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

JOÃO GUILHERME FACCIN

Automated Management of Remedial Behaviour

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

Advisor: Prof. Dr. Ingrid Oliveira de Nunes

Porto Alegre
August 2020

CIP — CATALOGING-IN-PUBLICATION

Faccin, João Guilherme

Automated Management of
Remedial Behaviour / João Guilherme Faccin. – Porto Alegre: PPGC da UFRGS, 2020.

134 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Advisor: Ingrid Oliveira de Nunes.

1. Remediation. 2. Multiagent systems. 3. Self-adaptation. 4. Resilience. I. Oliveira de Nunes, Ingrid. II. Título.

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

ACKNOWLEDGEMENTS

There are many people that must be acknowledge for this work. First, I would like to thank my supervisor, Prof. Ingrid Nunes, for her guidance through the development of this thesis. Having you as a reference certainly made me a better researcher. Thanks to the members of the examining committee, Prof. Genaina Rodrigues, Prof. Marcelo Pimenta, and Prof. Valter Camargo for their valuable comments. I would also like to thank my friends and colleagues from the Federal University of Rio Grande do Sul, for their support and friendship. A special thank you to Nicole, Frederico, Vanius, Jhonny and Fernando.

During this doctorate I had the privilege to spend some time at the Software Research and Technology lab from Concordia University, Montreal. I would like to thank Prof. Wahab Hamou-Lhadj and my colleagues from the lab for having me there. To my friends at Willowdale, thanks for making me feel like home. I will never forget the time I spent there with you.

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. Hope I made you proud. To my brother and sister, Carlos and Ana, thanks for always being with me when I needed. By following my dreams I try to show you that we are capable of anything. To Nalim, my girlfriend, thanks for the comprehension and affection, and for being the lighthouse that guided me when I sailed rough seas. I love you to the Moon and back. To my aunt, Luciana, thanks for your kindness. You are the one who made this possible.

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

Many software systems are nowadays built as sets of autonomous components, named agents, that interact with each other and are situated in an environment. Because these multiagent systems (MAS) perform a range of complex and critical tasks, they are expected to resist and recover from challenging situations that may compromise their operation and the quality of their services. D^2R^2 + DR is an existing strategy to provide software systems with this resilient behaviour. It specifies the execution of a set of operations, such as the detection of problems, their remediation, the diagnosis of their causes, and the recovery of the system to a normal operating state. Due to its abstract nature, instantiations of this strategy cannot be reused accross different application domains. Even though there are approaches that provide concrete solutions for some of these operations, they present limitations such as a lack of autonomy and adaptability. In this thesis, we propose a framework that aims at promoting resilience to MAS by providing them with the ability to autonomously manage the remediation, diagnosis and recovery operations in the face of adverse events. This framework detaches the resilient behaviour from domain-dependent code, thus promoting software reuse across different applications, and comprises three main techniques. The first automates the management of remedial actions as well as the diagnosis and solution of problem causes. The second is focused on the diagnosis of problem causes in MAS scenarios. It specifies an interaction protocol and roles that describe how agents can coordinate their actions and share information in order to keep operating with the expected quality. Finally, the third formalises an approach that allows agents to revoke the effects of executed actions without the need for an explicit declaration of which these actions are and how they should be reverted. These techniques are implemented as an extension of a platform for agent development, which serves as basis for conducting empirical studies with the aim of assessing different aspects of the proposed framework. The results show that our reusable framework and its underlying techniques are able to provide agents and multiagent systems with the abilities required to carry out the remediate, diagnose and recover operations specified by the D^2R^2 + DR strategy in different domains. The autonomy and adaptability provided by our proposal are also demonstrated.

Keywords: Remediation. multiagent systems. self-adaptation. resilience.

Gerenciamento Automatizado de Comportamento Remediativo

RESUMO

Atualmente, diversos sistemas de software são construídos como conjuntos de agentes que interagem entre si e estão situados em um ambiente. Por realizarem uma gama de tarefas críticas e complexas, é esperado que estes sistemas sejam capazes de resistir e se recuperar de situações que possam comprometer sua operação e a qualidade dos seus serviços. D^2R^2 + DR é uma estratégia existente para fornecer esse comportamento remediativo à sistemas de software. Ela especifica a execução de operações como a detecção de problemas, sua remediação, o diagnóstico de suas causas, e a recuperação do sistema a um estado normal de operação. Dada a sua natureza abstrata, instanciações desta estratégia não podem ser reutilizadas em diferentes domínios de aplicação. Mesmo que existam abordagens que forneçam soluções concretas para algumas dessas operações, elas apresentam limitações como falta de autonomia e adaptabilidade. Nesta tese, propomos um *framework* que visa fornecer resiliência a sistemas multiagentes por meio do gerenciamento automatizado das operações de remediação, diagnóstico e recuperação em face de eventos adversos. Esse *framework* dissocia o comportamento resiliente do código dependente de domínio, promovendo assim o reuso de software entre aplicações distintas, e compreende três técnicas. A primeira automatiza o gerenciamento de ações remediativas e o diagnóstico e solução das causas de problemas. A segunda especifica um protocolo de interação e papéis que descrevem como agentes podem coordenar suas ações e compartilhar informações para diagnosticar causas de problemas. Por fim, a terceira formaliza uma abordagem que permite que agentes desfaçam os efeitos de ações executadas sem a necessidade de uma declaração explícita de quais são essas ações e como elas devem ser revertidas. Essas técnicas são implementadas como uma extensão de uma plataforma de desenvolvimento de agentes, a qual serve como base para a condução de estudos empíricos com o objetivo de avaliar diferentes aspectos do framework proposto. Os resultados mostram que o framework reutilizável e as técnicas subjacentes são capazes de fornecer a agentes e sistemas multiagentes as habilidades necessárias para realizar as operações especificadas pela estratégia D^2R^2 + DR em diferentes domínios. A autonomia e adaptabilidade fornecidas pela nossa proposta também são demonstradas.

Palavras-chave: Remediação, sistemas multiagentes, autoadaptação, resiliência.

LIST OF ABBREVIATIONS AND ACRONYMS

ACID	Atomicity, Consistency, Isolation, Durability
BDI	Belief-Desire-Intention
CO	Carbon Monoxide
DDoS	Distributed Denial of Service
FHC	Failure-Handling Component
LOC	Lines of Code
MAS	Multiagent System
QoS	Quality of Service
StAC	Structured Activity Compensation

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

A wide range of modern software systems are nowadays built as multiagent systems (MAS) (JENNINGS, 2001). They are composed of distributed autonomous components that are situated in an environment. These components, also called *agents*, interact and collaborate with each other in order to carry out a wide range of tasks. Examples of such tasks are the management of power plants (HERNANDEZ et al., 2013), autonomous vehicles (HUANG; NITSCHKE, 2020), unmanned aerial vehicles (INSAURRALDE, 2014), and healthcare systems (IQBAL et al., 2016). The complexity and criticality of the tasks performed by these multiagent systems push the need for systems able to operate satisfying the required quality levels for as long as possible, resisting and recovering from abnormal situations (DOBSON et al., 2019; SANCTIS; BUCCHIARONE; MARCONI, 2020). Therefore, it is crucial to adopt techniques to make them resilient.

Resilience can be broadly understood as the system's ability to return to a normal operating condition after the occurrence of disrupting events (STRIGINI, 2012; HOSSEINI; BARKER; RAMIREZ-MARQUEZ, 2016). It combines ideas from many disciplines, such as fault tolerance (AVIZIENIS et al., 2004) and survivability (FISHER et al., 1997). For instance, a fault tolerant system, which is a system able to tolerate the existence of failures in a way they do not incur in errors that interfere in its operation, is typically provided with redundancy techniques (CREVELING, 1956; LYONS; VANDERKULK, 1962). This system can thus resist to a failure in a component by replacing it with a similar one, which is able to carry out the same task. However, there are situations in which this strategy only is not enough. In case of a failure caused by a malicious attack, no matter how many times a component is replaced, the failure will still occur. In these situations, the system may benefit from the diversity of alternative solutions proposed by survivability techniques (ELLISON et al., 2002). In this way, resilience can be achieved at different levels and in different manners.

An existing strategy of resilience, named $D^2R^2 + DR$, specifies a set of operations to be executed by systems in order to become resilient (STERBENZ et al., 2010; STERBENZ et al., 2013). Among these operations, there are the *detection*, *remediation*, *diagnosis*, and *recovery* from a given problem. Detecting problems consists of the identification of an unexpected behaviour or degradation in the system performance. Remediating them, in turn, corresponds to the minimisation of their impact in the services provided by the system. Diagnosing a problem implies uncovering its root cause, which allows corrective measures to be taken, thus preventing further failures and keeping the system operational. Finally, recovering refers to the system's capability of returning to a normal operation as soon as problems are solved.

Even though it provides design guidance, the D^2R^2 + DR strategy is still a conceptual model and its instantiation remains largely application specific (SCHAEFFER-FILHO et al., 2012; CARVALHO et al., 2018). It means that systems or their components that implement this strategy do it specifically focused on the domain or application to which they are designed. As a result, the practice of code reuse is compromised and new implementations must be made from scratch. There is thus a gap between the specification of this strategy of resilience and its realisation.

This gap is reduced by existing solutions able to carry out individual operations that a system must perform (UNRUH et al., 2005; WANG et al., 2018). However, these solutions bring limitations such as the lack of autonomy and adaptability. For instance, there are many approaches able to detect anomalous behaviour in software systems (CHANDOLA; BANERJEE; KUMAR, 2009). However, these approaches are not sufficient to diagnose the root cause of these abnormalities. This task is typically carried out by experts through the manual inspection of execution-related information obtained from different sources, such as tracing and profiling tools (ZHOU et al., 2018). It also occurs with the management of remedial actions, which are typically hard-coded and manually coordinated (HAN; LEI, 2012; NUNES; SCHARDONG; SCHAEFFER-FILHO, 2017). Performing these tasks is usually impractical for large-scale and dynamic systems, because of the amount of data to be analysed or the complexity of the managed system. Consequently, the performance of systems in face of situations that require an immediate response is compromised. Furthermore, this dependency on experts and developers goes against the increasing need for autonomic systems, which are expected to be able to overcome and adapt to challenging situations without the need for human intervention (BARESI; NITTO; GHEZZI, 2006).

Given this context, in this thesis we investigate ways of providing MAS with the abilities required to autonomously manage the remediation of problems, the diagnosis of their causes, and the recovery of the system to a normal state of operation. In next section, we detail the problem we are looking at in this thesis, as well as the limitations of existing work. In Section 1.2, we introduce the proposed solution and give an overview of the contributions of this work. Finally, Section 1.3 presents the structure of the remainder of the thesis.

1.1 Problem Statement and Limitations of Existing Work

As previously introduced, there are two main issues related to the adoption of the D^2R^2 + DR strategy of resilience by MAS. The first refers to the gap between its specification and implementation. The second, corresponds to the lack of autonomy and adaptability that comes with the adoption of solutions able to carry out the operations required by this strategy. Based on these problems, the research question considered in this thesis is stated as follows.

Research Question. *How to provide multiagent systems with the ability of remediating, diagnosing and recovering from abnormal situations in a domain-neutral and reusable way?*

Limitations of existing work related to this research question are as follows.

Lack of software reuse. Despite having a definition that allows its adoption in many domains, the D^2R^2 + DR strategy does not provide guidance on its practical implementation (STERBENZ et al., 2010). Many systems able to remediate, diagnose and recover from problems are designed and implemented to handle specific domains and applications (NOLAN et al., 2016; NUNES; SCHARDONG; SCHAEFFER-FILHO, 2017). This specificity makes these implementations not reusable. The lack of reusable resources that support the development of this resilient strategy requires it to be implemented from scratch, which becomes a limitation. It not only demands higher effort, but also makes implemented solutions more susceptible to bugs when compared to their development using reusable assets such as frameworks, libraries and code snippets (RAVICHANDRAN; ROTHENBERGER, 2003; MOHAGHEGHI; CONRADI, 2007). The development of reusable assets that encapsulate some of the operations specified by the D^2R^2 + DR strategy would reduce both the effort required by new systems to adopt it as well as the probability of bugs in recently implemented solutions. In addition, it would promote software reuse across different domains and applications.

Limited adaptability. Many systems that implement the D^2R^2 + DR strategy have their behaviour implemented in a rigid way (CARVALHO et al., 2018). It means that the sequence of actions executed by these systems to remediate, diagnose and recover from problems is hard coded and specified in advance by developers. It becomes a limitation when we consider the dynamic characteristics of many scenarios in which resilient systems are deployed (ABREU et al., 2017; JANUáRIO; CARDOSO; GIL, 2019). In these scenarios, the system behaviour must be frequently adapted to changes sometimes unpredicted at design time. An example of adaptation comprises selecting the best action to remediate a problem according to existing preferences (FACCIN; NUNES, 2015). The inability to perform adaptations at runtime may compromise the operation of the system in face of challenging situations. An alternative to deal with this limitation is providing systems with self-adaptive capabilities (DOBSON et al., 2006) that allow them to independently manage their actions at runtime.

Limited autonomy. In many existing solutions, tasks such as the coordination of remedial actions (CARVALHO et al., 2018) and the diagnosis of problem causes (ZHANG; LIN; HSU, 2007) end up being carried out by humans. This goes against the increasing need for autonomy in software systems (BARESI; NITTO; GHEZZI, 2006). Manually coordinating the system behaviour requires developers to modify particular portions of code in order to insert or remove sequences of actions, or even specify their reversion in a recovery context. The diagnosis of causes, in turn, requires the manual inspection of data by experts (ZHOU et al., 2018). These tasks are impractical in complex, large-scale systems. They become time consuming and error prone due to the complexity and amount of data to be considered, which can prevent systems to be adapted in a timely fashion. The inadequate coordination of actions may affect not only a particular functionality, but the entire system as well. An alternative to handling this limitation is providing solutions able to reduce the need for human configuration, management and intervention, thus increasing the autonomy of systems that adopt them.

1.2 Proposed Solution and Contributions Overview

Software reuse assumes the existence of assets that can be reused on the creation of new assets or the modification of existing ones (FRAKES; KANG, 2005; MOHAGHEGHI; CONRADI, 2007). Frameworks are a common example of such reusable assets. They implement and make available functionalities that are shared by applications across different domains. Systems that implement the D^2R^2 + DR strategy share a specific behaviour, which is performed after the detection of abnormal events or conditions. First, they act in order to remediate the detected issue, which implies the selection of the best action to be executed. In the meantime or after remediating these issues, the diagnosis of what caused their occurrence takes place, followed by their resolution. After the definitive solution of problems and their causes, a recovery step is performed. In this stage, the effects of remedial actions able to be undone are reverted. This shared sequence of steps gives an insight of what can be encapsulated into a reusable asset like a framework.

Because D^2R^2 + DR relies on the existence of components able to adapt to changes in their environment (DOBSON et al., 2019), its adoption by software agents becomes particularly convenient. An approach suited for the development of agents is the Belief-Desire-Intention (BDI) (RAO; GEORGEFF, 1995) architecture. This architecture structures agents in terms of three key components: (i) *beliefs*, which represent the current state of the agent and its environment; (ii) desires, also called *goals*, which correspond to states the agent wants to reach; and (iii) *intentions*, which are goals an agent is committed to achieve and for which it has plans capable of achieving them. These components are integrated into a reasoning cycle and manipulated by several functions that have, among other roles, the responsibility of selecting which actions the agent will perform. However, these functions are abstract in the BDI architecture and can be customised in specific applications to provide a desired behaviour. By taking advantage of this characteristic, it is possible to provide a solution in which the coordination of remedial actions as well as the diagnosis and resolution of problem causes, and the recovery of the system are completely undertaken by the BDI reasoning cycle.

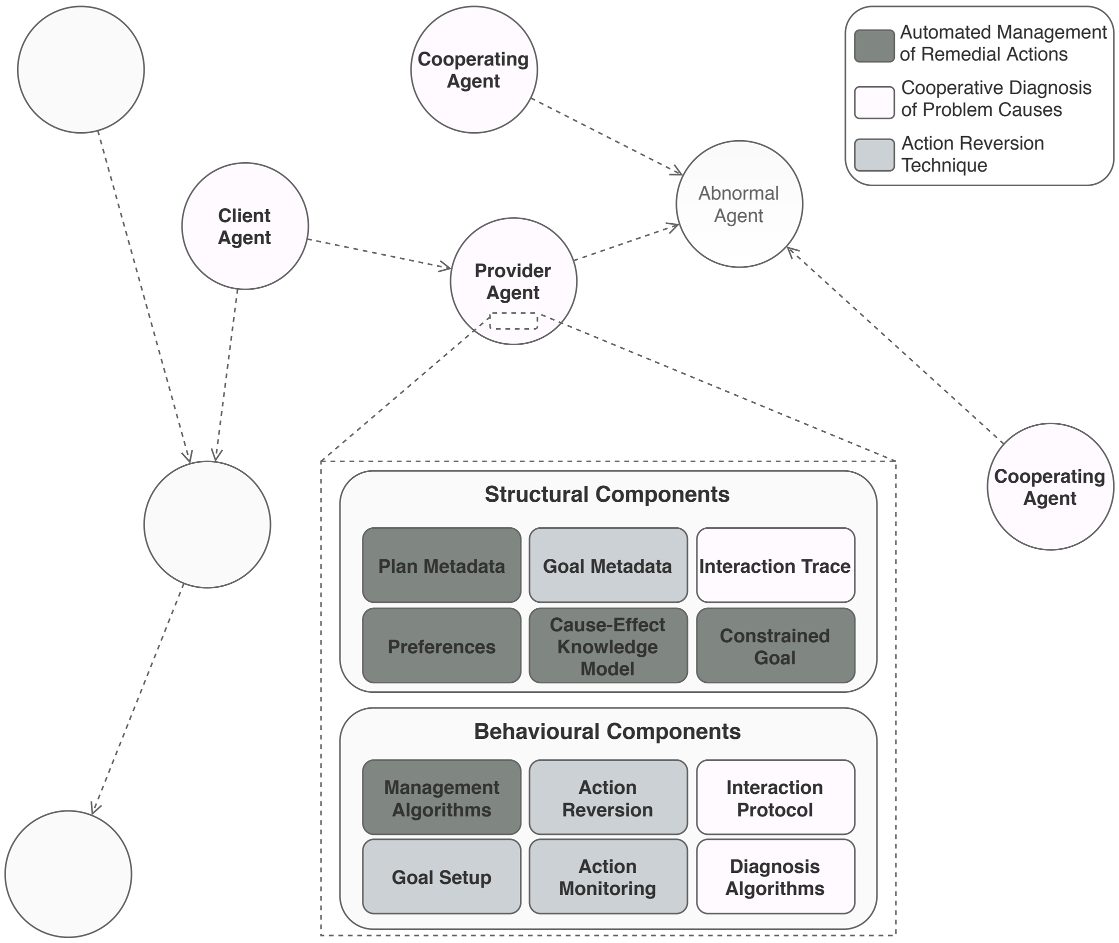
In this thesis, we thus investigate the following research hypothesis.

Research Hypothesis. *A domain-independent framework that extends the BDI architecture is an effective solution to realise the remediation, diagnosis and recovery steps specified by the D^2R^2 + DR strategy, promoting software reuse across different application domains while providing autonomous and adaptive capabilities to agents and multiagent systems implemented with it.*

We thus propose a domain-independent framework that extends the BDI architecture and provides the backbone for the implementation of software agents and multiagent systems able to carry out the D^2R^2 + DR strategy of resilience. This framework specifically focuses on the remediation, diagnosis, and recovery operations.

The proposed framework implements a set of techniques developed with the aim

Figure 1.1: Proposed Framework and Underlying Techniques.



of supporting agents on the automated coordination of their actions, whether they are for remediating problems, diagnosing their root causes, or recovering to a normal operation. Figure 1.1 presents an overview of our proposal. It depicts the structural and behavioural components that comprise each technique. The main contributions of this thesis are described as follows.

i) A **technique for the automated management of remedial actions**, which is responsible for coordinating agent behaviour with the aim of effectively handling challenging events. This technique is composed of a structural metamodel and control algorithms responsible for determining how problems are remediated, and the diagnosis and solution of their causes are carried out. It also specifies a model for representing cause-effect relationships, which is used in the diagnosis process. These structural and behavioural components, which are part of each agent, are depicted as dark grey rectangles in Figure 1.1.

ii) A **technique for the cooperative diagnosis of problem causes** in MAS that allows the creation and update of cause-effect relationship models at runtime. This technique comprises an interaction protocol and algorithms that specify the behaviour of agents while playing different roles. Structural and behavioural components are depicted in Figure 1.1 as white rectangles, while agents playing the specified roles are depicted as white circles.

iii) The formal specification of a **technique for reverting actions**, which allows agents to undo the effects of remedial actions after the complete solution of the problems for which they were executed. This functionality is particularly suited for scenarios in which valuable resources are allocated for remediating a problem but must be released as soon as possible in order to allow the execution of further actions. Figure 1.1 depicts components comprising this technique as light grey rectangles.

These techniques are implemented as part of an agent development platform named BDI4JADE (NUNES; LUCENA; LUCK, 2011), which implements the BDI architecture. They are evaluated in separate studies whose results show that the proposed framework is effective for developing agents and multiagent systems able to remediate, diagnose and recover from challenging situations. There is evidence of its reusability across several applications in different domains. Finally, the adaptive and autonomous behaviour presented by developed multiagent systems demonstrate the ability of the framework on providing such capabilities.

1.3 Outline

The remainder of this thesis is organised as follows. Chapter 2 presents theoretical background that serves as the foundations needed for this thesis. Related work that addresses the management of remedial actions, the diagnosis of problem causes, and the reversion of actions as well as their limitations are discussed in Chapter 3. Chapter 4 introduces the technique for automated management of remedial actions, detailing the structural metamodel and control algorithms that comprise it, as well as the cause-effect relationship model. The cooperative technique for diagnosing the root cause of problems and the technique for reverting actions are presented in Chapters 5 and 6, respectively. Finally, Chapter 7 concludes this thesis and outlines directions for future work.

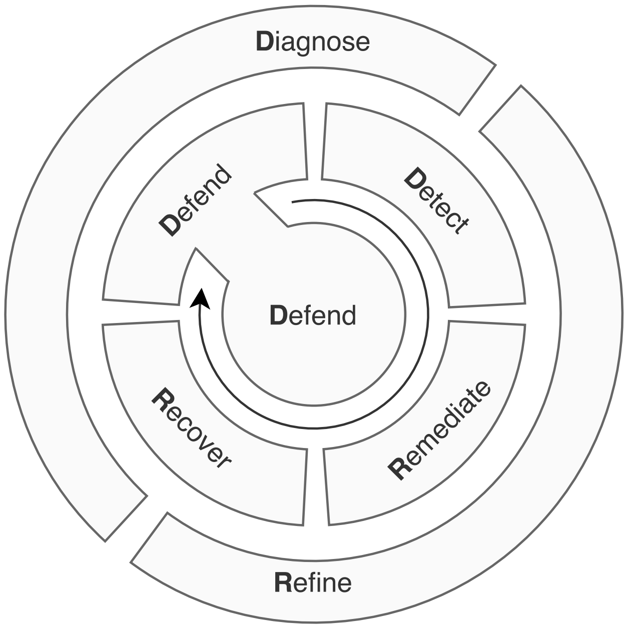
2 BACKGROUND

Before introducing our development framework and its underlying techniques, in this chapter we detail the resilience strategy that specifies the remedial behaviour that is the focus of this thesis. We also provide an overview of the Belief-Desire-Intention (BDI) architecture, which is used as a basis for our framework. The strategy, named D^2R^2 + DR, is presented in Section 2.1, while the BDI architecture is presented in Section 2.2.

2.1 The D^2R^2 + DR Strategy

The D^2R^2 + DR strategy is a conceptual model initially designed to promote a resilient behaviour in the context of networked systems (STERBENZ et al., 2010). Nevertheless, its abstract ideas can be applied to software systems in general. This strategy proposes the integration of a structural core and two active phases. The first phase occurs in real time and is responsible for defending, detecting, remediating an recovering the system from disruptive events. The second one, in turn, occurs on background and aims at diagnosing the cause of detected issues and refining the system behaviour in order to prevent further problems. Figure 2.1 depicts the D^2R^2 + DR strategy.

Figure 2.1: The D^2R^2 + DR Strategy (STERBENZ et al., 2010).



The core of this strategy comprises passive defences implemented in order to reduce the likelihood of faults. These defences are usually associated with structural measures such as the use of redundant components (CREVELING, 1956; LYONS; VAN-

DERKULK, 1962).

The first active phase, the D^2R^2 part, comprises the *defend*, *detect*, *remediate*, and *recover* operations. Defending has the same purpose and is directly related to the strategy's defence core. The difference is that, while the latter provides passive defences, the former is in charge of the active ones. These active defences consist of self-protection mechanisms designed to anticipate and prevent the occurrence of issues. An example is the filtering of known attack signatures in a network (MENG, 2018). In cases in which existing defence mechanisms are unable to prevent faults to take place, the system must be able to recognise their occurrence in a timely fashion. The detect operation holds that responsibility. The adoption of one of the many existing anomaly detection techniques (CHANDOLA; BANERJEE; KUMAR, 2009) is an example of how this operation can be carried out. The remediate operation is performed to mitigate the effects of detected issues. Its goal is to keep the system operating at an acceptable level as long as the adverse conditions persist. Limiting the traffic of an overused link in order to keep it working is an example of how to accomplish this task (SCHAEFFER-FILHO et al., 2012). Usually, remedial actions result in a graceful degradation of system operation or in an increase in its cost. However, if no further action is taken the system can remain in a degraded state or consuming valuable resources even after the adverse condition is under control. That is the reason for the existence of the recover operation, which is responsible for revoking the negative effects of remedial actions in order to return the system to its normal state of operation. These four steps interact in a cycle, which is triggered by the detection of an issue and can be executed simultaneously in different parts of the system.

The second phase comprises the *diagnose* and *refine* operations. They are usually performed offline and strongly rely on human intervention. The diagnose operation aims at identifying what led to the occurrence of a problem. It involves root cause analysis (WILSON, 1993) and allows corrective measures to be taken in order to handle the source of problems. Typically, this task is performed by experts through the manual inspection of data obtained from monitoring tools (ZHOU et al., 2018). Finally, the refine operation has the goal of enhancing this resilient behaviour based on past experience. This refinement can involve, for instance, the implementation of new defence mechanisms able to deal with diagnosed faults.

The framework presented in this thesis proposes techniques to automate the remediation, recover and diagnose operations. They are implemented as extensions of the reasoning cycle specified by the BDI architecture, which is presented in next section.

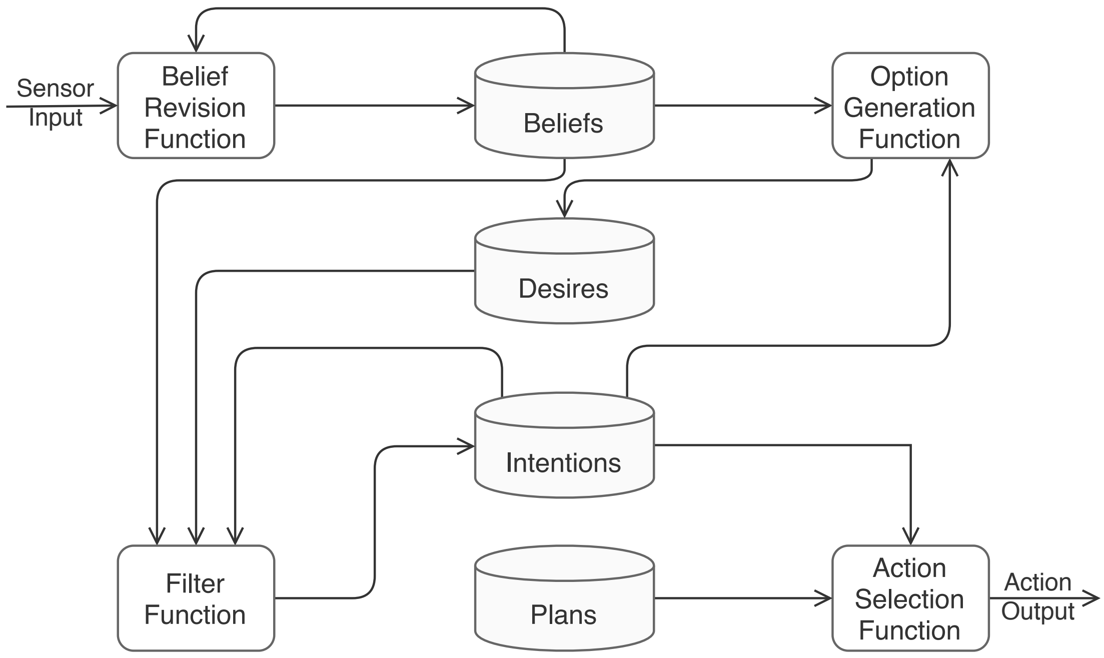
2.2 The Beflief-Desire-Intention Architecture

The Belief-Desire-Intention (BDI) architecture (RAO; GEORGEFF, 1995) is perhaps one of the most widely used approaches for developing autonomous agents. It is particularly suited when a flexible and robust behaviour is required. This architecture has its foundations on the concepts of practical reasoning (BRATMAN, 1987) and intentional systems (DENNETT, 1987), and comprises the base for several languages and platforms, such as Jason (BORDINI; HüBNER; WOOLDRIDGE, 2007) and BDI4JADE (NUNES; LUCENA; LUCK, 2011). Agents built based on the BDI architecture are structured in terms of three mental attitudes that name the approach. *Beliefs* are the informative components of the system. They represent the view agents have about their own state and the state of the environment they are inserted. It is not referred to as attribute, which is the term used to capture state in object orientation, because beliefs are assumed to possibly be inaccurate due to, e.g., noise in environment perceptions. Desires, or *goals*, specify the states of the world the agent wants to reach, while intentions arise from the agent commitment to achieve these goals. Desires intended to be achieved are carried out by the execution of predefined sets of actions, the so-called plans.

The flexible behaviour that characterises this architecture is achieved when beliefs, desires and intentions are integrated with four abstract functions into a reasoning cycle, which is depicted in Figure 2.2. The *belief revision function* is responsible for updating agent beliefs. It is triggered by the perception of internal or external events, such as messages received from other agents, measurements made by sensors and changes caused by agent actions. The *option generation function*, in turn, is responsible for updating the set of agent goals. This updating task comprises not only generating new goals but also dropping those that were achieved, are no longer desired or may become unachievable according to current beliefs. The *filter function* is in charge of generating new intentions through the filtering of existing goals. This task is associated with the deliberation process, i.e. the process of deciding what goal to achieve next. Finally, the *action selection function* is responsible for selecting the plan that will be executed in order to carry out a given intention. This function identifies a set of relevant plans able to handle the selected intention and chooses one according to a given selection strategy, which is thus executed. This process is related to the so-called means-end reasoning, i.e. the process of deciding how a goal will be reached.

To exemplify these concepts, consider the following situation. A person has the

Figure 2.2: The Belief-Desire-Intention (BDI) architecture (WOOLDRIDGE, 1999).



intention to commute to work. For this purpose, she can walk, ride a bike, or take a bus. These are her plans. Before leaving home, her belief about the weather is updated when she notices that it is raining. Given the current context, she thus decides that taking a bus is the best option to carry out her intention. That is the plan she executes to successfully reach her final destination. All the reasoning process leading to that result is abstracted by the functions comprising the BDI reasoning cycle.

Although able to provide a flexible and intelligent behaviour by default, these functions can be customised in order to provide desired characteristics. This customisation may include the use of sophisticated techniques, such as the selection of plans according to agent preferences (PADGHAM; SINGH, 2013; VISSER et al., 2016), for example. The framework proposed in this thesis implements techniques that extend some of these functions. Specifically, an action selection function that is able to choose remedial plans when needed, and an option generation function capable of identifying causal relationships, generating goals related to the solution of problem causes, and generating goals in order to revert actions.

2.3 Final Remarks

To provide systems with resilience, the D^2R^2 + DR strategy specifies a cycle that comprises, among other operations, the remediation of problems, the diagnosis of their causes, and the system recovery to a normal operating state. Performing these operations requires components to be able to adjust their behaviour to different conditions. An approach that supports the development of agents with this adaptive characteristic is the BDI architecture. It specifies agents in terms of three major concepts that are integrated with four functions into a reasoning cycle. Because these functions are abstract, they can be customised in order to implement operations such as those specified by D^2R^2 + DR strategy. The next chapter presents existing approaches that implement these operations in different contexts.

3 RELATED WORK

A substantial amount of effort has been directed towards the development of approaches focused on the remediation of problems, the diagnosis of their root causes, and the reversion of actions. However, these operations are rarely integrated into a single cohesive solution. In this chapter, we present related work categorised according to their main goals. Section 3.1 presents work focused on the remediation of adverse events. Section 3.2 introduces work related to root cause identification. Finally, Section 3.3 discusses approaches focused on the reversion of executed actions.

3.1 Management of Remedial Actions

Many approaches focused on resilience implement solutions whose goal is to remediate the effects of problematic events. A common challenge addressed in the context of network systems are distributed denial of service (DDoS) attacks. In this type of attack, a massive amount of requests is sent to a target component in order to overload it. This compromised state makes components unable to provide their services as expected if operating under normal conditions. The approach of Schaeffer-Filho et al. (2012) allows systems to minimise the impact of this kind of threat through the adoption of event-condition-action policies. In that solution, many activities to incrementally handle DDoS attacks are manually coordinated. From the conditions monitored by the network infrastructure, components such as rate limiters, flow exporters and classifiers are able to raise events that trigger the execution of operations to eventually limit the traffic of specific attack flows. The proposal of Carvalho et al. (2018) follows the same policy-based strategy. It describes the use of three different policies that are specified in advance. These policies describe the events that trigger a set of actions to be taken by the system as well as the context in which they can be executed. Nunes, Schardong and Schaeffer-Filho (2017) extend these approaches with an agent-based solution designed to reduce the need for a pre-specified arrangement of components and their interactions. In that solution, named BDI2DoS, components are associated with BDI agents able to act proactively in order to deal with DDoS attacks. While the two former solutions require component interactions to be anticipated and explicitly declared, the latter provides a resilient behaviour that emerges from agent cooperation. Nevertheless, because its implementation is still focused on a single application domain, reusing BDI2DoS becomes unfeasible.

The work of Januário, Cardoso and Gil (2019) also builds up on the concept of software agents. It proposes an architecture for cyber-physical systems in which agents with different functionalities are associated with each component of the system. These agents are responsible for ensuring system operation in case of communication failure, device malfunctioning or even used-induced errors. This task is carried out through the adoption of policies that must be specified according to the domain in which systems are deployed. The coordination of remedial actions thus arises from the interaction of agents that follow these policies. Similarly, the proposal of Abreu et al. (2017) presents a dedicated component responsible for coordinating remedial mechanisms spread throughout systems in the context of the Internet of Things. That component relies on specific modules for monitoring, protecting and recovering the system. These modules can be seen as achievers of the operations specified by the D^{2}R^2 + DR strategy. Because both solutions present an increased level of abstraction, they still require a manual specification of how policies will be carried out (JANUáRIO; CARDOSO; GIL, 2019) or how remedial actions will be coordinated by the designed coordinating component (ABREU et al., 2017).

Examples of more practical solutions are available in many different scenarios. Raiciu et al. (2011), for instance, demonstrate how the Multipath Transmission Protocol (FORD et al., 2013) can be used to coordinate the response of a network to link congestion and failures. This communication protocol enables the existence of alternative communication flows in the same connection between nodes in a network. When a communication flow is affected by external interference, such a disruption can be remediated by transferring the network traffic to a fully operational flow. The proposal of Nolan et al. (2016), in turn, aims at mitigating communication failures between sensors and their controllers, which may result on information loss. This solution is triggered by a simple *if-then* rule, which is embedded in sensors and states that if data could not be sent, it is cached. Once the communication problem is solved, cached data is sent to the corresponding controller, thus allowing the retrieval of missing information.

This diversity of approaches illustrates the many ways the coordination of remedial actions can be achieved. Nevertheless, existing solutions able to perform this task are either abstract, in a way that do not provide practical implementations, or specific to the challenge being addressed, which results in limited reusability. Even though approaches that rely on domain-specific rules can be reused across different applications in order to solve similar problems, their implementations do not share this fate. Most of these implementations also requires interactions among system components to be manually specified, thus increasing the need for human intervention.

3.2 Root Cause Diagnosis

The concept of Bayesian networks (PEARL, 2014) is frequently used as a starting point for the development of several approaches that focus on the identification of event causes. In these networks, variables of interest are represented as nodes while causal relationships are depicted as edges. To create such a network, the accountability framework proposed by Zhang, Lin and Hsu (2007) maps services and their dependencies to nodes and edges, respectively. A probabilistic analysis is thus performed on the generated structure with the aim of identifying the origin of service level agreement violations. Carrying out this process, however, depends on knowing the topology of service relationships in advance, which hinders the adoption of this solution for dynamic systems.

The CloudRanger framework (WANG et al., 2018) has a similar workflow. At runtime, it creates a network representing the impact services have on others, and uses it as the foundation to compute the correlation between linked services regarding a common cause. Traversing this network according to computed correlations allows one to rank the candidate root causes of a given abnormality. Parida, Marwala and Chakraverty (2018), in turn, proposed a domain-independent solution in which the concept of causal influence factor is introduced as a parameter to be used to identify not only the underlying causal structure of systems but also the directions of existing causal relationships. Despite allowing, the discovery of root causes in dynamic environments with higher accuracy, these proposals do not handle scenarios in which the data are decentralised. A solution that explores the cooperation between autonomous agents is proposed to handle this issue (MAES; MEGANCK; MANDERICK, 2007). It assumes that agents do not have a global view of the system, and thus are not able to observe the entire set of domain variables. This solution proposes a mechanism that allows agents to negotiate the disclosure of useful information, which is later used to create individual causal models and guide adaptation strategies. Even though it targets distributed domains, the feasibility of this approach is not evaluated in scenarios comprising more than two agents.

Despite not being designed for identifying the root cause of problems, the dynamic, adaptive modelling of the behaviour of cloud-based systems, as proposed by Chen and Bahsoon (2017), comprises an interesting alternative to achieve such a goal. Their focus is on predicting the value of Quality of Service (QoS) features based on environmental conditions and other relevant variables. By correct modeling the impact of these variables on QoS features, it becomes possible to identify the cause of quality requirement violations even before their occurrence. The work of Mendonça, Ali and Rodrigues (2014) follows a similar path. It proposes an approach for modelling and analysing contextual failures and their implications. The evaluation of this model at runtime allows the system to identify possible contextual effects and act in order to adapt to them.

Many techniques for root cause diagnosis are designed to handle distributed environments in a centralised manner. Even though they are capable of carrying out this task in controlled scenarios, these techniques are not suited for environments such as those in which multiagent systems are deployed because agents have only a partial view of the system. Alternative solutions that require causal models to be provided in advance by experts also present limitations. Changes in the environment need to be either anticipated by experts or manually introduced in existing models. It hinders system's adaptability and can present scalability issues when adopted in large scale scenarios.

3.3 Reverting Actions

Mechanisms that focus on the reversion of actions had been carried out mainly in the context of ACID transactions (GRAY; REUTER, 1992). A transaction is said to be ACID if the actions it comprises can only be successfully achieved or not achieved at all (*atomic*), and their execution leads to a correct transformation of the state of the system (*consistent*). Additionally, the execution of an ACID transaction cannot be influenced by the execution of others (*isolated*) and, once it is successfully completed, its results must survive to system failures (*durable*). In systems that adopt this concept of ACID transactions, reversion is usually performed in failure-handling situations, in which *rollback* mechanisms are used to restore the system to a state identical to that in which the system was before the execution of the actions being reverted.

Although applicable in fully controlled environments, this rollback process becomes impracticable in real world situations due to the dynamism and non-determinism to which actions are subjected. Korth, Levy and Silberschatz (1990) aimed at addressing this issue by formalising the concept of *compensating transactions*, which are those performed with the objective of reverting the effects of transactions that may or may not be completed. The main difference between the typical rollback process and the use of compensating transactions is that the latter does not necessarily restore the system to the same state it had before. Instead, it focuses on leading the system to a state that is semantically similar to its previous state and that becomes acceptable in the given context.

This same notion of compensation was used by Butler and Ferreira (2000) in the development of a textual business process modelling language called StAC (Structured Activity Compensation). In StAC, a system is specified as a set of equations that describe the execution order of the system actions. Within these equations, actions can be related to their compensating counterparts, whose need for execution is explicitly specified through the use of particular symbols. Chessell et al. (2002) extended this language with the concepts of selective and alternative compensation, in which subsets of available compensating actions can be selected for execution according to the context. In both versions of StAC, contrary to what occurs with ACID transactions, the need for compensating actions is not based on system failures, but explicitly determined by the system.

Similarly, in the work of Unruh, Bailey and Ramamohanarao (2004), the management of compensating actions is assigned to a particular system component, called FHC (failure-handling component). Compensating elements are thus associated with system goals instead of actions. In case of failure, the FHC is responsible for determining which and when these elements will be triggered. How they will be carried out, however, is decided by the system. A distributed version of this approach was presented in a subsequent work (UNRUH et al., 2005).

The main issue of most of these approaches concerns the lack of adaptation regarding the specification of what must be achieved when actions are reverted. Using StAC (BUTLER; FERREIRA, 2000; CHESSELL et al., 2002), it is not possible to explicitly determine which state a system must exhibit after compensating an action. If this state varies according to the context, one must specify different compensating actions to address all its possible instances. Moreover, these actions would have to be specified in a way that the most convenient would be performed according to the current context. In the work of Unruh, Bailey and Ramamohanarao (2004), in turn, how the system must look like is predefined and does not change at runtime. The problem is that, due to their dynamic nature, agent-based systems may adjust their needs regarding what must be achieved in order to accommodate changes in their environment. This characteristic demands from these systems the ability to dynamically determine the desired outcomes of compensating actions before they are selected and executed, which cannot be done when these outcomes are predefined at design time.

There are reverting mechanisms particularly developed to handle failures in systems structured with BDI agents, which are considered in our work. The Jason platform, for example, allows the specification of "clean up" plans (HÜBNER; BORDINI; WOOLDRIDGE, 2006), which are executed when the plans to which they are related fail to reach their goals. Developers are thus able to specify how the changes performed by failed plans must be reverted, and even whether goals must be reattempted or not. A similar approach concerning plan-aborting situations was proposed by Thangarajah et al. (2007). Their work specifies the semantics of a plan-aborting mechanism in which plans can be associated with aborting actions, which are thus able to revert the effects of plan executions when they are interrupted.

Techniques focused on the BDI architecture present a lack of adaptation similar to those approaches discussed previously. However, instead of limitations regarding the specification of *what* may be accomplished when reverting actions, their issues are associated with *how* this task is performed. In these approaches, a course of action is specified at design time and individually related to the plan whose effects it must revert. One of the main features of BDI agents, however, is their ability to select the most suitable plan to be executed in a given context. By constraining a compensating plan to a single course of action, alternative (and potentially better) reverting solutions may never be tried.

3.4 Final Remarks

Operations such as the remediation of problems, the diagnosis of their root causes, and the recovery of the system to a normal operating state, which are specified by the D^2R^2 + DR strategy, are individually carried out by many existing approaches. Nevertheless, these approaches present limitations that prevent their (re)use by multiagent systems and narrow the autonomous and adaptive capabilities of systems that adopt them. Implementations of solutions that coordinate remedial actions, for instance, are *specific* to the challenge being addressed. The identification of the root cause of problems in distributed domains, in turn, is performed mainly in a *centralised* manner. The only exception being a work that presents a formal approach to handle *distributed* scenarios, but whose differences to our proposed solution become explicit in the remainder of this thesis. Finally, besides being *conceptual* in their majority, solutions for reverting actions require the actions to be reverted and how it must be done to be *manually coordinated* in advance, thus reducing the ability of systems to dynamically adapt to different situations.

Table 3.1: Comparison of Related Work.

	Work	Specificity	Abstraction	Control	Organisation	Behaviour
Remediation	Schaeffer-Filho et al. (2012)	Application-specific	Concrete	Decentralised	Distributed	Manually-coordinated
	Carvalho et al. (2018)	Application-specific	Concrete	Decentralised	Distributed	Manually coordinated
	Nunes, Schardong and Schaeffer-Filho (2017)	Application-specific	Concrete	Decentralised	Distributed	Autonomous
	Januário, Cardoso and Gil (2019)	Application-specific	Conceptual	Decentralised	Distributed	Autonomous
	Abreu et al. (2017)	Application-specific	Conceptual	Centralised	Distributed	Autonomous
	Raiciu et al. (2011)	Application-specific	Concrete	Decentralised	Distributed	Autonomous
	Nolan et al. (2016)	Application-specific	Concrete	Decentralised	Distributed	Manually coordinated
Diagnosis	Zhang, Lin and Hsu (2007)	Domain-neutral	Concrete	Centralised	Distributed	Autonomous
	Wang et al. (2018)	Domain-neutral	Concrete	Centralised	Distributed	Autonomous
	Parida, Marwala and Chakraverty (2018)	Domain-neutral	Concrete	Centralised	NA	NA
	Maes, Meganck and Manderick (2007)	Domain-neutral	Concrete	Decentralised	Distributed	Autonomous
Recovery	Korth, Levy and Silberschatz (1990)	Domain-neutral	Conceptual	Centralised	NA	NA
	Butler and Ferreira (2000)	Domain-neutral	Conceptual	Centralised	NA	NA
	Chessell et al. (2002)	Domain-neutral	Conceptual	Centralised	NA	NA
	Unruh, Bailey and Ramamohanarao (2004)	Domain-neutral	Conceptual	Centralised	Monolithic	Manually coordinated
	Unruh et al. (2005)	Domain-neutral	Conceptual	Centralised	Distributed	Manually coordinated
	Hübner, Bordini and Wooldridge (2006)	Domain-neutral	Concrete	Decentralised	Distributed	Manually coordinated
	Thangarajah et al. (2007)	Domain-neutral	Conceptual	Decentralised	Distributed	Manually coordinated

Table 3.1 summarises the core characteristics of existing work associated with the remediation, diagnosis and recovery operations. Next chapters present alternatives to perform these operations reducing the limitations imposed by existing solutions.

4 MANAGEMENT OF REMEDIAL ACTIONS

In this chapter, we introduce a technique that extends the BDI architecture in order to allow agents to autonomously select the appropriate set of actions (plans) to remediate problems (i.e. achieve a goal) and handle their causes. This extension automates the coordination of agent plans, thus promoting *reuse* across domain-dependent solutions and allowing agents to flexibly decide the best action according to their context, goals and preferences. The extended BDI architecture includes a set of components to capture the domain knowledge required to support agents on making such decisions. This knowledge is used in a customised reasoning mechanism, which selects remedial plans, when needed, and generates goals to search and deal with problem causes.

4.1 Problem and Running Example

Before detailing the elements that comprise our technique, we introduce an illustrative example to provide a better understanding of the scenario we address. It is used throughout this chapter as a running example. Despite its simplicity and perhaps not adequacy of an agent-based solution to this problem, it allows us to clearly illustrate the class of problems we are targeting and concepts of our approach without having to detail additional domain background. Consider the problem of dealing with a ceiling leak. *Alice* is a person who notices a ceiling leak in the room of her house. To deal with it, she has three options (or **plans**): (i) cover the leak with *duct tape*; (ii) use a *towel* to absorb the liquid coming from the ceiling; or (iii) put a *bucket* underneath the drip.

The three available plans can achieve the **goal** of *dealing with the ceiling leak*. However, each of them has particular characteristics concerning how this goal is achieved. Each plan requires a different amount of *time* to be performed. Covering the leak with duct tape, for example, requires much more time than putting a towel under the drip. Plans are also related to different execution *costs*. If Alice chooses to use the towel to achieve the goal, she will have to afford the cost of such towel (assuming that it would be thrown away after being used). If she chooses to use the bucket instead, the cost she has to afford will be lower, given that a bucket can be reused. Therefore, plan executions are directly associated with the consumption of **resources**. Further, considering that the floor can become wet if the leak is not addressed as soon as possible, Alice has **constraints** over the execution time of her plans. In this context, using duct tape to cover the leak may not be a feasible solution. Moreover, every time Alice wants to deal with a ceiling leak, she may have different **preferences** on how to spend resources. For instance, if she wants to address it quickly, the time taken to execute a plan becomes more valuable than its cost.

Although performing one of the available plans achieves the given goal, Alice's problem is not completely solved. The leak is an **effect** of several possible **causes**, e.g. a broken pipe or a cracked roof tile, which must be addressed in order to stop the leak permanently. Moreover, there may be plans that deal with both cause and effect. However, they possibly cannot be executed in such a way that constraints are met. Therefore, assuming that an agent *A* is in charge of resolving this whole problem, two key issues must be addressed: (i) *how can agent A select the most adequate plan, possibly a remedial one, to achieve its goal based on its constraints and current preferences?*; and (ii) *how can agent A diagnose and address the causes of the problem associated with this goal to permanently solve it?*

In the traditional BDI architecture, agent goals are explicitly specified, and plans to reach these goals are provided at design time. However, there is no specified strategy to choose among available plans. Moreover, depending on how the problem is modelled, plans that deal with the effect of the problem may not achieve its causes. Therefore, once a plan achieves the goal associated with the effect, its root causes may remain unaddressed. Finally, the BDI architecture does not include means of diagnosing the causes of problems being tackled.

4.2 Software Agent Architecture

The example presented in Section 4.1 introduces many key concepts, such as resources and preferences. Some of them, such as goals and plans, can be associated with existing components of the BDI architecture. Others, however, are domain-independent concepts that can be incorporated to this architecture to provide agents with the ability of dealing with adverse situations. In these situations, goals correspond to remediating the effects of problems before diagnosing and dealing with their causes, and must be achieved considering a set of constraints. In this section, we first formalise these concepts and then integrate them as an extension to the traditional BDI architecture.

4.2.1 Constrained Goals

Goals express a state of affairs that an agent wants to bring about. However, in order to achieve a goal, the execution of actions consumes *resources*, such as processing time and allocated memory. In some scenarios, achieving this state of affairs is useless for an agent if the amount of consumed resources is not limited to a specified amount. Moreover, in other scenarios, if there are different means of achieving a goal, there may be restrictions on how resources should be spent, e.g. consuming the least as possible of a particular resource. Therefore, we define a particular type of goal, namely *constrained goal*, that incorporates these two notions, which are *operation constraints* and *objective functions*, respectively. These concepts are formalised as follows.

Definition 1 (Resource) *A resource $r_i \in R$ is any consumable supply of asset, whose consumption can be measured by an agent. Each resource r_i is associated with a value $v_c(r_i) \in \mathbb{R}$, which is a consumed amount of r_i.*

Example 1 There are two resources in our running example: time taken to prevent liquids to accumulate in the floor (*time*) and the money spent to do so (*cost*). The resource $v_c(time)$ is the number of seconds passed since the goal of dealing with the ceiling leak was set as a goal, while $v_c(cost)$ is the amount of money spent while performing a set of actions to achieve this goal.

Definition 2 (Operation Constraint) *An operation constraint c is a constraint over the desired consumption of resources. It is expressed with the following grammar.*

$$\begin{aligned} c &::= (c \wedge c) \mid (c \vee c) \mid (\neg c) \mid e \\ e &::= (r\, op\, v) \\ op &::= > \mid \geq \mid < \mid \leq \mid = \mid \neq \end{aligned}$$

where r is a resource and v is a value $v \in \mathbb{R}$. Moreover, an operation constraint is over a single resource r.

Example 2 Assuming that Alice does not want to spend more than \$5 to solve her problem, an operation constraint is $cost < \$5$.

Definition 3 (Objective Function) *$\mathcal{O} : R \nrightarrow \{min, max\}$ is a partial function that specifies whether the consumption of a resource r must be minimised or maximised.*

Example 3 Alice has no specific constraint regarding the amount of time taken to deal with the ceiling leak, but she wants to do so as soon as possible, in order to prevent permanent damage in her floor. Therefore, the resource *time* is associated with *min* to indicate that the minimum amount of time should be spent to accomplish this.

The definitions presented above are then used to define a constrained goal, which is a goal complemented by constraints over resources and specifications regarding how they should be spent. We assume that a goal is specified as proposed in the AgentSpeak language (RAO, 1996), in which there are achievement ($!g$) and test ($?g$) goals. The former indicates that an agent wants g to be a true belief, while the latter to test whether the formula g is a true belief.

Definition 4 (Constrained Goal) *A constrained goal g_c is a tuple $\langle g, C, \mathcal{O} \rangle$, where g is an achievement goal or a test goal; C is a conjunctive set of operation constraints that states restrictions over resources to be consumed to achieve g; and $\mathcal{O}$ is an objective function that states whether the consumption of resources should be minimised or maximised when achieving g.*

Example 4 Alice wants no ceiling drip in her roof. Consequently, she wants to believe that $\neg \mathit{ceiling\_drip}$ is true. Therefore, in order to deal with the ceiling leak, and considering the operation constraints and objective function defined above, she has the following goal: $\mathit{ceiling\_leak} = \langle !\neg \mathit{ceiling\_drip}, \{\mathit{cost} < 5\}, \{\langle \mathit{time}, \mathit{min} \rangle\} \rangle$. This means that this goal must be achieved spending less than \$5, as soon as possible.

In Example 4, C has a single operation constraint. However, given that constrained goals have a conjunctive set of constraints, if there were other constraints, all of them must be satisfied to achieve this goal.

4.2.2 Plan Required Resources

The introduced definitions allow the specification of how goals must be achieved. However, given that plans are those that consume resources while trying to achieve goals, there must be means of specifying how such resources are consumed. This is formalised next.

Definition 5 (Plan Required Resource) *A plan required resource $\mathcal{R}_{req_p} : R \to \mathbb{R}$, which gives, for a resource r, the amount $v \in \mathbb{R}$ that a plan p spends while executing.*

Example 5 Consider the different alternatives, presented in Section 4.1, to achieve the goal *ceiling_leak*. Using a duct tape has a small cost, but takes 600s to be accomplished, thus $\mathcal{R}_{req_{DT}}(cost) = 0.5$ and $\mathcal{R}_{req_{DT}}(time) = 600$, while using a towel costs \$6, but takes 120s to be accomplished, thus $\mathcal{R}_{req_{TP}}(cost) = 6.0$ and $\mathcal{R}_{req_{TP}}(time) = 120$.

Definition 6 (Plan) *A plan p is a tuple $\langle Pre, G, \mathcal{R}_{req_p}, Body\rangle$ where Pre is a set of context conditions in the form of logical predicates that specify the context in which p must be executed, i.e. they must hold true before the execution of p thus being its preconditions; G is the set of goals that can be achieved by p; $\mathcal{R}_{req_p}$ is the plan required resources to execute p; and Body is a set of actions that comprise the plan body.*

Example 6 Using a towel is considered a way to deal with a ceiling leak, and thus is a plan to achieve the goal *ceiling_leak*. Such plan is specified as detailed below, where $a_1, ..., a_n$ are actions to be executed.

$$\begin{aligned} towel\_plan = \langle &\{has\_towel\}, \{!\neg ceiling\_drip\}, \\ &\{cost \mapsto 6, time \mapsto 120\}, \{a_1, ..., a_n\}\rangle \end{aligned}$$

4.2.3 Cause-effect Modelling

The proposed technique deals with scenarios in which problems are resolved. Such problems involve an initial issue (*effect*) to be addressed, which has a *cause*—composed of a set of *cause factors*—which must also be tackled. Our goal is to address both effect and its cause but, to do so, domain knowledge regarding the *cause-effect relationship* must be provided. This section specifies how such knowledge is modelled. We assume that it is given as part of an agent belief base.

Definition 7 (Cause-effect Relationship) *A cause-effect relationship $ce_R(e)$ of an effect e is a tuple $\langle M, O, A_c\rangle$, where M is a set of mandatory cause factors; O is a set of optional cause factors; and A_c is a set of tuples $\langle A, min, max\rangle$, in which A is a set of alternative cause factors, and $min, max \in \mathbb{N}$, such that $min \leq\mid A \mid\leq max$, are the allowed cardinality of cause factors of A.*

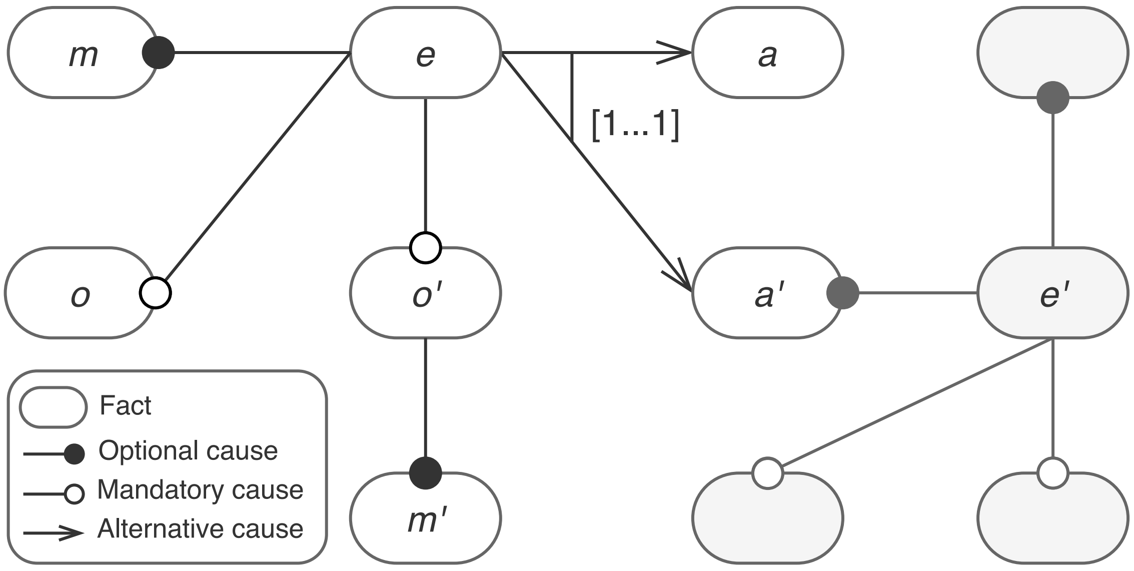
A cause-effect relationship thus associates a problem with possible factors that comprise its cause. There are cases in which a factor is necessarily part of the cause, and in this case it is a mandatory cause factor. Moreover, there may be additional factors that, depending on the situation, may be part of the cause (optional or alternative).

Example 7 The *ceiling_drip* fact has a set of alternative causes, as possible cause factors: *cracked_roof_tile* or *broken_pipe*, with 1 and 2 as minimum and maximum values, respectively, because both cause factors may be part of the cause. The cause-effect relationship is as shown below.

$$ce_R(ceiling\_drip) = \langle \varnothing, \varnothing, \{\langle \{broken\_pipe, cracked\_roof\_tile\}, 1, 2\rangle\}\rangle$$

This cause-effect relationship can be graphically visualised in a directed graph, referred to as *cause-effect knowledge model*, in which nodes are logical predicates that can be an effect or a cause factor. Edges link an effect with a cause factor, and we use a particular notation to distinguish those mandatory, optional or alternative, detailed in Figure 4.1. In this figure, an effect e has several cause factors, m, o, o', a, a'. Mandatory and optional cause factors are represented by edges with filled and unfilled rounded line ends, respectively. Alternative cause factors, in turn, are represented by edges with open arrowheads near its targets, with a line connecting cause factors within the same set. This notation is inspired by feature models (CZARNECKI; EISENECKER, 2000; KANG et al., 1990) that, although have a completely different purpose, also represent this notion of mandatory, optional and alternative concepts.

Figure 4.1: Cause-effect Knowledge Model.



Both cause factors and effects are logical predicates, therefore, an effect f may be associated with a cause factor f', which in turn may be associated with a cause factor f''. This can be seen in Figure 4.1, where o' is simultaneously a cause factor of e and an effect of m'. Cause factors can also be associated with more than one effect. This situation is also depicted in Figure 4.1, where a' is both a cause factor of e and e'. We assume there are no cycles.

Cause-effect relationships (and the cause-effect knowledge model) only indicate possible factors that comprise the cause of a problem. How, based on this knowledge, the actual cause is diagnosed at runtime and how this information is used are described when we detail our customised reasoning cycle.

4.2.4 Extended BDI Agent and Architecture

Before introducing our reasoning cycle, we describe our extended BDI agent, defined as follows. There is one single extension, which is a preference function that gives the importance of a resource for an agent.

Definition 8 (Extended BDI Agent) *An extended BDI agent is a tuple $\langle B, G, P, \mathcal{P} \rangle$, where B is a set of beliefs, G is a set of (possibly constrained) goals, P is a set of plans, and $\mathcal{P}$ is a preference function.*

Definition 9 (Preference Function) $\mathcal{P} : R \nrightarrow [0,1]$ *is a partial function that maps resources r_i to a value indicating the agent preferences over resources. A preference is a value $r_{i_{pref}} \in [0,1]$ that indicates the importance of a resource r, with 0 and 1 being the lowest and highest preference, respectively. Moreover, $\sum_{r_i \in \mathrm{dom}\,\mathcal{P}} \mathcal{P}(r_i) = 1$.*

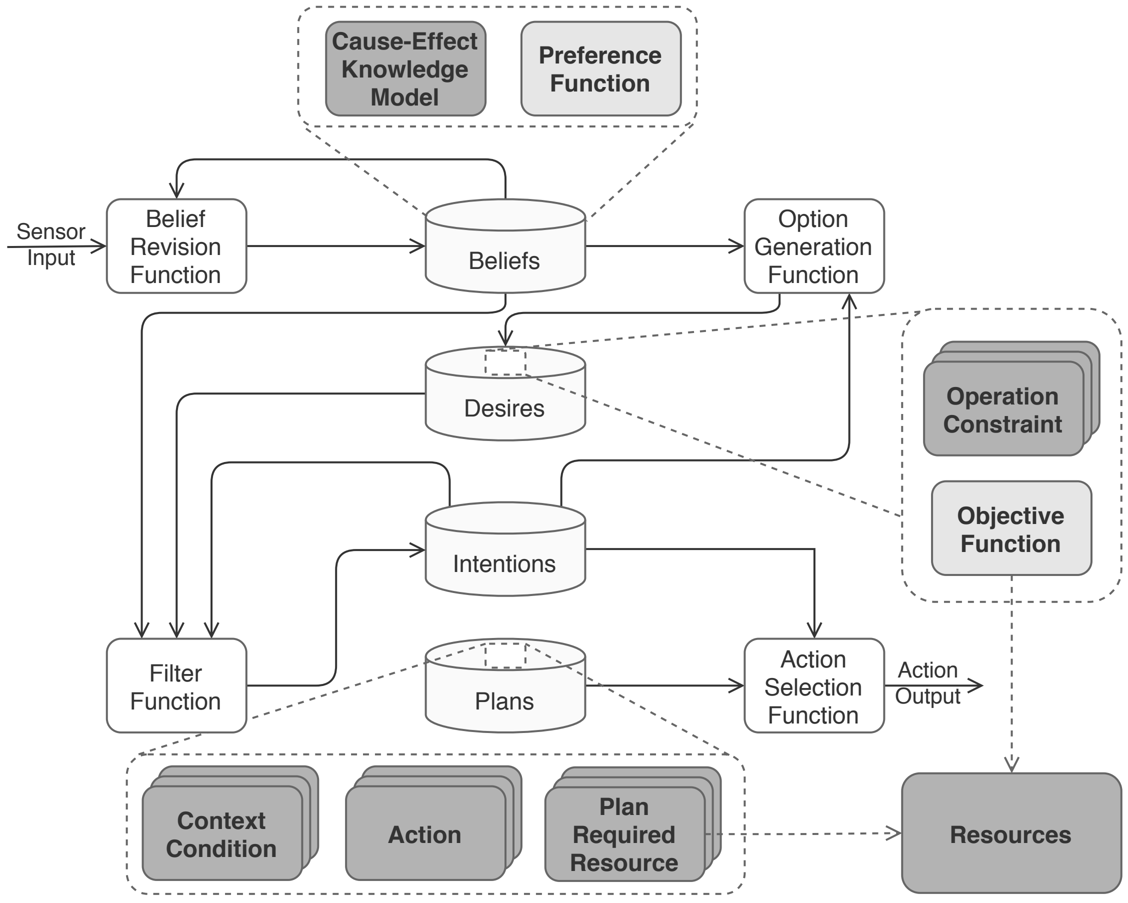
Example 8 In our example, agent A has a belief corresponding to the cause-effect relationship, detailed in Example 7. It has three plans, namely *duct_tape_plan*, *towel_plan*, and *bucket_plan*, each of them being able to achieve the goal *ceiling_leak*. The preference function $\mathcal{P}$ gives A's preferences over the use of resources *cost* and *time*, which are 0.3 and 0.7, respectively, thus $\{cost \mapsto 0.3, time \mapsto 0.7\}$.

This description of extended BDI agents allows us to derive the extended BDI architecture, shown in Figure 4.2. It details the different components that comprise an extended BDI agent. Modules comprising the reasoning cycle are detailed next.

4.3 Customised Reasoning Cycle

In the previous section, we focused on detailing the agent *structure*. We now focus on detailing its *behaviour* at runtime, by customising steps of the BDI reasoning cycle.

Figure 4.2: The Extended BDI Architecture.



This cycle can be generally described as an iterative sequence of steps performed by the belief revision, option generation, filter and plan selection functions.

To provide agents with the required behaviour to deal with problems and their causes, functions of this reasoning cycle must be customised. First, in order to choose among plans, including remedial ones, our plan selection function chooses for execution the plan that satisfies goal constraints and, at the same time, contributes most for the satisfaction of agent preferences over resource consumption. Second, our option generation function is able to identify, evaluate and generate goals for a cause and its factors. Note that the first function is focused on constrained goals, while the second takes into account causal relationships. In this way, we may have constrained goals with no associated cause-effect relationship, or traditional goals with cause-effect relationships. However, their combination is required for addressing our target situations.

4.3.1 Plan Selection

With a set of intentions, which are goals to be achieved, an agent must choose suitable plans to be executed. A single plan is chosen for each intention in an iteration of the reasoning cycle. Our plan selection function performs two key steps. The first selects a set of candidate plans and then, from these, the second step chooses the one that best satisfies preferences over resource consumption.

Definition 10 (Candidate Plan) *A plan p is candidate plan of a goal g, candidate(p, g), if it (i) has all its context conditions satisfied by the current context, i.e. for all pre $\in$ Pre, pre must be an agent true belief; (ii) can achieve g, i.e. g $\in$ G; and (iii) satisfies all its operation constraints C, i.e. for all c $\in$ C, $\mathcal{R}_{req_p}$ satisfies c, if g is a constrained goal.*

Example 9 Table 4.1 details the plan required resources of the three plans of our running example, for each considered resource. No plan has context conditions, they all achieve the goal *ceiling_leak*, and Alice has an operation constraint *cost* $< \$5$. Therefore, the set of candidate plans of this goal is $\{duct\_tape\_plan, bucket\_plan\}$.

Table 4.1: Required Resource of Agent A's Plans.

	duct_tape_plan	*towel_plan*	*bucket_plan*
time	600	120	150
cost	\$0.5	\$6.0	\$0.0

Plan preconditions and operation constraints are both used to discard plans that are not candidates to achieve a given goal. However, it is important to highlight that they are semantically different. The former is used to indicate the requirements needed for a *plan* to be executed, while the latter specifies the conditions in which a *goal* must be achieved.

After the selection of candidate plans, a utility value associated with resource consumption is assigned for the remaining plans. We assume that the amount of resource consumed by a plan has a linear relationship with the promoted utility value, which is in the range $[0, 1]$. Therefore, for each resource over which an agent has preferences, i.e. for each $r \in \operatorname{dom} \mathcal{P}$, we normalise the range of plan required resources, considering plans that can achieve the given goal, to the range of utility values. This normalisation takes into account the goal objective function. If the resource is associated with *min*, the lowest and highest required resource are associated with 1 and 0 utility values, respectively and, the opposite, if it is associated with *max*. If the objective function is undefined for a given resource, the utility value of that resource is 0 for all plans. In order to obtain plan utility, a weighted sum is calculated considering the agent preference function as weights. This is shown in the following equation, which gives the utility values of a plan *p*.

$$\mathcal{U}(p) = \sum_{r \in \operatorname{dom} \mathcal{P}} \mathcal{P}(r) \times \mathbb{T}(\mathcal{R}_{req_p}(r)) \tag{4.1}$$

where $\mathbb{T}$ is the function that normalises the plan required resource. Finally, our plan selector *PSel* chooses for execution the plan that has the highest utility value, as follows.

$$PSel(g_c, P) = \underset{p \in Candidates(P, g_c)}{\arg\max} \; \mathcal{U}(p) \tag{4.2}$$

where g_c is a constrained goal, $Candidates(P, g_c)$ gives the candidate plans of g_c, i.e. $p \in P$, such that $candidate(p, g_c)$.

Example 10 Considering the plans and their required resources to achieve *ceiling_leak*, described in Table 4.1, the range of possible values is $[120, 600]$ for *time*. Therefore, after normalisation and considering the need for minimising time consumption, *duct_tape_plan* has a required resource value equals to 0.0, i.e. the lowest desirable time possible, while *bucket_plan* has a required resource value equals to 0.9. Remember that the time required by *bucket_plan* is 150s, which is very close to the best time possible (120s from *towel_plan*). Given that the objective function is undefined for the resource *cost*, it is not taken into account. With respect to $\mathcal{U}$, the two candidate plans have the following utilities, considering preferences over resources: $\mathcal{U}(duct\_tape\_plan) = 0.3 \times 0.0 + 0.7 \times 0.0 = 0.0$, and $\mathcal{U}(bucket\_plan) = 0.3 \times 0.0 + 0.7 \times 0.9 = 0.6$. The selected plan is thus the *bucket_plan* plan.

Our plan selection function thus selects a plan that best satisfies a constrained goal. This selection process is similar to existing plan selection approaches satisfying other criteria (FACCIN; NUNES, 2015). In our proposal, our plan selection function is used in combination with the goal generation function to provide a remedial behaviour. The selected plan may be a plan that can simply reach the achievement or test goal associated with this goal, or may also address associated causes, if this goal is associated with a cause-effect relationship. The former would be a remedial plan, which can be selected considering preferences and required resources. However, if it is selected, the cause must still be addressed, and this is done with our goal generation function, as detailed next.

4.3.2 Goal Generation

The option generation function is responsible for managing agent goals. In our technique, this function also manages the tracking of goals associated with effects that have a cause. By performing this task, it is able to generate goals associated with cause factors and to conclude (based on them) whether a problem has been fully resolved. To keep track of the effect, cause and cause factors, we use a structure internal to the reasoning cycle, which is defined as follows. For simplicity in the explanation, we assume that effect and cause factors are logical propositions (a fact) instead of logical predicates. However, the approach can be generalised for covering the latter.

Definition 11 (Cause-effect Problem) *A cause-effect problem $ce_P(e)$ of an effect e is a tuple $\langle g_e, CF_S, ce_{ES}\rangle$, where g_e is an achievement goal in the form $!\neg e$; CF_S is a set of cause-factor status $\langle fact, s_i, s_u, ?g, !\neg g\rangle$, where fact is a possible cause factor of e according to the cause-effect relationship $ce_R(e)$, $s_i \in \{true, false, nil\}$ is the initial state, $s_u \in \{true, false\}$ is an updated state, $?g$ is a generated test goal, and $!\neg g$ is a generated achievement goal; and ce_{ES} is the problem end state.*

The key idea is that, when an achievement goal that focuses on dealing with the effect of a problem is added, a cause-effect problem is created to keep track of it. Based on the cause-effect relationship, all possible cause factors have their status monitored. s_i stores whether a cause factor holds, at the first time that this information is known. For example, when adding the *ceiling_leak* goal, Alice may already know that there is a broken pipe. In this case, s_i becomes *true*, without searching whether *broken_pipe* holds. s_u stores whether the state of the cause factor changed since its initial state is known. This is important because, if a non-remedial plan that also had as postcondition $\neg fact$ was performed to achieve g_e, it is possible to infer that the cause factor *fact* was already dealt with. If this was not the case, goals to investigate ($?fact$) and address ($!\neg fact$) the cause factor must be generated, and these are stored in $?g$, $!\neg g$, respectively. Whenever the state of *fact* changes, s_u is updated. Therefore, if other effects with a shared cause factor already addressed it, no goal is generated to do it again.

The idea described above is shown in our option generation function presented in Algorithm 1. Before detailing our algorithm, we introduce the definition of sets of elements that are used throughout this section. They are defined in Table 4.2, which also specifies the possible states available for goals and cause-effect problems. Moreover, the

Table 4.2: Status Set Descriptions.

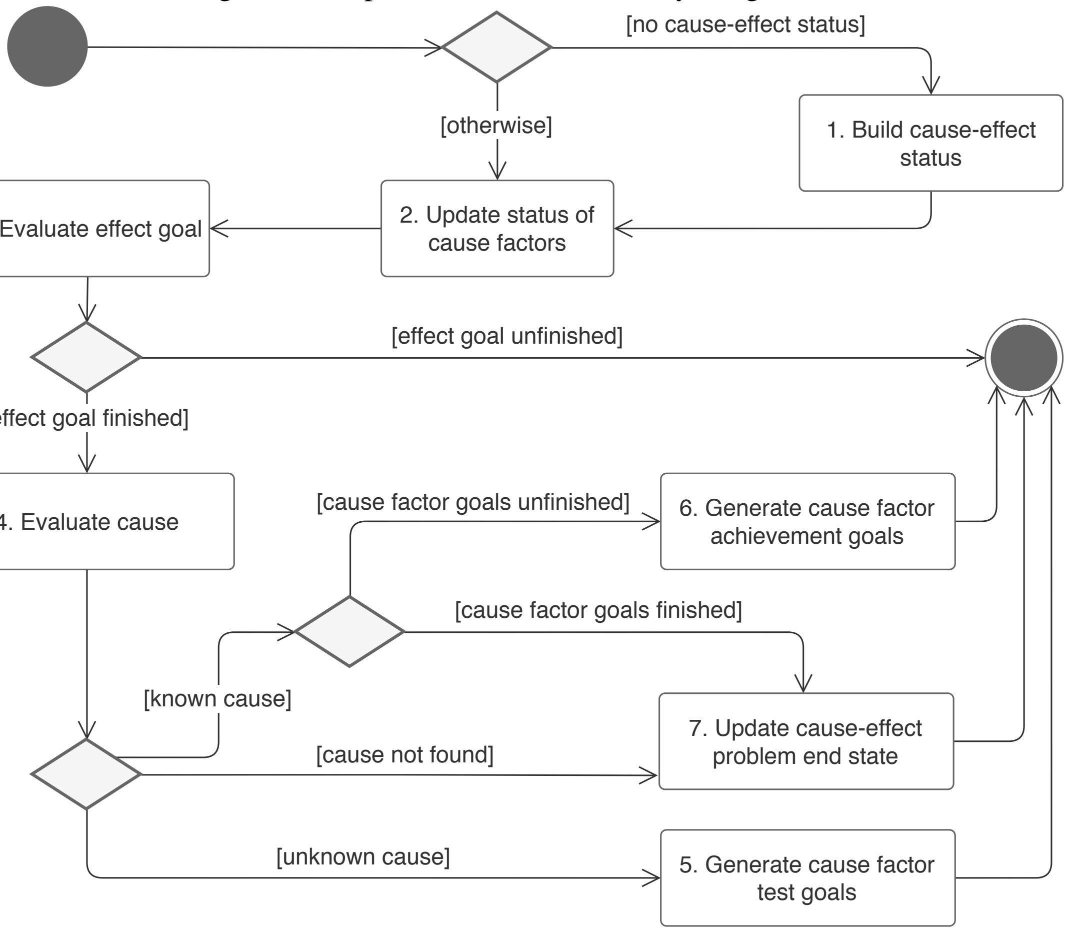
Set	Set Description	Set Elements	Element Description
G_S	The set of all possible statuses of a goal	*achieved*	The goal has been achieved
		unachievable	The goal cannot be achieved
		noLongerDesired	The goal is no longer desired
		nil	The agent is trying to achieve the goal
CE_{ES}	The set of all possible end states of a cause-effect problem	*unsuccessful*	Neither the effect or its cause are solved
		causePartiallyResolved	The effect is not solved and its cause is partially addressed
		causeResolved	The effect is not solved and its cause is completely addressed
		mitigated	The effect is solved and its cause is not addressed
		partiallyResolved	The effect is solved and its cause is partially addressed
		fullyResolved	The effect is solved and its cause is completely addressed

algorithm uses a set of auxiliary functions that are described in Table 4.3. In order to facilitate its understanding, we present in Figure 4.3 an overview of this algorithm in a UML activity diagram, focusing solely on what happens with achievement goals that focus on dealing with the effect of a problem. In each execution of the BDI reasoning cycle, this algorithm is executed. First, when a goal that is an effect according to a cause-effect relationship is added, a cause-effect problem is created associated with it (Activity 1, lines 2–5 of Algorithm 1). Then, for this new added cause-effect problem and all existing others, the cause factor status are updated (Activity 2, lines 7–16 of Algorithm 1). Next, the status of the goal associated with the effect is evaluated (Activity 3, line 17 of Algorithm 1). If this goal did not reach a finished state—see the $\mathit{finished}(g_S)$ function, in Table 4.3—no goals are generated. Otherwise, the cause is evaluated (Activity 4, lines 18 and 28 of Algorithm 1). If the cause is unknown, test goals associated with cause factors for those with an unknown initial state are generated, if needed (Activity 5, lines 32–34 of Algorithm 1). If the cause is known, there are two possibilities. The first is that achievement goals associated with cause factors have not been generated or have not finished yet and, in this case, they are created if needed (Activity 6, lines 23–26 of Algorithm 1). The second is that they all reached a finished state. In this case, the cause-effect problem reached an end state, and this state is updated according to the possibilities described in Table 4.4 (Activity 7, lines 19–21 of Algorithm 1). Finally, if after searching for the cause, it was not possible to diagnose it, the end state is updated accordingly (Activity 7, lines 28–29 of Algorithm 1).

Algorithm 1: *GenerateGoals*(G, B)

```
   Input: $G$: set of agent goals, $B$: set of agent beliefs
   Output: $G$: updated set of agent goals
1  foreach $!\neg g \in G$ do
2      if $\exists\, ce_R(g) \wedge \nexists ce_P(g)$ then
3          $ce_P(g) \leftarrow \langle !\neg g, \emptyset, nil \rangle$;
4          foreach $c_i \in$ causeFactors$(ce_R(g))$ do
5              $ce_P(g)[CF_S] \leftarrow ce_P(g)[CF_S] \cup \langle c_i, nil, nil, nil, nil \rangle$
6      if $\exists\, ce_P(g)$ then
7          foreach $cf \in ce_P(g)[CF_S]$ do
8              if $cf[s_i] = nil$ then
9                  if $cf[fact] \in B$ then
10                     $cf[s_i] = true$;
11                 else if $\neg cf[fact] \in B$ then
12                     $cf[s_i] = false$;
13                 $cf[s_u] = false$;
14             else
15                 if $(cf[s_i] = true \wedge \neg cf[fact] \in B) \vee (cf[s_i] = false \wedge cf[fact] \in B)$
                     then
16                     $cf[s_u] = true$;
17         if finished(goalStatus$(!\neg g)$) then
18             if knownCause$(ce_P(g), ce_R(g))$ then
19                 if causeFinished$(ce_P(g))$ then
20                     $ce_P(g)[ce_{ES}] \leftarrow$ endState$(ce_P(g))$;
21                     $ce_P(g) \leftarrow nil$;
22                 else
23                     foreach $cf \in ce_P(g)[CF_S]$ do
24                         if $cf[s_u] = false \wedge cf[!\neg g] = nil$ then
25                             $cf[!\neg g] \leftarrow !\neg cf[fact]$;
26                             $G \leftarrow G \cup \{cf[!\neg g]\}$;
27             else if causeNotFound$(ce_P(g))$ then
28                 $ce_P(g)[ce_{ES}] \leftarrow$ endState$(ce_P(g))$;
29                 $ce_P(g) \leftarrow nil$;
30             else
31                 foreach $cf \in ce_P(g)[CF_S]$ do
32                     if $cf[s_i] = nil \wedge cf[?g] = nil$ then
33                         $cf[?g] \leftarrow ?cf[fact]$;
34                         $G \leftarrow G \cup \{cf[?g]\}$;
35 return $G$;
```

Figure 4.3: Option Generation Activity Diagram.


[no cause-effect status]
1. Build cause-effect status
[otherwise]
2. Update status of cause factors
3. Evaluate effect goal
[effect goal unfinished]
[effect goal finished]
4. Evaluate cause
[cause factor goals unfinished]
6. Generate cause factor achievement goals
[cause factor goals finished]
[known cause]
7. Update cause-effect problem end state
[cause not found]
[unknown cause]
5. Generate cause factor test goals

Example 11 We assume, in our running example, that the constrained goal *ceiling_leak* is achieved by the *bucket_plan* plan, selected with our plan selection function. Such constrained goal has as goal $!\neg$*ceiling_drip*, which has an associated cause-effect relationship $ce_R(ceiling\_drip)$. Due to this, a cause-effect problem $ce_P(ceiling\_drip)$ is created. After *ceiling_leak* is achieved, two test goals are generated: $?$*broken_pipe* and $?$*cracked_roof_tile*. Assuming that both goals were achieved, and only *broken_pipe* is true, an achievement goal $!\neg$*broken_pipe* is generated. If this goal is achieved, the end state of our cause-effect problem is *Fully Resolved*.

4.4 Evaluation

Given that we presented both our extended architecture and customised reasoning cycle, we now focus on the evaluation of the proposed technique. We first briefly describe how we implemented it as an extension of an existing BDI agent platform, namely BDI4JADE (NUNES; LUCENA; LUCK, 2011). Then, we describe the procedure of our

Table 4.3: Description of Auxiliary Functions.

Predicate	Logic Expression	Description
`finished(`g_S`)`	$finished(g_S) = \begin{cases} true, & \text{if } g_S \in G_S \setminus \{nil\} \\ false, & \text{otherwise} \end{cases}$	It is true when a goal *g* has reached a finished status, that is, it is *true*, if $g_S \neq nil$, or *false*, otherwise.
`knownCause(`$ce_P(g), ce_R(g)$`)`	$(\forall\, cf \in ce_P(g)[CF_S], cf[s_i] \neq nil) \wedge (\forall f \in ce_R(g)[M]\, \exists\, cf \in ce_P(g)[CF_S] \mid cf[fact] = f, cf[s_i] = true) \wedge (\forall\, a_c \in ce_R(g)[A_c], a_c[min] \leq \lvert \{cf : cf \in ce_P(g)[CF_S] \wedge cf[fact] \in a_c[A] \wedge cf[s_i] = true\} \rvert \leq a_c[max])$	It is true when the cause of an effect was identified. It means that the initial state of all cause factors are known, mandatory cause factors are *true*, and those alternative satisfy the cardinality of an alternative set.
`causeNotFound(`$ce_P(g)$`)`	$\exists\, cf \in ce_P(g)[CF_S], cf[s_i] = nil \wedge finished(goalStatus(cf[?g]))$	It is true when it was not possible to identify the cause of an effect. It means that there are cause factors with an unknown initial state, even after associated test goals reached a finished status.
`causeFinished(`$ce_P(g)$`)`	$\forall\, cf \in ce_P(g)[CF_S] : cf[s_u] = true$	It is true when the update state of all cause factors that are actually part of the cause is *true*.

(a)

Function	Mathematical Expression	Description
`causeFactors(`$ce_R(g)$`)`	$\{f : f \in ce_R(g)[M] \vee f \in ce_R(g)[O] \vee f \in a_c[A], a_c \in ce_R(g)[A_c]\}$	Gives the set of all possible cause factors, including those that are mandatory, optional and alternatives within the alternative sets.
`goalStatus(`g`)`	$goalStatus(g) : G \rightarrow G_S$	Gives the current status of a goal g.
`endState(`$ce_P(g)$`)`	$endState(ce_P(g)) = CE_P \rightarrow CE_{ES}$	Gives the end state of a cause-effect problem. Possible states are detailed in Table 4.4.

(b)

evaluation, followed by obtained results.

4.4.1 BDI4JADE Implementation

We implemented the proposed approach using BDI4JADE, a Java-based platform for the development of BDI agents (NUNES; LUCENA; LUCK, 2011). This platform was selected because it provides means of an easy extension of the functions comprising the BDI reasoning cycle.

With respect to constrained goals, we added a new type of goal to the platform, with the class `ConstrainedGoal` implementing the `Goal` interface of BDI4JADE.

Table 4.4: Cause-Effect Problem End States.

End State	Logic Expression	Description	
		Effect Goal	Cause / Cause Factors
Unsuccessful	$(goalStatus(g) \in G_S \setminus \{achieved, nil\}) \wedge ((causeNotFound(ce_P(g)) \vee (\forall\, cf \in ce_P(g)[CF_S], cf[S_u] = false))$		Cause not found or **none** of the cause factors were achieved.
Cause Partially Resolved	$(goalStatus(g) \in G_S \setminus \{achieved, nil\}) \wedge (\neg causeNotFound(ce_P(g)) \wedge (\exists\, cf \in ce_P(g)[CF_S], cf[S_u] = false) \wedge (\exists\, cf \in ce_P(g)[CF_S], cf[S_u] = true)$	Unachievable or No longer desired	Cause identified and **some** of the cause factors were achieved.
Cause Resolved	$(goalStatus(g) \in G_S \setminus \{achieved, nil\}) \wedge (\neg causeNotFound(ce_P(g)) \wedge (\forall\, cf \in ce_P(g)[CF_S], cf[S_u] = true)$		Cause identified and **all** of the cause factors were achieved.
Mitigated	$(goalStatus(g) = achieved) \wedge ((causeNotFound(ce_P(g)) \vee (\forall\, cf \in ce_P(g)[CF_S], cf[S_u] = false))$		Cause not found or **none** of the cause factors were achieved.
Partially Resolved	$(goalStatus(g) = achieved) \wedge (\neg causeNotFound(ce_P(g)) \wedge (\exists\, cf \in ce_P(g)[CF_S], cf[S_u] = false) \wedge (\exists\, cf \in ce_P(g)[CF_S], cf[S_u] = true)$	Resolved	Cause identified and **some** of the cause factors were achieved.
Fully Resolved	$(goalStatus(g) = achieved) \wedge (\neg causeNotFound(ce_P(g)) \wedge (\forall\, cf \in ce_P(g)[CF_S], cf[S_u] = true)$		Cause identified and **all** of the cause factors were achieved.

In addition, the classes `ResourcePreferences` and `PlanRequiredResources` were implemented to represent preferences over resources and required resources of plans, respectively. The former is added as an agent belief, while the latter as plan metadata (available in the platform). These and other associated classes are used in a customised plan selection strategy (an extension point of the platform). We also created the classes needed to model the cause-effect knowledge model, which is also added as an agent belief. Such classes are used in an implemented option generation function, another platform extension point.

All these described components are added as part of a capability. BDI4JADE uses this concept to modularise agents. To implement a BDI agent that adopts the proposed technique, this capability must be instantiated and the knowledge regarding preferences over resources, plan required resources and cause-effect relationships must be provided.

4.4.2 Scenario and Procedure

The proposed technique aims at allowing agents to independently choose appropriate plans to remediate problems and solve their causes. In order to evaluate it, we selected an existing solution for combating distributed denial-of-service (DDoS) attacks. This solution is based on the D^2R^2 + DR strategy (STERBENZ et al., 2010) and manually implements the remedial behaviour we aim to automate.

This solution begins by initially observing an abnormal network traffic in a network link through a link monitor. This is a problem that should be addressed. Before investigating the cause of the abnormal traffic, it is essential to mitigate its effects, otherwise servers may become unavailable. Therefore, the link is limited to reduce the amount of traffic. The first step towards the cause diagnosis is to identify the victim of the attack. For this, an anomaly detection component makes a statistical analysis of the traffic. Now, that part of the cause was identified, the traffic destined to the attack victims is reduced. The second step towards the cause identification is to identify the malicious flows of the attack, i.e. not only the victim should be known, but also the source of the attack. When this is done, malicious flows can have their traffic limited, and this completes the resolution of the problem.

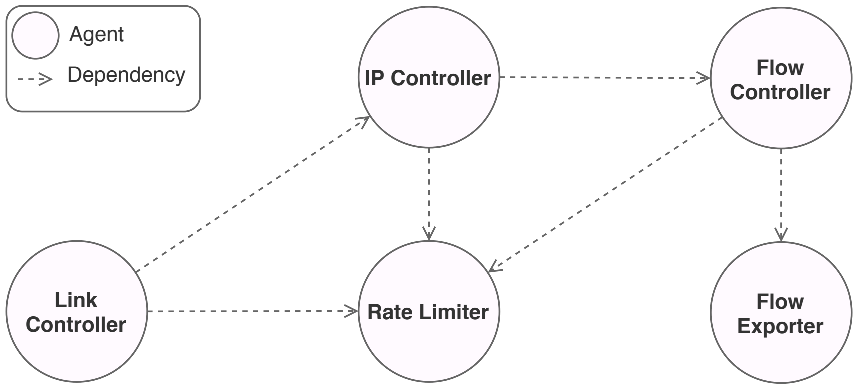
The described solution has a police-based (SCHAEFFER-FILHO et al., 2012) and a multiagent (NUNES; SCHARDONG; SCHAEFFER-FILHO, 2017) implementation. To evaluate our proposal, we implemented that solution using our described BDI4JADE extension. This allows us to: (i) demonstrate that agents can autonomously and flexibly deal with challenging scenarios by remediating and handling the cause of problems; (ii) assess the impact of the proposed domain-independent technique in the performance; and (iii) measure the reduction of development effort.

4.4.3 Results and Analysis

Our implemented solution to combat DDoS attacks used the previous multiagent implementation as a baseline (NUNES; SCHARDONG; SCHAEFFER-FILHO, 2017). This was performed in order to have similar plans, so that variance in our measurements would not be due to plan body implementations or other details not related to the automated coordination of agent actions. Such a solution comprises five agents interacting with each other, each of them with distinct capabilities. They are represented in Figure 4.4. We next describe how it was implemented.

Initially, the causal relationships between elements of the scenario were identified and instantiated in a cause-effect knowledge model, which is represented in Figure 4.5 together with plans to reach test and achievement goals. This model was later added to

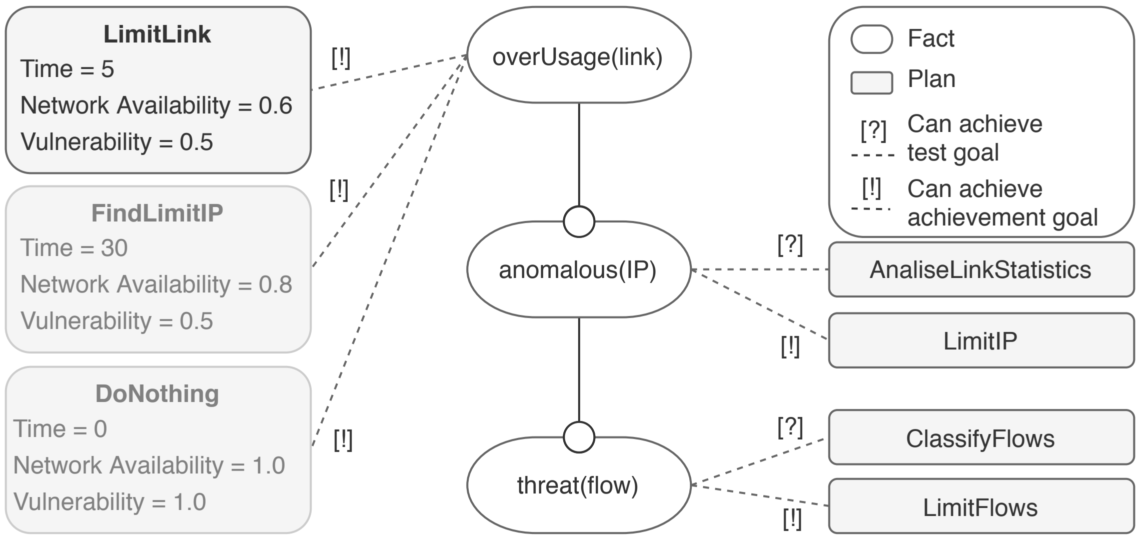
Figure 4.4: Multiagent scenario to combat DDoS attacks.



the `Link Controller` agent. The different stages of the solution described above are modelled as two cause-effect relationships. A detected abnormal traffic in a link ($overUsage(link)$) can be due to an anomalous usage, meaning that exists at least one server receiving abnormal incoming traffic ($anomalous(IP)$). This in turn can be due to a malicious attack, which means that there exists at least one flow that is a threat ($threat(flow)$). This model together with the plans work in the following way. First, when the traffic in a link is above a threshold, a $\neg overUsage(link)$ goal is added. Then, a cause-effect problem is created and the `LimitLink` plan is executed to achieve the goal. Next, because of the cause-effect problem, a test goal is generated to verify the cause, and the `AnaliseLinkStatistics` plan executes, resulting in a set of, if any, servers receiving anomalous traffic. This leads to the creation of achievement goals to limit the server traffic (`LimitIP`)—as stated before, here we made a simplification that causes are facts, but the approach can be generalised to many cause factor instances, as the approach is implemented. Finally, for each $anomalous(ip)$, its cause is diagnosed by the `ClassifyFlow` plan, and flows that are threats are resolved by the `LimitFlows` plan.

This implemented behaviour follows the solution as proposed, in which a remedial plan is always performed. We now show a benefit of our approach that provides agents with flexible behaviour by simply informing preferences over resources. Assume that, in order to achieve the $\neg overUsage(link)$ goal, there are two additional plans: (i) one that takes the time to analyse the traffic, find target servers, and only limits the traffic to these servers (`FindLimitIP`); and (ii) another to do nothing, assuming that the abnormal traffic was temporary (`DoNothing`). Then, we specify resources required by the plans. The first is time taken to execute (*time*), the second is associated with how much network bandwidth will become unavailable (*Network Availability*), and the last is asso-

Figure 4.5: DDoS Cause-Effect Model.



ciated with how much the network will become vulnerable (*Vulnerability*). Depending on preferences, and possibly constraints, specified for the $\neg overUsage(link)$ goal, which is then a *constrained goal*, different remedial plans can be selected. Moreover, note that the approach is robust enough for not searching for known causes. If the `FindLimitIP` plan is selected and executed, as result of its execution, which includes the analysis of the link statistics, $anomalous(IP)$ would already be known. Therefore, in this case, the cause becomes known while remediating the problem, being immediately addressed after that.

Finally, we compare our approach with our baseline from two perspectives: *performance* and *development effort* (estimated by lines of code). The baseline implementation is a project that has 3,260 lines of code (LOC). We limited ourselves to reengineer only the behaviour associated with the reasoning automated by our approach. The portion of code that implemented this reasoning (with goal generation, plans, etc.) has 733 LOC. Our application implementation, using our BDI4JADE extension, consists of 553 LOC. Consequently, our approach was able to reduce **24.5%** of the development effort in terms of LOC. Note that such behaviour that was automated is typically complex, thus hard to implement and maintain. We also assessed if, by providing a domain-independent solution, agent performance would decay. For this purpose, we used synthetic data to simulate a DDoS attack scenario to be dealt with and compared the executing time of both implementations. As shown in Table 4.5, which summarises obtained results, the executing time of the simulation of both implementations is similar—our approach took only **19 ms** more to execute. It is important to highlight that the use of real data would not impact the outcome of this experiment, given that it is directly associated with the behaviour implemented within plan bodies, which is the same in both system versions.

Moreover, because of the deterministic behaviour of both implementations—there is only one possible execution path—the presented results correspond to a single execution of each of them. We in fact run the simulation multiple times, always obtaining the same results.

Table 4.5: Evaluation Results.

	Executing Time (s)	**Development Effort (LOC)**
Baseline	82.086	733
Our approach	82.105	553

In summary, our approach is able to reduce the amount of code to be developed (in terms of LOC) by approximately 25%, by providing a reusable solution that automates the coordination of actions in order to remediate problems and handle their causes. Moreover, this domain-independent solution does not reduce system performance, despite its generality. Although LOC is not a synonym of development effort, because code varies in complexity, we emphasise that the code that is now automated by our approach is not a trivial part of the implementation, due to the many conditions that should be considered in the design and implementation of a remedial behaviour. Therefore, by automating the reasoning about remedial actions, we potentially reduce the development effort, thus reducing time-to-market and development costs. We do not expect a reduction of 25% of the effort in any software project, given that the proportion between the reasoning about remediation actions and other concerns to be implemented varies. However, some reduction in effort is always expected. By suppressing the need for this reasoning to be manually implemented in agents and multiagent systems, we also potentially improve maintainability because this concern is typically implemented interleaved with the code that implements the standard behaviour of plans. Therefore, our approach provides a better separation of concerns.

4.5 Final Remarks

This chapter presented a technique that allows agents to handle challenging situations by autonomously coordinating the remediation of problems and the handling of their causes. This technique is composed of structural and behavioural parts. The former consists of an architecture, which extends traditional BDI agents by introducing the notions of constrained goals, plan required resources and cause-effect relationships. The latter provides the specification of two abstract functions of the BDI reasoning cycle, which are the plan selection and goal generation functions. In order to evaluate the proposed technique, we considered an existing solution for combating DDoS attacks that remediates their effects before addressing their causes. By using our approach, we demonstrated that our extended BDI agent can autonomously select appropriate plans to deal with the attack and generate goals to diagnose and resolve its cause. Moreover, we compared the performance of our technique with the existing implementation of this solution, and results indicate that our approach reduces development effort and does not reduce performance due to the automated reasoning made at runtime. The proposed technique thus provides a ready-to-use solution in which agents are provided with a sophisticated and flexible remedial behaviour, promoting software reuse in software agents.

Despite the automated coordination of actions provided by our approach, the cause-effect knowledge model that guides the root cause diagnosis must still be manually provided. This task is not trivial and, therefore, would benefit from further automation. The next chapter introduces a technique that provides such causal information at runtime by taking advantage of particular aspects of multiagent systems.

5 ROOT CAUSE DIAGNOSIS

The technique for coordinating the remedial behaviour, which is presented in Chapter 4, allows agents to remediate problems and diagnose their causes in an autonomous way. However, that technique assumes that the causal knowledge required to guide the diagnose operation is provided in advance by experts. In this chapter, we introduce a solution that allows agents to build this knowledge at runtime. This solution takes advantage of specific characteristics of multiagent systems (MAS) and, when integrated to our technique for managing agent actions, reduces the need for human involvement while providing MAS with the ability to adapt and keep operating with the expected quality. Our decentralized solution consists of two main elements: (i) an interaction protocol that specifies agent roles and how they interact; and (ii) a set of algorithms that define how these agents behave in order to diagnose the problem causes to be repaired.

5.1 Problem and Definitions

As introduced in Chapter 1, a MAS is a collection of autonomous components (agents) that are situated in an environment and are able to interact and collaborate by consuming and providing sets of services (JENNINGS, 2001). We assume that, to execute their tasks with the expected quality, the agents require the services they consume to satisfy a set of predefined quality requirements, which are expressed in terms of measurable features and their range of acceptable values. An example of a quality requirement expressed in natural language is: *"no service may have response time greater than 10 ms."* When a quality requirement is not satisfied, there is a violation that is considered an *abnormality* or, alternatively, a problem. Concepts associated with services and their quality are defined as follows.

Definition 12 (Service) *A service $s \in \mathcal{S}$ is an action that can be performed by an agent.*

Definition 13 (Quality Feature) *A quality feature $q \in \mathcal{Q}$ is a measurable property associated with a service.*

Definition 14 (Quality Requirement) *Quality requirement $\mathcal{R} : \mathcal{Q} \nrightarrow \mathcal{K}$ is a function that maps a quality feature $q \in \mathcal{Q}$ to a constraint $\mathcal{K}$, which denotes the requirements to which a s must satisfy with respect to each q for which $\mathcal{R}$ is defined. $\mathcal{K}$ is expressed with* *the following grammar.*

$$
\begin{aligned}
\mathcal{K} &::= (\mathcal{K} \wedge \mathcal{K}) \mid (\mathcal{K} \vee \mathcal{K}) \mid (\neg \mathcal{K}) \mid e \\
e &::= (q\, op\, v) \\
op &::= > \mid \geq \mid < \mid \leq \mid = \mid \neq
\end{aligned}
$$

where q is the mapped quality feature and v is a value $v \in \mathbb{R}$.

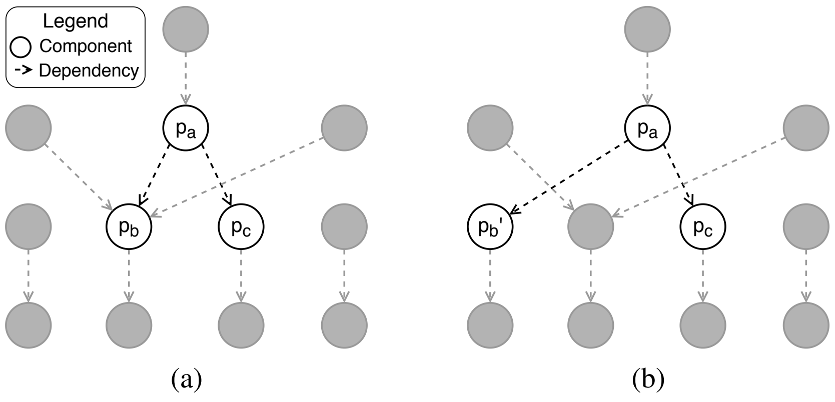
Agents interact through the exchange of messages. Our proposal considers two types of messages. *Request* messages are sent by agents in order to ask for services or particular information. *Inform* messages, in turn, are sent in order to either reply to requests or provide information that was not previously required. Messages with no recipient specified are broadcast to every component within a system.

Definition 15 (Message) *A message is a tuple $\langle id_m, id_c, c_s, c_r, type, s, cont \rangle$, where id_m is the identifier of the message; id_c is the identifier of the conversation of which the message is part; c_s is the message sender; c_r is the message receiver; $type \in \{inform, request\}$ is the type of the message; s is the service that may be associated with the message, if any; and cont is the message content, which can be of any type.*

Agents use services provided by other agents to make their own services available. As a result, when there is an abnormality, there are three possible points of failure. An abnormality cause can be located at the agent itself (i.e. an *internal cause*), at its service providers, or at their communication channel (both being *external causes*). As an example, consider the system depicted in Figure 5.1a in which agents (represented as circles) interact by requesting and delivering services (dependencies between agents are represented as traced arrows from clients to their providers). To provide service a, component p_a depends on services b and c, consumed from p_b and p_c, respectively. In order to deliver their own services, p_b and p_c, in turn, rely on services from other agents. In this context, an abnormality presented by p_a may not be necessarily caused by that component. Instead, its dependency on services from p_b and p_c indicates that these providers, or even those from which they consume services, could also be associated with the problem.

To identify the cause of an abnormality and be able to remediate and solve it, the system is required not only to recognise the existence of these component interactions, but also to evaluate which of them originated the abnormal behaviour. In a simple system, such as the one described above, it is possible to maintain a cause-effect knowledge

Figure 5.1: A MAS in which components interact with each other by consuming and providing services. (a) Agent p_a relies on services b and c from agents p_b and p_c, which, in turn, consume services from other agents. (b) Agent p_a replaces service provider p_b with $p_b{}'$.



model (see Chapter 4) that contains this causal information. For instance, if p_a replaces service provider p_b with $p_b{}'$ (Figure 5.1b), this new provider becomes a potential cause of further abnormalities presented by p_a. Consequently, this new dependency must be included in the existing causal model while the previous one must be removed from it as it is no longer a possible cause of abnormality. When we consider dynamic, large scale systems, however, maintaining such cause-effect knowledge model in a centralised way is impracticable, either because of the amount of dependencies to be managed, or their dynamic evolution. Our goal is thus to autonomously diagnose the root cause of abnormalities and use this information to remediate and solve them in order to comply with predefined quality requirements.

5.2 Cooperative Diagnosis and Solution of Problem Causes

To achieve this goal, we propose a technique that, when integrated with the automated management of agent actions introduced in Chapter 4, allows agents to collaborate in order to diagnose and further remediate and solve the root cause of detected abnormalities. Next, we present an overview of our solution.

5.2.1 Overview

Our proposed technique includes an interaction protocol and the specification of algorithms for agents to be able to play the protocol roles to diagnose the cause of abnormalities. We overview our interaction protocol with an example presented in Figure 5.2, which illustrates a scenario where there is an abnormality caused by a service provider (an external cause). In this scenario, an agent c consumes a service a, which must comply with a given quality requirement $\mathcal{R}(q)$. This service is provided by an agent p_a, which consumes services b and c from agents p_b and p_c, to deliver it. If there is a degradation of the quality level of the service provided by p_b, it is likely that this is propagated to the services provided by other agents that rely on p_b's services. Therefore, agents p_a, n and n' are affected. In particular, as a consequence, p_a also delivers degraded services (Figure 5.2a).

Assume that c detects that the service it consumes from p_a violates $\mathcal{R}(q)$. This causes c to send a message to p_a to notify the occurrence of such an abnormality (Figure 5.2b). Receiving this notification raises on p_a the need for normalising its operation. The core of our idea is that, to fulfill this need, the notified component carries out a step-wise cause identification. It first verifies whether the abnormality has an internal cause. If there is no evidence of internal issues, a second verification step is performed to identify which external source is the cause of the problem. By identifying the source of the abnormality, p_a is able to remediate and, if possible, definitely solve it, thus preventing further quality requirement violations. In our example, p_a has no evidence that the abnormality comes from internal sources and, therefore, self-healing mechanisms are unable to repair it. Instead, the issue is identified as coming from the consumption of service b. In this case, p_a remedies it by replacing the existing provider p_b with $p_b{}'$ and informing c that its operation was normalised (Figure 5.2c), while proceeding to the second step of the verification activity.

Next, p_a takes into account the perception of cooperating agents to determine the external cause of abnormality. To carry it out, p_a broadcasts a message requesting agents that also consumed service b from p_b to inform, based on their history, the likelyhood of receiving an anomalous reply from that provider. In our example, this request is replied by n and n' (Figure 5.2d). After evaluating the obtained replies, p_a can identify the problem cause and handle it accordingly. If it does not detect any abnormality coming from the suspicious agent, the communication link between them is treated as the cause, and it can act to repair it. However, if there is evidence that p_b caused the abnormality, p_a sends a message notifying that agent of the issue (Figure 5.2e). Receiving this notification triggers on p_b the same need for normalising its operation. Once it occurs, p_a is informed (Figure 5.2f) and becomes able to start using service b from its previous provider again, thus restoring the system to its initial state (Figure 5.2g). We detail the protocol that specifies how these agent interactions occur in the next section.

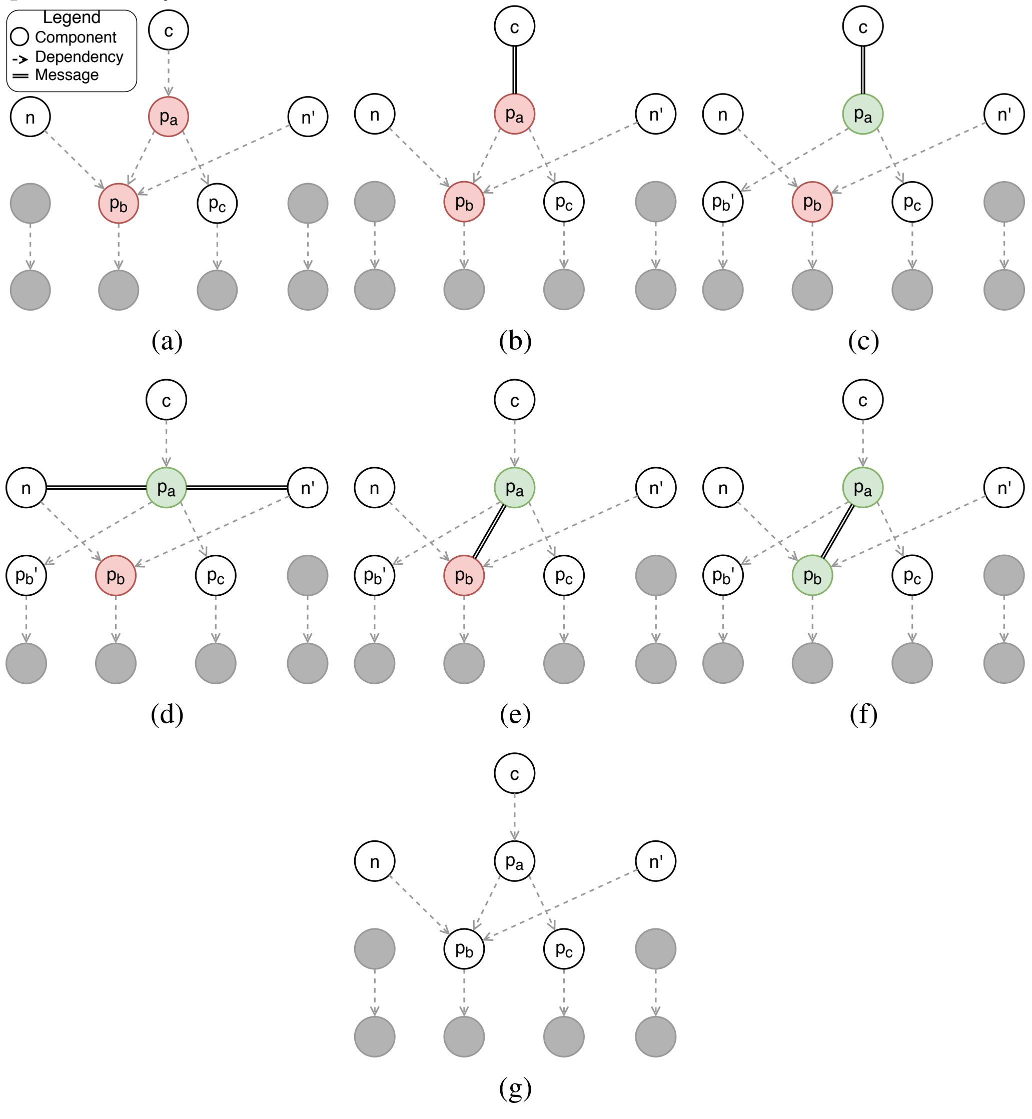


Figure 5.2: An overview of system behaviour implementing our proposed solution. (a) Agent p_b presents an abnormal behaviour that affects agents that depend on it. (b) Agent c perceives a violation on a quality requirement and notifies agent p_a. (c) Agent p_a replaces provider p_b with $p_b{}'$ and informs c that its operation is normalised. (d) p_a broadcasts a request of information, which is replied by n and n'. (e) p_a notifies p_b of its perceived abnormal behaviour. (f) p_b informs p_a that its operation is normalised. (g) p_a replaces $p_b{}'$ with p_b and the system returns to its former state.

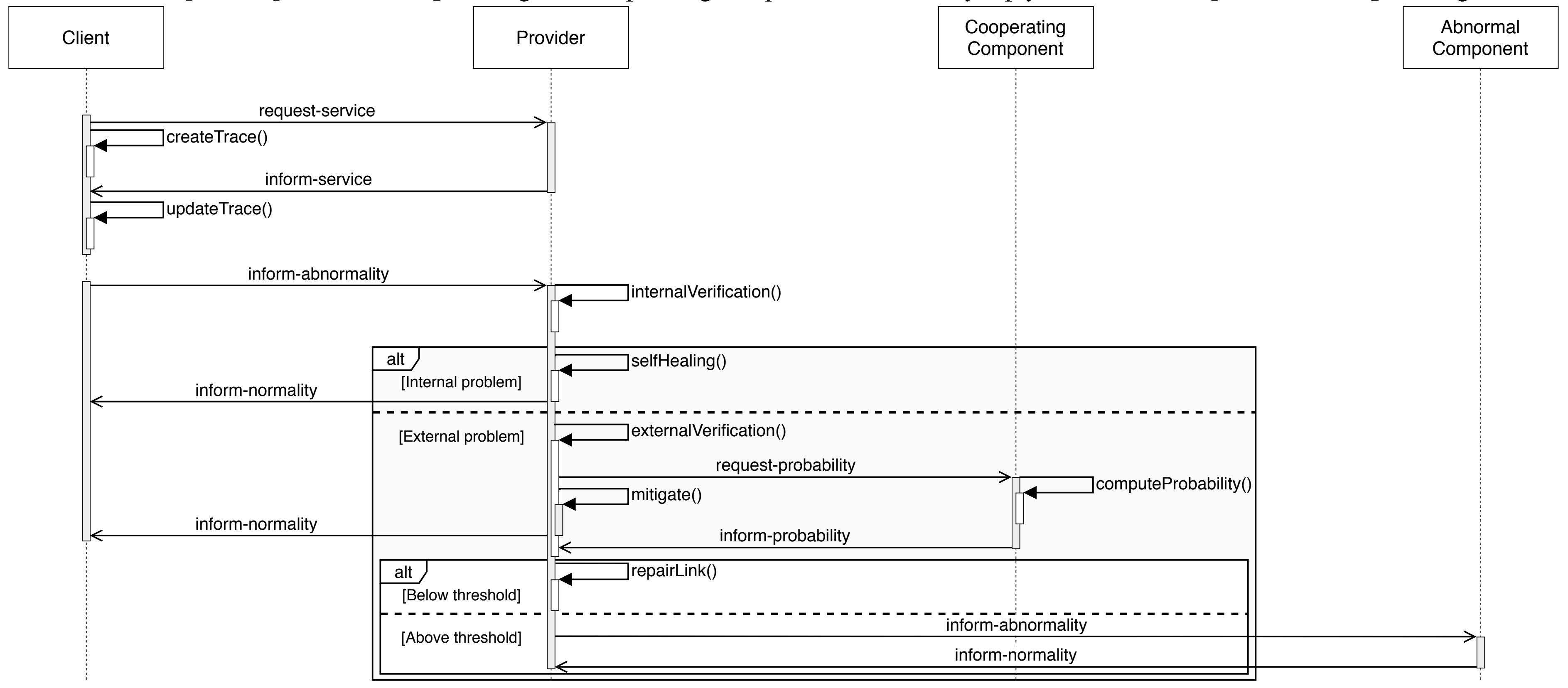
5.2.2 Interaction Protocol

Our protocol allows agents to exchange messages in order to collaboratively diagnose the cause of a service abnormality, assuming that agents use services provided by other agents. Figure 5.3 depicts the interaction among agents. The request of a service is done by client agents, when they send a `request-service` message, which specifies the service being requested. This request can be sent to many provider agents and a negotiation may take place to choose a provider. However, this can be done with existing protocols, such as the FIPA Contract Net Protocol (AGENTS, 2002). Our protocol thus focuses on the agent that actually provided the service. When the service has been completed, the provider replies the client request with an `inform-service` message containing the output of the corresponding service. During this exchange of messages, clients collect information that may be used later to identify abnormalities within the system (`createTrace()` and `updateTrace()`).

When the client detects a requirement violation on a consumed service, an `inform-abnormality` message is sent to the corresponding provider, notifying it of the quality feature and conversation in which the abnormal behaviour occurred. The provider replies this notification with an `inform-normality` message after it has taken corrective measures to address the abnormality. When this reply is sent depends on the result of the activity performed to verify whether the abnormality has an internal origin or not (`internalVerification()`). If it was caused by an internal problem, the `inform-normality` message is sent right after the execution of self-healing actions (`selfHealing()`). Otherwise, the message is sent after external causes are mitigated (`mitigate()`). This mitigation process occurs during the second step of the verification activity (`externalVerification()`), where the provider consults cooperating agents with the aim of determining which possible external cause originated the abnormality. The consultation is made through a `request-probability` message broadcast with the identification of a suspicious agent, the service under investigation, and the quality feature being considered. Recipients can reply with a refusal or an `inform-probability` message, until every agent replies or a deadline expires. The `inform-probability` message contains the likelihood of getting an anomalous reply from the suspicious agent when requesting the given service (`computeProbability()`).

From received replies, the provider computes a score that allows it to identify the external cause of abnormality. If the score falls below a given threshold, the communication link is considered the cause and is handled accordingly (`repairLink()`). Otherwise, if the score is above the threshold, the provider plays the role of a client, instantiating the protocol in a new context.

Figure 5.3: An interaction protocol between components of a system. Clients request services with `request-service` messages, which are replied by providers with corresponding `inform-service` messages. Clients notify abnormal components of quality requirement violations with `inform-abnormality` messages, which are replied with `inform-normality` messages after abnormalities are handled. Providers are able to issue `request-probability` messages to cooperating components, which may reply with `inform-probability` messages.



5.2.3 Agent Behaviour

According to our protocol, an agent can act as (i) a client, by requesting services and collecting information; (ii) a provider, by delivering services and carrying out the process of diagnosing, remediating and solving abnormalities; and (iii) a cooperating agent, by supporting providers when required. Next, we detail these three roles.

5.2.3.1 Client Agent

An agent plays the role of a client when it requests services from providers. During this interaction, it collects data that can also be used later when playing other roles. These data, which include information related to the performance of its providers regarding quality requirements, are collected and stored in structures named interaction traces.

Definition 16 (Interaction Trace) *An interaction trace t is a tuple $\langle m, \mathcal{M}_q, time\rangle$, where m is the traced message; $\mathcal{M}_q : \mathcal{Q} \nrightarrow \mathbb{R}$ is a partial mapping of quality features and their measured values; and time is the time at which the trace was recorded.*

Definition 17 (Agent) *An agent a is a tuple $\langle \mathcal{S}, \mathcal{Q}, \mathcal{R}, \mathcal{T}\rangle$, where $\mathcal{S}$ is the set of services it is able to provide; $\mathcal{Q}$ is the set of quality features it can measure; $\mathcal{R}$ corresponds to its quality requirements from the services it consumes; and $\mathcal{T}$ is the set of interaction traces collected by this component.*

An agent can only measure the quality of a service by using it. Therefore, only `request-service` and their corresponding `inform-service` messages are considered for tracing. After dispatching a service request, the client creates a new interaction trace through the `createTrace()` operation, and adds it to its set of traces. A recently created trace contains only the message with which it is associated. As an example, let agent p_a from Figure 5.2 be defined as

$$\begin{aligned} p_a = \langle &\{a\}, \{response\_time\}, \\ &\mathcal{R}(response\_time) = (response\_time \leq 15), \emptyset\rangle. \end{aligned}$$

Let $m_0 = \langle \#1, \#1, p_a, p_b, request, b, null\rangle$, be the message sent by p_a at instant 5 requesting service b from p_b. The interaction trace $t_0 = \langle m_0, null, null\rangle$ is thus created and added to p_a's set of traces, resulting in

$$\begin{aligned} p_a = \langle \{a\}, \{\mathit{response\_time}\}, \\ \mathcal{R}(\mathit{response\_time}) = (\mathit{response\_time} \leq 15), \{t_0\}\rangle. \end{aligned}$$

We assume every agent has the mechanisms required to measure the performance of providers regarding its quality requirements. Therefore, after receiving a reply to a service request, the client updates the previously created trace with the measured provider's performance as well as the time at which the reply was received and, consequently, the service was provided. It is done through the `updateTrace()` operation. In our example, p_a has a single quality requirement that states that consumed services must be delivered (*response_time*) within 15 units of time. After receiving the reply from p_b at instant 12, p_a updates its trace t_0 to $t_0 = \langle m_0, \mathcal{M}_q(\mathit{response\_time}) = 7, 12\rangle$, indicating that its request was fulfilled in 7 units of time. Constantly tracing their interactions allows clients to build a dataset that represents the performance of their providers over time.

Clients also notify service providers of quality requirement violations. These notifications can be either the result of a self-awareness mechanism or a reaction to a notification received by the client when playing the role of provider. While the former is not the focus of this work, the latter is detailed next, as we describe the behaviour of provider agents.

5.2.3.2 Provider Agent

An agent acting as a provider has the main goal of delivering services to clients with the expected quality. It is done by replying to service requests with the corresponding service output. However, when this goal is not achieved and the provider is notified that a delivered service was provided with an abnormality, a new goal is generated in order to normalise its operation.

In order to achieve this goal, the provider puts into action a strategy to diagnose, remediate and solve the cause of the abnormality. As mentioned in Section 5.2.2, the first step of this strategy comprises the `internalVerification()` activity, which is responsible for checking if the abnormality was introduced by the provider itself. The problem is considered to have an internal cause if, to deliver its service, the provider did not consume any service supplied with a largely different performance than what was usually recorded on its set of interaction traces. Algorithm 2 describes how this activity is carried out. Auxiliary functions are presented in Table 5.1.

Algorithm 2: *internalVerification*(T, q, id_c)

Input: The set T of interaction traces, the violated quality requirement feature q, the identifier id_c of the conversation in which the violation was perceived.

Output: A message informing that the agent operation is normalised.

```
1  $I_a \leftarrow \varnothing$;
2  $T' \leftarrow$ getTraces$(T, id_c)$;
3  foreach $t \in T'$ do
4      $s \leftarrow t[m][s]$;
5      $p \leftarrow t[m][c_r]$;
6      $time \leftarrow t[time]$;
7      $L_{\mathcal{M}_q} \leftarrow$ getMeasurements$(T, s, p, q, time)$;
8      if isAnomalous$(L_{\mathcal{M}_q})$ then
9          $id_m \leftarrow t[m][id_m]$;
10         $I_a \leftarrow I_a \cup \langle s, p, id_m \rangle$;
11 if $I_a = \varnothing$ then                                /* Internal cause */
12     selfHealing();
13     broadcast inform-normality();
14 else                                                 /* External cause */
15     foreach $\langle s, p, id_m \rangle \in I_a$ do
16         mitigate$(s)$;
17         broadcast inform-normality();
18         $score \leftarrow$ externalVerification$(s, p, q)$;
19         if $score \leq threshold$ then                      /* Link issue */
20             repairLink();
21         else                                         /* Provider issue */
22             send inform-abnormality$(q, id_m)$;
23             receive inform-normality();
24         undo();
```

Initially, the traces recorded during the abnormal conversation id_c are retrieved by the `getTraces`(T, id_c) function (line 2). Service s and its provider p are identified for each trace, and the list of all measurements of quality feature q that were taken when consuming s from p until the time the abnormal conversation occurred is retrieved (lines 4–7)—see `getMeasurements`$(T, s, p, q, time)$ function in Table 5.1.

A statistical analysis of the resulting list of measurements is performed in order to check whether p's performance on the abnormal conversation id_c can be classified as anomalous when compared to historical data. This analysis, executed within the `isAnomalous`$(L_{\mathcal{M}_q})$ function (line 8), applies a method called *Tukey's fences* (TUKEY, 1977) to compute the lower and upper boundaries—the fences—that determine the range of normal values from a sample. Any value laying outside these boundaries is considered an outlier and, consequently, an anomaly. To compute these fences, this method takes as input a list of values arranged in an ascending order and identifies its lower (Q_1) and upper

Table 5.1: Description of Auxiliary Functions

Function	**Mathematical Expression**	**Description**
`getTraces` (T, id_c)	$\{t\colon t \in T \wedge$ $m = t[m] \wedge$ $id_c = m[id_c]\}$	Gives the set of traces recorded during conversation with identification id_c.
`getMeasurements` $(T, s, p, q, time)$	$\{\mathcal{M}_q(q)\colon t \in T \wedge$ $m = t[m] \wedge$ $p = m[c_r] \wedge$ $s = m[s] \wedge$ $\mathcal{M}_q = t[\mathcal{M}_q] \wedge$ $time \geq t[time]\}$	Gives the list of measurements of quality feature q obtained from service s delivered by provider p before time *time*.
`getTimes` $(T, s, p, time)$	$\{t[time]\colon t \in T \wedge$ $m = t[m] \wedge$ $p = m[c_r] \wedge$ $s = m[s] \wedge$ $time \geq t[time]\}$	Gives the list of times at which service s was delivered by provider p before time *time*.

(Q_3) quartiles. Q_1 corresponds to the median of the range of values below the median of the entire sample, while Q_3 stands for the median of the range of values above the median of the entire sample. Equations 5.1 and 5.2 are then applied to calculate the lower (b_l) and upper (b_u) boundaries, respectively.

$$b_l = Q_1 - 1.5 * (Q_3 - Q_1) \tag{5.1}$$

$$b_u = Q_3 + 1.5 * (Q_3 - Q_1) \tag{5.2}$$

The `isAnomalous` $(L_{\mathcal{M}_q})$ function thus returns a boolean value that indicates if the last measurement from $L_{\mathcal{M}_q}$ is lower than b_l or greater than b_u when the *Tukey's fences* method takes $L^s_{\mathcal{M}_q}$ as input, being $L^s_{\mathcal{M}_q}$ the result of sorting $L_{\mathcal{M}_q}$ in ascending order. As an example, let a list of quality feature measurements $L_{\mathcal{M}_q}$ be $L_{\mathcal{M}_q} = (8, 7, 11, 8, 8, 9, 47)$. In this list, Q_1 is equal to 8, and Q_3 is equal to 10. Therefore, according to Equations 5.1 and 5.2, $b_l = 5$ and $b_u = 13$. As a result, `isAnomalous` $(L_{\mathcal{M}_q})$ would return true, as 47 is greater than the obtained upper boundary.

The interaction comprising the service s, its provider p, and the message identifier id_m of each trace whose measurement is classified as anomalous is added to a set of anomalous interactions (lines 8–10). If by the end of this first verification step no anomalous interaction is identified, the abnormality cause is considered to be internal. The abstract `self-healing()` operation is thus performed to repair the issue, normalising the agent behaviour, which is broadcast to the system (lines 11–13). Otherwise, if any interaction is classified as anomalous, the cause is considered external to the provider. In this case, for each anomalous interaction, an abstract `mitigate(`s`)` operation is carried out. This activity is expected to remediate the external issue, thus normalising the agent behaviour and similarly broadcasting this to the system (lines 14–17). `self-healing()` and `mitigate(`s`)` are abstract operations because they have the implementation that is suitable to satisfy the needs of the target domain.

Having the external cause mitigated, the second verification step takes place to identify which of the external sources, namely, the abnormal provider p or the communication link, is the current cause of the issue. This verification is carried out by the `externalVerification(`s,p,q`)` operation, whose outcome is a score for the performance of p when providing service s with respect to quality feature q (line 18). If the obtained *score* is less than or equal to a predefined *threshold* (in this thesis, we use the value of 0.5, or 50%), the communication link is considered the source of abnormality, and is thus repaired by the abstract `repairLink()` operation (lines 19–20). If the score is greater than the threshold, the provider p is considered the problem cause. This conclusion follows the rationale that an unusual behaviour from a suspicious provider may be perceived not only by the agent handling the abnormality, but also by cooperating agents that consumed the same service. The provider handling the issue acts as a client, notifying p of its abnormal performance regarding quality feature q when replying to message id_m, and waiting that agent to inform when its operation gets normalised (lines 21–23). Finally, when the problem cause is solved, the abstract `undo()` operation is performed to revoke any reversible measure taken to remediate the problem cause (line 24), as specified by the D^2R^2 + DR strategy.

Algorithm 3 describes how the external verification is performed. First, a message is broadcast to the system requesting to cooperating agents the probability of provider p to present an abnormal behaviour when delivering service s with respect to quality feature q (line 2). Replies are received until a stop condition is met, which can be either a deadline or a predefined number of replies. For each received reply, its sender c_s and the informed probability *prob* are added to a set *Inf* containing received information (lines 3–7).

Agents may differ from each other regarding several characteristics. Consequently, more importance is given to information from more similar agents. This is taken into account to combine received probabilities into a final score. The similarity index $\mathcal{I} : \mathcal{C} \times \mathcal{C} \to [0,1]$ is thus a function that maps a pair of agents $p, q \in \mathcal{C}$ to a value indicating the similarity between p and q. 0 and 1 are the lower and highest similarities, respectively.

Algorithm 3: *externalVerification*(s, p, q)

```
Input: The abnormal service $s$, the suspicious provider $p$, the violated quality
       requirement feature $q$.
Output: A score $score$ for the performance of $p$ be abnormal regarding quality
        feature $q$ when delivering service $s$.
1  $Inf \leftarrow \emptyset$;
2  broadcast request-probability($s,p,q$);
3  while $\neg condition$ do                  /* Deadline or # of replies */
4      $m \leftarrow$ receive inform-probability();
5      $c_s \leftarrow m[c_s]$;
6      $prob \leftarrow m[cont]$;
7      $Inf \leftarrow Inf \cup \langle c_s, prob \rangle$;
8  $prob_w \leftarrow 0.0$;
9  $idx_s \leftarrow 0.0$;
10 foreach $\langle c_s, prob \rangle \in Inf$ do
11     $prob_w \leftarrow prob_w + (prob \times \mathcal{I}(this, c_s))$;
12     $idx_s \leftarrow idx_s + \mathcal{I}(this, c_s)$;
13 $score \leftarrow prob_w / idx_s$;
14 return $score$;
```

In this work, this index is inversely proportional to the distance, measured in hops, between two agents. We assume that every agent has access to this information. The score of a suspicious provider p is computed as the average of probabilities given by cooperating agents weighted by their similarity indexes (lines 8–13), and returned at the end of this activity (line 14).

5.2.3.3 Cooperating Agent Behaviour

A cooperating agent aims to provide required information to agents handling an abnormal behaviour. This is achieved by the computation of the probability of a suspicious agent to provide an anomalous measurement of a given quality feature when delivering a particular service (`computeProbability()`). Algorithm 4 describes how this activity is performed.

First (lines 1–2), the lists of (i) all measurements of a quality feature q taken when consuming service s from p, and (ii) the corresponding times in which these measurements were recorded, are retrieved by the functions `getMeasurements`$(T, s, p, q, time)$ and `getTimes`$(T, s, p, time)$—see Table 5.1. If, at any time, this agent consumed s from p, the probability computation is carried on with the identification of the probability density function f that describes the recorded quality feature measurements. In our approach, given that f is unknown and available measurements constitute a sample from it, f is

Algorithm 4: *computeProbability*(s,p,q)

```
Input: The service $s$, the provider $p$, the quality requirement feature $q$.
Output: A probability $prob$ of $p$ to provide an anomalous measurement of $q$ when
        delivering service $s$.
1 $L_{\mathcal{M}_q} \leftarrow$ getMeasurements$(T,s,p,q,now)$;
2 $L_{time} \leftarrow$ getTimes$(T,s,p,now)$;
3 if $L_{\mathcal{M}_q} \neq \varnothing$ then
4     $f \leftarrow$ getFunction$(L_{\mathcal{M}_q}, L_{time})$;
5     $prob \leftarrow$ integrate$(f, L_{\mathcal{M}_q})$;
6     send inform-probability$(prob)$;
```

estimated through the use of a kernel density estimator.

A *kernel density estimator* (KDE) (PARZEN, 1962) is a non-parametric statistical method able to approximate f using a mixture of kernels K, each of them centred at the points x_i of an available dataset—in our case, the list $L_{\mathcal{M}_q}$ of quality feature measurements. Typically used kernels include Gaussian and Epanechnikov, although any symmetric probability density function can be adopted. A bandwidth h is used to acknowledge the existence of an unknown density of points in the neighbourhood of each point x_i, while weights w_i are used to consider that a point x_i may have higher surrounding densities than other points. The resulting probability density function $\hat{f}_h$ can thus be used to estimate the probability of any point x. A KDE is thus defined as

$$\hat{f}_h(x) = \sum_{i=1}^{n} w_i K_h(x - x_i), \tag{5.3}$$

where $K_h(x) = K(x/h)/h$ for kernel K and bandwidth h, and $\sum_{i=1}^{n} w_i = 1$. Recently collected measurements may have higher informative power than those earlier collected. Therefore, the weight w_i assigned to each measurement $x_i \in L_{\mathcal{M}_q}$ is proportional to the recency of that measurement. This is calculated according to Equation 5.4.

$$w_i = L_{time_i} \times \frac{1}{\sum_{j=1}^{|L_{time}|} L_{time_j}} \tag{5.4}$$

The obtained function $\hat{f}_h$ results from the execution of the `getFunction` $(L_{\mathcal{M}_q}, L_{time})$ operation (line 4).

Finally, the `integrate` $(f, L_{\mathcal{M}_q})$ operation is executed to determine the probability *prob* of obtaining an anomalous measurement from p (line 5). In this operation, the *Tukey's fences* method (Section 5.2.3.2) is applied to the list $L_{\mathcal{M}_q}$ of measurements with the aim of identifying the lower (b_l) and upper (b_u) boundaries that delimit the range of normal values. One minus f integrated using these boundaries (Equation 5.5) gives the estimated distribution. It is the probability *prob* of a random value from that distribution to fall in a range below b_l or above b_u, thus becoming an anomalous value. At the end, this obtained probability is informed to the agent that requested it (line 6).

$$prob = 1 - \int_{b_l}^{b_u} f \tag{5.5}$$

To exemplify this process, let n be a cooperating agent that receives a request to provide the probability of p_b behaving abnormally when delivering service b with respect to *response_time*. Let

$$L_{\mathcal{M}_q(response\_time)} = (8, 10, 9, 9, 11, 12, 10, 9, 12, 20, 43)$$

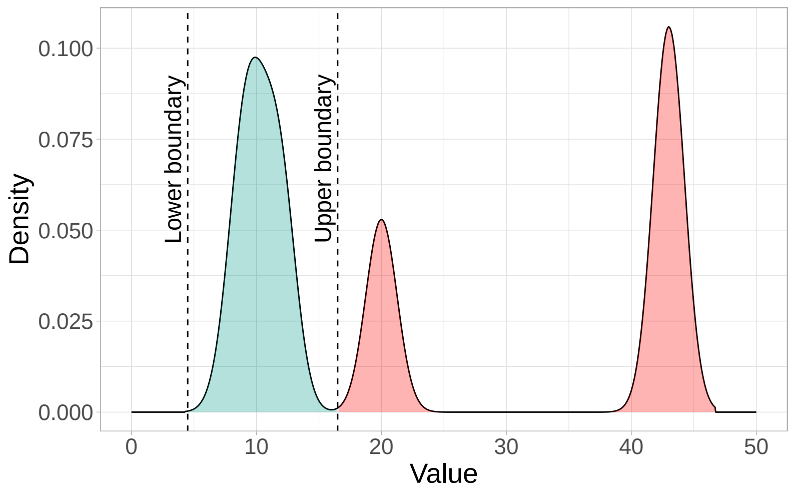
be the list of quality feature measurements and

$$L_{time} = (5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55)$$

the list of times at which these measurements were registered. The estimated probability distribution f obtained by n after performing the `getFunction` $(L_{\mathcal{M}_q(response\_time)}, L_{time})$ operation is depicted in Figure 5.4. In our example, the lower and upper boundaries of $L_{\mathcal{M}_q(response\_time)}$ are equal to 4.5 and 16.5, respectively. It means that, while any value that falls within these boundaries are considered normal (blue shaded in Figure 5.4), values that fall in any region outside them comprise an anomaly (red shaded area). After carrying out the `integrate` $(f, L_{\mathcal{M}_q(response\_time)})$ operation, n verifies that the probability *prob* of obtaining an anomalous measurement when requesting service b from p_b with respect to *response_time* is approximately 0.49 or 49%.

The interaction protocol and the algorithms that specify the behaviour of agents playing each described role were implemented as an extension of the BDI4JADE (NUNES; LUCENA; LUCK, 2011) platform. Activities comprising the cooperative behaviour are encapsulated in plans added to a `CooperativeCapability`, which extends the one implementing the action management technique introduced in Chapter 4. In order to adopt our cooperative technique, BDI agents are thus only required to instantiate this capability.

Figure 5.4: The estimated probability distribution obtained after executing `getFunction`$(L_{\mathcal{M}_q(response\_time)}, L_{time})$. Values laying outside the lower and upper boundaries are considered anomalous.



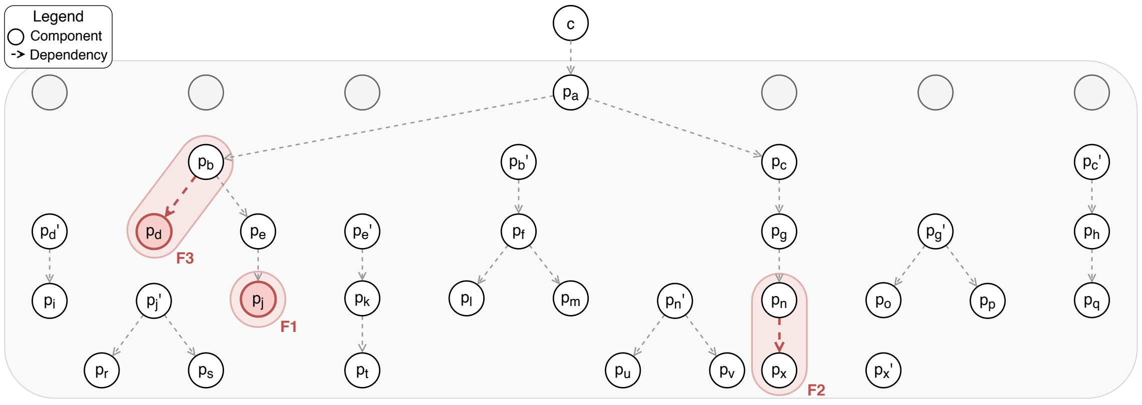
5.3 Evaluation

Having detailed our protocol and the behaviour of each role, we now evaluate our proposal with an empirical evaluation. In this evaluation we examine the performance of a MAS that adopts our cooperative approach as a supporting technique for carrying out the remediation, diagnosis and recovery operations specified by the D^2R^2 + DR strategy. This MAS is evaluated with respect to its ability to dynamically diagnose the root cause of problems in different locations and coming from different sources. We compare the obtained results with the performance of the same system when applying two other problem-solving strategies. In the first one, which we named *passive strategy*, agents ignore any notification of abnormality sent to them. In the second one, named *remedial strategy*, these agents are able to mitigate abnormalities but do not have the mechanisms to diagnose and solve their causes. As discussed in Chapter 3, even though there are many existing solutions focused on the diagnosis and resolution of problems causes in multiagent systems, they all differ from our approach in some key aspects. First, most of them do not address both facets of the problem, focusing either only on the diagnosing or on the problem solving aspect. Solutions capable of identifying problem causes differ from ours mainly regarding the size and complexity of the scenarios they are able to handle, being inefficient when dealing with scenarios comprising more than a fixed (and considerably low) number of agents. Solutions focused on problem solving, in turn, usually rely on centralised information in order to deal with challenging situations. Considering these characteristics, a direct comparison with existing solutions is not meaningfull. Next, we present the procedure adopted in our evaluation, followed by its results and discussion.

5.3.1 Procedure

Many MAS are open, so agents can join or leave the system at runtime, leading to dynamic scenarios. Consequently, there are no static agent topologies used as a standard for simulations. It is thus a common practice to elaborate synthetic topologies and workloads to simulate realistic scenarios in order to evaluate the proposed approaches (DÖTTERL et al., 2017). To evaluate our technique, we implemented the scenario depicted in Figure 5.5. It represents a service-oriented system comprising 37 autonomous agents able to interact with each other by consuming and delivering services, and one external agent acting as a top-level client. Services of different providers have varying costs, which means that the same service may be more or less expensive when requested from different sources. With the exception of unnamed agents, which use services from any provider despite of their costs, every other agent has a secondary goal of minimising the cost associated with consuming services. Dependencies represented on Figure 5.5 indicate the initial providers used by each agent.

Figure 5.5: A service-oriented system comprising 37 autonomous agents and an external client *c*. *F1*, *F2* and *F3* are failures introduced to the system to affect different agents and communication links.



As introduced, our technique includes abstract activities that are specific to concrete implementations. In this experiment, the `selfHealing()` and `repairLink()` activities are simulated. In real world scenarios, these operations are expected to comprise, e.g., the implementation of one of the many existing self-healing, self-configuration or repairing strategies (GHOSH et al., 2007). The implemented `mitigate(`s`)` operation, in turn, replaces the current (abnormal) provider of the given service s with a different (and presumably more costly) one. An alternative implementation of this operation may be, for instance, a prediction of the output of a given service based on the current context. Finally, the stop condition for providers to wait for replies from cooperating agents in the `externalVerification(`s, p, q`)` operation is set to a 5 seconds deadline.

Agent c has a quality requirement $\mathcal{M}_q(\mathit{response\_time}) = (\mathit{response\_time} \leq 250)$, which specifies that the services it consumes must be provided in no longer than 250 ms. In our experiment, services have an initial delivery time of 10 ms. This time increases by 250 ms for each failure affecting the service provision, which can occur either on the service provider or in the communication link between it and its client.

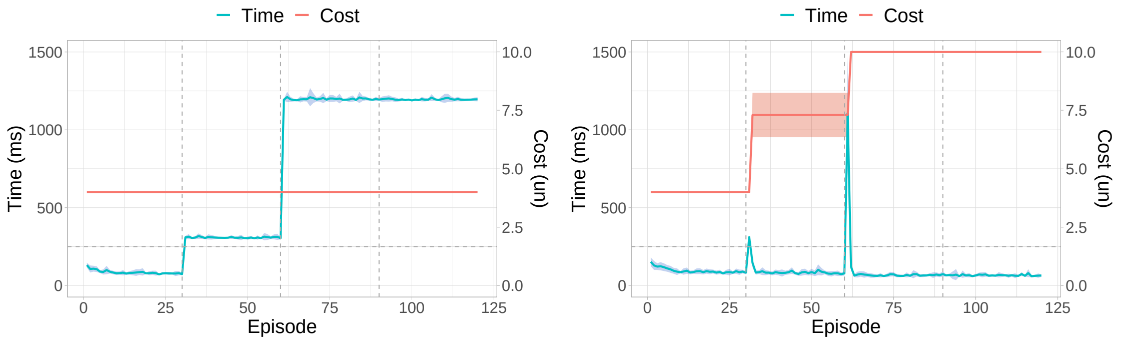
Our experiment thus consists of measuring the cost and response time observed by c when consuming service a from p_a. We ran a simulation with 120 episodes, each of them comprising the following sequence of steps: (i) c requests service a from p_a; (ii) unnamed agents request different services within the system; and (iii) the cost and response time measured by c are recorded. Three failures, *F1*, *F2*, and *F3*, are injected on the simulation at the 30th, 60th and 90th episode, respectively, with the aim of causing abnormalities on service provision. In *F1*, agent p_j is targeted, increasing its response time and becoming unable to provide its service as usual. F2 affects the link between agents p_n and p_x, thus slowing their communication down. Finally, F3 disturbs both the functioning of agent p_d and its link with p_b. These points of failure are depicted in Figure 5.5.

5.3.2 Results and Discussion

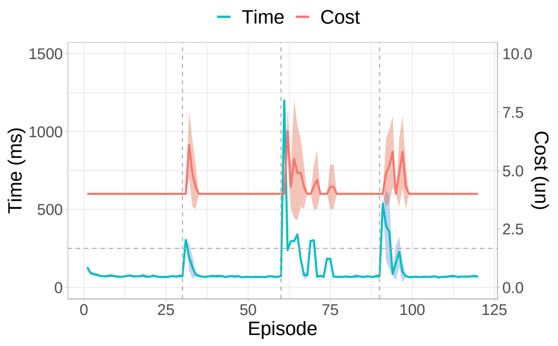
We executed our simulation 10 times for each adopted problem-solving strategy. The results are discussed focusing on the system behaviour observed during the execution of our experiment. These results are presented in Figure 5.6, where blue and red lines represent the average response time and average cost observed by agent c in each episode, respectively, and shaded areas represent their corresponding standard deviation.

Figure 5.6a presents the results obtained by adopting the passive strategy. It is possible to notice that, even with slightly higher values caused by the initialisation procedure at the beginning of the simulation, the response time measured by c maintains an average of $\approx$85 ms until the occurrence of the first failure. It is important to highlight that, as in our implementation agents have a single execution thread—meaning that service requests are queued in order to be fulfilled—the response time perceived by a client may vary according to the amount of requests being processed by its provider. This explains why the measurement recorded by c when consuming a from p_a is usually greater than the 40 ms that would be expected if there were no other agents consuming that and other related services.

Figure 5.6: Simulation results (time and costs by episode).



(a) Passive strategy (b) Remedial strategy



(c) Cooperative strategy

After the introduction of *F1*, the response time increases and stays above the quality requirement threshold at an average of 308.5 ms. It increases again after the occurrence of *F2* and remains stable from that moment, even with the occurrence of the third failure, which ends up being subsumed by the former two. Such an outcome takes place because, when provider p_a adopts the passive strategy, it simply ignores notifications of abnormality received from c. As a consequence, in order to deliver a, that agent keeps consuming services from providers affected by the introduced failures. The observed cost also reflects such a behaviour, standing steady for the entire simulation due to the fact that initial providers are not replaced with others.

The scenario changes completely when we consider results from simulations in which the remedial strategy is adopted. They are shown in Figure 5.6b. After the occurrence of *F1*, p_a's response time increases and surpasses the quality threshold, peaking at an average of 311 ms, but quickly decreasing and stabilising at an average of $\approx$85 ms. It happens because, by following the remedial strategy, agents are able to mitigate abnormalities coming from external sources by changing their service providers. Consequently, after receiving a notification of abnormality from c, p_a replaces p_b with p_b', an agent that does not make use—neither directly or indirectly—of the abnormal service provided by p_j in order to deliver b. It is also because of this provider replacement that the cost increases. The introduction of *F2* results on the same behaviour pattern, with response time increasing above the quality threshold, peaking at 740.6 ms, and decreasing after provider p_c is replaced with p_c', which also results in a higher associated cost. It is interesting to notice that the occurrence of *F3* does not affect the performance of p_a perceived by c. It is explained by that fact that p_a is no longer a client of p_b and, as a consequence, degraded services provided by that agent are not consumed.

Considering that the goals of agent c are to consume service a while satisfying its quality requirements and minimising the associated cost, our cooperative strategy presents an optimal trade-off towards the achievement of both objectives. The outcomes with this strategy are shown in Figure 5.6c. After *F1*, it is possible to observe the increasing response time, which peaks at 305 ms and decreases after that, following the same pattern presented when adopting the remedial strategy. In contrast to this strategy, the measured cost returns to its initial level after an increase due to the adopted mitigation. Such a behaviour is explained by the propagating nature of our approach as well as the existence of the `undo()` operation inherited from the D^2R^2 + DR strategy.

When the first failure affects p_j and its degraded performance spreads to its clients, p_a is notified of an abnormality. It thus carries out the cause verification operation and identifies the external nature of the issue, which is remediated with the replacement of p_b with p_b'. Because of this change, the response time perceived by c returns to an acceptable level, but at the expense of an increased cost. After consulting the cooperating—unnamed—agents, and identifying the external source of the issue, p_a notifies p_b of the detected abnormality. That agent thus carries out the same process, identifying the external origin of the problem, mitigating it and notifying the suspicious provider of its abnormality. After mitigating the problem cause, p_b informs p_a that its operation was normalised, which allows p_a to revert the remediation action taken and go back to consuming service b from p_b, thus reducing its cost. Meanwhile, p_e and p_j performs the same identification process after being notified of their abnormalities by p_b and p_e, respectively. When p_j identifies that its degraded performance was caused by an internal failure, the `selfHealing()` operation is executed, normalising the agent behaviour and allowing the system to return to its initial setup.

The same process is executed by the system when failures *F1* and *F2* are introduced. In Figure 5.6c, it is possible to notice some peaks after the occurrence of each of these failures. It happens because, in some executions, cooperating components are unable to immediately classify some suspicious agents as abnormal ones, which results on the incorrect problem cause being addressed. For instance, consider that, after the occurrence of *F2*, agent p_c mitigated an external cause by replacing p_g with p_g' and, by consulting its cooperating agents, it (mistakenly) decided that p_g did not present any abnormality. In this case, p_c would repair its communication link with p_g, acknowledge the problem as solved, and go back to consume services from that agent. As a consequence, the abnormality would reappear and be notified once again.

Nevertheless, this situation demonstrates that the execution of some additional episodes is sufficient to allow cooperating agents to correctly recognise the existence of abnormalities on suspicious components and to provide conditions for the system to find the correct root cause of its problems. Therefore, even with the introduction of three failures from different sources at different levels, at the end of our simulation the system is able to correctly identify and solve failure causes, with agent c satisfying its quality requirements at a reduced cost.

Finally, as shown in Table 5.2, simulations in which the system adopts our cooperative strategy took, on average, 616.5 seconds to be executed, which represents a reduction of 10.57% and 0.32% when compared to the adoption of passive (689.4 s) and remedial (618.5 s) strategies, respectively. The accumulated cost presented by our approach, in turn, was 3.81% higher (498.3 unities of cost) when compared to the passive strategy (480.0 unities of cost), and 46.59% lower in comparison to the remedial strategy (933.0 unities of cost).

Table 5.2: Average Execution Time and Accumulated Cost

Strategy	**Execution Time (s)**	**Accumulated Cost (un)**
Passive	689.4	480.0
Remedial	618.5	933.0
Cooperative	616.5	498.3

These results show that our approach is able to combine the benefits from both, passive and remedial strategies, into a single solution. However, to achieve this outcome there are some prerequisites that must be considered. First, as already mentioned, it is required that cooperating agents have consumed services from a suspicious agent after the occurrence of an abnormality in order to be able to correctly identify it. Otherwise, they could provide information that does not represent the current behaviour of that provider and, consequently, the agent handling the abnormality would not be able to correctly diagnose the source of the issue. Strategies to overcome this limitation may include, for instance, enforcing cooperating agents to issue a service request in order to update their sets of interaction traces, or limit agents able to cooperate to those whose last requirements to a suspicious provider occurred within a given time interval. The applicability of these strategies and their impact on our solution are subject to further investigation.

Our approach also relies on the quality of the remedial actions taken by the agent during the process of mitigating external causes. In our evaluation, we replace service providers to normalise system operation while the root cause of the problem is diagnosed and solved. If, for instance, the temporary provider also exhibits an abnormal behaviour, the entire identification activity would be triggered for the abnormality presented by that agent, thus delaying the restoration of a normal operation. Therefore, whatever are the remediation actions implemented in a given domain, it is essential to—at least temporarily—ensure their effectiveness in satisfying existing quality feature requirements. How to guarantee such a characteristic is another open research subject.

5.4 Final Remarks

This chapter presented a technique that, when integrated to our solution for coordinating the remedial behaviour, allows autonomous agents to cooperate with each other in order to diagnose the root causes of behaviour abnormalities, and remediate and solve them with the aim of normalising system operation. This cooperative technique comprises a protocol that provides guidance on how agents should interact as well as the specification of how agents are expected to behave when playing the role of clients, providers and cooperating agents. We remove the need for human involvement by taking advantage of information collected by individual agents at runtime and using it as the source of knowledge to diagnose the root causes of abnormal behaviour. The centralised information and processing is also relieved when we acknowledge that pieces of information may be spread throughout the system, and thus promote the cooperation among agents. The proposed technique was evaluated with an empirical experiment in which we simulated a distributed environment comprising autonomous agents. Results indicate that our solution is able to diagnose and handle a variety of failures at different levels, keeping the system operating with desired quality and reduced costs.

Nevertheless, abstract operations specified in our solution are still candidates for the development of more flexible and adaptive implementations. In particular, the `undo()` operation, which abstracts the recover operation from the D^2R^2 + DR strategy. This operation requires an explicit description of which remedial actions must be reverted and how it must be done. The next chapter addresses this limitation.

6 ACTION REVERSION

Remediation actions are performed to tackle the (negative) effects of an underlying problem when certain constraints limit the resolution of its causes. These actions prevent the occurrence of further problem effects before causes are addressed to permanently solve the problem. After these causes are permanently solved, changes made by remediation actions often must be reverted. That is, there is a need for *cleaning up (or reverting)* the effects of performed remediation actions.

In this chapter, we introduce a technique to provide agents with the ability of autonomously reverting actions. Although our focus is to revert remediation actions, our approach is general enough to be used in any recoverable circumstance, such as failure handling and task aborting situations. Our formal framework involves the specification of three main steps, integrated to the BDI reasoning cycle, to revert actions: (i) monitoring and recording changes made by plans; (ii) recognising the circumstances in which these actions should be reverted; and (iii) executing the reverting process.

6.1 Motivation Scenario

To illustrate the problem we address, we present an example scenario in which there is a need for reverting actions. The example is in the context of smart homes, part of a system in which there are mechanisms to promote *safety* to residents. Consider the scenario of dealing with a carbon monoxide leaking. Carbon monoxide (CO) is an odourless, colourless and tasteless gas produced by the incomplete burn of carbon-based fuels, such as natural gas and coal. Due to its undetectable nature, CO became the second most common cause of deaths by non-medical poisoning in the United States, having caused a total of 6136 deaths by unintentional poisoning between 1999 and 2012 (SIRCAR et al., 2015). Most of these deaths occurred at home and were usually related to malfunctioning in heating systems, water heaters, cooking equipment, and other fuel burning appliances. Therefore, the use of devices able to alert about the presence of high concentrations of CO, the so-called CO detectors, became common in most homes, as well as in industrial and commercial facilities.

Assume that a house is equipped with one of these CO detectors as well as with a series of other intelligent devices, such as different sensors (e.g., of temperature or presence) and actuators (e.g., lights, alarms and valve controllers). These devices are able to coordinate their actions in order to perform a wide range of tasks.

The expected behaviour in our smart home is that when the CO detector identifies a high amount of gas in the house, a set of actions are taken. First, an *alarm goes off* aiming to notify the residents about the problem, *lights are turned on* and *doors are unlocked* in order to allow the evacuation of the place. Moreover, *windows are opened* and the *ventilation system is turned on* to reduce the concentration of CO. These are remediation actions that need to be taken immediately. Next, the cause of the high amount of gas must be determined. By an automated diagnosing process, a malfunctioning in the water heater is identified. Then, a valve controller is activated and *interrupts the flow of natural gas* to such device and schedules a repair with a maintenance company, thus permanently solving the leaking cause. Once the concentration of CO is reduced, remediation actions to deal with the problem are undone. For instance, the alarm is silenced, lights and the ventilation system are turned off, and the windows and doors can be closed and locked, respectively. The valve responsible for providing gas to the water heater, however, remains closed.

This example allows us to make key observations. First, performing tasks may affect the system and its environment in many different ways. In our example, several environment variables are modified to achieve the goal of reducing the concentration of CO. How to keep track of the effects related to the execution of such tasks becomes a challenge to software systems, in particular if we consider that their environment may be shared with other systems, thus being affected by actions from different sources. In addition, it is possible to notice that not every task effect must be reverted. As an example, consider our scenario in which one of the effects obtained by opening windows and turning on the ventilation system is to have the concentration of CO reduced. When the reverting process is triggered, the desired outcome is, among others, to have windows closed and the ventilation system turned off, but not to have an increased concentration of CO. Therefore, in order to allow software systems to carry out this process, not only the ability to monitor the effects of performed tasks is required, but also the identification of which of those effects must be undone. Finally, the need for reverting actions only arises when particular context conditions are met. Our example shows, for instance, that lights and the ventilation system are turned off only when the CO concentration is below a given threshold and the flow of natural gas to the water heater is interrupted. It is thus necessary for systems featuring reverting actions to provide a means for these triggering conditions to be specified.

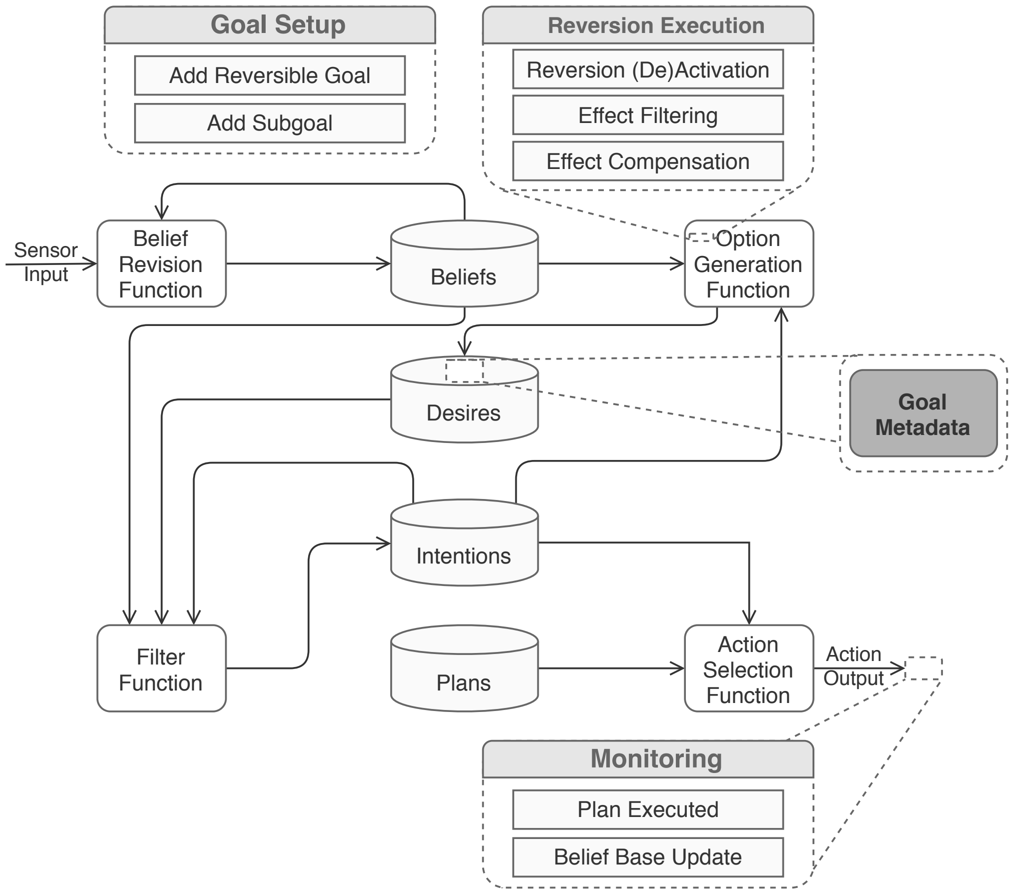
6.2 A Formal Framework for Reverting BDI Agent Actions

The observations made above are related to issues that must be addressed by our approach to autonomously revert actions. Next, we overview our proposed framework, then formalise its key concepts and describe its steps in next sections.

6.2.1 Framework Overview

To make BDI agents able to autonomously manage the reversion process of actions, we extend the BDI architecture with additional supporting data and add new operations to be incorporated within the BDI reasoning cycle. The operations of our reversion process and how they are integrated to the BDI architecture are presented in Figure 6.1. It shows the BDI components as well as the steps of the reasoning cycle, and highlights the metadata and operations that comprise our framework.

Figure 6.1: Overview of the Reversion Framework.



The supporting data added to agents consists of metadata associated with goals to (i) represent the conditions in which actions taken to achieve these goals must be reverted,

(ii) record executed actions, and also (iii) keep contextual information to evaluate whether triggering conditions of the reversion process are met.

The operations comprising our reversion process are grouped into three activities, namely goal setup, monitoring, and reversion execution. The *goal setup* activity consists of the creation and initialisation of the metadata needed to perform the reversion process, which is done when goals are added to the agent. The *monitoring* activity consists of the observation of agent behaviour and recording of (i) changes in the (external or internal) environment, and (ii) executed plans to verify whether reversion conditions are met. Recorded data is stored as metadata associated with individual goals. This step takes plan in parallel to the execution of plans to achieve goals.

Finally, the *reversion execution* has three main tasks, reversion (de)activation, effect filtering and effect compensation, which occur as part of the option generation function. In the reversion (de)activation, each goal metadata is evaluated to verify whether the goal becomes ineligible to be reverted or the system reached a condition that triggers the goal reversion. If the former occurs, the goal metadata is discarded and can no longer be reverted. If the latter is the case, the recorded changes in the environment (i.e. effects) are filtered to select those to be reverted. As explained in our example, not all effects must or can be reverted. Last, to compensate the effects that must be reverted, corresponding goals are generated. These goals are then handled by the BDI reasoning cycle as any other goal, to which plans will be selected and executed in order to achieve them.

6.2.2 Model Formalisation

The existing BDI architecture does not encompass all of the structures required to provide agents with the ability to manage the proposed action reversion process, such as goal metadata. Therefore, we next specify extensions proposed to this architecture with the structures needed to manage this process. Our specification is formalised using the Z language (SPIVEY, 1988). We also formalise key concepts from the BDI architecture needed for our customisation. A complete formalisation of the BDI architecture can be seen elsewhere (D'INVERNO; LUCK, 2004).

Information about the environment and the internal state of an agent is represented by logical predicates, which can be true or false. Let *PREDICATE* be the set of all possible predicates, and *BOOLEAN* be the set of boolean values *True* and *False*. The knowledge of an agent is then represented by a set of predicates, which are part of the belief base of the agent. Each predicate is evaluated as true or false according to the *belief* function in the *BeliefBase* schema, as shown below.

$[\mathit{PREDICATE}]$

$\mathit{BOOLEAN} ::= \mathit{True} \mid \mathit{False}$

BeliefBase

$\mathit{knowledge} : \mathbb{P}\,\mathit{PREDICATE}$

$\mathit{belief} : \mathit{PREDICATE} \mathrel{+\!\!\!\!\!\rightarrow} \mathit{BOOLEAN}$

$\mathit{knowledge} = \operatorname{dom} \mathit{belief}$

Goals to be achieved by an agent are associated with a predicate, and can be of two types. They can be an *Achievement* goal, meaning that the agent desires the predicate to be part of its knowledge and to believe that it is *True*, or a *Query* goal, meaning that the agent desires the predicate to be part of its knowledge. The set *TYPE* comprises all possible goal types. Goals are associated with an end state, which represents the result of making it an intention and trying to achieve it. Let *ENDSTATE* be the set of all possible end states of a goal. A goal is said to be *Achieved* when a plan successfully reached the desired state of world. The end state *Failed* indicates that the execution of the last executed plan failed while trying to achieve the goal, while *NoLongerDesired* is assigned to goals that are no longer desired by the agent. Finally, goals that are still being attempted to be achieved have their end state set as *Nil*.[1]

$\mathit{TYPE} ::= \mathit{Achievement} \mid \mathit{Query}$

$\mathit{ENDSTATE} ::= \mathit{Achieved} \mid \mathit{Failed} \mid \mathit{NoLongerDesired} \mid \mathit{Nil}$

Goal

$\mathit{predicate} : \mathit{PREDICATE}$

$\mathit{type} : \mathit{TYPE}$

$\mathit{endState} : \mathit{ENDSTATE}$

When the filter function of an agent selects a goal to which the agent will commit to

[1]Here, we make an abuse of notation by using *Nil* as a wildcard that specifies a null value, which is used later as a member of sets of different types.

achieve, the goal becomes an intention. Then, by the plan selection function, a plan is selected to be executed to achieve that goal/intention. To do so, the plan is instantiated, considering the current context, as a plan instance. We omit the formalisation of the concepts of *Plan* and *PlanInstance* (as they play less important roles in our solution), and simplify the specification of *Intention* as shown below, which is referred later.

Intention

$goal : Goal$

$planInstance : PlanInstance$

$planInstance = Nil \lor goal = planInstance.goal$

In order to be able to revert actions associated with the achievement of goals, metadata is stored and maintained. *GoalMetadata* is the concept that is the core of the reversion process, being related to a specific goal. Moreover, when a goal becomes an intention, its corresponding metadata also refers to that intention.

Here, we briefly introduce the elements of *GoalMetadata*, in the order that they appear in the schema. Semantic implications of its variables are discussed in next section, when we define the operations of the rollback process. A reversion trigger and an indication of rollback in the case of a plan failure establish the conditions in which reversion will be activated. Goals can be reverted after they are achieved, therefore their metadata is kept stored even after their achievement. However, we limit this information to be stored for a specified amount of time. There are two options to do so, by specifying either a maximum number of plans or a maximum amount of time that can be executed or elapsed, respectively, between the moment the goal was achieved and the moment of the reversion. Therefore, these are discarding conditions of the goal metadata. In order to control whether these conditions are met, the time in which the goal was achieved and a counter that registers the number of plans that have been executed since that are stored. Finally, changes in the agent belief base are stored as a trace, which indicates what should be reverted during the reversion process. For changed predicates, we keep a sequence of pairs of boolean and natural values. Predicates in the domain of this function denote belief predicates whose evaluation value was modified during the achievement of a given goal. The pairs of boolean and natural values represent, respectively, the new evaluation value of the associated predicate and the time at which that change occurred.

GoalMetadata

$goal : Goal$

$intention : Intention$

$reversionTrigger : PREDICATE$

$rollback : BOOLEAN$

$maxExecutedPlans : \mathbb{N}$

$maxTime : \mathbb{N}$

$achievedTime : \mathbb{N}$

$planCounter : \mathbb{N}$

$beliefChangeTrace : PREDICATE \pfun \operatorname{seq}(BOOLEAN \times \mathbb{N}_1)$

$intention = Nil \lor goal = intention.goal$

From these previous definitions, we are now able to complete the specification of our customised BDI model by introducing the concept of *Agent*. An *Agent* has a *beliefBase*, which captures the agent knowledge, and a set of current *goals* with their associated *goalMetadata*. Moreover, in order to adequately revert subgoals of a parent goal, goal parents are kept in the *parentGoal* function. Goals that the agent is committed to achieve are kept as *intentions*. These intentions have instances of plans maintained by the agent in its *planLibrary*.

Agent

$beliefBase : BeliefBase$

$goals : \mathbb{P}\ Goal$

$goalMetadata : Goal \pinj GoalMetadata$

$parentGoal : Goal \pfun Goal$

$intentions : \mathbb{P}\ Intention$

$planLibrary : \mathbb{P}\ Plan$

$\operatorname{dom} goalMetadata \subseteq goals$

$\operatorname{dom} parentGoal \subseteq goals$

$\operatorname{ran} parentGoal \subseteq goals$

$\forall i : Intention \mid i \in intentions \bullet i.goal \in goals$

6.3 Framework Activities and Operations

Based on the introduced formal definitions of our framework, we next detail the operations associated with each of its three activities.

6.3.1 Goal Setup

The motivational state of a BDI agent is kept as a set of goals. Goals can be added to the agent by simply including them in the set of agent goals. When a goal that can be reverted is added to the agent, additional data is needed. Therefore, a new operation *AddReversibleGoal* is specified for the agent. This operation receives as parameters the reversion (trigger and rollback) and discarding (maximum executed plans or time) conditions associated with the goal. As result of this operation a goal and its metadata are added to the agent.

AddReversibleGoal

$$
\begin{array}{l}
\Delta Agent \\
predicate? : PREDICATE \\
type? : TYPE \\
reversionTrigger? : PREDICATE \\
rollback? : BOOLEAN \\
maxExecutedPlans? : \mathbb{N} \\
maxTime? : \mathbb{N} \\
\hline
\forall goal : Goal \mid goal \in goals \bullet goal.predicate \neq predicate? \lor \\
\quad (goal.predicate = predicate? \land goal.type \neq type?) \\
\\
(\mathbf{let}\ goal == (\mu\, g : Goal \mid g.predicate = predicate? \land g.type = type? \land \\
\quad g.endState = Nil) \bullet goals' = goals \cup \{goal\} \land \\
\qquad (\mathbf{let}\ metadata == (\mu\, m : GoalMetadata \mid m.goal = goal \land \\
\qquad\quad m.intention = Nil \land m.reversionTrigger = reversionTrigger? \land \\
\qquad\quad m.rollback = rollBack? \land \\
\qquad\quad m.maxExecutedPlans = maxExecutedPlans? \land \\
\qquad\quad m.maxTime = maxTime? \land m.achievedTime = Nil \land \\
\qquad\quad m.planCounter = 0 \land m.beliefChangeTrace = \emptyset) \bullet \\
\quad goalMetadata' = goalMetadata \cup (\{goal\} \times \{metadata\})))
\end{array}
$$

Reversible goals are thus goals that have associated metadata. Given that these metadata include the intention associated with a goal, it must be updated when the goal becomes an intention. The schema below shows this update, specifying the changes that occur when a goal is made an agent intention.

$$
\begin{array}{l}
\textit{MakeIntention} \\
\hline
\Delta \mathit{Agent} \\
\mathit{goal?} : \mathit{Goal} \\
\hline
\mathit{goal?} \in \mathit{goals} \\
\mathit{goal?} \in \operatorname{dom} \mathit{goalMetadata} \\
\\
(\mathbf{let}\ \mathit{intention} == (\mu\, i : \mathit{Intention} \mid i.\mathit{goal} = \mathit{goal?} \land \\
\quad i.\mathit{planInstance} = \mathit{Nil}) \bullet \mathit{intentions}' = \mathit{intentions} \cup \{\mathit{intention}\} \land \\
\quad\quad (\mathbf{let}\ \mathit{metadata} == \mathit{goalMetadata}(\mathit{goal?}) \bullet \\
\quad\quad\quad \mathit{metadata}.\mathit{intention} = \mathit{intention} \land \\
\quad\quad\quad \mathit{goalMetadata}' = \mathit{goalMetadata} \cup (\{\mathit{goal?}\} \times \{\mathit{metadata}\})))
\end{array}
$$

When a reversible goal is achieved, its corresponding metadata is updated with the aim of registering the time in which this change of state occurred. Such update is shown by the *AddAchievedTime* schema.

$$
\begin{array}{l}
\textit{AddAchievedTime} \\
\hline
\Delta \mathit{Agent} \\
\mathit{goal?} : \mathit{Goal} \\
\mathit{time?} : \mathbb{N} \\
\hline
\mathit{goal?} \in \mathit{goals} \\
\mathit{goal?} \in \operatorname{dom} \mathit{goalMetadata} \\
\mathit{goal?}.\mathit{endState} = \mathit{Achieved} \\
\mathit{goalMetadata}(\mathit{goal?}).\mathit{achievedTime} = \mathit{Nil} \\
\\
(\mathbf{let}\ \mathit{metadata} == \mathit{goalMetadata}(\mathit{goal?}) \bullet \mathit{metadata}.\mathit{achievedTime} = \mathit{time?} \land \\
\quad \mathit{goalMetadata}' = \mathit{goalMetadata} \cup (\{\mathit{goal?}\} \times \{\mathit{metadata}\}))
\end{array}
$$

6.3.2 Monitoring

In order to revert goals, goal metadata must capture what should be reverted. This information comes from the results of actions performed by plans, which cause changes in the environment (which are perceived by the agent) or in its internal state. Therefore, reversion information is derived from changes that occur in the agent belief base while executing the instance of a plan. We do not consider all belief changes that simply occur while a plan is executing, for example as a consequence of receiving an external event, but belief changes that occur due to actions part of the plan being monitored. Consequently, belief changes that rely on predefined updating rules based on existing agent beliefs are not registered by the monitoring activity. The *UpdateBeliefBase* operation, shown as follows, registers changes in the belief value of a predicate from the belief base by recording this change in the *beliefChangeTrace*.

Given that belief changes may be originated from plans achieving either reversible or non-reversible goals, such as subgoals, we must consider different situations to guarantee the correct identification of the metadata to store such information. The function *selectParentGoal*, used in the *UpdateBeliefBase* operation, supports this by providing, given a goal and a parent goal mapping, the top level reversible goal considering the trees of goals and their sub-goals. Top level goals are those with which the collected data must be associated.

$$\left|\ \mathit{selectParentGoal} : \mathit{Goal} \times (\mathit{Goal} \nrightarrow \mathit{Goal}) \longrightarrow \mathit{Goal}\right.$$

During the execution of an instance of a plan, subgoals to be achieved can be added to the agent. In this case, if the plan instance is executing to achieve a reversible goal, belief changes that occur as a consequence of the achievement of the subgoal are also recorded in the metadata of the parent goal. The relationship between goals are stored within the agent in the *parentGoal* function, which is updated when a subgoal is added, as shown in the *AddSubgoal* operation. Finally, when plans are successfully executed, counters associated with goal metadata must be updated because they are used as criteria to discard metadata. The *IncreasePlanCounter* operation is executed in response to an event that occurs when a plan execution is completed. It increases the counters associated with plan executions of all metadata of already achieved goals.

UpdateBeliefBase

$$
\begin{array}{l}
\Delta Agent \\
g? : Goal \\
p? : Predicate \\
value? : BOOLEAN \\
time? : \mathbb{N} \\
\hline
(p? \in beliefBase.knowledge \Rightarrow beliefBase'.beliefs(p) = value?) \wedge \\
(p? \notin beliefBase.knowledge \Rightarrow beliefBase' = beliefBase \cup (\{p?\} \times \{value?\})) \\
\\
(\mathbf{let}\ goal == selectParentGoal\,(g?, parentGoal) \bullet \\
(\mathbf{let}\ m == goalMetadata\,(goal) \bullet \\
\quad (p? \in \operatorname{dom} m.beliefChangeTrace \Rightarrow \\
\quad (\mathbf{let}\ newTrace == m.beliefChangeTrace(p?) \frown \langle (value?, time?) \rangle \bullet \\
\quad\quad m.beliefChangeTrace = m.beliefChangeTrace \cup \\
\quad\quad\quad (\{predicate?\} \times \{newTrace\}) \wedge \\
\quad\quad goalMetadata' = goalMetadata \cup (\{m.goal\} \times \{m\}))) \wedge \\
\quad (p? \notin \operatorname{dom} m.beliefChangeTrace \Rightarrow \\
\quad (\mathbf{let}\ newTrace == \langle (value?, time?) \rangle \bullet \\
\quad\quad m.beliefChangeTrace = m.beliefChangeTrace \cup \\
\quad\quad\quad (\{predicate?\} \times \{newTrace\}) \wedge \\
\quad\quad goalMetadata' = goalMetadata \cup (\{m.goal\} \times \{m\})))))
\end{array}
$$

AddSubgoal

$$
\begin{array}{l}
\Delta Agent \\
predicate? : PREDICATE \\
type? : TYPE \\
parent? : Goal \\
\hline
parent? \in goals \\
\\
\forall goal : Goal \mid goal \in goals \bullet goal.predicate \neq predicate? \vee \\
\quad (goal.predicate = predicate? \wedge goal.type \neq type?) \\
\\
(\mathbf{let}\ goal == (\mu\, g : Goal \mid g.predicate = predicate? \wedge \\
\quad g.type = type? \wedge g.endState = Nil) \bullet goals' = goals \cup \{goal\} \wedge \\
\quad\quad parentGoal' = parentGoal \cup (\{goal\} \times \{parent?\})
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{IncreasePlanCounter} \\
\Delta \mathit{Agent} \\
\mathit{planInstance?} : \mathit{PlanInstance} \\
\hline
\forall g : \mathit{Goal} \mid g \in \mathit{goals} \land g.\mathit{endState} = \mathit{Achieved} \bullet \\
\quad (\mathbf{let}\ \mathit{metadata} == \mathit{goalMetadata}(g) \bullet \\
\qquad \mathit{metadata.planCounter} = \mathit{succ}\, \mathit{metadata.planCounter} \land \\
\qquad \mathit{goalMetadata}' = \mathit{goalmetadata} \cup (\{g\} \times \{\mathit{metadata}\})) \\
\hline
\end{array}
$$

6.3.3 Reversion Execution

The previously described activities provide the infrastructure needed for the reversion process to take place. Goals are created with metadata, which have the information needed to revert actions updated by the monitoring activity. Now, we describe how this information is used to revert goals or discard goal metadata, when the reversion is not possible anymore.

6.3.3.1 Reversion (De)activation

As said, goal metadata are kept for a limited amount of time. This prevents goals that occurred at a distant point in time to be reverted, because the current agent state may be too different for making the reversion reasonable. Moreover, we assume that agents have limited memory size, thus creating this need for preventing metadata to be stored and never discarded. There are two alternatives to constrain the amount of time goal metadata are kept stored. The first is the number of plan executions because each plan execution can potentially lead to changes in the belief base. The second criterion is the time elapsed since the goal was achieved. At least one of them must be specified and, if both are informed, the first one to be satisfied leads the goal metadata to be discard. Although there may be uncertainty regarding the specification of an adequate discarding condition, it is unrealistic to expect that metadata would be kept for an unlimited time.

We define next the evaluation of whether the reversion process of a goal must be deactivated. This is done by considering the discarding conditions of the goal metadata and the counter maintained by the monitoring activity. The reversion must be deactivated, i.e. *isReversionDeactivated* is true, when either the maximum number of plan executions is reached or the maximum time elapsed.

$$
\begin{array}{|l}
isReversionDeactivated : (GoalMetadata \times \mathbb{N}) \longrightarrow BOOLEAN \\
\hline
\forall m : GoalMetadata; time : \mathbb{N} \bullet \\
\quad (isReversionDeactivated\,(m, time) = True \Rightarrow \\
\quad\quad (m.planCounter > m.maxExecutedPlans \lor \\
\quad\quad (time - m.achievedTime) > m.maxTime)) \land \\
\quad (isReversionDeactivated\,(m, time) = False \Rightarrow \\
\quad\quad (m.planCounter \leq m.maxExecutedPlans \land \\
\quad\quad (time - m.achievedTime) \leq m.maxTime))
\end{array}
$$

When a goal reversion must be deactivated, the goal metadata must be expired, that is, removed from the agent. This removal is done by the *ExpireGoalMetadata* operation, which is shown below.

$$
\begin{array}{|l}
\hline
ExpireGoalMetadata \\
\Delta Agent \\
goal? : Goal \\
time? : \mathbb{N} \\
\hline
goal? \in goals \\
goal? \in \operatorname{dom} goalMetadata \\
\\
isReversionDeactivated(goalMetadata\, goal?, time?) = True \Rightarrow \\
\quad goalMetadata' = \{goal?\} \lhd goalMetadata \\
\hline
\end{array}
$$

Finally, we specify when the reversion is activated. There are two possibilities for this. First, if a plan failed during its execution to achieve the goal (indicated by $goal.endState = Failed$) and the rollback variable in the goal metadata is true, a reversion process must occur to revert the (partial) set of changes that were performed when attempting to achieve this goal. Second, if a goal was already achieved, the reversion should occur when the reversion trigger in the goal metadata holds considering the current context. This is given by the *isReversionActivated* function detailed as follows.

$$isReversionActivated : (GoalMetadata \times BeliefBase) \longrightarrow BOOLEAN$$

$$\begin{aligned}
&\forall\, m : GoalMetadata;\ bb : BeliefBase \bullet\\
&\quad (isReversionActivated\,(m, bb) = True \Rightarrow\\
&\qquad (m.reversionTrigger \in bb.knowledge \wedge\\
&\qquad\quad bb.belief(m.reversionTrigger) = True) \vee\\
&\qquad (m.rollback = True \wedge m.goal.endState = Failed)) \wedge\\
&\quad (isReversionActivated\,(m, bb) = False \Rightarrow\\
&\qquad (m.reversionTrigger \notin bb.knowledge \vee\\
&\qquad\quad (m.reversionTrigger \in bb.knowledge \wedge\\
&\qquad\quad bb.belief(m.reversionTrigger) = false)) \wedge\\
&\qquad (m.rollback = False \vee\\
&\qquad\quad (m.rollback = True \wedge m.goal.endState \neq Failed)))
\end{aligned}$$

6.3.3.2 Effect Filtering

Before reverting actions associated with a goal, we must select the changes that must be actually reverted. As explained in Section 6.1, some of the belief changes must not be undone. This is particularly important in the context of remedial actions, in which the effects of these actions must be persisted. Although in this thesis we focus on such context, our proposal (and consequently its implementation) is still open for customisation, allowing the specification of different parameters for effect filtering.

Therefore, the effect filtering makes the selection of the changes that must be reverted through the *filterBeliefChanges* shown below. Two types of changes are discarded. First, only actual changes in beliefs are considered. Multiple changes in a belief are irrelevant, because what is important is if the value of the belief before the plan execution is different from that after the execution. Second, we assume that what should be reverted are the additional actions made to achieve a goal and not the achieved goal itself, which would be the case of achieving the negation of the original goal. Therefore, we also discard changes in the belief associated with the achieved goal.

$$filterBeliefChanges : GoalMetadata \times BeliefBase \longrightarrow \mathbb{P}\ PREDICATE$$

$$\begin{aligned}
&\forall m : GoalMetadata; bb : BeliefBase \bullet \\
&\quad filterBeliefChanges\,(m, bb) = (\mu\, rev : \mathbb{P}\ PREDICATE \mid \\
&\quad (\forall p : PREDICATE \mid p \in rev \bullet \\
&(\exists pred : PREDICATE \mid pred \in bb.knowledge \bullet pred = p) \land \\
&\qquad (bb.beliefs(m.goal.predicate) = True \Rightarrow p \neq m.goal.predicate) \land \\
&\qquad (bb.beliefs(m.goal.predicate) = False \Rightarrow p \neq \neg m.goal.predicate) \land \\
&\qquad p \in \operatorname{dom} m.beliefChangeTrace \land \\
&\qquad bb.beliefs(p) = first(last(m.beliefChangeTrace(p))) \land \\
&\qquad first(last(m.beliefChangeTrace(p))) = \\
&\qquad\quad first(head(m.beliefChangeTrace(p))))
\end{aligned}$$

6.3.3.3 Effect Compensation

Now we know which belief changes must be reverted. Hence, to complete the reversion process, these changes—i.e. the effects of the achievement of a goal or partial changes made while executing a plan that failed—must be compensated. In order to do so, we rely on the BDI reasoning cycle to achieve a set of generated goals. Reversion goals correspond to the negation of the belief values associated with predicates changed while achieving a goal. This is done by the *reversionGoals* function below. These reversion goals are achievement goals.

$$reversionGoals : \mathbb{P}\ PREDICATE \longrightarrow \mathbb{P}\ Goal$$

$$\begin{aligned}
&\forall preds : \mathbb{P}\ PREDICATE \bullet \\
&\quad reversionGoals\,(preds) = (\mu\, goals : \mathbb{P}\ Goal \mid \\
&\qquad (\forall pred : PREDICATE \mid pred \in preds \bullet \\
&\qquad\quad (\exists goal : Goal \mid goal \in goals \bullet \\
&\qquad\qquad goal.predicate = \neg pred) \land \\
&\qquad\quad goal.type = Achievement \land \\
&\qquad\quad goal.endState = Nil))
\end{aligned}$$

Last, the *GenerateReversionGoals* operation thus adds goals to the agent to revert belief changes. Goals are created only for belief changes that must be reverted. Therefore, the *reversionGoals* function receives as parameter only belief changes that are not discarded after the belief change trace is filtered. We assume that goals to revert a goal are not reversible goals, in order to avoid a do-undo-redo cycle. Consequently, no metadata is associated with these goals. We highlight that a limitation of our approach is that it is dependent on how the agent knowledge is represented. For example, the action of sending a message to make an invitation can lead to different beliefs, e.g. *invited* and *msgSent*. An agent can have a plan to undo the former, but not the latter. Therefore, if *invited* is not modelled while developing the agent, it is not possible to revert this action.

$$
\begin{array}{l}
\textit{GenerateReversionGoals} \\
\hline
\Delta \textit{Agent} \\
m? : \textit{GoalMetadata} \\
\hline
m? \in \operatorname{ran} \textit{goalMetadata} \\
\textit{isReversionActivated}\,(m?, \textit{beliefBase}) = \textit{True} \\
\\
(\mathbf{let}\ \textit{preds} == \textit{filterBeliefChanges}\,(m?, \textit{beliefBase}) \bullet \\
\qquad \textit{goals}' = \textit{goals} \cup \textit{reversionGoals}\,(\textit{preds}) \wedge \\
\qquad m?.\textit{beliefChangeTrace} = \varnothing \wedge \\
\qquad \textit{goalMetadata}' = \textit{goalMetadata} \cup (\{m?.\textit{goal}\} \times \{m?\})) \\
\hline
\end{array}
$$

After adding goals to the agent, the trace of belief changes is cleared. This is done because the reversion process may occur to revert actions made by a plan that failed. Therefore, when a new plan is instantiated to achieve the goal, which remains as a goal (unless it is no longer desired), new belief changes are recorded, which can also be reverted. This completes the description of our framework.

6.4 Evaluation

Having described our formal framework, we now validate it with a case study. We first provide details of how we implemented our approach followed by a description of how we modelled our case study, which is based on the scenario described in Section 6.1. Finally, we present and discuss obtained results.

6.4.1 BDI4JADE Implementation

As in previous chapters, our framework was implemented using the BDI4JADE platform (NUNES; LUCENA; LUCK, 2011). In our implementation, we overloaded the `addGoal()` method of the `BDIAgent` class in order to allow the addition of reversible goals. Such method not only adds goals to the set of agent goals, but also creates the corresponding goal metadata. The goal metadata concept is captured by the `GoalAchievementMetadata` class. In order to not modify the core of BDI4JADE, we implemented the reasoning regarding reverting actions in a capability named `RevertingCapability`. Several other existing classes were extended to support the different operations specified in our solution, such as the `BeliefEvent` and `BeliefSet` classes, which are used for tracking changes performed by plans when achieving reversible goals. The monitoring process occurs by placing observers (provided by BDI4JADE) in the reasoning cycle to record information. The remaining parts are implemented in a customised option generation function.

As the focus of our proposal is to use the mechanism of reverting actions together with the automated management of remedial actions, which presented in previous chapters, we extended its implementation. However, they were kept modularised. Our previous implementation provided agents with the knowledge regarding the relationship between causes and effects of problems, which allow them to diagnose and solve problem causes while diagnosing their effects. This information is used to model the triggering conditions to revert goals.

6.4.2 Case Study Description

Our motivating scenario presented in Section 6.1 involves several devices that are managed by autonomous agents. These agents coordinate their actions in order to address a leaking of CO on the water heater of a house, thus preventing injuries that could be caused to residents by the exposure to high concentrations of CO. We implemented this scenario using the implementation described above.

In our case study, each device is managed by an individual agent that is able to control its range of capabilities. In addition to these agents, there is the `Manager` agent, which is responsible for broader decisions and for coordinating the interaction among agents associated with devices. It is the agent responsible for managing remedial actions and their reversion. The `Manager` agent is responsible for, e.g., requesting the `Lights Controller` agent to turn on the lights of a room when agents associated with light and presence sensors inform the `Manager` agent that it is dark and an individual entered the room. Each device controlling agent is provided with plans that allow them to respond to requests from the `Manager` agent. Agents responsible for controlling sensors, such as the `CO Detector Controller` agent, are able to perform additional tasks, such as monitoring their sensors and notifying the `Manager` agent when abnormal situations occur. Figure 6.2 depicts the set of agents from our study as well as their dependencies. Table 6.1 presents their corresponding plans.

Figure 6.2: Multiagent system architecture.

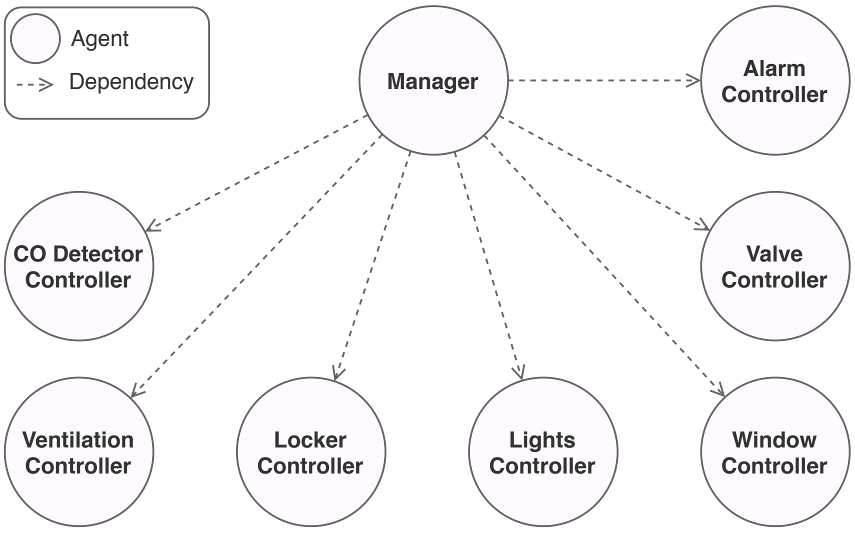


Table 6.1: Device Controlling Agents and their Plans.

Agent	**Plans**
`CO Detector Controller`	`MonitorCOLevel`
`Alarm Controller`	`TakeOffAlarm`; `SilenceAlarm`
`Lights Controller`	`TurnLightsOn`; `TurnLightsOff`
`Locker Controller`	`UnlockDoor`; `LockDoor`
`Window Controller`	`OpenWindows`; `CloseWindows`
`Ventilation Controller`	`TurnFansOn`, `TurnFansOff`
`Valve Controller`	`OpenValve`; `CloseValve`

The `Manager` agent has a plan library to address problems in the house as well as request other agents to accomplish certain actions. A possible problem is to have an increased concentration of CO and, if it happens, the agent generates a goal to reduce the CO level. Therefore, the `Manager` agent has a belief *abnormal(CO)* that must always be false. If *abnormal(CO)* becomes true, it may be an effect of several causes, being a leak on the water heater (*leak(waterHeater)*) one of them. This cause-effect relationship is provided in advance to the `Manager` agent.

Our case study thus consists of the implementation of all involved agents located in a simulated environment. We then observe the `Manager` agent behaviour when the concentration of CO in the house increases due to a leak in the water heater. To determine how the CO concentration changes according to the context, we use a simplified model of how this gas accumulates and dissipates in the presence of environmental conditions, such as open windows.

6.4.3 Results and Discussion

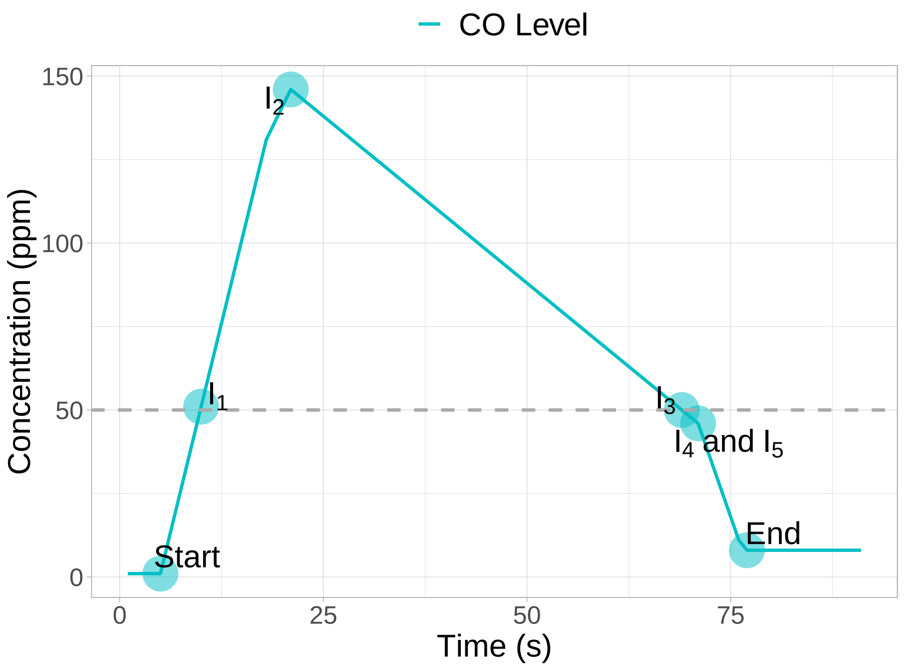
The results obtained by running our simulation is now described with a focus on the behaviour that was observed from `Manager` agent. Initially, the concentration of CO was at an acceptable level, below a given threshold. The `Manager` agent had a set of beliefs, corresponding to this CO level, as shown in the column and point *Start* of Table 6.2 and Figure 6.3, respectively. At some point, the water heater started to leak and the concentration of CO increased, as shown in Figure 6.3.

Table 6.2: `Manager` Agent State Evolution.

Belief	**State**						
	Start	I_1	I_2	I_3	I_4	I_5	*End*
$abnormal(CO)$	F	**T**	T	**F**	F	F	F
$takeOff(alarm)$	F	F	**T**	T	T	T	**F**
$on(lights)$	F	F	**T**	T	T	T	**F**
$locked(doors)$	T	T	**F**	F	F	F	**T**
$open(windows)$	F	F	**T**	T	T	T	**F**
$on(fans)$	F	F	**T**	T	T	T	**F**
$open(valve)$	T	T	T	T	T	**F**	F
$leak(waterHeater)$	-	-	-	-	**T**	**F**	F

When the amount of CO surpassed the tolerated concentration, the `CO Detector Controller` agent detected it and notified the `Manager` agent. By receiving this notification, the *abnormal(CO)* belief became true, as shown in column I_1, which corresponds to the first intermediate state of the agent. This triggered the generation of a goal to achieve $\neg abnormal(CO)$, which had a reverting trigger condition $leak(waterHeater)$ and a discarding condition of 60 minutes. Because no problem cause was known, preconditions to permanently solve the cause of high CO level did not hold, so the `Evacu-`

Figure 6.3: Level of CO over Time.



`ateAndVentilate` plan—a remedial plan—was selected to achieve the existing goal. This plan involves dispatching a series of subgoals that are handled by the corresponding requesting plans, as shown in Table 6.3. All actions were recorded as goal metadata. The new state of `Manager` agent is then I_2. Because of the remedial actions, the concentration of CO decreased slowly. When the amount of gas returned to an acceptable level, the `CO Detector Controller` agent detected it and notified the `Manager` agent (point I_3). The `Manager` agent updated its belief base (state I_3) leading to the successful execution of the remedial plan.

Table 6.3: `Manager` Agent: Remediation Goals and Plans.

Goals	**Plans**
$\neg abnormal(CO)$	`EvacuateAndVentilate`
$takeOff(alarm)$	`RequestAlarmTakeOff`
$on(lights)$	`RequestLightsOn`
$\neg locked(doors)$	`RequestUnlockDoors`
$open(windows)$	`RequestOpenWindows`
$on(fans)$	`RequestFansOn`

The cause of the problem is diagnosed with according to the technique introduced in Chapter 4, leading to the state I_4, in which the belief $\langle leak(waterHeater); True\rangle$ was added to the agent belief base, which caused a goal $\langle leak(waterHeater); False\rangle$ to be created. Differently from the previous goal, this one does not have trigger and discarding conditions. The `RequestCloseValve` plan was then executed, the goal was achieved and the evaluation value of *leak(waterHeater)* was set to false (state I_5). Figure 6.3 highlights this moment as well in point I_5.

The existence of $\langle leak(waterHeater); False \rangle$ in the agent belief base satisfied the trigger condition of the goal $\neg abnormal(CO)$, which triggered the filtering and reversing steps of our approach. The belief changes recorded in the goal metadata were thus filtered and a reversion goal was generated to each of them. Requesting plans were used once again, and after their execution, the end state of `Manager` agent was as shown in the last column of Table 6.2.

From the execution of this simulation, it is possible to observe that the behaviour presented by `Manager` agent corresponded to what was expected. The comparison of the *Start* and *End* states presented in Table 6.2 shows that the agent was able to *autonomously* return the system to a desired state even after addressing a challenging problem. Although the amount of CO registered after the problem being solved was different, it remained at an acceptable level, below the specified threshold.

6.5 Final Remarks

We presented in this chapter an approach to allow multiagent systems to autonomously manage the process of recovering to a normal operating state. We proposed a formal extension of the BDI architecture that includes the structures and operations needed to manage this process. These operations were grouped into three main activities, namely goal setup, monitoring and reversion execution. The last has three sub-activities in which (i) an evaluation of the reversion triggering conditions is performed; (ii) recorded changes that should not be reverted are discarded; and (iii) goals to revert changes are created. Our formal approach was implemented as an extension of the BDI4JADE platform. This extension was used to evaluate our proposal with a case study in which we simulated a gas leaking scenario in a smart home. As result, a MAS performed remedial actions to reduce the levels of gas in a timely fashion before diagnosing the cause of the problem. These actions were them autonomously reverted due to our proposed solution that was added to agents as a capability. This case study demonstrated the effectiveness of our approach, which can be used in many reversible circumstances. We next conclude this thesis, summarising its main contributions and pointing out directions for future work.

7 CONCLUSION

Many systems are nowadays built as multiagent systems. Due to the complex tasks undertaken by these systems, they are expected to resist and recover from adverse situations that may compromise their operation and the quality of their services. An existing strategy to provide software systems with this resilient behaviour, named D^2R^2 + DR (STERBENZ et al., 2010), specifies the execution of operations such as the detection of problems, their remediation, the diagnosis of their causes, and the recovery of the system to a normal operating state. Nevertheless, because of its abstract nature, systems that instantiate this strategy do it for specific purposes. As a consequence, instantiations of the D^2R^2 + DR strategy cannot be reused across different application domains. Even though there are approaches that provide concrete solutions for some of these operations, they present several limitations including a lack of autonomy and adaptability.

In this thesis, we presented a framework that aims at providing multiagent systems with remediation, diagnosis and recovery capabilities. We take advantage of the operations specified by the D^2R^2 + DR strategy to allow these systems to mitigate the effects of problematic events while diagnosing and solving their causes. This framework detaches this remedial behaviour from domain-dependent code, thus promoting software reuse across different applications. It extends the typical BDI architecture and comprises three main techniques. The first automates the management of remedial actions as well as the diagnosis and solution of problem causes. It includes a set of components to capture the domain knowledge that supports agents on carrying out these operations. The second is focused on the diagnosis of problem causes in MAS scenarios. It specifies an interaction protocol and roles that describe how agents can coordinate their actions and share information in order to keep operating with the expected quality. Finally, the third formalises an approach for reverting BDI agent actions. This technique allows agents to revoke the effects of executed actions without the need for an explicit declaration of which these actions are and how they should be reverted. This technique is focused on reverting remedial actions performed as outcomes of a resilient behaviour. It can, however, be adopted in any reversible circumstance.

All these techniques were implemented as an extension of a platform for developing BDI agents. This implementation served as basis for conducting empirical studies with the aim of assessing different aspects of the proposed framework. Regarding our research question, results showed evidence that our framework is able to provide agents and multiagent systems with the abilities required to carry out the remediate, diagnose and recover operations specified by the D^2R^2 + DR strategy. With respect to our hypothesis, we conclude that our proposed solution can serve as the core implementation for agents and MAS with remedial behaviour in different domains. Due to its modularised architecture, domain-specific capabilities can be incorporated without hindering the underlying behaviour. Finally, the autonomy and adaptability inherited from the BDI model of agency were demonstrated in our experiments.

7.1 Contributions

Many contributions can be enumerated as a result of the work presented in this thesis. Together, they frame our solution for the automated management of remedial behaviour.

Technique for Automated Management of Remedial Actions. The technique described in Chapter 4 is responsible for automating the coordination of actions performed by agents during the execution of the remediate and diagnose operations. This technique comprises structural and behavioural parts (FACCIN; NUNES, 2017a; FACCIN; NUNES, 2017b; FACCIN; NUNES, 2018b). The former introduces the concepts of constrained goals, plan required resources and cause-effect relationships to the BDI architecture. These concepts are manipulated by customised implementations of the plan selection and option generation functions, which are part of the BDI reasoning cycle and are specified by the latter. Because this technique abstracts the causal information used during the diagnosis process into a structure named cause-effect knowledge model, domain-dependent information becomes decoupled from the general remedial reasoning. As a result, our framework can be reused in many domains, requiring only the specification of the corresponding cause-effect knowledge model to operate. This model can be specified manually, at design time, or dynamically, during system execution. It enables the development of domain-specific strategies for autonomously collecting and maintaining causal information at runtime.

Technique for Cooperative Diagnosis of Problem Causes. The technique introduced in Chapter 5 provides agents with the ability to cooperate in order to collect the information used to diagnose the root cause of problems (FACCIN; NUNES; HAMOU-

LHADJ, 2020) in a MAS. This technique comprises an interaction protocol and algorithms that specify the behaviour of agents while playing the roles of service providers, clients and cooperating components. The protocol describes the message exchange that occurs among agents when consuming and providing services, as well as how they collaborate when quality requirement violations are identified. Provided algorithms describe how the information exchanged by agents is used to identify if an abnormal behaviour has internal or external causes. Because of the technique's top-down approach, a degraded service can have its quality level restored while the cause of the problem is dynamically identified and solved. By adopting this technique, a MAS is thus able to resist and adapt to the occurrence of failures at different layers of the system.

Technique for Reverting Actions. In Chapter 6, we formalised a technique that allows agents to undo the effects of remedial actions after the complete solution of the problems for which they were executed (FACCIN; NUNES, 2018a). This technique introduces the concept of goal metadata and specifies the goal setup, monitoring and reversion execution activities. These activities, which are executed within the BDI reasoning cycle, enables the tracing of action effects as well as the triggering of the reversion process when particular conditions are met. The ability provided by this technique is particularly suited for scenarios in which valuable resources are allocated for remediating a problem but must be released as soon as possible in order to allow the execution of further actions. Nevertheless, the proposed solution is general enough to be used in scenarios such as failure handling and task aborting situations.

Development Framework. The techniques described in this thesis were implemented as a framework (FACCIN; NUNES, 2018b; FACCIN; NUNES, 2018a; FACCIN; NUNES; HAMOU-LHADJ, 2020) that extends BDI4JADE (NUNES; LUCENA; LUCK, 2011), an existing Java-based platform that implements the BDI architecture. Our techniques are encapsulated into capabilities. Because of that, the can either be used for developing new agents or incorporated to existing ones, requiring only to be instantiated in order to be used.

Empirical Evaluations. Three empirical studies were conducted to evaluate different aspects of the proposed framework (FACCIN; NUNES, 2018b; FACCIN; NUNES, 2018a; FACCIN; NUNES; HAMOU-LHADJ, 2020). Our first study was focused on evaluating if our technique for the automated coordination of agent actions would be effective on replicate the remedial behaviour of an existing strategy for combating DDoS attacks, which was developed in a domain-specific way. Results showed that our solution not only is effective for developing agents and multiagent systems that implement the remediation, diagnosis and recovery steps specified by the D^2R^2 + DR strategy, but does it without reducing system performance. The second study focused on evaluating a service-based MAS that adopts our cooperative technique for diagnosing problem causes regarding its ability to dynamically identify and solve the root cause of problems. Results showed that our solution provides MAS with the ability to adapt to disrupting events without the need for human intervention. Finally, our third study was performed in a smart-home setup and evaluated the ability of a MAS implementing our framework to autonomously recover to a desired operating state after adverse conditions were remediated and solved. By conducting these studies, we provide evidence of the reusability of the developed framework across several applications in different domains.

7.2 Future Work

The contributions presented in this thesis advance research on the development of resilient MAS systems. However, there still remains several open challenges in this context that should be addressed in future work. These challenges are discussed as follows.

Management Technique Enhancement. Our technique for managing the remedial behaviour, presented in Chapter 4, leaves gaps that can be fulfilled to provide further automation. For instance, satisfying all goal constraints may be unfeasible in many scenarios. However achieving the goal may be more important than satisfying all constraints. The development of solutions to deal with over-constrained scenarios would increase the adaptability of systems, specially in situations that require actions to be taken quickly. Moreover, goals associated with cause factors may also be constrained goals, and there must be ways to specify constraints and optimisation function for them.

Domain-specific Techniques for Diagnosing Causes. The cause-effect knowledge model introduced in Chapter 4 to capture domain-dependent causal information can be provided either manually, at design time, or dynamically, at runtime. The technique presented in Chapter 5 relies on particular characteristics of MAS in order to collect this information from the system execution and be able to diagnose the root cause of service degradation. However, these characteristics may not apply to applications that are not modelled as service-based systems. Our general framework would thus benefit from the development of additional techniques that take advantage of particularities of different domains to dynamically build cause-effect knowledge models.

User Study. The evaluations conducted in Chapters 4–6 demonstrated that the framework presented in this thesis has the potential to be reused in different application domains. Nevertheless, a user study that assesses the benefits regarding its reuse from the perspective of developers would provide valuable information with respect to its strengths and weaknesses in order to be used in real-world software systems.

Incorporate Additional D^2R^2 + DR Operations. The framework proposed in this thesis automates the remediation, diagnosis and recovery operations specified by the D^2R^2 + DR. As presented in Chapter 2, this strategy also describes additional operations, such as detection of problems and the refinement of system operation based on past experiences. In this context, the development of techniques that allows these operations to be incorporated into our framework becomes highly desirable.

Manual and Tutorials. The implementation of the framework presented in this thesis is available as an extension of the BDI4JADE platform[1]. Manuals and tutorials that document our framework and detail its use would certainly promote its adoption in different projects, and thus become a valuable resource.

In summary, this thesis advances research on reducing the gap between the abstract definition of a remedial behaviour and its implementation in agents and MAS. Much more work is required in order to develop a general solution that provides software components with resilience, but our work is one of the steps towards this direction.

[1]<https://www.inf.ufrgs.br/prosoft/resources/2020/tse-mas-self-adaptive-protocol>

REFERENCES

ABREU, D. P. et al. A resilient internet of things architecture for smart cities. **Annals of Telecommunications**, v. 72, n. 1, p. 19–30, Feb 2017.

AGENTS, F. for I. P. **FIPA Contract Net Interaction Protocol Specification**. [S.l.], 2002. Available from Internet: <http://www.fipa.org/specs/fipa00029/SC00029H.html>. Accessed in: 2020-08-10.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, IEEE, v. 1, n. 1, p. 11–33, 2004.

BARESI, L.; NITTO, E. D.; GHEZZI, C. Toward open-world software: Issues and challenges. **Computer**, v. 39, n. 10, p. 36–43, 2006.

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

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

BUTLER, M.; FERREIRA, C. A process compensation language. In: ____. **Integrated Formal Methods**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. p. 61–76. ISBN 978-3-540-40911-3.

CARVALHO, L. F. et al. An ecosystem for anomaly detection and mitigation in software-defined networking. **Expert Systems with Applications**, v. 104, p. 121–133, 2018. Available from Internet: <https://doi.org/10.1016/j.eswa.2018.03.027>. Accessed in: 2020-08-10.

CHANDOLA, V.; BANERJEE, A.; KUMAR, V. Anomaly detection: A survey. **ACM Computing Surveys**, Association for Computing Machinery, New York, NY, USA, v. 41, n. 3, jul. 2009. Available from Internet: <https://doi.org/10.1145/1541880.1541882>. Accessed in: 2020-08-10.

CHEN, T.; BAHSOON, R. Self-adaptive and online QoS modeling for cloud-based software services. **IEEE Transactions on Software Engineering**, v. 43, n. 5, p. 453–475, 2017. Available from Internet: <https://doi.org/10.1109/TSE.2016.2608826>. Accessed in: 2020-08-10.

CHESSELL, M. et al. Extending the concept of transaction compensation. 2002.

CREVELING, C. J. Increasing the reliability of electronic equipment by the use of redundant circuits. **Proceedings of the IRE**, v. 44, n. 4, p. 509–515, 1956.

CZARNECKI, K.; EISENECKER, U. **Generative Programming: Methods, Tools, and Applications**. [S.l.]: Addison Wesley Longman, 2000.

DENNETT, D. **The Intentional Stance**. [S.l.]: MIT Press, 1987.

D'INVERNO, M.; LUCK, M. **Understanding agent systems**. [S.l.]: Springer Science & Business Media, 2004. ISBN 9783540407003.

DOBSON, S. et al. A survey of autonomic communications. **ACM Transactions on Autonomous and Adaptive Systems**, Association for Computing Machinery, New York, NY, USA, v. 1, n. 2, p. 223–259, 2006. Available from Internet: <https://doi.org/10.1145/1186778.1186782>. Accessed in: 2020-08-10.

DOBSON, S. et al. Self-organization and resilience for networked systems: Design principles and open research issues. **Proceedings of the IEEE**, v. 107, n. 4, p. 819–834, 2019. Available from Internet: <https://doi.org/10.1109/JPROC.2019.2894512>. Accessed in: 2020-08-10.

DÖTTERL, J. et al. Towards dynamic rebalancing of bike sharing systems: An event-driven agents approach. In: OLIVEIRA, E. et al. (Ed.). **Proceedings...** Cham: Springer International Publishing, 2017. p. 309–320.

ELLISON, R. et al. Foundations for survivable systems engineering. **The Journal of Defense Software Engineering**, p. 10–15, 2002.

FACCIN, J.; NUNES, I. BDI-agent plan selection based on prediction of plan outcomes. In: INTERNATIONAL CONFERENCE ON WEB INTELLIGENCE AND INTELLIGENT AGENT TECHNOLOGY. **Proceedings...** 2015. v. 2, p. 166–173. Available from Internet: <http://dx.doi.org/10.1109/WI-IAT.2015.58>. Accessed in: 2020-08-10.

FACCIN, J.; NUNES, I. Modelling and reasoning about remediation actions in BDI agents. In: CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS. **Proceedings...** [S.l.], 2017. p. 1526–1528.

FACCIN, J.; NUNES, I. Raciocínio causal em agentes BDI: um modelo abstrato. In: WORKSHOP-SCHOOL ON AGENTS, ENVIRONMENTS AND APPLICATIONS. **Proceedings...** [S.l.], 2017. p. 211–216.

FACCIN, J.; NUNES, I. Cleaning up the mess: A formal framework for autonomously reverting bdi agent actions. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING FOR ADAPTIVE AND SELF-MANAGING SYSTEMS. **Proceedings...** New York, NY, USA: Association for Computing Machinery, 2018. p. 108–118. ISBN 9781450357159. Available from Internet: <https://doi.org/10.1145/3194133.3194156>. Accessed in: 2020-08-10.

FACCIN, J.; NUNES, I. Remediating critical cause-effect situations with an extended BDI architecture. **Expert Systems with Applications**, v. 95, p. 190–200, 2018.

FACCIN, J.; NUNES, I.; HAMOU-LHADJ, A. A problem-solving strategy for self-adaptation in multiagent systems. **IEEE Transactions on Software Engineering**, 2020. Submitted.

FISHER, D. et al. **Survivable Network Systems: An Emerging Discipline**. Pittsburgh, PA, 1997. Available from Internet: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12905>. Accessed in: 2020-08-10.

FORD, A. et al. **TCP extensions for multipath operation with multiple addresses**. [S.l.], 2013. No. RFC 6824.

FRAKES, W. B.; KANG, K. Software reuse research: status and future. **IEEE Transactions on Software Engineering**, v. 31, n. 7, p. 529–536, 2005.

GHOSH, D. et al. Self-healing systems — survey and synthesis. **Decision Support Systems**, v. 42, n. 4, p. 2164–2185, 2007. Available from Internet: <https://doi.org/10.1016/j.dss.2006.06.011>. Accessed in: 2020-08-10.

GRAY, J.; REUTER, A. **Transaction Processing: Concepts and Techniques**. [S.l.]: Elsevier Science, 1992. (The Morgan Kaufmann Series in Data Management Systems). ISBN 9780080519555.

HAN, W.; LEI, C. A survey on policy languages in network and security management. **Computer Networks**, Elsevier, v. 56, n. 1, p. 477–489, 2012.

HERNANDEZ, L. et al. A multi-agent system architecture for smart grid management and forecasting of energy demand in virtual power plants. **IEEE Communications Magazine**, v. 51, n. 1, p. 106–113, 2013.

HOSSEINI, S.; BARKER, K.; RAMIREZ-MARQUEZ, J. E. A review of definitions and measures of system resilience. **Reliability Engineering & System Safety**, Elsevier, v. 145, p. 47–61, 2016.

HUANG, A.; NITSCHKE, G. Automating coordinated autonomous vehicle control. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS. **Proceedings...** Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2020. p. 1867–1868. ISBN 9781450375184.

HÜBNER, J. F.; BORDINI, R. H.; WOOLDRIDGE, M. Programming declarative goals using plan patterns. In: ____. **Declarative Agent Languages and Technologies IV: 4th International Workshop, DALT 2006, Hakodate, Japan, May 8, 2006, Selected, Revised and Invited Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 123–140. ISBN 978-3-540-68961-4.

INSAURRALDE, C. C. Service-oriented agent architecture for unmanned air vehicles. In: IEEE/AIAA DIGITAL AVIONICS SYSTEMS CONFERENCE. **Proceedings...** [S.l.], 2014. p. 1–19.

IQBAL, S. et al. Application of intelligent agents in health-care. **Artificial Intelligence Review**, Springer, v. 46, n. 1, p. 83–112, 2016.

JANUáRIO, F.; CARDOSO, A.; GIL, P. A distributed multi-agent framework for resilience enhancement in cyber-physical systems. **IEEE Access**, v. 7, p. 31342–31357, 2019.

JENNINGS, N. R. An agent-based approach for building complex software systems. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 4, p. 35–41, abr. 2001. Available from Internet: <https://doi.org/10.1145/367211.367250>. Accessed in: 2020-08-10.

KANG, K. et al. **Feature-oriented domain analysis (FODA) feasibility study**. [S.l.], 1990.

KORTH, H. F.; LEVY, E.; SILBERSCHATZ, A. A formal approach to recovery by compensating transactions. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES. **Proceedings...** San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990. p. 95–106. ISBN 1-55860-149-X.

LYONS, R. E.; VANDERKULK, W. The use of triple-modular redundancy to improve computer reliability. **IBM Journal of Research and Development**, v. 6, n. 2, p. 200–209, 1962.

MAES, S.; MEGANCK, S.; MANDERICK, B. Inference in multi-agent causal models. **International Journal of Approximate Reasoning**, v. 46, n. 2, p. 274–299, 2007. Available from Internet: <https://doi.org/10.1016/j.ijar.2006.09.005>. Accessed in: 2020-08-10.

MENDONçA, D. F.; ALI, R.; RODRIGUES, G. N. Modelling and analysing contextual failures for dependability requirements. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE ENGINEERING FOR ADAPTIVE AND SELF-MANAGING SYSTEMS. **Proceedings...** New York, NY, USA: Association for Computing Machinery, 2014. p. 55–64. ISBN 9781450328647. Available from Internet: <https://doi.org/10.1145/2593929.2593947>. Accessed in: 2020-08-10.

MENG, W. Intrusion detection in the era of IoT: Building trust via traffic filtering and sampling. **Computer**, v. 51, n. 7, p. 36–43, 2018.

MOHAGHEGHI, P.; CONRADI, R. Quality, productivity and economic benefits of software reuse: A review of industrial studies. **Empirical Software Engineering**, Kluwer Academic Publishers, Hingham, MA, USA, v. 12, n. 5, p. 471–516, Oct 2007. Available from Internet: <http://dx.doi.org/10.1007/s10664-007-9040-x>. Accessed in: 2020-08-10.

NOLAN, K. E. et al. Techniques for resilient real-world IoT. In: INTERNATIONAL WIRELESS COMMUNICATION AND MOBILE COMPUTING CONFERENCE. **Proceedings...** [S.l.], 2016. p. 222–226.

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. p. 88–103.

NUNES, I.; SCHARDONG, F.; SCHAEFFER-FILHO, A. BDI2DoS: an application using collaborating BDI agents to combat DDoS attacks. **Journal of Network and Computer Applications**, 2017.

PADGHAM, L.; SINGH, D. Situational preferences for BDI plans. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS. **Proceedings...** [S.l.]: International Foundation for Autonomous Agents and Multiagent Systems, 2013. p. 1013–1020. ISBN 978-1-4503-1993-5.

PARIDA, P. K.; MARWALA, T.; CHAKRAVERTY, S. A multivariate additive noise model for complete causal discovery. **Neural Networks**, v. 103, p. 44–54, 2018. Available from Internet: <https://doi.org/10.1016/j.neunet.2018.03.013>. Accessed in: 2020-08-10.

PARZEN, E. On estimation of a probability density function and mode. **Annals of Mathematical Statistics**, The Institute of Mathematical Statistics, v. 33, n. 3, p. 1065–1076, 1962. Available from Internet: <https://doi.org/10.1214/aoms/1177704472>. Accessed in: 2020-08-10.

PEARL, J. **Probabilistic reasoning in intelligent systems: networks of plausible inference**. [S.l.]: Elsevier, 2014.

RAICIU, C. et al. Improving datacenter performance and robustness with multipath TCP. In: ACM SIGCOMM. **Proceedings...** New York, NY, USA: ACM, 2011. (SIGCOMM ’11), p. 266–277. ISBN 978-1-4503-0797-0.

RAO, A. S. AgentSpeak(L): BDI agents speak out in a logical computable language. In: EUROPEAN WORKSHOP ON MODELLING AUTONOMOUS AGENTS IN A MULTI-AGENT WORLD. **Proceedings...** Springer-Verlag New York, Inc., 1996. p. 42–55. ISBN 3-540-60852-4. Available from Internet: <http://dl.acm.org/citation.cfm?id=237945.237953>. Accessed in: 2020-08-10.

RAO, A. S.; GEORGEFF, M. P. BDI agents: From theory to practice. In: INTERNATIONAL CONFERENCE ON MULTIAGENT SYSTEMS. **Proceedings...** [S.l.], 1995. p. 312–319.

RAVICHANDRAN, T.; ROTHENBERGER, M. A. Software reuse strategies and component markets. **Communications of ACM**, Association for Computing Machinery, New York, NY, USA, v. 46, n. 8, p. 109–114, 2003. Available from Internet: <https://doi.org/10.1145/859670.859678>. Accessed in: 2020-08-10.

SANCTIS, M. D.; BUCCHIARONE, A.; MARCONI, A. Dynamic adaptation of service-based applications: a design for adaptation approach. **Journal of Internet Services and Applications**, v. 11, n. 2, 2020. Available from Internet: <https://doi.org/10.1186/s13174-020-00123-6>. Accessed in: 2020-08-10.

SCHAEFFER-FILHO, A. E. et al. A framework for the design and evaluation of network resilience management. In: IEEE/IFIP NETWORK OPERATIONS AND MANAGEMENT SYMPOSIUM. **Proceedings...** [S.l.]: IEEE, 2012. p. 401–408. ISBN 978-1-4673-0267-8.

SIRCAR, K. et al. Carbon monoxide poisoning deaths in the united states, 1999 to 2012. **The American Journal of Emergency Medicine**, v. 33, n. 9, p. 1140–1145, 2015.

SPIVEY, J. M. **Understanding Z: a specification language and its formal semantics**. [S.l.]: Cambridge University Press, 1988.

STERBENZ, J. P. et al. Evaluation of network resilience, survivability, and disruption tolerance: analysis, topology generation, simulation, and experimentation. **Telecommunication Systems**, Springer, v. 52, n. 2, p. 705–736, 2013.

STERBENZ, J. P. G. et al. Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines. **Computer Networks**, v. 54, n. 8, p. 1245–1265, 2010. Available from Internet: <https://doi.org/10.1016/j.comnet.2010.03.005>. Accessed in: 2020-08-10.

STRIGINI, L. Resilience: What is it, and how much do we want? **IEEE Security Privacy**, v. 10, n. 3, p. 72–75, May 2012.

THANGARAJAH, J. et al. Aborting tasks in BDI agents. In: INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS. **Proceedings...** New York, NY, USA: ACM, 2007. p. 8–15. ISBN 978-81-904262-7-5.

TUKEY, J. W. **Exploratory Data Analysis**. [S.l.]: Addison-Wesley, 1977.

UNRUH, A.; BAILEY, J.; RAMAMOHANARAO, K. A framework for goal-based semantic compensation in agent systems. In: INTERNATIONAL WORKSHOP ON SAFETY AND SECURITY IN MULTI-AGENT SYSTEMS. **Proceedings...** [S.l.], 2004.

UNRUH, A. et al. Semantic-compensation-based recovery in multi-agent systems. In: MULTI-AGENT SECURITY AND SURVIVABILITY. **Proceedings...** [S.l.], 2005. p. 85–94.

VISSER, S. et al. Preference-based reasoning in BDI agent systems. **Autonomous agents and multi-agent systems**, Springer, v. 30, n. 2, p. 291–330, 2016.

WANG, P. et al. Cloudranger: Root cause identification for cloud native systems. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER, CLOUD AND GRID COMPUTING. **Proceedings...** 2018. p. 492–502. Available from Internet: <https://doi.org/10.1109/CCGRID.2018.00076>. Accessed in: 2020-08-10.

WILSON, P. F. **Root cause analysis: A tool for total quality management**. [S.l.]: ASQ Quality Press, 1993.

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

ZHANG, Y.; LIN, K.-J.; HSU, J. Y. J. Accountability monitoring and reasoning in service-oriented architectures. **Service Oriented Computing and Applications**, v. 1, n. 1, p. 35–50, 2007. Available from Internet: <https://doi.org/10.1007/s11761-007-0001-4>. Accessed in: 2020-08-10.

ZHOU, X. et al. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. **IEEE Transactions on Software Engineering**, v. 14, n. 8, p. 1–18, 2018.

APPENDIX A — FRAMEWORK USAGE

The framework presented in this thesis is implemented as an extension of an existing Java-based platform for the development of BDI agents, named BDI4JADE[1]. The source code of our framework is available in a git repository[2] under a branch named `remediation-undo`. To have access to it, clone the repository and switch to the mentioned branch. This appendix describes how this extended platform can be used to provide agents with the remedial behaviour that is the focus of this work. A typical way to implement an agent in BDI4JADE is by creating an instance of the `SingleCapabilityAgent` class or a class that extends it. Listing A.1 presents an example in which an agent named `myAgent` is created.

Listing A.1: Agent instantiation.

```
public static void main(String[] args) {
   BDIAgent myAgent = new SingleCapabilityAgent();
}
```

As introduced in Chapter 3, a BDI agent comprises three key components: beliefs, desires (or goals), and intentions. There are also the plans, which represent the set of actions performed by the agent when trying to achieve a given goal. To implement an agent, beliefs, goals and plans are explicitly declared as parts of capabilities, while intentions remain as internal structures that are transparent to developers. Details on how to implement these concepts can be found elsewhere (NUNES; LUCENA; LUCK, 2011).

To take advantage of the remedial behaviour introduced in Chapter 4, developers are required to implement a capability that extends the `RemediationCapability`, which is provided by our framework. Such an implementation is exemplified in Listing A.2, where the `RemediationCapability` class is extended by a new capability class named `MyCapability` (line 1). Besides allowing the use of customised beliefs and plans, the `RemediationCapability` class also provides the `causeEffectKnowledgeModel` belief, which can be accessed and updated in order to provide a domain-specific cause-effect knowledge model. Listing A.2 presents its usage in lines 11–13, where a `CauseEffectRelationship cer` containing an effect and its mandatory cause is instantiated and added to the existing `causeEffectKnowledgeModel` belief.

[1]<https://www.inf.ufrgs.br/prosoft/bdi4jade/>
[2]<http://prosoft.inf.ufrgs.br/git/bdi4jade.git>

Listing A.2: Agent instantiation.

```
 1 public class MyCapability extends RemediationCapability {
 2   @Belief
 3   private String myBelief;
 4   @Plan
 5   private Plan myPlan;
 6 
 7   public MyCapability(BDIAgent agent) {
 8         super(agent);
 9     this.myBelief = Boolean.TRUE;
10     this.myPlan = new MyPlan();
11     CauseEffectRelationship cer = new CauseEffectRelationship(new Fact(
          ↪ new UnaryPredicate("effect"), true));
12     cer.addMandatoryCause(new Fact(new UnaryPredicate("cause"), true));
13     causeEffectKnowledgeModel.addCauseEffectRelationship(cer);
14   }
```

Constrained goals are created through the instantiation of the `Constrained-Goal` class or a class that extends it. Methods `addObjectiveFunction()` and `addOperationConstraint()` are respectively used to inform the objective function associated to resources that are related to a goal and to specify particular operation constraints. Listing A.3 exemplifies how a constrained goal `myGoal` is created and an objective function to minimise the use of a `time` resource as well as a constraint are added to it (lines 5–7).

Listing A.3: Constrained goal instantiation.

```
1 public static void main(String[] args) {
2     TimeResource time = new TimeResource();
3     UnaryLogicalExpression constraint = new UnaryLogicalExpression();
4     ...
5     ConstrainedGoal myGoal = new ConstrainedGoal(null);
6     myGoal.addObjectiveFunction(time, ObjectiveFunction.MINIMISE);
7     myGoal.addOperationConstraint(constraint);
8 }
```

A plan required resource can be specified by instantiating the `PlanRequire-dResource` class. It can be added to an existing plan using the `putMetadata()` method. Listing A.4 shows the instantiation of a plan required resource named `prr`, which states that 10 units of time are required by a given plan to be executed (lines 4–5). This plan required resource is thus added to `myPlan` in line 8.

Listing A.4: Plan Required Resource instantiation.

```
public static void main(String[] args) {
   TimeResource time = new TimeResource();
   ...
   PlanRequiredResource prr = new PlanRequiredResource();
   prr.setRequiredResource(time, 10.0);
   ...
   MyPlan myPlan = new MyPlan();
   myPlan.putMetadata(PlanRequiredResource.METADATA_NAME, prr);
}
```

Because the `RemedationCapability` extends the `RevertingCapability` class, there is no need for an explicit declaration of reversible goals and their corresponding goal achievement metadata, as these operations are already handled by the extended class. Goals are considered reversible when they play the role of effects in the existing cause-effect knowledge model. Their addition to the agent automatically triggers the instantiation of corresponding goal achievement metadata.

Regarding the interaction protocol and its underlying roles presented in Chapter 5, our framework provides a class named `CooperativeCapability`, which can serve as the basis for implementing this protocol. This capability already provides a set of plans and beliefs that allow agents to perform the reasoning process required for diagnosing the root cause of problems. Examples of plans include the `VerifyInternalOrExternalPlanBody` and `VerifySuspiciousComponentPlanBody` classes, which implement the internal and external verification operations, respectively. Nevertheless, this class can still be extended in order to handle different applications as its current implementation only gives support for the tracing of the time resource.

APPENDIX B — RESUMO ESTENDIDO

Gerenciamento Automatizado de Comportamento Remediativo

Muitos sistemas de software são atualmente construidos como sistemas multiagentes (MAS). Eles são compostos por componentes autônomos distribuídos, situados em um ambiente. Tais componentes, também chamados de *agentes*, interagem e colaboram entre si para realizar uma série de tarefas como, por exemplo, o gerenciamento de plantas de energia, veículos autônomos, veículos aéreos não tripulados e sistemas de cuidados médicos. A complexidade e criticidade dessas tarefas exigem que esses sistemas sejam capazes de operar satisfazendo os níveis de qualidade existentes pelo maior tempo possível, resistindo e se recuperando de situações anormais. Assim, a adoção de técnicas que os façam resilientes se torna essencial.

Resiliência pode ser entendida como a capacidade de um sistema em retornar a uma condição normal de operação após a ocorrência de eventos disruptivos. Ela combina ideias de diversas disciplinas, como tolerância à falhas e sobrevivência e, por isso, pode ser obtida em diferentes níveis e de diferentes maneiras. Uma estratégia de resiliência existente, chamada $D^2R^2 + DR$, especifica um conjunto de operações a serem executadas por sistemas de modo a se tornarem resilientes. Entre essas operações estão a *detecção*, a *remediação*, o *diagnóstico* e a *recuperação* de um dado problema. Detectar problemas consiste em identificar comportamentos inesperados ou degradações no desempenho do sistema. Remediá-los, por sua vez, corresponde à minimizar seu impacto nos serviços fornecidos pelo sistema. Diagnosticar um problema implica em revelar sua causa raíz, o que permite que medidas corretivas sejam tomadas e possíveis falhas sejam prevenidas, mantendo o sistema operacional. Finalmente, recuperar diz respeito à capacidade de retornar a um estado normal de operação assim que os problemas forem solucionados.

Apesar de fornecer diretrizes de projeto, a estratégia $\mathrm{D}^2\mathrm{R}^2 + \mathrm{DR}$ é um modelo conceitual. Assim, sistemas ou componentes que a implementam o fazem com foco nos domínios ou aplicações para os quais eles são projetados. Como resultado, o reuso de código é comprometido e novas implementações devem ser feitas do zero. Diversas soluções existentes foram desenvolvidas para auxiliar na implementação de operações específicas. Entretanto, essas soluções trazem limitações como a falta de autonomia e adaptabilidade, e são raramente integradas em uma única solução coesa. Existem, por exemplo, muitas abordagens capazes de detectar um comportamento anômalo em sistemas de software. Entretanto, essas abordagens não são suficientes para diagnosticar a causa raíz dessas anormalidades. Essa tarefa é normalmente desempenhada por especialistas por meio da inspeção manual de informações obtidas de diferentes fontes. Isso também ocorre com o gerenciamento de ações remediativas, que normalmente é feito de maneira rígida e manualmente coordenado. Realizar essas tarefas é um desafio em sistemas dinâmicos e de larga escala dada a quantidade de dados a serem analisados e a complexidade do sistema a ser gerenciado. Consequentemente, o desempenho de sistemas frente à situações que requerem uma resposta imediata é comprometida. Além disso, essa dependência de especialistas e desenvolvedores vai contra a crescente necessidade de sistemas autônomos, os quais devem ser capazes de superar e se adaptar à situações desafiadoras sem a necessidade de intervenção humana.

Com base nestas limitações, a questão de pesquisa que guia esta tese é a seguinte. *Como fornecer a sistemas multiagentes, de forma neutra e reutilizável em relação ao domínio, a habilidade de remediar, diagnosticar e se recuperar de situações anormais?*

Para responder esta questão e construir nossa hipótese de pesquisa, nos baseamos na arquitetura BDI. A arquitetura BDI é uma abordagem bastante utilizada no desenvolvimento de agentes autônomos com características adaptativas. Essa arquitetura especifica três componentes principais: as crenças, os desejos e as intenções. Eles representam, respectivamente, a percepção que o agente tem do seu ambiente e de si mesmo, os estados que o agente deseja alcançar, e os desejos com os quais este agente está comprometido. Esses componentes são manipulados por uma série de funções abstratas em um ciclo de raciocínio. Como essas funções são customizáveis, é possível fornecer uma solução na qual a arquitetura BDI seja encarregada da coordenação de ações remediativas, assim como do diagnóstico e solução de causas de problemas, e da recuperação do sistema. Dessa forma, a hipótese de pesquisa investigada é a de que *um framework independente de domínio que estende a arquitetura BDI é uma solução efetiva para realizar as operações de remediação, diagnóstico e recuperação especificadas na estratégia D^2R^2 + DR, promovendo reuso entre diferentes domínios de aplicação enquanto fornece a agentes e sistemas multiagentes implementados a partir dele características autônomas e adaptativas.*

Assim, propomos um framework independente de domínio que implementa um conjunto de técnicas desenvolvidas com o objetivo de auxiliar agentes na coordenação automatizada de suas ações, seja para a remediação de problemas, diagnóstico de suas causas raíz, ou recuperação a um estado normal de operação. A primeira técnica estende a arquitetura BDI de modo a permitir que agentes selecionem de maneira autônoma o conjunto apropriado de ações para remediar problemas e lidar com suas causas. Essa técnica automatiza a coordenação de planos do agente, promovendo assim o seu reuso em diferentes domínios e permitindo que agentes decidam de maneira flexível a melhor ação a ser executada de acordo com seu contexto, objetivos e preferências. A arquitetura BDI estendida, ilustrada na Figura 4.2, inclui um conjunto de componentes estruturais que capturam o conhecimento de domínio necessário para dar suporte aos agentes na tomada de tais decisões. Esse conhecimento é usado em um mecanismo customizado de raciocínio, o qual seleciona planos remediativos, quando necessário, e gera objetivos para diagnosticar e lidar com causas de problemas.

As extensões estruturais tem o objetivo de fornecer as informações necessárias para a execução do comportamento remediativo e dar suporte às decisões que ele compreende. Alguns dos componentes estruturais, como preferências e objetivos restritos, permitem uma maior flexibilidade sobre a escolha das ações que vão ser executadas pelo agente, sejam elas remediativas ou não. Um papel particularmente importante é desempenhado pelo modelo de causa e efeito (Figura 4.1). Esse modelo, adaptado do modelo de características das linhas de produto de software, é responsável por fornecer a informação que relaciona diferentes eventos às suas possíveis causas. Nele, é possível especificar eventos obrigatórios, opcionais ou alternativos que devem ocorrer para que um determinado efeito aconteça.

Essas informações são utilizadas pelas funções customizadas de geração de opções e de seleção de planos. A função de geração de opções é responsável por gerenciar os objetivos do agente. Na extensão desenvolvida, essa função implementa um algoritmo de controle que gerencia o rastreamento dos objetivos associados a efeitos que tenham uma causa especificada no modelo de causa e efeito. Quando um objetivo com essa característica é gerado, esse algoritmo verifica se as causas existentes no modelo de causa e efeito coincidem com o contexto atual em que o agente está inserido. Caso essa informação não esteja disponível, são criados objetivos para que ela seja adquirida. Dessa forma, a função pode gerar objetivos associados à resolução das causas identificadas e verificar quando um problema foi completamente resolvido. Para isso, foi criada uma estrutura auxiliar, chamada *cause-effect status*, que mantém o registro dos problemas endereçados e dos diferentes elementos que podem compor suas causas. A função de seleção de planos, por sua vez, considera informações relacionadas à execução de planos, assim como as restrições impostas pelos objetivos, para selecionar as ações que melhor satisfaçam as preferências do agente.

Dado que o objetivo desta técnica é permitir que agentes coordenem de maneira independente e automatizada as ações para remediar problemas e resolver suas causas, ela foi avaliada em relação à sua efetividade na realização dessa tarefa. Foi desenvolvido um sistema multiagentes que implementa o comportamento remediativo no combate à ataques distribuídos de negação de serviços. Nesse cenário, quando um tráfego anormal é identificado em um link de uma rede, ele deve ser prontamente remediado pra evitar possíveis efeitos prejudiciais ao serviço fornecido. Para isso, o tráfego é limitado nesse link e o sistema tenta identificar o alvo do ataque. Quando o alvo é identificado, o tráfego destinado a ele é reduzido e o sistema então identifica a fonte do ataque. Quando ela é identificada, seu tráfego é limitado e o problema é considerado completamente resolvido.

O sistema que implementa a abordagem proposta foi comparado à uma solução existente que implementa o comportamento remediativo de forma manual. Como resultado, foi possível verificar que o agente que fez uso da abordagem automatizada de gerenciamento de ações remediativas foi capaz de remediar e resolver o problema de forma efetiva, sem degradação significativa de performance quando comparado ao sistema no qual essas ações são gerenciadas manualmente. Tal resultado de performance é relevante pois, normalmente, implementações genéricas apresentam um desempenho pior por não permitirem ajustes específicos com foco no domínio em que elas são implantadas. Destaca-se ainda que a efetividade do comportamento remediativo foi obtida com uma redução de quase 25% no esforço de desenvolvimento em termos de linhas de código, já que a coordenação do comportamento do agente é todo encapsulado na abordagem desenvolvida. Tal resultado é relevante visto que o código que implementa esse comportamento é complexo, contendo diversos pontos de decisão que podem levar a diferentes caminhos de execução.

A segunda técnica que compõe o framework proposto se aproveita de uma característica particular de sistemas multiagentes para diagnosticar a causa de problemas de modo cooperativo e em tempo de execução. Nesse tipo de sistema, agentes interagem fornecendo e consumindo serviços. Quando uma anormalidade em um serviço fornecido por um agente é detectada, ela pode ter três origens possíveis: (i) no próprio agente que fornece o serviço; (ii) em componentes que fornecem serviços àquele agente; ou (iii) no canal de comunicação entre eles.

A técnica de diagnóstico cooperativo especifica um protocolo de interação entre agentes e o comportamento que eles devem assumir quando desempenham diferentes papéis dentro do sistema. Esse protocolo, apresentado na Figura 5.3, determina como os agentes interagem entre si por meio da troca de mensagens de requisição e de informação. O principal objetivo dessa troca de mensagens é o consumo e fornecimento de serviços. Porém, quando um comportamento anormal é detectado em um agente, tais mensagens também são utilizadas pra requisitar informações adicionais ou pra notificar problemas e suas soluções.

De acordo com esse protocolo, um agente pode desempenhar os papéis de cliente, fornecedor, e agente cooperativo. Como cliente, um agente consome serviços de agentes fornecedores. Normalmente, tais serviços devem obedecer a determinados requisitos de qualidade que, quando violados, dão origem a problemas que devem ser resolvidos. Um agente cliente registra todas as suas interações com seus fornecedores em uma estrutura chamada *traço de interação*. Quando uma requisição de serviço é feita, um traço é criado contendo a identificação do fornecedor, o serviço requisitado e os requisitos de qualidade do cliente. Quando essa requisição é respondida, o traço de interação é atualizado com informações sobre como o serviço atendeu aos requisitos de qualidade, além do registro do instante em que a requisição foi atendida. Além disso, clientes também são responsáveis por notificarem seus fornecedores quando requisitos de qualidade são violados.

Como fornecedor, um agente atende à requisições de serviços de seus clientes e também age quando notificado de que violou requisitos de qualidade. Quando recebe uma notificação dessa natureza, o comportamento remediativo tem início e um processo de verificação em duas etapas entra em ação. Esse processo realiza uma verificação interna e uma verificação externa. A primeira tem o objetivo de diagnosticar se a violação de requisitos teve origem no próprio agente ou não, ou seja, se é uma causa interna ou externa. Para isso, o agente verifica se, ao fornecer o serviço que apresentou uma anomalia, consumiu serviços de terceiros. Em caso afirmativo, é feita uma análise estatística utilizando o método de cercas de Tukey pra comparar as métricas de qualidade dos serviços consumidos com dados históricos. Nesse método estatístico, tendo por base uma amostra de dados, são calculados valores que determinam limites superiores e inferiores de normalidade. Qualquer valor fora desse intervalo é considerado anormal.

Se nenhuma anormalidade for verificada ou se nenhum serviço de terceiros tiver sido consumido para fornecer o serviço anormal, assume-se que o problema seja interno ao fornecedor. Uma ação de reparação é então executada e o cliente é notificado de que o serviço foi restabelecido. Caso contrário, uma ação remediativa é executada e o cliente é notificado. Nesse caso a segunda etapa de verificação é realizada com o objetivo de diagnosticar se a causa da anormalidade tem origem no serviço de um terceiro ou no meio de comunicação entre o agente fornecedor e seu provedor. Para isso, o agente solicita que outros componentes do sistema que já tenham consumido o mesmo serviço forneçam uma probabilidade de que aquele serviço apresente uma anomalia. As probabilidades informadas são computadas em uma pontuação que é utilizada como parâmetro pra decidir qual das causas externas deve ser corrigida.

Caso se verifique que o canal de comunicação entre agentes é a causa do problema, uma ação capaz de reparar o mesmo é executada. Por outro lado, caso o serviço de um terceiro seja considerado anormal, seu fornecedor é notificado dessa anormalidade e o agente aguarda um informe de que o serviço retornou ao normal. Em ambos os casos, as ações remediativas tomadas para mitigar o problema externo são revertidas após a solução definitiva do problema. Esta técnica permite que o modelo de causa e efeito seja construído dinamicamente e de forma transparente ao desenvolvedor.

Finalmente, o último papel que pode ser desempenhado por um agente é o de agente cooperador. Quando desempenha esse papel, o agente é responsável por responder às requisições de probabilidade enviadas por agentes que lidem com uma situação problemática. Para isso, o agente cooperativo consulta seus traços de execução com o intuito de verificar se já consumiu o serviço requisitado do agente suspeito. Em caso afirmativo, uma função de densidade é estimada a partir de um estimador de densidade de kernel. Dado que dados coletados mais recentemente tendem a ter um poder informativo maior, esse kernel atribui pesos maiores a traços mais recentes. O método de cercas de Tukey é usado novamente para determinar o intervalo de valores normais. O agente então calcula a probabilidade de um valor randômico da distribuição estimada cair fora deste intervalo. Essa probabilidade é então retornada como resposta à requisição feita ao agente cooperativo.

Dado que o objetivo desta técnica é permitir o diagnóstico de causas de problemas em tempo de execução, foi realizada uma avaliação em relação à habilidade de um sistema multiagente em diagnosticar a causa raíz de problemas em diferentes locais e vindos de diferentes fontes. Foi implementada uma simulação compreendendo diversos agentes que interagem pra consumir e fornecer serviços (Figura 5.5). Tais serviços possuem custos associados a eles e o requisito de que sejam fornecidos dentro de um intervalo determinado. Três falhas foram inseridas no decorrer da simulação representando problemas em diferentes pontos: (i) em um componente; (ii) em um link de comunicação; e (iii) em ambos. Os resultados foram comparados com duas outras estratégias de resolução de problemas.

Na primeira, chamada de passiva, os agentes ignoram qualquer notificação de anormalidade que lhes é enviada. Na segunda, chamada de remediativa, os agentes são capazes de mitigar anormalidades mas não tem os mecanismos pra diagnosticar e resolver suas causas.

A Figura 5.6 apresenta os resultados obtidos em relação ao custo e tempo percebidos pelo agente p_a. É possível perceber que, utilizando a estratégia passiva, nada acontece no sistema quando a primeira falha ocorre e o requisito de qualidade *tempo* é violado. Já na estratégia remediativa, assim que essa violação ocorre, ela é mitigada. Entretanto, essa mitigação tem um impacto no custo do serviço fornecido. Quando se observam os resultados da estratégia cooperativa, é possível notar que tanto o tempo quanto o custo para o fornecimento de um serviço retornam a um patamar normal após a ocorrência de cada uma das falhas. Isso acontece não apenas pela capacidade de se mitigar anomalias, mas também de corrigir suas causas e retornar o sistema à uma configuração menos onerosa.

A terceira técnica desenvolvida tem o objetivo de automatizar o retorno do sistema à sua operação normal. Ela também estende a arquitetura BDI em termos de componentes estruturais e funcionais. Os componentes estruturais integrados à arquitetura tem a responsabilidade de registrar as ações executadas pelo agente e informar quando essas ações podem ser revertidas. Já os elementos funcionais são responsáveis por monitorar as ações executadas e realizar o processo de reversão. A notação Z foi utilizada a fim de formalizar estas extensões.

O principal elemento estrutural adicionado à arquitetura foi chamado de *goal achievement metadata*, e suas instâncias dizem respeito a objetivos individuais. Este componente compreende três conjuntos de informação: (i) as condições de reversão, que fornecem a informação sobre quando um processo de reversão pode ocorrer; (ii) as condições de descarte, que informam quando o processo de reversão não está mais disponível; e (iii) o traço de mudanças de crenças, que registra os efeitos das ações executadas pelo agente pra atingir o objetivo ao qual estes metadados estão relacionados.

Além deste componente estrutural, três operações foram especificadas. A operação de *configuração de desejos* é a responsável por instanciar os metadados de objetivos cujas ações executadas para alcançá-los podem ser revertidas. A operação de *monitoramento*, por sua vez, faz a atualização dos parâmetros utilizados para verificar a disponibilidade ou não do processo de reversão, assim como do traço de mudanças de crenças. Por fim, a operação de *execução da reversão* estende a função de geração de opções da arquitetura BDI. É nesta operação que o processo de decisão sobre a realização ou não de uma reversão acontece, as ações que devem ser revertidas são identificadas, e os objetivos cuja satisfação resultam no retorno do sistema a um estado desejado são criados.

Dado que o objetivo dessa técnica é reverter ações, com foco especial em ações remediativas, foi realizada uma avaliação empírica sobre a sua efetividade em reverter ações em um cenário de Internet das Coisas onde um sistema multiagente é capaz de gerenciar diversas tarefas dentro de uma casa. Os agentes que compõem esse sistema são capazes de controlar dispositivos como alarmes, lâmpadas e detectores de gás, e devem lidar com um vazamento de gás. É esperado que, quando um aumento no nível de gás seja detectado dentro da casa, uma ação remediativa seja imediatamente coordenada para ligar exaustores, abrir janelas e destrancar portas de modo a permitir a evacução da residência. Quando a causa do problema for diagnosticada e resolvida, todas as ações, exceto aquelas que corrigiram definitivamente o problema, devem ser revertidas.

A Figura 6.3 e Tabela 6.2 mostram o comportamento do sistema nessa simulação com relação às ações do sistema e a concentração de gás observada dentro da casa. Quando essa concentração passa de um limite aceitável no instante I_1, as ações de remediação são executadas. Elas tem efeito no instante I_2 e permanecem ativas até que a anormalidade seja corrigida no instante I_3. Nos instantes I_4 e I_5 a causa do problema é detectada e corrigida. Isso permite que as ações remediativas sejam revertidas, o que acaba estabilizando a concentração de gás quando o processo de reversão é finalizado. Essa simulação demonstrou a efetividade do sistema na reversão autônoma das ações remediativas.

Sistemas multiagentes realizam tarefas complexas e potencialmente críticas e, portanto, um comportamento resiliente é necessário para que tais sistemas possam operar de acordo com as expectativas. Nesta tese, apresentamos um framework cujo objetivo é prover sistemas multiagentes com capacidade de remediar, diagnosticar e se recuperar de problemas. Nós nos aproveitamos das operações especificadas pela estratégia D^2R^2 + DR para permitir que esses sistemas mitiguem os efeitos de eventos problemáticos enquanto diagnosticam e resolvem suas causas. Esse framework disassocia esse comportamento remediativo do código dependente de domínio, promovendo assim o reuso de software entre aplicações distintas. Ele estende a arquitetura BDI típica e compreende três técnicas principais. A primeira automatiza o gerenciamento de ações remediativas e o diagnóstico e solução de causas de problemas. Ela inclui um conjunto de componentes que capturam o conhecimento de domínio que dá suporte a agentes na realização dessas operações. A segunda é focada no diagnóstico de causas de problemas em cenários multiagentes. Ela especifica um protocolo de interação e papéis que descrevem como agentes podem coordenar suas ações e compartilhar informações para manterem-se operando com a qualidade esperada. Finalmente, a terceira técnica formaliza uma abordagem para reverter ações de agentes BDI. Essa técnica permite que agentes desfaçam os efeitos de ações executadas sem a necessidade de uma declaração explícita de quais são essas ações e como elas devem ser revertidas.

Todas essas técnicas foram implementadas como uma extensão de uma plataforma para o desenvolvimento de agentes BDI. Essa implementação serviu como base para a condução de estudos empíricos com o objetivo de avaliar diferentes aspectos do framework proposto. Com relação à questão de pesquisa apresentada, os resultados mostraram evidência de que nosso framework é capaz de prover agentes e sistemas multiagentes com as habilidades necessárias para desempenhar as operações de remediação, diagnóstico e recuperação especificadas pela estratégia D^2R^2 + DR. No que diz respeito à nossa hipótese, concluímos que a solução proposta pode servir como base para a implementação de agentes e sistemas multiagentes com comportamento remediativo em diferentes domínios. Devido à sua arquitetura modularizada, habilidades específicas de domínio podem ser incorporadas sem prejudicar o comportamento subjacente. Finalmente, a autonomia e adaptabilidade herdadas do modelo BDI foram demonstrados em nossos experimentos.

Diversas contribuições podem então ser enumeradas como resultado do trabalho apresentado nesta tese. Juntas, elas compõem nossa solução para o gerenciamento automatizado de comportamento remediativo.

Técnica para o Gerenciamento Automatizado de Ações Remediativas. Essa técnica é responsável por automatizar a coordenação de ações executadas por agentes durante as operações de remediação e diagnóstico de problemas. Ela abstrai a informação causal utilizada durante o processo de diagnóstico em uma estrutura chamada de modelo de causa e efeito. Assim, a informação dependente de domínio é dissassociada do raciocínio remediativo. Como resultado, o framework pode ser reutilizado em diferentes domínios, necessitando apenas da especificação do modelo de causa e efeito correspondente para operar. Esse modelo pode ser especificado manualmente, em tempo de projeto, ou dinamicamente, durante a execução do sistema. Isso permite o desenvolvimento de estratégias específicas de domínio para coletar e manter informações causais em tempo de execução.

Técnica para o Diagnóstico Cooperativo de Causas de Problemas. Fornece a agentes a habilidade de cooperar para coletar informações usadas no diagnóstico da causa raíz de problemas em sistemas multiagentes. Essa técnica compreende um protocolo de interação e algoritmos que especificam o comportamento de agentes ao desempenharem os papéis de clientes, fornecedores de serviços e componentes cooperativos. A adoção desta técnica permite que sistemas multiagentes resistam e se adaptem à ocorrência de falhas em diferentes camadas do sistema.

Técnica para a Reversão de Ações. Esta técnica permite que agentes desfaçam os efeitos de ações remediativas após a completa resolução dos problemas para os quais elas foram executadas. A habilidade fornecida por esta técnica é particularmente adequada para cenários nos quais recursos valiosos são alocados para a remediação de um problema mas devem ser liberados tão logo seja possível para permitir a execução de futuras ações.

Framework de Desenvolvimento. As técnicas descritas nesta tese foram implementadas como um framework que estende uma plataforma existente, chamada BDI4JADE, que implementa a arquitetura BDI. Nossas técnicas são encapsuladas em capacidades e, devido a isso, podem ser usadas tanto para o desenvolvimento de novos agentes quanto incorporados à agentes já existentes, necessitando apenas sua instanciação para que sejam utilizados.

Avaliações Empíricas. Três estudos empíricos foram conduzidos para avaliar diferentes aspectos do framework proposto. Realizando tais estudos, fornecemos evidência da reusabilidade do framework proposto entre diferentes aplicações em diferentes domínios.

Em resumo, essa tese avança a pesquisa na redução da lacuna entre a definição conceitual de um comportamento remediativo e sua implementação em agentes e sistemas multiagentes. Mais trabalho é necessário para desenvolver uma solução geral para prover componentes de software com resiliência, mas este trabalho é um dos passos nessa direção.