# Cleaning Up the Mess: a Formal Framework for Autonomously Reverting BDI Agent Actions

João Faccin
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Porto Alegre, Brazil
jgfaccin@inf.ufrgs.br

Ingrid Nunes
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Porto Alegre, Brazil
ingridnunes@inf.ufrgs.br

## ABSTRACT

In order to cope with abnormal situations, self-adaptive systems may perform remediation actions to guarantee an acceptable system behaviour while dealing with critical execution problems. When problems are solved, the effect of such actions may need to be reverted to return the system to a normal state. This leads to the general problem of providing systems with the ability of autonomously reverting actions. In this paper, we address this problem by proposing a formal framework that customises the *belief-desire-intention* (BDI) architecture, typically used to implement autonomous agents. This customisation involves steps to monitor agent actions and store reversion metadata, identify when and which actions must be reverted, and execute action reversion. 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 the context of smart homes.

## CCS CONCEPTS

• **Software and its engineering** → *Software development techniques*;

## KEYWORDS

Self-adaptive systems, BDI architecture, reversion actions, remediation behaviour, autonomous agents

**ACM Reference Format:**
João Faccin and Ingrid Nunes. 2018. Cleaning Up the Mess: a Formal Framework for Autonomously Reverting BDI Agent Actions. In *SEAMS '18: SEAMS '18: 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3194133.3194156

## 1 INTRODUCTION

Self-adaptive systems perform many tasks to be able to adapt themselves according to changes in their environment. These tasks range from monitoring the environment where these systems are located, e.g. by collecting data from sensors [11], to acting towards the achievement of particular goals, e.g. turning on the lights in a room when someone is present [7]. Performed actions may be an atomic simple action in reaction to a change in a variable, or a set of actions selected by a sophisticated reasoning process. To achieve goals that require a system to overcome an abnormal execution scenario, it may adopt remediation actions to keep the system operational, before taking the time to actually find the cause of the abnormality and address it.

*Remediation actions* are those performed to tackle the (negative) effects of an underlying problem when certain constraints limit the resolution of its causes [6]. These actions prevent the occurrence of further problem effects before causes are addressed to permanently solve the problem. An example is when there is a failure in the communication with a temperature sensor. In this case, the system may, as a remediation action, rely on last known data as well as activate new neighbour sensors to estimate the temperature to keep its temperature-dependent behaviour operational while searching for causes and eventually resolving them. After these causes are permanently solved, changes made by remediation actions often must be reverted, such as deactivating neighbour sensors, which are not needed anymore and demand more energy if kept activated. That is, there is a need for *cleaning up (or reverting)* the effects of performed remediation actions. This need emerges in many domains such as smart grids and computer networks [13, 17].

The idea of reverting or undoing actions also appears in the context of transactions, mainly in databases, in which actions already completed should be undone when the transaction cannot be fully completed with success. This restores a system to a previous valid state, process which is referred to as *rollback*. The issue we address in this paper is similar, but it has key differences. On the one hand, (database) transactions have a set of properties—atomicity, consistency, isolation, and durability referred to as ACID properties [8]—to guarantee validity and be able to rollback if needed. On the other hand, our goal is to revert the effects of actions that have been done at a certain point in time, in a not necessarily atomic and isolated manner. That is, the actions to have their effects reverted have already been "committed," and at a posterior point in time must have their effects (partially) undone.

In this paper, we thus address the general problem of reverting (the effects of) actions, which is a fundamental issue in self-adaptive systems. Existing approaches address instances of this problem by dealing with failure handling and task aborting situations, e.g. the specification of compensating transactions and cleaning up plans (i.e., sets of actions to achieve a goal) in agent-based systems [9, 10]. Although these approaches do not have the same premises of database transactions, they are limited to reverting a predefined set of


*SEAMS '18, May 28–29, 2018, Gothenburg, Sweden*

ACM ISBN 978-1-4503-5715-9/18/05…$15.00
https://doi.org/10.1145/3194133.3194156

actions that must be undone to recover from plan failures. Alternatively, there are existing application-specific solutions to revert remediation actions. Nevertheless, these solutions are not reusable because the specification of *which*, *when* and *how* actions are undone is fixed to the scenario being developed. As a consequence, the use of existing solutions becomes constrained to particular application domains.

Our proposal consists of a formal framework to provide self-adaptive systems 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 reversible circumstance, such as failure handling and task aborting situations. We assume that systems are composed of one or more autonomous components, referred to as *agents*, which are structured according to the BDI architecture [14], that is, in terms of beliefs, desires (or goals) and intentions (with plans to achieve them). This architecture allows the development of components with reactive and proactive characteristics, which are arguibly suited to deal with problems in which a more flexible and intelligent behaviour is required. Such components are thus able to pursue goals while adapting their course of actions with the aim of reacting to events perceived at runtime. Our formal framework involves the specification of three main steps, integrated to the BDI reasoning cycle, to revert actions: (i) to monitor and record changes made by plans; (ii) to recognise the circumstances in which these actions should be reverted; and (iii) to execute the reverting process. The framework was implemented as an extension of the BDI4JADE platform [12]. This extension was used to practically validate our approach with the implementation of a smart home application that aims to prevent catastrophic events in homes, thus providing safety to residents.

We next describe a motivating scenario, which illustrates our problem and is used to validate our approach. In Sections 3 and 4, we overview our proposed framework and describe its operations, respectively. Our framework is then evaluated in Section 5. Section 6 discusses related work, followed by Section 7, which presents our conclusion.

## 2 MOTIVATION SCENARIO

To illustrate the problem that our work addresses, 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 [15]. 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 inspection 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.

## 3 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.

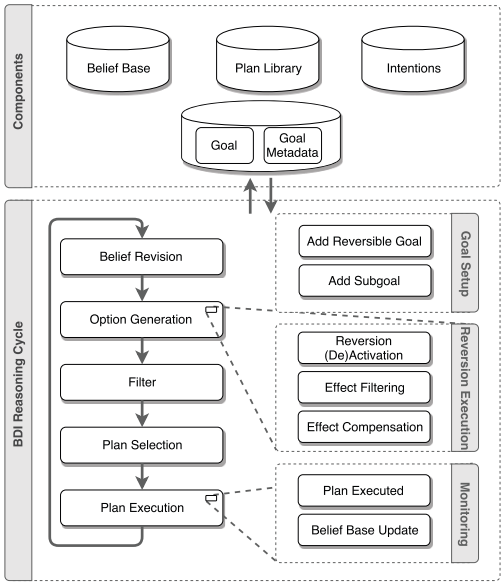


**Figure 1: Overview of the Reversion Framework.**

## 3.1 Framework Overview

As introduced, the *belief-desire-intention* (BDI) architecture [14] consists of a promising approach to develop autonomous components, hereafter referred to as agents. A key advantage provided by this architecture is to explicitly separate the motivational state of a system, expressed by *goals* (or desires), from its behaviour. This leads to a cycle of practical reasoning to deliberate goals to be achieved and to choose plans (sets of actions) to achieve selected goals. This is complemented by an informative state of the system—a set of beliefs, or belief base—which is updated according to agent perceptions.

In order for agents to have a sophisticated proactive and reactive behaviour, four functions within the BDI reasoning cycle can be customised [5]. First, the *belief revision* function is responsible for updating existing beliefs based on what the agent perceives. This updated information and the current motivational agent state are used by the *option generation* function to update the set of current agent goals, generating new goals or dropping existing ones. Then, the *filter* function selects goals the agent will commit to achieve, turning them into intentions. Finally, for each selected goal, the *plan selection* function chooses a plan from those available in a plan library. These selected plans are then executed to hopefully achieve their goals.

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 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.

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.

## 3.2 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 [16]. We also formalise key concepts from the BDI architecture needed for our customisation. A complete formalisation of the BDI architecture can be seen elsewhere [4].

Information about the environment and the internal state of an agent is represented by logic 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.

$[PREDICATE]$
$BOOLEAN ::= True \mid False$

*BeliefBase*

$knowledge : \mathbb{P}\ PREDICATE$
$belief : PREDICATE \pfun BOOLEAN$

---

$knowledge = \mathrm{dom}\ belief$

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]

$TYPE ::= Achievement \mid Query$
$ENDSTATE ::= Achieved \mid Failed \mid NoLongerDesired \mid Nil$

*Goal*
$predicate : PREDICATE$
$type : TYPE$
$endState : ENDSTATE$

When the filter function of an agent selects a goal to which the agent will commit to 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. Due to space restrictions, 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.

*GoalMetadata*
$goal : Goal$
$intention : Intention$
$reversionTrigger : PREDICATE$
$rollback : BOOLEAN$
$maxExecutedPlans : \mathbb{N}$
$maxTime : \mathbb{N}$
$achievedTime : \mathbb{N}$
$planCounter : \mathbb{N}$
$beliefChangeTrace : PREDICATE \pfun \text{seq}(BOOLEAN \times \mathbb{N}_1)$

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

[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.

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.

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*
$beliefBase : BeliefBase$
$goals : \mathbb{P}\ Goal$
$goalMetadata : Goal \rightarrowtail\!\!\!\!\!\!\!\!\rightarrow\ GoalMetadata$
$parentGoal : Goal \pfun Goal$
$intentions : \mathbb{P}\ Intention$
$planLibrary : \mathbb{P}\ Plan$

$\text{dom}\ goalMetadata \subseteq goals$
$\text{dom}\ parentGoal \subseteq goals$
$\text{ran}\ parentGoal \subseteq goals$

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

## 4 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.

### 4.1 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*

$\Delta Agent$
$predicate? : PREDICATE$
$type? : TYPE$
$reversionTrigger? : PREDICATE$
$rollback? : BOOLEAN$
$maxExecutedPlans? : \mathbb{N}$
$maxTime? : \mathbb{N}$

---

$\forall\, goal : Goal \mid goal \in goals \bullet$
$\quad goal.predicate \neq predicate? \lor$
$\quad (goal.predicate = predicate? \land goal.type \neq type?)$

$(\textbf{let}\ goal == (\mu\, g : Goal \mid g.predicate = predicate? \land$
$\quad g.type = type? \land g.endState = Nil) \bullet$
$\quad\quad goals' = goals \cup \{goal\} \land$
$\quad\quad (\textbf{let}\ metadata == (\mu\, m : GoalMetadata \mid$
$\quad\quad\quad m.goal = goal \land m.intention = Nil \land$
$\quad\quad\quad m.reversionTrigger = reversionTrigger? \land$
$\quad\quad\quad m.rollback = rollBack? \land$
$\quad\quad\quad m.maxExecutedPlans = maxExecutedPlans? \land$
$\quad\quad\quad m.maxTime = maxTime? \land$
$\quad\quad\quad m.achievedTime = Nil \land$
$\quad\quad\quad m.planCounter = 0 \land$
$\quad\quad\quad m.beliefChangeTrace = \varnothing) \bullet$
$\quad goalMetadata' = goalMetadata \cup (\{goal\} \times \{metadata\})))$

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.

*MakeIntention*

$\Delta Agent$
$goal? : Goal$

---

$goal? \in goals$
$goal? \in \mathrm{dom}\, goalMetadata$

$(\textbf{let}\ intention == (\mu\, i : Intention \mid i.goal = goal? \land$
$\quad i.planInstance = Nil) \bullet$
$\quad\quad intentions' = intentions \cup \{intention\} \land$
$\quad\quad (\textbf{let}\ metadata == goalMetadata(goal?) \bullet$
$\quad\quad\quad metadata.intention = intention \land$
$\quad\quad\quad goalMetadata' = goalMetadata \cup$
$\quad\quad\quad\quad (\{goal?\} \times \{metadata\})))$

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.

*AddAchievedTime*

$\Delta Agent$
$goal? : Goal$
$time? : \mathbb{N}$

---

$goal? \in goals$
$goal? \in \mathrm{dom}\, goalMetadata$
$goal?.endState = Achieved$
$goalMetadata(goal?).achievedTime = Nil$

$(\textbf{let}\ metadata == goalMetadata(goal?) \bullet$
$\quad metadata.achievedTime = time? \land$
$\quad goalMetadata' = goalMetadata \cup (\{goal?\} \times \{metadata\}))$

## 4.2 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.

$selectParentGoal : Goal \times (Goal \pfun Goal) \longrightarrow Goal$

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*

$\Delta Agent$
$g? : Goal$
$p? : Predicate$
$value? : BOOLEAN$
$time? : \mathbb{N}$

---

$(p? \in beliefBase.knowledge \Rightarrow$
$\quad beliefBase'.beliefs(p) = value?) \land$
$(p? \notin beliefBase.knowledge \Rightarrow$
$\quad beliefBase' = beliefBase \cup (\{p?\} \times \{value?\}))$

$(\textbf{let}\ goal == selectParentGoal\,(g?, parentGoal) \bullet$
$(\textbf{let}\ m == goalMetadata\,(goal) \bullet$
$\quad (p? \in \text{dom}\ m.beliefChangeTrace \Rightarrow$
$\quad (\textbf{let}\ newTrace ==$
$\quad\quad m.beliefChangeTrace(p?) \frown \langle (value?, time?) \rangle \bullet$
$\quad\quad m.beliefChangeTrace = m.beliefChangeTrace \cup$
$\quad\quad\quad (\{predicate?\} \times \{newTrace\}) \land$
$\quad\quad goalMetadata' = goalMetadata \cup$
$\quad\quad\quad (\{m.goal\} \times \{m\}))) \land$
$\quad (p? \notin \text{dom}\ m.beliefChangeTrace \Rightarrow$
$\quad (\textbf{let}\ newTrace == \langle (value?, time?) \rangle \bullet$
$\quad\quad m.beliefChangeTrace = m.beliefChangeTrace \cup$
$\quad\quad\quad (\{predicate?\} \times \{newTrace\}) \land$
$\quad\quad goalMetadata' = goalMetadata \cup (\{m.goal\} \times \{m\})))))$

*AddSubgoal*

$\Delta Agent$
$predicate? : PREDICATE$
$type? : TYPE$
$parent? : Goal$

---

$parent? \in goals$

$\forall\, goal : Goal \mid goal \in goals \bullet$
$\quad goal.predicate \neq predicate? \lor$
$\quad (goal.predicate = predicate? \land goal.type \neq type?)$

$(\textbf{let}\ goal == (\mu\, g : Goal \mid g.predicate = predicate? \land$
$\quad g.type = type? \land g.endState = Nil) \bullet$
$\quad\quad goals' = goals \cup \{goal\} \land$
$\quad\quad parentGoal' = parentGoal \cup (\{goal\} \times \{parent?\})$

*IncreasePlanCounter*

$\Delta Agent$
$planInstance? : PlanInstance$

---

$\forall\, g : Goal \mid g \in goals \land g.endState = Achieved \bullet$
$\quad (\textbf{let}\ metadata == goalMetadata(g) \bullet$
$\quad\quad metadata.planCounter = succ\ metadata.planCounter \land$
$\quad\quad goalMetadata' = goalmetadata \cup (\{g\} \times \{metadata\}))$

## 4.3 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.

*4.3.1 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.

$isReversionDeactivated : (GoalMetadata \times \mathbb{N}) \longrightarrow BOOLEAN$

---

$\forall\, m : GoalMetadata;\ time : \mathbb{N} \bullet$
$\quad (isReversionDeactivated\,(m, time) = True \Rightarrow$
$\quad\quad (m.planCounter > m.maxExecutedPlans \lor$
$\quad\quad (time - m.achievedTime) > m.maxTime)) \land$
$\quad (isReversionDeactivated\,(m, time) = False \Rightarrow$
$\quad\quad (m.planCounter \leq m.maxExecutedPlans \land$
$\quad\quad (time - m.achievedTime) \leq m.maxTime))$

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.

*ExpireGoalMetadata*

$\Delta Agent$
$goal? : Goal$
$time? : \mathbb{N}$

---

$goal? \in goals$
$goal? \in \text{dom}\ goalMetadata$

$isReversionDeactivated(goalMetadata\ goal?, time?) = True \Rightarrow$
$\quad goalMetadata' = \{goal?\} \lhd goalMetadata$

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 was 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.

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

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

*4.3.2 Effect Filtering.* Before reverting actions associated with a goal, we must select the changes that must be actually reverted. As explained in our motivating example, some of the belief changes must not be undone. This is particularly important in the context of remediation actions, in which the effects of these actions must be persisted. Although in this work 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.

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

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

*4.3.3 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.

*GenerateReversionGoals*

$$\begin{aligned}
&\Delta Agent \\
&m? : GoalMetadata \\
&\hline
&m? \in \mathrm{ran}\ goalMetadata \\
&isReversionActivated\,(m?, beliefBase) = True \\
&(\mathbf{let}\ preds == filterBeliefChanges\,(m?, beliefBase) \bullet \\
&\quad goals' = goals \cup reversionGoals\,(preds) \land \\
&\quad m?.beliefChangeTrace = \varnothing \land \\
&\quad goalMetadata' = goalMetadata \cup (\{m?.goal\} \times \{m?\}))
\end{aligned}$$

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.

## 5 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 2. Finally, we present and discuss obtained results.

### 5.1 Implementation

Our framework was implemented using the BDI4JADE platform [12], which supports the development of BDI agents with a Java-based API. This platform provides easy means of adapting the BDI reasoning cycle due to customisation points, which motivates its use in our work.

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`. Capabilities in BDI4JADE are used to build agents by composing modular components, providing separation of concerns. 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.

A key motivation for our proposal is to use the mechanism of reverting actions together with the reasoning and management of remediation actions, which was done in previous work [6]. This work was also implemented in BDI4JADE and we, therefore, extended this 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 identify and solve problem causes while mitigating their effects. This information is used to model the triggering conditions to revert goals.

## 5.2 Case Study Description

Our motivating scenario presented in Section 2 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 `ManagerAgent`, which is responsible for broader decisions and for coordinating the interaction among agents associated with devices. It is the agent that includes the reasoning and management of remediation actions and their reversion. The `ManagerAgent` is responsible for, e.g., requesting an agent responsible for controlling the lights of a room to turn them on when agents associated with a presence and light sensors inform the `ManagerAgent` 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 `ManagerAgent`. Agents responsible for controlling sensors, such as the `CODetectorAgent`, are able to perform additional tasks, such as monitoring their sensors and notifying `ManagerAgent` when abnormal situations occur. Table 1 presents the set of device controlling agents from our case study as well as their corresponding plans.

The `ManagerAgent` 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

**Table 1: Device Controlling Agents and their Plans.**

| Agent | Plans |
|---|---|
| `CODetectorAgent` | `MonitorCOLevel` |
| `AlarmAgent` | `TakeOffAlarm`; `SilenceAlarm` |
| `LightsAgent` | `TurnLightsOn`; `TurnLightsOff` |
| `LockerAgent` | `UnlockDoor`; `LockDoor` |
| `WindowsAgent` | `OpenWindows`; `CloseWindows` |
| `VentilationAgent` | `TurnFansOn`, `TurnFansOff` |
| `ValveAgent` | `OpenValve`; `CloseValve` |

**Table 2: `ManagerAgent` 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 |

and, if it happens, the agent generates a goal to reduce the CO level. Therefore, the `ManagerAgent` 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 previously provided to the `ManagerAgent`, as required by the remediation approach integrated to our solution.

Our case study thus consists of the implementation of all involved agents located in a simulated environment. We then observe the `ManagerAgent` 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.

## 5.3 Results and Discussion

The results obtained by running our simulation is now described with a focus on the behaviour that was observed from `ManagerAgent`. Initially, the concentration of CO was at an acceptable level, below a given threshold. The `ManagerAgent` had a set of beliefs, corresponding to this CO level, as shown in the column and point *Start* of Table 2 and Figure 2, respectively. At some point, the water heater started to leak and the concentration of CO increased, as shown in Figure 2.

When the amount of CO surpassed the tolerated concentration, the `CODetectorAgent` detected it and notified the `ManagerAgent`. 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 ¬*abnormal*(*CO*), which had a reverting trigger

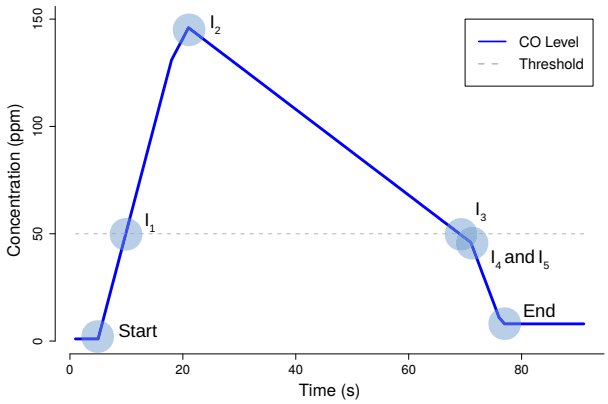


**Figure 2: Level of CO over Time.**

**Table 3: `ManagerAgent`: 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` |

condition $leak(waterHeater)$ and a discarding condition of 60 minutes. Because no problem cause was known, pre-conditions to permanently solve the cause of high CO level did not hold, so the `EvacuateAndVentilate` plan—a remediation 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 3. All actions were recorded as goal metadata. The new state of `ManagerAgent` is then $I_2$. Because of the remediation actions, the concentration of CO decreased slowly. When the amount of gas returned to an acceptable level, the `CODetectorAgent` detected it and notified the `ManagerAgent` (point $I_3$). The `ManagerAgent` updated its belief base (state $I_3$) leading to the successful execution of the remediation plan.

The cause of the problem was found with our remediation action mechanism (details can be found elsewhere [6]), 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 2 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 `ManagerAgent` was as shown in the last column of Table 2.

From the execution of this simulation, it is possible to observe that the behaviour presented by `ManagerAgent` corresponded to what was expected. The comparison of the *Start* and *End* states presented in Table 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 RELATED WORK

Mechanisms that focus on the reversion of actions has been carried out mainly in the context of ACID transactions [8]. 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 et al. [10] 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 [1] 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. [2] 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 et al. [19], the management of compensating actions is assigned to a particular system component, called FHC (failure-handling component). In their approach, compensating elements are associated with system goals instead of actions. In case of failure, the FHC is thus 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 [20].

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 [1, 2], 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 et al. [19], 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, self-adaptive 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 components, which we considered in our work. The Jason platform, for example, allows the specification of "clean up" plans [9], 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. [18]. In their work, the semantics of a plan-aborting mechanism is specified. As a consequence, plans can be associated with aborting actions, which are able to revert the effects of plan executions when they are interrupted. In the work of Wang et al. [21], compensating actions can have priorities, which may vary according to the context. For instance, a given action may present a higher priority when it is required by the system in order to release consumed resources, allowing goals to be reattempted.

Techniques focused on the BDI architecture present a lack of adaptation similar to that of 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-based systems, 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 exploited.

Finally, we can still relate our solution to more general approaches in the context of autonomous systems, such as the MAPE-K model [3]. In fact, some of the activities performed by our customised BDI model could be implemented as a feedback loop focused on agent goals and actions. In such feedback loop, *monitoring* would correspond to our activity of the same name. *Analysis* and *planning*, in turn, would be associated with the reversion (de)activation, and the effect filtering and compensation tasks from our reversion execution activity, respectively. Nevertheless, this feedback loop would still rely on additional activities to carry out the *executing* part of the model, which in our approach is addressed by the inherent features of the BDI reasoning cycle.

## 7 CONCLUSION

The need for reverting actions appears in many circumstances in software systems. This is typically done to guarantee that systems do not reach an invalid state when a set of actions that only make sense if all are successfully executed is performed. Self-adaptive systems can also need to revert actions for other reasons. These systems must adapt to cope with changes in the environment. Therefore, when environment changes cause system malfunctioning, remediation actions may be performed to keep the system operational before the cause of the problem is identified and addressed. When causes are resolved, the effects of remediation actions must be reverted.

Motivated by this scenario, we presented in the paper an approach to allow self-adaptive systems to autonomously manage the process of reverting actions, including when and what actions should be performed. We consider systems that are built as a set of autonomous components, named agents, which are structured using the BDI architecture. This architecture decouples what should be achieved (motivational state) from how it is achieved, being an arguably flexible solution to develop autonomous components. We proposed a formal extension of the BDI architecture that includes the structures needed to manage the process of reverting actions. Moreover, many operations incorporated to the BDI reasoning cycle were specified to control 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 set of autonomous agents performed remediation actions to reduce the levels of gas in a timely fashion before identifying 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 any application domain.

Future work involves addressing other issues associated with the provision of a complete approach to manage remediation actions. Currently, our solution is able to choose remediation actions to be performed, generate goals to find problem causes, resolve them and revert the effects of remediation actions. However, possible causes of the problem must be specified through a causal model. Our aim is to autonomously learn this causal model based on past system executions.

## ACKNOWLEDGMENTS

This work was supported by the National Council for Scientific and Technological Development (CNPq) (grant numbers 141840/2016-1, 303232/2015-3).

## REFERENCES


[1] Michael Butler and Carla Ferreira. 2000. *A Process Compensation Language.* Springer Berlin Heidelberg, Berlin, Heidelberg, 61–76. https://doi.org/10.1007/3-540-40911-4_5

[2] Mandy Chessell, Catherine Griffin, David Vines, Michael Butler, Carla Ferreira, and Peter Henderson. 2002. Extending the Concept of Transaction Compensation. (January 2002), 743–758 pages.

[3] IBM Corporation. 2006. An architectural blueprint for autonomic computing. (2006).

[4] Mark d'Inverno and Michael Luck. 2004. *Understanding agent systems.* Springer Science & Business Media. https://doi.org/10.1007/978-3-662-10702-7

[5] J. Faccin and I. Nunes. 2015. BDI-Agent Plan Selection Based on Prediction of Plan Outcomes. In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, Vol. 2. 166–173. https://doi.org/10.1109/WI-IAT.2015.58

[6] J. Faccin and I. Nunes. 2018. Remediating critical cause-effect situations with an extended BDI architecture. *Expert Systems with Applications* 95, Supplement C (2018), 190–200. https://doi.org/10.1016/j.eswa.2017.11.036

[7] J. Feng and Y. Yang. 2017. Design and implementation of lighting control system for smart rooms. In *IEEE International Conference on Computational Intelligence and Applications (ICCIA).* 476–481. https://doi.org/10.1109/CIAPP.2017.8167263

[8] J. Gray and A. Reuter. 1992. *Transaction Processing: Concepts and Techniques.* Elsevier Science.

[9] Jomi F. Hübner, Rafael H. Bordini, and Michael Wooldridge. 2006. *Programming Declarative Goals Using Plan Patterns.* Springer Berlin Heidelberg, Berlin, Heidelberg, 123–140. https://doi.org/10.1007/11961536_9

[10] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. 1990. A Formal Approach to Recovery by Compensating Transactions. In *International Conference on Very Large Data Bases.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 95–106.

[11] K. E. Nolan, M. Y. Kelly, M. Nolan, J. Brady, and W. Guibene. 2016. Techniques for resilient real-world IoT. In *International Wireless Communications and Mobile Computing Conference (IWCMC).* 222–226. https://doi.org/10.1109/IWCMC.2016.7577061

[12] Ingrid Nunes, Carlos J. P. De Lucena, and Michael Luck. 2011. BDI4JADE: a BDI layer on top of JADE. In *International Workshop on Programming Multi-Agent Systems.* 88–103.

[13] Ingrid Nunes, Frederico Schardong, and Alberto Schaeffer-Filho. 2017. BDI2DoS: An application using collaborating BDI agents to combat DDoS attacks. *Journal of Network and Computer Applications* 84 (2017), 14–24. https://doi.org/10.1016/j.jnca.2017.01.035

[14] Anand S. Rao and Michael P. Georgeff. 1995. BDI Agents: From Theory to Practice. In *First International Conference on Multi-Agent Systems (ICMAS).* 312–319.

[15] Kanta Sircar, Jacquelyn Clower, Mi kyong Shin, Cathy Bailey, Michael King, and Fuyuen Yip. 2015. Carbon monoxide poisoning deaths in the United States, 1999 to 2012. *The American Journal of Emergency Medicine* 33, 9 (2015), 1140–1145. https://doi.org/10.1016/j.ajem.2015.05.002

[16] J Michael Spivey. 1988. *Understanding Z: a specification language and its formal semantics.* Vol. 3. Cambridge University Press.

[17] Naser G. Tarhuni, Nagy I. Elkalashy, Tamer A. Kawady, and Matti Lehtonen. 2015. Autonomous control strategy for fault management in distribution networks. *Electric Power Systems Research* 121 (2015), 252–259. https://doi.org/10.1016/j.epsr.2014.11.011

[18] John Thangarajah, James Harland, David Morley, and Neil Yorke-Smith. 2007. Aborting Tasks in BDI Agents. In *International Joint Conference on Autonomous Agents and Multiagent Systems.* ACM, New York, NY, USA, 8–15. https://doi.org/10.1145/1329125.1329133

[19] Amy Unruh, James Bailey, and Kotagiri Ramamohanarao. 2004. A framework for goal-based semantic compensation in agent systems. In *International Workshop on Safety and Security in Multi-Agent Systems.*

[20] A. Unruh, H. Harjadi, J. Bailey, and K. Ramamohanarao. 2005. Semantic-compensation-based recovery in multi-agent systems. In *Multi-Agent Security and Survivability.* 85–94. https://doi.org/10.1109/MASSUR.2005.1507051

[21] M. Wang, K. Ramamohanarao, and A. Unruh. 2007. Utilizing BDI Features for Transactional Agent Execution. In *IEEE/WIC/ACM International Conference on Intelligent Agent Technology.* 215–221. https://doi.org/10.1109/IAT.2007.62