I N F S Y S

R E S E A R C H

R E P O R T

INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG WISSENSBASIERTE SYSTEME

Error-Tolerant Agents

Thomas Eiter   Viviana Mascardi   V.S. Subrahmanian

INFSYS RESEARCH REPORT 1843-00-07

OCTOBER 2000; MAY 2001

Institut für Informationssysteme
Abtg. Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

TECHNISCHE UNIVERSITÄT WIEN

INFSYS RESEARCH REPORT
INFSYS RESEARCH REPORT 1843-00-07, OCTOBER 2000; MAY 2001

Error-Tolerant Agents

Thomas Eiter[1] Viviana Mascardi[2,3] V.S. Subrahmanian[4]

Abstract. The use of agents in today's Internet world is expanding rapidly. Yet, agent developers proceed largely under the optimistic assumption that agents will be error-free. Errors may arise in agents for numerous reasons — agents may share a workspace with other agents or humans and updates made by these other entities may cause an agent to face a situation that it was not explicitly programmed to deal with. Likewise, errors in coding agents may lead to inconsistent situations where it is unclear how the agent should act. In this paper, we define an agent execution model that allows agents to continuing acting "reasonably" even when some errors of the above types occur. More importantly, in our framework, agents take "repair" actions automatically when confronted with such situations, but while taking such repair actions, they can often continue to engage in work and/or interactions with other agents that are unaffected by repairs.

[1]Institut und Ludwig Wittgenstein Labor für Informationssysteme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Wien, Austria. E-mail: eiter@kr.tuwien.ac.at
[2]DISI, Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy. email: mascardi@disi.unige.it
[3]Part of the work done while visiting TU Wien.
[4]Institute for Advanced Computer Studies, Institute for Systems Research and Department of Computer Science, University of Maryland, College Park, Maryland 20742. E-mail: vs@cs.umd.edu

Acknowledgements: This work was supported in part by the Austrian Science Fund projects P13871-INF and Z29-INF, by the Army Research Lab under contract number DAAL01-97-K0135, by an NSF Young Investigator Award IRI-93-57756, by a TASC/DARPA grant number J09301S98061, by ARO grant DAAD190010484, and by Dara/AFRL grant F306029910552.

In: 'Computational Logic: From Logic Programming into the Future' (in honour of Bob Kowalski), A. Kakas and F. Sadri (eds). Springer Verlag, to appear.

Contents

1 Introduction

Agents are a rapidly growing area of research in artificial intelligence and databases, with an ever increasing range of applications, spanning e-commerce servers to web search engines. Numerous paradigms for agents have been proposed in the AI literature [11, 30, 27]. In past work, two of the authors have been working on a framework called *IMPACT* (Interactive Maryland Platform for Agents Collaborating Together) [14, 4, 26] in which they develop a theory by which existing legacy code bases and data sources can be "agentized". In their framework, each agent has a state (composed of whatever resides in its data structures and message box). Whenever the agent's state changes, the agent must take actions in accordance with some clearly specified operating principles so as to ensure that the resulting state satisfies some integrity constraints. Examples of state changes include receipt of a message, a clock tick, a receipt of a service request, receipt of a response to a service request, update of a data source, and many others. Eiter *et al.*[14] show strong connections between the agent theory they propose with classical methods for logic programming, nonmonotonic reasoning. They further show how Shoham's AOP ("agent oriented programming") system [24] can largely be simulated within *IMPACT*, and that large parts of the well known belief, desires, and intentionality architecture (BDI) can be captured within their framework. Most of these frameworks all agree on the fact that an agent decides on what to do in response to a state change, and then does it. However, two major problems need to be addressed.

1. First, most agent frameworks (cf. [11, 30, 27]) including *IMPACT* assume that the rules used are sufficient to appropriately respond to all requests that arrive. Unfortunately, this assumption that the agent developer covered "all possibilities" is rather optimistic and as unreasonable as an assumption that all programs in C (or any other programming language) are bug-free. Hence, there is a question of what to do when an agent is confronted with a situation for which it does not know how to act.

2. Second, in the case of legacy systems, we note that the legacy system's existing GUI and the agent both access and update the same data. Thus, the legacy GUI may alter the agent's state in ways that the agent may find unacceptable.

An agent is said to be *corrupted* if either (i) changes caused by external entities have caused the agent's current state to violate one or more integrity constraints, or (ii) the agent is unable to find a "valid"[1] set of actions to execute in its current state (which may, perhaps, have been caused by a coding error). In this paper, we tackle the first problem above — the second is considered only to the extent that nonexistence of a status set is because of an integrity constraint violation.

This paper presents a theory, architecture and algorithms so that agents may exhibit two important properties.

1. **Recovery.** Agents must be able to recover from being "corrupted" to being "uncorrupted."

2. **Continuity.** Agents must continue to process some (though perhaps not all) requests while continuing to recover. This is important when an agent is servicing lots of requests.

The organization of this paper is as follows. In Section 2, we present a brief overview of *IMPACT*'s agent architecture (see [14, 4, 26] for more details). To this architecture, we add one component — an *error recovery component* whose architecture is described in Section 3. In Section 3, we provide a formal set of

[1] W.r.t. the semantics of the agent. In this paper, we will assume that either the feasible, rational or reasonable status set semantics of agents [14, 26] is used.

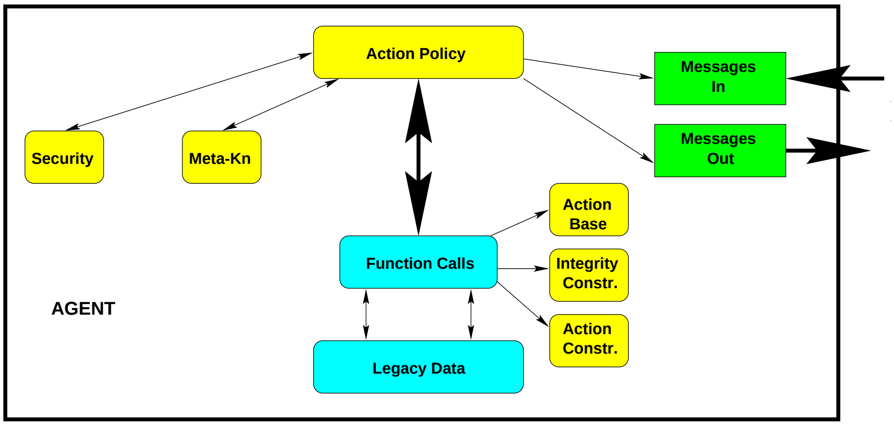

Figure 1: Basic Architecture of *IMPACT* Agents

definitions specifying what requests are affected (or may be affected) when an agent is known to be corrupted in a certain way. Unaffected requests may continue to be processed by a corrupted agent, even while the corrupted agent attempts to recover. Then, in Section 4, we describe special `repair` data structures and repair actions which are to be used by the recovery component. The latter may be selected from a repair action library, which provides a host of different realizations for repair. In Section 5, we discuss how an agent can, using the results and tools of the previous section, recover from an error. We not only show how *IMPACT* agents may use our recovery methods, but also present a modification of the Kowalski-Sadri agent cycle [20] as in [14, 26] which incorporates the desired properties of recovery and continuity. In Section 6, we discuss how our work may be applied to three different agent frameworks out there in the literature: Kowalski and Sadri's framework, the *BDI* (Belief, Desires, Intentionality) framework, and the work of Wooldridge. Other related work is discussed in Section 7. Directions for future work are discussed in Section 8.

2 *IMPACT* Preliminaries

As different application programs reason with different types of data, and even programs dealing with the same types of data often manipulate them in a variety of ways, it is critical that any notion of agenthood be applicable to arbitrary software programs. Agent developers should be able to select data structures that best suit the application functions desired by users of the application they are building. Figure 1 shows the architecture of a full-fledged *IMPACT* software agent. It is important to note that all agents have the same architecture and hence the same components, but the *content* of these components can be different, leading to different behaviors and capabilities offered by different agents.

Agent Data Structures. As all agents are built "on top" of some existing body of code, we first need an abstract definition of what that body of code looks like.

- First, we need a specification of the data types or data structures, $\mathcal{T}$, that the agent manipulates. As usual, each data type has an associated *domain* which is the space of objects of that type. For example, the data type `countries` may be an enumerated type containing names of all countries. At any given point, the *instantiation* or *content* of a data type is some subset of the space of the data-objects associated with that type.

- The above set of data structures is manipulated by a set of functions, $\mathcal{F}$, that are callable by external programs. Such functions constitute the *application programmer interface* or API of the package on top of which the agent is being built. An agent includes a specification of all signatures of these API function calls (i.e., types of the inputs to such function calls and types of the output of such function calls).

We use a unified language to query software packages by leveraging from $\mathcal{T}$ and $\mathcal{F}$. If $\mathtt{f} \in \mathcal{F}$ is an n-ary function defined in that package, and $\mathtt{t_1}, \ldots, \mathtt{t_n}$ are *terms* (either values, i.e., constants, or variables) of appropriate types, then $\mathcal{S} : \mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ is a *code call*. This code call says "Execute function f as defined in package $\mathcal{S}$ on the stated list of arguments." For evaluation, the code call must be ground, i.e., all arguments $\mathtt{t_i}$ must be values. We assume that it returns, as output, a *set* of objects— if a single object is returned, it can be coerced into a set anyway.

A *code call atom* is an expression cca of the form $\mathtt{in}(\mathtt{t}, \mathtt{cc})$ or $\mathtt{notin}(\mathtt{t}, \mathtt{cc})$, where t is a term and cc is a code call. For ground t, cca succeeds (i.e., has answer true) if t is in (resp., not in) the set of values returned by cc, and it fails (i.e., has answer false) otherwise. If t is a variable X, then cca returns each value from the result of cc, i.e., its answer is the set of ground substitutions θ for X such that $\mathtt{cca}\theta$ returns true. A uniform view of ground and non-ground case identifies the answer true with the set $\{\emptyset\}$ of the void substitution and the answer false with the empty set of substitutions.

For each code call atom cca, we denote by $\sim$cca the logically negated code call atom, i.e., $\sim\mathtt{in}(\mathtt{t}, \mathtt{cc}) = \mathtt{notin}(\mathtt{t}, \mathtt{cc})$ and $\sim\mathtt{notin}(\mathtt{t}, \mathtt{cc}) = \mathtt{in}(\mathtt{t}, \mathtt{cc})$. We extend this naturally to sets X of code call atoms by $\sim X = \{\sim\mathtt{cca} \mid \mathtt{cca} \in X\}$.

A *code call condition* is a conjunction of code call atoms and *constraint atoms*, which may involve deconstruction operations. An example of a constraint atom is $\mathtt{V.x} > 25$, where V.x accesses the x field of a variable V ranging over records that have an x field. It checks whether the stated condition is true; in general, constraint atoms are of the form $\mathtt{t_1}$ *op* $\mathtt{t_2}$ where *op* is any of $=$, $\neq$, $<$, $\leq$, $>$, $\geq$ and $\mathtt{t_1}, \mathtt{t_2}$ are terms.

Code call conditions provide a simple, but powerful syntax to access heterogeneous data structures. For example, the code call condition

$$\mathtt{in}(\mathtt{X}, \mathtt{oracle} : \mathtt{select}(\mathtt{emp}, \mathtt{sal}, >, \mathtt{100000}))\ \&$$
$$\mathtt{in}(\mathtt{Y}, \mathtt{image} : \mathtt{select}(\mathtt{imdb}, \mathtt{X.name}))\ \&\ \mathtt{in}(\text{“}\mathtt{Mary}\text{”}, \mathtt{imagedb} : \mathtt{findpeople}(\mathtt{Y}))$$

is a complex condition that joins data across Oracle and an image database. It first selects all people who make over 100K from an Oracle database and for each such person, finds a picture containing that person with another person called Mary. It generalizes the notion of join in relational databases to a join across a relational and image database.

Each agent is also assumed to have access to a message box data structure, together with some API function calls to access it. Details of the message box in *IMPACT* may be found in [14, 26].

At any given point in time, the actual set of objects in the data structures (and message box) managed by the agent constitutes the *state* of the agent. We shall identify a state $\mathcal{O}$ with the set of ground code calls which are true in it.

Actions. The agent has a set of *actions* $\alpha(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are variables for parameters, that can change its state. Such actions may include reading a message from the message box, responding to a message, executing a request, cloning a copy of the agent and moving it to a remote host, updating the agent data structures, etc. Even doing nothing may be an action. Expressions $\alpha(\vec{t})$, where $\vec{t}$ is a list terms of appropriate types, are *action atoms*. They represent the sets of (ground) actions which result if all variables in $\vec{t}$ are instantiated by values. Only such actions may be executed by an agent. Every action α has a precondition $Pre(\alpha)$ (which is a code call condition), a set of effects (given by an add list $Add(\alpha)$ and a delete list $Del(\alpha)$ of code call atoms) that describe how the agent state changes when the action is executed, and an *execution method* (which can be implemented in any programming language or scripting language that the user deems appropriate) consisting of a body of physical code that implements the action.

Notion of Concurrency. The agent has an associated body of code implementing a *notion of concurrency* $\mathbf{conc}(AS, \mathcal{O})$. Intuitively, it takes a set of actions AS and the current agent state $\mathcal{O}$ as input, and returns a single action (which "combines" the input actions together) as output. Various possible notions of concurrency are described in [14, 26]. They all have the property that the changes to the state $\mathcal{O}$ are restricted to the code call atoms occurring in the add and delete lists of the actions in AS. We make the same assumption in this paper.

Action Constraints. Each agent has a finite set of *action constraints* which are rules of the form "If the state satisfies some code call condition, then actions $\{\alpha_1, \ldots, \alpha_n\}$ cannot be concurrently executed." In the present paper, we disregard actions constraints, sine they can be easily eliminated (see [14]).

Integrity Constraints. Each agent has a finite set $\mathcal{IC}$ of *integrity constraints* ic that states $\mathcal{O}$ of the agent must satisfy (written $\mathcal{O} \models ic$ resp. $\mathcal{O} \models \mathcal{IC}$), of the form $\psi \;\Rightarrow\; \chi_a$ where ψ is a code call condition, and χ_a is a code call atom or constraint atom. Informally, ic has the meaning of the universal statement "If ψ is true, then χ_a must be true."[2] For example, a functional dependency $A1\ A2 \to B$ on a relation $\mathtt{r}$ in some database package $\mathtt{db}$ can be expressed as an integrity constraint

$$\mathtt{in}(\mathtt{T1}\ \mathtt{db}:\mathtt{all}(\mathtt{r}))\&\mathtt{in}(\mathtt{T2}\ \mathtt{db}:\mathtt{all}(\mathtt{r}))\&(\mathtt{T1.A1}=\mathtt{T2.A1})\&(\mathtt{T1.A2}=\mathtt{T2.A2}) \Rightarrow \mathtt{T1.B}=\mathtt{T2.B}$$

where $\mathtt{all}(\mathtt{r})$ returns all tuples in the relation $\mathtt{r}$. Throughout this paper, we assume that the integrity constraints are consistent, i.e., there exists at least one agent state $\mathcal{O}_0$ which satisfies all integrity constraints in $\mathcal{IC}$. It may happen, though, that a set of integrity constraints is not consistent. Determining such an inconsistency is, in general, an undecidable problem, and thus can not be done by an automated check. However, a software agent usually has a legal initial state $\mathcal{O}_0$ when it is deployed, and this state is known (or, it might be one out of a collection of possible states). The state $\mathcal{O}_0$ must satisfy all integrity constraints. Thus, in the specification of integrity constraints, only those may be accepted which hold on $\mathcal{O}_0$.

Agent Program. Each agent has a set of rules called the *agent program* specifying the principles under which the agent is operating. These rules specify, using deontic modalities, what the agent may do, must do, may not do, etc. Expressions $\mathbf{O}\alpha(\vec{t})$, $\mathbf{P}\alpha(\vec{t})$, $\mathbf{F}\alpha(\vec{t})$, $\mathbf{Do}\,\alpha(\vec{t})$, and $\mathbf{W}\alpha(\vec{t})$, where $\alpha(\vec{t})$ is an action atom, are called *action status atoms*. These action status atoms are read (respectively) as $\alpha(\vec{t})$ is *obligatory, permitted, forbidden, done*, and the obligation to do $\alpha(\vec{t})$ is *waived*. If A is an action status atom, then A and $\neg A$ are called *action status literals*. An *agent program* $\mathcal{P}$ is a finite set of rules of the form:

$$A \;\leftarrow\; \chi \,\&\, L_1 \,\&\, \cdots \,\&\, L_n \tag{1}$$

where A is an action status atom, χ is a code call condition, and $L_1, \ldots, L_n$ are action status literals. Due to space constraints, we do not repeat the semantics of agent programs here. A brief overview is given in Appendix A, while comprehensive details are given in [14, 26].

3 Architecture and Formal Definitions

In this section, we discuss how to extend the architecture in Fig. 1 to handle the cases where agent errors cause, due to violated integrity constraints, non-existence of valid status sets, and where an agent's state can be autonomously updated by a third party.

[2] For simplicity, we omit here and in other places safety aspects (see Appendix B and [14, 26] for details).

3.1 Architecture

We assume that there is some mechanism that notifies the agent when its state has been changed by a third party. Thus, we may assume in abstraction that every agent $\mathbf{a}$ receives messages of the following forms:

1. $\mathsf{ask}(\mathbf{b}, \mathtt{cca})$, where agent $\mathbf{b}$ is asking agent $\mathbf{a}$ the answer to a code call atom $\mathtt{cca} = \mathtt{in}(\mathtt{t}, \mathtt{cc})$ resp. $\mathtt{cca} = \mathtt{notin}(\mathtt{t}, \mathtt{cc})$, where $\mathtt{t}$ is a term and $\mathtt{cc}$ is ground.

2. $\mathsf{tell}(\mathbf{b}, \mathtt{cca}, \mathbf{ans})$, where agent $\mathbf{b}$ is telling agent $\mathbf{a}$ the answer $\mathbf{ans}$ to a code call atom $\mathtt{cca}$ of the previous form.

3. $\mathsf{done}(\mathtt{cca}, \mathbf{ans}^+, \mathbf{ans}^-)$, where $\mathtt{cca}$ is a code call atom and $\mathbf{ans}^+, \mathbf{ans}^-$ are sets of ground substitutions. Its meaning is that a third party (which may not be an agent) has updated agent $\mathbf{a}$'s state so that the answer to $\mathtt{cca}$ has changed — the new answer is the old one minus the substitutions in $\mathbf{ans}^-$ plus the substitutions in $\mathbf{ans}^+$.

Errors occur in the agent in one of two situations. In the first, incoming messages of the form $\mathsf{ask}(\cdot)$ or $\mathsf{tell}(\cdot)$ trigger errors as there is no valid status set associated with the incoming message.[3] In the second, another entity sends the agent a message of the form $\mathsf{done}(\cdot)$ and the update violates the integrity constraints of the agent, leaving it in a state which is invalid.

We deal with these two situations as follows. When an agent developer builds an *IMPACT* agent, she needs to perform the following tasks in order to specify how her *IMPACT* agents must recover when corrupted. She must specify

1. a set $\mathcal{RA}$ of *repair* actions having some properties (see Section 3); and,

2. an objective function (to be maximized) used to evaluate the cost of a state. The idea is that the agent code's repair component will automatically use repair actions to compute a state (which satisfies the integrity constraints or generates a valid status set).

Once the user specifies the various components of an agent as described in Section 2 and specifies the above parameters, the *IMPACT* Agent Development Environment should automatically convert the agent components plus the repair components into an executable body of Java bytecode which may then be deployed.

The continuity property of agents may be preserved by requiring that whenever an agent's state is corrupted by the actions of an external agent, the agent continues to process requests for its services as long as those requests are not "affected" by the ongoing repairs to the corrupted part. For example, an agent managing 30 relations in a relational database may find that external changes have corrupted one relation. In this case, queries that do not access that one relation may be processed by the agent while the corrupted relation is being repaired.

[3] The reader may wonder why an $\mathsf{ask}(\cdot)$ message can cause an error. All incoming messages to an agent cause a change in the agent's state because the message updates the agent's message box. No "sensible" integrity constraint should be violated because of an $\mathsf{ask}(\cdot)$ message. However, it is possible for an agent developer to write patently absurd integrity constraints. For instance, the syntax of ICs allows an agent developer to write rules such as "If the message box contains a message from agent B, then $\mathbf{F}a$" as well as "If the message box contains a message from agent B, then $\mathbf{P}a$". This causes an agent to become corrupt whenever a message from agent B arrives. The problem can be avoided by adding restrictions to the syntax of agent programs (e.g. certain types of *regular* agent programs introduced in [15] avoid this problem. In addition requiring that ICs not mention code call atoms involving $\mathsf{ask}(\cdot)$ messages would also help alleviate this problem.

We proceed as follows. In Section 3.2, we address the problem of specifying, given an agent $\mathbf{a}$ and a "corrupted"[4] code call atom $\mathtt{cca}$ resp. a set of such code call atoms, what other code call atoms may be potentially corrupted. The method we apply is based on a syntactic analysis of the agent's integrity constraints. We then introduce in Section 3.3 the notion of "suspiciousness" for code call atoms. Using this notion, we are able to determine which decisions that an agent tries to make are affected by these potentially corrupted code calls. This will be central for recovery in Section 5.

3.2 Corrupted code call atoms

When a set X of code call atoms is known to be corrupted, we would like to know what other code call atoms and integrity constraints are affected by this. In this section, we define a procedure called $\mathtt{corrcca}(X)$ that takes X as input, and returns, as output, the set of code call atoms in integrity constraints which are (potentially) corrupted by X. We first need some preliminary definitions. The first introduces the notion of subsumption for code call atoms.

Definition 3.1 (Code Call Subsumption) *A set of code call atoms X is subsumed by a set of code call atoms Y, written $X \lhd Y$, if each $\mathtt{cca} \in X$ is an instance of some $\mathtt{cca}' \in Y$ or its complement, i.e., $\mathtt{cca} = \mathtt{cca}'\theta$ or $\mathtt{cca} = \sim\mathtt{cca}'\theta$ for some substitution θ. If X (resp., Y) is a singleton set $\{\mathtt{cca}\}$, we omit parentheses and write $\mathtt{cca} \lhd Y$ (resp., $X \lhd \mathtt{cca}$).*

Here, and in the rest of the paper, we implicitly assume that code call atoms are standardized apart before unification.

Example 3.1 (Subsumption) *The code call atoms $\mathtt{in(a,cc1)}$ and $\mathtt{notin(Y,cc1)}$, where $\mathtt{cc1}$ is ground, are both subsumed by $\mathtt{in(X,cc1)}$. Thus, $\{\mathtt{in(a,cc1)}, \mathtt{notin(Y,cc1)}\} \lhd \{\mathtt{in(X,cc1)}, \mathtt{notin(a,cc2)}\}$.*

We next define how to associate with any code call condition χ, a set $\mathit{CCA}(\chi)$ of code call atoms. Informally, $\mathit{CCA}(\chi)$ is the set of code call atoms occurring somewhere in χ.

Definition 3.2 (Code-Call Atoms Set ($\mathit{CCA}(\chi)$)) *For any code call condition χ, the code call atom set $\mathit{CCA}(\chi)$ is inductively defined as follows:*

$$\mathit{CCA}(\chi) = \begin{cases} \{\mathtt{cca}\}, & \text{if } \chi \text{ is a code call atom } \mathtt{cca}; \\ \emptyset, & \text{if } \chi \text{ is a constraint atom}; \\ \mathit{CCA}(\chi_1) \cup \mathit{CCA}(\chi_2), & \text{if } \chi \text{ is a code call condition } \chi_1 \,\&\, \chi_2. \end{cases}$$

For any integrity constraint $ic : \psi \Rightarrow \chi_a$, define $\mathit{CCA}(ic) = \mathit{CCA}(\psi) \cup \mathit{CCA}(\chi_a)$.

$\mathit{Corr}(ic, \mathtt{cca})$ defined below describes the set of potentially corrupted code call atoms given that code call atom $\mathtt{cca}$ is corrupted.

Definition 3.3 ($\mathit{Corr}(ic, \mathtt{cca})$) *For any integrity constraint ic and code call atom $\mathtt{cca}$,*

$$\mathit{Corr}(ic, \mathtt{cca}) \;=\; \bigcup \left\{ \mathit{CCA}(ic\theta) \;\middle|\; \begin{array}{l} \mathtt{cca} \text{ and some } \mathtt{cca}' \in \mathit{CCA}(ic) \cup \sim\mathit{CCA}(ic) \\ \text{unify with most general unifier (mgu) } \theta \end{array} \right\}.$$

[4]By "corrupted" we mean that the current result of the code call atom may lead to an inconsistency in one or more integrity constraints. A code call atom could turn out to be corrupted either because an external entity has modified the state in an "uncontrolled" way, or due to a "propagation" of corruptedness, as described in Section 3.2.

If $\mathtt{cca}$ is considered corrupted, then each code call atom occurring in $ic\theta$ is considered corrupted as well. Notice that unifiers and most general unifiers (mgu's) θ are easily computed, since there are no nested terms.

Example 3.2 (Corruptedness) *Let us consider the integrity constraint*

$$ic : \mathtt{in}(\mathtt{X}, \mathtt{cc1}) \,\&\, \mathtt{in}(\mathtt{X}, \mathtt{cc2}) \Rightarrow \mathtt{in}(\mathtt{X}, \mathtt{cc3}).$$

Then we have $\mathit{CCA}(ic) = \{\mathtt{in}(\mathtt{X}, \mathtt{cc1}), \mathtt{in}(\mathtt{X}, \mathtt{cc2}), \mathtt{in}(\mathtt{X}, \mathtt{cc3})\}$ *and, furthermore,* $\mathit{Corr}(ic, \mathtt{in}(\mathtt{p}, \mathtt{cc1})) = \{\mathtt{in}(\mathtt{p}, \mathtt{cc1}), \mathtt{in}(\mathtt{p}, \mathtt{cc2}), \mathtt{in}(\mathtt{p}, \mathtt{cc3})\}$.

We may now define the procedure $\mathsf{corrcca}(X)$ which computes, given a set X of code call atoms considered corrupted, the set of all code call atoms considered corrupted as follows.

proc $\mathsf{corrcca}(X$: set of code call atoms$)$: set of code call atoms;

$old := \emptyset$; $new := X$;
while $new \neq old$ **do**
  $old := new$;
  for each $ic \in \mathcal{IC}$, $\mathtt{cca} \in old$ **do**
    $new := new \cup \mathit{Corr}(ic, \mathtt{cca})$;
endwhile;
return old.

end proc

Notice that $\mathsf{corrcca}$ implements a monotone, inflationary operator over the set of code call atoms, and terminates on finite input X. Furthermore, the output can be compacted by removing subsumed code call atoms from new.

The reason why we have to iteratively apply the *Corr* operator in the above procedure is because errors might be masked. For illustration, consider the following four integrity constraints:

$$ic_1 : \mathtt{in}(\mathtt{X}, \mathtt{cc1}) \,\&\, \mathtt{in}(\mathtt{Y}, \mathtt{cc2}) \Rightarrow \mathtt{X} = \mathtt{Y},$$

$$ic_2 : \mathtt{in}(\mathtt{W}, \mathtt{cc3}) \,\&\, \mathtt{W} = \mathtt{c} \Rightarrow \mathtt{in}(\mathtt{a}, \mathtt{cc1}),$$

$$ic_3 : \mathtt{in}(\mathtt{Z}, \mathtt{cc4}) \Rightarrow \mathtt{Z} > 8,$$

$$ic_4 : \mathtt{in}(\mathtt{Z}, \mathtt{cc4}) \,\&\, \mathtt{in}(\mathtt{J}, \mathtt{cc5}) \Rightarrow \mathtt{Z} < \mathtt{J}.$$

Suppose that in the current state all and only the following code call atoms are true:

$$\mathtt{in}(\mathtt{a}, \mathtt{cc1}),\ \mathtt{in}(\mathtt{b}, \mathtt{cc2}),\ \mathtt{in}(\mathtt{c}, \mathtt{cc3}),\ \mathtt{in}(\mathtt{10}, \mathtt{cc4}),\ \text{and } \mathtt{in}(\mathtt{20}, \mathtt{cc5}).$$

In the current state ic_1 is violated. Then, both $\mathtt{in}(\mathtt{a}, \mathtt{cc1})$ and $\mathtt{in}(\mathtt{b}, \mathtt{cc2})$ are potentially corrupted, since their evaluation returns a result which causes a violation of an integrity constraint; at least one of them reflects a condition on the current state which is not coherent with the agent's setting. The other integrity constraints are not violated in the current state. As we know that $\mathtt{in}(\mathtt{a}, \mathtt{cc1})$ is potentially corrupted, its correct evaluation may well have been `false` rather than `true` (though this is not necessary !). If in fact $\mathtt{in}(\mathtt{a}, \mathtt{cc1})$'s correct evaluation should have been `false`, then it may well be the case that $\mathtt{in}(\mathtt{c}, \mathtt{cc3})$ is also corrupted. This is because $\mathtt{in}(\mathtt{c}, \mathtt{cc3})$ should evaluate to `false` in order to satisfy ic_2.

We do not know whether $\mathtt{in}(\mathtt{a}, \mathtt{cc1})$ or $\mathtt{in}(\mathtt{b}, \mathtt{cc2})$ is the cause of the violation. Hence, we cannot exclude the possibility that the problem is with $\mathtt{in}(\mathtt{a}, \mathtt{cc1})$ and that it propagates to $\mathtt{in}(\mathtt{c}, \mathtt{cc3})$. Thus, to be on the safe side, we consider an integrity constraint (potentially) corrupted whenever it contains a potentially corrupted code call.

The integrity constraints ic_3 and ic_4 are not violated in the current state, and there is no reason to suspect that the code call atoms appearing in them are corrupted. This is because they are completely unrelated to the corrupted atoms.

The soundness of this approach is expressed by the following proposition which states that a coherent state can be reached from an incoherent one only by changing the return values of (some) corrupted code call atoms, and by maintaining the return values of the uncorrupted ones.

For any agent state $\mathcal{O}$, let $\mathcal{VGI}(\mathcal{O})$ be the set of ground instances of integrity constraints from $\mathcal{IC}$ which are violated in the state $\mathcal{O}$, and let $\mathcal{CGI}(\mathcal{O}) = \bigcup_{ic \in \mathcal{VGI}} \mathit{CCA}(ic)$ be the set of code call atoms in $\mathcal{VGI}(\mathcal{O})$.

Proposition 3.1 *Let Y be any set of code call atoms such that* $\mathsf{corrcca}(\mathcal{CGI}(\mathcal{O})) \lhd Y$*. Then, there exists an agent state $\mathcal{O}'$ such that $\mathcal{O}' \models \mathcal{IC}$ and, for any ground code call atom* $\mathtt{cca}$*, $\mathcal{O}$ and $\mathcal{O}'$ differ on* $\mathtt{cca}$ *only if* $\mathtt{cca} \lhd Y$*.*

This means that $\mathcal{O}$ can be turned into $\mathcal{O}'$ by modifying the return result for some corrupted code call atoms, and without changing the results of non-corrupted code call atoms.

Proof. We define a suitable $\mathcal{O}'$ as follows. Recall that at least one agent state exists which satisfies all integrity constraints, and let $\mathcal{O}_0$ be an arbitrary such agent state. For any ground code call atom $\mathtt{cca}$, we define

$$\mathcal{O}' \models \mathtt{cca} \;\Leftrightarrow\; \begin{cases} \mathcal{O}_0 \models \mathtt{cca}, & \text{if } \mathtt{cca} \lhd Y; \\ \mathcal{O} \models \mathtt{cca}, & \text{otherwise.} \end{cases}$$

Notice that $\mathcal{O}'$ is well-defined, and differs from $\mathcal{O}$ only on ground cca's which are subsumed by Y. Let ic be any ground instance of some integrity constraint in $\mathcal{IC}$. Then, one of the following two cases applies:

(1) There exists some $\mathtt{cca} \in \mathit{CCA}(ic)$ such that $\mathtt{cca} \lhd Y$. Then, by definition of $\mathsf{corrcca}$, $\mathit{CCA}(ic) \lhd Y$ holds. Hence, for each $\mathtt{cca} \in \mathit{CCA}(ic)$, we have $\mathcal{O}' \models \mathtt{cca}$ iff $\mathcal{O}_0 \models \mathtt{cca}$. Since $\mathcal{O}_0 \models ic$, it follows $\mathcal{O}' \models ic$.

(2) For no $\mathtt{cca} \in \mathit{CCA}(ic)$ it holds that $\mathtt{cca} \lhd Y$. This implies $\mathcal{CGI}(\mathcal{O}) \cap \mathit{CCA}(ic) = \emptyset$; hence, $\mathcal{O} \models ic$. Similarly, we conclude that $\mathcal{O}' \models \mathtt{cca}$ iff $\mathcal{O} \models \mathtt{cca}$. It follows $\mathcal{O}' \models ic$.

Hence, in both cases $\mathcal{O}' \models ic$. Therefore, $\mathcal{O}' \models \mathcal{IC}$, which proves the result. ∎

To continue the previous example, let us consider the state where the code call atoms

$$\mathtt{in}(\mathtt{a}, \mathtt{cc1}),\ \mathtt{in}(\mathtt{c}, \mathtt{cc3}),\ \mathtt{in}(10, \mathtt{cc4}),\ \text{ and } \mathtt{in}(20, \mathtt{cc5})$$

are true and all the other code call atoms are false. This is a consistent state, and we can reach it by simply changing the return value of $\mathtt{cc2}$ so that $\mathtt{in}(\mathtt{b}, \mathtt{cc2})$ becomes false.

The implementation of $\mathsf{corrcca}(X)$ which we have described is cautious and considers, in general, a larger set of code call atoms corrupted than may be semantically necessary. By applying a case by case distinction, we could get a refined picture in which a minimal set of code calls is identified as (potentially) corrupted. In the example above, $\mathtt{in}(\mathtt{c}, \mathtt{cc3})$ is viewed as corrupted, as well as $\mathtt{in}(\mathtt{a}, \mathtt{cc1})$, but we were able to reach a coherent state without changing the values of *all* these code call atoms. Unfortunately, computing a minimal set of code call atoms which need to be changed leads to intractability, which is the gist of the following result.

Theorem 3.2 *Given the sets $\mathcal{GI}$ and $\mathcal{VGI}(\mathcal{O})$ of ground and violated ground integrity constraints in the current agent state $\mathcal{O}$, respectively, and a ground code call atom* $\mathtt{cca}$*, deciding whether* $\mathtt{cca}$ *is in some smallest (w.r.t. inclusion) set of ground cca's X such that, by changing values of cca's in X only, a consistent state $\mathcal{O}'$ results is* NP*-hard.*

Proof. (Sketch) A variant of the satisfiability problem can be reduced to this problem. Suppose $C = \{C_1, \ldots, C_m\}$ is a set of clauses $C_i = L_{i,1} \lor L_{i,2} \lor L_{i,3}$ where each $L_{i,j}$ is a propositional atom a or its negation $\neg a$. The software package $\mathcal{S}$ maintains truth assignments to propositional atoms, and the API $\mathtt{tvars}()$ returns all variables set to true. Suppose a_0 is a distinguished atom such that an assignment in which a_0 is true satisfies C iff all other atoms are false. Now let $\mathcal{O}$ be the agent state in which all atoms are true, and set up for each clause C_i an integrity constraint $\sim\tau(L_{i,1})\ \&\ \sim\tau(L_{i,2}) \Rightarrow \tau(L_{i,3})$ where $\tau(L_{i,j}) = \mathtt{in}(\mathtt{a}, \mathtt{tvars}())$ if $L_{i,j} = a$ and $\tau(L_{i,j}) = \mathtt{notin}(\mathtt{a}, \mathtt{tvars}())$ if $L_{i,j} = \neg a$. Then, some of these integrity constraints are violated by $\mathcal{O}$. The cca $\mathtt{in}(\mathtt{a}_0, \mathtt{tvars}())$ belongs to some smallest change of ground code call atoms X that turns $\mathcal{O}$ into a consistent state $\mathcal{O}'$ iff C has a satisfying assignment in which a_0 is false. Since deciding the latter, under the above assumption, is NP-hard, and since $\mathcal{GI}$ and $\mathcal{VGI}(\mathcal{O})$ are easily constructed in polynomial time, the result follows. ∎

3.3 Suspicious code call atoms

Changing an appropriate subset of the ground instances of corrupted code call atoms will recover the agent to an "uncorrupted" state. This will be done in the agent cycle by a repair procedure. However, while this repair is going on, some message(s) might arrive. Rather than simply queuing the message(s) until the agent has recovered, it should:

1. find out whether processing the message interferes with the repair process, and
2. proceed with handling it if this is not the case.

For this purpose, we introduce the notion of "affected" action atom and rule, and the notion of "suspicious" code calls. Informally, the evaluation of an action atom is affected by a repair if it accesses a code call atom which is possibly changed by the repair process. The deontic status (is it permitted? forbidden? to be done? etc) of an action atom might change after the repair is completed. This also might have an impact on other action atoms whose deontic status is determined by running the agent program. In particular, a rule in the program that involves an affected action atom or a corrupted code call atom might propagate affectedness to other action atoms. The code call atoms in the body of such a rule are considered "suspicious" because they allow an affected rule to fire.

If we treat at least all corrupted code calls as being suspicious, then any unsuspicious code call may be safely evaluated in the current agent state. This is because (i) it is not affected by whatever corrupted the state and (ii) it will not be affected by any attempt to repair the corrupted part of the state. Hence, unsuspicious atoms may be safely evaluated even during the repair process. In particular, if the agent processes a message $\mathsf{ask}(\mathbf{b}, \mathtt{cca})$, say, during which it naturally evaluates the code call atom $\mathtt{cca}$, then the processing of this message does not interfere with the repair of the state as long as $\mathtt{cca}$ is unsuspicious. On the other hand, if $\mathtt{cca}$ is suspicious, then processing of a message should be delayed to avoid potentially incorrect results. A similar rationale applies when processing messages of the form $\mathsf{tell}(\mathbf{b}, \mathtt{cca}, \mathbf{ans})$.

As in the case of corrupted code call atoms, we determine suspicious code call atoms by a syntactic analysis of the agent program. We define a procedure $\mathsf{suscca}(X)$ which takes as input, a set X of code call atoms which subsumes all corrupted ground code call atoms of agent $\mathbf{a}$ and returns, as output, a set of suspicious code call atoms. The procedure operates in two phases. In the first phase, it determines what code call atoms are corrupted. In the second phase, it backward propagates possible integrity constraint violations that may arise after the completed repair.

We first define direct affectedness of an action atom by a code call atom.

Definition 3.4 (Directly Affected Action Atom) *An action atom $\alpha(\vec{t})$ is* directly θ-affected *by some code call atom* `cca`*, if there exists a* $\mathtt{cca}' \in \mathit{CCA}(\mathit{Pre}(\alpha(\vec{\mathtt{t}}))) \cup {\sim}\mathit{CCA}(\mathit{Pre}(\alpha(\vec{\mathtt{t}})))$ *which unifies with* `cca` *via mgu θ. We say that $\alpha(\vec{t})$ is directly affected if it is directly θ-affected for some θ.*

Informally, $\alpha(\vec{t})$ is directly θ-affected, if the status evaluation of its ground instances involves overlaps with the ground instances of the code call atom `cca`. If `cca` is corrupted, the value of the precondition of $\alpha(\vec{t})$ might change by the repair.

We next define affectedness of action atoms from a rule, given sets of affected action and code call atoms.

Definition 3.5 (Affected Rule and Action Atom) *Let*

$$r : A \leftarrow \chi \,\&\, L_1 \,\&\, \cdots \,\&\, L_n$$

be a rule, and let $AC(r)$ be the set of all action atoms occurring in r. Let X and Y be sets of action and code call atoms, respectively. Then r is θ-affected by X, Y if either

1. *some* $\mathtt{cca} \in \mathit{CCA}(\chi) \cup {\sim}\mathit{CCA}(\chi)$ *unifies with some* $\mathtt{cca}' \in Y$ *with mgu θ, or*
2. $\alpha(\vec{t}) \in AC(r)$ *is directly θ-affected by some* $\mathtt{cca}' \in Y$*, or*
3. $\alpha(\vec{t}) \in AC(r)$ *unifies with some* $\alpha'(\vec{t}') \in X$ *with mgu θ.*

The set $\Theta \mathit{AFF}(r, X, Y)$ is the union of all $AC(r\theta)$ such that r is θ-affected by X, Y. The set $\Theta \mathit{CCA}(r, X, Y)$ is the union of all $\mathit{CCA}(r\theta)$ such that r is θ-affected by X, Y. The rule r is affected by X, Y, if it is θ-affected for some θ. We define $\mathit{AFF}(r, X, Y) = AC(r)$ if such a θ exists and $\mathit{AFF}(r, X, Y) = \emptyset$ otherwise.

Informally, the affectedness set $\Theta \mathit{AFF}(r, X, Y)$ contains the actions atoms into which the affectedness of the actions atoms in X propagates, assuming that the code call atoms in Y are corrupted. Clearly, $\mathit{AFF}(r, X, Y)$ subsumes $\Theta \mathit{AFF}(r, X, Y)$ and takes a coarser view in which more ground atoms are affected, which we may choose for simplicity or efficiency.

We remark that by taking the particular semantics applied to an agent program into account, the definition of $\Theta \mathit{AFF}(r, X, Y)$ may be further refined. For instance, in the case of reasonable status set semantics [26], only the action atom of $A\theta$ needs to be added to $\Theta \mathit{AFF}(r, X, Y)$ if $\alpha(\vec{t})$ is from the body of r.

Example 3.3 (Affectedness) *Consider an agent which manages the advertisement policy of a department store by classifying customers as high, medium or low spenders. This classification may be used to send appropriate advertisements to customers (clearly, in practice more sophisticated classifications could be applied). A rule in the agent program could be:*

$$\begin{array}{rcl} r : \mathbf{Do}\,(\mathtt{high\_spender(C)}) & \leftarrow & \mathtt{in(C, oracle : select(person, sal, >, 100000))}\& \\ & & \mathbf{Do}\,(\mathtt{new\_customer(C)}) \end{array}$$

This rule says that when a new customer is entered into the database, she is assumed to be a high-spender customer if she has a high salary. The pre, add, and del lists of `new_customer` *are*

$Pre(\texttt{new\_customer(P)}) = \mathtt{notin(P,oracle:all(customers))}$,

$Add(\texttt{new\_customer(P)}) = \mathtt{in(P,\ oracle:all(customers))}$,

$Del(\texttt{new\_customer(P)}) = \emptyset$.

Some examples of θ-affectedness of r for pairs X, Y are:

1. $\langle \emptyset, \{\texttt{in}(\texttt{mary}, \texttt{oracle}:\texttt{select}(\texttt{person}, \texttt{sal}, >, \texttt{100000}))\}\rangle$*: The code call atom in Y unifies with* $\texttt{in}(\texttt{C}, \texttt{oracle}:\texttt{select}(\texttt{person}, \texttt{sal}, >, \texttt{100000}))$ *under mgu* $\theta = \{C = \texttt{mary}\}$.

2. $\langle \emptyset, \{\texttt{in}(\texttt{george}, \texttt{oracle}:\texttt{all}(\texttt{customers}))\}\rangle$*:* $\alpha(\vec{t}) = \texttt{new\_customer}(\texttt{C})$ *is directly θ-affected by Y since its code call atom unifies with* $\sim CCA(Pre(\alpha(\vec{\texttt{t}}))) = \{\texttt{in}(\texttt{C}, \texttt{oracle}:\texttt{all}(\texttt{customers}))\}$ *under mgu* $\theta = \{C = \texttt{george}\}$.

3. $\langle \{\texttt{high\_spender}(\texttt{steve})\}, \emptyset\rangle$*: the action atom unifies with the one in the head of r under mgu* $\theta = \{C = \texttt{steve}\}$.

The above notions help us to determine which action atoms may be affected when building the status set of the agent. Depending on the semantics applied, however, there are different ways to include an action status atom into a status set:

- Under rational and reasonable status set semantics, an action status atom $Op(\alpha)$ may only belong to a status set S if it occurs in a rule, or if it is derived by some action or deontic closure rule (cf. Def. A.3 in the appendix);
- under feasible status set semantics, any $Op(\alpha)$ may be included (even in some cases where it occurs in no rule).

We respect this by assuming that in the latter case, the program $\mathcal{P}$ contains dummy rules $\mathbf{P}(\alpha(\vec{X})) \leftarrow \mathbf{P}(\alpha(\vec{X}))$ for every action name α. Such rules can easily be added without changing the semantics of the program. We are now in a position to define how to compute a set of suspicious code call atoms from a given set of code call atoms known to be suspicious — the procedure $\texttt{suscca}(X)$ defined in Table 1 does this.

Informally, suspicious code call atoms are determined as follows. In Phase 1 of the procedure, we iteratively determine which code call atoms are affected by syntactically examining the rules of the agent program and starting with the knowledge that the code call atoms in the input to the algorithm are known to be corrupted. The code call atoms in the body of each rule which is found to be affected become suspicious. At the end of Phase 1, all code call atoms *possibly* affected by the corrupted code call atoms are determined — as this might lead to the agent taking actions which vary dramatically from what the agent developer originally intended, these code call atoms may have unintended consequences that need to be addressed. Specifically, these corrupted code call conditions might trigger unintended actions and this needs to be taken care of.

We further have to take into account the fact that such an action α might interfere with some other (yet unconsidered) action β through an integrity constraint, i.e., some effects of α and β occur together in an integrity constraint. In such a case, the joint execution of α and β might not be possible. If, on the corrupted state, β were executed, then on the repaired state β could no longer be executed if α must be executed in it.

We illustrate this by an example. Suppose the add list of α contains the code call atom $\texttt{in}(\texttt{a}, \texttt{cca1})$, while the add list of β contains $\texttt{in}(\texttt{b}, \texttt{cca2})$, and there is an integrity constraint ic : $\texttt{in}(\texttt{a}, \texttt{cca1}) \Rightarrow$

proc suscca(X: set of code call atoms) : set of code call atoms;
/* X `subsumes all corrupted ground code call atoms` */

/* `Phase 1: propagation of corruptedness` */
$old := \emptyset$; $S := X$; $A := \emptyset$;
while $S \cup A \neq old$ **do**
  $old := S \cup A$;
  for each $r \in \mathcal{P}$ **do**
    if $\Theta AFF(r, A, X) \neq \emptyset$ **then**
    begin $S := S \cup \Theta CCA(r, A, X)$;
      $A := A \cup \Theta AFF(r, A, X)$;
    end;
endwhile;
/* `Find action atoms which may cause troubles with IC` */
$B := \emptyset$;
for each $\alpha(\vec{t}) \in A$ **do**
  $C := \bigcup \left\{ CCA(ic\theta) \;\middle|\; \begin{array}{l} ic \in \mathcal{IC},\ \text{some } \mathtt{cca} \in CCA(Add(\alpha)\,\&\,Del(\alpha)) \text{ and} \\ \mathtt{cca}' \in CCA(ic) \cup {\sim}CCA(ic) \text{ unify with mgu } \theta \end{array} \right\}$;
  $B := B \cup \left\{ \beta(\vec{X}\theta) \;\middle|\; \begin{array}{l} \text{some } \mathtt{cca} \in CCA(Add(\beta(\vec{\mathtt{X}}))\,\&\,Del(\beta(\vec{\mathtt{X}}))) \\ \text{and } \mathtt{cca}' \in C \cup {\sim}C \text{ unify with mgu } \theta \end{array} \right\}$;
endfor;
/* `Phase 2: back propagate poss. IC-violation by atoms` B */
$old := S$;
while $S \cup B \neq old$ **do**
  $old := S \cup B$;
  for each $r \in \mathcal{P}$ **do**
    if $\Theta AFF(r, B, \emptyset) \neq \emptyset$ **then**
    begin $S := S \cup \Theta CCA(r, B, \emptyset)$;
      $B := B \cup \Theta AFF(r, B, \emptyset)$;
    end;
endwhile;
/* `return` S `plus precond's of affected action atoms in` B */
return $S \cup \{\mathtt{cca} \mid \mathtt{cca} \in CCA(Pre(\alpha(\vec{\mathtt{t}}))) \wedge \alpha(\vec{t}) \in B\}$.

end proc

Table 1: Procedure suscca

$\mathtt{notin}(\mathtt{b},\mathtt{cca2})$. Assume that in the current (corrupted) state, both $\mathtt{in}(\mathtt{a},\mathtt{cca1})$ and $\mathtt{in}(\mathtt{b},\mathtt{cca2})$ are false, and that α is not executed but β is, where ic is not an incriminated integrity constraint involving corrupted code call atoms. Furthermore, suppose that in the repaired agent state, α is executed. Then β could not be executed simultaneously unless ic is violated. Hence, in the repaired state, the agent would compute a status set according to which β is not executed. But this means that as for the status of $\mathtt{in}(\mathtt{b},\mathtt{cca2})$, the action taken by the agent on the corrupted state is (possibly) different from the one taken on the repaired state, which is undesired.

To eliminate such cases, the procedure suscca computes action atoms $\beta(\vec{X}\theta)$ which could lead to this problem. In Phase 2, it then computes action atoms which may be used in a derivation of these action atoms. This is done by analyzing in which rules of the program such atoms occur. Here $\Theta AFF(r, B, \emptyset)$ means that some $\alpha(\vec{t}) \in B$ unifies with some action atom $\beta(\vec{t'})$ in the body of rule r. No suspicious code call atoms in the rule body need to be considered since in this analysis, the effects of possible changes of the result of a code call atom are not relevant (they have already been considered earlier in Phase 1). Nonetheless, as in Phase 1, the code call atoms in affected rules become suspicious, since their value might contribute to deriving a problematic action atom.

After the back propagation, we take care of the fact that for an action atom $\alpha(\vec{t}) \in B$ the code calls in $Pre(\alpha)$ might be evaluated when computing the status set. Thus, all these code calls are also considered to be suspicious if α was found to be affected.

Example 3.4 (Suspiciousness) *Let us consider a simple agent* s_ag *that has the integrity constraint ic from Example 3.2. Suppose the agent program consists of the following rules, and rational status semantics is applied:*

$$r1 : \mathbf{Do}\,(\mathtt{a_1}(\mathtt{X})) \leftarrow \mathtt{in}(\mathtt{X},\mathtt{cc1}) \,\&\, \mathtt{X} \neq \mathtt{q} \,\&\, \mathbf{F}(\mathtt{a_2}(\mathtt{q})),$$

$$r2 : \mathbf{F}(\mathtt{a_2}(\mathtt{q})) \leftarrow \mathbf{P}(\mathtt{a_1}(\mathtt{q})),$$

$$r3 : \mathbf{F}(\mathtt{a_2}(\mathtt{s})) \leftarrow \mathtt{in}(\mathtt{s},\mathtt{cc3}) \,\&\, \neg\mathbf{Do}\,(\mathtt{a_3}),$$

$$r4 : \mathbf{Do}\,(\mathtt{a_4}) \leftarrow \mathtt{notin}(\mathtt{q},\mathtt{cc3}).$$

Let $Pre(\mathtt{a_i}(\mathtt{X})) = \{\mathtt{in}(\mathtt{X},\mathtt{cci})\}$, $Add(\mathtt{a_i}(\mathtt{X})) = \{\mathtt{in}(\mathtt{q},\mathtt{cci})\}$, and $Del(\mathtt{a_i}(\mathtt{X})) = \{\mathtt{in}(\mathtt{X},\mathtt{cci})\}$, for $i \in \{1,2\}$, and furthermore $Pre(\mathtt{a_j}) = \{\mathtt{in}(\mathtt{q},\mathtt{ccj})\}$, $Add(\mathtt{a_j}) = \emptyset$, and $Del(\mathtt{a_j}) = \{\mathtt{in}(\mathtt{q},\mathtt{ccj})\}$, for $j \in \{3,4\}$.

Suppose we are told that $\mathtt{in}(\mathtt{p},\mathtt{cc1})$ is corrupted, and we want to find out the suspicious code call atoms given this information. As already seen, $Corr(ic, \mathtt{in}(\mathtt{p},\mathtt{cc1})) = \{\mathtt{in}(\mathtt{p},\mathtt{cc1}), \mathtt{in}(\mathtt{p},\mathtt{cc2}), \mathtt{in}(\mathtt{p},\mathtt{cc3})\}$.

Let us call $\mathsf{suscca}(X)$ with $X = Corr(ic, \mathtt{in}(\mathtt{p},\mathtt{cc1}))$. We iteratively augment the initial sets $S := Corr(ic, \mathtt{in}(\mathtt{p},\mathtt{cc1}))$ and $A := \emptyset$ until we reach a fixpoint.

1. *In the first iteration, rule $r1$ is θ-affected for $\theta = \{X = p\}$, and we add to A the action atoms $\mathtt{a_1}(\mathtt{p})$ and $\mathtt{a_2}(\mathtt{q})$. The set S remains unchanged, since code call atom $\mathtt{in}(\mathtt{X},\mathtt{cc1})\theta = \mathtt{in}(\mathtt{p},\mathtt{cc1})$ from the body of $r1\theta$ already occurs in S. Rule $r2$ is now affected since because $\mathtt{a_2}(\mathtt{q})$ from A occurs in its head; thus, the action atom $\mathtt{a_1}(\mathtt{q})$ is added to A, while S remains unchanged. Rules $r3$ and $r4$ are not affected.*

2. *In the second iteration, rule $r1$ is newly affected for $\theta = \emptyset$, because $\mathtt{a_2}(\mathtt{q})$ from S occurs in its body, and for $\theta = \{X = q\}$, since $\mathtt{a_1}(\mathtt{q})$ unifies with the atom in its head. As a consequence, $\mathtt{a_1}(\mathtt{X})$ is newly added to A, and $\mathtt{in}(\mathtt{X},\mathtt{cc1})$ and $\mathtt{in}(\mathtt{q},\mathtt{cc1})$ are added to S. Rule $r2$ is not newly affected, and no further rule is affected.*

A further iteration brings no change, and phase 1 of $\texttt{suscca}(X)$ *terminates. We have* $S = \{\texttt{in}(\texttt{p},\texttt{cc1}), \texttt{in}(\texttt{p},\texttt{cc2}), \texttt{in}(\texttt{p},\texttt{cc3}), \texttt{in}(\texttt{X},\texttt{cc1}), \texttt{in}(\texttt{q},\texttt{cc1})\}$ *and* $A = \{\, \texttt{a}_1(\texttt{X}), \texttt{a}_1(\texttt{p}), \texttt{a}_1(\texttt{q}), \texttt{a}_2(\texttt{q}) \,\}$.

In computing B*, we have* $C := CCA(ic)$ *for* $\alpha(\vec{t}) = \texttt{a}_1(\texttt{X})$*, since* $\texttt{a}_1(\texttt{X})$*'s delete list contains* $\texttt{in}(\texttt{X},\texttt{cc1})$*, which occurs in* ic*. Thus,* B *is set to* $\{\, \texttt{a}_1(\texttt{X}), \texttt{a}_2(\texttt{X}), \texttt{a}_3 \,\}$ *on the next line. The further actions in* A *only add subsumed actions to* B*; we obtain* $B = \{\, \texttt{a}_1(\texttt{X}), \texttt{a}_1(\texttt{p}), \texttt{a}_1(\texttt{q}), \texttt{a}_2(\texttt{X}), \texttt{a}_2(\texttt{p}), \texttt{a}_2(\texttt{q}), \texttt{a}_3 \,\}$.

Phase 2 of $\texttt{suscca}(X)$ *then looks for the rules which are affected by* $(B, \emptyset)$*. Note that the only way for a rule to be affected by* $(B, \emptyset)$ *is to contain an action status atom unifying with an action status atom in* B.

1. $r1$ *is affected by* $(B, \emptyset)$*, but nothing new is added to* S *and* B.
2. *Also* $r2$ *is affected by* $(B, \emptyset)$*, but nothing new is added to* S *and* B.
3. $r3$ *is affected by* $(B, \emptyset)$*, because* $\texttt{a}_2(\texttt{X})$ *unifies with its head for* $\theta = \{X = s\}$*. Thus,* $\texttt{in}(\texttt{s},\texttt{cc3})$ *is added to* S *and* $\texttt{a}_2(\texttt{s})$ *is added to* B.
4. $r4$ *is not affected by* $(B, \emptyset)$.

The while loop terminates; we have $\texttt{S} = \{\, \texttt{in}(\texttt{p},\texttt{cc1}), \texttt{in}(\texttt{p},\texttt{cc2}), \texttt{in}(\texttt{p},\texttt{cc3}), \texttt{in}(\texttt{X},\texttt{cc1}), \texttt{in}(\texttt{q},\texttt{cc1}), \texttt{in}(\texttt{s},\texttt{cc3})\}$*. The return value of S, evaluated adding the preconditions of action status atoms in B, is* $S = \{\, \texttt{in}(\texttt{p},\texttt{cc1}), \texttt{in}(\texttt{p},\texttt{cc2}), \texttt{in}(\texttt{p},\texttt{cc3}), \texttt{in}(\texttt{X},\texttt{cc1}), \texttt{in}(\texttt{q},\texttt{cc1}), \texttt{in}(\texttt{s},\texttt{cc3}), \texttt{in}(\texttt{q},\texttt{cc2}), \texttt{in}(\texttt{q},\texttt{cc3}), \texttt{in}(\texttt{X},\texttt{cc2}), \texttt{in}(\texttt{s},\texttt{cc2}) \,\}$*. Omitting subsumed code call atoms, the result is* $S = \{\, \texttt{in}(\texttt{X},\texttt{cc1}), \texttt{in}(\texttt{X},\texttt{cc2}), \texttt{in}(\texttt{p},\texttt{cc3}), \texttt{in}(\texttt{q},\texttt{cc3}), \texttt{in}(\texttt{s},\texttt{cc3}) \,\}$.

The following theorem states that the procedure $\texttt{suscca}(X)$ — where X is an input set of code call atoms — returns, as output, a set Y of code call atoms having the following property: If an arbitrary code call atom cca (or its complement) is not unifiable with any code call atom in Y, then decisions based on cca are not affected by ongoing attempts to repair the code call atoms in X. That is, action decisions and resulting state changes that involve cca are isolated from the corrupted code call atoms, and would be the same if the state were repaired before running the agent program.

For example, if agent **a** should reply to a message $\texttt{ask}(\mathbf{b}, \texttt{in}(\texttt{jeff}, \texttt{db}:\texttt{persons}))$ querying a table persons, it might do so if the corrupted code call atoms are restricted to $\texttt{in}(\texttt{X}, \texttt{db}:\texttt{cars})$ where cars is a different table which is currently being repaired, provided that answering this message doesn't refer to cars.

We need some preliminary definitions. For a (fixed) agent program $\mathcal{P}$ and a given ground code call atom cca, the *influence set* IS of cca is the smallest set of (ground) actions that contains (1) all actions directly affected by cca and (2) all actions in $AC(r)$ where $r : A \leftarrow \chi \& \ L_1 \& \ldots \& L_n$ is any ground instance of a rule in $\mathcal{P}$ such that either $\texttt{cca} \in CCA(\chi)$ or $AC(r) \cap IS \neq \emptyset$. The influence set of an arbitrary code call atom, denoted $IS(\texttt{cca})$, is the union of all $IS(\texttt{cca}')$ where $\texttt{cca}'$ is a ground instance of cca.

Theorem 3.3 *Let* $\mathcal{O}$ *be an agent state and let* $\mathcal{O}_r$ *be a repair of* $\mathcal{O}$*. Let* X *be any set of code call atoms such that* $\texttt{corrcca}(\mathcal{CGI}(\mathcal{O})) \lhd X$*. Suppose* cca *is a code call atom not unifiable with any* $\texttt{cca}' \in \texttt{suscca}(X)$ *nor* $\sim\texttt{cca}'$*, and suppose* S *is a valid status set on* $\mathcal{O}$ *disregarding* $\mathcal{VGI}(\mathcal{O})$*. Then there exists a valid status set* S' *w.r.t.* $\mathcal{O}_r$ *and* $\mathcal{IC}$ *such that* $Op(\alpha) \in S'$ *iff* $Op(\alpha) \in S$ *holds for all modalities* Op *and* $\alpha \in IS(\texttt{cca})$.

Proof. By our assumption, some status set S' exists on $\mathcal{O}'$, leading to a state $\mathcal{O}'_r = \mathbf{conc}(\mathbf{Do}\,(S'), \mathcal{O}_r)$. It holds that no action $\alpha(\vec{t}) \in IS(\texttt{cca})$ belongs to $IS(\texttt{cca}')$ for any ground code call $\texttt{cca}'$ on which $\mathcal{O}$ and $\mathcal{O}_r$ are different. Otherwise, since $\texttt{cca}'$ must be a corrupted code call atom, $\texttt{cca}'$ is subsumed by X, and by virtue of Phase 1 of suscca, it follows that cca would have an instance which is subsumed by $\texttt{suscca}(X)$.

This is in contradiction to the hypothesis on $\mathtt{cca}$. Thus, the value of a status atom $Op(\alpha(\vec{t}))$ in the status sets S and S' is computable by accessing only (1) ground code call atoms on which $\mathcal{O}$ and $\mathcal{O}_r$ coincide, and (2) using only other action status atoms $Op'(\alpha'(\vec{t}'))$ such that $\alpha'(\vec{t}') \notin IS(\mathtt{cca}')$ for every ground code call atom $\mathtt{cca}'$ on which $\mathcal{O}$ and $\mathcal{O}_r$ are different.

Let AF be the set of all (ground) actions which instantiate action atoms in the sets A and B computed by $\mathsf{suscca}(X)$. We define the status set S'' by

$$S'' := \{Op(\alpha) \in S' \mid \alpha \in AF) \cup \{Op(\alpha) \in S \mid \alpha \notin AF\}.$$

That is, for affected actions we take the status from S' and for non-affected actions from S. We show that S'' is a Sem-status set on $\mathcal{O}_r$, leading to a state $\mathcal{O}''_r = \mathbf{conc}(\mathbf{Do}\,(S''), \mathcal{O}_r)$. Since it coincides with S on the $IS(\mathtt{cca})$, the result follows.

We first show that S'' is a feasible status set, i.e., satisfies conditions ($S1$)–($S4$) of Def. A.4. The key fact is that every ground instance r of a rule in $\mathcal{P}$ satisfies either $AC(r) \subseteq AF$ or $AC(r) \cap AF = \emptyset$

Since S and S' satisfies all rules of $\mathcal{P}$, it is thus clear that also S'' satisfies each rule of $\mathcal{P}$. Hence, condition ($S1$) is satisfied. Since, for any ground action α, all action status atom $Op(\alpha)$ in S'' belong either to S or to S' and S, S' are feasible status sets, it is clear that S'' satisfies the conditions ($S2$) and ($S3$).

As for ($S4$), a case analysis yields that every ground instance ic of an integrity constraint in $\mathcal{IC}$ is satisfied by $\mathcal{O}''_r$: (i) Assume first that $CCA(ic)$ contains some corrupted code call. Then all code calls in ic are corrupted, and only actions α where in $\alpha \in AF$ may change the value of any these code calls. Thus, for no such action $\mathbf{Do}\,(\alpha)$ can belong to $S \setminus S'$. Since $\mathcal{O}'_r \models ic$, it follows that also $\mathcal{O}''_r \models ic$. (ii) Assume next that no code call in ic is corrupted, but some action $\alpha \in AF$ specifies a change of some code call in ic. Then, by Phase 2 in procedure $\mathsf{suscca}(X)$, we have $\beta \in AF$ for every action β that specifies a change of some code call atom in ic. Again, since $\mathcal{O}'_r \models ic$ it follows that $\mathcal{O}''_r \models ic$. (iii) If neither (i) nor (ii) applies, then every code call atom in ic is uncorrupted and may be changed only by actions $\alpha \notin AF$. Since (i) does not apply, ic is not violated in state $\mathcal{O}$, and thus $\mathbf{conc}(\mathbf{Do}\,(S), \mathcal{O}) \models ic$. It follows that $\mathcal{O}''_r \models ic$. Summarizing, we have that ic is satisfied in the state $\mathcal{O}''_r$. Hence, $\mathcal{O}''_r \models \mathcal{IC}$, and thus condition ($S4$) is satisfied. This shows that S'' is a feasible status set w.r.t. $\mathcal{O}_r$.

If Sem is rational status set semantics, we must further show that S'' is grounded, i.e., no proper subset $T'' \subset S''$ satisfies ($S1$–$S3$). Suppose such a T'' exists; we shall derive a contradiction. Assume first that T'' is smaller than S'' on the action status atoms set over actions $\alpha \notin AF$. Then,

$$T := \{Op(\alpha) \in S \mid \alpha \in AF\} \cup \{Op(\alpha) \in T'' \mid \alpha \notin AF\}$$

is a smaller status set $T \subset S$ which satisfies ($S1$)–($S3$) on state $\mathcal{O}$: Indeed, note that each ground instance of rule satisfies either $AC(r) \subseteq AF$ or $AC(r) \cap AF = \emptyset$, and obviously T is deontically and action consistent and action closed. This would mean that S is not a rational status set on $\mathcal{O}$ (disregarding $\mathcal{VGI}(\mathcal{O})$), which is a contradiction to the hypothesis. Hence, T'' must coincide with S'' w.r.t. the status of actions not in AF, and thus T'' is smaller w.r.t. AF. Then, the status set

$$T' := \{Op(\alpha) \in T'' \mid \alpha \in AF\} \cup \{Op(\alpha) \in S' \mid \alpha \notin AF\}$$

is a smaller status set $T' \subset S'$ which satisfies, by similar arguments, ($S1$)–($S3$) on state $\mathcal{O}_r$. This means that S' is not a rational status set on $\mathcal{O}_r$, which is a contradiction. Thus, such a T'' can not exist, which proves that S'' is indeed a rational status set.

If Sem is reasonable status set semantics, we must show that S'' is a rational status set of the reduct $\mathcal{P}' = red^{S''}(\mathcal{P}, \mathcal{O}')$. In fact, since every reasonable status set is also rational, S'' is w.r.t. $\mathcal{O}_r$ a feasible status set for $\mathcal{P}$ and thus also for $\mathcal{P}'$. Observe that the reduct preserves the key property that either $AC(r) \subseteq AF$ or $AC(r) \cap AF = \emptyset$. By similar arguments as above, we thus obtain that S'' is grounded for $\mathcal{P}'$. Consequently, S'' is a reasonable status set of $\mathcal{P}$ w.r.t. $\mathcal{O}_r$. ∎

In particular, this formal result assures us that in case an agent program admits a single status set in each state (which, e.g., is true for the *IMPACT* target class of regular agent programs [15]), then the actions taken in reply to a message *must* also be taken if the corrupted state were repaired before.

4 Agent State Repair

Recall that an agent's state is characterized by the contents of its data structures. In order for an agent to automatically handle integrity constraint violations (or lack of a status set), we will add a special set of data structures to each agent called repair *data structures*. These data structures will have their own specialized API function calls.

4.1 The repair data structures

The repair data structures contain:

1. A buffer waitbuf consisting of messages that are waiting to be serviced because they involve accesses to part of the agent state that is "corrupted."
2. A buffer repbuf consisting of corrupted code call atoms.
3. A buffer icbuf consisting of (instances of) integrity constraints that are currently undergoing repairs.
4. An auxiliary buffer susbuf which contains the suspicious code call atoms.
5. A set cons_state consisting of all ground code call atoms true in a distinguished consistent state.
6. A set curr_state consisting of all ground code call atoms true in the current state.

The repair data structures support the following API functions:

suspicious(cca) : This function takes a code call atom cca as input, and returns true if cca is implied by the set of code call atoms contained in repbuf under a notion of inference fixed by the concrete implementation of the function. There are many ways to implement suspicious(cca). For example, it may:

  1. check whether cca is physically present in repbuf or
  2. check whether cca is an instance of a code call atom in repbuf, or
  3. check whether cca is implied by repbuf using some set of axioms and some set of implication rules.

suscca(X): This is the procedure defined in Section 3.3.

$\mathsf{add\_repbuf}(\mathtt{cca})$: This function "inserts" the code call atoms corrupted by $\mathtt{cca}$ into the repair buffer, such that after insertion, $\mathsf{suspicious}(\mathtt{cca}')$ returns `true` if $\mathtt{cca}'$ is from $\mathsf{suscca}(\mathsf{repbuf})$, and returns `false` otherwise. Its implementation depends on the one of $\mathsf{suspicious}(\mathtt{cca})$, and different possibilities exist (see Section 5.1).

It is important to note that the repair data structures and API calls can be included as part of the *IMPACT* agent development environment (see [15, 26]) and do not need to be programmed over and over again for each agent by the agent developer.

4.2 Repair action library

In addition to the repair data structures, we augment the agent with a set of "repair" actions. Each agent has a set of actions that may be used to "repair" the agent state. The repair actions can be implemented as a straightforward extensible dynamic linked library (DLL) provided by the *IMPACT* agent development environment.

Definition 4.1 *Suppose* $\mathbf{a}$ *is an agent and* $\mathcal{O}, \mathcal{O}'$ *are two states of agent* $\mathbf{a}$*. Let* $\mathcal{RA}$ *be the repair action library of agent* $\mathbf{a}$*. Then* $\mathcal{O}'$ *is said to be:*

1. $\mathcal{RA}_0$-reachable *from* $\mathcal{O}$ *iff* $\mathcal{O} = \mathcal{O}'$*,*
2. $\mathcal{RA}_{i+1}$-reachable *from* $\mathcal{O}$*,* $i \geq 0$*, iff there is a state* $\mathcal{O}''$ *such that* $\mathcal{O}''$ *is* $\mathcal{RA}_i$*-reachable from* $\mathcal{O}$ *and there is an action* α *in* $\mathcal{RA}$ *which is executable in* $\mathcal{O}''$ *and the execution yields* $\mathcal{O}'$*.*

State $\mathcal{O}'$ *is* $\mathcal{RA}$-reachable *from* $\mathcal{O}$ *iff* $\mathcal{O}'$ *is* $\mathcal{RA}_i$*-reachable from* $\mathcal{O}$*, for some* $i \geq 0$*.*

Intuitively, when we say a state $\mathcal{O}'$ is $\mathcal{RA}$ reachable from a given state $\mathcal{O}$, this means that there is a sequence of repair actions which allow $\mathcal{O}$ to be transformed into $\mathcal{O}'$. The following example illustrates this.

Example 4.1 (Simple Grid Scenario) *Let us consider a simple scenario where a* grid *agent manages three robots moving on an* $n \times n$ *grid,* $n \geq 2$*. The repair actions* $\mathcal{RA}_{grid}$ *are composed of the actions for moving a robot in one direction (north, south, east, west). We describe the* $\mathtt{go\_north}$ *action; the others are similar. We assume that the underlying software has a* $\mathtt{Pos(Robot)}$ *API function which may returns the position of the specified robot at the time the function call is made.*

Name*:* $\mathtt{go\_north}$

Schema*:* $\mathtt{(Robot)}$

$Pre(\mathtt{go\_north}) = \mathtt{in}(\mathtt{P}, \mathtt{grid} : \mathtt{Pos}(\mathtt{Robot})) \,\&\, \mathtt{P.y} \neq \mathtt{n}$

$Add(\mathtt{go\_north}) = \mathtt{in}(\mathtt{P}', \mathtt{grid} : \mathtt{Pos}(\mathtt{Robot})) \,\&\, \mathtt{P}'.\mathtt{x} = \mathtt{P}'.\mathtt{x} \,\&\, \mathtt{P}'.\mathtt{y} = \mathtt{P.y} + 1$

$Del(\mathtt{go\_north}) = \mathtt{in}(\mathtt{P}, \mathtt{grid} : \mathtt{Pos}(\mathtt{Robot}))$

Let $\mathcal{O}$ *be the state*

$$\mathcal{O} = \{\mathtt{in}((0,0), \mathtt{grid} : \mathtt{Pos(r1)}),\ \mathtt{in}((0,0), \mathtt{grid} : \mathtt{Pos(r2)}),\ \mathtt{in}((0,0), \mathtt{grid} : \mathtt{Pos(r3)})\}.$$

Then

$$\mathcal{O}' = \{\mathtt{in}((0,1), \mathtt{grid} : \mathtt{Pos(r1)}),\ \mathtt{in}((0,0), \mathtt{grid} : \mathtt{Pos(r2)}),\ \mathtt{in}((0,0), \mathtt{grid} : \mathtt{Pos(r3)})\}$$

is $\mathcal{RA}_1$-reachable from $\mathcal{O}$, while

$$\mathcal{O}'' = \{\texttt{in}((0,1), \texttt{grid} : \texttt{Pos}(\texttt{r1})),\ \texttt{in}((0,0), \texttt{grid} : \texttt{Pos}(\texttt{r2})),\ \texttt{in}((1,1), \texttt{grid} : \texttt{Pos}(\texttt{r3}))\}$$

is $\mathcal{RA}_3$-reachable from $\mathcal{O}$. Both $\mathcal{O}'$ and $\mathcal{O}''$ are $\mathcal{RA}$-reachable from $\mathcal{O}$.

Definition 4.2 *A set $\mathcal{RA}$ of repair actions is said to be* complete *w.r.t. an agent state $\mathcal{O}$ iff there exists an $\mathcal{RA}$-reachable state $\mathcal{O}'$ such that $\mathcal{O}' \models \mathcal{IC}$. Furthermore, $\mathcal{RA}$ is said to be* complete *w.r.t. an agent* $\mathbf{a}$*, iff $\mathcal{RA}$ is complete w.r.t. $\mathcal{O}$ for* every *state $\mathcal{O}$ of* $\mathbf{a}$.

Intuitively, $\mathcal{RA}$ is complete for an agent iff whatever possible state the agent is in, there is always some way of executing repair actions so that a consistent (w.r.t. integrity constraints) agent state is obtained. When an agent developer specifies his or her repair actions, it is critical that they be complete w.r.t. the rest of the agent.

Example 4.2 (Grid Scenario Continued) *Suppose that in the previous scenario an integrity constraint exists stating that a position can be occupied by at most one robot.*

$\mathcal{RA}_{grid}$ is complete *w.r.t. $\mathcal{O}$, since there exists a state ($\mathcal{O}''$) which is $\mathcal{RA}_{grid}$-reachable from $\mathcal{O}$ and satisfies the integrity constraints. $\mathcal{RA}_{grid}$ is also* complete *w.r.t. agent* grid*, it is alway possible to move, in any agent state, the robots in such a way that they occupy three different positions.*

The set of repair actions in the grid example is domain-dependent. In order to provide the system developer with already defined strategies, we propose some domain-independent sets of repair actions which can be adopted whatever the context is. They use the repair data structures introduced in Section 4.1.

Example 4.3 (Initialized State Repair Actions $\mathcal{RA}_{init}$) *We assume that an agent* $\mathbf{a}$ *is in an initial state $\mathcal{O}_{init}$ at the time of deployment. $\mathcal{O}_{init}$ is assumed to satisfy the integrity constraints. We may then set* cons_state-*set* = $\mathcal{O}_{init}$*. Then $\mathcal{RA} = \{\texttt{ra}\}$ is complete w.r.t. $\mathcal{O}_{init}$ if we define action* `ra` *to be defined as follows:*

Name: `ra`

Schema: ()

$Pre(\texttt{ra}) = \{\texttt{in}(\texttt{O}, \texttt{repair} : \texttt{curr\_state}())\ \&\ \texttt{in}(\texttt{I}, \texttt{repair} : \texttt{cons\_state}())\}$

$Add(\texttt{ra}) = \{\texttt{in}(\texttt{I}, \texttt{repair} : \texttt{curr\_state}())\}$

$Del(\texttt{ra}) = \{\texttt{in}(\texttt{O}, \texttt{repair} : \texttt{curr\_state}())\}$

Here, two functions `curr_state`() *and* `cons_state`() *are used which are provided by the repair package. The former returns, as output, the set of all ground code call atoms which are true in the current state, and the latter the set of all ground code call atoms which are true in a distinguished state $\mathcal{O}_0$ that satisfies the integrity constraints (see Section 2).*

The action `ra` *can be applied in any state: we assume it is always possible to retrieve the ground code call atoms characterizing the current state ($\texttt{in}(\texttt{O}, \texttt{repair}:\texttt{curr\_state}())$) and the consistent state $\mathcal{O}_0$ ($\texttt{in}(\texttt{I}, \texttt{repair}:\texttt{cons\_state}())$). Then, the current state is changed to $\mathcal{O}_0$*

Example 4.4 (Preferred State Repair Actions $\mathcal{RA}_{pref}$) *Preferred state repair actions are exactly like the above except that the agent developer initializes the* cons_state*-set with a state $\mathcal{O}_{pref}$ which is known to satisfy the integrity constraints.*

Example 4.5 (Rollback Repair Actions $\mathcal{RA}_{roll}$) *In rollback-based repair, at any given instant t of time, the agent tracks its last known consistent state $\mathcal{O}_{lk}$ and sets* cons_state*-set equal to $\mathcal{O}_{lk}$. This is done by the* mkrepair *function which identifies the proper repair actions to perform for reaching a consistent state, and updates* cons_state *accordingly. When integrity constraints are violated, repairs cause the agent state to be reset to the last known consistent state. Thus, the set of repair actions consists of the single action* `ra`*, which is exactly like that in Examples 4.3 and 4.4. What is different, though, is the content of* cons_state*, which dynamically changes during the agent's life cycle. This strategy is usable only when actions are reversible (e.g., an agent that executes a* `fax` *action will probably find it impossible to recall the fax).*

Example 4.6 (IC-Oriented Repair $\mathcal{RA}_{ic}$) *This repair strategy can be applied under the condition that each integrity constraint with a comparison atom in the head has at least one code call atom in its body. Suppose* **a** *is an agent having integrity constraints $ic_i : \psi_i \Rightarrow \chi_i$, where $i \in \{1, \ldots, n\}$. We now construct repair actions* $\mathtt{ra_i}$ *for them:*

> **Name***:* $\mathtt{ra_i}$
>
> **Schema***:* ()
>
> $Pre(\mathtt{ra_i}) = \emptyset$
>
> $Add(\mathtt{ra_i}) = \{\chi_i\}$*, if* χ_i *is a code call atom, and* $Add(\mathtt{ra_i}) = \emptyset$ *otherwise.*
>
> $Del(\mathtt{ra_i}) = \emptyset$*, if* χ_i *is a code call atom, and* $Del(\mathtt{ra_i}) = \{\mathtt{cca}\}$*, for some* $\mathtt{cca} \in CCA(\psi_i)$ *otherwise.*

Example 4.7 (IC-Repair with Protected Atoms $\mathcal{RA}_{icp}$) *A slight variant of the preceding strategy, called $\mathcal{RA}_{icp}$, may include a list of "protected" code call atoms. The repair actions* $\mathtt{ra_i}$ *are similar except that* $\mathtt{ra_i}$*'s delete list may contain only non-protected code call atoms if χ_i is a comparison atom. Prior to deployment of an agent, the system must check that each integrity constraint with a comparison atom in the head has at least one non-protected code call atom in its body.*

The following results give us some idea about the difficulty of checking completeness. For concrete statements about complexity, we need some assumptions about the complexity of evaluating code calls and the domains of different data types. The assumptions we make are similar to those in the comprehensive analysis of the complexity of agent programs in [13], and request that the size of an agent state is bounded by a polynomial in the size of the problem input (e.g., this can be ensured by assuming that the number of arguments in code calls is bounded by a constant, and that the number of values is polynomial in the input size), and that each code call to an agent state can be evaluated in polynomial time. For further ramifying assumptions concerning state changes, we refer to [13].

Theorem 4.1 *1. Checking the completeness of a given set of repair actions $\mathcal{RA}$ w.r.t. a given agent state $\mathcal{O}$ is* PSPACE*-complete under the above assumptions and undecidable in general.*

2. Checking completeness of a given set of repair actions $\mathcal{RA}$ w.r.t. a given agent **a** *is* PSPACE*-complete under the above assumptions and undecidable in general.*

Proof. (Sketch) The PSPACE upper bound is a consequence of the fact that the size of the agent state is bounded by a polynomial. The PSPACE lower bounds are explained by the fact that Turing machines with polynomial work space can be easily encoded to this problem. However, checking completeness of a set of repair actions with respect to an agent is harder than checking w.r.t. an agent state. Even if the latter is polynomial, the completeness test w.r.t. an agent might be undecidable. This can be shown by reducing to this problem e.g. the one of deciding whether a given SQL-query returns true over all possible instances of a relational database, which is undecidable (cf. [1]). ∎

An extensible library of complete sets of repair actions may easily be incorporated within *IMPACT*. The agent developer - once she has specified her agent's integrity constraints, agent program, etc., can automatically select $\mathcal{RA}_{init}, \mathcal{RA}_{roll}, \mathcal{RA}_{ic}$ repair action strategies. In this case, the relevant repair actions may automatically be computed and filled in for the agent by the *IMPACT* Agent Development Environment.

5 Error Tolerant Agent Cycle

In this section, we specify a solution to the problem of how an agent can recover from corrupted states while continuing to process requests that are unaffected by ongoing repairs. Note that at any point in time, the agent's state may be under repair or not.

If it is not under repair and a message of the form $\mathsf{ask}(\cdot)$ or $\mathsf{tell}(\cdot)$ arrives, then we attempt to process the request as usual (nothing needs to be done to account for the repairs). Two possibilities now arise. Either the message yields a valid status set, or not. In the first case, we are done. Otherwise, we need to add the message to waitbuf and start repairing the state.

If, on the other hand, the agent's state is being repaired, we need to check whether ongoing repairs will interfere with processing of the current request. This can be done by checking whether the code call atom in the message is affected by the ongoing repairs. If so, we must add the message to the waitbuf buffer. Otherwise we can process it, secure in the knowledge that repairs being made to the agent state are not going to affect decisions depending on the current value of code call atom. Note that the two cases where the state is not under repair, and the state is under repair without affecting the incoming message, are both captured by the condition $\mathsf{suspicious}(\mathtt{cca}\theta) = \mathtt{false}$. In fact, if the state is not under repair, repbuf is empty, and nothing can be "derived" from it (in particular, $\mathtt{cca}\theta$ cannot be derived). Otherwise, repbuf is not empty but again $\mathtt{cca}\theta$ cannot be "derived" since it is not involved with the ongoing repairs. These two cases are dealt with uniformly. We need a few simple definitions.

Definition 5.1 (Sem- and Sem-Sem-Compatible Update) *Let $\mathcal{P}$ be an agent program. Then, an agent state $\mathcal{O}$ is Sem-compatible with $\mathcal{P}$, if* **a** *has a Sem-status set w.r.t. $\mathcal{O}$. Furthermore, $\mathcal{O}$ is Semi-Sem-compatible with $\mathcal{P}$, if it is not Sem-compatible but $\mathcal{P}$ has a status set w.r.t. $\mathcal{O}$ modulo condition $(S4)$ of a feasible status set (cf. Definition A.4 in Appendix A).*

We now show how the agent decision cycle given in [14, 26] may be modified so as to handle the requirements of recovery and continuity. The modified decision cycle, et_agent_cycle ("et" stands for error tolerant), defined in Table 2 uses a special procedure mkrepair that takes, as input, an agent state as well as repbuf, icbuf, waitbuf, and (i) assembles a list of ground action status atoms whose serial execution is guaranteed to change the agent state to one satisfying all integrity constraints and (ii) executes this list and causes that waitbuf is flushed, i.e., all buffered messages are handled.

Here, icbuf contains instances of violated integrity constraints, repbuf represents (perhaps a superset) of the set of corrupted code calls, and waitbuf contains messages which need to be serviced/handled. The

```
proc et_agent_cycle(a:agent; 𝒪:agent-state; msg:message);
if msg = ask(b, cca) or msg = tell(b, cca, ans) then
    if msg = ask(b, cca) then ans := {id}
                        /* take identity as dummy substitution:  ccaθ = cca if θ = id */
    if suspicious(ccaθ) = true for some θ ∈ ans then insert(waitbuf, msg);
      /* add affected msg to waitbuf (state is under repair) */
    else /* no state repair or doesn't affect msg */
        if some Sem-status set S w.r.t. ℐ𝒞 ⊖ icbuf exists on 𝒪 + msg then
            execute the action conc({α | Do (α) ∈ S}, 𝒪 + msg)
        else /* no status set exists - error condition */
            if some Semi-Sem-status set S′ exists on 𝒪 + msg then
            begin /* switch to new (corrupted) state; needs repair */
                𝒪′ := conc({α | Do (α) ∈ S′}, 𝒪 + msg);
                icbuf := ∅; repbuf := ∅;                          /* reinitialize buffers */
                for each instance ic′ of an ic ∈ ℐ𝒞 s.t. 𝒪′ ⊭ ic′ do
                    insert(icbuf, ic′);                      /* add int.cons. requiring repair */
                for each cca′ ∈ CCA(ic′) do add_repbuf(cca′);
                                     /* add poss. corrupted cc-atoms in ic′ to repbuf */
                insert(waitbuf, msg);
                mkrepair(𝒪′, repbuf, icbuf, waitbuf)
            end
else /* msg = done(b, cca, ans⁺, ans⁻) */
begin X := {ic | ic ∈ ℐ𝒞 ⊖ icbuf and 𝒪 + msg ⊭ ic};
    if X ≠ ∅ or no Sem-status set for 𝒪 + msg w.r.t. ℐ𝒞 ⊖ icbuf exists then
    begin add_repbuf(cca);
        for each ic ∈ X do insert(icbuf, ic);
        mkrepair(𝒪, repbuf, icbuf, waitbuf)
    end
    else compute a Sem-status set S for 𝒪 + msg w.r.t. ℐ𝒞 ⊖ icbuf and
           execute the action conc({α | Do (α) ∈ S}, 𝒪 + msg)
end
end proc
```

Table 2: Modified agent decision cycle

expression $\mathcal{IC} \ominus \mathsf{icbuf}$ denotes the set of all integrity constraints which are ground instances of some integrity constraint in $\mathcal{IC}$ but not in icbuf. Note that they can be described at the non-ground level. Furthermore, $\mathcal{O} + msg$ describes the agent state that updates the state $\mathcal{O}$ with the message msg.

Let us see how the above algorithm captures our requirements of Recovery and Continuity. Recovery is supported via (i) Steps 8-16 and (ii) Steps 19-23 of the algorithm, where (i) handles the case when a message that yields no valid status set is encountered, and (ii) is used when an external update causes integrity constraint violations.

Continuity is supported as well. In Step 6, execution of actions according to S can be safely done by Theorem 3.3, even though possible repairs are going on. In two cases, however, processing the message is deferred: In Step 3, when it is realized that the current state repair might interfere with the processing of the message, and in Step 12, after it is realized that the agent program *per se* violates some integrity constraints, and thus the state needs repair.

When the state has been repaired, the messages are flushed from the waitbuf buffer. That is, they are processed one by one and new status sets are computed.

5.1 Different methods to implement suspicious and add_repbuf

In this section, we propose a couple of alternative ways of implementing the functions suspicious and add_repbuf.

5.1.1 A first implementation

As mentioned earlier, there are many ways to implement add_repbuf and suspicious. One simple way is given below. It is important to note that add_repbuf and suspicious must be mutually compatible.

```
proc suspicious1(cca)
  if cca ◁ susbuf then return true
  else return false.
end proc

proc add_repbuf1(cca)
  repbuf := repbuf ∪ corrcca(cca)
  susbuf := suscca(repbuf)
end proc
```

When inserting a message's code call atom into repbuf, we compute all other corrupted code call atoms (using the function corrcca defined in Section 3.3) and add them to repbuf. Starting from repbuf we also evaluate the suspicious code call atoms and put them in an auxiliary buffer, susbuf.

This procedure has the advantage that when evaluating suspicious1, all that is needed is a simple subsumption check which is executable in time proportional to the product of the length of the table and the longest code call atom stored in it. However, it has the disadvantage that whenever a message is to be inserted, all corrupted and suspicious code calls must be computed. Hence, insertion is an expensive and space consuming operation. The use of suspicious1 and add_repbuf1 is appropriate if we expect the agent's state to be corrupted infrequently in comparison to the number of messages that can be processed without being concerned about corruption of the agent state.

5.1.2 A second implementation

Another implementation of suspicious and add_repbuf would work as follows. When a code call atom (in a message) causes problems, then we insert the code call atoms corrupted by it into repbuf without computing the suspicious code call atoms. Later, when a new message is received, we explicitly determine if it is affected using the suscca function.

```
proc suspicious2(cca)
   if cca ◁ suscca(repbuf) then
      return true
   else return false.
end proc

proc add_repbuf2(cca)
   if not (cca ◁ repbuf) then
      repbuf := repbuf ∪ corrcca(cca)
end proc
```

Unlike the first implementation, this one spends minor effort when inserting code call atoms into repbuf. However, for each arriving request, it attempts to check if that request is affected by the ongoing repairs. Thus, in using this application, we may find that repbuf is large (as lots of things are inserted into it) and hence the time for checking if a given request is affected by the ongoing repairs as in repbuf can be significant. Thus, this method is worth using if the number of repairs is large and there are few requests.

5.2 Implementing mkrepair

The mkrepair procedure (it is a procedure rather than a function in programming language terminology as it has side effects) takes as input, a current agent state $\mathcal{O}$ and values of repbuf, icbuf, and waitbuf. The procedure does the following:

1. It finds a state $\mathcal{O}_{new}$ that satisfies all the agent's integrity constraints (and in particular repairs those in icbuf).
2. It resets icbuf and repbuf to $\emptyset$ as the integrity constraints are now repaired and as the code calls causing problems are now no longer causing problems.
3. It then iteratively reinvokes et_agent_cycle with the messages in waitbuf (they will no longer trigger errors as the repairs that caused them to originally be placed in waitbuf are now fixed).
4. It resets waitbuf to $\emptyset$ as the waiting messages are now handled.

Steps (2)–(4) above are simple to handle and understand, and hence, in the rest of this section, we focus on step (1).

It is easy to see that Step (1) may be formulated as a classical AI planning problem. Specifically, we have a current state and a set of goal states (those where $\mathcal{IC}$ is satisfied) and a set $\mathcal{RA}$ of repair actions — we wish to find a sequence of (some) appropriate actions in $\mathcal{RA}$ that yield a goal state. When $\mathcal{RA}$ is a complete set of actions, it is possible that there are multiple consistent states that the agent can transition to. In this situation, the agent should transition to a "best" repair state w.r.t. some state evaluation function. This again is a classical AI planning problem [21]. Hence, in this section, we confine ourselves to specify how such a cost function to evaluate states may be set up. Solutions already proposed in the AI literature [21] may be easily adopted to actually find a "best" state w.r.t. such a cost function.

Definition 5.2 *A* state evaluation function, $\mathsf{sef}_{\mathbf{a}}(\mathcal{O})$, *associated with agent* **a** *is one that takes as input, an agent state* $\mathcal{O}$*, and provides as output, an integer.*

Definition 5.3 *A state* $\mathcal{O}'$ *of an agent* **a** *is* optimal *w.r.t. an agent state* $\mathcal{O}$ *iff*

1. $\mathcal{O}' \models \mathcal{IC}$,
2. $\mathcal{O}'$ *is* $\mathcal{RA}$*-reachable from* $\mathcal{O}$*, and*
3. *there is no other agent state* $\mathcal{O}^\star$ *satisfying 1 and 2 such that* $\mathsf{sef}_{\mathbf{a}}(\mathcal{O}') < \mathsf{sef}_{\mathbf{a}}(\mathcal{O}^\star)$.

Example 5.1 (Optimal Agent State) *Let us reconsider Example 4.1. We may set* $\mathsf{sef}(\mathcal{O})$ *to be the sum of the Hamming distances between the positions of the three robots. Suppose we have an API function "*$\mathtt{hdist}(\mathtt{P}_1,\mathtt{P}_2)$*" which computes the Hamming distance between two points* $\mathtt{P}_1,\mathtt{P}_2$ *in the integer plane. Then we can formally set*

$$\mathsf{sef}(\mathcal{O}) = \mathtt{hdist}(\mathtt{R}_1,\mathtt{R}_2) + \mathtt{hdist}(\mathtt{R}_2,\mathtt{R}_3) + \mathtt{hdist}(\mathtt{R}_3,\mathtt{R}_1).$$

The results of $\mathtt{in}(\mathtt{R}_1,\mathtt{grid}:\mathtt{Pos(r1)})$, $\mathtt{in}(\mathtt{R}_2,\mathtt{grid}:\mathtt{Pos(r2)})$, *and* $\mathtt{in}(\mathtt{R}_3,\mathtt{grid}:\mathtt{Pos(r3)})$ *describe the agent state. Assume a* 5×5 *grid and suppose the current agent state is*

$$\mathcal{O} = \{\mathtt{in}((0,1),\mathtt{grid}:\mathtt{Pos(r1)}),\ \mathtt{in}((0,0),\mathtt{grid}:\mathtt{Pos(r2)}),\ \mathtt{in}((1,1),\mathtt{grid}:\mathtt{Pos(r3)})\}.$$

Then,

$$\mathcal{O}' = \{\mathtt{in}((0,4),\mathtt{grid}:\mathtt{Pos(r1)}),\ \mathtt{in}((0,0),\mathtt{grid}:\mathtt{Pos(r2)}),\ \mathtt{in}((4,4),\mathtt{grid}:\mathtt{Pos(r3)})\}$$

is an optimal state w.r.t. $\mathcal{O}$ *as it satisfies* $grid$*'s integrity constraint, it can be reached from* $\mathcal{O}$ *through a series of actions from* { `go_north`, `go_south`, `go_east`, `go_west` } *and, as the robots are located on three corners of the grid, the sum of their Hamming distances is maximal. Note that more than one state may be optimal. For example,*

$$\mathcal{O}'' = \{\mathtt{in}((4,0),\mathtt{grid}:\mathtt{Pos(r1)}),\ \mathtt{in}((4,4),\mathtt{grid}:\mathtt{Pos(r2)}),\ \mathtt{in}((0,0),\mathtt{grid}:\mathtt{Pos(r3)})\}$$

is another optimal state w.r.t. $\mathcal{O}$.

The goal of mkrepair is to take a state $\mathcal{O}$ that violates the agent's integrity constraints, and to find a sequence of repair actions which yields an optimal state $\mathcal{O}'$.

This is easily seen to be an AI planning problem. However, there is one major difference. Whereas in AI planning problems, the cost of a plan is typically taken to be the sum of the costs (or some monotonic function of the costs) of the actions in the plan, in this case, the repair actions are not being assessed any cost. Instead, each state has an associated "value" captured by the state evaluation function, and we want to find a reachable state satisfying the integrity constraints that has the maximal value.

We now specify how we may define the value of a state.

Definition 5.4 (Variable Specification) *Suppose* χ *is a code call condition involving an integer variable* X*. Then* $X : \chi$ *is a* variable specification.

We assume the existence of a specialized package `math` which supports a number of standard arithmetic functions, including a binary "`sum`" operation on integers and a binary "`hdist`" function on points (pairs of integers).

Example 5.2 (Variable Specification) *The expression*

$$\mathtt{HDR_1R_2} : \mathtt{in}(\mathtt{R}_1,\mathtt{grid{:}Pos(r1)})\ \&\ \mathtt{in}(\mathtt{R}_2,\mathtt{grid{:}Pos(r2)})\ \&\ \mathtt{in}(\mathtt{HDR_1R_2},\mathtt{math{:}hdist}(\mathtt{R}_1,\mathtt{R}_2))$$

is a variable specification of $\mathtt{HDR_1R_2}$*, while*

$$\mathtt{V} : \mathtt{in}(\mathtt{Res}_1,\mathtt{math{:}sum}(\mathtt{HDR_1R_2},\mathtt{HDR_2R_3}))\ \&\ \mathtt{in}(\mathtt{V},\mathtt{math{:}sum}(\mathtt{Res}_1,\mathtt{HDR_3R_1}))$$

is a variable specification of `V`*. Their intended meaning is to specify the Hamming distance between Robot* `R1`, `R2` *and the sum of the three Hamming distances, respectively.*

Definition 5.5 (Math Code Call Conditions and Specifications) *A* math code call condition with input variables $\mathbf{X} = X_1, \ldots, X_n$ and output variable X *is a code call condition which is safe modulo* $\mathbf{X}$,[5] *contains* X *and involves only code call atoms accessing* `math`*. A* math variable specification with input variables $\mathbf{X}$ and output variable X *is a variable specification whose associated code call condition is a math code call condition with input variables* $\mathbf{X}$ *and output variable* X.

Example 5.3 *The variable specification*

$$\mathtt{V} \;:\; \mathtt{in}(\mathtt{Res}_1, \mathtt{math} : \mathtt{sum}(\mathtt{HDR}_1\mathtt{R}_2, \mathtt{HDR}_2\mathtt{R}_3)) \;\&\; \mathtt{in}(\mathtt{V}, \mathtt{math} : \mathtt{sum}(\mathtt{Res}_1, \mathtt{HDR}_3\mathtt{R}_1))$$

is a math variable specification with input variables $\mathtt{HDR}_1\mathtt{R}_2$, $\mathtt{HDR}_2\mathtt{R}_3$, $\mathtt{HDR}_3\mathtt{R}_1$ *and output variable* V.

Definition 5.6 (Objective Function Specification) *An* objective function specification *is given by a pair* $\langle X : \chi_{math}, \{VS_1, \ldots, VS_n\}\rangle$ *where:*

1. *each* VS_i *is a variable specification of the form* $X_i : \chi_i$*, and*

2. $X : \chi_{math}$ *is a math variable specification with input variables* $X_1, \ldots, X_n$ *and output variable* X.

Example 5.4 (Objective Function Specification) *We continue the grid example and describe an objective function which assigns higher values to states where the three robots are further apart. Such an objective function specification may look like* $\langle \mathtt{V} : \chi_{\mathtt{math}}\rangle, \{\mathtt{VS}_1, \mathtt{VS}_2, \mathtt{VS}_3\}$*, where*

$$\chi_{\mathtt{math}} = \mathtt{in}(\mathtt{Res}_1, \mathtt{math}\!:\!\mathtt{sum}(\mathtt{HDR}_1\mathtt{R}_2, \mathtt{HDR}_2\mathtt{R}_3)) \;\&\; \mathtt{in}(\mathtt{V}, \mathtt{math}\!:\!\mathtt{sum}(\mathtt{Res}_1, \mathtt{HDR}_3\mathtt{R}_1))$$

$$\begin{aligned}
\mathtt{VS}_1 &= \mathtt{HDR}_1\mathtt{R}_2 : && \mathtt{in}(\mathtt{R}_1, \mathtt{grid}\!:\!\mathtt{Pos(r1)}) \;\&\; \mathtt{in}(\mathtt{R}_2, \mathtt{grid}\!:\!\mathtt{Pos(r2)}) \;\&\\
& && \mathtt{in}(\mathtt{HDR}_1\mathtt{R}_2, \mathtt{math}\!:\!\mathtt{hdist}(\mathtt{R}_1, \mathtt{R}_2)),\\
\mathtt{VS}_2 &= \mathtt{HDR}_2\mathtt{R}_3 : && \mathtt{in}(\mathtt{R}_2, \mathtt{grid}\!:\!\mathtt{Pos(r2)}) \;\&\; \mathtt{in}(\mathtt{R}_3, \mathtt{grid}\!:\!\mathtt{Pos(r3)}) \;\&\\
& && \mathtt{in}(\mathtt{HDR}_2\mathtt{R}_3, \mathtt{math}\!:\!\mathtt{hdist}(\mathtt{R}_2, \mathtt{R}_3)),\\
\mathtt{VS}_3 &= \mathtt{HDR}_3\mathtt{R}_1 : && \mathtt{in}(\mathtt{R}_3, \mathtt{grid}\!:\!\mathtt{Pos(r3)}) \;\&\; \mathtt{in}(\mathtt{R}_1, \mathtt{grid}\!:\!\mathtt{Pos(r1)}) \;\&\\
& && \mathtt{in}(\mathtt{HDR}_3\mathtt{R}_1, \mathtt{math}\!:\!\mathtt{hdist}(\mathtt{R}_3, \mathtt{R}_1)).
\end{aligned}$$

Intuitively, an objective function specification measures the value of agent state $\mathcal{O}'$ by

1. setting $v_i = \max\{X_i\theta \mid \chi_i\theta \text{ is ground and } \mathcal{O}' \models \chi_i\theta\}$;

2. grounding out the values of the X_i's in χ_{math} and setting $v = X\gamma\theta \mid \chi_i\gamma\theta$ is ground and is true w.r.t. $\mathcal{O}'$, where $\gamma = \{X_i = v_i \mid 1 \leq i \leq n\}$;

3. returning v.

Example 5.5 (Value of the Objective Function) *We continue Example 5.1 by considering* $\mathcal{O}$, $\mathcal{O}'$ *and* $\mathcal{O}''$*. Then, for* $\mathcal{O}$ *we have that* $v_1 = 1, v_2 = 2, v_3 = 1$ *and thus* $v = 4$*. For* $\mathcal{O}'$*, instead, we have that* $v_1 = 4, v_2 = 8, v_3 = 4$ *and thus* $v = 16$*. The value of* $\mathcal{O}''$ *is also 16 since* $v_1 = 4, v_2 = 8, v_3 = 4$.

[5] Informally, this means that after assigning $X_1, \ldots, X_n$ values, the code call condition can be reordered so that an evaluation from left to right is possible (see Appendix B and [14, 26]).

6 Relevance to Other Agent Frameworks

In this paper, we have shown how to define an "error tolerant" agent decision cycle that can apply to *IMPACT* agents when they are corrupted. This decision cycle allows the agent to continue processing unaffected requests and conditions, while repairing the state so that affected requests may be processed effectively. A natural question to ask is how the results of this paper may be applied to other agent frameworks. In this section, we show how this may be done in the context of the following three agent frameworks: the Kowalski-Sadri agent framework [18, 20], the Belief-Desires-Intentionality framework due to Rao and Georgeff [22] and the rational agent framework due to Wooldridge [29]. For further frameworks, this is briefly discussed in Section 7.2.

6.1 Kowalski and Sadri's Unified Agent Architecture

Kowalski and Sadri [18, 19] analyze the similarities and differences between rational and reactive agent architectures and propose a unified architecture which aims to capture both as special cases. An agent's reasoning is captured via a proof procedure and a logic programming style search engine is used to reduce goals to subgoals in a "rational" manner. The complete proof reduction procedure given in the papers is based on the observation that in many cases it is possible to replace a goal G by an equivalent set of condition-action rules R. The problem of controlling the reasoning process so that it works correctly with bounded resources is also addressed.

The resulting cycle governing the architecture is the following:

1. observe any input coming from the environment at time T;
2. record all input;
3. resume the execution of proof procedure (applied to the current goal statement) by first propagating the input[6];
4. continue applying the proof procedure for a total of n inference steps;
5. select an atomic action respecting time constraints;
6. execute any such action and record the results.

The extension of such a cycle to take error-tolerance into account may appear complex at first glance because *integrity constraints* dynamically evolve during the execution of the cycle itself: goal reduction replaces goal statements with simpler goal statements which have the form of integrity constraints. In the case of *IMPACT* agents, the integrity constraints are established once for all, and we made the same assumption for our error-tolerant extension of the *BDI* architecture.

Fortunately, despite the use of the same phrase ("integrity constraint") Kowalski and Sadri's *integrity constraints* have a different meaning than ours: they just represent a condition to be checked on the current state, but they do not need to be necessarily satisfied. As shown in various examples from [18] and [19], the proof procedure continues to execute when their integrity constraints are not satisfied, leading to a new goal which takes the unsatisfied integrity constraints into account.

[6]The *propagation of input* replaces the current goal statement with a simpler one, taking into account the observed input and the integrity constraints characterizing the agent's behavior.

Thus, to avoid confusion, let us suppose that a set of *Static Integrity Constraints* are included in the knowledge base of Kowalski and Sadri's agents, and let us suppose that these *Static Integrity Constraints* have the same meaning as *IMPACT*'s *Integrity Constraints*: they are established once and for all, and if they are violated, a repair procedure must immediately start.

The Unified Agent Architecture cycle may now be modified as outlined below.

1. observe any input coming from the environment at time T;
2. record all input;
3. check if the new inputs cause some violation to the *Static Integrity Constraints*: if they are violated then start a repair procedure as a concurrent thread;
4. evaluate if resuming the proof procedure of the current goal statement leads to some conflict with the current repair procedure; if at least one *error-tolerant* atomic action (namely, an action which is unrelated to the current repairs) turns out to be executable
   (a) continue applying the proof procedure for a total of n inference steps;
   (b) select any *error-tolerant* atomic action respecting time constraints;
   (c) execute any such action and record the results.

   else,

   (a) interrupt processing inputs and complete the repair procedure.

The key obstacle in applying this definition is to determine what it means for a "conflict" to occur between the proof procedure and a repair procedure. This can be addressed in many ways. One way is to determine which atoms are affected by the repair procedure and which ones are affected by the proof procedure and if there is an intersection between the two sets, then declaring a conflict. A notion of affectedness similar to that in our paper can be used. An alternative solution is to simulate in advance what should occur by going on with the proof procedure and using some syntactic check.

6.2 Rao and Georgeff's BDI architecture

The *BDI* architecture [22] is based on the notion of agents as *intentional systems* [12]; for an excellent introduction, see [30]. The architecture is characterized by the following structures, as depicted in Figure 2:

- **beliefs**, which represent the knowledge of the agents;
- **goals**, which are beliefs, or conjunctions and disjunctions of beliefs, which must be achieved or tested in the current state;
- **plans**, which contain the procedural knowledge of agents. They are characterized by a trigger, a context, a body; a maintenance condition, a set of "success actions" and a set of "failure actions"; and
- **intentions**, which are partially instantiated plans.

A typical *BDI* engine is characterized by the following cycle

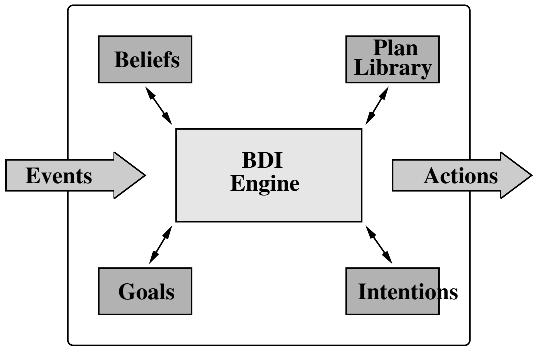

Figure 2: The *BDI* Architecture

1. observe the world and the agent's internal state, and subsequently update the *event queue*;

2. generate possible new plans whose trigger event matches an event in the event queue and whose context is satisfied;

3. select one from this set of matching plans for execution;

4. push the selected plan onto an existing or new *intention stack*, according to whether or not the event is a (sub)goal;

5. select an intention stack, take the topmost plan and execute the next step of this current plan: if the step is an action, perform it, otherwise, if it is a subgoal, post it on the event queue.

In order to extend this agent cycle to handle error-tolerance, we must modify steps 2, 3 and 5 of the above cycle. In particular, we must choose an event from the event queue only if it is safe to process it. Likewise, we must select a plan for execution only if it is safe to execute it, and we must select an intention stack only if the next step in its topmost plan can be safely executed.

Let us suppose that some integrity constraints hold while some other integrity constraints are currently violated, and a repair is being done on the current state to recover to a correct state. In this case, we may modify steps 2,3, and 5 of the *BDI* agent cycle as follows.

- **Step 2**: *When is it safe to choose an event from the event queue*? We say that an event is *error-tolerant* if there is at least one *error-tolerant* plan (see definition below) among the plans whose trigger event matches the chosen event and whose context is satisfied.

- **Step 3**: *When is it safe to select a plan for execution*? In order to decide if a plan is *error-tolerant* (namely, it can be safely executed without leading to inconsistencies due to the repairs which are being made on the state), it is necessary to evaluate the consequences of executing it (before actually executing it). If pushing the selected plan onto an existing (resp. new) intention stack, and selecting this newly modified (resp. created) stack[7] leads to executing an action which could be affected by some repair, the plan is not *error-tolerant*. Giving details of which kind of actions are affected by a repair is out of the scope of this paper. However, the good news is that our definitions of affectedness and corruptedness may be adapted (with some work) to the *BDI* framework.

[7]We can ignore the other intention stacks at this point, since they will be analyzed in step 5 of the cycle.

Even if an *error-tolerant* plan is chosen in step 3, it is possible that an *unsafe* intention stack is chosen in step 5. This may causes problems because the action to be performed may interfere current repairs. Thus, it is also necessary to consider error-tolerance of intention stacks.

- **Step 5**: *When is it safe to select an intention stack*? As in the previous case, here too, it is necessary to evaluate the consequences of executing the next step of the topmost plan in the stack (before actually doing so). If the action to execute is potentially affected by some repair, the intention stack cannot be selected (it is not *error-tolerant*). If at least one plan can be selected for execution in step 3, then there is at least one error-tolerant intention stack to chose (the one modified in step 3).

If at a certain moment there are no error-tolerant events in the event queue, the execution cycle must stop until the repairs have been completed. If no repairs are currently made, all the events in the event queue are error-tolerant. Given the definitions above, the "error-tolerant" *BDI* cycle is outlined below:

1. observe the world and the agent's internal state, and subsequently update the *event queue*. if some violation of the integrity constraints occurs, start a repair procedure as a concurrent thread;
2. if at least one error-tolerant event exists in the event queue
   (a) choose an error-tolerant event from the event queue;
   (b) generate possible new plans whose trigger event matches the chosen event and whose context is satisfied;
   (c) select one from this set of matching plans for execution, provided that the plan is error-tolerant;
   (d) push the selected plan onto an existing or new *intention stack*, according to whether or not the event is a (sub)goal;
   (e) select an error-tolerant intention stack, take the topmost plan and execute the next step of this current plan: if the step is an action, perform it, otherwise, if it is a subgoal, post it on the event queue.
3. else interrupt processing events and complete the repair procedure.

6.3 Wooldridge's Computational Multi-Agent System

In chapter 4 of his Ph.D. thesis, Wooldridge [29] gives a formal model intended to capture diverse aspects of a variety of agent systems. The model is based on some assumptions:

- agents have significant but finite computational resources;
- agents have a set of explicitly represented *beliefs* and are able to reason about these beliefs in accordance with the computational resources afforded to them;
- beliefs are expressed in some *logical* language;
- in addition to being believers, agents can *act*: in particular, they are capable of *communicative actions*;
- finally, agents are able to *revise* their beliefs by means of a belief revision function.

Each agent in the system continuously executes the following cycle:

1. interpret any message received;
2. update beliefs according to previous action and message interpretation;
3. derive deductive closure of belief set;
4. derive set of possible messages, choose one and send it;
5. derive set of possible actions, choose one and apply it.

Wooldridge defines two execution models for multi-agent systems: a synchronous model, and an asynchronous one. All agents in the synchronous model begin and end an execution cycle together. In the more realistic asynchronous model, where execution is interleaved, at most one agent is allowed to act at any fixed point of time.

A naive error-tolerant extension of his agent cycle may be defined as follows:

1. interpret any message received;
2. update beliefs according to previous action and message interpretation;
3. check if the new beliefs violate the agent's integrity constraints: if they are violated then start a repair procedure as a concurrent thread;
4. derive deductive closure of belief set: if the deductive closure does not contain beliefs which interfere with the integrity constraints under repair, then
   (a) derive set of *error-tolerant* messages, choose one and send it;
   (b) derive set of *error-tolerant* actions, choose one and apply it;
5. else interrupt processing received messages and complete the repair procedure.

As usual, we are assuming that each agent has a set of static integrity constraints to be satisfied in any state. By *error-tolerant* messages and actions we mean those messages and actions which do not interfere with the current repair procedure. Determining when a belief "interferes" with an ongoing repair may be defined by adapting the notion of affectedness given in our paper to the case of Wooldridge's syntax.

7 Related Work

To our knowledge, there has been no work on error tolerance in agent systems. As a consequence, we compare our work with related work in other areas. In the previous section, we have already shown how many of the ideas proposed in this paper for *IMPACT* agents also apply to other agent systems.

7.1 Inconsistency in databases

Sources of information and services are often required to satisfy integrity constraints (ICs). When the ICs are not satisfied, the source is in an inconsistent state and no interaction between the source and its users should take place until recovery from inconsistency has been completed. Though it is clear that excluding users from interacting with the "consistent part" of an inconsistent data source is beneficial in practice, research on providing consistent services over inconsistent data sources has not been as widespread.

Reasoning about inconsistent databases has been studied extensively in the context of "paraconsistent" databases, and in the cases of reasoning with multiple knowledge bases [6, 25, 5]. However, there was no notion of an agent decision cycle – for an agent that is a continuously running process to steadily execute requests even while corrupted, the decision cycle must be modified so that error tolerant processing methods can be incorporated into the decision cycle. This is one of the key contributions of this paper.

An important effort to deal with consistent query answering in information systems with inconsistent ICs was done by Bry [8]. He proposed an approach which makes it possible to recognize whether an answer to a query has been derived from possibly corrupted data. He exploits a notion of local inconsistency, formalized in terms of *minimal logic*[8]. Data which cause an IC violation is considered potentially corrupted. An answer which cannot be established (i.e., which is not derivable in minimal logic) without using some potentially corrupted data is called *inconsistent*. Conversely, an answer is *consistent* if it can be computed without using data involved in IC violation. He shows that minimal logic suffices as a foundation of query answering in positive, definite or disjunctive, deductive databases. However, the problem is not addressed in the context of an agent system that accesses external data sources via code calls and where rules involve actions and deontic modalities. Also, the way consistent and inconsistent answers should be computed is not addressed. This represents a significant difference between Bry's approach and ours, as we have provided algorithms to evaluate corrupted and affected items, and formally proved that these items correspond to the intuitive notion of "corruptedness" and "suspiciousness" resp. "affectedness".

Arenas, Bertossi, and Chomicki [2] provide a logical characterization of consistent query answering in relational databases that may be inconsistent with the given ICs. An answer to a query posed to a database that violates the ICs is "consistent," if it is the same as that obtained from any minimally repaired version of the database. A method for computing such answers and an analysis of their properties is provided: on the basis of a query Q, the method computes, using an iterative procedure, a new query $T_\omega(Q)$ whose evaluation in an arbitrary database (consistent or inconsistent) state returns the set of consistent answers to the original query Q. $T_\omega(Q)$ is based on the notion of *residue* in the context of semantic query optimization. The soundness of the approach is proven, as well as its completeness for particular ICs (*binary ICs*). Termination of computing T_ω is also guaranteed under proper conditions. A variant of the T_ω operator is described in [9], which is proven to be sound, terminating and complete for some classes of ICs extending those in [2]. In [3] the *Annotated Predicate Calculus* (APC) is adopted, a logic where inconsistent information does not unravel logical inference and where causes of inconsistencies can be reasoned about. The inconsistent database is embedded in APC which is then used to define database repairs and query answers. This approach has been used to help understand the results of [2] and to provide a more general algorithm that covers classes of queries beyond [2]. The main difference between the approach in [2] and ours is the way how consistent answers are evaluated. In fact, [2] *rewrites* a query so as to take into account the ICs, and then evaluates the answers of the rewritten query, which are proved to be consistent answers of the original one. What we do, instead, is to evaluate whether processing the incoming message involves "unsafe" data: if not, we process the message as it is (as shown, this yields in this case the same results as if the ICs would not be violated), otherwise we defer it. A further complication in our work is that when an external request is made of an agent, the agent state may get modified while the repairs are going on.

As for repair of constraint violations, an interesting approach has been proposed in [16], where basic concepts from model-based diagnosis are adopted to discover minimal sets of simultaneous reasons for

[8]Minimal logic is a constructivistic weakening of classical logic defined in terms of the natural deduction proof system by Gentzen, deprived of the absurdity rule.

violations of (different) constraints. These reasons indicate possible repair actions that guarantee elimination of violations. The adopted repair actions depend on the “repair strategy” which the user can choose. The proposed strategies are domain independent and range from minimal undo or consistent completion of a violating transaction up to user interaction with the repair process. A sound and complete algorithm for enumerating possible minimal repair transactions for an inconsistent database is also proposed. Our repair strategies are similar to those of [16] in that they allow the user to choose the strategy which is most suitable for her application from an application independent library of strategies, eventually specifying priorities or preferences for use during the repair. The interaction between the user and the repair process is briefly sketched in [16], assuming that a suitable environment exists. We believe that *IMPACT can be* such an environment as user interaction with the repair process can be easily performed in *IMPACT*’s multi-agent setting.

7.2 Agent frameworks

The problems tackled and solved in this paper, namely how to let an agent go on working even when its state is corrupted, and how to ensure that an agent recovers from a corrupted state to an uncorrupted state, are critical for the agent community. Agents find application in domains such as telecommunication [28], process control (e.g. [10]), electronic commerce and many others (see `http://agents.umbc.edu/`) where the reliability of a multiagent system is a key issue. In these domains, as well as many others, *continuity* and *recovery* properties should be supported so as to guarantee a high-quality service. Error tolerance is a must. We are not aware of any research on agent architectures, environments or formalisms which allow the development of error-tolerant agents. However, we believe that this is an important aspect in the endeavor of building rational agents, which should make good (but not necessarily perfect) decisions about what to do in any given situation [30]. In particular, if it turns out that there is an inconsistency, then the agent should still be able to go ahead and take decisions and actions which *seem to make sense*. Of course, there must be some underlying assumptions – for example, that the integrity constraints are correct. We could imagine that some integrity constraint is not correct, and withdrawal or modification of that integrity constraint could remedy the situation. However, if the agent has the choice between modifying, on the one hand, the agent state and, on the other hand, its integrity constraints, which are part of its specification, given that other agents or entities might have unprevented access to its state, the former seems to be more plausible to us. However, the agent designer could be informed of violations of the integrity constraints, and decide whether a change of the integrity constraints is needed.

Our notions of corruptedness and affectedness are auxiliary technical concepts which helped to formalize the intuition that actions which are unrelated to errors may be still executed; the peculiarities of the framework, however, make this a nontrivial task.

Fortunately, the results of the preceding section show how our techniques for error-tolerance may be adapted to different types of agent architectures. In addition, there are many other works in the agent community that are related to that proposed here. Shoham [24] was perhaps the first to propose an explicit programming language for agents, based on object oriented concepts, and based on the concept of an agent state. Shoham [24, Section 3] states that a complete AOP (“agent oriented programming”) system will have three components.

1. a restricted formal language for describing mental state;

2. an interpreted agent programming language with primitive commands such as `REQUEST` and `INFORM`; and,

3. an “agentifier”.

We have already shown, in [14], that *IMPACT* agents can express most of Shoham’s AOP framework. Hence, the results of this paper may be applied to Shoham’s AOP framework in this way.

Hindriks *et al.* [17] have developed a deontically based agent programming framework. In their framework, an agent’s mental state consists of a set of goals and a set of beliefs. An *agent program* in their framework consists of a quadruple $(\mathcal{T}, \Pi_0, \sigma_0, \Gamma)$ where $\mathcal{T}$ is a transition function specificying the effects of basic actions, Π_0 is an initial set of goals, σ_0 is an initial set of beliefs, and Γ is a set of rules of the form

$$Head \quad \leftarrow \quad Guard \,|\, Body.$$

In general, $Head$ is a (potentially) complex formula describing a goal. The syntax of goals supported by Hindriks *et al.*[17] allows goals to be elementary actions, but also includes sequential compositions of actions, disjunctive goals, and/or conjunctive goals. The $Guard$ is a logical formula, while the $Body$ has the same structure as the head. While not everything in Hindriks *et al.* [17] can be expressed in *IMPACT* (and vice versa), their agent decision cycle is very similar to ours, and hence, the results on error tolerance may be applied to their agent decision cycle in much the same way as it is applied in this paper to *IMPACT*’s agent cycle. This is also the case for agent decision making frameworks such as the initial frameworks of Rosenschein [23] who was perhaps the first to say that agents act according to states, and which actions they take are determined by rules of the form “When P is true of the state of the environment, then the agent should take action A.” Their decision cycle too, is similar to ours. When a state change occurs, determine what to do based on the rules involved. Hence, in such a decision cycle, our notions of affectedness can be directly used to only allow rules involving unaffected atoms to be used to process requests and the same repair mechanism proposed by us may be used to conduct repairs to the agent state. This also applies to the IRMA system by Bratman *et al.* [7], where the agent generates different possible courses of actions (Plans) based on the agent’s intentions. These plans are then evaluated to determine which ones are consistent and optimal with respect to achieving these intentions. A cycle similar to ours may be used there - in particular, only plans that involve “unaffected” atoms may be used.

8 Conclusion

Software agents provide a powerful new paradigm for distributed, collaborative, and mobile applications. Software agent systems that build on top of legacy software in a principled way, and that support automatic coupling of simple and complex actions to changes in their environment, have a wide variety of applications in e-commerce. Nonetheless, it is dangerous to assume that just because agents are prototyped using a declarative language such as in *IMPACT*, they will be free of errors. Prolog programs over the years have not been error-free. The history of programming has shown that bugs in code must always be accounted for.

In an *IMPACT* based agent system, and for that matter, in any agent system that builds on top of legacy code, bugs may arise for one of several reasons. First, the agent developer may have written rules that do not account for all possible states of the agent that arise. Second, the agent may not be in full control of its state — this is true in legacy applications where the agent is just one vehicle to access the legacy application’s state. Third, the legacy code on top of which the agent is being built may itself have bugs, causing unexpected agent states to arise.

In this paper, we have taken a modest first step toward addressing this extraordinarily difficult problem. Specifically, we have proposed for the first time (to our knowledge) an agent decision algorithm that has two good features. First, it incorporates a method for the agent to recover from a “corrupted” state to an "uncorrupted" one. Second, it allows the agent to continue processing requests during such a recovery/repair process, as long as such requests are unaffected by the ongoing repairs.

Our work may be seen as a contribution in the endeavor of building rational agents [30]. We have shown how our methods may be applied to various agent frameworks, and in particular to the BDI model. An important aspect is reasoning about the behavior of agents, which for the BDI model has been amply discussed and demonstrated by Wooldridge using $\mathcal{LORA}$ ($\mathcal{L}$ogic $\mathcal{O}$f $\mathcal{R}$ational $\mathcal{A}$gents) [30]. It remains an interesting issue to see how error-tolerance can be modeled in $\mathcal{LORA}$, or which extension is needed for that. Observe that $\mathcal{LORA}$ builds on top of classical logic; thus, if the agent state and integrity constraints would be modeled as sets of classical facts and axioms, from a violation of an integrity constraint we could conclude everything; this may be avoided using methods from paraconsistent logic or suitable belief operators.

Our contribution in this paper is admittedly not a panacea for all problems involving bugs in agent programs. It handles the case when agent's don't have status sets due to violation of integrity constraints. Such violations may occur because third parties are manipulating the agent's state without the agent having any veto on such updates. It also arises when the agent's rules are not adequate to deal with such IC violations. However, these scenarios only represent a small microcosm of the space of errors that can arise when agents are programmed. This forms a rich avenue for future research.

The results of this paper may be extended in future work in many different ways. For instance, rather than considering action atoms as affected, we could view action *status* atoms as affected, and determine suspicious code call atoms on the basis of a syntactic analysis of the agent programs similar as described in the this paper. Due to the interplay of the various semantics components of feasible status sets including deontic consistency, action closure and integrity constraints, this would provide a more refined approach. However, its study would also be substantially more complex.

Acknowledgments.

We thank our colleagues and reviewers for their constructive comments which helped to improve this paper.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[2] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proc. PODS'99*, pp. 68–79. ACM Press, 1999.

[3] M. Arenas, L. Bertossi, and M. Kifer. Applications of annotated predicate calculus to querying inconsistent databases. In J. Lloyd et al. (editors), *Proc. CL'2000/DOOD'2000*, pp. 926–941, LNCS 1861. Springer, 2000.

[4] K. Arisha, T. Eiter, S. Kraus, F. Ozcan, R. Ross, and V.S. Subrahmanian. IMPACT: A platform for collaborating agents. *IEEE Intelligent Systems*, 14(2):64–72, March/April 1999.

[5] C. Baral, S. Kraus, J. Minker, and V.S. Subrahmanian. Combining multiple knowledge bases consisting of first order theories. *Computational Intelligence*, 8(1):45–71, 1992.

[6] H. Blair and V.S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68:135–154, 1989.

[7] M. Bratman, D. Israel, and M. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(4):349–355, 1988.

[8] F. Bry. Query answering in information systems with integrity constraints. In S. Jajodia, W. List, G. McGregor, and L. Strous, editors, *Integrity and Internal Controls in Information Systems, vol. I: Increasing the confidence in information systems, Proceedings 1997 IFIP WG 11.5 Working Conference on Integrity and Control in Information Systems*. Chapman & Hall, December 1997.

[9] A. Celle and L. Bertossi. Querying inconsistent databases: algorithms and implementation. In J. Lloyd et al. (editors), *Proc. CL'2000/DOOD'2000*, pp. 942–956, LNCS 1861. Springer, 2000.

[10] J. M. Corera, I. Laresgoiti and N. R. Jennings. Using archon, part 2: Electricity transportation management. In *IEEE Expert*, 11(6):71-79, 1996.

[11] K. Decker, K. Sycara, and M. Williamson. Middle agents for the internet. In *Proc. 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pp. 578–583. Morgan Kaufmann, 1997.

[12] D. C. Dennet. *The Intentional Stance*. MIT Press, 1987.

[13] T. Eiter and V.S. Subrahmanian. Heterogeneous active agents, II: Algorithms and complexity. *Artificial Intelligence*, 108(1-2):257–307, 1999.

[14] T. Eiter, V.S. Subrahmanian, and G. Pick. Heterogeneous active agents, I: Semantics. *Artificial Intelligence*, 108(1-2):179–255, 1999.

[15] T. Eiter, V.S. Subrahmanian, and T. Rogers. Heterogeneous active agents, III: Polynomially implementable agents. *Artificial Intelligence*, 117(1):107–167, 2000.

[16] M. Gertz and U. Lipeck. An extensible framework for repairing constraint violations. In S. Jajodia, W. List, G. McGregor, and L. Strous, editors, *Integrity and Internal Controls in Information Systems, vol. I: Increasing the confidence in information systems, Proceedings 1997 IFIP WG 11.5 Working Conference on Integriy and Control in Information Systems*, pp. 89–111. Chapman & Hall, December 1997.

[17] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. J. C. Meyer. Formal semantics for an abstract agent programming language. In *Proc. International Workshop on Agent Theories, Architectures, and Languages (ATAL'97)*, LNCS/LNAI 1365, pp. 215–230. Springer, 1998.

[18] R. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In *Proc. International Workshop on Logic in Databases (LID'96)*, LNCS/LNAI 1154, pp. 137–149. Springer, 1996.

[19] R. Kowalski and F. Sadri. An agent architecture that unifies rationality with reactivity. Technical Report, Imperial College, London, UK, 1997.

[20] R. Kowalski and F. Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3/4):391–491, 1999.

[21] N. J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 1980.

[22] A. Rao and R. Georgeff. Modeling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proc. Second International Conference on Knowledge Representation and Reasoning (KR-91)*, pp. 473–484. Morgan Kaufmann Pub, 1991.

[23] S. J. Rosenschein. Formal theories of knowledge in AI and robotics. *New Generation Computing*, 3(4):345–357, 1985.

[24] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[25] V.S. Subrahmanian. Paraconsistent disjunctive deductive databases. *Theoretical Computer Science*, 93(1):115–141, 1992.

[26] V.S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems: Theory and Implementation*. MIT Press, 2000.

[27] K. Sycara and D. Zeng. Multi-agent integration of information gathering and decision support. In Wolfgang Wahlster, editor, *European Conference on Artificial Intelligence (ECAI '96)*, pp. 549–556. Wiley & Sons, 1996.

[28] R. Weihmayer and H. Velthuijsen. Intelligent agents in telecommunications. In N. R. Jennings and M. J. Wooldridge, editors, *Agent Technology: Foundations, Applications and Markets*, pp. 203–217. Springer, Berlin, Germany, 1998.

[29] M. Wooldridge. *The Logical Model of Computational Multi–Agent Systems*. PhD thesis, Department of Computation, UMIST, Manchester, UK, October 1992.

[30] M. Wooldridge. *Reasoning about Rational Agents*. MIT Press, 2000.

A Appendix: Feasible, Rational, and Reasonable Status Sets

This appendix provides in succinct form the definition of various concepts of status sets from [14, 26], to which the reader is referred for more information.

Definition A.1 (Status Set) A *status set* is any set S of ground action status atoms over the values from the type domains of a software package $\mathcal{S}$. For any operator $Op \in \{\mathbf{P}, \mathbf{Do}\,, \mathbf{F}, \mathbf{O}, \mathbf{W}\}$, we denote by $Op(S)$ the set $Op(S) = \{\alpha \mid Op(\alpha) \in S\}$.

Definition A.2 (Operator $\mathbf{App}_{\mathcal{P},\mathcal{O}_S}(S)$) *Let $\mathcal{P}$ be an agent program and $\mathcal{O}$ be an agent state. Then, $\boldsymbol{App}_{\mathcal{P},\mathcal{O}_S}(S) = \{Head(r\theta) \mid r \in \mathcal{P},\ R(r,\theta,S)$ is true on $\mathcal{O}\}$, where the predicate $R(r,\theta,S)$ is true iff (1) $r\theta : A \leftarrow \chi \,\&\, L_1 \,\&\, \cdots \,\&\, L_n$ is a ground rule, (2) $\mathcal{O} \models \chi$, (3) if $L_i = Op(\alpha)$ then $Op(\alpha) \in S$, and (4) if $L_i = \neg Op(\alpha)$ then $Op(\alpha) \notin S$, for all $i \in \{1, \ldots, n\}$.*

Definition A.3 ($\mathbf{A\text{-}Cl}(S)$) *A status set S is* deontic and action closed*, if for every ground action α, it is the case that $(DC1)$ $\mathbf{O}\alpha \in S$ implies $\mathbf{P}\alpha \in S$, $(AC1)$ $\mathbf{O}\alpha \in S$ implies $\mathbf{Do}\,\alpha \in S$, and $(AC2)$ $\mathbf{Do}\,\alpha \in S$ implies $\mathbf{P}\alpha \in S$.*

For any status set S, we denote by $\mathbf{A\text{-}Cl}(S)$ the smallest set $S' \supseteq S$ such that S' is closed under $(AC1)$ and $(AC2)$, i.e., action closed.

Definition A.4 (Feasible Status Set) Let $\mathcal{P}$ be an agent program and let $\mathcal{O}$ be an agent state. Then, a status set S is a *feasible status set* for $\mathcal{P}$ on $\mathcal{O}$, if $(S1)$-$(S4)$ hold:

$(S1)$ $\mathbf{App}_{\mathcal{P},\mathcal{O}_S}(S) \subseteq S$;

$(S2)$ For any ground action α, the following holds: $\mathbf{O}\alpha \in S$ implies $\mathbf{W}\alpha \notin S$, and $\mathbf{P}\alpha \in S$ implies $\mathbf{F}\alpha \notin S$.

$(S3)$ $S = \mathbf{A\text{-}Cl}(S)$, i.e., S is action closed;

$(S4)$ The state $\mathcal{O}' = \mathbf{conc}(\mathbf{Do}\,(S), \mathcal{O})$ which results from $\mathcal{O}$ after executing (according to some execution strategy **conc**) the actions in $\mathbf{Do}\,(S)$ satisfies the integrity constraints, i.e., $\mathcal{O}' \models \mathcal{IC}$.

Definition A.5 (Groundedness; Rational Status Set) A status set S is *grounded*, if no status set $S' \neq S$ exists such that $S' \subseteq S$ and S' satisfies conditions $(S1)$–$(S3)$ of a feasible status set. A status set S is a *rational status set*, if S is a feasible status set and S is grounded.

Definition A.6 (Reasonable Status Set) Let $\mathcal{P}$ be an agent program, let $\mathcal{O}$ be an agent state, and let S be a status set.

1. If $\mathcal{P}$ is positive, i.e., no negated action status atoms occur in it, then S is a *reasonable status set* for $\mathcal{P}$ on $\mathcal{O}$, iff S is a rational status set for $\mathcal{P}$ on $\mathcal{O}$.

2. The reduct of $\mathcal{P}$ w.r.t. S and $\mathcal{O}$, denoted by $red^S(\mathcal{P}, \mathcal{O})$, is the program which is obtained from the ground instances of the rules in $\mathcal{P}$ over $\mathcal{O}$ as follows.

   (a) Remove every rule r such that $Op(\alpha) \in S$ for some $\neg Op(\alpha)$ in the body of r;

   (b) remove all negative literals $\neg Op(\alpha)$ from the remaining rules.

   Then S is a *reasonable status set* for $\mathcal{P}$ w.r.t. $\mathcal{O}$, if it is a reasonable status set of the program $red^S(\mathcal{P}, \mathcal{O})$ with respect to $\mathcal{O}$.

B Appendix: Safety

A variable is a *root variable*, if it does not involve deconstruction of an object. Given any variable $\mathtt{Y}$ (possibly involving deconstruction), its root $\mathit{root}(\mathtt{Y})$ is the variable which refers to the non-deconstructed object.

Definition B.1 (Safe Code Call (Condition)) *A code call* $\mathcal{S} : \mathtt{f}(\mathtt{d}_1, \ldots, \mathtt{d}_n)$ *is* safe *iff each* $\mathtt{d_i}$ *is ground. A code call condition* $\chi_1 \,\&\ldots\&\, \chi_n$*,* $n \geq 1$*, is* safe *iff there exists a permutation* π *of* $\chi_1, \ldots, \chi_n$ *such that for every* $i = 1, \ldots, n$ *the following holds:*

1. If $\chi_{\pi(i)}$ *is a comparison* $\mathtt{s}_1$ *op* $\mathtt{s}_2$*, then*

   1.1 at least one of $\mathtt{s}_1, \mathtt{s}_2$ *is a constant or a variable* $\mathtt{X}$ *such that* $\mathit{root}(\mathtt{X})$ *belongs to* $RV_\pi(i) = \{\mathit{root}(\mathtt{Y}) \mid \exists j < i \text{ s.t. } \mathtt{Y} \text{ occurs in } \chi_{\pi(j)}\}$*;*

   1.2 if $\mathtt{s_i}$ *is neither a constant nor a variable* $\mathtt{X}$ *such that* $\mathit{root}(\mathtt{X}) \in RV_\pi(i)$*, then* $\mathtt{s_i}$ *is a root variable.*

2. *If $\chi_{\pi(i)}$ is a code call atom of the form* $\mathtt{in}(\mathtt{X}_{\pi(\mathtt{i})}\ \mathtt{cc}_{\pi(\mathtt{i})})$ *or* $\mathtt{notin}(\mathtt{X}_{\pi(\mathtt{i})}\ \mathtt{cc}_{\pi(\mathtt{i})})$*, then the root of each variable* Y *occurring in* $\mathtt{cc}_{\pi(i)}$ *belongs to* $RV_\pi(i)$*, and either* $\mathtt{X}_{\pi(\mathtt{i})}$ *is a root variable, or* $root(\mathtt{X}_{\pi(\mathtt{i})})$ *is from* $RV_\pi(i)$.

Intuitively, a code call is safe, if we can reorder the code call atoms occurring in it in a way such that we can evaluate these atoms left to right, assuming that root variables are incrementally bound to objects.

Definition B.2 (Safety Modulo Variables) *Suppose χ is a code call condition, and let $\mathbf{X}$ be any set of root variables. Then, χ is said to be* safe modulo $\mathbf{X}$ *iff for an (arbitrary) assignment θ of objects to the variables in $\mathbf{X}$, it is the case that $\chi\theta$ is safe.*