusemos uma técnica que utiliza a informação fornecida pelo nosso meta-modelo para selecionar planos de maneira adequada. A inovação desta técnica consiste em aprender como o ambiente em que um agente se encontra influencia a execução dos planos deste agente e, consequentemente, seus resultados. Utilizando esta abordagem, um agente é capaz de predizer as contribuições esperadas de um plano em relação às suas preferências, e selecionar o plano que tenha mais chances de satisfazer estas preferências quando executado. Esta capacidade foi demonstrada em uma simulação realizada para validar nossa abordagem.

Implementação de uma Técnica para Seleção de Planos. Um dos objetivos deste estudo é demonstrar a aplicabilidade prática da tecnologia de agentes. Assim, apresentamos o meta-modelo e técnica propostos implementados como uma extensão do framework BDI4JADE. A partir desta implementação, desenvolvedores podem criar agentes usando a linguagem de programação Java, sendo capazes de estender ou integrar tecnologias já existentes.

Ferramenta de Suporte ao Desenvolvimento de Agentes. Fornecemos uma ferramenta para auxiliar o processo de desenvolvimento de agentes de software baseados em nossa abordagem. Essa ferramenta, a qual chamamos de *Sam*, permite que usuários modelem um agente de forma gráfica, seguindo uma notação específica que representa os elementos e relacionamentos definidos em nosso meta-modelo. Adicionalmente, uma funcionalidade de geração de código foi implementada, sendo capaz de gerar a completa estrutura de pacotes e classes necessária, incluindo relacionamentos representados em um modelo instanciado.

Em resumo, fornecemos uma abordagem composta de um meta-modelo e uma técnica capaz de selecionar um plano que tende a melhor satisfazer as preferências de um agente sobre seus softgoals. Também apresentamos uma implementação desta técnica, assim como uma ferramenta para auxiliar o desenvolvimento de agentes de software baseados em nosso trabalho. Nosso principal objetivo é fazer com que esta abordagem possa ser facilmente adotada por desenvolvedores de software comuns, promovendo a adoção em larga escala da tecnologia de agentes na indústria.