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

HETEROGENEOUS ACTIVE AGENTS, III: POLYNOMIALLY IMPLEMENTABLE AGENTS

Thomas EITER V.S. SUBRAHMANIAN T.J. ROGERS

INFSYS RESEARCH REPORT 1843-99-07

MAY & DECEMBER 1999

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 1843-99-07, MAY & DECEMBER 1999

HETEROGENEOUS ACTIVE AGENTS, III: POLYNOMIALLY IMPLEMENTABLE AGENTS

REVISED VERSION, NOVEMBER 1999

Thomas Eiter[1] V.S. Subrahmanian[2] T.J. Rogers[3]

Abstract. In [17], two of the authors have introduced techniques to build agents on top of arbitrary data structures, and to "agentize" new/existing programs. They provided a series of successively more sophisticated semantics for such agent systems, and showed that as these semantics become epistemically more desirable, a computational price may need to be paid. In this paper, we identify a class of agents that are called *weakly regular*—this is done by first identifying a fragment of agent programs [17] called weakly regular agent programs (*WRAPs* for short). It is shown that *WRAPs* are definable via three parameters—checking for a property called "safety", checking for a property called "conflict freedom" and checking for a "deontic stratifiability" property. Algorithms for each of these are developed. A *weakly regular agent* is then defined in terms of these concepts , and a *regular* agent is one that satisfies an additional boundedness property. We then describe a polynomial algorithm that computes (under suitable assumptions) the reasonable status set semantics of regular agents—this semantics was identified in [17] as being epistemically most desirable. Though this semantics is coNP-complete for arbitrary agent programs [16], it is polynomially computable via our algorithm for regular agents. Finally, we describe our implementation architecture and provide details of how we have implemented *RAPs*, together with experimental results.

---

[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]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

[3]Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, Maryland 20742. E-mail: rogers@cs.umd.edu

Acknowledgements: We would like to thank Anatoly Levkov, Rob Ross, and Mustafa Tikir for writing code to implement some algorithms defined in this paper. We have benefited from discussions with Jürgen Dix, Bob Kowalski, and Sarit Kraus, and comments from other people. This work was supported by the Army Research Office under Grants DAAH-04-95-10174, DAAH-04-96-10297, and DAAH04-96-1-0398, by the Army Research Laboratory under contract number DAAL01-97-K0135, by an NSF Young Investigator award IRI-93-57756, by a DAAD grant, and by the Austrian Science Fund Project N Z29-INF.

This is Part III, in a revised version, which extends the Technical Reports INFSYS TR 1843-9802 and 1843-9803 "Heterogeneous Active Agents." This paper will appear in the *Artificial Intelligence* Journal.

Contents

1 Introduction

Over the last few years, there has been tremendous interest in the area of intelligent software agents. Such agents provide a wide range of services, ranging from data mediation agents [5, 6, 12, 14, 26], to mobile agents [34], to personalized visualization agents [13, 23], to agents that monitor newspapers, prioritize mail buffers and the like [18, 21, 36, 38].

Most such existing work on agents subscribes to the view that agents should be autonomous, and that such autonomous agents should behave according to a clearly articulated set of *operating principles*. These operating principles allow agents to take actions that change the state of the agent in accordance with the operating principles [31, 32]. Declarative languages to encode such operating principles were proposed by Shoham [37] and Hindriks *et al.* [22]. Recently, Eiter *et al.* [17] have proposed a notion of an Agent Program and showed how agent programs can be layered on top of arbitrary data structures. This allows the creator of an agent to agentize existing bodies of software code by "adding on" such operating principles on top of the code. A series of successively more sophisticated declarative semantics for agent programs have been described, whose epistemic appeal comes at a price of increased computational complexity. The creator of the agent can choose which semantics his agent will use.

Every agent in the [17] framework has an associated notion of *state*, and this state includes certain special data structures shown in Figure 1. While most of these data structures are not needed for this paper, the message box is critical. An agent's state changes when it receives messages from other agents or from other programs. This allows agents to be be *reactive* by responding to a message sent by another agent, but also allows them to be *clock driven* and *random*. A clock-driven action is taken by an agent when the agent receives a message from the clock agent (or program). A random-action is taken by an agent when the agent receives a message from a random number generation agent. Thus, agents in our framework are continuously engaged in a cycle consisting of the following three steps, which fits the observe-think-act paradigm described in [24]:

(S1) Evaluate state changes,

(S2) Determine what actions to take,

(S3) Concurrently execute the selected actions.

Step (S2) above requires that the agent computes certain kinds of semantical structures called *status sets*, described in [17], and also briefly described in Appendix B of this paper. Of the alternative semantics described in [17], it was argued that the "best" ones were all defined in terms of a "reasonable" status set semantics. However, computing such reasonable status sets is intractable in general. The primary contributions of this paper, which build in part on some ideas from logic programming (see also Section 9), are the following:

1. First and foremost, we identify a class of agent programs called *weakly regular agent programs* (*WRAPs* for short)—later, these are extended to *regular agent programs* (*RAPs*) that are guaranteed to have polynomially computable reasonable status sets, provided such status sets exist. *WRAPs* are characterized by three basic properties:

   (a) *Safety.* Every rule in a *WRAP* must satisfy a syntactic property called "safety" which guarantees that calls to external packages do not cause a run-time error. We also provide a notion of "strong safety" that guarantees that all computations return finite answers. Safety extends the idea of *mode realizability* [33, 8] of logic programs to handle calls to software packages.

(b) *Conflict Freedom.* The set of rules in a *WRAP* should not lead to conflicts. For example, the rules must not force an agent to do something it is forbidden to do.

(c) *Deontic Stratifiability.* This is a property in the spirit of stratification in logic programs [4], which prevents problems with negation in rule bodies. However, as we will see, deontic stratification is more complex than ordinary stratification because of (i) presence of deontic modalities in rule bodies, and because of (ii) the fact that rules can be inconsistent due to conflicting modalities in rule heads.

A regular agent program (*RAP*) is just a *WRAP* which satisfies an additional *boundedness condition* that we will introduce later in Section 6. Thus, all *RAPs* are *WRAPs*, but not vice versa. We will define all the above concepts, and provide algorithms that may be used by a syntax-checker in a *RAP* compiler. We will describe the results of experiments conducted with implementations of these algorithms.

2. Second, given a *RAP*, we will develop an iterative fixpoint computation procedure through which we can compute the reasonable status set (if it exists) of the agent program. *We will show that this procedure is polynomially executable under suitable conditions.* We will describe the results of experiments conducted with implementations of this procedure.

3. Third, we develop a package called a *Query Maintenance Package* (QMP for short) that may be used to process *RAPs*. The QMP manipulates a few simple data structures, and provides a succinct representation of a potentially very large set of actions an agent may be obliged to do, may be forbidden from doing, etc. In effect, it associates *at compile time* a set of queries which when executed at run-time are guaranteed to specify what the do's and dont's for the agent are. This allows the agent to be much more efficient at run-time, at the price of a static compile time computation.

4. Finally, we will describe our *IMPACT* Agent Development Environment, and outline how an agent developer may use it to build agents for her application. We will also include experimental results for our framework, demonstrating the efficiency of various components of our architecture.

This paper is part of the *IMPACT* ("Interactive Maryland Platform for Agents Collaborating Together") project, which is a joint effort between the University of Maryland (Subrahmanian), Technical University of Vienna (Eiter), Bar-Ilan University (Kraus), University of Koblenz (Dix), and Università di Milano (Bonatti).

The organization of this paper is as follows. Section 2 describes the syntax of agent programs and recapitulates important concepts from [17]. Section 3 lays out the basic definitions of a weakly regular agent program, while Section 4 presents algorithms for checking weak regularity. Section 5 discusses how weakly regular agent programs can be evaluated, and Section 6 defines regular agent programs. In Section 7, we describe our Query Maintenance Package, and in Section 8, we report experimental results and briefly describe the *IMPACT* Agent Development Environment. The final Section 9 discusses related work and gives some conclusions.

In order to distract not from the flow of reading, proofs of technical results have been moved to Appendix A.

2 Background: Agent Programs

In this section, we briefly review the concept of *agent program* proposed in [17]. We will first show how [17] allows us to define conditions on arbitrary bodies of software code, and then we will show how this might be used to build the rules defining the operating principles of an agent.

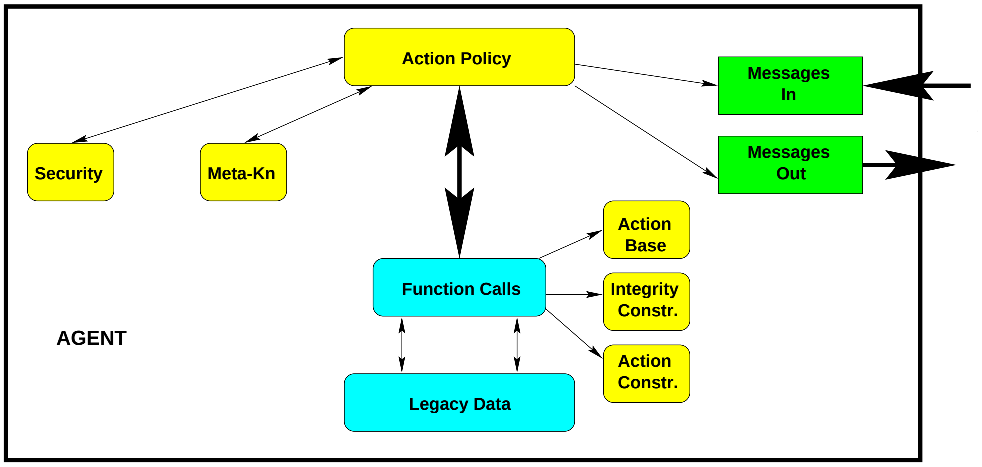


Figure 1: IMPACT Agent Architecture

2.1 Software code access

An agent program is built on top of a body of software code $\mathcal{S} = (\mathcal{T}, \mathcal{F})$ which consists of a set $\mathcal{T}$ of data types on which a set $\mathcal{F}$ of functions is available. The functions $\mathtt{f}$ in $\mathcal{F}$ provide the interface to the underlying data structures which are maintained by the agent; each $\mathtt{f}$ has a signature specifying the types of the input arguments and the type of the output (return) value. It is assumed that the output value is a set type; the result of ordinary functions is easily coerced into a singleton set.

Variables. For each $\tau \in \mathcal{T}$, we assume the existence of a set $Var(\tau)$ of variable symbols ranging over data objects of type τ. If $X \in Var(\tau)$ is such a variable symbol, and if τ is a complex record type having fields $f_1, \ldots, f_n$, then we require that $X.f_i$ be a variable of type τ_i, where τ_i is the type of field f_i. In the same vein, if f_i itself has a sub-field g of type γ, then $X.f_i.g$ is a variable of type γ, and so on. A *term* is either a data object (constant) or a variable, whose components may be referred to with a path expression (as in the case of variables). We often use t to denote a term. Furthermore, we use $\vec{S}$ to denote a list $S_1, S_2, \ldots, S_k$ of variables or terms S_i.

A *root-variable* is a variable symbol X from $Var(\tau)$. If τ is a complex record type as above, then $X.f_i$, $X.f_i.g_j$ etc are called *path-variables*. If Y is a path-variable of the form $X.path$, where X is a root variable, then X is called the *root* of Y, denoted by $root(Y)$; any root variable is its own root.

Agent State. At any point in time tp, the agent is in a particular agent state, $\mathcal{O}_{\mathcal{S}}(tp)$, which consists of a finite set of objects that are instances of the data types in $\mathcal{T}$. When tp is clear from the context, we will drop it.

Code Calls. The agent may access the agent state by means of *code calls*. If $\mathtt{f} \in \mathcal{F}$ is a function with n arguments, and $\mathtt{d_1}, \ldots, \mathtt{d_n}$ are objects or variables such that each $\mathtt{d_i}$ respects the type requirements of the i'th argument of $\mathtt{f}$, then $\mathcal{S}{:}\mathtt{f}(d_1, \ldots, d_n)$ is a *code call*. For example,

$$\mathbf{oracle}{:}\mathtt{select}(\mathtt{emp.rel}, \mathtt{salary}, >, \mathtt{150000})$$

could be a code call atom to the Oracle database package, asking for all records from an employee relation with SALARY field over 150K. Similarly, $\mathbf{terrain}{:}\mathtt{route}(3, \mathtt{map5938}, 45, 50, 90, 100)$ may be a code call requesting the three best routes from location (45,50) to (90,100) where the coordinates reference a terrain map `map5938`. Both these code calls are implemented in our *IMPACT* system.

Expressions of the form $\mathbf{in}(t, \mathtt{cc})$ and $\mathbf{not\_in}(t, \mathtt{cc})$, where t is a term of the output type of `cc` and `cc` is a code call, are called *code call atoms*.

A *code call condition* is defined as follows:

1. Every code call atom is a code call condition.
2. If `s` and `t` are either variables or objects, then $\mathtt{s} = \mathtt{t}$, $\mathtt{s} \neq \mathtt{t}$ are code call conditions.
3. If `s` and `t` are either integers/real valued objects, or are variables over the integers/reals, then $\mathtt{s} < \mathtt{t}$, $\mathtt{s} > \mathtt{t}$, $\mathtt{s} \geq \mathtt{t}$, and $\mathtt{s} \leq \mathtt{t}$ are code call conditions.
4. If χ_1, χ_2 are code call conditions, then $\chi_1 \,\&\, \chi_2$ is a code call condition.

A code call condition satisfying any of the first three criteria above is an *atomic* code call condition.

An example of a code call condition is:

$$\begin{array}{l}\mathbf{in}(X, \mathbf{oracle}{:}\mathtt{select}(\mathtt{emp.rel}, \mathtt{salary}, >, \mathtt{150000}))\ \& \\ \mathbf{in}(Y, \mathbf{face}{:}\mathtt{findpictureof}(\mathtt{mugshotdb}, X.name))\end{array} \qquad (1)$$

This condition may be viewed as a query requesting that we find all X and Y such that X is a person who makes over 150K (as determined by querying an Oracle relation called `emp.rel`), and finding all pictures Y of such a person from the mugshot database.

Code call conditions provide the basic syntax by which agents can access specialized (or legacy) data types of objects on which they are built.

Integrity Constraints. Agents may often require that their states satisfy integrity constraints which are rules of the form

$$\psi \quad \Rightarrow \quad \chi,$$

where ψ is a code call condition, and χ is an atomic code call condition such that every root variable in χ occurs in ψ. An example of an integrity constraint is one that says that all messages must be processed within 10 time units of the time received:

$$\begin{array}{l}\mathbf{in}(X, \mathbf{msgbox}{:}\mathtt{select}(\mathtt{Todo}))\ \& \\ \mathbf{in}(CurTime, \mathbf{clock}{:}\mathtt{now}())\ \& \\ \mathbf{in}(Delay, \mathbf{math}{:}\mathtt{minus}(CurTime, X.received)) \Rightarrow Delay < 10.\end{array}$$

2.2 Action rules

Once the developer of an agent knows what the data types and external function calls on top of which she wants to build an agent are, she may encode the *behavior* or *operating principles* that she wants her agent to obey.

Actions. An action α is described via 5-components:

(i) a list of data types $(\tau_1, \ldots, \tau_n)$ specifying the types of α's parameters,

(ii) a *precondition* which is a code call condition,

(iii) an *add list* of effects that are code call conditions,

(iv) a *delete list* of effects that are code call conditions, and

(v) an *execution codebase* that specifies how the action is implemented.

If the precondition of the action is true in the current state, then we may execute the action by invoking the execution codebase, and the new state that results will be just like the current state, except that the items in the add (resp., delete) list will be true (resp., false) in the new state. Notice that unlike classical AI planning systems, actions are implemented by (possibly imperative) codebases—the preconditions, add and delete lists merely allow the agent to reason about the new state that results from the execution of the execution code. (Later in the paper, we will require that actions satisfy certain additional “safety” criteria.)

Notion of Concurrency. A notion of concurrency is a function **conc** that takes as input a set of actions A and an agent state $\mathcal{O}_\mathcal{S}$, and returns as output a single action α such that if $A = \{\alpha'\}$ is a singleton set, then $\mathbf{conc}(A, \mathcal{O}_\mathcal{S}) = \alpha'$. Intuitively, a notion of concurrency is a “black box” that “merges” a set of actions together producing a new action. For example, if a robotic agent concurrently executes two operations $\mathtt{move}(\mathtt{north})$ and $\mathtt{move}(\mathtt{east})$, the concurrent execution of these two actions may be a new action which moves north-east. The condition in the definition of **conc** says that concurrent execution of a single action results in executing just that action. The paper [17] proposes three alternative notions of concurrency that may be used for agents. Which notion of concurrency is used for a given application depends on the application—our implementation provides a library and also allows an agent developer to build her own notion of concurrency and add it to the library.

Observe that the “merged” action produced by **conc** may depend upon the state. For example, the concurrent execution of the actions $\mathtt{jump}()$ and $\mathtt{open\_parachute}()$ may lead to two different merged actions (with consequently differing resulting states), depending upon the state in which the actions are executed (jumping from an airplane at 20,000 feet and jumping off a 400 foot cliff). We emphasize that the merged action may be a completely new (compound) action.

Action Constraints. Action constraints are formulas that an agent designer may explicitly write so that his agents cannot execute certain actions concurrently under specified circumstances. An action constraint AC has the syntactic form

$$\{\alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)\} \hookleftarrow \chi, \tag{2}$$

where $\alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)$ are action names, and χ is a code call condition. The above constraint says that if condition χ is true, then the actions $\alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)$ are not concurrently executable.

Action Status Atoms. An *action status atom* is a formula $Op(\alpha(t_1, \ldots, t_n))$, where $Op \in \{\mathbf{P}, \mathbf{F}, \mathbf{O}, \mathbf{W}, \mathbf{Do}\}$, α is the name of an action, and each t_i is a term, i.e., an object or a variable, of a suitable type τ_i, for all $i = 1, \ldots, n$. The operator Op expresses a deontic modality with respect to taking this action:

- $\mathbf{P}\alpha$ means that the agent is permitted to take action α;
- $\mathbf{F}\alpha$ means that the agent is forbidden from taking action α;
- $\mathbf{O}\alpha$ means that the agent is obliged to take action α;
- $\mathbf{W}\alpha$ means that obligation to take action α is waived; and,
- $\mathbf{Do}\alpha$ means that the agent does take action α.

Action Rules. An *action rule* (*rule*, for short) is a clause r of the form

$$A \leftarrow L_1, \ldots, L_n \tag{3}$$

where A is an action status atom, denoted $head(r)$, and each of $L_1, \ldots, L_n$ is either an action status literal, i.e., an action status atom B or a negated action status atom $\neg B$, or a code call atom.[1] The symbol "," in the body of the above rule is used to denote conjunction, and will sometimes be used interchangeably with the symbol "&." The body of r is denoted by $B(r)$. The set of code call atoms (resp., action status literals) in $B(r)$ is denoted by $B_{cc}(r)$ (resp., $B_{as}(r)$). The subset of positive (resp., negated) action status atoms in $B_{as}(r)$ is denoted by $B^{+}_{as}(r)$ (resp., $B^{-}_{as}(r)$). Furthermore, for each literal L, $\neg.L$ denotes the opposite literal, and for each set X of literals, $\neg.X = \{\neg.L \mid L \in X\}$.

Agent Program. An *agent program* $\mathcal{P}$ is a finite collection of rules. It is called *positive*, if no negation occurs in it.

Throughout this paper, we will often allude to *IMPACT* agents. For the sake of clarity, an *IMPACT* agent is defined below.

Definition 2.1 (*IMPACT* agent) *An IMPACT agent* $\mathbf{a}$ *consists of the following components: (i) a body of software code* $\mathcal{S} = (\mathcal{T}, \mathcal{F})$*, (ii) an initial state* $\mathcal{O}(0)$ *at the time of deployment, (iii) a finite set* $\mathcal{IC}$ *of integrity constraints, (iv) a finite set* $\mathcal{AB}$ *of actions, (vi) a finite set* $\mathcal{AC}$ *of action constraints, (vii) a notion* $\mathbf{conc}$ *of concurrency, and (viii) an agent program* $\mathcal{P}$*.*

While an agent $\mathbf{a}$ is deployed, all the above parameters stay constant. However, the agent's state changes over time, as messages are received from other agents, and appropriate actions are taken or responses are made.

The semantics of an agent is formally defined in [17] in terms of *feasible status sets*. A feasible status set S is a *status set*, i.e., a set of ground action status atoms, which satisfies various conditions. Firstly, it should comply with "common-sense" deontic axioms (listed formally below) and be closed under the rules of the program. Secondly, concurrently executing all actions α such that $\mathbf{Do}(\alpha) \in S$ should take the agent from its current state $\mathcal{O}$ to another state $\mathcal{O}'$ which satisfies the integrity constraints $\mathcal{IC}$. Thirdly, concurrent execution of the set of all actions α such that $\mathbf{Do}(\alpha) \in S$ should not violate any of the action constraints $\mathcal{AC}$. We formally describe these informal intuitions below.

The *deontic consistency axioms* are: For any ground action α,

$(D1)$ *If* $\mathbf{O}\alpha \in S$*, then* $\mathbf{W}\alpha \notin S$

$(D2)$ *If* $\mathbf{P}\alpha \in S$*, then* $\mathbf{F}\alpha \notin S$

$(D3)$ *If* $\mathbf{P}\alpha \in S$*, then* $\mathcal{O}_S \models Pre(\alpha)$ *(i.e.,* α *is executable in the state* $\mathcal{O}_S$*).*

These constraints are supplemented by *deontic and action closure rules*. A status set S is *deontic and action closed*, if for every ground action α, it is the case that

$(DC1)$ $\mathbf{O}\alpha \in S \rightarrow \mathbf{P}\alpha \in S$,

$(AC1)$ $\mathbf{O}\alpha \in S \rightarrow \mathbf{Do}\alpha \in S$, and

$(AC2)$ $\mathbf{Do}\alpha \in S \rightarrow \mathbf{P}\alpha \in S$.

[1] Note that [17] allows for using negation in front of code call atoms in rule bodies, which is eliminated here without loss of expressiveness.

Observe that $(DC1)$ holds if $(AC1)$ and $(AC2)$ hold; thus, we will not explicitly deal with $(DC1)$ in the rest of this paper. For any status set S, we denote by $\mathbf{A}\text{-}\mathbf{Cl}(S)$ the smallest set $S' \supseteq S$ such that S' is closed under $(AC1)$ and $(AC2)$, i.e., it is *action closed*. Intuitively, $\mathbf{A}\text{-}\mathbf{Cl}(S)$ represents the closure of S under actions and deontic modalities. The definition of a feasible status set is provided in Appendix B.

Along with feasible status sets, epistemically more appealing semantics are presented in [17]. In particular, feasible status sets may contain action status atoms $Op(\alpha)$ which are not "sufficiently grounded" in the rules of the program and the deontic closure rules. *Rational status sets* informally require that a feasible status set is minimal, i.e., no subset of action status set can be removed while the rules and deontic axioms are satisfied. *Reasonable status sets* further restrict rational status sets by a treatment of negation in the spirit of stable negation for logic programs [19]; this way, logical contraposition is eliminated, and the computational complexity of evaluating agent programs is reduced in general. The definition of rational and reasonable status sets is provided in Appendix B. As shown in [17], reasonable status sets have many elegant important properties. Firstly, reasonable status sets extend the well known stable model semantics of logic programs; secondly, reasonable status sets are all "grounded" in the sense that whenever a status atom is in a reasonable status set, then there is a non-cyclic "support" for it; and thirdly, reasonable status sets are computationally not as hard to compute as the other semantics [16]. Hence, in this paper, we will largely concentrate on computing the reasonable status set semantics.

3 Conflicts and Weak Regularity

In this section, we will define the class of *weakly regular* agent programs (*WRAPs*). The main intent of *WRAPs* is to provide polynomial computation algorithms, while still supporting a very broad range of applications. For this purpose, we employ a notion of *deontic stratification*, which layers an agent program and enables bottom up evaluation, and a concept of *strong safety*, which guarantees that only a finite number of rule instances is relevant in each agent state. By imposing some further boundedness condition on recursion in *WRAPs*, we shall obtain *regular agent programs* (*RAPs*, cf. Section 5), which can be evaluated in polynomial time under suitable conditions.

When developing a *RAP*, agent developers need to make some choices that influence the compile-time and run-time behavior of agent programs. These choices include their answers to the following questions:

Executability and Finiteness. For a code-call to be executable at run-time, its arguments must be instantiated. (Imagine asking a $\mathbf{calculator}$ program to compute $3 + X$ where X is an uninstantiated variable!) Safety is a *compile-time* check that ensures that all code calls generated at *run-time* have instantiated parameters. Finiteness is a different property that requires that whenever a code call condition is evaluated, it yields a finite answer. *Strong safety* guarantees finiteness as well. Section 3.1 defines safety and strong safety, together with algorithms to compute them.

Conflict Freedom. Agent programs may contain conflicts in their operating rules. Even a simple agent program may contain a rule $\mathbf{F}\alpha \leftarrow$ and another conflicting rule $\mathbf{Do}\alpha \leftarrow$. The agent developer must decide, when creating the program (i.e. at compile-time or before), what syntactic conditions will guarantee that conflicts will always be avoided, and how much these syntactic conditions restrict the agent programs he wants to write. Section 3.2 defines conflict freedom, together with different possible conflict freedom algorithms.

After these two choices are made by the agent designer, we will provide definitions of what it means for an agent program to satisfy some *layering* conditions in Subsection 3.3. *WRAPs* are then defined in Subsection 3.4 based on the preceding subsections. In Subsection 4, we develop algorithms for regularity checking and describe their efficiency.

3.1 Safety

We are now ready to define the concept of safety. Whenever an attempt is made to execute a code call, the code call must be fully instantiated. This motivates the definition of a *safe* code call.

Definition 3.1 (safe code call [17]) *A code call $\mathcal{S}$:$\mathtt{f}(d_1, \ldots, d_n)$ is* safe *iff each 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 $\mathtt{s}_1 \,\mathtt{op}\, \mathtt{s}_2$, then*
   1.1 *at least one of $\mathtt{s}_1$ and $\mathtt{s}_2$ is a constant or a variable X such that $root(X)$ belongs to $RV_\pi(i) = \{root(Y) \mid Y$ occurs in $\chi_{\pi(j)}$ for some $j < i\}$;*
   1.2 *if $\mathtt{s_i}$ is neither a constant nor a variable X such that $root(X) \in RV_\pi(i)$, then $\mathtt{s_i}$ is a root variable.*
2. *If $\chi_{\pi(i)}$ is a code call atom of the form $\mathbf{in}(X_{\pi(i)}, \mathtt{cc}_{\pi(i)})$ or $\mathbf{not\_in}(X_{\pi(i)}, \mathtt{cc}_{\pi(i)})$, then the root of each variable Y occurring in $\mathtt{cc}_{\pi(i)}$ belongs to $RV_\pi(i)$, and either $X_{\pi(i)}$ is a root variable, or $root(X_{\pi(i)})$ is from $RV_\pi(i)$.*

In this case, π is called a witness *to the safety of $\chi_1, \ldots, \chi_n$.*

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

For example, the code call condition $\mathbf{in}(X, \mathcal{S}$:$\mathtt{f}_1(a, Y))$ is not safe because it requires the function $\mathtt{f}_1$ to be executed on two arguments, one of which is not instantiated. On the other hand, the code call condition $\mathbf{in}(X, \mathcal{S}$:$\mathtt{f}_1(a, Y)) \,\&\, \mathbf{in}(Y, \mathcal{S}$:$\mathtt{f}_2(b))$ is safe, because we can execute the code call atom $\mathbf{in}(Y, \mathcal{S}$:$\mathtt{f}_2(b))$ to obtain a ground binding θ for Y and then execute $\mathbf{in}(X, \mathcal{S}$:$\mathtt{f}_1(a, Y\theta))$.

3.1.1 algorithm for checking safety

Checking safety of code call conditions can be done at compile time. If χ is found to be safe, then its individual conjuncts $\chi_1, \ldots, \chi_n$ can be reordered according to a permutation that witnesses the safety of χ. Before describing the algorithm to check safety of a code call condition, we define the notion of safety relativized to a set of variables—intuitively, this definition checks if a code call condition is safe, given that certain variables in it are instantiated.

Definition 3.2 (safety modulo $\vec{X}$) *Suppose χ is a code call condition, and $\vec{X}$ is a list of root variables. Then, χ is said to be* safe modulo $\vec{X}$ *iff for an (arbitrary) assignment θ of objects to the variables in $\vec{X}$ (which inherits an assignment to the path variables with roots in $\vec{X}$), it is the case that $\chi\theta$ is safe.*

Observe that an assignment of an object to a variable X implicitly assigns a value to all path variables $X.f$, $X.f.g$ etc. Intuitively, a code call condition as actually written may be unsafe. However, it could be the intent that the code call condition only be invoked with certain arguments instantiated. If any such instantiation leads to a safe code call, then the original code call is said to be safe modulo the arguments in question.

The reader will easily observe that checking safety of a code call χ modulo variables $\vec{X}$ can be reduced to a call to a routine that checks for safety. This may be done as follows:

1. For each variable X in $\vec{X}$, take an arbitrary value $\mathtt{c}_X$ from the domain X ranges over, and let θ be the assignment which sets every X to the value $\mathtt{c}_X$.

2. Check whether $\chi\theta$ is safe.

Our algorithm for checking safety of a code call condition χ, **Safe_ccc**, is described below.

Algorithm 3.1 ***Safe_ccc(χ: code call condition; $\vec{X}$: list of root variables)***

($\star$ input is a code call condition $\chi = \chi_1 \& \cdots \& \chi_n$; output is a proper reordering $\star$)
($\star$ $\chi' = \chi_{\pi(1)} \& \cdots \& \chi_{\pi(n)}$ if χ is safe modulo $\vec{X}$; otherwise, the output is "$unsafe$"; $\star$)

$L := \chi_1, \ldots, \chi_n$;
$\chi := \mathtt{true}$;
while *L is not empty* ***do***
{ *Select all $\chi_{i_1}, \ldots, \chi_{i_m}$ from L such that χ_{i_j} is safe modulo $\vec{X}$;*
if $m = 0$ ***then return*** *"$unsafe$" and halt;*
else
{ $\chi := \chi \& \chi_{i_1} \& \cdots \& \chi_{i_m}$;
Remove $\chi_{i_1}, \ldots, \chi_{i_m}$ from L;
$\vec{X} = \vec{X} + (root(Y) \mid Y$ *occurs in some* $\chi_{i_1}, \ldots \chi_{i_m})$;
}
}
return χ' *and halt;*
end.

The following result states that our algorithm is correct.

Theorem 3.1 (safety computation) *Suppose $\chi = \chi_1 \& \ldots \& \chi_n$ is a code call condition. Then, χ is safe modulo a list of root variables $\vec{X}$, if and only if **Safe_ccc**$(\chi, \vec{X})$ returns a reordering χ' of χ. Moreover, for any assignment θ to the variables in $\vec{X}$, $\chi'\theta$ is a safe code call condition which can be evaluated left-to-right.*

A straightforward implementation of **Safe_ccc** runs in quadratic time, as the number of iterations is bounded by the number n of conjuncts χ_i in χ, and as the body of the while loop can be executed in linear time. By using appropriate data structures, the algorithm can be implemented to run in linear time. Briefly, the method uses standard cross reference lists of variable occurrences.

Observe that safety of a code call condition χ can be checked by calling **Safe_ccc**$(\chi, \emptyset)$. Thus, checking the safety of χ, combined with a reordering of its constituents for left-to-right execution can be done very efficiently—this statement will be experimentally validated in Section 8.1.

3.1.2 safe rules and programs

We are now ready to define a safe rule and a safe agent program.

Definition 3.3 (safe agent program) *A rule r is said to be* safe *iff*

1. *$B_{cc}(r)$ is safe modulo the root variables occurring explicitly in $B^+_{as}(r)$, and*
2. *the root of each variable in r occurs in $B_{cc}(r) \cup B^+_{as}(r)$.*

An agent program is safe if every rule in it is safe.

Suppose we have a safe rule r, and suppose further that we know that Θ is the set of all ground substitutions such that $B^+_{as}(r)$ is true. Then we can evaluate the set of instances of the head of the rule that must be true as follows:

1. Set $Ans = \emptyset$.
2. For each $\theta \in \Theta$ do
   (a) Evaluate $B_{cc}(r\theta)$—this code call condition is executable because it is safe modulo the variables in $B^+_{as}(r)$. Let Γ_θ be the set of all answer substitutions returned by this evaluation.
   (b) For every answer substitution $\gamma \in \Gamma_\theta$, check to see whether $B^-_{as}(r')$ is true, where $r' = r\theta\gamma$, and let Γ^+_θ be the set of all such γ. This step is executable as all variables in $B^-_{as}(r\theta\gamma)$ are instantiated by condition (2) in the definition of a safe rule.
   (c) Set $Ans = Ans \cup \{head(r') \mid r' = r\theta\gamma,\ \gamma \in \Gamma^+_\theta$, and the action α from each atom $Op\alpha \in B^+_{as}(r) \cup \{head(r')\}$ such that $Op \in \{\mathbf{Do}, \mathbf{O}, \mathbf{P}\}$ is executable $\}$.
3. Return Ans.

The above procedure will obviously terminate if the sets Θ and Γ_θ are finite (assuming the code calls implemented terminate). In Section 3.1.3, we will introduce methods to guarantee this.

The following example demonstrates some safe rules.

Example 3.1 Consider the following five rules.

$$\begin{array}{lcl}
\mathbf{P}\alpha(X,Y) & \leftarrow & \mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,b))\,\&\,\mathbf{in}(Y,\mathcal{S}{:}\mathtt{g}(X.\mathit{field},b,c)).\\
\mathbf{P}\beta(X,Y) & \leftarrow & \mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,\,Y)).\\
\mathbf{P}\beta(X,Y) & \leftarrow & \mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,\,Y))\,\&\,\mathbf{F}\gamma(Y).\\
\mathbf{P}\alpha(X,Y) & \leftarrow & \mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,b))\,\&\,\mathbf{in}(Y,\mathcal{S}{:}\mathtt{g}(X.\mathit{field},b,c))\,\&\,\neg\mathbf{F}\gamma(Z).\\
\mathbf{P}\alpha(X,Y) & \leftarrow & \mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,b))\,\&\,\mathbf{in}(Y,\mathcal{S}{:}\mathtt{g}(X.\mathit{field},b,c))\,\&\\
 & & \mathbf{F}\beta(Z)\,\&\,\neg\mathbf{F}\gamma(Z).
\end{array}$$

The reader can see that the first rule is safe, but the second rule is not safe. Notice that the second rule is safe modulo the set $\{Y\}$ of variables. Suppose we use r to refer to the third rule. Then the code call part $B_{cc}(r)$—which coincides with the body of the second rule—is safe modulo $\{Y\}$. As Y occurs in a positive action status atom in r's body, this rule is safe by definition. The fourth rule is not safe, because the variable Z occurs in a negative action status atom in the rule's body, but does not occur in either the code call part of the rule body or in a positive status atom in the rule body. However, the fifth rule is safe, because the variable Z occurs in a positive status atom in the rule body.

3.1.3 strong safety

Suppose the developer of an agent examines the code calls supported by a given data structure and specifies which of them are finite and which are not. As is well known, determining whether a function is finite or not is undecidable, cf. [30], hence we require input from the agent developer here. If `f` is a function whose arguments have types $(\tau_1, \ldots, \tau_n)$, then a *binding pattern* is an expression of the form $(bt_1, \ldots, bt_n)$ where each bt_i (called a *binding term*) is either:

1. A value of type τ_i, or
2. The expression $\flat$ denoting that this argument is bound to an unknown value.

The agent developer specifies a *finiteness* predicate that may be defined via a *finiteness table* having three columns—the first column is the name of the software package or data structure, the second column is the name of the function in question, and the third argument is a binding pattern for the function in question.[2] A simple example of a finiteness table is shown in Table 1.

Package	Function	Binding Pattern
terrain	`route`	$(\flat, \flat, \flat, \flat, \flat)$
oracle	`select`	$(\flat, \flat, =, \flat)$

Table 1: Example finiteness table $\mathbf{FINTAB}_0$

The first row of Table 1 says that executing the **route** function defined in the **terrain** package always yields a finite result. The second row of Table 1 says that the result of an Oracle select is always finite when the selection condition is an "equality" condition, and the relation on which the select is done is known and the value on which the equality is tested is known.

Binding terms may be partially ordered as follows: $\flat \leq val$ for all values val of type τ_i, and $\leq$ is reflexively closed. We say a binding pattern $(bt_1, \ldots, bt_n)$ is *less informative* than another binding pattern $(bt'_1, \ldots, bt'_n)$ iff $bt_i \leq bt'_i$, for all $1 \leq i \leq n$. For example, $(\flat, \flat, 5) \leq (30, \flat, 5)$ because the second binding pattern is more informative about the arguments to a code call—it says the first parameter is bound to 30, while the second will be bound. The first binding pattern only says the first parameter will be bound.

Definition 3.4 (finiteness) *Suppose* $\mathbf{FINTAB}$ *is a finite finiteness table, and* $(bt_1, \ldots, bt_n)$ *is a binding pattern associated with a package* $\mathcal{S}$ *and a function* $\mathtt{f} \in \mathcal{S}$*. Then* $\mathbf{FINTAB}$ *is said to* entail the finiteness of $\mathcal{S}\mathtt{:f}(bt_1, \ldots, bt_n)$ *iff there exists an entry of the form* $(\mathcal{S}, \mathtt{f}, (bt'_1, \ldots, bt'_n))$ *in* $\mathbf{FINTAB}$ *such that* $(bt_1, \ldots, bt_n)$ *is more informative than* $(bt'_1, \ldots, bt'_n)$.

Below, we show how the small example finiteness table introduced earlier allows us to entail the finiteness of some simple code calls.

Example 3.2 Using the example finiteness table $\mathbf{FINTAB}_0$ specified above, we see that

[2] In the *IMPACT* implementation, we ask the user to represent the infinite, rather than finite code calls. This is because in most cases, we expect code calls to be finite—thus, representing relatively few infinite code calls might reduce the burden on the agent developer. In addition, the implementation allows an extra "constraint" column, and variables in binding patterns. Thus, a user can write that a code call $\mathbf{d}\mathtt{:f}(X, Y)$ yields an infinite answer when $X > 400$. We have chosen to keep the presentation in this paper in the current form so as to make it more understandable.

1. $\mathbf{FINTAB}_0$ entails the finiteness of the code call $\mathbf{route}{:}\mathtt{plan}(3, \mathtt{map1}, 10, 10, 20, 30)$ requesting the three best routes from point $(10, 10)$ to $(20, 30)$ in map1.

2. Similarly, $\mathbf{FINTAB}_0$ entails the finiteness of $\mathbf{oracle}{:}\mathtt{select}(\mathtt{emp}, \mathtt{name}, =, \mathtt{john})$.

3. However, $\mathbf{FINTAB}_0$ does *not* entail the finiteness of $\mathbf{oracle}{:}\mathtt{select}(\mathtt{emp}, \mathtt{salary}, >, \mathtt{50})$, even though this code call has a finite answer if the relation emp is finite.

4. Notice that $\mathbf{FINTAB}_0$ does *not* entail the finiteness of $\mathbf{route}{:}\mathtt{plan}(X, \mathtt{map1}, 10, 10, 20, 30)$. This code call asks for all routes between the points $(10, 10)$ and $(20, 30)$.

According to the above definition, when we know that $\mathbf{FINTAB}$ entails the finiteness of the binding pattern $\mathcal{S}{:}\mathtt{f}(bt_1, \ldots, bt_n)$, then we know that every code call of the form $\mathcal{S}{:}\mathtt{f}(\ldots)$ whose arguments satisfy the binding requirements are guaranteed to yield finite answers. However, defining strong safety of a code call condition is more complex. For instance, even if we know that $\mathcal{S}{:}\mathtt{f}(t_1, \ldots, t_n)$ is finite, the code call atom $\mathbf{not\_in}(X, \mathcal{S}{:}\mathtt{f}(t_1, \ldots, t_n))$ may have an infinite answer. Likewise, comparison conditions such as $\mathtt{s} > \mathtt{t}$ may have finite answers in some cases and infinite answers in other cases, depending upon whether we are evaluating variables over the reals, the integers, the positive reals, the positive integers, etc. In the sequel, we make two simplifying assumptions, though both of them can be easily modified to handle other cases:

1. First, we will assume that every function f has a complement $\overline{\mathtt{f}}$. An object o is returned by the code call $\mathcal{S}{:}\overline{\mathtt{f}}(t_1, \ldots, t_n)$ iff o is not returned by $\mathcal{S}{:}\mathtt{f}(t_1, \ldots, t_n)$. Once this occurs, all code call atoms of the form $\mathbf{not\_in}(X, \mathcal{S}{:}\mathtt{f}(t_1, \ldots, t_n))$ may be rewritten as $\mathbf{in}(X, \mathcal{S}{:}\overline{\mathtt{f}}(t_1, \ldots, t_n))$, thus eliminating the negation membership predicate. When the agent developer creates $\mathbf{FINTAB}$, he must also specify the finiteness conditions (if any) associated with function calls $\overline{\mathtt{f}}$.

2. Second, in the definition of strong safety below, we assume that all comparison operators involve variables over types having the following property.

Downward Finiteness Property. A type τ is said to have the *downward finiteness property* iff it has an associated partial ordering $\leq$ such that for all objects o of type τ, the set $\{o' \mid o'$ is an object of type τ and $o' \leq o\}$ is finite.

We are now ready to define strong safety.

Definition 3.5 (strong safety) *A safe code call condition* $\chi = \chi_1 \,\&\ldots\&\, \chi_n$ *is* strongly safe *with respect to a list* $\vec{X}$ *of root variables iff there is a permutation* π *witnessing the safety of* χ *modulo* $\vec{X}$ *such that for each* $1 \leq i \leq n$, $\chi_{\pi(i)}$ *is strongly safe modulo* $\vec{X}$, *where strong safety of* $\chi_{\pi(i)}$ *is defined as follows:*

1. $\chi_{\pi(i)}$ is a code call atom.
Here, let the code call of $\chi_{\pi(i)}$ *be* $\mathcal{S}{:}\mathtt{f}(t_1, \ldots, t_n)$ *and let the binding pattern* $\mathcal{S}{:}\mathtt{f}(bt_1, \ldots, bt_n)$ *be defined as follows:*

   (a) If t_i *is a value, then* $bt_i = t_i$.
   (b) Otherwise t_i *must be a variable whose root occurs either in* $\vec{X}$ *or in* $\chi_{\pi(j)}$ *for some* $j < i$. *In this case,* $bt_i = \flat$.

Then, $\chi_{\pi(i)}$ is strongly safe iff $\mathbf{FINTAB}$ *entails the finiteness of* $\mathcal{S}$:$\mathtt{f}(bt_1, \ldots, bt_n)$.

2. $\chi_{\pi(i)}$ *is* $\mathtt{s} \neq \mathtt{t}$.
   In this case, $\chi_{\pi(i)}$ is strongly safe iff each of $\mathtt{s}$ *and* $\mathtt{t}$ *is either a constant or a variable whose root occurs either in $\vec{X}$ or in $\chi_{\pi(j)}$ for some $j < i$.*

3. $\chi_{\pi(i)}$ *is* $\mathtt{s} < \mathtt{t}$ *or* $\mathtt{s} \leq \mathtt{t}$.
   In this case, $\chi_{\pi(i)}$ is strongly safe iff $\mathtt{t}$ *is either a constant or a variable whose root occurs either in $\vec{X}$ or somewhere in $\chi_{\pi(j)}$ for some $j < i$.*

4. $\chi_{\pi(i)}$ *is* $\mathtt{s} > \mathtt{t}$ *or* $\mathtt{s} \geq \mathtt{t}$.
   In this case, $\chi_{\pi(i)}$ is strongly safe iff $\mathtt{t} < \mathtt{s}$ *or* $\mathtt{t} \leq \mathtt{s}$*, respectively, are strongly safe.*

It is important to note that if we consider variables over types that do *not* satisfy the downward finiteness property (as in the case of the reals), then Case 1 and Case 2 above define strong safety—all code calls of the forms shown in Cases 3 and 4 are not strongly safe. Thus, the definition of strong safety applies both to types satisfying the downward finiteness property and to types that do not satisfy it.

A straightforward algorithm to check strong safety of a code call condition is to require that the test "select all $\chi_{i_1}, \ldots, \chi_{i_m}$ from L such that χ_{i_j} is safe modulo $\vec{X}$" in algorithm **Safe_ccc** on page 9 be replaced by the condition "select all $\chi_{i_1}, \ldots, \chi_{i_m}$ from L such that χ_{i_j} is *strongly* safe modulo $\vec{X}$." As a consequence, it is not hard to see that strong safety can be checked in time proportional to the product of the time taken to check safety and the time to look up items in $\mathbf{FINTAB}$. The former is quadratic (using appropriate data structures, even linear) in the length of the code call condition, and the latter is linear in the number of entries in $\mathbf{FINTAB}$.

Definition 3.6 (strongly safe agent program) *A rule r is* strongly safe *iff it is safe, and $B_{cc}(r)$ is a strongly safe code call condition. An agent program is* strongly safe *iff all rules in it are strongly safe.*

We will require that all agent programs be strongly safe—even though this increases the development cycle time, and compilation time, these are "one time" costs which are never incurred at run-time. Hence, the price is well worth paying. When we know that an agent program rule r is strongly safe, we are guaranteed that the computation of the set of instances of the head of the rule that is true involves only finite subcomputations.

3.2 Conflict freedom

Section 3.1 specifies the safety condition that an agent program must satisfy for it to be called *regular*. In this section, we specify another condition for regularity—namely that conflicts must be avoided.

The deontic consistency requirement associated with a feasible status set mandates that all feasible status sets (and hence all rational and reasonable status sets) must be *deontically consistent*. Therefore, we need some way of ensuring that agent programs are conflict-free, and this means that we first need to define what a conflict is.

Definition 3.7 (conflicting modalities) *Given two action modalities $Op, Op' \in \{\mathbf{P}, \mathbf{F}, \mathbf{O}, \mathbf{Do}, \mathbf{W}\}$ we say that Op* conflicts with *Op' iff there is an entry "$\times$" in the following table at row Op and column Op':*

$Op \setminus Op'$	**P**	**F**	**O**	**W**	**Do**
P		$\times$			
F	$\times$		$\times$		$\times$
O		$\times$		$\times$	
W			$\times$		
Do		$\times$			

Observe that the conflicts-with relation is symmetric, i.e. if Op conflicts-with Op', then Op' conflicts-with Op.

We may now use the definition of conflicting modalities to specify what it means for two ground action status literals to conflict.

Definition 3.8 (conflicting action status literals) *Suppose L_i, L_j are two action status literals. L_i is said to* conflict with *L_j iff*

- *L_i, L_j are unifiable and their modalities conflict, or*
- *L_i, L_j are of the form $L_i = Op(\alpha(\vec{t}))$ and $L_j = \neg Op'(\alpha(\vec{t'}))$, where $Op(\alpha(\vec{t}))$ and $Op'(\alpha(\vec{t'}))$ are unifiable, and the entry "$\times$" is in the following table at row Op and column $\neg Op'$:*

$Op \setminus \neg Op'$	$\neg$**P**	$\neg$**F**	$\neg$**O**	$\neg$**W**	$\neg$**Do**
P	$\times$				
F		$\times$			
O	$\times$		$\times$		$\times$
W				$\times$	
Do	$\times$				$\times$

For example, the action status atoms $\mathbf{F}\alpha(a,b,X)$ and $\mathbf{P}\alpha(Z,b,c)$ conflict. However, $\mathbf{F}\alpha(a,b,X)$ and $\neg\mathbf{P}\alpha(Z,b,c)$ do not conflict. Furthermore, $\neg\mathbf{P}\alpha(Z,b,c)$ and $\mathbf{Do}\alpha(Z,b,c)$ conflict, while the literals $\mathbf{P}\alpha(Z,b,c)$ and $\neg\mathbf{Do}\alpha(Z,b,c)$ do not conflict. As these examples show, the conflicts-with relation is *not* symmetric when applied to (complemented) action status literals. Before defining what it means for two rules to conflict, we point out that an agent's state is constantly changing. Hence, when our definition says that an agent program does not conflict, then this must apply not just to the current state, but to all possible states the agent can be in. We will first define conflicts with respect to a single state, and then define a conflict free program to be one that has no conflicts in all possible states.

Definition 3.9 (conflicting rules w.r.t. a state) *Consider two rules r_i, r_j (whose variables are standardized apart) having the form*

$$\begin{array}{rcl} r_i : Op_i(\alpha(\vec{t})) & \leftarrow & B(r_i) \\ r_j : Op_j(\beta(\vec{t'})) & \leftarrow & B(r_j) \end{array}$$

We say that r_i and r_j conflict *with respect to an agent state $\mathcal{O}_S$ iff Op_i conflicts with Op_j, and there is a substitution θ such that:*

1. *$\alpha(\vec{t}\theta) = \beta(\vec{t'}\theta)$;*

2. $(B_{cc}(r_i) \wedge B_{cc}(r_j))\theta\gamma$ *is true in* $\mathcal{O}_\mathcal{S}$ *for some substitution* γ *that causes* $(B_{cc}(r_i) \wedge Bcc(r_j))\theta$ *to become ground;*

3. *if* $Op_i \in \{\mathbf{P}, \mathbf{Do}, \mathbf{O}\}$ *(resp.,* $Op_j \in \{\mathbf{P}, \mathbf{Do}, \mathbf{O}\}$*) then* $\alpha(\vec{t}\theta)$ *(resp.,* $\beta(\vec{t'}\theta)$*) is executable in* $\mathcal{O}_\mathcal{S}$*; and*

4. $(B_{as}(r_i) \cup B_{as}(r_j))\theta$ *contains no pair of conflicting action status literals.*

Intuitively, the above definition says that for two rules to conflict in a given state, they must have a unifiable head and conflicting head-modalities, and furthermore, their bodies must be deontically consistent (under the unifying substitution) and their bodies' code call components must have a solution. The above definition merely serves as a stepping stone to defining a conflict free agent program.

Definition 3.10 (conflict free program) *An agent program,* $\mathcal{P}$*, is said to be* conflict free *iff it satisfies two conditions:*

1. *For every possible agent state* $\mathcal{O}_\mathcal{S}$ *there is no pair* r_i, r_j *of conflicting rules in* $\mathcal{P}$*.*

2. *For any rule* $Op_i(\alpha(\vec{t})) \leftarrow \ldots, (\neg)Op_j(\alpha_j(\vec{t'})), \ldots$ *in* $\mathcal{P}$*,* $Op_i(\alpha(\vec{t}))$ *and* $(\neg)Op_j(\alpha(\vec{t'}))$ *do not conflict.*

Unfortunately, as the following theorem shows, the problem of determining whether an agent program is conflict-free is undecidable, because checking the first condition is undecidable.

Theorem 3.2 (undecidability of conflict freedom checking) *The problem of deciding whether an input agent program* $\mathcal{P}$ *satisfies the first condition of conflict freedom is undecidable. Hence, the problem of deciding whether an input agent program* $\mathcal{P}$ *is conflict free is undecidable.*

The ability to check that an agent program is conflict free is very important. When an agent developer builds an agent, in general, s/he cannot possibly anticipate all the future states of the agent. Thus, the developer must build guarantees into the agent which ensure that no conflicts can possibly arise in the future. This can be done in two possible ways.

1. One way is to have a run-time conflict resolution procedure and guarantee that this conflict resolution procedure *always* resolves conflicts.

2. Another way is to identify at *compile time* a class of agent programs for which conflicts are guaranteed to never occur. This procedure moves the run-time (and potentially expensive) conflict resolution procedure to compile time, and hence increases run-time efficiency at the cost of some potential expressive power.

We choose the second way. As checking conflict freedom of agent programs is undecidable, we cannot hope for an effective algorithm to check conflict freedom. However, there are many possible ways to define *sufficient* conditions on agent programs that guarantee conflict freedom. If an agent developer encodes her agent program in a way that satisfies these sufficient conditions, then she is guaranteed that her agent is going to be conflict free. The concept of a conflict freedom test defined below provides such a mechanism.

Definition 3.11 (conflict freedom test) *A* conflict freedom test *is a function* cft *that takes as input any two rules* r_1, r_2*, and provides a boolean output such that: if* $\mathsf{cft}(r_1, r_2) = \mathtt{true}$*, then the pair* r_1, r_2 *satisfies the first condition of conflict freedom.*

Note that conflict freedom tests provide a sufficient (i.e., sound) condition for checking whether two rules r_1 and r_2 satisfy the first condition in the definition of conflict freedom. The second condition can be directly checked using the definition of what it means for two action status literals to conflict. This motivates the definition of a conflict-free agent program relative to a conflict freedom test below.

Definition 3.12 (conflict-free agent program w.r.t. cft**)** *An agent program $\mathcal{P}$ is* conflict free with respect to cft *iff for all pairs of distinct rules $r_i, r_j \in \mathcal{P}$, $\mathsf{cft}(r_i, r_j) =$ true, and all rules in $\mathcal{P}$ satisfy the second condition in the definition of conflict free programs.*

Intuitively, different choices of the function cft may be made, depending upon the complexity of such choices, and the accuracy of such choices (i.e., how often does a specific function cft return "false" on arguments (r_i, r_j) when in fact r_i, r_j do not conflict?). In the *IMPACT* Agent Development Environment (*IADE* for short), the agent developer can choose one of several conflict-freedom tests to be used for her application (and she can add new ones of her choosing to his list). Some instances of this test are given below.

Example 3.3 (Head-CFT, cft_h**)** Let r_i,r_j be two rules of the form

$$\begin{aligned} r_i : Op_i(\alpha(\vec{t})) &\leftarrow Body_i \\ r_j : Op_j(\beta(\vec{t'})) &\leftarrow Body_j. \end{aligned}$$

Now let the head conflict-freedom test cft_h be as follows:

$$\mathsf{cft}_h(r_i, r_j) = \begin{cases} \mathtt{true}, & \text{if either } Op_i, Op_j \text{ do not conflict, or} \\ & \alpha(\vec{t}) \text{ and } \beta(\vec{t'}) \text{ are not unifiable;} \\ \mathtt{false}, & \text{otherwise.} \end{cases}$$

Example 3.4 (Body-Code Call CFT, cft_{bcc}**)** Let us continue using the same notation as in Example 3.3. Now let the body-code conflict-freedom test cft_{bcc} be as follows:

$$\mathsf{cft}_{bcc}(r_i, r_j) = \begin{cases} \mathtt{true}, & \text{if either } Op_i, Op_j \text{ do not conflict, or} \\ & \alpha(\vec{t}) \text{ and } \beta(\vec{t'}) \text{ are not unifiable, or} \\ & Op_i, Op_j \text{ conflict and } \alpha(\vec{t}), \beta(\vec{t'}) \text{ are unifiable via mgu } \theta \text{ and} \\ & B_{cc}(r_1\theta), B_{cc}(r_2\theta) \text{ has a pair of contradictory code call atoms;} \\ \mathtt{false}, & \text{otherwise.} \end{cases}$$

Here "contradictory code call atoms" means that there exist code call atoms $\mathbf{in}(X, \mathtt{cc})$ and $\mathbf{not\_in}(X, \mathtt{cc})$ which occur in in $B_{cc}(r_1\theta) \cup B_{cc}(r_2\theta)$, or comparison atoms of the form $\mathtt{s}_1 = \mathtt{s}_2$ and $\mathtt{s}_1 \neq \mathtt{s}_2$; $\mathtt{s}_1 < \mathtt{s}_2$ and $\mathtt{s}_1 \geq \mathtt{s}_2$; etc.

Example 3.5 (Body-Modality-CFT, cft_{bm}**)** This head conflict-freedom test is similar to the previous one, except that action status atoms are considered instead. Now let cft_{bm} be as follows:

$$\mathsf{cft}_{bm}(r_i, r_j) = \begin{cases} \mathtt{true}, & \text{if } Op_i, Op_j \text{ do not conflict or} \\ & \alpha(\vec{t}), \beta(\vec{t'}) \text{ are not unifiable or} \\ & Op_i, Op_j \text{ conflict, and } \alpha(\vec{t}), \beta(\vec{t'}) \text{ are unifiable via mgu } \theta \text{ and} \\ & \text{literals } (\neg)Op_i\alpha(\vec{t''}) \text{ in } B_{as}(r_i\theta) \text{ for } i = 1, 2 \text{ exist such that} \\ & (\neg)Op_1 \text{ and } (\neg)Op_2 \text{ conflict;} \\ \mathtt{false}, & \text{otherwise.} \end{cases}$$

Example 3.6 (Precondition-CFT, cft_{pr}) Often, we might have action status atoms of the form $\mathbf{P}\alpha$, $\mathbf{Do}\alpha$, $\mathbf{O}\alpha$ in a rule. For a rule r_i as shown in Example 3.3, we denote by $r_i^\star$ the new rule obtained by appending to $Body_i$ the precondition of any action status atom of the form $\mathbf{P}\alpha, \mathbf{Do}\alpha, \mathbf{O}\alpha$ (appropriately standardized apart) from the head or body of r_i. Thus, suppose r is the rule

$$\mathbf{Do}\alpha(X,Y) \quad\leftarrow\quad \mathbf{in}(X, \mathbf{d}{:}\mathtt{f}(Y))\,\&\,\mathbf{P}\beta\,\&\,\mathbf{F}\gamma(Y).$$

Suppose $pre(\alpha(X,Y)) = \mathbf{in}(Y, \mathbf{d_1}{:}\mathtt{f}_1(X))$ and $pre(\beta) = \mathbf{in}(3, \mathbf{d_2}{:}\mathtt{f}_2())$. Then $r^\star$ is the rule

$$\begin{aligned}\mathbf{Do}\alpha(X,Y) \quad\leftarrow\quad & \mathbf{in}(X, \mathbf{d}{:}\mathtt{f}(Y))\,\&\,\mathbf{in}(Y, \mathbf{d_1}{:}\mathtt{f}_1(X))\,\&\,\mathbf{in}(3, \mathbf{d_2}{:}\mathtt{f}_2())\,\& \\ & \mathbf{P}\beta\,\&\,\mathbf{F}\gamma(Y).\end{aligned}$$

We now define cft_{pr} as follows.

$$\mathsf{cft}_{pr}(r_i, r_j) = \begin{cases} \mathtt{true}, & \text{if } \mathsf{cft}_{bcc}(r_i^\star, r_j^\star) = \mathtt{true}; \\ \mathtt{false}, & \text{otherwise.} \end{cases}$$

The following theorem tells us that whenever we have actions that have safe preconditions, then the rule $r^\star$ obtained as described above from a safe (resp., strongly safe) rule is also a safe (resp., strongly safe).

Theorem 3.3 *Suppose r is a rule, and $\alpha(\vec{Y})$ is an action such that some atom $Op\alpha(\vec{t})$ appears in r's body where $Op \in \{\mathbf{P}, \mathbf{O}, \mathbf{Do}\}$. Then:*

1. *If r is safe and $\alpha(\vec{Y})$ has a safe precondition modulo the variables in $\vec{Y}$, then $r^\star$ is safe.*
2. *If r is strongly safe and $\alpha(\vec{Y})$ has a strongly safe precondition modulo $\vec{Y}$, then $r^\star$ is strongly safe.*

Note. Throughout the rest of this paper, we will assume that an arbitrary, but fixed conflict-freedom test cft is used.

3.3 Deontic stratification

In this section, we define the concept of what it means for an agent program $\mathcal{P}$ to be *deontically stratified*—this definition extends the classical notion of stratification in logic programs introduced by [4]. The first concept we define is that of a layering function.

Definition 3.13 (layering function) *Let $\mathcal{P}$ be an agent program. A layering function ℓ is a function $\ell : \mathcal{P} \to \mathbb{N} = \{0, 1, 2, \dots\}$ which assigns a nonnegative integer to each rule in $\mathcal{P}$.*

Any layering function groups rules naturally into layers as defined below.

Definition 3.14 (layers of an agent program) *If $\mathcal{P}$ is an agent program, and ℓ is a layering function over $\mathcal{P}$, then the i-th layer of $\mathcal{P}$ with respect to ℓ, denoted $\mathcal{P}_i^\ell$, is defined as:*

$$\mathcal{P}_i^\ell \quad=\quad \{r \in \mathcal{P} \mid \ell(r) = i\}.$$

When ℓ is clear from the context, we will write $\mathcal{P}_i$ instead of $\mathcal{P}_i^\ell$.

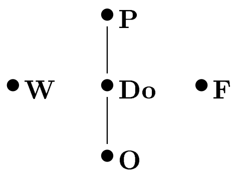


Figure 2: Modality ordering

Example 3.7 Consider the set of agent program rules listed below.

r_1: $\mathbf{P}\alpha(X,Y) \leftarrow \mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,b))\,\&\,\mathbf{in}(Y,\mathcal{S}{:}\mathtt{g}(X.\mathit{field},b,c))$

r_2: $\mathbf{P}\beta(X,a) \leftarrow \mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,a))$

r_3: $\mathbf{P}\beta(X,a) \leftarrow \mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,b))\,\&\,\neg\mathbf{Do}\gamma(X)$

r_4: $\mathbf{F}\gamma(Y) \leftarrow \mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,b))$

r_5: $\mathbf{F}\gamma(Y) \leftarrow \mathbf{P}\alpha(a,Y)\,\&\,\neg\mathbf{Do}\beta(a,Y)$.

Consider the agent program $\mathcal{P}_1 = \{r_i \mid 1 \leq i \leq 4\}$. The layering function ℓ_1 which assigns 0 to rule r_4 and 1 to the rules r_1, r_2, r_3 induces the program layers $\mathcal{P}_0^{\ell_1} = \{r_4\}$ and $\mathcal{P}_1^{\ell_1} = \{r_1, r_2, r_3\}$. Likewise, the function ℓ_1' which assigns 0 to rules r_1, r_4 and 1 to rules r_2, r_3 is a layering function.

Similarly, consider the agent program $\mathcal{P}_2 = \{r_i \mid 1 \leq i \leq 5\}$. An example of a layering function is one that assigns 0 to all rules.

Using the concept of a layering function, we would like to define what a *weakly regular* agent program is. Before doing so, we introduce a simple ordering on modalities.

Definition 3.15 (modality ordering) *The partial ordering "$\leq$" on the set of deontic modalities $M = \{\mathbf{P}$, $\mathbf{O}$, $\mathbf{Do}$, $\mathbf{W}$, $\mathbf{F}\}$ is defined as follows (see Figure 2): $\mathbf{O} \leq \mathbf{Do}$, $\mathbf{O} \leq \mathbf{P}$, $\mathbf{Do} \leq \mathbf{P}$, and $Op \leq Op$, for each $Op \in M$. Furthermore, for ground action status atoms A and B, we define that $A \leq B$ iff $A = Op\alpha$, $B = Op'\alpha$, and $Op \leq Op'$ all hold.*

Intuitively, the ordering reflects deontic consequence of one modality from another under the policy that each obligation is strictly obeyed, and that taking an action implies that it is permitted to execute it. We are now ready to define what it means for an agent program to be deontically stratifiable.

Definition 3.16 (deontically stratifiable agent program) *An agent program $\mathcal{P}$ is* deontically stratifiable *iff there exists a layering function ℓ such that:*

1. *For every rule $r_i :\ Op_i(\alpha(\vec{t})) \leftarrow \ldots, Op_j(\beta(\vec{t'})), \ldots$ in $\mathcal{P}_i^{\ell}$, if $r :\ Op(\beta(\vec{t''})) \leftarrow \ldots$ is a rule in $\mathcal{P}$ such that $\beta(\vec{t'})$ and $\beta(\vec{t''})$ are unifiable and $Op \leq Op_j$, then $\ell(r) \leq \ell(r_i)$.*

2. *For every rule $r_i :\ Op_i(\alpha(\vec{t})) \leftarrow \ldots, \neg Op_j(\beta(\vec{t'})), \ldots$ in $\mathcal{P}_i$, if $r :\ Op(\beta(\vec{t''})) \leftarrow \ldots$ is a rule in $\mathcal{P}$ such that $\beta(\vec{t'})$ and $\beta(\vec{t''})$ are unifiable and $Op \leq Op_j$, then $\ell(r) < \ell(r_i)$.*

Any such layering function ℓ is called a witness *to the stratifiability of $\mathcal{P}$.*

The following example presents a couple of agent programs, and discusses why they are (or are not) deontically stratifiable.

Example 3.8 Consider the agent program $\mathcal{P}_1$ given in Example 3.7. Clearly, $\mathcal{P}_1$ is deontically stratifiable: the layering function ℓ_1 of Example 3.7, for instance, is a witness to the deontic stratifiability of $\mathcal{P}_1$. The agent program $\mathcal{P}_2$ from Example 3.7 is also deontically stratifiable. To see this, note that condition (1) of deontic stratifiability requests $\ell(r_1) \leq \ell(r_5)$, while condition (2) imposes no restriction on ℓ. Thus, the layering ℓ_2 assigning 0 to all rules is a witness to the deontic stratifiability of $\mathcal{P}_2$.

An important note at this point is that rules with the same head may end up in different layers of a witness to a deontic stratification. E.g., the layering $\ell_2'(r_i) = i$ is another witness to the deontic stratifiability of $\mathcal{P}_2$, in which the rules r_2 and r_3, as well as r_4 and r_5 reside in different layers.

For an example of an agent program $\mathcal{P}$ which is not deontically stratifiable, consider the following two rules:

$$
\begin{aligned}
s_1 : &\quad \mathbf{Do}\alpha \leftarrow \neg\mathbf{P}\beta \\
s_2 : &\quad \mathbf{O}\beta \leftarrow \neg\mathbf{P}\alpha.
\end{aligned}
$$

According to condition 2 in the definition of deontic stratifiability, s_1 must be in a higher layer than s_2 since $\neg\mathbf{P}\beta$ occurs in the body of s_1, $\mathbf{O}\beta$ occurs in the head of s_2, and $\mathbf{O} \leq \mathbf{P}$. Similarly, s_2 must be in a higher layer than s_1 because $\neg\mathbf{P}\alpha$ occurs in the body of s_2, $\mathbf{Do}\alpha$ occurs in the head of s_1, and $\mathbf{Do} \leq \mathbf{P}$.

It is worth noting that if $\mathcal{P}$ is deontically stratifiable, then condition (2) in the definition of a conflict free agent program (Definition 3.10) is immediately true. Informally speaking, if we have a positive literal in the body of a rule, then the rule can only fire if that literal is derived; this means that heads conflict; otherwise, if the literal is negative, then the rule must be in a lower layer than itself, which cannot be the case.

3.3.1 canonical layering

As we have seen in the preceding discussion, an agent program may have multiple witnesses to its deontic stratifiability, and each of these witnesses yields a *different layering*. In this subsection, we will define what we call a *canonical layering* of a regular agent program $\mathcal{P}$.

Given an agent program $\mathcal{P}$, we denote by $\mathsf{wtn}(\mathcal{P})$ the set of all witnesses to the deontic stratifiability of $\mathcal{P}$. The *canonical layering* of $\mathcal{P}$, denoted $\mathsf{can}^{\mathcal{P}}$ is defined as follows.

$$
\mathsf{can}^{\mathcal{P}}(r) \quad = \quad \min\{\ell_i(r) \mid \ell_i \in \mathsf{wtn}(\mathcal{P})\}.
$$

Example 3.9 Let us return to the agent program $\mathcal{P}_2$ in Example 3.7. Both the layering functions ℓ_2 and ℓ_2' presented there are witnesses to the stratifiability of $\mathcal{P}_1$. Recall that ℓ_1 assigns 0 to all rules; thus, it is clearly the canonical layering associated with $\mathcal{P}_2$.

The following proposition asserts that $\mathsf{can}^{\mathcal{P}}$ is indeed always a witness to the deontic stratifiability of an agent program $\mathcal{P}$.

Proposition 3.4 *Let $\mathcal{P}$ be an agent program which is deontically stratifiable. Then* $\mathsf{can}^{\mathcal{P}} \in \mathsf{wtn}(\mathcal{P})$, *i.e.* $\mathsf{can}^{\mathcal{P}}$ *is a witness to the deontic stratifiability of* $\mathcal{P}$.

Note. Throughout this paper, whenever we discuss a *WRAP*, unless stated otherwise we will use its canonical layering,

3.4 Weakly regular agent programs (*WRAPs*)

We are now almost ready to define a weakly regular agent program. It is important to note that weak regularity depends upon a variety of parameters including a $\mathbf{FINTAB}$ and a conflict freedom implementation. In addition, we need a definition of what it means for an action to be strongly safe.

Definition 3.17 (strongly safe action) *An action $\alpha(\vec{X})$ is said to be* strongly safe *with respect to* $\mathbf{FINTAB}$ *if its precondition is strongly safe modulo $\vec{X}$, and each code call from the add list and delete list is strongly safe modulo $\vec{Y}$ where $\vec{Y}$ includes all root variables in $\vec{X}$ as well as in the precondition of α.*

The intuition underlying strong safety is that we should be able to check whether a (ground) action is safe by evaluating its precondition. If so, we should be able to evaluate the effects of executing the action.

We now define a weakly regular agent program.

Definition 3.18 (weakly regular agent program) *Let $\mathcal{P}$ be an agent program,* $\mathbf{FINTAB}$ *a finiteness table, and* cft *a head conflict-free test. Then $\mathcal{P}$ is called a* weakly regular agent program *(WRAP) with respect to* $\mathbf{FINTAB}$ *and* cft*, iff the following three conditions all hold:*

1. **Strong Safety***: All rules in $\mathcal{P}$ and actions α in the agent's action base are strongly safe with respect to* $\mathbf{FINTAB}$.
2. **Conflict-Freedom***: $\mathcal{P}$ is conflict free under* cft.
3. **Deontic Stratifiability***: $\mathcal{P}$ is deontically stratifiable.*

Example 3.10 Consider the actions α, β, γ of Example 3.7. These actions may have the preconditions, add and delete lists shown in Table 2. Clearly, each of these actions is strongly safe with respect to any $\mathbf{FINTAB}$, because each variable in the code calls is an action parameter.

Action	Part	List
$\alpha(X,Y)$	Pre	$\mathbf{in}(Y, \mathcal{S}{:}\mathtt{f}(c,c))$
	Add	$\mathbf{in}(Y, \mathcal{S}{:}\mathtt{f}(a,c))$
	Del	$\mathbf{in}(Y, \mathcal{S}{:}\mathtt{f}(c,c))$
$\beta(X,Y)$	Pre	$X = a \,\&\, \mathbf{in}(Y, \mathcal{S}{:}\mathtt{f}(a,c))$
	Add	$\mathbf{in}(Y, \mathcal{S}{:}\mathtt{f}(c,c))$
	Del	$\mathbf{in}(Y, \mathcal{S}{:}\mathtt{f}(a,c))$
$\gamma(X)$	Pre	$\mathbf{in}(X, \mathcal{S}{:}\mathtt{f}(c,c))$
	Add	$\emptyset$
	Del	$\mathbf{in}(X, \mathcal{S}{:}\mathtt{f}(c,c))$

Table 2: Action specification

Suppose $\mathbf{FINTAB}$ includes an entry $(a, \flat)$ for $\mathcal{S}$ and function $\mathtt{f}$ and an entry $(\flat, b, \flat)$ for $\mathcal{S}$ and $\mathtt{g}$. Then, each of the rules $r_1, \ldots, r_5$ in Example 3.7 is strongly safe with respect to $\mathbf{FINTAB}$. Thus, both agent programs $\mathcal{P}_1$ and $\mathcal{P}_2$ satisfy the first condition of a weakly regular agent program.

Under the head conflict-free test cft_h, the programs $\mathcal{P}_1$ and $\mathcal{P}_2$ are both conflict free; thus, the second condition of weak regularity is also satisfied.

Finally, as we have argued in Example 3.8, both $\mathcal{P}_1$ and $\mathcal{P}_2$ are deontic stratifiable.
Thus, both $\mathcal{P}_1$ and $\mathcal{P}_2$ are weakly regular with respect to $\mathbf{FINTAB}$ and cft_h.

3.5 Weakly regular agents

The framework in [17] specifies that in addition to an agent program, each agent has an associated set $\mathcal{IC}$ of integrity constraints, specifying conditions that an agent state must satisfy, and action constraints $\mathcal{AC}$, which describe conditions under which a certain collection of actions may not be concurrently executed). In order for an agent to evaluate what it must do in a given state, the ability to effectively or even polynomially evaluate the agent program is not enough—polynomial evaluation of the integrity and action constraints is also required.

Definition 3.19 (strongly safe integrity and action constraint) *An integrity constraint of the form $\psi \Rightarrow \chi$ is* strongly safe *iff ψ is strongly safe and χ is strongly safe modulo the root variables in ψ. An action constraint $\{\alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)\} \hookleftarrow \chi$ is* strongly safe *iff χ is strongly safe.*

The following example presents some action and integrity constraints, together with a specification of which ones are strongly safe.

Example 3.11 For example, using the same code calls in $\mathcal{P}_1$ of Example 3.7, we may have an associated integrity constraint

$$\mathbf{in}(X, \mathcal{S}{:}\mathtt{f}(a,b)) \quad \Rightarrow \quad \mathbf{in}(X, \mathcal{S}{:}\mathtt{f}(a,c)).$$

This integrity constraint says that in any given state of the agent, if X is in the set of objects returned by executing the code call $\mathcal{S}{:}\mathtt{f}(a,b)$, then X must also be in the set of objects returned by executing the code call $\mathcal{S}{:}\mathtt{f}(a,c)$. Under the same assumptions as in Examples 3.7 and its continuation, we see that this integrity constraint is strongly safe.

Consider the following action constraint:

$$\{\alpha(X,Y), \beta(X,Z)\} \quad \hookleftarrow \quad \mathbf{in}(X, \mathcal{S}{:}\mathtt{f}(a,b)).$$

This action constraint says that we cannot concurrently execute two actions $\alpha(X,Y), \beta(X,Z)$ whenever the current state causes X to be in the result of executing the code call $\mathcal{S}{:}\mathtt{f}(a,b)$. Under the assumptions of our ongoing example, we notice that this is a strongly safe action constraint, because the code call condition $\mathbf{in}(X, \mathcal{S}{:}\mathtt{f}(a,b))$ is strongly safe.

Last, but not least, the notion of concurrency used by the agent must conform to strong safety.

Definition 3.20 (strongly safe notion of concurrency) *A notion of concurrency, **conc**, is said to be* strongly safe *iff for every set $\mathcal{A}$ of actions and agent state $\mathcal{O}_\mathcal{S}$, it holds that if all members of $\mathcal{A}$ are strongly safe, then so is **conc**$(\mathcal{A}, \mathcal{O}_\mathcal{S})$.*

In this paper, we will not go into details of different notions of concurrency; [17] provides three alternative examples of such notions of concurrency, and many more are possible.

Definition 3.21 (weakly regular agent) *An agent* $\mathbf{a}$ *is* weakly regular *iff its associated agent program is weakly regular and the action constraints, integrity constraints, and the notion of concurrency in the background are all strongly safe.*

4 Algorithms for Checking Weak Regularity

In this section, we present an algorithm, **Check_WRAP**, for checking whether a given agent program $\mathcal{P}$ is weakly regular. As we have already discussed methods for checking safety and strong safety of code call conditions earlier in the paper, we will focus our discussion on checks for the conflict-freedom and deontic stratifiability conditions. Note that these two conditions are *closely interlinked*. It is easy to use the strong safety check algorithm to check whether an agent is safe because this algorithm can be directly used to verify whether an action is strongly safe, an action constraint is strongly safe, and an integrity constraint is strongly safe.

The conflict-freedom conditions can be readily checked, as they do not depend on a layering ℓ. The function $\mathsf{cft}(r_i, r_j)$ is used to check the first conflict freedom condition, while adapted efficient unification algorithms, e.g. [28], may be used to check the second condition. However, the check for deontic stratification conditions is more complex. Different methods can be applied, and we outline here a method which is based on computing the (maximal) strongly connected components of a graph $G = (V, E)$. This method extends similar methods for finding stratifications of logic programs, cf. [39].

A *strongly connected component* (*SCC*) of a directed graph $G = (V, E)$ is a maximal set $C \subseteq V$ of vertices (maximal with respect to set inclusion) such that between every pair of vertices $v, v' \in C$, there exists a path from v to v' in G involving only vertices from C. For any graph G, we can defined its *supergraph* $S(G) = (V^*, E^*)$ as the graph whose vertices are the strongly connected components of G, and such that there is an edge $C \to C'$ in E^*, if there is an edge from some vertex $v \in C$ to some vertex $v' \in C'$ in the graph G. Note that the supergraph $S(G)$ is acyclic. Using Tarjan's algorithm, see e.g. [27], the SCCs of G, and thus the supergraph $S(G)$, is computable in time $O(|V| + |E|)$, i.e., in linear time from G.

The method for checking the stratification conditions is to build a directed graph $G = (V, E)$ whose vertices V are the rules in $\mathcal{P}$. There is an edge from rule r to rule r' if $\ell(r') \leq \ell(r)$ follows from one of the two deontic stratification conditions. From the SCCs of G, we may easily check whether $\mathcal{P}$ is deontically stratified, and from $S(G)$, a layering $\ell \in \mathtt{wtn}(\mathcal{P})$ witnessing this fact can be obtained by a variant of topological sorting. The following example discusses the graph and supergraph associated with an example agent program, and illustrates the intuition underlying this algorithm.

Example 4.1 Consider the two example programs $\mathcal{P}_1$ and $\mathcal{P}_2$ in Example 3.7. Suppose we use the conflict freedom implementation cft_h, and suppose all code calls here are finite. Notice that in the case of $\mathcal{P}_1$, the associated graph $G_1 = G(\mathcal{P}_1)$ has no edges, while in the case of $\mathcal{P}_2$, the graph $G_2 = G(\mathcal{P}_2)$ has the single edge $r_5 \to r_1$. Hence, in both cases, the supergraph $S(G_1)$ resp. $S(G_2)$ consists of vertices labeled with a singleton set of rules. The supergraph $S(G_1)$ has no edges, while $S(G_2)$ has a single edge $\{r_5\} \to \{r_1\}$.

The algorithm, **Check_WRAP**, used to check whether an agent program $\mathcal{P}$ is weakly regular is shown below. Example 4.2 shows how it works on two example agent programs.

Example 4.2 Let us return to the agent programs $\mathcal{P}_1$ and $\mathcal{P}_2$ in Example 3.7, and use $\mathbf{FINTAB}$ from Example 3.10. Then for both $\mathcal{P}_1$ and $\mathcal{P}_2$, all tests in steps 1 through 3 of **Check_WRAP** are negative, i.e., Step 4 is reached and the graph G_1 (resp., G_2) is built, and subsequently the supergraph $S(G_1)$ (resp., $S(G_2)$) as described in Example 4.1. In case of $S(G_1)$, all $C \in V^*$ have out-degree 0 which means that Step 7 sets $\ell(r) = 0$ for each rule r in $\mathcal{P}_1$ and the algorithm returns ℓ in Step 9. In case of $S(G_2)$, Step 7 sets $\ell(r_i) = 0$ for $1 \leq i \leq 4$, and sets $\ell(r_5) = 1$ in the next iteration, and then returns ℓ. Both $\mathcal{P}_1$ and $\mathcal{P}_2$ are deontically stratified via this layering.

The following theorem states that algorithm **Check_WRAP** is correct.

Algorithm 4.1 ***Check_WRAP($\mathcal{P}$)***

($\star$ input: agent program $\mathcal{P}$ (cft and $\mathbf{FINTAB}$ are further parameters) $\star$)
($\star$ output: a layering $\ell \in \mathtt{wtn}(\mathcal{P})$, if $\mathcal{P}$ is regular and "no" otherwise $\star$)

1. **If** *some action α or rule r in $\mathcal{P}$ is not strongly safe* **then return** *"no" and* **halt**.
2. **If** *some rules $r : Op(\alpha(\vec{X}))$ and $r' : Op'(\alpha(\vec{Y}))$ in $\mathcal{P}$ exist such that $\mathsf{cft}(r, r') = false$,* **then return** *"no" and* **halt**.
3. **If** *a rule $r : Op_i(\alpha(\vec{X})) \leftarrow \ldots, (\neg)Op_j(\alpha(\vec{Y})), \ldots$ is in $\mathcal{P}$ such that $Op_i(\alpha(\vec{X}))$ and $Op_j(\alpha(\vec{Y}))$ conflict,* **then return** *"no" and* **halt**.
4. *Build the graph $G = (V, E)$, where $V = \mathcal{P}$ and an edge $r_i \to r$ is in E for each pair of rules r_i and r as in the two stratifiability conditions.*
5. *Compute, using Tarjan's algorithm, the supergraph $S(G) = (V^*, E^*)$ of G.*
6. **If** *some rules r_i, r as in the second stratifiability condition exist such that $r_i, r \in C$ for some $C \in V^*$,* **then return** *"no" and* **halt else** *set $i := 0$.*
7. *For each $C \in V^*$ having out-degree 0 (i.e., no outgoing edge) in $S(G)$, and each rule $r \in C$, define $\ell(r) := i$.*
8. *Remove each of the above C's from $S(G)$, and remove all incoming edges associated with such nodes in $S(G)$ and set $i := i + 1$;*
9. **If** *$S(G)$ is empty, i.e., $V^* = \emptyset$,* **then return** *ℓ and* **halt else** *continue at 7.*

Theorem 4.1 *For any conflict freedom test cft and finitentess table $\mathbf{FINTAB}$, given agent program $\mathcal{P}$,* **Check_WRAP**$(\mathcal{P})$ *returns a layering $\ell \in \mathtt{wtn}(\mathcal{P})$ if $\mathcal{P}$ is a WRAP, and returns "no" if $\mathcal{P}$ is not regular.*

Let us now analyze the complexity of algorithm **Check_WRAP**. We start by observing that steps 5. through 9. can be implemented to run in time linear in the size of G by using appropriate data structures, that is, in $O(|\mathcal{P}|^2)$ time (note that $S(G)$ is not larger than G).

In Step 1, checking whether an action α or a rule r is strongly safe can be done in time linear in the size of the description of α or the size of r, respectively, times the size of $\mathbf{FINTAB}$. If we suppose that the action base $\mathcal{AB}$ in the background and the $\mathbf{FINTAB}$ are fixed, this means that Step 1 is feasible in $O(\|\mathcal{P}\|)$ time, where $\|\mathcal{P}\|$ is the size of the representation of $\mathcal{P}$, i.e., in linear time. The time for Step 2 depends on the time required by $\mathsf{cft}(r, r')$—if $t_{\mathsf{cft}}(\mathcal{P})$ is an upper bound for the time spent on a call of $\mathsf{cft}(r, r')$, then Step 2 needs $O(|\mathcal{P}|^2 \cdot t_{\mathsf{cft}}(\mathcal{P}))$ time. Step 3 can be done in $O(\|\mathcal{P}\| t_u(\mathcal{P}))$ time, where $t_u(\mathcal{P})$ is the maximal time spent on unifying two atoms in $\mathcal{P}$. Finally, Step 4 can be done in $O(|\mathcal{P}|^2 t_u(\mathcal{P}))$ time.

Thus, extending the assumption for Step 1 by further assuming that atoms and rule bodies have size bounded by a constant—an assumption that is certainly plausible, since the number of literals in a rule body is not expected to exceed 20, say, and each literal will, as a string, hardly occupy more than 1024 characters—we obtain that **Check_WRAP**($\mathcal{P}$) can be executed in $O(|\mathcal{P}|^2 t_{\mathsf{cft}}(\mathcal{P}))$ time. This bound further decreases to $O(|\mathcal{P}| t_{\mathsf{cft}}(\mathcal{P}))$ time if for each action α and modality Op, only a few rules (bounded by a constant) with head $Op\alpha(\cdots)$ exist in $\mathcal{P}$. These assumptions on the "shape" of the rules and the program seem to be reasonable with respect to agent programs in practice. Thus, we may expect that **Check_WRAP** runs in $O(|\mathcal{P}| \cdot t_{\mathsf{cft}}(\mathcal{P}))$ time. In particular, for an efficient implementation of a cft as in Examples 3.3–3.5, it runs in $O(|\mathcal{P}|)$ time, i.e., in linear time in the number of rules.

We conclude this subsection with the remark that **Check_WRAP** can be modified to compute the canonical layering $\mathsf{can}^{\mathcal{P}}$ as follows. For each node $C \in V^*$, use two counters $out(C)$ and $block(C)$, and initialize them in Step 5 to the number of outgoing edges from C in E^*. Steps 7 and 8 of **Check_WRAP** are replaced by the following steps:

$7'$. Set $U := \emptyset$;
while some $C \in V^*$ exists such that $block(C) = 0$ **do**
  $U := U \cup \{C\}$;
  Set $out(C') := out(C') - 1$ for each $C' \in V^*$ such that $C' \to C$;
  Set $block(C') := block(C') - 1$ for each $C' \in V^*$ such that $C' \to C$ due to the first stratification condition but not the second stratification condition.
for each rule r in $\bigcup U$ **do** $\ell(r) := i$;

$8'$. Set $i := i + 1$;
Remove each $C \in U$ from $S(G)$, and set $block(C) := out(C)$ for each retained node C.

When properly implemented, the steps $7'$ and $8'$ can be executed in linear time in the size of $S(G)$, and thus of G. Thus, the upper bounds on the time complexity of **Check_Regular** discussed above also apply to the variant which computes the canonical layering.

Thus, at this stage we have provided a complete definition of a weakly regular agent program, together with an efficient compile-time algorithm for determining whether an agent program is weakly regular or not.

5 Semantics and Evaluation of *WRAPs*

In this section, we focus on the run-time behavior of *WRAPs*. A weakly regular agent must continuously compute reasonable status sets. We provide an algorithm for computing reasonable status sets. In particular, computing a reasonable status set of *WRAPs*—if one exists—is possible by means of a fixpoint iteration procedure, followed by a check to see whether the action and integrity constraints of the agent are satisfied. Later, in Section 7, we will provide an architecture and an implementation strategy for *WRAPs*. For this purpose, we shall restrict *WRAPs* by imposing a boundedness condition on recursion, which then gives rise to *regular agent programs*. Under suitable conditions, such programs can be evaluated in polynomial time.

We first recall the operator $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}$ from [17], and that a positive agent program $\mathcal{P}$ has at most one reasonable status set, which coincides with its rational status set (if the latter exists). This reasonable status set can be discovered by computing the least fixpoint $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S} \uparrow \omega(\emptyset)$ of the operator $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}$, and checking feasibility of it. In case $\mathcal{P}$ is a *WRAP*, it is sufficient to check whether $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S} \uparrow \omega(\emptyset)$ satisfies the integrity and actions constraints associated with the agent. If no such constraints exist, then $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S} \uparrow \omega(\emptyset)$ is guaranteed to be the (unique) reasonable status set of $\mathcal{P}$.

After that, we consider general *WRAPs* and show that each of them has a unique reasonable status set under the same ramifying assumptions. We also present an algorithm for computing this reasonable status set.

5.1 An operator on agent programs

Definition 5.1 **($\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}$ [17])** *Let $\mathcal{P}$ be an agent program and $\mathcal{O}_S$ be an agent state. Then for any status set S, the operator $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}$ is defined as follows:*

$$\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}(S) = \mathbf{App}_{\mathcal{P},\mathcal{O}_S}(S) \cup \textbf{A-Cl}(S),$$

where $\mathbf{App}_{\mathcal{P},\mathcal{O}_S}(S)$ is the operator as in [17] (see Appendix B).

Intuitively, if we know that all status atoms in S are true, then the $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}(S)$ operator computes all status atoms that are derivable from the agent program by applying the rules in it once, and by closing S under the action closure rules introduced in Section 2.2.

Example 5.1 Let us return to the running example introduced in Example 3.7. Consider an agent state in which the code calls behave as follows:

$$\begin{aligned} \mathcal{S}{:}\mathtt{f}(a,b) &= \{a,c\}. \\ \mathcal{S}{:}\mathtt{f}(a,a) &= \{c\}. \\ \mathcal{S}{:}\mathtt{g}(\mathtt{a.field},b,c) &= \{c\}. \\ \mathcal{S}{:}\mathtt{g}(\mathtt{c.field},b,c) &= \{a,c\}. \end{aligned}$$

Furthermore, suppose here that the actions α, β, and γ have empty preconditions, and take $S = \{\mathbf{Do}\alpha(a,b)\}$. Then

$$\begin{aligned} \mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}(S) = \{ & \mathbf{P}\alpha(a,c), \mathbf{P}\alpha(c,a), \mathbf{P}\alpha(c,c), \\ & \mathbf{P}\beta(a,a), \mathbf{P}\beta(c,a), \mathbf{F}\gamma(a), \mathbf{F}\gamma(b), \mathbf{F}\gamma(c)\} \cup \{\mathbf{Do}\alpha(a,b), \mathbf{P}\alpha(a,b)\}. \end{aligned}$$

The operator $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow$defined below iteratively applies the $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}$ operator.

Definition 5.2 **($\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow i$ and $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow\omega$ operators)** *Suppose $\mathcal{P}$ is an agent program and $\mathcal{O}_S$ an agent state, and S is a status set. Then, the operators $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow i$, $i \geq 0$, and $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow\omega$ are defined as follows.*

$$\begin{aligned} \mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow 0(S) &= S, \\ \mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow (i+1)(S) &= \mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}(\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow i(S)), \\ \mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow \omega(S) &= \bigcup_{i=0}^{\infty} \mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow i(S). \end{aligned}$$

An example of the behavior of the $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow$operator is given below.

Example 5.2 Let us continue with the agent program of Example 3.7 and $S = \{\mathbf{Do}\alpha(a,b)\}$. Then,

$$\begin{aligned} \mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow 0(S) = \{ & \mathbf{Do}\alpha(a,b)\}, \\ \mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow 1(S) = \{ & \mathbf{P}\alpha(a,c), \mathbf{P}\alpha(c,a), \mathbf{P}\alpha(c,c), \\ & \mathbf{P}\beta(a,a), \mathbf{P}\beta(c,a), \mathbf{F}\gamma(a), \mathbf{F}\gamma(b), \mathbf{F}\gamma(c)\} \cup \{\mathbf{Do}\alpha(a,b), \mathbf{P}\alpha(a,b)\}, \\ \mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow 2(S) = & \ \mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow 1(S); \end{aligned}$$

thus, $\mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow\omega(S) = \mathcal{T}_{\mathcal{P}}^{\mathcal{O}_S}\uparrow 1(S)$.

As mentioned earlier on in the paper, we drop the superscript $\mathcal{O}_S$ when the agent's state is clear from the context.

We remark that the increase $\mathcal{T}_\mathcal{P} \uparrow 0(S) \subseteq \mathcal{T}_\mathcal{P} \uparrow 1(S) \subseteq \cdots$ of $\mathcal{T}_\mathcal{P} \uparrow i(S)$ in the previous example is not incidental, and a consequence of the properties of $\mathcal{T}_\mathcal{P}$. For later reference, we display some noticeable properties of $\mathcal{T}_\mathcal{P}$ and $\mathcal{T}_\mathcal{P} \uparrow \omega$ in the next proposition, which can be easily established following the proof of similar results concerning the classical T_P operator in logic programming [25].

Proposition 5.1 *Suppose $\mathcal{P}$ is an agent program and $\mathcal{O}_S$ an agent state. Then*

1. *$\mathcal{T}_\mathcal{P}$ is* inflationary*, i.e., $S \subseteq \mathcal{T}_\mathcal{P}(S)$ holds for every status set S;*
2. *If $\mathcal{P}$ is positive, then $\mathcal{T}_\mathcal{P}$ is continuous, i.e. if $S_1 \subseteq S_2 \subseteq \cdots$ then $\mathcal{T}_\mathcal{P}(\bigcup_i S_i) = \bigcup_i \mathcal{T}_\mathcal{P}(S_i)$ and, as a consequence, also* monotone*, i.e. $S_1 \subseteq S_2$ implies that $\mathcal{T}_\mathcal{P}(S_1) \subseteq \mathcal{T}_\mathcal{P}(S_2)$;*
3. *If $\mathcal{P}$ is positive, then $\mathcal{T}_\mathcal{P} \uparrow \omega(\emptyset)$ is the least fixpoint of $\mathcal{T}_\mathcal{P}$.*

Observe that properties 2 and 3 do not hold for arbitrary, non-positive programs in general.

5.2 Semantics of a positive *WRAP*

The following known result tells us that for positive agent programs, only one status set, viz. $\mathcal{T}_\mathcal{P} \uparrow \omega(\emptyset)$ has any chance of being rational.

Proposition 5.2 (Eiter *et al.* [17]) *Let $\mathcal{P}$ be a positive agent program. Then, a status set S is a rational status set of $\mathcal{P}$ iff $S = \mathcal{T}_\mathcal{P} \uparrow \omega(\emptyset)$ and S is a feasible status set of $\mathcal{P}$.*

The preceding result guarantees that *positive* agent programs always have an iteratively computable least fixpoint. This fixpoint is a rational status set, and thus a reasonable status set, if S satisfies deontic consistency as well as the action constraints and integrity constraints. If the program is weakly regular, then we obtain the following result.

Proposition 5.3 *Let $\mathcal{P}$ be a* positive *agent program, and suppose that $\mathcal{P}$ is weakly regular. Then, $\mathcal{P}$ has at most one rational status set on $\mathcal{O}_S$, and $S = \mathcal{T}_\mathcal{P} \uparrow \omega(\emptyset)$ is the unique rational status set, iff S satisfies the action and the integrity constraints.*

A straightforward corollary of the above result is that when action constraints and integrity constraints are absent, then weakly regular agent programs are guaranteed to have a rational status set.

Though positive agent programs may appear to be unnecessarily restrictive, they are in fact very useful to express many complex agent applications. For example, we have built a large scale logistics application that manipulates (US) Army War Reserves data using positive agent programs. Below, we show a *WRAP* that generalizes a simple driving example given in [17].

Example 5.3 (Simple Driving Example Generalizing one in [17]) Consider an agent selecting—in a simplified setting—the driving lane for a car (`left_lane` or `right_lane`), according to the following agent program $\mathcal{P}$:

$$
\begin{aligned}
r_1:&\quad \mathbf{O}(\texttt{go\_right}) \leftarrow \\
r_2:&\quad \mathbf{O}(\texttt{drive}(\texttt{right\_lane})) \leftarrow \mathbf{Do}(\texttt{go\_right}) \\
r_3:&\quad \mathbf{F}(\texttt{drive}(X)) \leftarrow \mathbf{not\_in}(X, \mathbf{driving}:\texttt{free\_lanes}()) \\
r_4:&\quad \mathbf{Do}(\texttt{drive}(\texttt{left\_lane})) \leftarrow \mathbf{F}(\texttt{drive}(\texttt{right\_lane}))
\end{aligned}
$$

Here, the action $\texttt{go\_right}$ has no precondition and has the effect of selecting the rightmost lane. The action $\texttt{drive}(X)$ involves the action "drive in lane X" and has the precondition $\mathbf{in}(\texttt{X}, \mathbf{driving}{:}\texttt{free\_lanes}())$, where the code call $\mathbf{driving}{:}\texttt{free\_lanes}()$ returns the free lanes.

Rule r_1 says that we must select the rightmost lane, and r_2 that we must drive on the right lane if we follow the go-right policy; note that r_2 only fires if that lane is free. Rule r_3 says that it's forbidden to drive on a blocked lane, and r_4 says that we go on the left lane (provided it's free), if we can not use the right lane.

As the underlying domain is finite, it is easy to see that all rules and actions are strongly safe with respect to any $\mathbf{FINTAB}$ which states that each code call is finite. Clearly, $\mathcal{P}$ has no negative action status atoms in it, and thus it is deontically stratified. As for conflict-freedom, the head conflict-freedom test, cft_h, returns false when applied to r_2, r_3, and also when applied to r_3, r_4. Thus, the program is not weakly regular under cft_h. The same is true for the body-code call CFT and the body-modality CFT. However, the Precondition-CFT, cft_{pr}, returns true for all pairs of rules. Indeed, the extended rule $r_2^\star$ has the code call atom $\mathbf{in}(\texttt{right\_lane}, \mathbf{driving}{:}\texttt{free\_lanes}())$ in its body. As the heads of $r_2^\star$ and $r_3^\star$ are unifiable via $\theta = \{X/\texttt{right\_lane}\}$, the code call parts of $r_2^\star\theta$ and $r_3^\star\theta$ contain a contradictory pair of code call atoms. For the pair $r_3^\star$ and $r_4^\star$, a similar situation arises as the code call part of $r_4^\star$ contains an atom $\mathbf{in}(\texttt{left\_lane}, \mathbf{driving}{:}\texttt{free\_lanes}())$. As is easily seen, $\mathcal{P}$ also satisfies condition (2) of conflict-freedom, i.e., there are no head-body conflicts in a rule. Thus, $\mathcal{P}$ is weakly regular under Precondition-CFT. Assuming that there are no action and integrity constraints, Proposition 5.3 tells us that the program has one (and only one) rational status set on each possible agent state; this was shown ad hoc in [17] by a detailed case analysis.

5.3 Reasonable status sets of arbitrary *WRAPs*

We now generalize the characterization of a reasonable status set for a positive *WRAP* to an arbitrary *WRAPs*. We will define below an operator $\Gamma^\ell_{\mathcal{P},\mathcal{O}_S} \uparrow \omega$ that evaluates (from bottom to top) the layers of a *WRAP* generated by a layering function ℓ. Before doing so, we observe that for arbitrary agent programs $\mathcal{P}$, iterating the $\mathcal{T}_\mathcal{P}\uparrow i$ operator will eventually lead to a fixpoint $\mathcal{T}_\mathcal{P}\uparrow\omega(S)$ of the $\mathcal{T}_\mathcal{P}$ operator.

Proposition 5.4 *Suppose $\mathcal{P}$ is any agent program and $\mathcal{O}_S$ an agent state. Then, for every S, $\mathcal{T}_\mathcal{P}\uparrow\omega(S)$ is a fixpoint of $\mathcal{T}_\mathcal{P}$ and $\mathcal{T}_\mathcal{P}\uparrow\omega(S)$ is action closed.*

The operator $\Gamma^\ell_{\mathcal{P},\mathcal{O}_S} \uparrow i$ evaluates the layer i by computing the fixpoint $\mathcal{T}_{\mathcal{P}_i} \uparrow \omega$ for the program $\mathcal{P}_i$, starting from the result that has been computed at the previous layer $i-1$. Formally, the definition is as follows.

Definition 5.3 ($\Gamma^\ell_{\mathcal{P},\mathcal{O}_S}\uparrow i$ and $\Gamma^\ell_{\mathcal{P},\mathcal{O}_S}\uparrow\omega$ operators) *Suppose $\mathcal{P}$ is a deontically stratified agent program, which is witnessed by layering function ℓ, and suppose the layers of $\mathcal{P}$ induced by ℓ are $\mathcal{P}_0, \ldots, \mathcal{P}_k$. The operators $\Gamma^\ell_{\mathcal{P},\mathcal{O}_S}\uparrow i(S)$ and $\Gamma^\ell_{\mathcal{P},\mathcal{O}_S}\uparrow\omega(S)$ are defined as follows.*

$$\begin{aligned}
\Gamma^\ell_{\mathcal{P},\mathcal{O}_S}\uparrow 0(S) &= \mathcal{T}^{\mathcal{O}_S}_{\mathcal{P}_0}\uparrow\omega(S)\\
\Gamma^\ell_{\mathcal{P},\mathcal{O}_S}\uparrow (i+1)(S) &= \mathcal{T}^{\mathcal{O}_S}_{\mathcal{P}_{i+1}}\uparrow\omega(\Gamma^\ell_{\mathcal{P},\mathcal{O}_S}\uparrow i(S))\\
\Gamma^\ell_{\mathcal{P},\mathcal{O}_S}\uparrow\omega(S) &= \bigcup_{i=0}^{k}\Gamma^\ell_{\mathcal{P},\mathcal{O}_S}\uparrow i(S).
\end{aligned}$$

We will write $\Gamma^{\ell}_{\mathcal{P},\mathcal{O}_{\mathcal{S}}} \uparrow i$ and $\Gamma^{\ell}_{\mathcal{P},\mathcal{O}_{\mathcal{S}}} \uparrow \omega$ for $\Gamma^{\ell}_{\mathcal{P},\mathcal{O}_{\mathcal{S}}} \uparrow i(\emptyset)$ and $\Gamma^{\ell}_{\mathcal{P},\mathcal{O}_{\mathcal{S}}} \uparrow \omega(\emptyset)$, respectively. Furthermore, we will drop the subscript $\mathcal{O}_{\mathcal{S}}$ when it is clear from the context.

The following example illustrates the computation of $\Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$.

Example 5.4 Reconsider the rules in Example 3.7. Suppose that $\mathcal{P} = \{r_1, \ldots, r_4\}$ is partitioned by the layering ℓ in two layers $\mathcal{P}_0 = \{r_1, r_4\}$ and $\mathcal{P}_1 = \{r_2, r_3\}$. Suppose the code calls behave as described in Example 5.1. Then,

$$\begin{array}{lclcl} \Gamma^{\ell}_{\mathcal{P}} \uparrow 0 & = & \mathcal{T}_{\mathcal{P}_0} \uparrow \omega(\emptyset) & = & \{\mathbf{F}\gamma(a), \mathbf{F}\gamma(c), \mathbf{P}\alpha(a,c), \mathbf{P}\alpha(c,c), \mathbf{P}\alpha(c,a)\}, \\ \Gamma^{\ell}_{\mathcal{P}} \uparrow 1 & = & \mathcal{T}_{\mathcal{P}_1} \uparrow \omega(\Gamma^{\ell}_{\mathcal{P}} \uparrow 0) & = & \Gamma^{\ell}_{\mathcal{P}} \uparrow 0 \cup \{\mathbf{P}\beta(c,a), \mathbf{P}\beta(a,a)\}; \end{array}$$

Thus, $\Gamma^{\ell}_{\mathcal{P}} \uparrow \omega = \bigcup_{i=0}^{1} \Gamma^{\ell}_{\mathcal{P}} \uparrow i = \Gamma^{\ell}_{\mathcal{P}} \uparrow 1$.

The next theorem tells us that for *all* layerings $\ell \in \mathtt{wtn}(\mathcal{P})$, $\Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$ is a reasonable status set of any *WRAP* that has no associated action constraints or integrity constraints.

Theorem 5.5 *Suppose $\mathcal{P}$ is a WRAP. Let $\ell \in \mathtt{wtn}(\mathcal{P})$ be any witness to the regularity of $\mathcal{P}$. If $\mathcal{IC}$ and $\mathcal{AC}$ are both empty, then $\Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$ is a reasonable status set of $\mathcal{P}$ with respect to $\mathcal{O}_{\mathcal{S}}$.*

From this result, we can conclude that $\Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$ is a fixpoint of the global $\mathcal{T}_{\mathcal{P}}$ operator.

Corollary 5.6 *Let $\mathcal{P}$ be a WRAP and let $\ell \in \mathtt{wtn}(\mathcal{P})$. Then, $S = \Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$ is a fixpoint of $\mathcal{T}_{\mathcal{P}}$.*

The previous theorem shows that when a *WRAP* $\mathcal{P}$ has no associated integrity constraints and action constraints, then $\Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$ is guaranteed to be a reasonable status set of $\mathcal{P}$. The next result shows that any reasonable status set of $\mathcal{P}$ must be of this form, and in fact coincides with $\Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$.

Theorem 5.7 *Suppose $\mathcal{P}$ is a WRAP, and let $\ell \in \mathtt{wtn}(\mathcal{P})$. Let S be any reasonable status set of $\mathcal{P}$. Then, $S = \Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$.*

The following are immediate corollaries of the above result.

Corollary 5.8 *Suppose $\mathcal{P}$ is a WRAP and suppose ℓ_1, ℓ_2 are in $\mathtt{wtn}(\mathcal{P})$. Then $\Gamma^{\ell_1}_{\mathcal{P}} \uparrow \omega = \Gamma^{\ell_2}_{\mathcal{P}} \uparrow \omega$.*

Corollary 5.9 *Suppose $\mathcal{P}$ is a WRAP and let $\ell \in \mathtt{wtn}(\mathcal{P})$ be arbitrary. If $\Gamma^{\ell_1}_{\mathcal{P}} \uparrow \omega$ satisfies the action and integrity constraints $\mathcal{AC}$ and $\mathcal{IC}$, respectively, then $\Gamma^{\ell_1}_{\mathcal{P}} \uparrow \omega$ is the (unique) reasonable status of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$. Otherwise, $\mathcal{P}$ has no reasonable status set on $\mathcal{O}_{\mathcal{S}}$.*

Based on these results, the algorithm **Reasonable-SS** described below can be used to determine whether a reasonable status set for a regular agent program exists on an agent state, and if this is true, to find the unique such reasonable status set. The algorithm simply computes the iterative fixpoint $\Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$ and checks whether it satisfies the action constraints in the current agent state, and furthermore whether the new state to which the agent is taken by concurrently executing all actions α where $\mathbf{Do}\alpha \in \Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$ is consistent with the integrity constraints.

Algorithm **Reasonable_SS** can be executed on *WRAPs*—however, in the general case, there is no guarantee of termination. A *regular agent program* requires that in addition to the agent being weakly regular, its associated agent program must satisfy an extra condition called *boundedness.*

Algorithm 5.1 ***Reasonable_SS***($\mathcal{P}, \ell, \mathcal{IC}, \mathcal{AC}, \mathcal{O}_{\mathcal{S}}$)

($\star$ input is a regular agent consisting of a WRAP $\mathcal{P}$, a layering $\star$)
($\star$ $\ell \in \mathtt{wtn}(\mathcal{P})$, a strongly safe set $\mathcal{IC}$ of integrity constraints, $\star$)
($\star$ a strongly safe set $\mathcal{AC}$ of action constraints, and an agent state $\mathcal{O}_{\mathcal{S}}$ $\star$)
($\star$ output is a reasonable status set S of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$, if one exists, and "no" otherwise. $\star$)

1. $S := \Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$;
2. $\mathbf{Do}(S) := \{\alpha \mid \mathbf{Do}(\alpha) \in S\}$;
3. **while** $\mathcal{AC} \neq \emptyset$ **do**
   Select and remove some $ac \in \mathcal{AC}$;
   if ac *is not satisfied w.r.t.* $\mathbf{Do}(S)$ **then return** *"no" and* **halt**;
4. $\mathcal{O}'_{\mathcal{S}} :=$ *apply* ***conc***$(\mathbf{Do}(S), \mathcal{O}_{\mathcal{S}})$ *on* $\mathcal{O}_{\mathcal{S}}$; *($\star$ resulting successor state $\star$)*
5. **while** $\mathcal{IC} \neq \emptyset$ **do**
   Select and remove some $ic \in \mathcal{IC}$;
   if $\mathcal{O}'_{\mathcal{S}} \not\models ic$ **then return** *"no" and* **halt**.
6. **return** S *and* **halt**.

6 Regular Agent Programs

In this section, we define what it means for a *WRAP* to be *bounded*. A regular agent program then is a program which is weakly regular and bounded. Intuitively, boundedness means that by repeatedly unfolding the positive parts of the rules in the program, we will eventually get rid of all positive action status atoms. Thus, in this section, we will associate with any agent program $\mathcal{P}$ an operator $Unfold_{\mathcal{P}}$, which is used for this purpose. Before doing so, we need some additional syntax. Let us call any positive action status atom $Op\alpha$ occurring in the body of a rule r a *prerequisite* of r.

Definition 6.1 (prerequisite-free (pf) constraint) *A prerequisite-free (pf) constraint (*pfc*) is defined as follows:*

- *"*`true`*" and "*`false`*" are distinguished* pfc*s (with obvious meaning).*
- *the body of each rule r such that $B^{+}_{as}(r) = \emptyset$ (i.e., r contains no prerequisites) is a* pfc*;*
- *If γ_1, γ_2 are* pfc*'s, then so are $\gamma_1 \,\&\, \gamma_2$ and $\gamma_1 \vee \gamma_2$.*

Definition 6.2 (prerequisite-free (pf) constraint rule) *A prerequisite-free constraint rule is of the form*

$$A \leftarrow \mathit{pfc}$$

where A is an action status atom and pfc *is a pf-constraint.*

An agent program $\mathcal{P}$ may certainly contain rules r which have prerequisites $Op\alpha$. Each such prerequisite might be replaced by the body of a rule r' which derives $Op\alpha$. This way, the prerequisites can be eliminated from r, replacing them by rule bodies. This step may introduce new prerequisites from the body of some rule r', though. Such prerequisites may be eliminated by repeating the process.

The operator $\mathit{Unfold}_{\mathcal{P}}$ is used for describing this process. Informally, it maps a set R of pf-constraint rules, which compactly represent already unfolded rules from $\mathcal{P}$, to another set $\mathit{Unfold}_{\mathcal{P}}(R)$ of pf-constraint rules, implementing the unfolding step described above, but using pf-constraint rules from R rather than rules r' from $\mathcal{P}$. The collect set $\mathit{Coll}_{\mathcal{P}}$ introduced next is an intermediate operator for defining $\mathit{Unfold}_{\mathcal{P}}$.

Definition 6.3 (collect set $\mathit{Coll}_{\mathcal{P}}$) *Let $\mathcal{P}$ be an agent program and R be a set of pf-constraint rules which are standardized apart. Suppose $Op \in \{\mathbf{P}, \mathbf{O}, \mathbf{Do}, \mathbf{W}, \mathbf{F}\}$ and let α be any action name. Then the* collect set, *$\mathit{Coll}_{\mathcal{P}}(Op, \alpha, R)$, is defined as the following set of pf-constraint rules:*

$$\begin{aligned}\mathit{Coll}_{\mathcal{P}}(Op,\alpha,R) = \Big\{\gamma \mid{}& Op\alpha(\vec{X}) \leftarrow \gamma_0 \in R,\\ & \textit{there exists a rule } r \in \mathcal{P} \textit{ such that } head(r) = Op'\alpha(\vec{t}), Op' \leq Op,\\ & B^+_{as}(r) = \{Op_1\alpha_1(\vec{t_1}),\ldots,Op_k\alpha_k(\vec{t_k})\},\\ & Op_i\alpha_i(\vec{X_i}) \leftarrow \gamma_i \in R, i = 1,\ldots,k,\\ & \textit{and } \gamma = \gamma_0 \vee \Big[(\vec{X} = \vec{t})\&B_{cc}(r)\& \textstyle\bigwedge_{i=1}^{k}\Big((\vec{X_i} = \vec{t_i})\&\gamma_i\Big)\&B^-_{as}(r)\Big]\Big\}\end{aligned}$$

Here, an equality formula $(\vec{X} = \vec{t})$ stands for the conjunction of all equality atoms $X = t$, where X and t are from the same position of $\vec{X}$ and $\vec{t}$, respectively.

What this operator does is the following. It takes a pf-constraint rule from R which defines $Op\alpha(\vec{X})$ through its body γ_0, and weakens this constraint γ_0 (i.e., increases the set of solutions) by taking the disjunction with an unfolded rule r whose head either defines an instance of the action status atom $Op\alpha(\vec{X})$, or of an action status atom $Op'\alpha(\vec{X})$ which, by deontic and action closure rules, defines an instance of $Op\alpha(\vec{X})$. The unfolding of rule r is obtained by replacing each positive action status atom $Op_i\alpha_i(\vec{t_i})$ in r's body with the body γ_i of a pf-constraint rule pfc_i from R which defines $Op_i\alpha(\vec{X_i})$.

Informally, one may think of the rules in R as being known for sure. For instance, if $Op\alpha(\vec{X}) \leftarrow \gamma_0$ is in R, then one may think of this as saying that all instances θ of $\alpha(\vec{X})$ such that the existential closure of $\gamma_0\theta$ is true in the current agent state are true. The $\mathit{Coll}_{\mathcal{P}}$ operator takes such an R as input, and uses the rules in $\mathcal{P}$ to identify ways of weakening the constraint γ_0, thus extending the set of instances θ of $\alpha(\vec{X})$ satisfying the above condition.

Note. We will assume when no pf-constraint rule in R has $Op\alpha(\vec{X})$ in the head, then that R is augmented via the insertion of the pf-constraint rule $Op\alpha(\vec{X}) \leftarrow \texttt{false}$. The rest of our treatment is based on this assumption.

We remark that in the above definition, the constraint γ may be simplified by obvious operations such as pushing through equalities, or eliminating true/false subparts of a constraint; we do not pursue the issue of simplifications further here.

Example 6.1 Consider the example program $\mathcal{P}_1$ shown in Example 3.7. Suppose R is empty. Then:

$$\begin{aligned}\mathit{Coll}_{\mathcal{P}_1}(\mathbf{P},\alpha(X,Y),R) &= \{\ \mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,b))\,\&\,\mathbf{in}(Y,\mathcal{S}{:}\mathtt{g}(X.\mathit{field},b,c))\},\\ \mathit{Coll}_{\mathcal{P}_1}(\mathbf{P},\beta(X,Y),R) &= \{\ Y = a\,\&\,\mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,a)),\\ &\qquad Y = a\,\&\,\mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,b))\,\&\,\neg\mathbf{Do}\gamma(X)\},\\ \mathit{Coll}_{\mathcal{P}_1}(\mathbf{P},\gamma(Y),R) &= \{\ \mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,b))\}.\end{aligned}$$

The operator $\mathit{Unfold}_{\mathcal{P}}$ defined below uses $\mathit{Coll}_{\mathcal{P}}$ to compute a single constraint for each Op and each action name α.

Definition 6.4 (operator $\mathit{Unfold}_{\mathcal{P}}$)

$$\begin{array}{rcl} \mathit{Unfold}_{\mathcal{P}}(Op,\alpha,R) & = & Op\alpha(\vec{X}) \leftarrow \bigvee_{\gamma \in \mathit{Coll}_{\mathcal{P}}(Op,\alpha,R)} \gamma, \qquad \textit{and} \\ \mathit{Unfold}_{\mathcal{P}}(R) & = & \bigcup_{Op,\alpha} \mathit{Unfold}_{\mathcal{P}}(Op,\alpha,R). \end{array}$$

When $\mathit{Coll}_{\mathcal{P}}$ is empty in the above definition, the right hand side of the above implication is set to `false`.

The operator $\mathit{Unfold}_{\mathcal{P},\alpha}$ may be iterated; as usual, its powers are denoted by

$$\begin{array}{rcl} \mathit{Unfold}^0_{\mathcal{P}}(,Op,\alpha,R) & = & R, \\ \mathit{Unfold}^{i+1}_{\mathcal{P}}(Op,\alpha,R) & = & \mathit{Unfold}_{\mathcal{P}}(Op,\alpha,\mathit{Unfold}^i_{\mathcal{P}}(Op,\alpha,R)), \quad i \geq 0, \end{array}$$

and similar with $\mathit{Unfold}_{\mathcal{P}}(R)$.

Example 6.2 The following pf-constraint rules are contained in $\mathit{Unfold}^1_{\mathcal{P}_1}(\emptyset)$:

$$\begin{array}{rcl} \mathbf{P}\alpha(X,Y) & \leftarrow & \mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,b))\,\&\,\mathbf{in}(Y,\mathcal{S}{:}\mathtt{g}(X.\mathit{field},b,c)), \\ \mathbf{F}\gamma(Z) & \leftarrow & \mathbf{in}(Z,\mathcal{S}{:}\mathtt{f}(a,b)), \\ \mathbf{P}\beta(X,Y) & \leftarrow & Y = a\,\&\,\mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,a)) \,\vee \\ & & Y = a\,\&\,\mathbf{in}(X,\mathcal{S}{:}\mathtt{f}(a,b))\,\&\,\neg\mathbf{Do}\gamma(X). \end{array}$$

Note that the above pf-constraints have been simplified so as to make them more readable.

When we iteratively compute $\mathit{Unfold}^i_{\mathcal{P}}$, it is important to note that we may often *redundantly* fire the same rule in $\mathit{Coll}_{\mathcal{P}}$ many times without deriving anything new. Constraint equivalence tests (to be defined shortly below) may be used to terminate this.

This raises the question of what it means for two pf-constraints to be equivalent. We provide a simple model-theoretic answer to this question below, and then explain what a constraint equivalence test is.

Definition 6.5 (bistructure) *A* bistructure *for an agent program $\mathcal{P}$ is a pair $(\mathcal{O}_{\mathcal{S}}, S)$ where $\mathcal{O}_{\mathcal{S}}$ is a possible state of the agent in question, and S is a status set.*

We now define what it means for a bistructure to satisfy a pf-constraint.

Definition 6.6 (satisfaction of a ground pf-constraint by a bistructure) *A bistructure $(\mathcal{O}_{\mathcal{S}}, S)$* satisfies

1. *a ground code call condition χ iff χ is true in $\mathcal{O}_{\mathcal{S}}$;*
2. *a ground action status atom $\neg Op\alpha$ iff $Op\alpha \notin S$;*
3. *a conjunction $\mathit{pfc}_1 \,\&\, \mathit{pfc}_2$ iff it satisfies pfc_1 and pfc_2;*
4. *a disjunction $\mathit{pfc}_1 \vee \mathit{pfc}_2$ iff it satisfies either pfc_1 or pfc_2.*

Definition 6.7 (solutions of a pr-constraint on a bistructure) *Suppose* pfc *is a pf-constraint involving free variables* $\vec{X}$*. The* solutions *of* pfc *with respect to a bistructure* $(\mathcal{O}_S, S)$ *is the set of all ground substitutions* θ *such that* $(\mathcal{O}_S, S)$ *satisfies* $\mathsf{pfc}\theta$*.*

We are now ready to define what it means for two constraints to be equivalent in the presence of an arbitrary but fixed underlying agent program $\mathcal{P}$.

Definition 6.8 (a-equivalent pf-constraints) *Suppose* $\mathbf{a}$ *is an agent, and* $\mathsf{pfc}_1, \mathsf{pfc}_2$ *are pf-constraints involving variables* $\vec{X}, \vec{Y}$ *respectively. Let* $\vec{X}', \vec{Y}'$ *be subvectors of* $\vec{X}, \vec{Y}$ *respectively of the same length. Then* $\mathsf{pfc}_1, \mathsf{pfc}_2$ *are said to be* $(\mathbf{a}, \vec{X}', \vec{Y}')$-equivalent*, denoted* $\mathsf{pfc}_1 \sim_{\mathbf{a},\vec{X}',\vec{Y}'} \mathsf{pfc}_2$ *iff for every bistructure* $(\mathcal{O}_S, S)$ *such that* S *is a reasonable status set of* $\mathbf{a}$*'s agent program with respect to state* $\mathcal{O}_S$*, it is the case that* $\pi_{\vec{X}'}(\mathsf{Sol}(\mathsf{pfc}_1)) = \pi_{\vec{Y}'}(\mathsf{Sol}(\mathsf{pfc}_2))$*, where* $\mathsf{Sol}(\mathsf{pfc}_i)$ *denotes the set of all solutions of* pfc_i *and* $\pi_{\vec{Z}}(\mathsf{Sol}(\mathsf{pfc}_i))$ *denotes the set of projections of solutions of* pfc_i *on the variables in* $\vec{Z}$*.*

The intuition behind the above definition is that two pf-constraints may appear in the body of two different pf-constraint rules. Each of these rules may "output" some variables in the body to the head. The condition involving the check that $\pi_{\vec{X}'}(\mathsf{Sol}(\mathsf{pfc}_1)) = \pi_{\vec{Y}'}(\mathsf{Sol}(\mathsf{pfc}_2))$ above ensures that the outputs of the constraints involved are identical, when we restrict them to the variables specified.

In general, the problem of checking $\mathbf{a}$-equivalence of two pf-constraints is easily seen to be undecidable, and as a consequence, we introduce the notion of a pf-constraint equivalence test below which provides a sufficient condition for two pf-constraints to be equivalent.

Definition 6.9 (pf-constraint equivalence test) *A* constraint equivalence check test $\mathsf{eqi}_{\mathbf{a},\vec{X}',\vec{Y}'}$ *is a function that takes as input two pf-constraints* $\mathsf{pfc}_1, \mathsf{pfc}_2$*, such that if* $\mathsf{eqi}_{\mathbf{a},\vec{X}',\vec{Y}'}(\mathsf{pfc}_1, \mathsf{pfc}_2) = \mathtt{true}$ *then* $\mathsf{pfc}_1, \mathsf{pfc}_2$ *are equivalent with respect to* $\mathcal{P}$*.*

We will often write eqi instead of $\mathsf{eqi}_{\mathbf{a},\vec{X}',\vec{Y}'}$ when the parameters $\mathbf{a}, \vec{X}', \vec{Y}'$ are clear from the context.

Note that just as in the case of a conflict freedom test, a pf-constraint equivalence test merely implements a *sufficient* condition to guarantee equivalence of two pf-constraint rules. It may well be the case that $\mathsf{pfc}_1, \mathsf{pfc}_2$ are in fact equivalent on all agent states, but $\mathsf{eqi}(\mathsf{pfc}_1, \mathsf{pfc}_2) = \mathtt{false}$.

Some examples of constraint equivalence tests are given below.

Example 6.3 (renaming permutation equivalence) The function eqi^{rp} returns true on two pf-constraints $\mathsf{pfc}_1, \mathsf{pfc}_2$ whose variables are standardized apart iff there is a renaming substitution θ such that $\{C\theta \mid C \in \mathsf{pfc}_1^\star\} = \{C'\theta \mid C' \in \mathsf{pfc}_2^\star\}$ where $\mathsf{pfc}_i^\star$ is a conjunctive normal form representation of pfc_i.

Example 6.4 (rewrite-based equivalence) Another way to check equivalence of two pf-constraints is to expect the agent developer to write a set, RW, of rewrite rules of the form

$$\chi \quad \rightarrow \quad \mathsf{pfc}_1 = \mathsf{pfc}_2$$

where χ is a code call condition not involving the $\mathbf{in}(,)$ predicate, i.e., it only involves comparison operations $=, <, \leq, >, \geq$. RW encodes domain knowledge about what equivalences hold in the data structures and actions involved. It may be viewed as an equational theory [29]. Let $\Upsilon^k(\mathsf{pfc})$ denote the set of all pf-constraints that pfc can be rewritten to by applying at most k rules in RW.

We say that pfc_1 and pfc_2 are k-equivalent with respect to RW iff $\Upsilon^k(\mathsf{pfc}_1) \cap \Upsilon^k(\mathsf{pfc}_2) \neq \emptyset$.

It is easy to see that as long as each rule in the equational theory RW is sound (i.e., it accurate with respect to the data structures and actions in question), this is a valid pf-constraint equivalence test.

Based on the notion of equivalent pf-constraints, we may define a notion of equivalence for sets of pf-constraint rules as follows.

Definition 6.10 (equivalence of two sets of pf-constraint rules) *Two sets R_1, R_2 of pf-constraint rules are equivalent with respect to a pf-constraint equivalence test* eqi*, denoted $R_1 \equiv_{\mathsf{eqi}} R_2$, iff there is a bijection $\psi : R_1 \to R_2$ such that for all $r_1 \in R_1$,* $\mathsf{eqi}(r_1, \psi(r_1)) = \mathtt{true}$ *and r_1, r_2 both have heads of the form $Op\alpha(\cdots)$, i.e., their heads involve the same action name and the same deontic modality.*

We now define the notion of a bounded agent program. Informally, a program $\mathcal{P}$ is bounded, if after unfolding rules in $\mathcal{P}$ a certain number of times, we end up with a set of pf-constraints which does not change semantically if we do further unfolding steps.

Definition 6.11 (b-bounded agent program) *An* agent program $\mathcal{P}$ *is* bounded *with respect to an equivalence check test* eqi *iff there is an integer b such that* $\mathsf{eqi}(\mathit{Unfold}^b_{\mathcal{P}}(R), \mathit{Unfold}^{b+1}_{\mathcal{P}}(R)) = \mathtt{true}$*, for any set of pf-constraints R. In this case, $\mathcal{P}$ is (eqi, b)-bounded.*

Observe that when $\mathcal{P}$ is a program which does not contain a truly recursive collection of rules (i.e., the graph G built in Algorithm **Check_WRAP** does not contain a cycle), then $\mathcal{P}$ is (eqi, b)-bounded where eqi is an arbitrary pf-constraint equivalence test such that $\mathsf{eqi}(\mathsf{pfc}, \mathsf{pfc}) = \mathtt{true}$ for every pfc and b is the number of rules in $\mathcal{P}$. Thus, only truly recursive rules—which seem to play a minor rule in many agent programs in practical applications—may prevent boundedness. If, moreover, $\mathcal{P}$ is deontically stratified and has a layering $\mathcal{P}_0, \ldots, \mathcal{P}_k$ then $\mathcal{P}$ is even $(\mathsf{eqi}, k+1)$-bounded.

Rather than unfolding a *WRAP* $\mathcal{P}$ in bulk, we can unfold it along a layering $\ell \in \mathsf{wtn}(\mathcal{P})$ using a pf-constraint equivalence test $\mathsf{eqi}^{(i)}$ which is suitable for each layer $\mathcal{P}_i$. Such an $\mathsf{eqi}^{(i)}$ may be selected automatically by the implementation, or the agent designer may be prompted to select one from a catalog or provide her own equivalence test implementation. In particular, if $\mathcal{P}_i$ contains no set of truly recursive rules, then a test $\mathsf{eqi}^{(i)}$ which always returns true is suitable, which can be automatically selected.

Let us define sets of pf-constraint rules R^ℓ_i, $i \geq 0$, as follows:

$$\begin{aligned} R^\ell_0 &= \emptyset, \\ R^\ell_{i+1} &= \mathit{Unfold}^b_{\mathcal{P}_i}(R^\ell_i), \text{ for all } i \geq 0, \end{aligned}$$

where $\mathcal{P}_i$ (the i'th layer of $\mathcal{P}$) is $(b, \mathsf{eqi}^{(i)})$-bounded. The *unfolding of $\mathcal{P}$ along ℓ* is given by the set R^ℓ_{k+1} where k is the highest nonempty layer of $\mathcal{P}$.

Definition 6.12 (b-regular agent program) *Suppose a layering ℓ and equivalence tests* $\mathsf{eqi}^{(i)}$ *($i \geq 0$) have been fixed for an agent program $\mathcal{P}$. Then, $\mathcal{P}$ is said to be a b-*regular agent program *with respect to ℓ and the* $\mathsf{eqi}^{(i)}$*, if and only if $\mathcal{P}$ is a WRAP, $\ell \in \mathsf{wtn}(\mathcal{P})$, and each layer $\mathcal{P}_i$ of $\mathcal{P}$ is $(\mathsf{eqi}^{(i)}, b)$-bounded.*

Definition 6.13 (regular agent) *An agent is said to be* regular *with respect to a layering ℓ and a selection of pf-constraint equivalence tests* $\mathsf{eqi}^{(i)}$*, if it is weakly regular and its associated agent program is b-regular with respect to ℓ and the* $\mathsf{eqi}^{(i)}$*, for some $b \geq 0$.*

In the above definition, an agent's regularity depends on several parameters ℓ, $\mathsf{eqi}^{(i)}$, and b. The implementation may use them as follows: it generates a layering of a $\mathcal{P}$, and equivalence tests $\mathsf{eqi}^{(i)}$ are fixed for each layer $\mathcal{P}_i$ with the help of the agent developer. It then sets b to a default value, and iteratively constructs the sequence $R^\ell_0, R^\ell_1, \ldots, R^\ell_{k+1}$; if in some step, the equivalence test $\mathsf{eqi}^{(i)}(\mathit{Unfold}^b_{\mathcal{P}}(R), \mathit{Unfold}^{b+1}_{\mathcal{P}}(R))$

returns false, then an error is flagged at compile time. The b parameter can be reset by the agent developer. However, for most agents, a sufficiently large b (e.g., $b = 500$) may be adequate.

In case an agent is regular, Step 1 of the algorithm **Reasonable_SS** is guaranteed to terminate for any agent state $\mathcal{O}_{\mathcal{S}}$ in finite time. The reason is that computing the fixpoint of the operator $\mathcal{T}_{\mathcal{P}_i} \uparrow \omega$ at each level is possible in a constant number of steps, each of which may only involve finitely many objects, and the number of levels is bounded by a constant. Step 1 of **Reasonable_SS** can be realized by computing solutions to the pf-constraint rules from the set R^{ℓ}_{k+1}; how this can be implemented is outlined in the next section.

The following theorem states the result that for a regular agent, its reasonable status set on an agent state is effectively computable.

Theorem 6.1 *If* $\mathbf{a}$ *is a regular agent, then algorithm* **Reasonable_SS** *computes a reasonable status set (if one exists) in finite time.*

This leaves us with the question *in what time* the reasonable status set can be computed. We cannot be sure, a priori, that this is possible in polynomial time, as strong safety of rules just ensures a finite but arbitrarily large of solutions to a code call; likewise, comparison atoms $X < \mathtt{t}$, where $\mathtt{t}$ is e.g. an integer, may instantiate to an exponential number of solutions (measured in the number of bits needed to store $\mathtt{t}$). Thus, we need some further assertions to guarantee evaluation in polynomial time.

For convenience, call an occurrence of a variable X in a strongly safe code call condition χ *loose* with respect to a set $\vec{X}$ of variables, if X is not from $\vec{X}$ and does not occur as the result of a code call $\mathbf{in}(X, \cdots)$ or $\mathbf{not\_in}(X, \cdots)$ in χ. Intuitively, a loose occurrence of a variable X may be instantiated without accessing the agent state, with some value drawn from X's domain. Based on this, loose occurrence of a variable X in a strongly safe action, rule, integrity and action constraint is defined in the obvious way.

Theorem 6.2 *Suppose* $\mathbf{a}$ *is a fixed regular agent. Assume that the following holds: (1) Every ground code call* $\mathcal{S}{:}\mathtt{f}(d_1, \ldots, d_n)$*, has a polynomial set of solutions, which is computed in polynomial time; and (2) no occurrence of a variable in* $\mathbf{a}$*'s description is loose. Furthermore, assume that assembling and executing* $\mathbf{conc}(\mathbf{Do}(S), \mathcal{O}_{\mathcal{S}})$ *is possible in polynomial time in the size of* $\mathbf{Do}(S)$ *and* $\mathcal{O}_{\mathcal{S}}$*. Then, algorithm* **Reasonable_SS** *computes a reasonable status set (if one exists) on a given agent state* $\mathcal{O}_{\mathcal{S}}$ *in polynomial time (in the size of* $\mathcal{O}_{\mathcal{S}}$*).*

Forbidding loose occurrences of a variable X in an atom such as $X < \mathtt{t}$ is not overly restrictive in general; using a special domain function $\mathbf{types}{:}\mathtt{dom}(\tau)$, which returns the elements of type τ, we can eliminate the loose occurrence by joining the code call atom $\mathbf{in}(X, \mathbf{types}{:}\mathtt{dom}(\tau))$, where τ is the domain of X. Or, we might use a special domain comparison function $\mathbf{types}{:}\mathtt{less\_than}(\tau, X)$, which returns all values of τ which are less than X, and replace $X < \mathtt{t}$ by a code call atom $\mathbf{in}(X, \mathbf{types}{:}\mathtt{less\_than}(\tau, \mathtt{t}))$. The latter has a guaranteed finite set of solutions, which is not true for the code call atom $\mathbf{in}(X, \mathbf{types}{:}\mathtt{dom}(\tau))$ if τ is infinite.

7 Query Maintenance Package

In this section, we describe the *RAP* implementation strategy via a specialized package called the *Query Maintenance Package*. The basic idea behind this package is simple. Agents need to *continuously* recompute their reasonable status sets, based on the latest set of state changes that have occurred (which in turn are triggered by messages received). We would like to reduce this run-time computation load on the agent as much as possible. We do this by ensuring that when the agent is deployed, a certain data structure called the $\mathbf{QMPtab}$ defined in Section 7.1 below, is initialized. The $\mathbf{QMPtab}$ contains a succinct, non ground description of the agent's reasonable status set at *any given point in time*, i.e. it is state independent. With every operator $Op \in \{\mathbf{O}, \mathbf{P}, \mathbf{Do}, \mathbf{F}, \mathbf{W}\}$, and every action $\alpha(\vec{X})$ with all nonground parameters, it associates a single query, which when evaluated against the current agent state specifies which instances of $Op\alpha(\vec{X})$ are true in the current reasonable status set of the agent. Then, in Section 7.2, we describe a set of functions that may be used, both at run-time and compile time, to perform computations based on the $\mathbf{QMPtab}$. These operators perform the basic computations needed by any agent.

7.1 The QMPtab data structure

The $\mathbf{QMPtab}$ is a table having the schema $(\mathtt{Op}, \mathtt{Action}, \mathtt{PFC})$ where:

1. Op is one of the five operators $\mathbf{F}, \mathbf{P}, \mathbf{O}, \mathbf{Do}, \mathbf{W}$;
2. Action is of the form $\alpha(X_1, \ldots, X_n)$ where α is an action name having schema $(\tau_1, \ldots, \tau_n)$ and each $X_i \in \mathit{Var}(\tau_i)$ (i.e., each X_i is a variable over objects of type τ_i);
3. PFC is a pf-constraint.

For each $Op \in \{\mathbf{F}, \mathbf{P}, \mathbf{O}, \mathbf{Do}, \mathbf{W}\}$ and each action name α, $\mathbf{QMPtab}$ contains *exactly one* row having $\mathtt{Op} = Op$ and $\mathtt{Action} = \alpha(\ldots)$.

Example 7.1 A small example $\mathbf{QMPtab}$ is shown in Table 3. Intuitively, the first row of this table says that to determine the set of all action status atoms of the form $\mathbf{O}\alpha(t)$ that are true in this agent's unique reasonable status set, all we need to do is to evaluate the code call condition $\mathbf{in}(X, \mathbf{d}{:}\mathtt{f}(a, b))\ \&\ X < 5$ with respect to the current agent state. The other rows in this table may be similarly interpreted. Note that the $\mathbf{QMPtab}$ therefore does not depend on the state, though the evaluation of the entries in the PFC column certainly does.

Op	Action	PFC
$\mathbf{O}$	$\alpha(X)$	$\mathbf{in}(X, \mathbf{d}{:}\mathtt{f}(a, b))\ \&\ X < 5$
$\mathbf{Do}$	$\alpha(X)$	$\mathbf{in}(X, \mathbf{d}{:}\mathtt{f}(a, b))\ \&\ X < 2$
$\mathbf{F}$	$\alpha(X)$	$\mathbf{in}(X, \mathbf{d}{:}\mathtt{f}(a, b))\ \&\ X > 10$
$\mathbf{P}$	$\alpha(X)$	$\mathbf{in}(X, \mathbf{d}{:}\mathtt{f}(a, b))\ \&\ X < 8$
$\mathbf{W}$	$\alpha(X)$	false

Table 3: Example $\mathbf{QMPtab}$ for action $\alpha(X)$

In general, given any $\mathbf{QMPtab}$, we may associate with it a unique status set $\mathbf{ss}(\mathbf{QMPtab})$ as follows:

$$\mathbf{ss}(\mathbf{QMPtab}) = \{\, Op\alpha(\vec{X}\theta) \mid \text{there is a row } r \in \mathbf{QMPtab} \text{ such that } r.\mathtt{Op} = Op, \\ r.\mathtt{Action} = \alpha(\vec{X}),\ \alpha(\vec{X}\theta) \text{ is ground and } r.\mathtt{PFC}\theta \text{ is true}\}.$$

Here, truth of the pf-constraint $r.\mathtt{PFC}$ is with respect to the current agent state. As we can see, $\mathbf{QMP}$ may be used to succinctly represent status sets. What we would *like* is to ensure that at any given point in time $\mathbf{ss}(\mathbf{QMPtab})$ is in fact the unique reasonable status set of the underlying *RAP* with respect to the agent's state at that point in time.

The basic idea is to first compute such a $\mathbf{QMPtab}$ when the agent is deployed. As changes occur, we would like to use the $\mathbf{QMPtab}$ to rapidly compute the set of all $\mathbf{Do}$-actions that the agent must perform, given the occurrence of the changes. This will be done via the $\mathbf{qmp}{:}\mathtt{update}$ function described in Section 7.2.

7.2 QMP function calls

The $\mathbf{QMP}$ package supports a set of function calls. The input/output signatures of these function calls, together with the intended use of these function calls is listed here—succeeding sections discuss their implementation.

- $\mathbf{qmp}{:}\mathtt{init}(\texttt{action base } \mathcal{AB}; \texttt{software code } \mathcal{S})$

  This function initializes the $\mathbf{QMPtab}$ so that $\mathbf{ss}(\mathbf{QMPtab})$ coincides with the unique reasonable status set of the agent program. This is done by first directly computing $\mathit{Unfold}^b_{\mathcal{P}}$. If $Op\alpha \leftarrow \chi$ is in $\mathit{Unfold}^b_{\mathcal{P}}$, then the triple (Op, α, χ) is placed in $\mathbf{QMPtab}$. The implementation of $\mathbf{qmp}{:}\mathtt{init}$ is described in Section 7.2.1.

- $\mathbf{qmp}{:}\mathtt{eval}(\texttt{action status atom } Op\alpha(\vec{t}); \texttt{agent state } \mathcal{O}_{\mathcal{S}}; \texttt{sw code } \mathcal{S})$

  Given an action status atom $Op\alpha(\vec{t})$, we may associate with it a pfc-constraint γ by finding the pfc-constraint γ' associated with $Op\alpha(\vec{X})$ in $\mathbf{QMPtab}$ and setting $\gamma = \gamma' \,\&\, \vec{X} = \vec{t}$. We then need to evaluate γ over the current agent state. The $\mathbf{qmp}{:}\mathtt{eval}$ function does this, building on top of the existing implementation of the HERMES Heterogeneous Reasoning and Mediator System developed at the University of Maryland [2, 12, 26]. The implementation of $\mathbf{qmp}{:}\mathtt{eval}$ is described in Section 7.2.2.

- $\mathbf{qmp}{:}\mathtt{update}(\texttt{software code } \mathcal{S})$

  At any given point t in time, the agent has a reasonable status set. When the state of the agent changes through the receipt of one or more messages, the agent must determine what new actions must be performed, i.e. it must compute the set of all actions of the form $\mathbf{Do}\alpha(\vec{t})$ that are true in the new reasonable status set of the agent (with respect to the state of the agent after receipt of the messages). The $\mathbf{qmp}{:}\mathtt{update}$ function tries to avoid a brute-force computation of this set. Section 7.2.3 describes how this function is implemented.

7.2.1 function $\mathbf{qmp}{:}\mathtt{init}$

Note that for any Op and any action $\alpha(\vec{X})$, $\mathit{Unfold}^i_{\mathcal{P}}$ contains exactly one rule of the form

$$Op\alpha(\vec{X}) \quad \leftarrow \quad \mathsf{pfc}_i.$$

This pf-constraint can be represented in a straightforward way as an AND/OR tree. The leaves of the tree are either code call conditions, or negated action status atoms, while the interior nodes are labeled with either the connective $\vee$ or $\&$. For example, consider the following pf-constraint rule drawn from Example 6.2:

$$\begin{aligned}\mathbf{P}\beta(X,Y) \quad \leftarrow \quad & Y = a \,\&\, \mathbf{in}(X, \mathcal{S}{:}\mathtt{f}(a,a)) \;\vee \\ & Y = a \,\&\, \mathbf{in}(X, \mathcal{S}{:}\mathtt{f}(a,b)) \,\&\, \neg\mathbf{Do}\gamma(X).\end{aligned}$$

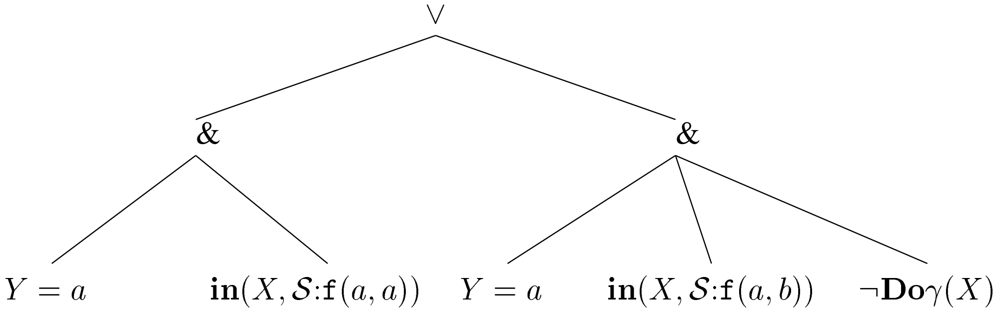


Figure 3: An example AND/OR tree associated with a PFC

Figure 3 shows the AND/OR tree associated with the body of this pf-constraint rule. Note that this AND/OR tree is nothing more than a parse tree associated with a pf-constraint. When unfolding is performed, and pfc_i is updated to a new AND/OR parse tree pfc_{i+1}, the AND/OR parse tree associated with pfc_i can be straightforwardly extended to one associated with pfc_{i+1}.

7.2.2 function $\mathbf{qmp}{:}\mathtt{eval}$

Consider any pair $(Op, \alpha(\vec{X}))$. After unfolding the *RAP* out completely using the $\mathbf{qmp}{:}\mathtt{init}$ function, this pair has an associated pf-constraint in $\mathbf{QMPtab}$ which is represented as an AND/OR tree. The $\mathbf{qmp}{:}\mathtt{eval}$ function evaluates these pf-constraints bottom up (i.e., starting from leaf nodes). It associates a Sol field with each non-leaf node in the AND/OR tree. This field is intended to find the set of all solutions of the pf-constraint rooted at that node, and is initially set to be empty. The following algorithm specifies how to associate a Sol field with a non-leaf node and compute the final answer.

1. **Associating Sol fields with interior nodes.**

(a) *The node is an "&" node.* In this case, the children of the node fall into three categories. Either they are code call atoms, or they are negated action status atoms, or they are the root of a tree representing a sub-pf-constraint. In this case, we may proceed as follows. Recursively evaluate the conjunction of the code call part and the sub-pf-constraint part—these must be evaluable as they are each strongly safe. Let Θ be the set of all solutions of this evaluation. Now observe that every variable occurring in a negative action status literal part (e.g. $\neg Op'\beta(\vec{Y})$) must occur in the evaluated part, and hence, each solution $\theta \in \Theta$ instantiates each such variable. For each $\theta \in \Theta$ check if $\neg Op'\beta(\vec{Y}\theta)$ is true for *each* negated literal that is a child of the "&" node referenced above. Set Sol to $\{\theta \mid \theta \in \Theta$ and $\neg Op'\beta(\vec{Y}\theta)$ is true for all negated action status atoms that are children of $\Theta\}$.

(b) *The node is an "∨" node.* Recursively evaluate all its children $C_1, \ldots, C_n$. If they have $Sol_1, \ldots, Sol_n$ as their solutions, respectively, let $Sol = \bigcup_{i=1}^{n} Sol_i$.

2. **Returning the Answer.** Return the set $\{\pi_{\vec{X}}(\theta) \mid \theta \in Sol(RNode)\}$ where $RNode$ is the root of the tree associated with $(Op, \alpha(\vec{X}))$.

Notice that for the above procedure to work, for each ground negated action status atom $\neg Op'\beta(\vec{Y}\theta)$, we must already know the set of solutions of $Op'\beta(\vec{Y})$. This means that we must always invoke the $\mathbf{qmp}{:}\mathtt{eval}$

function bottom up starting with action status atoms defined in the bottom most layer of the *RAP*, and work our way up.

Let us illustrate this procedure on the example AND/OR tree shown in Figure 3. We assume of course, that $\neg\mathbf{Do}\gamma(X)$ has been evaluated prior to the evaluation of this parse tree. The evaluation of the left "&" node is straightforward. Let us suppose this returns two substitutions, $\theta_1 = \{X/c, Y/a\}$ and $\theta_2 = \{X/d, Y/a\}$. Now evaluate the right "&" expression. This is done by first executing the code call condition $Y = a \,\&\, \mathbf{in}(X, \mathbf{scode}{:}\mathtt{f}(a, b))$. Let us assume this returns two substitutions, $\sigma_1 = \{X/a, Y/a\}$ and $\sigma_2 = \{X/b, Y/a\}$. For each of these two substitutions, we invoke $\mathbf{Do}\gamma(X)$. Suppose $\mathbf{Do}\gamma(a)$ fails, but $\mathbf{Do}\gamma(b)$ succeeds. Then *Sol* of the right "&" node consists just of σ_1. *Sol* of the root is now $\{\sigma_1, \theta_1, \theta_2\}$. The algorithm restricts them to the variables (X, Y), so it just returns $\{\sigma_1, \theta_1, \theta_2\}$.

7.2.3 function $\mathbf{qmp}{:}\mathtt{update}$

All changes to the state of the agent occur in one of two ways—(i) either the agent computes a reasonable status set which causes it to perform some state-changing actions, or (ii) the agent receives some messages from other agents that cause it to compute a new reasonable status set and take the actions prescribed in that new reasonable status set.

The first case above is clearly one that is triggered by the agent itself after computing a valid reasonable status set, so no new reasonable status set needs to be computed.

One the other hand, case (ii) above forces the agent to find a new reasonable status set based on the messages received. When an agent is regular, it can have at most one reasonable status set, viz. $\mathbf{ss}(\mathbf{QMPtab})$. If $\mathbf{ss}(\mathbf{QMPtab})$, when evaluated agains the state of the agent *after* the messages are received, satisfies the integrity and action constraints, then we know that it is reasonable.

In order to determine what actions to execute concurrently, the agent only needs to perform the steps listed below. In what follows, when we refer to "evaluation" of a status atom, $Op\alpha(\vec{X})$, we mean finding the set of all ground instances of that status atom such that the pf-constraint associated with $Op\alpha(\vec{X})$ is true.

1. Create a graph whose vertices are action status atoms as follows:

   (a) For each row r in $\mathbf{QMPtab}$ such that $r.\mathtt{Op} = \mathbf{Do}$ and $r.\mathtt{Action} = \alpha(\vec{X})$ create the node $\mathbf{Do}\alpha(\vec{X})$.

   (b) If $Op\beta(\vec{X})$ is a node in the graph, corresponding to the row r in $\mathbf{QMPtab}$ with r.Opr $= Op$ and r.Action $= \beta(\vec{X})$, and a negated action status atom $\neg Op'\beta'(\vec{t})$ appears in r.PFC, then $Op'\beta'(\vec{Y})$ is a node in the graph, where $Op'\beta'(\vec{Y})$ is in $\mathbf{QMPtab}$, and there is an edge in the graph from $Op'\beta'(\vec{Y})$ to $Op\beta(\vec{X})$.

   As $\mathcal{P}$ is a *RAP*, it is easy to see that this graph is acyclic. Given an action status atom A in this graph, $pred(A)$ is the set of all action status atoms A' such that there is a direct edge from A' to A in the graph. Initially, all nodes are unmarked.

2. **while** there is an unmarked node left **do**

   (a) Select an action status atom $Op\beta(\vec{X})$ such that all $B \in pred(Op\beta(\vec{X}))$ are marked.

   (b) Evaluate $Op\beta(\vec{X})$. The code call part of the associated pfc can be evaluated using any incremental view maintenance algorithm in the literature. (By selecting only action status atoms using the previous condition, we know that the negative action status atoms associated with $Op\beta(\vec{X})$'s associated pf-constraint have been fully evaluated).

(c) Mark the node labeled $Op\beta(\vec{X})$.

3. Let

$$DoSet = \bigcup_{DoNodes} \{\mathbf{Do}(\alpha(\vec{X}\sigma)) \mid \mathbf{Do}(\alpha(\vec{X}\sigma)) \text{ is evaluated to true }\},$$

where $DoNodes$ is the set of all nodes in this graph labeled with an action status atom of the form $\mathbf{Do}(\cdot)$.

The $\mathbf{P}$-, $\mathbf{F}$-part etc. can be computed similarly, reusing already computed nodes. The actions in $DoSet$ are merged calling **conc**$(\mathbf{Do}(DoSet), \mathcal{O}_S)$, and it is checked whether executing the resulting action satisfies the action and integrity constraints.

8 Implementation and Experiments

Our implementation of the regular agent program paradigm consists of two major parts. The first part is the *IMPACT* Agent Development Environment (*IADE* for short) which is used by the developer to build and compile agents. The second part is the run-time part that allows the agent to autonomously update its reasonable status set and execute actions as its state changes. Below, we describe each of these two parts. *IADE* supports both these tasks as follows.

- First, it provides an easy to use, network accessible graphical user interface through which an agent developer can specify the data types, functions, actions, integrity constraints, action constraints, notion of concurrency and agent program associated with her agent.
- Second, it provides support for compilation and testing. In particular, the *IADE* allows the agent developer to specify various parameters (e.g. conflict freedom test, finiteness table) he wants to use for compilation. It implements the $\mathbf{qmp}$:`init` function and also accesses the $\mathbf{qmp}$:`eval` function. It allows the agent developer to view the reasonable status set associated with her agent program with respect to the current state of the agent.

The *IADE* includes the safety, strong safety, conflict freedom algorithms, and **Check_WRAP** algorithms (the last is slightly modified), and the unfold algorithm currently works on positive agent programs - this is being extended to the full fledged case. Figure 4 shows a screendump of the *IADE*'s top-level screen. By selecting one of the "tabs" shown in the figure, the agent developer can specify types, functions, etc. Figure 5 shows the various compilation options—which conflict freedom test is to be used, which semantics is to be computed, etc. Figure 6 shows the result of computing a reasonable status set. This result shows several tabs, one corresponding to each of the deontic operators. For instance, by clicking on the "Doable" tab, the agent developer can view all status atoms of the form $\mathbf{P}\alpha(\vec{t})$ that are true in the unique reasonable status set of the agent.

It is important to note that the above interfaces are only intended for use by the agent developer, and the status set computations shown in Figure 6 are for the agent developer's testing needs. The run-time execution module runs as a background applet and performs the following steps: (i) Monitoring of the agent's message box, (ii) execution of the **Update_Reasonable_SS** algorithm, and (iii) concurrent execution of the actions α such that $\mathbf{Do}(\alpha)$ is in the updated reasonable status set.

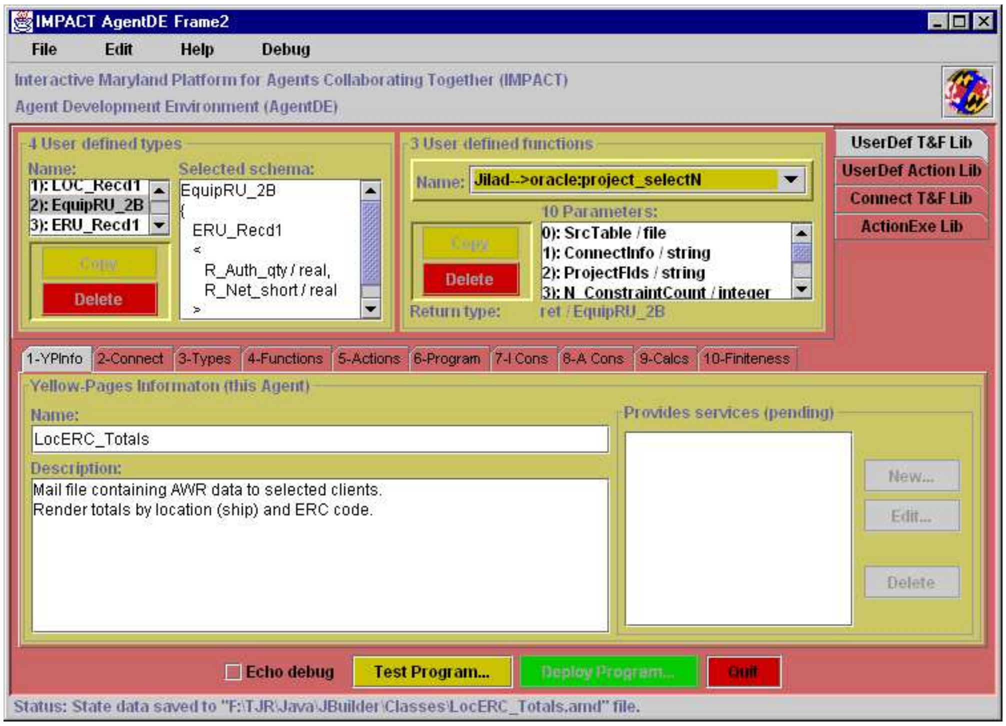


Figure 4: Main *IADE* Screen

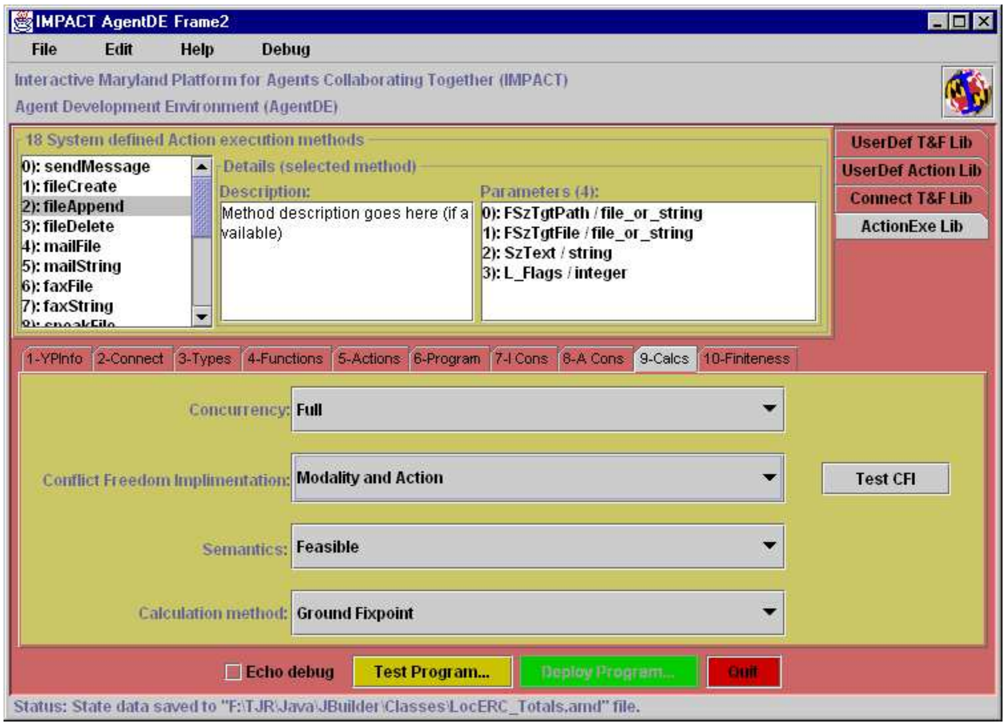


Figure 5: *IADE* Option Selection Screen

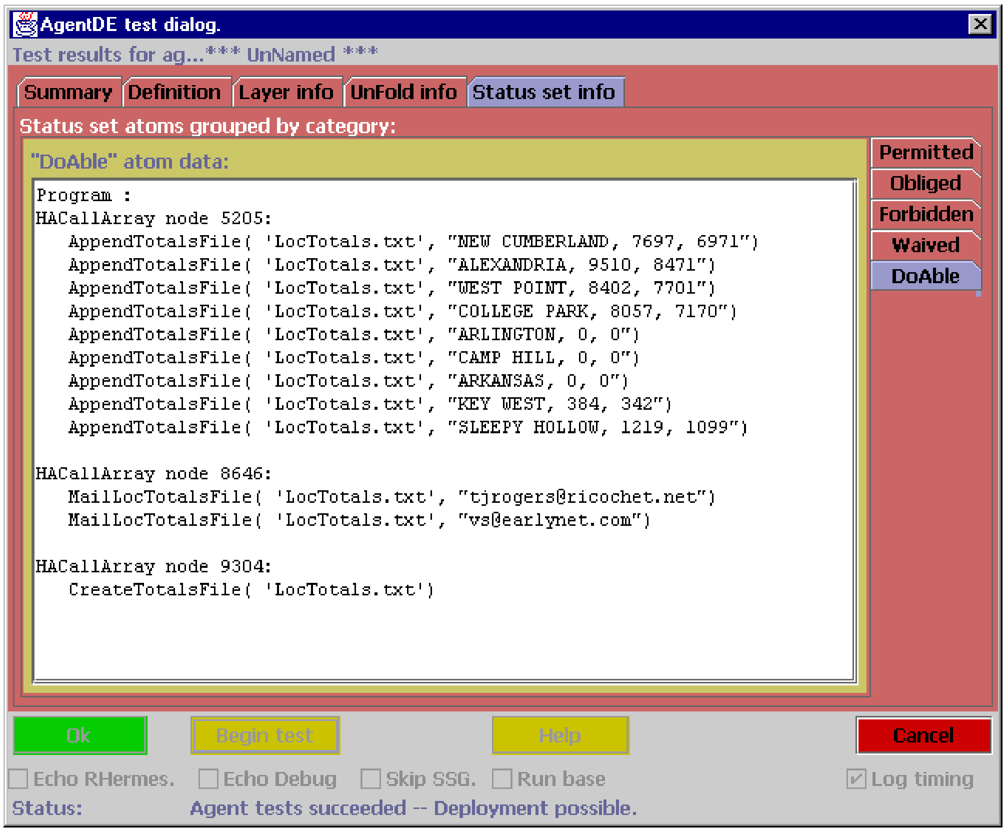


Figure 6: *IADE* Status Set Screen

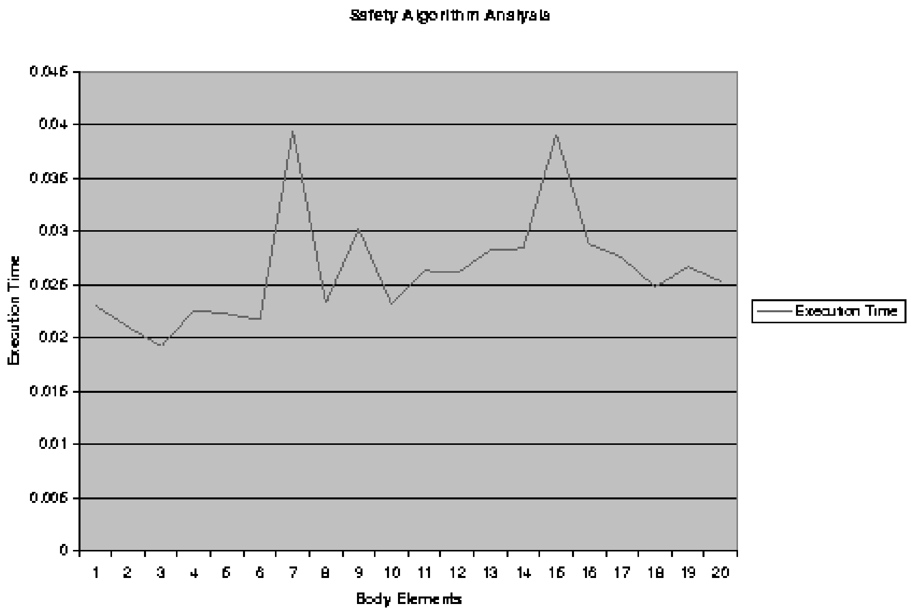


Figure 7: Safety Experiment Graphs

8.1 Performance of safety algorithm

Figure 7 shows the performance of our implemented safety check algorithm. In this experiment, we varied the number of conjuncts in a code call condition from 1 to 20 in steps of 1. This is shown on the x-axis of Figure 7. For each $1 \leq i \leq 20$, we executed the $\mathbf{Safe\_ccc}$ algorithm 1000 times, varying the number of arguments of each code call from 1 to 10 in steps of 1, and the number of root variables occurring in the code call conditions from 1 to twice the number of conjuncts (i.e., 1 to $2i$). The execution time shown on the y-axis represents the average over these 1000 runs, and is in milliseconds. The reader can easily see that the $\mathbf{Safe\_ccc}$ is extremely fast, taking between 0.02 milliseconds and 0.04 milliseconds. Thus, checking safety for an agent program with a 1000 rules can probably be done in 20-40 milliseconds.

Notice that the bounds used in our experiments are a good reflection of reality—we do not expect to see many agent programs with more than 20 conjuncts in the code call part of a single rule body. This is both difficult for a human being to write, and is difficult to read.

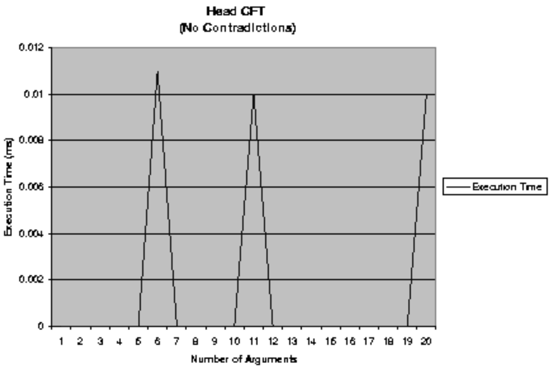
8.2 Performance of selected conflict-freedom tests

We have implemented the Head-CFT and Body-Modality-CFT—several other CFTs are being implemented to form a library of CFTs that may be used by agent developers. Figure 8 shows the time taken to execute the Head-CFT and Body-Modality-CFTs. Note that Head-CFT is clearly much faster than Body-Modality-CFT when returning "`false`"—however, this is so because Head-CFT returns "`false`" on many cases when Body-Modality-CFT does not do so. However, on returns of "`true`", both mechanisms are very fast, usually taking time on the order of a hundredth to a tenth of a millisecond, with some exceptions. These very small times also explain the "zigzag" nature of the graphs—even small discrepancies (on the order of a hundredth of a second) appear as large fluctuations in the graph. Even if an agent program contains a 1000 rules (which we expect to be an exceptional case), one would expect the Body-Modality-CFT to only take a matter of seconds to conduct the one-time, compile-time test—a factor that is well worth paying for in our opinion.

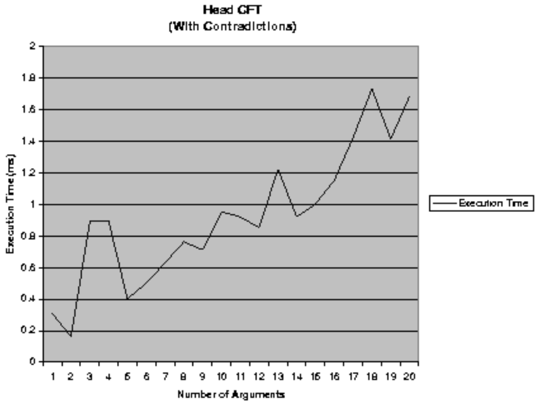
8.3 Performance of deontic stratification algorithm

We conducted experiments with the **Check_WRAP** algorithm. Our experiments did not include timings on the first two steps of this algorithm as they pertain to safety and conflict freedom tests rather than to deontic stratification, and experimental results on those two tests have already been provided above. Furthermore, our experiments generated graphs randomly (as described below) and the programs associated with those graphs can be reconstructed from the graphs.

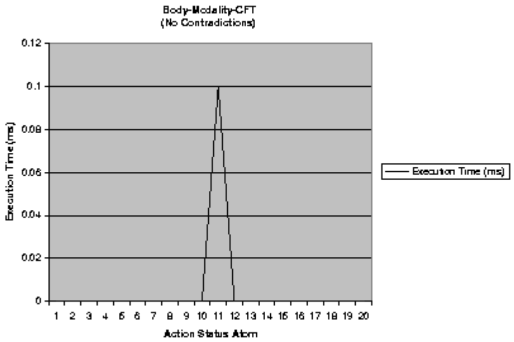
In our experiments, we randomly varied the number of rules from 0 to 200 in steps of 20, and ensured the there were between n and $2n$ edges in the resulting graph, where n is the number of rules (vertices). The precise number was randomly generated. For each such selection, we performed 20 runs of the algorithm. The time taken to generate the graphs was included in these experimental timings. Figure 9 shows the results of our experiments. Figure 9(a) shows the time taken to execute all but the safety and conflict freedom tests of the **Check_WRAP** algorithm. The reader will note that the algorithm is very fast, taking only about 260 milliseconds on an agent program with 200 rules. Figure 9(b) shows the relationship between the number of SCCs in a graph, and the time taken to compute whether the agent program in question is deontically stratified. In this case, we note that as the number of SCCs increases to 200, the time taken goes to about 320 milliseconds. Again, the deontic stratifiability requirement seems to be very efficiently computable.



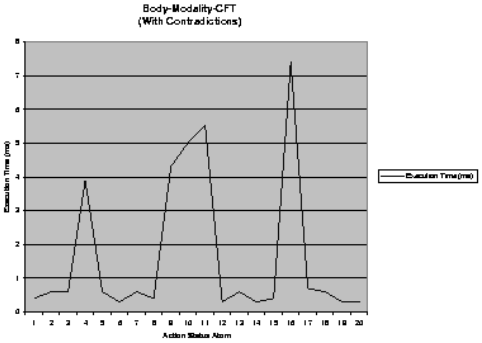
(a) HeadCFT returning "`true`"



(b) HeadCFT returning "`false`"



(c) BodyModalityCFT returning "`true`"



(d) BodyModalityCFT returning "`false`"

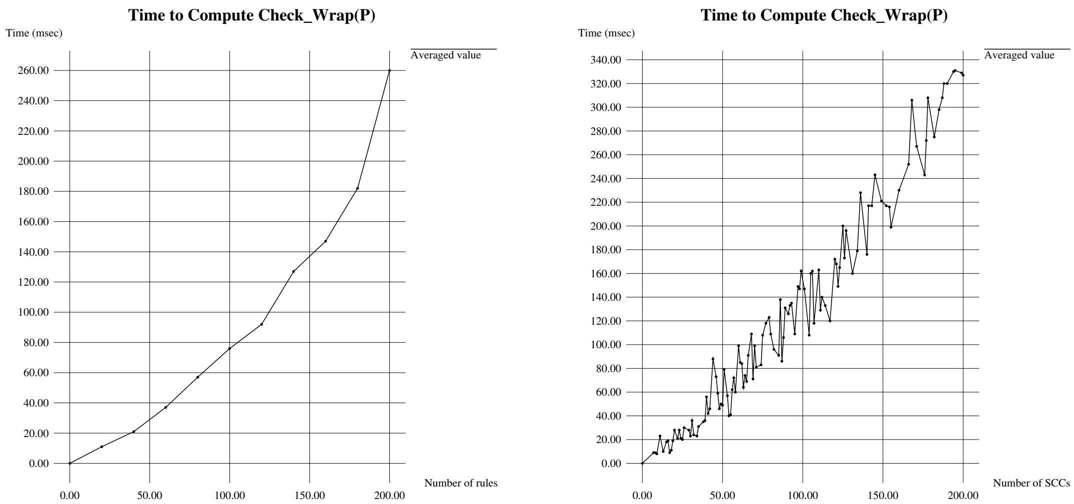
Figure 8: Performance of Conflict Freedom Tests

8.4 Performance of unfolding algorithm

We were unable to conduct detailed experiments on the time taken for unfolding and the time taken to compute status sets as there are no good benchmark agent programs to test against, and no easy way to vary the very large number of parameters associated with an agent. In a sample application (described in Section 8.5 below, it took 1.43 seconds to unfold a program containing 13 agent rules used to implement a logistics agent. Another agent for logistics purposes that contained 16 agent rules took 150 milliseconds.

8.5 Performance of reasonable status set computation algorithm

We implemented and executed the status set computation algorithm on two different agents used for logistics monitoring on US Army War Reserves Data [35]. Both agents access two data sources—one is an Oracle data source containing two tables—an *EquipRU* and an *Apr_loc* file comprising of 4,721 and 155 records, respectively. The rest of the data accessed by these agents is stored in a system called LogTAADS (Logistics



(a) Varying Rules (b) Varying SCC's

Figure 9: Performance of Deontic Stratification

- The Army Authorization Document System) which consists of a single-file, multitable structure. In other words, this file consists of a set of (actually four) distinct tables. The *WM_MOC* file contains 68,146 records. No functions were available to access this data—hence, we had to implement our own functions to do so [35]. Both these data sources were located on different machines at the University of Maryland, and the *IMPACT* agent development environment was located on yet another machine.

One of the agents contained 8 rules—we ran experiments five times and found that on the average, it took 26.925 seconds to compute its unique reasonable status set. However, of these 26.925 seconds, 20.286 seconds were spent inside the code on top of which the agent was built (in this case, Oracle and LogTAADS) and/or in network communications—the time spent within IMPACT itself was only 6.639 seconds.

In the second case, the agent contained 11 rules and averaging over five runs, it took 30.131 seconds to compute its unique reasonable status set. Of this, 23.463 seconds were spent either in network time or inside the code on top of which the agent was built (in this case, Oracle and LogTAADS)—the time spent within IMPACT itself was only 6.668 seconds.

Thus, the cost of computing reasonable status sets is very small as the cost of computing with legacy code and commercial code cannot be improved (by us).

9 Related Work and Conclusions

In this paper, we have attempted to define a polynomially implementable class of agent programs and described how we implemented this class of programs.

As defined in this paper, a regular agent program satisfies four conditions—strong safety, conflict freedom, deontic stratifiability, and a boundedness condition. Each of these parameters has been studied in the literature, at least to some extent, and we have built upon those works.

- The concept of safety is related to the notion of *mode realizability* in logic programs [33, 8]. In order to evaluate the truth or falsity of some atoms in a logic program, certain arguments of that atom may need to be instantiated. This is similar, but not identical to the notion of safety where we have similar conditions on code call conditions. Strong safety requires the important finiteness property in addition to this.

- The concept of conflict freedom has been studied in logic programming when negations are allowed in both the head and the body of a rule. Such logic programs were introduced by Gelfond and Lifschitz [20] and contradiction removal in such programs was studied extensively by Pereira's group [3]. Our work differs from these in the sense that we are looking for syntactic conditions on agent programs (rather than logic programs) that guarantee that under all possible states of the agent, conflicts will not occur. Such a test can be encoded at compile time.

- The notion of deontic stratifiability of an agent program, builds directly on top of the concept of a stratified logic program introduced by Apt *et al.* [4]. We extend the concept of stratified logic programs to the case of a deontic stratified agent program modulo a conflict freedom test. Checking deontic stratifiability is somewhat more complex than checking ordinary stratifiability, and hence, our algorithms to do this are new.

- The notion of boundedness of an agent program builds upon the well known idea of unfolding (or partial evaluation) in logic programs. This area has been recently studied formally for semantics of (disjunctive) logic programs (well-founded as well as stable) in [9, 10, 11]. The use of Tarjan's algorithm for computing the well-founded semantics in almost linear time has been explicitly addressed e.g. in [7, 15].

To date, we are not aware of any existing work on the semantics of agent programs that is polynomial and that has been implemented. In this paper, we have described a wide variety of parameters (e.g. conflict freedom tests, finiteness tables, etc.) that go into the design and development of an agent, and we have provided experimental data showing that these algorithms work effectively. To our knowledge, this is one of the first attempts to do this for a *generic, application independent* agent programming paradigm.

Acknowledgements. We would like to thank Anatoly Levkov, Rob Ross, and Mustafa Tikir for writing code to implement some algorithms defined in this paper. We have benefited from discussions with Jürgen Dix, Bob Kowalski, and Sarit Kraus. We would also like to thank for the review comments which helped to improve the paper.

This work was supported by the Army Research Office under Grants DAAH-04-95-10174, DAAH-04-96-10297, and DAAH04-96-1-0398, by the Army Research Laboratory under contract number DAAL01-97-K0135, by an NSF Young Investigator award IRI-93-57756, by a DAAD grant, and by the Austrian Science Fund Projects N Z29-INF and P13871-INF.

A Appendix: Proofs

Proof of Theorem 3.1. Clearly, by construction of χ', we have that χ' is a safe code call condition modulo $\vec{X}$ (a suitable permutation for χ' is the identity). Thus, if the call of **Safe_ccc**$(\chi, \vec{X})$ returns χ', then χ is safe modulo $\vec{X}$, as a suitable permutation for π is given by the reordering of the constituents of χ realized in χ'.

On the other hand, suppose χ is safe modulo $\vec{X}$. Then, some i from $1, \dots, n$ must exist such that χ_i is safe modulo $\vec{X}$. If, without loss of generality, $\chi_1, \dots, \chi_k$ are all such χ_i, then it is easily seen that $\chi^{(1)} = \chi_{k+1} \& \cdots \& \chi_m$ must be safe modulo the variables $\vec{X}^{(1)} = \vec{X} + (root(Y) \mid Y \text{ occurs in } \chi_1 \& \cdots \& \chi_k)$. Employing an inductive argument, we obtain that the continuation of the computation succeeds for $L = \chi_{k+1}, \dots, \chi_n$ modulo $\vec{X}^{(1)}$, and hence **Safe_ccc**$(\chi, \vec{X})$ returns a reordered code call condition χ'. (The interested reader may observe that **Safe_ccc** can be easily rewritten to a recursive algorithm.) Obviously, χ' is safe, and for any assignment θ to the variables in $\vec{X}$ (which are all root variables), the code call condition $\chi'\theta$ can be readily evaluated left-to-right. ∎

Proof of Theorem 3.2. The undecidability of this problem is inherited from the undecidability of the problem whether a function $\mathtt{f} \in \mathcal{F}$ from a software code $\mathcal{S} = (\mathcal{T}, \mathcal{F})$ returns a particular value on at least one agent state $\mathcal{O}_{\mathcal{S}}$. We may choose for $\mathcal{S}$ a standard relational database package, and let $\mathtt{f}$ be a Boolean query on the database written in e.g. SQL. Then, it is undecidable whether $\mathtt{f}$ evaluates to true over some (finite) database, i.e., agent state $\mathcal{O}_{\mathcal{S}}$; this follows from well-known results in the area of database theory. In particular, relational calculus is undecidable, and every query in relational calculus an be expressed in SQL; see [1]. Now, define the rules:

$$\begin{aligned} r_1 : \mathbf{P}(\alpha) &\leftarrow \\ r_2 : \mathbf{F}(\alpha) &\leftarrow \mathbf{in}(\mathtt{true}, \mathbf{oracle}{:}\mathtt{f}()). \end{aligned}$$

Then, the rules r_1 and r_2 are conflict free iff $\mathtt{f}$ does not return $\mathtt{true}$, over any database instance. This proves the result. ∎

Proof of Theorem 3.3. (1) To show that $r^\star$ is safe we need to show that the two conditions defining safety hold for $r^\star$. Suppose r is of form

$$A \leftarrow B_{cc}(r) \,\&\, Op\alpha(\vec{t}) \,\&\, B^+_{as,rest}(r) \,\&\, B^-_{as}(r).$$

where the precondition of $\alpha(\vec{Y})$ is $\chi(\vec{Z})$ (both standardized apart from r) where $\vec{Z}$ contains all variables occurring in α's precondition. Then $r^\star$ is the rule

$$A \leftarrow B_{cc}(r) \,\&\, \chi(\vec{Z}\theta) \,\&\, Op\alpha(\vec{t}) \,\&\, B^+_{as,rest}(r) \,\&\, B^-_{as}(r)$$

where θ is the substitution $\vec{Y} = \vec{t}$. Since χ is safe modulo $\vec{Y}$, $\chi(\vec{Z}\theta)$ is safe modulo the list of variables in $\vec{t}$. As all variables in $\vec{t}$ occur in $B^+_{as}(r)$, $\chi(\vec{Z}\theta)$ is safe modulo the variables in $B^+_{as}(r)$. It follows immediately that $B_{cc}(r)\,\&\chi(\vec{Z}\theta)$ is safe modulo the variables in $B^+_{as}(r)$. Thus, $r^\star$ satisfies the first definition of safety. The second condition in the definition of safety is trivially satisfied since the only new variables in $r^\star$ are in $B_{cc}(r^\star)$.

(2) Follows immediately from the strong safety of α's precondition, and part (1) above. ∎

Proof of Proposition 3.4. *Item 1 of deontic stratifiability*: Suppose that $r_i : Op_i(\alpha(\vec{t})) \leftarrow \ldots, Op_j(\beta(\vec{t'})), \ldots$ is in $\mathcal{P}_i$, and that $r : Op(\beta(\vec{t''})) \leftarrow \cdots$ is a rule in $\mathcal{P}$ such that $\beta(\vec{t'})$ and $\beta(\vec{t''})$ are unifiable and $Op \leq Op_j$. Since $\mathcal{P}$ is deontic stratifiable, in every layering $\ell \in \mathsf{wtn}(\mathcal{P})$ it holds that $\ell(r) \leq \ell(r_i)$. Taking minimal values as in the definition of $\mathsf{can}^{\mathcal{P}}$, it follows that $\mathsf{can}^{\mathcal{P}}(r) \leq i = \mathsf{can}^{\mathcal{P}}(r_i)$.

Item 2 of deontic stratifiability: As in the previous case, for rules r_i and r as in the second stratifiability condition, every layering $\ell \in \mathsf{wtn}(\mathcal{P})$ satisfies $\ell(r) < \ell(r_i)$. Thus, it follows that $\mathsf{can}^{\mathcal{P}}(r) < \mathsf{can}^{\mathcal{P}}(r_i)$. ∎

Proof of Theorem 4.1. It is straightforward to show that if **Check_WRAP** returns a layering ℓ, then $\mathcal{P}$ is weakly regular and $\ell \in \mathsf{wtn}(\mathcal{P})$ holds. On the other hand, suppose the algorithm returns "no". If it halts in Step 2 or 3, then the first (resp., second) condition of conflict freedom is violated for any layering ℓ, and thus $\mathcal{P}$ is not weakly regular. If it halts in Step 6, then, by definition of G, a sequence of rules $r_0, r_1, \ldots, r_n$, $n \geq 1$, exists such that any layering ℓ satisfying the stratifiability conditions must, without loss of generality, satisfy $\ell(r_0) \leq \ell(r_1) \leq \cdots \leq \ell(r_n)$ and $\ell(r_n) < \ell(r_0)$. However, this means $\ell(r_0) < \ell(r_0)$, which implies that such an ℓ is impossible. Thus, the algorithm correctly returns that $\mathcal{P}$ is not weakly regular. ∎

Proof of Proposition 5.3. That $\mathcal{P}$ has at most one rational status set follows from Proposition 5.2. Since S is deontic and action closed, it remains to verify that S is deontically consistent. Suppose this is not the case. Then, at least one of the three conditions (D1)–(D3) of deontic consistency of S is violated. For each case, we derive a contradiction, which proves the result.

(D1) $\mathbf{O}\alpha \in S$ and $\mathbf{W}\alpha \in S$, for some ground action α. This means that there are rules r and r' in $\mathcal{P}$ with heads $\mathbf{O}(\beta(\vec{t}))$ and $\mathbf{W}(\beta(\vec{t'}))$ such that, standardizing their variables apart, for some ground substitution θ it holds that $\beta(\vec{t}\theta) = \beta(\vec{t'}\theta)$, $(\exists)(B_{cc}(r) \wedge B_{cc}(r'))\theta$ is true and $\beta(\vec{t}\theta)$ is executable with respect to $\mathcal{O}_{\mathcal{S}}$, and $B_{as}(r) \cup B_{as}(r')$ is true in S, and hence does not contain a pair of conflicting literals. However, this means that the rules r and r' conflict with respect to $\mathcal{O}_{\mathcal{S}}$. This implies that $\mathsf{cft}(r, r')$ = *false*, and provides a contradiction to the conflict-freedom condition of the definition of weak regularity.

(D2) $\mathbf{P}\alpha \in S$ and $\mathbf{F}\alpha \in S$. As in the previous case, we conclude that for each agent state $\mathcal{O}_{\mathcal{S}}$, $\mathcal{P}$ contains rules r and r' with heads $\mathbf{F}(\beta(\vec{t}))$ and $Op(\beta(\vec{t'}))$, respectively, where $Op \in \{\mathbf{P}, \mathbf{Do}, \mathbf{O}\}$, such that r and r' are conflicting with respect to $\mathcal{O}_{\mathcal{S}}$. Again, this contradicts the fact that $\mathcal{P}$ is weakly regular.

(D3) $\mathbf{P}\alpha \in S$ but α is not executable in $\mathcal{O}_{\mathcal{S}}$. Then, there must exist a rule $r \in \mathcal{P}$ and a θ such that $R(r, \theta, S)$ is true, $head(r\theta) \in \mathbf{App}_{\mathcal{P},\mathcal{O}_{\mathcal{S}}}(S)$, and $head(r\theta)$ =$Op\alpha$, where $Op \in \{\mathbf{P}, \mathbf{Do}, \mathbf{O}\}$. The definition of $\mathbf{App}_{\mathcal{P},\mathcal{O}_{\mathcal{S}}}(S)$ implies that α must be executable in $\mathcal{O}_{\mathcal{S}}$. This is a contradiction. ∎

Proof of Proposition 5.4. Let $X = \mathcal{T}_{\mathcal{P}}\uparrow\omega(S)$ (=$\bigcup_{i=0}^{\infty} \mathcal{T}_{\mathcal{P}}\uparrow i(S)$). We have to show that $\mathcal{T}_{\mathcal{P}}(X) = X$ holds. As $\mathcal{T}_{\mathcal{P}}$ is inflationary (Proposition 5.1), it remains to show $\mathcal{T}_{\mathcal{P}}(X) \subseteq X$, i.e., that each atom $A \in \mathcal{T}_{\mathcal{P}}(X)$ is in X. The are two cases to consider.

(1) $A \in \mathbf{App}_{\mathcal{P}}(X)$. Then, a rule $r \in \mathcal{P}$ and a θ exist such that $head(r\theta) = A$ and $R(r, \theta, X)$ is true, i.e., (a) $r\theta$ is ground; (b) $B_{cc}(r\theta)$ is true in $\mathcal{O}_{\mathcal{S}}$; (c) $B^+_{as}(r\theta) \subseteq X$; (d) $\neg.B^-_{as}(r\theta) \cap X = \emptyset$; (e) for every atom $Op\,\alpha \in B^+_{as}(r\theta) \cup H(r\theta)$ where $Op \in \{\mathbf{P}, \mathbf{Do}, \mathbf{O}\}$, the action α is executable in $\mathcal{O}_{\mathcal{S}}$.

As $B^+_{as}(r\theta)$ is finite and $\mathcal{T}_{\mathcal{P}}$ is inflationary, condition (c) implies that $B^+_{as}(r\theta) \subseteq \mathcal{T}_{\mathcal{P}} \uparrow k(S)$ holds for some $k \geq 0$. Furthermore, condition (d) implies that $\neg.B^-_{as}(r\theta) \cap \mathcal{T}_{\mathcal{P}} \uparrow k(S) = \emptyset$. This means that $R(r, \theta, \mathcal{T}_{\mathcal{P}}\uparrow k(S))$ is true; hence, $A \in \mathcal{T}_{\mathcal{P}}(\mathcal{T}_{\mathcal{P}}\uparrow k(S)) \subseteq \mathcal{T}_{\mathcal{P}}\uparrow k+1(S)$, which means $A \in X$.

(2) $A \in \mathbf{A\text{-}Cl}(X)$. Then, a $B \in X$ and a $k \geq 0$ exist such that $B \leq A$ and $B \in \mathcal{T}_{\mathcal{P}} \uparrow k(S)$. Hence, $A \in \mathcal{T}_{\mathcal{P}}\uparrow k+1(S)$ holds, and thus $A \in X$.

This proves that $\mathcal{T}_{\mathcal{P}}\uparrow\omega(S)$ is a fixpoint of $\mathcal{T}_{\mathcal{P}}$. The argument in case (2) implies that $X = \mathcal{T}_{\mathcal{P}}\uparrow\omega(S)$ is action-closed. ∎

Notation. Suppose $\mathcal{P}$ is a deontically stratified agent program, which is witnessed by layering function ℓ, and suppose the layers of $\mathcal{P}$ induced by ℓ are $\mathcal{P}_0, \ldots, \mathcal{P}_k$. We use $S^\ell_{0,j}$ to denote $\mathcal{T}_{\mathcal{P}_0}\uparrow j(\emptyset)$ for all $j \geq 0$ and use $S^\ell_{i+1,j}$ to denote $\mathcal{T}_{\mathcal{P}_{i+1}}\uparrow j(\Gamma^\ell_{\mathcal{P}}\uparrow i)$, for all $i, j \geq 0$, i.e., $S^\ell_{i,j}$ contains the result of computing $\Gamma^\ell_{\mathcal{P}}\uparrow\omega$ after step j in level i. We shall drop the superscript ℓ when it is clear from the context. Note that $S_{i,j}$ monotonically increases, i.e., $S_{i,j} \subseteq S_{i',j'}$ if $(i, j) < (i', j')$ under the standard lexicographic ordering.

Proof of Theorem 5.5. For the proof of this result, we use the following technical lemma, which says that in the computation of $\Gamma^\ell_{\mathcal{P}}\uparrow\omega$, the applicability of rules is preserved.

Lemma A.1 *Suppose $\mathcal{P}$ is a WRAP, and let $\ell \in \mathsf{wtn}(\mathcal{P})$. If, for some rule $r \in \mathcal{P}_i$ and stage $S_{i,j}$, it holds that $R(r, \theta, S_{i,j})$ is true, then $R(r, \theta, S_{i',j'})$ is true, for every stage $S_{i',j'}$ such that $(i, j) < (i', j')$, and $R(r, \theta, S)$ is true where $S = \Gamma^\ell_{\mathcal{P}}\uparrow\omega$.*

Proof (of Lemma A.1). Suppose that $R(r, \theta, S_{i,j})$ is true. Thus, (1) $r\theta$ is ground, (2) $B_{cc}(r\theta)$ is true in $\mathcal{O}_{\mathcal{S}}$, (3) $B^+_{as}(r\theta) \subseteq S_{i,j}$, and (4) $\neg.B^-_{as}(r\theta) \cap S_{i,j} = \emptyset$, and (5) for every atom $Op\alpha \in B^+_{as}(r\theta) \cup \{A\}$ such that $Op \in \{\mathbf{P}, \mathbf{O}, \mathbf{Do}\}$, α is executable in $\mathcal{O}_{\mathcal{S}}$.

Since $S_{i,j} \subseteq S_{i',j'} \subseteq S$ holds for all i', j' as in the statement of the lemma, proving $\neg.B^-_{as}(r\theta) \cap S = \emptyset$ will establish the lemma. Assume this is not true, i.e., an atom $A \in \neg.B^-_{as}(r\theta) \cap S$ exists. This implies that there exists a rule $r' \in \mathcal{P}$ such that for some θ' and i^*, j^*, $R(r', \theta', S_{i^*,j^*})$ is true and $head(r'\theta') \leq A$. Condition (2) of deontic stratifiability implies $\ell(r') < \ell(r)$, and thus $i^* < i$ can be assumed. Since the stages in the construction of S monotonically increase, it follows that $head(r'\theta')$ and A are contained in $S_{i,j}$; thus, $\neg.B^-_{as}(r\theta) \cap S_{i,j} \neq \emptyset$. This means that $R(r, \theta, S_{i,j})$ is not true, which is a contradiction. Thus, $\neg.B^-_{as}(r\theta) \cap S = \emptyset$. ∎

Let $S = \Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$. To show that S is a reasonable status set of $\mathcal{P}$, we must show that S is a feasible status of $\mathcal{P}' = red^S(\mathcal{P}, \mathcal{O}_S)$, and no smaller $S' \subset S$ exists which satisfies conditions (S1)–(S3) of feasibility for $\mathcal{P}'$.

To show that S is a feasible status set of $\mathcal{P}'$, we must show that (S1) S is closed under program rules of $\mathcal{P}'$, that (S2) S is deontically and action consistent, that (S3) S is action closed, and that (S4) the state consistency condition is satisfied.

(S1) To see that S is closed under rules from $\mathcal{P}'$, suppose there exists a rule r in $\mathcal{P}'$ of the form $r : A \leftarrow L_1, \ldots, L_n$ such that $R(r, \theta, S)$ is true on $\mathcal{O}_S$ for some θ. As r is in fact ground (and thus $r\theta = r$ and θ is irrelevant) and $B^-_{as}(r) = \emptyset$, this implies

1. $B_{cc}(r)$ is true on $\mathcal{O}_S$,
2. $B^+_{as}(r) \subseteq S$, and
3. for every atom $Op\alpha \in B^+_{as}(r) \cup \{A\}$ such that $Op \in \{\mathbf{P}, \mathbf{O}, \mathbf{Do}\}$, α is executable in $\mathcal{O}_S$.

We have to show that $A \in S$. As $B^+_{as}(r)$ is finite, item 2. implies that $B^+_{as}(r) \subseteq \Gamma^{\ell}_{\mathcal{P}} \uparrow k$ for some integer k. Let k^* be the least such integer. For each atom $A \in B^+_{as}(r)$, there is rule $r' \in \mathcal{P}$ and a ground substitution θ' such that $r'\theta'$ is applied in the construction of $\Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$ and $head(r'\theta') \leq A$ (i.e., A is either included directly by applying $r'\theta'$, or indirectly by applying $r'\theta'$ and action closure rules). The rule r stems from the ground instance $r''\theta''$ of a rule $r'' \in \mathcal{P}$. Item (1) of deontic stratifiability implies that $\ell(r') \leq k^* \leq \ell(r'')$ holds. As $\Gamma^{\ell}_{\mathcal{P}} \uparrow k^* \subseteq S$ and $\neg.B^-_{as}(r''\theta'') \cap S = \emptyset$, it follows that the rule $r''\theta''$ is applied in the construction of $\Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$, and thus $head(r''\theta'') = A$ is included in S.

(S2) Since $\mathcal{AC} = \emptyset$, S is trivially action consistent. To see that S is deontically consistent, assume it is not. Thus, it must violate some deontic consistency rule (D1)–(D3). As in the proof of Proposition 5.3, it can be shown using Lemma A.1 that each such violation raises a contradiction.

(S3) Clearly, $\mathbf{A\text{-}Cl}(S) = \bigcup_{i=0}^{\infty} \mathbf{A\text{-}Cl}(\Gamma^{\ell}_{\mathcal{P}} \uparrow i)$ holds by definition of S and the fact that $\mathbf{A\text{-}Cl}(X) = \bigcup_{X' \in \mathcal{C}} \mathbf{A\text{-}Cl}(X')$ holds for every collection $\mathcal{C}$ of subsets of an arbitrary status set X such that $\bigcup_{X' \in \mathcal{C}} X' = X$. Applying Proposition 5.4, it follows $\mathbf{A\text{-}Cl}(S) = S$.

(S4) This is trivial as $\mathcal{IC} = \emptyset$ is assumed.

At this stage, we have shown that S is a feasible status set of $\mathcal{P}'$. To establish that S is rational, suppose $S' \subset S$ is a status set satisfying conditions (S1)–(S3) for $\mathcal{P}'$. Let $S_{i,j}$ be the first stage in the construction of S such that $S_{i,j} \setminus S' \neq \emptyset$. Let $A \in S_{i,j} \setminus S'$ be any atom. It follows that there exist a rule $r \in \mathcal{P}_i$ and a θ such that $A = head(r\theta)$ and $R(r, \theta, S_{i,j-1})$ is true. By Lemma A.1, also $R(r, \theta, S)$ is true; item (4) of the definition of $R(r, \theta, S)$ implies that the rule r' obtained from $r\theta$ by removing $B^-_{as}(r\theta)$ belongs to $\mathcal{P}'$. Furthermore, the minimality of $S_{i,j}$ implies that $B^+_{as}(r')$ (=$B^+_{as}(r\theta)$) is contained in S'. Thus, $R(r', \theta, S')$ is true. This implies $A \in S'$, which is a contradiction. Thus, a feasible status set $S' \subset S$ of $\mathcal{P}$ does not exist. This proves the result. ∎

Proof of Corollary 5.6. By Lemma 5.2 in [17] every status S such that conditions (S1) and (S3) of feasibility hold satisfies $\mathcal{T}_{\mathcal{P}}(S) \subseteq S$. Theorem 5.5 implies that $S = \Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$ satisfies (S1) and (S3). Since $S \subseteq \mathcal{T}_{\mathcal{P}}(S)$ holds for every status set S (Proposition 5.1), the result follows. ∎

Proof of Theorem 5.7. To prove this result, it is sufficient to show by induction on $i \geq 0$ that for every rule $r \in \mathcal{P}_i$ and ground substitution θ it holds that

$$R(r, \theta, S) \longleftrightarrow \exists j. R(r, \theta, S_{i,j}).$$

Without loss of generality, we assume that $\mathcal{P}_0 = \emptyset$.

Then, the base case $i = 0$ is trivial, as no rule in $\mathcal{P}_0$ exists. For the inductive case, assume the statement holds for all $j \leq i$ and consider the case $i + 1 > 0$. We have to show that

$$\forall r \in \mathcal{P}_{i+1} \forall \theta. R(r, \theta, S) \longleftrightarrow \exists j. R(r, \theta, S_{i+1,j})$$

holds. We consider the two directions of this equivalence.
($\longleftarrow$) Suppose that $R(r, \theta, S_{i+1,j})$ is true for a particular j. We have to show that $R(r, \theta, S)$ is true. By the definition of predicate R, it remains to show that (i) $B^+_{as}(r\theta) \subseteq S$ and (ii) $\neg.B^-_{as}(r\theta) \cap S = \emptyset$. We prove this by induction on $j \geq 0$.

For the base case $j = 0$, we obtain from item 1 of deontic stratifiability that for each atom $A \in B^+_{as}(r\theta)$, a rule $r' \in \mathcal{P}_{i'}$ where $i' \leq i$ and θ' exist such that $head(r'\theta') \leq A$ and $R(r', \theta', S_{i',j'})$ is true for some $j' \geq 0$. Then, by the outer induction hypothesis on i, it follows that $R(r', \theta', S)$ is true, which implies $A \in S$. Thus, (i) holds. For (ii), truth of $R(r, \theta, S_{i+1,j})$ implies that no atom $A \in \neg.B^-_{as}(r\theta)$ is contained in $S_{i+1,j}$. Item 2 of deontic stratifiability of $\mathcal{P}$ implies that every rule r' such that $head(r'\theta') \leq A$ for some θ' is contained in $\mathcal{P}_{i'}$ for some $i' \leq i$. Furthermore, $S_{i',j'} \subseteq S_{i+1,j}$ implies that $R(r', \theta', S_{i',j'})$ is false, for every $j' \geq 0$. Hence, by the outer induction hypothesis on i, $R(r', \theta', S)$ is false. This implies $A \notin S$, and hence $\neg.B^-_{as}(r\theta) \cap S = \emptyset$ is true. This concludes the proof of the inner base case $j = 0$.

For the inner induction step, suppose the statement holds for all $j' \leq j$ and consider $j + 1 > 0$. The proof of (i) is similar to the case $j = 0$, but takes into account that r' and θ' for A may also be such that $r' \in \mathcal{P}_{i+1}$ and $R(r', \theta', S_{i+1,j'})$ holds where $j' \leq j$. In this case, the inner induction hypothesis on j implies that $R(r', \theta', S)$ is true. The proof of (ii) is analogous to the case $j = 0$.
($\longrightarrow$) We have to show that $R(r, \theta, S)$, where $r \in \mathcal{P}_{i+1}$, implies that $\exists j. R(r, \theta, S_{i+1,j})$ is true. We prove the following equivalent claim. Let $\mathcal{P}' = red^S(\mathcal{P}, \mathcal{O}_{\mathcal{S}})$. Then, for every atom $A \in \mathcal{T}_{\mathcal{P}'} \uparrow \omega(\emptyset)$ for which $r \in \mathcal{P}_{i+1}$ and θ exist such that $A = head(r\theta)$ and $R(r, \theta, S)$ is true, $\exists j. R(r, \theta, S_{i+1,j})$ is true. The proof is by induction on the stages $\mathcal{T}_{\mathcal{P}'} \uparrow k(\emptyset)$, $k \geq 0$, of the fixpoint iteration for $\mathcal{P}'$.

The base case $k = 0$ is trivial, since $\mathcal{T}_{\mathcal{P}'} \uparrow 0(\emptyset) = \emptyset$. For the induction step, suppose the statement holds for all $k' \leq k$, and consider $k + 1 > 0$. Let $A \in \mathcal{T}_{\mathcal{P}'} \uparrow k + 1(\emptyset) \setminus \mathcal{T}_{\mathcal{P}'} \uparrow k(\emptyset)$ and r, θ as in the premise of the statement. From item (1) of deontic stratifiability of $\mathcal{P}$, the outer induction hypothesis on i, and the inner induction hypothesis on k, it follows that each $A \in B^+_{as}(r\theta)$ is contained in S_{i+1,j_A} for some $j_A \geq 0$. Since $B^+_{as}(r\theta)$ is finite, $B^+_{as}(r\theta) \subseteq S_{i+1,j}$ holds where $j = \max\{j_A \mid A \in B^+_{as}(r\theta)\}$. To show that $R(r, \theta, S_{i+1,j})$ holds for this j, it remains by the definition of predicate R to show that $\neg.B^-_{as}(r\theta) \cap S_{i+1,j} = \emptyset$ holds. Item (2) of deontic stratifiability and the outer induction hypothesis imply that no atom $A \in \neg.B^-_{as}(r'\theta')$ is contained in $S_{i+1,j}$; thus, $\neg.B^-_{as}(r'\theta') \cap S_{i+1,j} = \emptyset$, where $j \geq 0$ is arbitrary. This proves that $R(r, \theta, S_{i+1,j})$ holds, and thus $\exists j. R(r', \theta', S_{i+1,j})$ is true. This concludes the proof of the inner induction step on $k + 1$, and also the proof of the outer inductive step $i + 1$. ∎

Proof of Theorem 6.1. We have to show that each of the steps 1 – 5 of the algorithm can be done in finite time. As for Step 1, the boundedness of the agent program $\mathcal{P}$ associated with $\mathbf{a}$ ensures that the set $\Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$

is computable within a bounded number of steps: For computing $\Gamma^{\ell}_{\mathcal{P}} \uparrow i$, we must compute the operator $\mathcal{T}_{\mathcal{P}_i} \uparrow \omega$, which needs to apply the operator $\mathcal{T}_{\mathcal{P}_i}$ associated with the layer $\mathcal{P}_i$ only a bounded number of times. Furthermore, the number of nonempty layers $\mathcal{P}_i$ is bounded as well. An inductive argument shows that in each step of the computation, any rule r from the layer $\mathcal{P}_i$ currently considered instantiates because of strong safety only to a finite number of ground rules r' which fire. These r' can be effectively computed from the status set S_k derived so far (where $S_0 = \emptyset$), proceeding in the way as described after Def. 3.3. Indeed, the set Θ of ground substitutions there such that $B^+_{as}(r)$ is true with respect to S_k is finite (as, by induction hypothesis, S_k is finite), and can be computed in finite time. Next, for each $\theta \in \Theta$, the set of all ground substitutions Γ_θ that satisfy $B_{cc}(r\theta)$ is finite and effectively computable. Furthermore, Γ^+_θ and Ans are effectively computable from Γ_θ. Thus, firing instances of the rule r in the current step can yield only finitely many new ground action status atoms, which means that S_{k+1} finitely computed. It follows that $\Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$ is computed within finite time.

Step 2 is simple. Step 3 can be effectively accomplished: Strong safety of each action constraint ac : $\{\alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)\} \hookleftarrow \chi$ ensures that χ has only finitely many solutions θ, which can be effectively computed; furthermore, matching the head $\alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)$ to atoms $\alpha_1(\vec{t}_1), \ldots, \alpha_k(\vec{t}_k)$ in $\mathbf{Do}(S)$ such that $\alpha_i(\vec{X}_i\theta') = \alpha_i(\vec{t}_i)$, $i = 1, \ldots, k$, where θ' extends θ, can be done in polynomial time in the size of $\mathbf{Do}(S)$.

The new agent state $\mathcal{O}'_{\mathcal{S}}$ in Step 4 is, by specification, effectively computable. Finally, also Step 5 can be done in finite time, since strong safety implies that for each integrity constraint $ic : \psi \Rightarrow \chi$ in $\mathcal{IC}$, the body ψ has only a finite number of ground instances $\psi\theta$ which are true in the agent state, and they are effectively computable; since χ is strongly safe checking whether $\chi\theta$ is true is possible in finite time. ∎

Proof of Theorem 6.2. We have to argue that each of the steps 1 – 5 can be done in polynomial time, rather than in arbitrary finite time. This can be accomplished by refining the analysis in the proof of Theorem 6.1.

As for Step 1, the cardinality of the set Θ of substitutions such that $B^+_{as}(r)$ is true with respect to the already derived status set S_k is polynomial in the the size of S_k. Under the assumptions, this set is computable in polynomial time. Next, for each $\theta \in \Theta$, the assumptions imply that the set Γ_θ contains only polynomially many assignments γ, each of which is computable in polynomial time. The check whether $B^-_{as}(r')$ where $r' = (r\theta\gamma)$ is true with respect to S_k is easy, and also the test whether $head(r')$ is added to Ans is feasible in polynomial time. Overall, the number of instances r' of r which eventually fire is polynomial in the size S_k and the agent state.

This means the number $N_k = |S_k|$ of atoms $Op\alpha$ that are derived after k steps of firing rules (where $N_0 = 0$), is bounded by $p(N_{k-1}, |\mathcal{O}_{\mathcal{S}}|)$, where p is some polynomial and $|\mathcal{O}_{\mathcal{S}}|$ is the size of the agent state, and S_k is computable in polynomial time. Since $\mathcal{P}$ is associated with a regular agent, the number of steps in computing $S := \Gamma^{\ell}_{\mathcal{P}} \uparrow \omega$ is bounded by some (a priori known) constant b. Thus, it follows that the number of atoms in S is polynomial, and that S is computable in polynomial time. This shows that Step 1 is computable in polynomial time.

Step 3 can be done in polynomial time, since the assumptions imply that each body χ of an action constraint ac has only a polynomial number of solutions, which are computable in polynomial time, and matching the head of ac against S is polynomial. Step 4 can be done in polynomial time, since assembling and executing $\mathbf{conc}(\mathbf{Do}(S), \mathcal{O}_{\mathcal{S}})$ is polynomial and $\mathbf{Do}(S)$ is polynomial in the size of $\mathcal{O}_{\mathcal{S}}$, which means that the size of the resulting state $\mathcal{O}'_{\mathcal{S}}$ is polynomial in the size of $\mathcal{O}_{\mathcal{S}}$.

Finally, Step 5 is polynomial, since the body ψ of each integrity constraint $ic : \psi \Rightarrow \chi$ has a polynomial number of solutions θ, which are computable in polynomial time, and checking whether $\chi\theta$ is true in state $\mathcal{O}'_{\mathcal{S}}$ is polynomial in the size of $\mathcal{O}'_{\mathcal{S}}$ and thus also in the size of $\mathcal{O}_{\mathcal{S}}$. ∎

B Appendix: Feasible, Rational, and Reasonable Status Sets

This appendix provides in succinct form the definition of various concepts of status sets from [17], to which the reader is referred for more information.

Definition B.1 (status set) A *status set* is any set S of ground action status atoms over $\mathcal{S}$. 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 B.2 (operator $\mathbf{App}_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S)$) *Let $\mathcal{P}$ be an agent program and $\mathcal{O}_\mathcal{S}$ be an agent state. Then,*

$$\mathbf{App}_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S) = \{\, head(r\theta) \mid r \in \mathcal{P},\ R(r,\theta,S) \textit{ is true on } \mathcal{O}_\mathcal{S}\},$$

where the predicate $R(r,\theta,S)$ is true iff

1. *$r\theta$ is ground;*
2. *$B_{cc}(r\theta)$ is true in the agent state $\mathcal{O}_\mathcal{S}$;*
3. *$B^+_{as}(r\theta) \subseteq S$ and*
4. *$\neg.B^-_{as}(r\theta)\} \cap S = \emptyset$;*
5. *For every atom $Op\alpha \in B^+_{as}(r\theta) \cup Head(r\theta)$ where $Op \in \{\mathbf{P}, \mathbf{Do}, \mathbf{O}\}$, the action α is executable in $\mathcal{O}_\mathcal{S}$.*

Definition B.3 (feasible status set) Let $\mathcal{P}$ be an agent program and let $\mathcal{O}_\mathcal{S}$ be an agent state. Then, a status set S is a *feasible status set* for $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, if the following conditions hold:

($S1$) (closure under the program rules) $\mathbf{App}_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S) \subseteq S$;

($S2$) (deontic and action consistency) S satisfies the deontic consistency axioms $(D1)$–$(D3)$, and $S, \mathcal{O}_\mathcal{S} \models \mathcal{AC}$, i.e., S satisfies the action constraints with respect to $\mathcal{O}_\mathcal{S}$;

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

($S4$) (state consistency) $\mathcal{O}'_\mathcal{S} \models \mathcal{IC}$, where $\mathcal{O}'_\mathcal{S}$ is the state which results after executing $\mathbf{conc}(\mathbf{Do}(S), \mathcal{O}_\mathcal{S})$ in state $\mathcal{O}_\mathcal{S}$, i.e., $\mathcal{O}_\mathcal{S}$ satisfies the integrity constraints.

Definition B.4 (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 B.5 (reasonable status set) Let $\mathcal{P}$ be an agent program, let $\mathcal{O}_\mathcal{S}$ be an agent state, and let S be a status set.

1. If $\mathcal{P}$ is a positive agent program, then S is a *reasonable status set* for $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, iff S is a rational status set for $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$.
2. The reduct of $\mathcal{P}$ with respect to S and $\mathcal{O}_\mathcal{S}$, denoted by $red^S(\mathcal{P}, \mathcal{O}_\mathcal{S})$, is the program which is obtained from the ground instances of the rules in $\mathcal{P}$ over $\mathcal{O}_\mathcal{S}$ as follows.

(a) First, remove every rule r such that $B^-_{as}(r) \cap S \neq \emptyset$;

(b) Remove all atoms in $B^-_{as}(r)$ from the remaining rules.

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

References

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

[2] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 137–148, Montreal, Canada, June 1996.

[3] J. J. Alferes and L. M. Pereira. *Reasoning with Logic Programming*. Lecture Notes in AI, volume 1111, Springer, 1996.

[4] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Washington DC, 1988.

[5] Y. Arens, C. Y. Chee, C.-N. Hsu, and C. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent Cooperative Information Systems*, 2(2):127–158, 1993.

[6] Bayardo, R., et al. Infosleuth: Agent-based semantic integration of information in open and dynamic environments. In J. Peckham, editor, *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 195–206, Tucson, Arizona, 1997.

[7] K. A. Berman, J. S. Schlipf, and J. V. Franco. Computing the well-founded semantics faster. In A. Nerode, W. Marek, and M. Truszczynski, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Third International Conference*, LNCS 928, pages 113–126, Berlin, Germany, Springer, 1995.

[8] J. Boye and J. Maluszynski. Two aspects of directional types. In *Proceedings of the 12th International Conference on Logic Programming*, pages 747–761, Tokyo, Japan, 1995. MIT Press.

[9] S. Brass and J. Dix. Characterizations of the disjunctive stable semantics by partial evaluation. *The Journal of Logic Programming*, 32(3):207–228, 1997.

[10] S. Brass and J. Dix. Characterizations of the disjunctive well-founded semantics: confluent calculi and iterated GCWA. *Journal of Automated Reasoning*, 20(1):143–165, 1998.

[11] S. Brass and J. Dix. Semantics of (disjunctive) logic programs based on partial evaluation. *The Journal of Logic Programming*, 38(3):167–213, 1999.

[12] A. Brink, S. Marcus, and V. Subrahmanian. Heterogeneous multimedia reasoning. *IEEE Computer*, 28(9):33–39, 1995.

[13] K. S. Candan, B. Prabhakaran, and V. S. Subrahmanian. CHIMP: A framework for supporting multimedia document authoring and presentation. In *Proceedings of ACM Multimedia Conference*, pages 329–340, Boston, MA, November 1996.

[14] Chawathe, S., et al. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of the 10th Meeting of the Information Processing Society of Japan*, Tokyo, Japan, October 1994. Also available via anonymous FTP from host db.stanford.edu, file `/pub/chawathe/1994/tsimmis-overview.ps`.

[15] J. Dix, U. Furbach, and I. Niemela. Nonmonotonic reasoning: Towards efficient calculi and implementations. In A. Voronkov and A. Robinson, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999. to appear.

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

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

[18] P. W. Foltz and S. T. Dumais. Personalized information delivery: An analysis of filtering methods. *Communications of the ACM*, 35(12):51–60, 1992.

[19] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings Fifth International Conference and Symposium*, pages 1070–1080, Cambridge, Massachusetts, 1988. MIT Press.

[20] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[21] D. Goldberg, D. Nichols, B. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, December 1992.

[22] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. J. C. Meyer. Formal Semantics of an Abstract Agent Programming Language. In *International Workshop on Agent Theories, Architectures, and Languages (ATAL '97)*, LNCS 1365, pages 215–230, 1998. Springer.

[23] S. Ishizaki. Multiagent model of dynamic design: Visualization as an emergent behavior of active design agents. In M. Huhns and M. Singh, editors, *Readings in Agents*, pages 172–179. Morgan Kaufmann, 1997.

[24] R. Kowalski and F. Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):391–419, 1999.

[25] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, Germany, 1984, 1987.

[26] J. Lu, A. Nerode, and V. S. Subrahmanian. Hybrid knowledge bases. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):773–785, 1996.

[27] B. M. E. Moret and H. D. Shapiro. *Design and Efficiency*, volume I of *Algorithms from P to NP*. Benjamin/Cummings Publishing Company, Redwood City, CA, 1991.

[28] M. Paterson and M. Wegman. Linear Unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[29] D. Plaisted. Equational reasoning and term rewriting systems. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1, pages 274–364. Oxford: Clarendon Press, 1993.

[30] H. Rogers Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.

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

[32] S. J. Rosenschein and L. P. Kaelbling. A situated view of representation and control. *Artificial Intelligence*, 73:149–173, 1995.

[33] Y. Rouzaud and L. Nguyen-Phoung. Integrating modes and subtypes into a prolog type checker. In *Proceedings of the International Joint Conference/Symposium on Logic Programming*, pages 85–97. MIT Press, 1992.

[34] D. Rus, R. Gray, and D. Kotz. Transportable information agents. In M. Huhns and M. Singh, editors, *Readings in Agents*, pages 283–291. Morgan Kaufmann, 1997.

[35] J. Schafer, T. J. Rogers, and J. Marin. Networked visualization of heterogeneous US Army war reserves readiness data. In S. Jajodia, T. Ozsu, and A. Dogac, editors, *Advances in Multimedia Information Systems, 4th International Workshop, MIS'98*, LNCS 1508, pages 136–147, Istanbul, Turkey, 1998. Springer.

[36] B. Sheth and P. Maes. Evolving agents for personalized information filtering. In *Proceedings of the 9th Conference on Artificial Intelligence for Applications (CAIA'93)*, pages 345–352, Orlando, FL, March 1993. IEEE Computer Society Press.

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

[38] J.-D. Sta. Information filtering: A tool for communication between researchers. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems – Adjunct Proceedings*, Short Papers (Posters): Help and Information Retrieval, pages 177–178, 1993.

[39] J. D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.