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, I: Semantics

Thomas Eiter V.S. Subrahmanian George Pick

INFSYS RESEARCH REPORT 1843-98-02

DECEMBER 1998

Institut für Informationssysteme
Abtg. Wissensbasierte Systeme
Technische Universität Wien
Treitlstraße 3
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 1843-98-02, DECEMBER 1998

Heterogeneous Active Agents, I: Semantics

Thomas Eiter[1] V.S. Subrahmanian[2] George Pick[3]

Abstract. Over the years, many different agent programming languages have been proposed. In this paper, we propose a concept called Agent Programs using which, the way an agent should act in various situations can be declaratively specified by the creator of that agent. Agent Programs may be built on top of arbitrary pieces of software code and may be used to specify what an agent is obliged to do, what an agent may do, and what an agent may not do. In this paper, we define several successively more sophisticated and epistemically satisfying declarative semantics for agent programs. We further show that agent programs cleanly extend well understood semantics for logic programs, and thus are clearly linked to existing results on logic programming and nonmonotonic reasoning.

[1]Institut und Ludwig Wittgenstein Labor für Informationssysteme, Technische Universität Wien, Paniglgasse 16, 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]Department of Computer Science, University of Maryland, College Park, Maryland 20742. E-mail: george@cs.umd.edu

Acknowledgements: We wish to thank Alex Dekhtyar, Jürgen Dix, Sarit Kraus, Munindar Singh, Terrence Swift, and others for a very close reading of previous versions of this manuscript, and for the numerous detailed comments they made. These comments have significantly improved the quality of this paper. We have also benefited from discussions with Swati Allen, Piero Bonatti, Steve Choy, Phil Emmerman, Dana Nau, Anil Nerode, Dino Pedreschi, and Jose Salinas.
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, and by a DAAD grant. his work was initiated by R. Stewart.

This is Part I of a revised and extended version of the Technical Report UMCP-CSD CS-TR-3880, "Heterogeneous Active Agents", Computer Science Department, University of Maryland.

Copyright © 1999 by the authors

Contents

1 Introduction **1**

2 Software Code Access **4**
2.1 Code Calls and Code Call Atoms 5
2.2 Integrity Constraints 9

3 Agent Actions **10**
3.1 Action Base 10
3.2 Action Constraints 17
3.3 Agent Programs: Syntax 17

4 Why this Framework? **20**

5 Semantics for Agent Programs **21**
5.1 Feasible status sets 23
5.2 Rational status sets 27
5.2.1 Reading of rational status sets 31
5.3 Reasonable status sets 32
5.4 Violating obligations: weak rational status sets 35
5.4.1 Characterization of weak rational status sets 38
5.5 Expressing action constraints in an agent program 39
5.6 Preferred and Complete Status Sets 40
5.6.1 Preference 41
5.6.2 Complete Status Sets 42
5.7 Optimal Status Sets 43
5.8 Feasible status sets and models of logic programs 45
5.9 Rational status sets and minimal models of logic programs 46
5.10 Reasonable status sets and stable semantics 47
5.11 Discussion 48

6 Related Work **49**
6.1 Shoham's AGENT-0 49
6.2 BDI Framework 50
6.3 Hindriks' *et al.*'s Agent Programs 51
6.4 Deontic Logic Programming 52
6.5 Other approaches 54

7 Conclusions and Future Work **56**

References **58**

A Appendix: Complexity of S- and F-Concurrent Executability **64**

B Appendix: Proofs of Other Results **67**

C Appendix: Table of Notation Used in the Paper **71**

1 Introduction

Over the last few years, there has been intense work in the area of intelligent agents [49, 105]. Applications of such agent technology have ranged from intelligent news and mail filtering programs [62], to agents that monitor the state of the stock market and detect trends in stock prices, to intelligent web search agents [33], to the digital battlefield where agent technology closely monitors and merges information gathered from multiple heterogeneous information sources [4, 56, 57, 94, 103].

In the long run, a platform to support the creation and deployment of multiple software agents will need to interoperate with a wide variety of custom-made, as well as legacy software sources. Any definition *Def* of what it takes for a software package $\mathcal{S}$ (in any programming language) to be considered an agent program, must come accompanied with tools to augment, modify, or massage $\mathcal{S}$ into an agent according to the definition *Def*.

Figure 1 shows the architecture of our IMPACT System for the creation and deployment of multiple interacting agents. IMPACT is a joint project between the University of Maryland, Bar-Ilan University (Israel), the Technical University of Vienna (Austria), and the University of Koblenz (Germany). In IMPACT, an agent consists of two parts:

1. A body of software code (built in any programming language) that supports a well defined application programmer interface (either part of the code itself, or developed to augment the code). In general, we will assume that a piece of software $\mathcal{S}$ is represented by a pair $\mathcal{S} = (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S})$ where:

   - $\mathcal{T}_\mathcal{S}$ is the set of all data types manipulated by the software package $\mathcal{S}$. $\mathcal{T}_\mathcal{S}$ is assumed to be closed under sub-types, i.e. if τ is a subtype of a type in $\mathcal{T}_\mathcal{S}$, then τ must also be in $\mathcal{T}_\mathcal{S}$.
   - $\mathcal{F}_\mathcal{S}$ is the set of all pre-defined functions of the package $\mathcal{S}$ that are provided by the package's application programmer interface.

   In other words, in the strict sense of object systems, $\mathcal{S}$ is definable as a collection (or hierarchy) of object classes in any standard object data management language such as ODL [21]. Almost all existing servers used in real systems, as well as most commercial packages available on the market are instances of the above definition.

   For example, consider the well known Oracle DBMS. This may be viewed as a body of software code $\mathcal{S} = (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S})$ where:

   - $\mathcal{T}_\mathcal{S}$ consists of the following types: a set of attribute domains, tuples over different combinations of these attribute domains, and relations (sets of tuples) over different attribute domains.
   - $\mathcal{F}_\mathcal{S}$ consists of the classical relational operations: select, project, cartesian product, join, union, intersection, difference and aggregate operations, together with combinations of these.

   At any given point t in time, the *state of an agent* will refer to a set $\mathcal{O}_\mathcal{S}(t)$ of objects from the types $\mathcal{T}_\mathcal{S}$, managed by its internal software code. An agent may change its state by taking an action – either triggered internally or by processing a message received from another agent. However, one agent cannot directly change another agent's state, though it might do so indirectly by shipping the other agent a message issuing a change request. The precise definitions of messages and message management, as well as actions and action management, will be described in detail below.

2. A *semantic wrapper* that contains a wealth of semantic information. Such information includes, but is not restricted to the following:

   (a) A *service description* expressed in some tightly specified language. While a multiplicity of languages may be used for this purpose, in IMPACT [5] we have developed an HTML-like language for creating and manipulating service descriptions. This language has been characterized with a formal declarative semantics, as well as sound and complete algorithms for matching requests for services with an archive of service descriptions. Such matches are "correct" only w.r.t. an underlying similarity measure.

   (b) A *message manager* that (a) manages the data structures associated with an IMPACT agent's mailbox, and (b) specifies and implements policies on how commonalities between requests may be exploited to reduce the load on the agent.

   (c) An *action module* that will take as input, a newly read message (which will constitute an *event*), and use this to trigger zero, one, or many actions. For this purpose, the action module will require a specification of:

      i. (Action Base) The actions that the agent may take in principle, and the conditions that the agent state must satisfy for these actions to be executable, as well as the effects on the agent state of taking such actions;

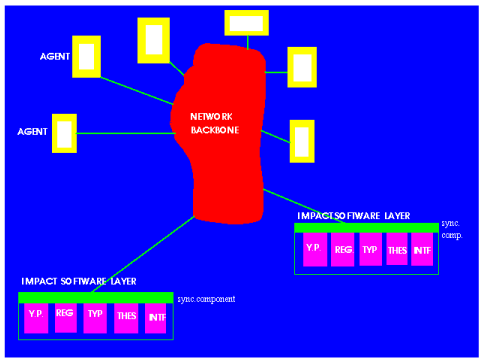
      ii. (Action Requirements) The conditions (on the agent state) under which the agent is either *obliged* or *forbidden* to take certain actions, as well as the conditions under which an agent is *permitted* (at its discretion) to take an action;

      iii. (Action Policy) The conditions on the agent state that determine how to choose which of several permissible actions should in fact be executed.

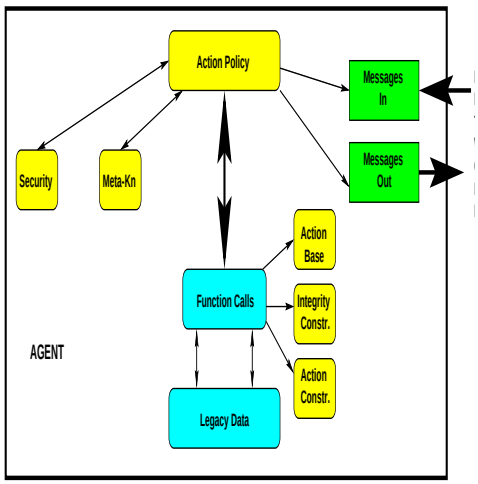
   (d) A *metaknowledge module* that provides the agent valuable information both about itself, as well as about other existing agents in the world. Such metaknowledge may include statistical information on the reliability of other agents, the speeds with which other agents provided certain services, and the financial charges (if any) levied for such transactions. It may also include self-knowledge – such self-knowledge may include statistics about its own performance, as well as analyses of operations on which it has performed well or badly. The metaknowledge module is described in a separate paper [30].

The IMPACT architecture contains a set of replicated, mirrored IMPACT servers that provide a variety of services. Such services include agent yellow-page location services (to find agents that provide a requested service), an agent ontology service, as well as type/thesaurus services. The locations of all mirrored replicas of the IMPACT servers are known to all agents – mirroring and replication guarantee that the system infrastructure will survive "downtimes" experienced by one or more servers. A synchronization layer guarantees that different IMPACT servers will reflect the same state, propagating changes at one server to other mirrored sites. Due to space concerns, the reader interested in details of the IMPACT architecture is referred to [5].

With this background in mind, we are now ready to go into the main aim of this paper – which is to design a theory and implementation of methods by which an agent may decide what actions it is obligated to take in a given state, what actions it is permitted to take (in a given state), and how it chooses which actions to in fact perform, given such a state of the world. Such choices are expressed by an *agent program* developed in a logical agent programming language that we introduce in this paper. We are not the first to



(a) IMPACT Architecture



(b) IMPACT Agent Architecture

propose agent programming paradigms – several others, notably [91, 48, 42], have done so before us. Our work builds upon these previous, pioneering efforts, in the following ways:

1. We will formally define the concept of an Agent Program that allows agents (of the sort described above) to be built on top of arbitrary software code with application programmer interfaces;
2. We will show that using such Agent Programs, we can access legacy software as well as custom built software;
3. We will provide several alternative declarative semantics specifying the meaning of such Agent Programs. In particular, this declarative semantics will specify what actions an agent will perform, given a currently prevailing agent state, and how the execution of these actions will modify the agent state.
4. We will establish relationships between some of these semantics and existing semantical characterizations of non-monotonic logic programs.

The concept of Agent Programs described here has led so far to the development of two applications — the first builds a simulation of a supply chain management example, facilitating interoperation between database agents and route planning and scheduling agents. This application involves specifically building agents on top of commercial software packages including Microsoft ACCESS and ESRI's MapObjects. The second application describes a multiagent solution to the "Controlled Flight into Terrain" (CFIT) Problem that is the single largest cause of airline fatalities in the world. Our CFIT multiagent application includes flight planning agents, terrain elevation agents, and GPS (Global Positioning System) agents. In this paper, we restrict to consider the following plain example.

Example: Tax Scenario. Consider a tax agency to determine which returns to audit. Tax agencies are usually required to follow some explicit rules in who to audit (so that tax officers cannot audit ex-spouses against whom they hold a grudge, or unfairly prosecute one or another racial/ethnic group, etc.).

[]A simple tax application may in fact have several agents. For the sake of simplicity, we will consider just one agent, which we shall call the Audit-Agent that determines which users should be audited. It may do so by monitoring two relations:

- The first relation, called `returns`, contains a relational representation of the returns filed by taxpayers.

- The second relation, called `employer_decs`, specifies the payments to various individuals that have been reported by employers.

The agent monitors discrepancies between the amounts reported by individual taxpayers, and amounts reported by *all* employers who have made payments to that person. Based on these discrepancies, it may either be obliged to take some audit actions, or forbidden to do so, or permitted to do so at its discretion. Based on the actions this agent takes, it may be forced to take other actions (such as notifying the taxpayer that he is required to explain his return, or to appear in person in tax court, etc.).

This scenario is a simplified scenario, that permits pedagogical clarity when it is used to illustrate the theories and definitions introduced in this paper. We will use this example repeatedly in this paper. □

The present paper is the first in a series of papers — Part II of this series will develop results on the computational cost of these alternative semantics, giving rise to a hierarchy of increasing complexity. It will also include algorithms for computing these alternative semantics.

The organization of Part I is as follows. Section 2 specifies how agent reasoning may be built on top of existing legacy software and how this may be used to define the concept of an agent state. Section 3 specifies the language within which agent actions are specified, and the language of agent programs which specifies the conditions governing an agent's behavior. Section 4 adresses why the framework we propose is useful. The subsequent Section 5 forms the main contribution of this paper, and describes the semantics of agent programs. In fact, Section 5 gives a set of successively more desirable semantics to agent programs. We discuss the advantages and disadvantages of these semantics, and shows the relationship between our semantics and well-known semantics in logic programming. As the relationship between logic program semantics and nonmonotonic reasoning semantics (for default logic, autoepistemic logic and truth maintenance systems) is well known, this section also shows the relationship between our semantics and classical nonmonotonic logic semantics. In Section 6 we review related work and compare our Agent Programs to various other systems for agent based programming. Section 7 closes this paper and gives some outlook to future work.

Note. As this paper is long and contains a fair amount of notation, Appendix C contains a table summarizing the notation, as a handy reference for the reader. Proofs of technical results have been moved to Appendices A and B, in order not to distract from the flow of reading.

2 Software Code Access

In this section, we focus on the "internal" data managed by the software code underlying an agent. As mentioned in the Introduction, we may characterize the code implementing an agent to be a pair $\mathcal{S} = (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S})$ where $\mathcal{T}_\mathcal{S}$ is the set of all data types provided by $\mathcal{S}$ and $\mathcal{F}$ is a set of predefined functions $\mathcal{S}$ which makes access to the data objects in the agent's state $\mathcal{O}_\mathcal{S}$ available to external processes.

This characterization of a piece of software code is a well accepted and widely used specification – for example, the Object Data Management Group's ODMG standard [21] and the CORBA framework are existing industry standards that are consistent with this specification.

2.1 Code Calls and Code Call Atoms

In this section, we introduce the reader to the important concept of a code call atom – this concept forms the basic syntactic object using which we may access multiple heterogeneous data sources. Before proceeding to this definition, we need to introduce some syntactic assumptions.

The content of Section 2.1 is not new work. It builds upon a previous effort called HERMES by one of the authors on heterogeneous data and software integration [1, 20, 61, 60, 63]. The reader familiar with that syntax may skip this section.

Suppose we consider a body $\mathcal{S} = (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S})$ of software code. Given any type $\tau \in \mathcal{T}_\mathcal{S}$, we will assume that there is a set $Var(\tau)$ of variable symbols ranging over τ. If $\mathtt{X} \in Var(\tau)$ is such a variable symbol, and if τ is a complex record type having fields $\mathtt{f_1}, \ldots, \mathtt{f_n}$, then we require that $\mathtt{X.f_i}$ be a variable of type τ_i where τ_i is the type of field $\mathtt{f_i}$. In the same vein, if $\mathtt{f_i}$ itself has a sub-field g of type γ, then $\mathtt{X.f_i.g}$ is a variable of type γ, and so on. In such a case, we will call X a *root-variable*, and the variables $\mathtt{X.f_i}$, $\mathtt{X.f_i.g}$, etc. *path-variables*. For any path variable Y of the form X.path, where X is a root variable, we refer to X as the root of Y, denoted by $root(\mathtt{Y})$; for technical convenience, $root(\mathtt{X})$, where X is a root variable, refers to itself.

An assignment of objects to the variables is a set of equations of the form $\mathtt{V_1} = \mathtt{o_1}, \ldots, \mathtt{V_k} = \mathtt{o_k}$ where the $\mathtt{V_i}$'s are variables (root or path) and the $\mathtt{o_i}$'s are objects – such an assignment is *legal* if the types of objects and corresponding variables match.

Definition 2.1 (code call) Suppose $\mathcal{S} = (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S})$ is some software code and $\mathtt{f} \in \mathcal{F}$ is a predefined 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 f. Then $\mathcal{S} : \mathtt{f}(\mathtt{d_1}, \ldots, \mathtt{d_n})$ is called a *code call*. A code call is *ground*, if all the $\mathtt{d_i}$'s are objects. □

In general, as we will see later, code calls are executable when they are ground. Thus, non-ground code calls must be "instantiated" prior to attempts to execute them.

In general, each function $\mathtt{f} \in \mathcal{F}$ has a *signature*, specifying the types of inputs it takes, and the types of outputs it returns. Here are some examples of code calls that we have implemented:

- oracle : select(emp.rel, salary, >, 150000).
  Consider a domain called oracle representing the Oracle Universal Server. One of the relations in such a database may be called emp.rel. The above code call executes a *select* operation on the emp.rel table, and returns as output, the set of all tuples in emp.rel whose salary field is over 150,000 (dollars).

- face : match(mugshotdb, queryface).
  Consider a domain called face implementing a face recognition program. This program may manage a mugshot archive called mugshotdb of individuals whose identities are known. The match function takes a picture of someone whose identity is to be determined, and matches it against the mugshot database, returning a ranked set of pairs (File,Name), of faces and associated names that match the query face.

- terrain : planroute(map1, 97, 97, 102, 103).
  Consider a domain called terrain representing a terrain reasoning system. map1 may be one of several maps in this system. The function planroute plans an optimal route (according to some

trafficability criteria we will not go into here) from a given origin to a given destination. The above code call asks the terrain reasoner to plan an optimal route from the point $(97, 97)$ on `map1` to the point $(102, 103)$ on `map1`.

Assumption. We will assume that the output signature of any code call is a set. There is no loss of generality in making this assumption – if a function does not return a set, but rather returns an atomic value, then that value can be coerced into a set anyway – by treating the value as shorthand for the singleton set containing just the value.

Definition 2.2 (code call atom) If `cc` is a code call, and `X` is either a variable symbol, or an object of the output type of `cc`, then $\texttt{in}(\texttt{X}, \texttt{cc})$ is a *code call atom*. □

Code call atoms, when evaluated, return boolean values (i.e. they may be thought of as special types of logical atoms [90]). Intuitively, a code call atom succeeds just in case `X` is in the result set returned by `cc` (when `X` is an object), or when `X` can be made to point to one of the objects returned by executing the code call. Let us return to the code calls we introduced earlier, and see examples of some code call atoms.

- $\texttt{in}(\texttt{X}, \texttt{oracle} : \texttt{select}(\texttt{emp.rel}, \texttt{salary}, >, 150000))$.
  Here, this code call atom would succeed, instantiating `X` to any single tuple in relation `emp` that has a salary field of over 150,000.
- $\texttt{in}(\texttt{X}, \texttt{face} : \texttt{match}(\texttt{mugshotdb}, \texttt{queryface}))$.
  This code call atom would succeed, instantiating `X` to any record R (having a `file` and `name` fields) whose $R.\texttt{file}$ image matches the `queryface` image as determined by the image processing code implementing the `match` operation.
- $\texttt{in}(\texttt{X}, \texttt{terrain} : \texttt{planroute}(\texttt{map1}, 97, 97, 102, 103))$.
  This code call atom would succeed, instantiating `X` to some optimal route (as deemed by the route planner code) between points (97,97) and (102,103).

Definition 2.3 (code call condition) A *code call condition* is defined as follows:

1. Every code call atom is a code call condition.
2. If $\texttt{s}, \texttt{t}$ are either variables or objects, then $\texttt{s} = \texttt{t}$ is a code call condition.
3. If $\texttt{s}, \texttt{t}$ are either integers/real valued objects, or are variables over the integers/reals, then $\texttt{s} < \texttt{t}$, $\texttt{s} > \texttt{t}$, $\texttt{s} \geq \texttt{t}$, $\texttt{s} \leq \texttt{t}$ 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:

```
in(X,oracle:select(emp.rel,salary,>,150000))&          (1)
in(Y,face:findpictureof(mugshotdb,X.name))
```

This condition may be viewed as a query requesting that we find all X, 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.

One aspect to keep in mind about code calls is that while code call syntax allows variables to appear in a code call, it is usually impossible to evaluate a code call when it has uninstantiated variables.

Most commercial packages' application programmer interface functions assume that when the functions are invoked, all arguments expected by that function will be passed, else an error will result. However, it is possible to write code call conditions that expect functions defined in external packages to work even when some arguments are not instantiated. The concept of safety defined below ensures that this cannot happen.

Definition 2.4 (safe code call) A code call $\mathcal{S} : \mathtt{f}(\mathtt{d}_1, \ldots, \mathtt{d}_n)$ is *safe* if and only if each $\mathtt{d}_i$ is ground. A code call condition $\chi_1 \,\&\, \ldots \,\&\, \chi_n$, $n \geq 1$, is *safe* if and only if there exists a permutation π of $\chi_1, \ldots, \chi_n$ such that for every $i = 1, \ldots, n$ the following holds:

1. If $\chi_{\pi(i)}$ is a comparison $\mathtt{s}_1 \; op \; \mathtt{t}_2$, then
   - 1.1 at least one of $\mathtt{s}_1, \mathtt{s}_2$ is a constant or a variable X such that $root(\mathtt{X})$ belongs to $RV_\pi(i) = \{root(\mathtt{Y}) \mid \exists j < i$ s.t. Y occurs in $\chi_{\pi(j)}\}$;
   - 1.2 if $\mathtt{s}_i$ is neither a constant nor a variable X such that $root(\mathtt{X}) \in RV_\pi(i)$, then $\mathtt{s}_i$ is a root variable.
2. If $\chi_{\pi(i)}$ is a code call atom of the form $\mathtt{in}(\mathtt{X}_{\pi(i)}, \mathtt{cc}_{\pi(i)})$, then the root of each variable Y occurring in $\mathtt{cc}_{\pi(i)}$ belongs to $RV_\pi(i)$, and either $\mathtt{X}_{\pi(i)}$ is a root variable, or $root(\mathtt{X}_{\pi(i)})$ is from $RV_\pi(i)$.

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

For example, the code call condition $\mathtt{in}(\mathtt{X}, \mathcal{S}_1 : \mathtt{f}_1(\mathtt{a}, \mathtt{Y}))$ 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 $\mathtt{in}(\mathtt{X}, \mathcal{S}_1 : \mathtt{f}_1(\mathtt{a}, \mathtt{Y})) \,\&\, \mathtt{in}(\mathtt{Y}, \mathcal{S}_2 : \mathtt{f}_2(\mathtt{b}))$ is safe, because we can execute the code call atom $\mathtt{in}(\mathtt{Y}, \mathcal{S}_2 : \mathtt{f}_2(\mathtt{b}))$ to obtain a ground binding θ for Y and then execute $\mathtt{in}(\mathtt{X}, \mathcal{S}_1 : \mathtt{f}_1(\mathtt{a}, \mathtt{Y}\theta))$ Similarly, the code call (1) described earlier in this section is safe. However, the code call

```
in(X,oracle:select(emp.rel,salary,>,150000))&
in(Y,face:findpictureof(mugshotdb,Z.name))
```

is not safe. The reason is that if the ordering of code call atoms above is used, then the face database is looking for an instantiated argument, Z.name, which it does not find. The reader can easily ascertain that reordering the literals in the example does not establish the safety property either.

As we will see later when defining actions, given a code call condition χ, it may be useful to talk about whether χ is safe if we assume that certain variables in χ are instantiated. Consider the code call condition $\mathtt{in}(\mathtt{X}, \mathcal{S}_1 : \mathtt{f}_1(\mathtt{a}, \mathtt{Y}))$ given above. If we know that Y will be bound to a constant (i.e., an object) (but do not know what this contant is), then we can certainly say that this code call condition is safe *modulo* the assumption that Y is bound. The following definition captures this.

Definition 2.5 (safety modulo variables) *Suppose* $\mathbf{X}$ *is a set of variables and* χ *is a code call condition. Let* χ' *be the result of replacing all occurrences of variables in* $\mathbf{X}$ *by objects from an arbitrary legal assignment to them. Then,* χ *is said to be* safe modulo the variables in $\mathbf{X}$ *if and only if the code call condition* χ' *is safe.*

We are now ready to define what constitutes a solution of a code call condition.

Definition 2.6 (code call solution) Suppose χ is a code call condition involving the variables $\vec{X}$, and $\mathcal{S} = (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S})$ is some software code. A *solution* of χ w.r.t. $\mathcal{T}_\mathcal{S}$ in a state $\mathcal{O}_\mathcal{S}$ is a legal assignment of objects o to the variables X in $\vec{X}$, written as a compound equation $\vec{X} = \vec{o}$, such that the application of the assignment makes χ true in state $\mathcal{O}_\mathcal{S}$.

We denote by $Sol(\chi)_{\mathcal{T}_\mathcal{S},\mathcal{O}_\mathcal{S}}$ (omitting subscripts $\mathcal{O}_\mathcal{S}$ and $\mathcal{T}_\mathcal{S}$ when clear from the context), the set of all solutions of the code call condition χ in state $\mathcal{O}_\mathcal{S}$, and by $\mathcal{O}_Sol(\chi)_{\mathcal{T}_\mathcal{S},\mathcal{O}_\mathcal{S}}$ (where subscripts are occasionally omitted) the set of all objects appearing in $Sol(\chi)_{\mathcal{T}_\mathcal{S},\mathcal{O}_\mathcal{S}}$ □

For example, consider the Oracle/Face database code call discussed earlier. A valid solution to this code call may be the assignment

$$\texttt{X} = \texttt{John Smith}, \texttt{Y} = \texttt{john_smith.gif}.$$

We are now ready to introduce an important assumption we make in our paper. As the reader surely knows, most legacy programs that manipulate a certain data structure have existing code to insert and delete objects from that data structure. This is certainly true of most commercial relational DBMSs, geographic information systems (e.g. ArcInfo, ArcView), spatial databases (quadtrees, R-trees), face databases (e.g. Informix face data blade), scheduling systems (e.g. Microsoft Schedule), etc.

Assumption. Throughout this paper, we assume that the set $\mathcal{F}_\mathcal{S}$ associated with a software code package $\mathcal{S}$ contains two functions described below:

- A function $\mathsf{ins}_\mathcal{S}$, which takes as input a set of objects $\mathcal{O}$ manipulated by $\mathcal{S}$, and a state $\mathcal{O}_\mathcal{S}$, and returns a new state $\mathcal{O}'_\mathcal{S} = \mathsf{ins}_\mathcal{S}(\mathcal{O}, \mathcal{O}_\mathcal{S})$ which accomplishes the insertion of the objects in $\mathcal{O}$ into $\mathcal{O}_\mathcal{S}$, i.e. $\mathsf{ins}_\mathcal{S}$ is an insertion routine.
- A function $\mathsf{del}_\mathcal{S}$, which takes as input a set of objects $\mathcal{O}$ manipulated by $\mathcal{S}$ and a state $\mathcal{O}_\mathcal{S}$, and returns a new set of objects $\mathcal{O}'_\mathcal{S} = \mathsf{del}_\mathcal{S}(\mathcal{O}, \mathcal{O}_\mathcal{S})$ which describes the deletion of the objects in $\mathcal{O}$ from $\mathcal{O}_\mathcal{S}$, i.e. $\mathsf{del}_\mathcal{S}$ is a deletion routine.

In the above two functions, it is possible to specify the first argument, $\mathcal{O}$, through a code-call atom or a code-call condition involving a single variable. Intuitively, suppose we execute the function, $\mathsf{ins}_{\texttt{quadtree}}(\chi[\texttt{X}])$ where $\chi[\texttt{X}]$ is a code call involving the (sole) free variable X. This may be interpreted as the statement: "Insert, using a quadtree insertion routine, all objects o such that $\chi[\texttt{X}]$ is true w.r.t. the current agent state when $\texttt{X} = o$." In such a case, the code call condition, χ is used to identify the objects to be inserted, and the $\mathsf{ins}_{\texttt{quadtree}}$ function specifies the insertion routine to be used. Assuming the existence of such insertion and deletion routines is very reasonable – almost all implementations of data structures in computer science include insertion and deletion routines !

As a single agent program may manage multiple data types $\tau_1, \ldots, \tau_n$, each with its own insertion routine, $\mathsf{ins}_{\tau_1}, \ldots, \mathsf{ins}_{\tau_n}$, respectively, it is often more convenient to associate with any agent a, an insertion routine, ins_a, that exhibits the following behavior: given either a set $\mathcal{O}$ of objects (or a code call condition $\chi[\texttt{X}]$ of the above type), $\mathsf{ins}_a(\chi[\texttt{X}], \mathcal{O}_\mathcal{S})$ is a generic *method* that selects which of the insertion routines ins_{τ_i}, associated with the different data structures, should be invoked in order to accomplish the desired insertion. A similar comment applies to deletion as well. Throughout the rest of this paper, we will assume that an insertion function ins_a and a deletion function del_a may be associated with any agent a in this way. Where a is clear from context, we will drop the subscript.

At this point, we have briefly shown how the mechanism of code-calls, and code-call atoms, provides a unified syntax within which different software packages and databases may be accessed through their application programmer interfaces. All the above code call mechanisms have been implemented by us.

Code calls, and code call atoms, form the basic theoretical mechanism through which an agent may access its internal code. In addition, using the code call mechanism, an agent A might send a request to agent B, with full assurance that agent B will be able to execute what is being requested by agent A. The service description layer of the IMPACT architecture will include descriptions of the code-calls provided by each agent – this will also be included in the Yellow Pages Server contained as part of the IMPACT Server. These sources can be used by agent A to structure its code call message to agent B. We will not go into the description of the service description component of IMPACT here – that is done in a companion paper [5].

2.2 Integrity Constraints

In addition to code-calls, each agent also has an associated set of *Integrity Constraints*. Agent integrity constraints specify properties that states of the agent must satisfy. For example, if we have an Oracle agent maintaining an employee database, we may have an integrity constraint of the form:

$$\texttt{in}(\texttt{X}, \texttt{oracle} : \texttt{select}(\texttt{emp.rel}, \texttt{salary}, >, \texttt{100000})) \Rightarrow \texttt{X.grade} \geq \texttt{6}.$$

This integrity constraint on the Oracle software states requires that the `emp.rel` relation must always ensure that individuals with salaries over 100K are at salary grade 6 or higher.

Similarly, consider an agent whose internal state is determined not just by one package, but by a hybrid of two packages – a face recognition system and an image processor. Here, we may want to use an integrity constraint which states that:

```
in(X,face:match(mugshotdb,queryface))&
in(Y,oracle:select(convicts.rel,name,=,X.name)&
=(X.sex,male)
  ⇒
in(X,face:match(mugshotdb_male,queryface)).
```

This constraint says that if `X` is returned by matching a query face using a face recognition program, and we know that the person shown in `X` is a male convict (using a relational database), then it should be the case that `X` is also returned by executing the match on just male mugshots.

Definition 2.7 (integrity constraint) An *integrity constraint* is an expression of the form

$$\psi \quad \Rightarrow \quad \chi_a$$

where ψ is a safe code call condition, and χ_a is an atomic code call condition such that every root variable in χ_a occurs in ψ. □

Note that the safety requirement on the precondition of an integrity constraint guarantees a mechanism to evaluate the precondition of an integrity constraint whose head is grounded.

Definition 2.8 (integrity constraint satisfaction) A state $\mathcal{O}_\mathcal{S}$ satisfies an integrity constraint $\mathcal{IC}$ of the form $\psi \Rightarrow \chi_a$, denoted $\mathcal{O}_\mathcal{S} \models IC$, if for every legal assignment of objects from $\mathcal{O}_\mathcal{S}$ to the variables in IC, either ψ is false or χ_a is true.

Let $\mathcal{IC}$ be a (finite) collection of integrity constraints, and let $\mathcal{O}_\mathcal{S}$ be an agent state. We say that $\mathcal{O}_\mathcal{S}$ satisfies $\mathcal{IC}$, denoted $\mathcal{O}_\mathcal{S} \models \mathcal{IC}$, if $\mathcal{O}_\mathcal{S}$ satisfies every constraint $IC \in \mathcal{IC}$. □

3 Agent Actions

Every agent's actions are completely determined by three parameters that the individual creating the agent must specify:

- An "Action Base" specifying a set of actions that the agent can execute (under the right conditions),
- A set of "Action Constraints" that specify, for example, mutual exclusion between actions, etc.
- An "Agent Program" that determines which of the (instances of) actions in the agent base the agent is obligated, permitted, or forbidden to execute, together with a mechanism to actually determine what actions will be taken. Actions are triggered by events. For example, reading a message may be an event that triggers one or more actions. A clock-event (e.g. the clock reaching 0800 hours) may be another event that triggers another action.

In this section, we will introduce the concepts of an Action Base, Action Constraint, and Action Program, and discuss how they work together.

3.1 Action Base

In this section, we will introduce the concept of an action and describe how the effects of actions are implemented. In most work in AI [73, 41, 82] and logical approaches to action [12], it is assumed that states are sets of ground logical atoms. In the fertile area of active databases, it is assumed that states reflect the content of a relational database. However, neither of these two approaches is adequate for our purpose because the state of an agent which uses the software code $\mathcal{S} = (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S})$ is described by the set $\mathcal{O}_\mathcal{S}$. The data objects in $\mathcal{O}_\mathcal{S}$ *could* be logical atoms (as is assumed in most AI settings), or they *could* be relational tuples (as is assumed in active databases), but in all likelihood, the objects manipulated by $\mathcal{S}$ are much more complex, structured data types.

Definition 3.1 (action; action atom) An *action* α consists of five components:

- A name, usually written $\alpha(\mathtt{X}_1, \ldots, \mathtt{X}_n)$, where the $\mathtt{X}_i$'s are root variables;
- A schema, usually written as $(\tau_1, \ldots, \tau_n)$, of types. Intuitively, this says that the variable $\mathtt{X}_i$ must be of type τ_i, for all $1 \leq i \leq n$.
- a code-call condition χ, called the *precondition* of the action, denoted by $Pre(\alpha)$;
- a set $Add(\alpha)$ of code-call conditions;
- a set $Del(\alpha)$ of code-call conditions.

The precondition $Pre(\alpha)$ must be safe modulo the variables $\mathtt{X}_1$,...,$\mathtt{X}_n$. Furthermore, every code-call condition χ in $Add(\alpha) \cup Del(\alpha)$ must be safe modulo the union of $\mathtt{X}_1$,...,$\mathtt{X}_n$ and the root variables $\mathtt{Y}_1, \ldots \mathtt{Y}_m$ occurring in $Pre(\alpha)$.

An *action atom* is a formula $\alpha(t_1, \ldots, t_n)$, where t_i is a term, i.e., an object or a variable, of type τ_i, for all $i = 1, \ldots, n$. □

Let us now consider some examples of actions and their associated descriptions in specific domains.

Example 3.1 (Simple Tax Audit Agent) Let us return to the tax agent scenario described in Section 1. Suppose that the agent has some action

`run_audit(Person)`.

This action might be executed by the agent if she runs an audit on a person. This action maybe specified as follows:

- Name: `run_audit(Person)`
- Schema: `(Person)`, where `Person` is an object of type person (for simplicity, identified with its name)
- Pre:

  ```
  in(T,taxdb:select(returns,name,=,Person))
  ```

  A record `T` for the person must be filed in relation `returns`.
- Add:

  ```
  X.Name = T.Name &
  X.Result = taxdb:eval_audit(T.Name) &
  X.Date = system:today()
  ```

  Here, the type of `X` is the domain of another relation `audit` in the tax database, which stores an entry for each audit on a person which has been run, together with the result – which is determined by a procedure `eval_audit` – and the date, which is returned by a system function. Adding `X` includes this tuple in the relation `audit`.[1]
- Delete:

  In this case, there is nothing to delete. □

In our framework, we assume that any explicit state change initiated by an agent is an action. For example, sending messages and reading messages is an action. Similarly, making an update to an internal data structure is an action. Performing a computation on the internal data structures of an agent is also an action (as the result of the computation in most cases is returned by modifying the agent's state).

Example 3.2 (Message Box) Throughout this paper, we will assume that each agent's associated software code includes a special type called `msgbox` (short for message box). Each agent has its own personal instance of this data type, reflecting its personal mailbox[2].

The message box data structure is a.buffer When an agent receives a message or when it instructs that a message be sent, then the buffer gets updated. When the agent reads the message, or when a message is physically sent off, then the buffer gets flushed. We assume the existence of an operating-systems level

[1] Observe that, in general, membership of an object in a class could be made explicit through a special "instance_of" attribute – which is common in object-oriented systems – that is specified in the condition. E.g., we could include then `X.instance_of = 'audit'` in the above code call condition to stress membership of `X` in the relation `audit`.

[2] Unix's `mail` daemons are implemented this way.

messaging protocol (e.g. sockets or TCP/IP [104]) that can fill in (with incoming messages) or flush (when a message is physically sent off) this buffer.

We will assume that the agent has the following functions that are integral in managing this message box. Note that over the years, we expect a wide variety of messaging languages to be developed (examples of such messaging languages include KQML [57] at a high level, and remote procedure calls at a much lower level). In order to provide maximal flexibility, we will merely specify below, the "core" interface functions available on the `msgbox` type. Note that this set of functions may be augmented by the addition of other functions on an agent by agent basis.

- `SendMessage(Src,Dest,Msg)`: This causes a quintuple $(o, Src, Dest, Msg, Time)$ to be placed in `msgbox`. The o signifies an outgoing message. When `SendMessage(Src, Dest,Msg)` is executed, the state of `msgbox` changes by the insertion of the above quintuple denoting the sending of a message from the source (`Src`) agent to a given destination agent (`Dest`) involving the message body `Msg`; `Time` denotes the time at which the message was sent.
- `GetMessage(Src)`: This causes a collection of quintuples $(i, Src, agent\text{-}id, Msg, Time)$ to be read from `msgbox`. The i signifies an incoming message. Note that all messages from the given source to the agent `agent-id` whose message box is being examined, are returned by this operation. `Time` denotes the time at which the message was received.
- `TimedGetMessage(op,Val)`: This causes the collection of all quintuples $tup = (i, Src, agent\text{-}id, Msg, Time)$ to be read from `msgbox`, where $tup.\texttt{Time}\texttt{op}Val$ holds; `op` is required to be any of the standard comparison operators $\leq, <, \geq, >, =$. □

Agents interact with the external world through the `msgbox` code – in particular, external agents may update agent A's `msgbox`, thus introducing new objects to agent A's state, and triggering state changes which are not triggered by agent A.

Throughout this paper, we will assume that every agent has as part of its state, the specialized type `msgbox` defined here, together with the code calls on this type defined here.

Example 3.3 (Java Agents) In today's world, the word "agent" is often considered (in certain non-AI communities) to be synonymous with the Java applets. What is unique about an applet is that it is *mobile*. A Java applet hosted on machine H can "move" across the network to a target machine T, and execute its operations there. The actions taken by a Java agent `agent-id`, may be captured within our framework as follows.

1. **Name:** `do(Op,Host,Target,ArgumentList)`

   which says "Perform the operation `Op` on the list `ArgumentList` of arguments located at the `Target` address by moving there from the `Host` address."

2. **Precondition:**

   $$\texttt{in}(\texttt{Host}, \texttt{java} : \texttt{location}(\texttt{agent-id})) \,\&$$
   $$\texttt{in}(\text{“}\texttt{ok}\text{”}, \texttt{security} : \texttt{authorize}(\texttt{agent-id}, \texttt{Op}, \texttt{Target}, \texttt{ArgumentList}).$$

   This says that the Java implementation recognizes that the agent in question is currently at the `Host` machine and that the security system of the remote machine authorizes the agent to download itself on the target and execute its action.

3. **Add/Delete-Set:** This consists of whatever insertions and deletions must be done to data in the `Host`'s workspace. □

We are now ready to define an action base. Intuitively, each agent has an associated action base, consisting of actions that it can perform on its object state.

Definition 3.2 (action base) An *action base*, $\mathcal{AB}$, is any finite collection of actions. □

The above definition states what an action is, but allows the possibility that an action should simultaneously add and and delete some object. Classical AI systems like STRIPS also allow this to happen in the syntax (e.g. in a STRIPS rule, an atom could occur in both the Add and $Delete$ lists) and handle potential problems of this sort by first doing deletions and then insertions. We mimic this as well. Later, in Section 3.1, we will show that it is possible using a simple syntax to be introduced in Section 3.1 to forbid execution of any action whose add and delete sets have an overlap.

A difference between our work and classical AI systems, is that in the latter, change is modeled solely as the insertion and deletion of logical atoms from a state which is a set of logical atoms [73]. In the real world, however, states are usually instances of fairly complex data structures. Therefore, in our case, changes affect components of objects in $\mathcal{O}_\mathcal{S}$ where $\mathcal{S}$ is the software code manipulated by the agent in question. The following definition shows what it means to execute an action in a given state.

Definition 3.3 ((θ,γ)-executability) Let $\alpha(\vec{X})$ be an action, and let $\mathcal{S} = (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S})$ be an underlying software code accessible to the agent. A ground instance $\alpha(\vec{X}\theta)$ of $\alpha(\vec{X})$ is said to be *executable* in state $\mathcal{O}_\mathcal{S}$, if and only if there exists a solution γ of $Pre(\alpha(\vec{X}\theta))$ w.r.t. $\mathcal{O}_\mathcal{S}$. In this case, $\alpha(\vec{X})$ is said to be (θ,γ)-*executable* in state $\mathcal{O}_\mathcal{S}$, and $(\alpha(\vec{X}),\theta,\gamma)$ is a feasible *execution triple* for $\mathcal{O}_\mathcal{S}$. By $FE(\alpha(\vec{X}),\mathcal{O}_\mathcal{S})$ we denote the set of all pairs (θ,γ) such that $(\alpha(\vec{X}),\theta,\gamma)$ is a feasible execution triple in state $\mathcal{O}_\mathcal{S}$. □

Intuitively, in $\alpha(\vec{X})$, the substitution θ causes all variables in $\vec{X}$ to be grounded. However, it is entirely possible that the precondition of α has occurrences of other free variables not in $\vec{X}$. Appropriate ground values for these variables are given by solutions of $Pre(\alpha(\vec{X}\theta))$ with respect to the current state $\mathcal{O}_\mathcal{S}$. These variables can be viewed as "hidden parameters" in the action specification, whose value is of less interest for an action to be executed.

The following definition tells us what the result of (θ,γ)-execution is.

Definition 3.4 (action execution) Suppose $(\alpha(\vec{X}),\theta,\gamma)$ is a feasible execution triple in state $\mathcal{O}_\mathcal{S}$. Then the *result* of executing $\alpha(\vec{X})$ w.r.t. (θ,γ) is given by the state

$$apply((\alpha(\vec{X}),\theta,\gamma),\mathcal{O}_\mathcal{S}) = \mathsf{ins}(\mathcal{O}_{add}, \mathsf{del}(\mathcal{O}_{\mathsf{del}},\mathcal{O}_\mathcal{S})),$$

where $\mathcal{O}_{add} = \mathcal{O}_Sol(Add(\alpha(\vec{X}\theta\gamma)))$ and $\mathcal{O}_{del} = \mathcal{O}_Sol(Del(\alpha(\vec{X}\theta\gamma)))$; i.e., the state which results if first all objects in solutions of call conditions from $Del(\alpha(\vec{X}\theta\gamma))$ on $\mathcal{O}_\mathcal{S}$ are removed, and then all objects in solutions of call conditions from $Add(\alpha(\vec{X}\theta\gamma))$ on $\mathcal{O}_\mathcal{S}$ are inserted. □

We reiterate here that ins refers to the insertion routine associated with the agent whose actions are being discussed above. The ins function may in turn call specific insertion routines associated with each data structure manipulated by the agent in question.

Furthermore, observe that in the above definition, we do not pay attention to integrity constraints. Possible violation of such constraints due to the execution of an action will be handled later in the definition of the semantics of agent programs that we are going to develop, and will of course prevent that integrity-violating actions from being executed on the current agent state.

While we have stated above what it means to execute a feasible execution triple on an agent state $\mathcal{O}_\mathcal{S}$, there remains the possibility that many different execution triples are feasible on a given state, which may stem from different actions $\alpha(\vec{X})$ and $\alpha(\vec{X}')$, or even from the same grounded action $\alpha(\vec{X}\theta)$. Thus, in general, we have a set AS of feasible execution triples which should be executed. It is natural to assume that AS is the set of all feasible execution triples. However, it is perfectly imaginable that only a subset of all feasible execution triples should be executed. E.g., if only one from many solutions γ is selected –in a well-defined way– such that $(\alpha(\vec{X}), \theta, \gamma)$ is feasible, for a grounded action $\alpha(\vec{X})\theta$; we do not discuss this any further here.

Suppose then we wish to simultaneously execute a set of (not necessarily all) feasible execution triples AS. There are many ways to define this. We present three possible definitions below, and assess their merits and disadvantages.

Definition 3.5 (weakly-concurrent execution) Suppose AS is a set of feasible execution triples in the agent state $\mathcal{O}_\mathcal{S}$. The *weakly-concurrent execution of* AS in $\mathcal{O}_\mathcal{S}$, is defined to be the agent state

$$apply(AS, \mathcal{O}_\mathcal{S}) = \mathsf{ins}(\mathcal{O}_{add}, \mathsf{del}(\mathcal{O}_{\mathsf{del}}, \mathcal{O}_\mathcal{S})),$$

where

$$\begin{aligned} \mathcal{O}_{add} &= \bigcup_{(\alpha(\vec{X}),\theta,\gamma)\in AS} \mathcal{O}_Sol(Add(\alpha(\vec{X}\theta\gamma))), \\ \mathcal{O}_{del} &= \bigcup_{(\alpha(\vec{X}),\theta,\gamma)\in AS} \mathcal{O}_Sol(Del(\alpha(\vec{X}\theta\gamma))). \end{aligned}$$

For any set A of actions, the execution of A on $\mathcal{O}_\mathcal{S}$ is the execution of the set

$$AS = \{(\alpha(\vec{X}), \theta, \gamma) \mid \alpha(\vec{t}) \in A,\ \alpha(\vec{X}\theta) = \alpha(\vec{t}\theta) \text{ is ground, and } (\theta, \gamma) \in FE(\alpha(\vec{X}))\}$$

of all feasible execution triples stemming from some grounded action in A, and $apply(A, \mathcal{O}_\mathcal{S})$ denotes the resulting state. □

As the reader will note, this is a definition which does everything in parallel – it first does all deletions and then all insertions. While weakly-concurrent executions work just fine when the set A of actions involves no "conflicts", they are problematic when the actions in A compete for resources.

Example 3.4 (Grid) Suppose we have a two-dimensional grid and an object is placed at location $(5, 7)$ on the grid, and suppose two actions `go-right` and `go-left` are both possible. If we define `go-right(X,Y)` and `go-left(X,Y)` in the obvious way, then on the execution of both actions `go-left(5,7)` and `go-right(5,7)`, the final result says that the object is at both locations $(6, 7)$ and $(4, 7)$ which is clearly absurd! □

Thus, *before* attempting to perform a weakly-concurrent execution of a set of actions, we must ensure that the set of actions satisfy some consistency criteria, otherwise there is a danger of doing something absurd.

The following definition, called *sequential-concurrent execution*, (or S-concurrent execution for short) removes some, but not all of these problems, and in turn, introduces some new problem. In effect, it says that a set of actions is S-concurrently executable iff there is some way of ordering the actions so that they can be sequentially executed.

Definition 3.6 (sequential-concurrent executability) Suppose that we have a set

$$AS = \{t_i \mid t_i = \alpha_i(\vec{X}_i, \theta_i, \gamma_i)), 1 \leq i \leq n\}$$

of feasible execution triples and an agent state $\mathcal{O}_\mathcal{S}$. Then, AS is said to be *S-concurrently executable* in state $\mathcal{O}_\mathcal{S}$, if and only if there exists a permutation π of AS and a sequence of states $\mathcal{O}^0_\mathcal{S}, \ldots, \mathcal{O}^n_\mathcal{S}$ such that:

- $\mathcal{O}^0_\mathcal{S} = \mathcal{O}_\mathcal{S}$ and
- for all $1 \leq i \leq n$, the action $\alpha_{\pi(i)}(\vec{X}_{\pi(i)})$ is $(\theta_{\pi(i)}, \gamma_{\pi(i)})$-executable in the state $\mathcal{O}^{i-1}_\mathcal{S}$, and $\mathcal{O}^i_\mathcal{S} = apply(t_{\pi(i)}, \mathcal{O}^{i-1}_\mathcal{S})$.

In this case, AS is said to be *π-executable*, and $\mathcal{O}^n_\mathcal{S}$ is the *final state resulting from the execution* $AS[\pi]$.

A set A of actions is S-concurrently executable on the agent state $\mathcal{O}_\mathcal{S}$, if the set

$$AS = \{(\alpha(\vec{X}), \theta, \gamma) \mid \alpha(\vec{t}) \in A, \alpha(\vec{X}\theta) = \alpha(\vec{t}\theta) \text{ is ground, and } (\theta, \gamma) \in FE(\alpha(\vec{X}))\}$$

is S-concurrently executable on $\mathcal{O}_\mathcal{S}$. □

S-concurrent executions eliminate the problems of consistency that plague the weakly-concurrent executions. For instance, in the `go-right`, `go-left` example above, if the two moves are made one after the other, then the object ends up in one location, which is in this case (5,7) (i.e., the original one). However, this also introduces two weaknesses:

- First, we would like to *deterministically* predict the result of executing a set of actions concurrently. Weakly-concurrent executions allow such predictions, but S-concurrent ones do not.
- Second, the problem of checking whether a set of feasible execution triples is S-concurrently executable is NP-hard (see below), and the intractability shows up even in very simple settings.

The notion of *full-concurrent execution* (F-concurrent execution given below), removes the first of these problems, but not the second. It removes the first problem by saying that a set of feasible execution triples is F-concurrently executable iff each and every sequence of triples from this set is serially executable and the results of each of these serial executions is identical.

Definition 3.7 (full-concurrent executability) Suppose that we have a set

$$AS = \{t_i \mid t_i = (\alpha_i(\vec{X}_i, \theta_i, \gamma_i), 1 \leq i \leq n\}$$

of feasible execution triples and an agent state $\mathcal{O}_\mathcal{S}$. Then, AS is said to be *F-concurrently executable* in state $\mathcal{O}_\mathcal{S}$, if and only if the following holds:

1. For every permutation π, AS is π-executable.
2. For any two permutations π_1, π_2 of AS, the final states resulting from the executions $AS[\pi_1]$ and $AS[\pi_2]$ are identical.

A set A of actions is F-concurrently executable on the agent state $\mathcal{O}_\mathcal{S}$, if the set

$$AS = \{(\alpha(\vec{X}), \theta, \gamma) \mid \alpha(\vec{t}) \in A,\ \alpha(\vec{X}\theta) = \alpha(\vec{t}\theta) \text{ is ground, and } (\theta, \gamma) \in FE(\alpha(\vec{X}))\}$$

is F-concurrently executable on $\mathcal{O}_\mathcal{S}$. □

For instance, the `go-right`, `go-left` example from above is an F-concurrently executable action set, since regardless in which order we execute the actions, we always end up in the same location as moves on the grid commute. However, like S-concurrent execution, F-concurrent execution also suffers from intractability.

The following result specifies the complexity of weakly-concurrent executability, S-concurrent executability, and F-concurrent executability of a set of feasible execution triples. In general, it shows that only weakly-concurrent executability is tractable, while the other notions are intractable.

For deriving this result, we assume that we have a set of feasible execution triples AS to be executed on a given state $\mathcal{O}_\mathcal{S}$, such that following operations are possible in polynomial time:

1. testing whether the grounded precondition $Pre(\alpha(\vec{X}\theta\gamma))$ for any triple $(\alpha(\vec{X}), \theta, \gamma) \in AS$ is satisfied in an agent state;
2. determining all objects in solutions of $Add(\alpha(\vec{X}\theta\gamma))$ and in $Del(\alpha(\vec{X}\theta\gamma))$ on an agent state, as well as insertion/deletion of objects from an agent state;
3. construction of all objects that may be involved in the state and intermediate states evolving from execution of AS on $\mathcal{O}_\mathcal{S}$ under any permutation π.

Such a setting applies e.g. in the case where the agent state is a collection of ground facts, which is maintained under the domain closure axiom.

The following result compares the computation complexity of the three different types of concurrent execution we have introduced in this section. We notice that an epistemically satisfying notion of concurrency (e.g. F-concurrency) is computationally expensive compared to epistemically less desirable notions of concurrency.

Theorem 3.1 *Let AS be a given set of feasible execution triples, and let $\mathcal{O}_\mathcal{S}$ be a given object state. Then, under the previous assumptions, deciding whether AS is*

- *weakly-concurrently executable is polynomial;*
- *S-concurrently executable is* NP*-complete; and*
- *F-concurrently executable is* co-NP*-complete.* ∎

The polynomial time result for weakly-concurrent execution is immediate from the assumptions that we made. The other results are proved in Appendix A. In fact, NP-hardness (resp. coNP-hardness) is present already in a very simple and relevant setting, in which the software package $\mathcal{S}$ is a relational database, and the actions insert and delete tuples from tables, while the preconditions of actions are simple conjunctive queries.

3.2 Action Constraints

As we have already seen in the preceding section, concurrent execution of multiple actions is often difficult. An *action constraint* is an explicit statement saying that a given set of actions is not concurrently executable if certain conditions are met.

Definition 3.8 (action constraint) 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.

Example 3.5 (Grid cont'd) Returning to our simple `go-left` and `go-right` actions on a grid, we may have an action constraint of the form:

$$\{\texttt{go_left}(O_1, X_1, Y_1), \texttt{go_right}(O_2, X_2, Y_2)\} \quad \leftarrow \quad X_1 - 1 = X_2 + 1 \,\&\, Y_1 = Y_2.$$

This says that two objects cannot be simultaneously moved onto the same grid location. □

Definition 3.9 (action constraint satisfaction) A set S of ground actions satisfies an action constraint AC as in (2) on a state $\mathcal{O}_\mathcal{S}$, denoted $S, \mathcal{O}_\mathcal{S} \models AC$, if there is no legal assignment θ of objects in $\mathcal{O}_\mathcal{S}$ to the variables in $\mathcal{AC}$ such that $\chi\theta$ is true and $\{\alpha_1(\vec{X}\theta), \ldots, \alpha_k(\vec{X}\theta)\} \subseteq S$ holds (i.e., no concurrent execution of actions excluded by AC is included in S). We say that S *satisfies* a set $\mathcal{AC}$ of actions constraints on $\mathcal{O}_\mathcal{S}$, denoted $S, \mathcal{O}_\mathcal{S} \models \mathcal{AC}$, if $S, \mathcal{O}_\mathcal{S} \models AC$ for every $AC \in \mathcal{AC}$. □

Clearly, action constraint satisfaction is *hereditary* w.r.t. the set of actions involved, i.e., $S, \mathcal{O}_\mathcal{S} \models \mathcal{AC}$ implies that $S', \mathcal{O}_\mathcal{S} \models \mathcal{AC}$, for every $S' \subseteq S$.

The reader might notice that the action constraint in the previous example can also be *simulated* by an integrity constraint which says that the agent state cannot allow two objects to simultaneously occupy the same grid location. As we will see later in Section 5.5, various action constraints can be simulated with the machinery already in place. In particular, it will turn out that action constraints merely provide syntactic sugar for operations that can be executed within our existing framework. Hence, we do not go into further detail on them for now.

3.3 Agent Programs: Syntax

So far, we have introduced the following important concepts:

- Software Code Calls – this provides a single framework within which the interoperation of diverse pieces of software may be accomplished;
- Software states – this describes exactly what data objects are being managed by a software package at a given point in time;

- Integrity Constraints – this specifies exactly which software states are "valid" or "legal";
- Action Base – this is a set of actions that an agent can physically execute (if the preconditions of the action are satisfied by the software state);
- Action Constraints – this specifies whether a certain set of actions is incompatible.

However, in general, an agent must have an associated "action" policy or action strategy. In certain applications, an agent may be *obliged* to take certain actions when the agent's state satisfies certain conditions. For example, an agent monitoring a nuclear power plant may be obliged to execute a `shutdown` action when some dangerous conditions are noticed. In other cases, an agent may be explicitly forbidden to take certain actions – for instance, agents may be forbidden from satisfying requests for information on US advanced air fighters from Libyan nationals.

In this section, we introduce the concept of *Agent Programs* – programs that specify what an agent must do (in a given state), what an agent must not do, and what an agent is permitted to do, and how the agent can actually select a set of actions to perform that honor its permissions, obligations, and restrictions. Agent Programs are declarative in nature, and have a rich semantics that will be discussed in Section 5.

Definition 3.10 (action status atom) Suppose $\alpha(\vec{t})$ is an action atom, where $\vec{t}$ is a vector of terms (variables or objects) matching the type schema of α. Then, the formulas $\mathbf{P}(\alpha(\vec{t}))$, $\mathbf{F}(\alpha(\vec{t}))$, $\mathbf{O}(\alpha(\vec{t}))$, $\mathbf{W}(\alpha(\vec{t}))$, and $\mathbf{Do}(\alpha(\vec{t}))$ are *action status atoms*. The set $AS = \{\mathbf{P}, \mathbf{F}, \mathbf{O}, \mathbf{W}, \mathbf{Do}\}$ is called the action status set. □

We will often abuse notation and omit parentheses in action status atoms, writing $\mathbf{P}\alpha(\vec{t})$ instead of $\mathbf{P}(\alpha(\vec{t}))$, and so on.

An action status atom has the following intuitive meaning (a more detailed description of the precise reading of these atoms will be provided later in Section 5.2):

- $\mathbf{P}\alpha$ means that the agent is permitted to take action α;
- $\mathbf{F}\alpha$ means that the agent is forbidden from taking 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.

Notice that the operators $\mathbf{P}$, $\mathbf{F}$, $\mathbf{O}$, and $\mathbf{W}$ have been extensively studied in the area of deontic logic [70, 3]. Moreover, the operator $\mathbf{Do}$ is in the spirit of the "praxiological" operator $E_a A$ [51], which informally means that "agent a sees to it that A is the case" [70, p.292].

We borrow from the field of deontic logic the syntax of deontic statements; however we do not lay down the semantics of action programs on the basis of one of the numerous deontic logical systems (e.g., Standard Deontic Logic (SDL), which amounts to the modal logic KD [3, 70]). We discuss the relationship between our approach and deontic logic in detail in Section 6.

Another reason for not building upon existing deontic logic systems is that actions in deontic logic typically do not have effects – hence, the fact that a set of actions may all be individually permitted, but mutually impossible to be concurrently executed is not addressed in deontic logic.

Definition 3.11 (action rule) 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, and each of $L_1, \ldots, L_n$ is either an action status atom, or a code call atom, each of which may be preceded by a negation sign ($\neg$). □

We require that every root variable which occurs in the head A of a rule r and every root- or path-variable occurring in a negative atom also occurs in some positive atom in the body (this is the well-known *safety* requirement on rules [97]).

A rule r is to be understood as being implicitly universally quantified over the variables in it. A rule is called *positive*, if no negation sign occurs in front of an action status atom in its body.

Definition 3.12 (agent program) An *agent program* $\mathcal{P}$ is a finite collection of rules. An agent program $\mathcal{P}$ is *positive*, if all its rules are positive. □

Example 3.6 (Tax Audit Agent Revisited) Here is an example of a simple agent program P, that may be used by a tax agent for the scenario described in Section 1. Recall that this agent uses two relations – one called `returns` that contains a relational representation of the returns filed by tax-payers, and another relation called `employer_decs` which specifies the payments to various individuals reported by employers. The agent monitors discrepancies between the amounts reported by individual taxpayers, and amounts reported by *all* employers who have made payments to that person. If the amount reported by the individual is less than 70% of the employer-reported income, then triggering an audit program is mandatory. If the amount reported by the individual is less than 80% of the employer-reported income, then triggering an audit program is permitted. However, if the amount reported by the individual is over 80% then it is forbidden to run the audit program. This is captured by the following agent program:

$$\begin{aligned}
\mathbf{O}(\texttt{run_audit}(Person)) \leftarrow\ & \texttt{in}(R, \texttt{taxdb:select}(\texttt{returns}, \texttt{name}, =, Person)), \\
& \texttt{in}(TotalInc, \texttt{taxdb:sum_employer_decs}(Person)), \\
& R.amount \leq 0.7 \times TotalInc. \\
\mathbf{P}(\texttt{run_audit}(Person)) \leftarrow\ & \texttt{in}(R, \texttt{taxdb:select}(\texttt{returns}, \texttt{name}, =, Person)), \\
& \texttt{in}(TotalInc, \texttt{taxdb:sum_employer_decs}(Person)), \\
& R.amount \leq 0.8 \times TotalInc. \\
\mathbf{F}(\texttt{run_audit}(Person)) \leftarrow\ & \texttt{in}(R, \texttt{taxdb:select}(\texttt{returns}, \texttt{name}, =, Person)), \\
& \texttt{in}(TotalInc, \texttt{taxdb:sum_employer_decs}(Person)), \\
& R.amount \geq 0.8 \times TotalInc. \\
\mathbf{Do}(\texttt{run_audit}(Person)) \leftarrow\ & \mathbf{P}(\texttt{run_audit}(Person)), \\
& \texttt{in}(R, \texttt{taxdb:select}(\texttt{returns}, \texttt{name}, =, Person)), \\
& \texttt{in}(TotalInc, \texttt{taxdb:sum_employer_decs}(Person)), \\
& TotalInc \geq 200,000. \\
\mathbf{O}(\texttt{send_note}(Person)) \leftarrow\ & \mathbf{Do}(\texttt{run_audit}(Person)).
\end{aligned}$$

Here, $\texttt{sum_employer_decs}(Person)$ is a query provided by the tax database which returns the sum of all the payments reported in `employer_decs` for an indiviual person.

The second last rule says that audits are run on all people making over 200K (according to employer filed returns) on whom an audit run is permitted. The last rule says that if the agent decides to run an audit program on the person's tax return, then the agent is obliged to notify the person that such an audit has been run. The second last rule causes a change in object state – something that classical deontism does not do as it reasons about actions that cause such state changes, but does not trigger them directly. □

Before we complete our discussion of the syntax of agent programs, we add some useful notation. For any action rule r of the form shown in Equation (3) above, we denote by $H(r)$ the atom in the head of r, and by $B(r)$ the collection of literals in the body; by $B^-(r)$ we denote the negative literals among them, and by $B^+(r)$ the positive ones. Moreover, by $\neg.B^-(r)$ we denote the atoms of the negative literals in $B^-(r)$. Finally, the index as (resp., cc) for any of these sets denotes restriction to the literals involving action status atoms (resp., code call atoms).

Example 3.7 For rule r

$$\mathbf{Do}\alpha \leftarrow \mathbf{P}\alpha, \neg\mathbf{Do}\gamma, p(X,Y), \neg s(Y),$$

we have:

$$\begin{aligned}
H(r) &= \mathbf{Do}\alpha,\\
B(r) &= B^+(r) \cup B^-(r), \text{ where } B^+(r) = \{\mathbf{P}\alpha, p(X,Y)\} \text{ and } B^-(r) = \{\neg\mathbf{Do}\gamma, \neg s(Y)\};\\
B_{as} &= B^+_{as} \cup B^-_{as}, \text{ where } B^+_{as} = \{\mathbf{P}\alpha\} \text{ and } B^-_{as} = \{\neg\mathbf{Do}\gamma\}; \text{ and}\\
B_{cc} &= B^+_{cc} \cup B^-_{cc}, \text{ where } B^+_{cc} = \{p(X,Y)\} \text{ and } B^-_{cc} = \{\neg s(Y)\}.
\end{aligned}$$

Likewise, $\neg.B^-(r) = \neg.B^-_{as} \cup \neg.B^-_{cc}$, where $\neg.B^-_{as} = \{\mathbf{Do}\gamma\}$ and $\neg.B^-_{cc} = \{s(Y)\}$. □

Having defined the syntax of agent programs, we are now ready to turn to developing a formal semantics for agent programs.

4 Why this Framework?

Shoham [91, Section 3] states that a complete AOP ("agent oriented programming") system will have three components:

1. "*a restricted formal language with clear syntax and semantics for describing mental state; the mental state will be defined uniquely by several modalities such as belief and committment;*

2. *an interpreted programming language in which to define and program agents with primitive commands such as* `REQUEST` *and* `INFORM`*; the semantics of the programming language will be required to be faithful to the semantics of the mental state;*

3. *an 'agentifier', converting neutral devices into programmable agents.*"

As stated in [91, Section 3], Shoham's paper [91] discussed solutions only to the second component — furthermore, Shoham states [91, Section 3] that "*In contrast to the first two components, of which I have a relatively good understanding, the third component is still somewhat mysterious to me . . .*."

Agentification. Our paper's first contribution is towards the definition of an "Agentification" methodology called AGENTIFY. This procedure specifies how to "agentify" an arbitrary program, thus taking a first step towards the concept of agentification. Figure 1 shows the procedure in the IMPACT Agent Development Environment (**IADE** for short), which provides software tools for its support.

Interpreted Agent Programming Language. A *second* contribution of our paper is the definition of an agent programming language, together with a well defined syntax and semantics. This is also a goal of Shoham's [91] ground-breaking paper, where a syntax is defined, and an *informal* semantics is given. The differences between Agent Programs and Shoham's AGENT-0 may be summed up as follows:

- Agent Programs provide a syntax to access data stored in *arbitrary* data structures, while AGENT-0 does not — in fact, our syntax of code call conditions may also be used to extend AGENT-0 and provide it with this capability.

- Agent Programs provide a mechanism for the agent developer to specify concurrent action execution and in fact allows arbitrary notions of concurrency to be "plugged-in" to agent programs. The semantics of such "plugins" are automatically handled by our semantics (Section 5 will contain the details).

- Agent Programs have a *suite of formal semantics*. As we will see in this paper, and in Part II of this paper, these different semantics offer a range of options to the agent developer — as semantics grow epistemically more desirable, they incur a computational cost. In addition, it is important to note that some agents will decide what to do in a given situation based on some epistemic criteria, while others might use a cost/benefit analyses (e.g. utility theoretic measures). We have recognized this fact, and provide a detailed complexity analysis of these semantics, which makes it possible for the agent developer to select the semantics she deems most suitable for our application.

- While AGENT-0's programming language allows agents to be programmed "with primitive commands such as `REQUEST` and `INFORM`" [91, Section 3], we program these requests *just once*, and the resulting code can get automatically plugged into an agent as described in Steps 2 and 4 of the AGENTIFY algorithm. The ability to do this comes about as a result of our code call condition mechanism.

Mental States. Shoham uses the concept of a "mental state" to denote the beliefs and committments of the agent. This leads to specific modalities that may be used. In contrast, in IMPACT, we allow an agent state to be any set of objects conforming to the agent's associated data types. Specific types within an agent's repertoire can be used to store beliefs, and specific functions within an agent's repertoire can be used to reason with those beliefs. In this paper, we do not focus very much on beliefs — that is discussed in a companion paper by Dix *et al.* [30]. There, it is shown that reasoning with beliefs can be encoded into the concept of agent programs described here. Like Shoham[91], we use modalities to describe committments of the agent.

5 Semantics for Agent Programs

If an agent uses an action program $\mathcal{P}$, the question that the agent must answer, over and over again is: *What is the set of all action status atoms of the form* $\mathbf{Do}\alpha$ *that are true with respect to* $\mathcal{P}$*, the current state,* $\mathcal{O}_\mathcal{S}$*, the underlying set* $\mathcal{AC}$ *of action constraints, and the set* $\mathcal{IC}$ *of underlying integrity constraints on agent*

Procedure AGENTIFY

Input: A program P developed in an arbitrary programming language to be made into an "agent."

Method

1. $\mathcal{T}_P$ = *Set of data types managed by* P.
   IADE allows this to be specified by the agent developer, but it also provides a type-catalog from which the agent developer can cut and paste;
2. $\mathcal{T}_P = \mathcal{T}_P \cup$ *all data types associated with the* `msgbox` *domain described in Example 3.2.*
3. $\mathcal{F}_P$ = *Set of function calls supported by* P.
   IADE allows this to be specified by the agent developer, but it also provides a function-catalog from which the agent developer can cut and paste;
4. $\mathcal{F}_P = \mathcal{F}_P \cup$ all functions supported by the `msgbox` domain described in Example 3.2.
5. Act_P = *Set of all actions the agent is capable of taking. For each such action, specify its preconditions, add set, and delete set, in terms of code call conditions.*
   IADE provides a browsing and editing environment through which the agent developer can build such actions by viewing past action definitions, and by browsing the type, function and action definition catalogs.
6. $\mathcal{IC}_P$ = *Set of all integrity constraints the agent's state must preserve.*
   This must be specified by the agent developer;
7. conc_P = *notion of concurrency this agent uses;*
   IADE allows the agent developer to either select this from a catalog of predefined strategies, or the agent developer can add her own to the catalog.
8. $\mathcal{AC}_P$ = *Set of all action constraints the agent's state must preserve.*
   This must be specified by the agent developer;
9. Sem_P = *Semantics of agent.*
   This is selected by the agent developer from a predefined catalogue.

Figure 1: Agentification in the IMPACT Agent Development Environment

states? This defines the set of actions that the agent must take. In this section, we will provide a series of successively more refined semantics for action programs that answers this question.

In Section 5.1, we will introduce the concept of a feasible status set. Feasible status sets do not, by themselves constitute a semantics for agent programs, but they form the *basic construct* upon which all our semantics will be built.

In Section 5.2, we will define the semantics of Agent Programs to be those feasible status sets that are deemed to satisfy certain rationality requirements. In Subsection 5.3, we add a further requirement – the semantics of an agent program $\mathcal{P}$ is characterized by a subset of rational status sets – those that satisfy an additional *reasonable*-ness condition. This is further refined in Section 5.6.1, where two alternative policies for selecting the "right" reasonable status sets are provided. As feasible status sets may allow certain actions to be neither permitted nor forbidden, we introduce the notation of a complete status set in Section 5.6.2. Two policies are allowed – one of these policies is akin to the closed world assumption in databases [77] (all actions that are not explicitly permitted are forbidden) and the other is akin to the open world assumption (all actions that are not explicitly forbidden are allowed).

All the preceding semantics describe ways of syntactically selecting one or more feasible status sets as somehow being the "right" feasible status sets. For example, rational status sets are all feasible status sets, but not vice-versa. Reasonable status sets are also feasible status sets (and in fact rational status sets) but not vice-versa. The same applies to the other types of status sets. In Section 5.7, we use numerical cost measures to select status sets. Given a semantics Sem where $\mathsf{Sem} \in \{$ Feasible, Rational, Reasonable, F-preferential, P-preferential, Weak Rational$\}$, Section 5.7 shows how to associate a "cost" with each Sem-status set. An optimal Sem-status set is one which minimizes cost. A status set's cost may be defined in terms of (1) the cost of performing the $\mathbf{Do}$ actions in that status set, and/or (2) the "badness" value of the state that results, and/or (3) a mix of the previous two criteria. Section 5.7 will define these expressions formally.

Figure 2 captures the relationship between these different semantic structures. The definition of Sem-status sets is layered on top of the definitions of the other semantics – hence, to avoid clutter, we do not include them in this figure.

5.1 Feasible status sets

In this section, we will introduce the important concept of a feasible status set. While feasible status sets do not constitute a semantics for agent programs, every semantics we define for Agent Programs will build upon this basic definition.

Intuitively, a feasible status set consists of assertions about the status of actions, such that these assertions are compatible with (but are not necessarily forced to be true by) the rules of the agent program and the underlying action and integrity constraints.

In what follows, we assume the existence of a body of software code $\mathcal{S} = (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S})$, an action base $\mathcal{AB}$, and action and integrity constraints $\mathcal{AC}$ and $\mathcal{IC}$, respectively, in the background. The first concept we introduce are status sets.

Definition 5.1 (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 $Op(S) = \{\alpha \mid Op(\alpha) \in S\}$. □

Informally, a status set S represents information about the status of ground actions. If some atom $Op(\alpha)$ occurs in S, then this means that the status Op is true for α. For example, $\mathbf{Do}(\alpha), \mathbf{F}(\beta) \in S$ means that action α will be taken by the agent, while action β is forbidden. Of course, not every status set is

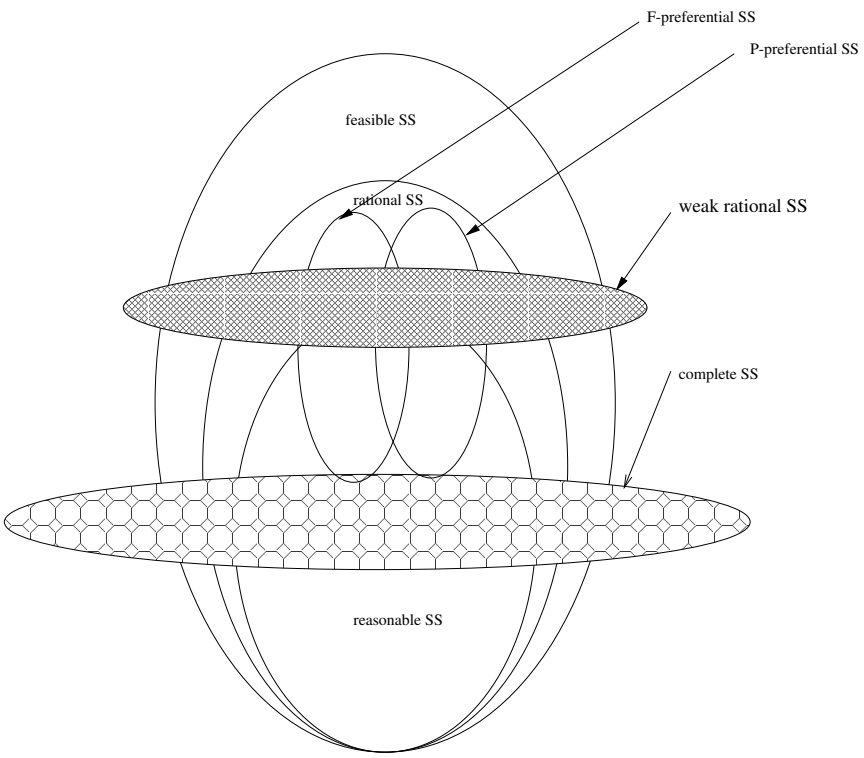


Figure 2: Relationship between different Status Sets (SS)

meaningful. For example, if both $\mathbf{F}(\alpha)$ and $\mathbf{P}(\alpha)$ are in S, then S is intuitively inconsistent, since α can not be simultaneously permitted and forbidden. In order to characterize the meaningful status sets, we introduce the concepts of deontic and action consistency.

Definition 5.2 (deontic and action consistency) A status set S is called *deontically consistent*, if it satisfies the following rules for any ground action α:

- *If* $\mathbf{O}\alpha \in S$, *then* $\mathbf{W}\alpha \notin S$
- *If* $\mathbf{P}\alpha \in S$, *then* $\mathbf{F}\alpha \notin S$
- *If* $\mathbf{P}\alpha \in S$, *then* $\mathcal{O}_\mathcal{S} \models Pre(\alpha)$ *(i.e.,* α *is executable in the state* $\mathcal{O}_\mathcal{S}$*).*

A status set S is called *action consistent*, if $S, \mathcal{O}_\mathcal{S} \models \mathcal{AC}$ holds. □

Besides consistency, we also wish that presence of particular atoms in S entails the presence of other atoms in S. For example, if $\mathbf{O}\alpha$ is in S, then we expect that $\mathbf{P}\alpha$ is also in S, and if $\mathbf{O}\alpha$ is in S, then we would like to have $\mathbf{Do}\alpha$ in S. This is captured by the concept of deontic and action closure.

Definition 5.3 (deontic and action closure) The *deontic closure* of a status S, denoted $DCl(S)$, is the closure of S under the rule

If $\mathbf{O}\alpha \in S$, *then* $\mathbf{P}\alpha \in S$

where α is any ground action. We say that S is *deontically closed*, if $S = DCl(S)$ holds.

The *action closure* of a status set S, denoted $ACl(S)$, is the closure of S under the rules

If $\mathbf{O}\alpha \in S$, *then* $\mathbf{Do}\alpha \in S$

If $\mathbf{Do}\alpha \in S$, *then* $\mathbf{P}\alpha \in S$

where α is any ground action. We say that a status S is action-closed, if $S = ACl(S)$ holds. □

The reader will easily notice that status sets that are action-closed are also deontically closed, i.e.

- $ACl(S) = S$ *implies* $DCl(S) = S$
- $DCl(S) \subseteq ACl(S)$, *for all* S.

A status set S which is consistent and closed is certainly a meaningful assignment of a status to each ground action. Notice that we may have ground actions α that do not occur anywhere within a status set – this means that no commitment about the status of α has been made.

Given a status set S, the operator $App_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S)$ computes all action status atoms that may be inferred to be true by allowing the rules in $\mathcal{P}$ to fire exactly once.

Definition 5.4 (Operator $App_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S)$) Suppose $\mathcal{P}$ is an agent program, and $\mathcal{O}_\mathcal{S}$ is an agent state. Then, $App_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S)$ is defined to be the set of all ground action status atoms A such that there exists a rule in $\mathcal{P}$ having a ground instance of the form $r : A \leftarrow L_1, \ldots, L_n$ such that

1. $B^+_{as}(r) \subseteq S$ and $\neg.B^-_{as}(r) \cap S = \emptyset$, and
2. every code call $\chi \in B^+_{cc}(r)$ succeeds in $\mathcal{O}_\mathcal{S}$, and
3. every code call $\chi \in \neg.B^-_{cc}(r)$ does not succeed in $\mathcal{O}_\mathcal{S}$, and
4. for every atom $Op(\alpha) \in B^+(r) \cup \{A\}$ such that $Op \in \{\mathbf{P}, \mathbf{O}, \mathbf{Do}\}$, the action α is executable in state $\mathcal{O}_\mathcal{S}$. □

Note that part (4) of the above definition only applies to the "positive" modes $\mathbf{P}, \mathbf{O}, \mathbf{Do}$. It does not apply to atoms of the form $\mathbf{F}\alpha$ as such actions are not executed, nor does it apply to atoms of the form $\mathbf{W}\alpha$ because such actions are executed only if $\mathbf{Do}\alpha$ is true.

Example 5.1 For example, consider a simple agent program P consisting of the rules:

$$\begin{aligned} \mathbf{Do}\alpha(X) &\leftarrow \mathtt{in}(X, \mathtt{d{:}f(a,b)}),\ \mathbf{F}\beta_1(X),\ \neg\mathbf{P}\beta_2(X). \\ \mathbf{P}\gamma(X) &\leftarrow \mathtt{in}(X, \mathtt{d{:}f(a,b)}),\ \mathbf{P}\beta_2(X). \end{aligned}$$

Suppose that in the current state, $\mathtt{d{:}f(a,b)}$ returns $\{a, b\}$. Further, suppose that $\beta_1(b)$ and $\gamma(b)$'s precondition is false in the current state, and that all other actions have true preconditions. Then:

$$App_{\mathcal{P},\mathcal{O}_\mathcal{S}}(\{\mathbf{F}\beta_1(a)\}) = \{\mathbf{Do}\alpha(a)\}; \quad (4)$$

$$App_{\mathcal{P},\mathcal{O}_\mathcal{S}}(\{\mathbf{F}\beta_1(a), \mathbf{P}\beta_2(a), \mathbf{P}\beta_2(b)\}) = \{\mathbf{P}\gamma(a)\}. \quad (5)$$

Note that in Equation (5), $\mathbf{P}\gamma(b)$ is not obtained from the second rule, since $\gamma(b)$ is not executable in the current state (i.e., its precondition is false). ∎

Our approach is to base the semantics of agent programs on consistent and closed status sets. However, we have to take into account the rules of the program as well as integrity constraints. This leads us to the notion of a feasible status set.

Definition 5.5 (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) $\quad App_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S) \subseteq S$;

$(S2)$ (deontic and action consistency) $\quad S$ is deontically and action consistent;

$(S3)$ (deontic and action closure) $\quad S$ is action closed and deontically closed;

$(S4)$ (state consistency) $\quad \mathcal{O}'_\mathcal{S} \models \mathcal{IC}$, where $\mathcal{O}'_\mathcal{S} = apply(\mathbf{Do}(S), \mathcal{O}_\mathcal{S})$ is the state which results after taking all actions in $\mathbf{Do}(S)$ on the state $\mathcal{O}_\mathcal{S}$. □

Notice that condition $(S2)$ is hereditary, i.e., if a status set S satisfies $(S2)$, then any subset $S' \subseteq S$ satisfies $(S2)$ as well.

In general, there are action programs that have zero, one or several feasible status sets.

Example 5.2 (Tax Audit Agent Revisited) Let us return to the Tax Example (cf. Example 3.6). Let us consider the case where the tax database system, `taxdb`, contains the following relations:

Relation `returns`

Name	it Amount
John Smith	50,000
Jane Shady	78,000
Denis Rumble	35,000
...	...

Relation `employer_decs`

Name	*Company*	*Amount*
John Smith	ABC Inc.	46,000
John Smith	DEF Inc.	35,000
Jane Shady	DEF Inc.	100,000
Denis Rumble	ABC Inc.	34,000
...	...	...

It is easy to see that of these individuals, John Smith has declared under 70% of his income (as reported by ABC Inc. and DEF Inc.). By the rules, it is obligatory to audit him, i.e. if we take $\mathcal{P}$ to be the set of rules in Example 3.6, then $\mathbf{O}$`run_audit`(John Smith) is implied by the program. In the same vein, $\mathbf{P}$`run_audit`(Jane Shady) is true. Last but not least, $\mathbf{F}$`run_audit`(Denis Rumble) is true.

If we assume there are no integrity constraints and no action constraints, then this leads to the following two possible feasible status sets:

$$\begin{aligned} FSS_1 \quad = \{ \quad & \mathbf{O}\texttt{run_audit}(\text{John Smith}), \mathbf{Do}\texttt{run_audit}(\text{John Smith}), \\ & \mathbf{P}\texttt{run_audit}(\text{John Smith}), \mathbf{F}\texttt{run_audit}(\text{Denis Rumble}), \\ & \mathbf{P}\texttt{run_audit}(\text{Jane Shady})\}, \end{aligned}$$

$$\begin{aligned} FSS_2 \quad = \{ \quad & \mathbf{O}\texttt{run_audit}(\text{John Smith}), \mathbf{Do}\texttt{run_audit}(\text{John Smith}), \\ & \mathbf{P}\texttt{run_audit}(\text{John Smith}), \mathbf{F}\texttt{run_audit}(\text{Denis Rumble}), \\ & \mathbf{P}\texttt{run_audit}(\text{Jane Shady}), \mathbf{Do}\texttt{run_audit}(\text{Jane Shady}) \}. \end{aligned}$$

(Other feasible status sets exist, though.) The two feasible status sets differ on whether Jane Shady is actually audited or not. □

Example 5.3 The program $\mathcal{P}$ containing the rules

$$\begin{aligned} \mathbf{P}\alpha &\leftarrow \\ \mathbf{F}\alpha &\leftarrow \end{aligned}$$

clearly does not have any feasible status set. □

The following are immediate consequences of the definition of a feasible status set, which confirm that it appropriately captures a "possible" set of actions dictated by an agent program that is consistent with the obligations and restrictions on the agent program.

Proposition 5.1 *Let S be a feasible status set. Then,*

If $\mathbf{Do}(\alpha) \in S$, then $\mathcal{O}_{\mathcal{S}} \models pre(\alpha)$;

If $\mathbf{P}\alpha \notin S$, then $\mathbf{Do}(\alpha) \notin S$;

If $\mathbf{O}\alpha \in S$, then $\mathcal{O}_{\mathcal{S}} \models Pre(\alpha)$;

If $\mathbf{O}\alpha \in S$, then $\mathbf{F}\alpha \notin S$; ∎

The reader may be tempted to believe that Condition 4. in Definition 5.4 is redundant. However, as the following agent program $\mathcal{P}$ amply demonstrates, this is not the case.

Example 5.4 Consider the agent program $\mathcal{P}$ given by:

$$\mathbf{P}\alpha \leftarrow$$

Assume that α is not executable in state $\mathcal{O}_{\mathcal{S}}$. Then, under the current definition, no feasible status set S contains $\mathbf{P}\alpha$; e.g., $S = \emptyset$ is a feasible status set. If we drop condition 4 from Definition 5.4, then no feasible status set S exists, as $\mathbf{P}\alpha$ must be contained in every such S, which then violates deontic consistency. □

5.2 Rational status sets

Intuitively, a feasible status set describes a set of status atoms that are compatible with the state of the software, the obligations and restrictions imposed on the agent by its associated Agent Program, and the deontic consistency requirements. Nevertheless, we note that feasible status sets may include **Do**ing actions that are not strictly necessary.

For example, let us return to our tax audit scenario. Our system may have the following rules.

Example 5.5 (Expanded Tax Audit Agent) For some reason, the tax agent has decided that it is obliged to sue an unfortunate individual called Jim Black. However, there is a rule that says that if it has failed to previously interview the person and failed to previously issue a notice to the person, then it is forbidden to sue to the taxpayer. This can be represented as the following set of rules. These rules may be added to the Agent Program described in Example 3.6.

$$\begin{aligned} \mathbf{O}\texttt{sue}(\text{Jim Black}, T+1) &\leftarrow . \\ \mathbf{F}\texttt{sue}(Person, T) &\leftarrow \neg\mathbf{Do}\texttt{issue_notice}(P, T_1)\,,\ T_1 < T\,, \\ &\quad\ \neg\mathbf{Do}\texttt{interview}(P, T_2)\,,\ T_2 < T. \end{aligned}$$

The action $sue(Person, T)$ has no preconditions and no effects on the state of the system (except to send a message to another agent that initiates the lawsuit). This agent program has the following feasible status sets. (We assume that the only time points we are interested in are now and $(now + 1)$ – furthermore, for notational simplicity, we do not explicitly list implied action status atoms of them form $\mathbf{W}\alpha$ and $\mathbf{P}\alpha$).

$$\begin{aligned} FSS_1 &= \{\, \mathbf{O}\texttt{sue}(\text{Jim Black}, now+1), \\ &\quad \mathbf{Do}\texttt{sue}(\text{Jim Black}, now+1), \mathbf{Do}\texttt{issue_notice}(\text{Jim Black}, now), \ldots\}. \\ FSS_2 &= \{\, \mathbf{O}\texttt{sue}(\text{Jim Black}, now+1), \\ &\quad \mathbf{Do}\texttt{sue}(\text{Jim Black}, now+1), \mathbf{Do}\texttt{interview}(\text{Jim Black}, now), \ldots\}. \\ FSS_3 &= \{\, \mathbf{O}\texttt{sue}(\text{Jim Black}, now+1), \mathbf{Do}\texttt{sue}(\text{Jim Black}, now+1), \\ &\quad \mathbf{Do}\texttt{interview}(\text{Jim Black}, now), \mathbf{Do}\texttt{issue_notice}(\text{Jim Black}, now), \ldots\}. \end{aligned}$$

Here, we only show $\mathbf{O}$ and $\mathbf{Do}$ atoms in the feasible status sets; each status may be completed by adding appropriate further atoms involving other modalities.

The first status set says that to sue Jim Black at time $(now + 1)$, the tax agent must issue a notice to him now. The second status set says that to sue Jim Black at time $(now + 1)$, the tax agent must interview him now. The last status set says that we should both interview him and issue a notice to him now. (The reader can easily see how this example may be expanded to accommodate a larger time window, allowing Jim Black some extra time to respond, etc.)

If one examines FSS_3, this is a perfectly valid feasible status set. However, it takes one action that is not strictly necessary. To meet the obligation of suing Jim Black at time $(now + 1)$, either he should be issued a notice now, or should be interviewed now – there is no need for both. Surely, then, FSS_3 represents a case where the agent is doing "too much" ? □

The notion of a rational status set is postulated to accommodate this kind of reasoning. It is based on the principle that each action which is executed should be sufficiently "grounded" or "justified" by the agent program. That is, there should be evidence from the rules of the agent program that a certain action must be executed. For example, it seems unacceptable that an action α is executed, if α does not occur in any rule of the agent program at all.

This way, we also have to make sure that execution of an action must not be driven by the need to preserve the consistency of the agent's state. Rather, the integrity constraints should serve to prevent executions which appear to be rational if no integrity constraints were present. This motivates the following formal notion of groundedness.

Definition 5.6 (groundedness; rational status set) A status set S is *grounded*, if there exists no status set S' different from S such that $S' \subset S$ 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. □

Notice that if S is a feasible status set, then every $S' \subseteq S$ satisfies the condition $(S2)$ of feasibility. Therefore, the requirement of $(S2)$ for S' in the definition of groundedness is redundant. However, it seems more natural to have this condition included in the definition of groundedness. Moreover, if we did not have hereditary action consistency, then inclusion of action consistency would be indispensable.

Example 5.6 (Tax Example Continued) The program $\mathcal{P}$ of Example 5.5 has two rational status sets: FSS_1 and FSS_2. In this case, as no integrity constraints $\mathcal{IC}$ are specified, the rational status sets happen to be the minimal feasible status sets with respect to set inclusion. □

Example 5.7 (Simple Driving Example) Suppose we want an agent which selects – in a simplified setting – the driving lane for a car. The action base contains the two actions `go_right` and `drive(`$Lane$`)`. The precondition of the `go_right` is empty, while the precondition of `drive(`$Lane$`)` is $free(Lane)$. The agent program contains the following rules:

$$\begin{aligned}
\mathbf{O}(\texttt{go_right}) &\leftarrow \\
\mathbf{O}(\texttt{drive}(\texttt{right_lane})) &\leftarrow \mathbf{Do}(\texttt{go_right}) \\
\mathbf{F}(\texttt{drive}(Lane)) &\leftarrow \neg free(Lane) \\
\mathbf{Do}(\texttt{drive}(\texttt{left_lane})) &\leftarrow \mathbf{F}(\texttt{drive}(\texttt{right_lane}))
\end{aligned}$$

Here, we suppose that there are two lanes, a left lane and a right lane. The first rule says that we must go on the right side, and the second that we must drive on the right lane if we actually go on the right side. The third rule tells that it is forbidden to use a lane if it is not free, i.e., it is blocked, while the last rules says that we go on the left lane if we can not use the right lane.

Depending on the status of the lanes (free or blocked), the program has different rational status sets. In each of the four possible cases, the program has a unique rational status set. All of them contain $\mathbf{O}(\texttt{go_right})$, $\mathbf{Do}(\texttt{go_right})$, and $\mathbf{P}(\texttt{go_right})$. Two cases arise, depending on whether the right lane is free or not.

(1) Suppose the right lane is free. Then, by the second rule the obligation to drive in the right lane is derived. The agent can take this action without raising an inconsistency, regardless of whether the left lane is free or blocked. In either case, we obtain a rational status set S such that $\mathbf{Do}(S) = \{\texttt{go_right}, \texttt{drive}(\texttt{right_lane})\}$.

(2) Suppose the right lane is blocked. Then, the precondition of $\texttt{drive}(\texttt{right_lane})$ fails, and the atom $\mathbf{F}(\texttt{drive}(\texttt{right_lane}))$ is derived by the third rule. Now, if the left lane is free, then we can derive by the last rule $\mathbf{Do}(\texttt{drive}(\texttt{left_lane}))$, and we obtain a rational status set S such that $\mathbf{Do}(S) = \{\texttt{go_right}, \texttt{drive}(\texttt{left_lane})\}$; on the other hand, if the left lane is blocked, the last rule does not yield the atom $\mathbf{Do}(\texttt{drive}(\texttt{left_lane}))$ as the precondition of $\texttt{drive}(\texttt{left_lane})$ fails. In this case, we obtain a rational status set S such that $\mathbf{Do}(S) = \{\texttt{go_right}\}$.

The behavior of the agent is perfectly as desired. □

Observe that the definition of groundedness does not include condition $(S4)$ of a feasible status set. A moment of reflection will show that omitting this condition is indeed appropriate. Recall that the integrity constraints must be maintained when the current agent state is changed into a new agent state. If we were to include the condition $(S4)$ in groundedness, it may happen that the agent is forced to execute some actions which the program does not mention, just in order to maintain the integrity constraints. The following example illustrates this point.

Example 5.8 Suppose that in the Tax Audit Agent example, the table `employer_decs` has attached the integrity constraint IC which says that for each person P and company C, at most one record for this person and that company is filed. Furthermore, assume that actions $\texttt{add_ed}(P, C, A)$ and $\texttt{del_ed}(P, C)$ for adding and deleting a record from `employer_decs`, respectively, are available, and an agent program has the simple rule:

$$\mathbf{Do}\texttt{add_ed}(\text{JohnSmith}, \text{ABC Inc.}, 30,000) \leftarrow .$$

However, in the state reported in Example 5.2 there is already a record (John Smith,ABC Inc.,46,000) in the table, and thus adding the record (John Smith,ABC Inc.,30,000) violates the integrity constraint IC. Therefore,

$$S_1 = \{\mathbf{Do}\texttt{add_ed}(\text{John Smith,ABC Inc.,30,000}), \mathbf{P}\texttt{add_ed}(\text{John Smith,ABC Inc.,30,000})\}$$

is not a feasible status set. However, the status set

$$\begin{aligned} S_2 = \{&\mathbf{Do}\texttt{add_ed}(\text{John Smith, ABC Inc., 30, 000}), \mathbf{P}\texttt{add_ed}(\text{John Smith,ABC Inc.,30,000}),\\ &\mathbf{Do}\texttt{del_ed}(\text{John Smith,ABC Inc.,46,000}), \mathbf{P}\texttt{del_ed}(\text{John Smith,ABC Inc.,46,000})\} \end{aligned}$$

is a feasible status set, and it is easily seen that no smaller feasible status set S_3 exists such that $S_3 \subset S_2$. The implicit execution of `del_ed`(John Smith,ABC Inc.,46,000) may be unwanted, however, and thus S_2 is intuitively not acceptable as the "right" set of actions to take by the agent. This is expressed by the fact that the smaller status set $S_1 \subset S_2$ is sound with the rules of the program, and no extra actions for maintaining the integrity constraints should be taken by the agent.

Observe that automatic maintenance of integrity constraints is an ongoing research issues in databases, and a simple, declarative solution to this problem is by no means clear [45]. Therefore, we do not delve into the intricated and complex more general problem here. □

The fact that the program in Example 5.7 always had a unique rational status set in each of the possible scenarios, was not accidental. In fact, as will be shown below, *positive* programs enjoy the benign property of having a unique rational status sets, if any rational status set exists. Observe that this property does not hold for non-positive agent programs in general.

It is possible to give a characterization of the unique rational status set in terms of a fixpoint operator, akin to the least fixpoint of logic programs [58, 2]. For that, we define for every positive program $\mathcal{P}$ and agent state $\mathcal{O}_\mathcal{S}$ an operator $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}$ which maps a status set S to another status set.

Definition 5.7 ($T_{\mathcal{P},\mathcal{O}_\mathcal{S}}$ Operator) Suppose $\mathcal{P}$ is an agent program and $\mathcal{O}_\mathcal{S}$ an agent state. Then, for any status set S,

$$T_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S) = App_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S) \cup DCl(S) \cup ACl(S).$$ □

Note that as $DCl(S) \subseteq ACl(S)$, we may equivalently write this as

$$T_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S) = App_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S) \cup ACl(S).$$

The following technical property of feasible status sets says that if S is a status set which is deontically/action closed and closed under the program rules, then $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S) \subseteq S$. In other words, $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S)$ cannot add any new action status atoms to S, though it might eliminate some from S.

Lemma 5.2 *Let $\mathcal{P}$ be an agent program, let $\mathcal{O}_\mathcal{S}$ be any agent state, and let S be any status set. If S satisfies $(S1)$ and $(S3)$ of feasibility, then S is a pre-fixpoint of $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}$, i.e., $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S) \subseteq S$.*

The above lemma is a technical stepping stone that enables us to show that when positive agent programs $\mathcal{P}$ are considered, then the $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}$ operator is guaranteed to have a fixpoint. The reason for this is that if the program $\mathcal{P}$ is positive, then $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}$ is a monotone operator, i.e., $S \subseteq S'$ implies $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S) \subseteq$

$T_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S')$, and hence, it has a least fixpoint $lfp(T_{\mathcal{P},\mathcal{O}_\mathcal{S}})$. Moreover, since $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}$ is in fact continuous, i.e., $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}(\bigcup_{i=0}^{\infty} S_0) = \bigcup_{i=0}^{\infty} T_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S_i)$ for any chain $S_0 \subseteq S_1 \subseteq S_2 \subseteq \cdots$ of status sets, the least fixpoint is given by

$$lfp(T_{\mathcal{P},\mathcal{O}_\mathcal{S}}) = \bigcup_{i=0}^{\infty} T^i_{\mathcal{P},\mathcal{O}_\mathcal{S}},$$

where $T^0_{\mathcal{P},\mathcal{O}_\mathcal{S}} = \emptyset$ and $T^{i+1}_{\mathcal{P},\mathcal{O}_\mathcal{S}} = T_{\mathcal{P},\mathcal{O}_\mathcal{S}}(T^i_{\mathcal{P},\mathcal{O}_\mathcal{S}})$, for all $i \geq 0$ (see e.g. [58, 2]).

We may use the above reasoning to establish the following important result. The result states that for positive agent programs, the only possible rational status set is the least fixpoint of $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}$. This theorem therefore provides a mechanism to construct a rational status set for positive agent programs, by directly computing $lfp(T_{\mathcal{P},\mathcal{O}_\mathcal{S}})$.

Theorem 5.3 *Let $\mathcal{P}$ be a positive agent program, and let $\mathcal{O}_\mathcal{S}$ be an agent state. Then, S is a rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, if and only if $S = lfp(T_{\mathcal{P},\mathcal{O}_\mathcal{S}})$ and S is a feasible status set.*

Notice that in case of a positive program, $lfp(T_{\mathcal{P},\mathcal{O}_\mathcal{S}})$ always satisfies the conditions $(S1)$ and $(S3)$ of a feasible status set (i.e., all closure conditions), and thus is a rational status set if it satisfies $(S2)$ and $(S4)$, i.e., the consistency criteria. The uniqueness of the rational status set is immediate from the previous theorem.

Corollary 5.4 *Let $\mathcal{P}$ be a positive agent program. Then, on every agent state $\mathcal{O}_\mathcal{S}$, the rational status set of $\mathcal{P}$ (if one exists) is unique, i.e., if S, S' are rational status sets for $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, then $S = S'$.*

As shown by Example 5.5, Corollary 5.4 is no longer true in the presence of negated action status atoms. We postpone the discussion of the existence of a unique rational status set at this point, since we will introduce a stronger concept than rational status sets below for which this discussion seems more appropriate. Nonetheless, we note the following property on the existence of a (not necessarily unique) rational status set.

Proposition 5.5 *Let $\mathcal{P}$ be an agent program. If $\mathcal{IC} = \emptyset$, then $\mathcal{P}$ has a rational status set if and only if $\mathcal{P}$ has a feasible status set.*

Moreover, we remark at this point that the unique rational status set of a positive program (if it exists) can be computed in polynomial time under suitable assumptions.

5.2.1 Reading of rational status sets

We are now ready to return to the question "*Exactly how should we read the atoms $Op(\alpha)$ for $Op \in \{\mathbf{P}, \mathbf{F}, \mathbf{W}, \mathbf{O}, \mathbf{Do}\}$ appearing in a rational status set ?*" In Section 3.3, we had promised a discussion of this issue. It appears that an interpretation:

$Op(\alpha) \equiv$ "It is the case that α is Op^*,"

where Op^* is the proper verb corresponding to operator Op (forbidden, permitted, etc), is *not* the one which is expressed inherently by rational status sets. Rather, a status atom in a rational status set should be more appropriately interpreted as follows:

$Op(\alpha) \equiv$ "It is derivable that α should be Op^*."

where "derivable" – without giving a precise definition here – means that $Op(\alpha)$ is obtained from the rules of the agent program and the deontic axioms, under reasonable assumptions about the status of actions; the groundedness property of the rational status set ensures that the adopted assumptions are as conservative as possible.

Furthermore, a literal $\neg Op(\alpha)$ in a program should be interpreted as:

$\neg Op(\alpha) \equiv$ "It is not derivable that α should be Op^*."

It is important to emphasize that there is no reason to view a rational status set as an ideally rational agent's three-valued model of a two-valued reality, in which each action is either forbidden or permitted. For example, the agent program

$$\begin{aligned} \mathbf{P}\alpha &\leftarrow \\ \mathbf{F}\alpha &\leftarrow \mathbf{P}\beta \\ \mathbf{F}\alpha &\leftarrow \mathbf{F}\beta \end{aligned}$$

has a unique rational status set, namely $S = \{\mathbf{P}\alpha\}$. A possible objection against this rational status set (which arises naturally from similar arguments in logic programming with incomplete information) is the following.

1. β is either forbidden or permitted.
2. In either of these two cases, α is forbidden.
3. Therefore, the rational status set $S = \{\mathbf{P}\alpha\}$ is "wrong."

"Complete" status sets, defined in Section 5.6 remedy this problem (at a cost, as we shall see later when complexity issues are discussed).

This brings us back to our interpretation of $Op\alpha$. The fallacy in the above argument is the implicit equivalence assumed to hold between the statement "β is either forbidden or permitted" and the statement $\mathbf{P}\beta \vee \mathbf{F}\beta$. The latter statement is read "It is either derivable that β is permitted, or it is derivable that β is forbidden" which is certainly very different from the former statement.

In addition, we believe that deontic logic is different from the setting of reasoning with incomplete information because the true state of affairs need not be one in which the status of every particular action is decided. In fact, the status may be open – and it may even be impossible to refine it without arriving at inconsistency. For example, this applies to the legal domain, which is one of the most fertile application areas of deontic logic.

5.3 Reasonable status sets

A more serious attack against rational status sets, stemming from the authors' background in nonmonotonic logic programming is that for agent programs with negation, the semantics of rational status sets allows logical contraposition of the program rules. For example, consider the following program:

$$\mathbf{Do}(\alpha) \leftarrow \neg\mathbf{Do}(\beta).$$

This program has two rational status sets: $S_1 = \{\mathbf{Do}(\alpha), \mathbf{P}(\alpha)\}$, and $S_2 = \{\mathbf{Do}(\beta), \mathbf{P}(\beta)\}$. The second rational status set is obtained by applying the contrapositive of the rule:

$$\mathbf{Do}(\beta) \leftarrow \neg\mathbf{Do}(\alpha)$$

However, the second rational set seems less intuitive than the first as there is no explicit rule in the above program that justifies the derivation of this $\mathbf{Do}(\beta)$.

This observation leads us to the following remarks. First, in the area of logic programming and knowledge representation, the meaning of negation in rules has been extensively discussed and there is broad consensus in that area that contraposition is a proof principle which should not be applied: rather, derivation should be constructive from rules. These observations led to the well known stable model semantics for logic programs due to Gelfond and Lifschitz [37] which in turn was shown to have strong equivalences with the classical nonmonotonic reasoning paradigms such as default logic [78] and auto-epistemic logic[71] (see [38, 65]), as well as numerical reasoning paradigms such as linear programming and integer programming [14, 15].

Second, the presence of derivation by contraposition may have a detrimental effect on the complexity of programs, since it inherently simulates disjunction. Therefore, it is advisable to have a mechanism which cuts down possible rational status sets in an effective and appealing way, so that negation can be used without introducing high computational cost.

For these reasons, we introduce the concept of a *reasonable status set*. The reader should note that if he really does want to use contraposition, then he should choose the rational status set approach, rather than the reasonable status set approach.

Definition 5.8 (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 $\mathcal{O}_\mathcal{S}$, if and only if S is a rational status set for $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$.

2. The reduct of $\mathcal{P}$ w.r.t. 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 $\mathcal{P}$ w.r.t. $\mathcal{O}_\mathcal{S}$, if it is a reasonable status set of the program $red^S(\mathcal{P}, \mathcal{O}_\mathcal{S})$ with respect to $\mathcal{O}_\mathcal{S}$. □

Suppose we think of S as a guess about all action status atoms that are possibly true in a feasible status set. We then want to check what the necessary consequences of this assumption are, assuming it is correct. If these necessary consequences coincided with our initial guess, then our guess was "good" in the sense that we did not guess too much or too little.

The program $red^S(\mathcal{P}, \mathcal{O}_\mathcal{S})$ is constructed according to the above intuition: if a rule r has an action status literal of the form $\neg Op\alpha$ in its body where $\alpha \in S$, then if guess S were correct, then rule r cannot possibly fire, and hence it can be eliminated. On the other hand if all negative action status literals of the form $\neg Op\alpha$ in its body are not in S, then none of the $Op\alpha$'s is deemed possible by our guess, so we may eliminate them from rule r's body. Note that we cannot eliminate positive literals because we need to know that they are necessarily true, but our guess merely says they are possible.

Now we check if all the action status atoms that necessarily follow from the simplified program $red^S(\mathcal{P}, \mathcal{O}_\mathcal{S})$ coincide with our initial guess S — if so, then we did not guess too much or too little, and we declare S to be a reasonable status set.

Let us quickly revisit our Tax Audit Agent scenario to see why reasonable status sets reflect an improvement on rational status sets.

Example 5.9 (Tax Audit Agent Example, Revisited) Suppose we reconsider the tax audit agent example, as described in Examples 3.6 and 5.2. However, we merely consider the two rules listed below.

$$\begin{aligned}
\mathbf{F}\texttt{send_refund}(Person) &\leftarrow \mathbf{Do}\texttt{run_audit}(Person),\\
&\quad \neg\texttt{in}(Person, \texttt{taxdb} : \texttt{refund_authorized}()).\\
\mathbf{Do}\texttt{send_refund}(Person) &\leftarrow \neg\mathbf{F}\texttt{send_refund}(Person),\\
&\quad \texttt{in}(Person, \texttt{taxdb} : \texttt{refund_authorized}()).
\end{aligned}$$

The first rule above says that sending a refund to an audited person is forbidden unless the refund has been explicitly authorized. One may think of a situation where all audit results are sent to a human being who examines the audit result and determines whether to authorize a refund or not. In the former case, he explicitly updates a list of people to whom refunds may be sent - this list of people is retrieved by the `refund_authorized()` call. The second rule says that we may send a refund to anyone who is not explicitly forbidden from receiving a refund. Here, the preconditions of the actions `send_refund` and `run_audit` are assumed to be void for the sake of simplicity.

Rational Status Sets: Consider now the case of an individual, John Doe, who has been audited, and whose refund has been authorized by a human being. In this case, the precondition of the first rule is not true. It is important to note that the reduct of this program does *not* affect the first rule, because the negation (in the body of the first rule) is in front of a code call and not in front of an action status atom.

Now consider the second rule – we have two rational status sets – one in which John Doe's refund is sent, while in the other, it is forbidden. However, the latter is clearly incorrect. The reason for this is because in rational status sets, the second rule is treated as equivalent to its converse:

$$\begin{aligned}
\mathbf{F}\texttt{send_refund}(Person) &\leftarrow \neg\mathbf{Do}\texttt{send_refund}(Person),\\
&\quad \texttt{in}(Person, \texttt{taxdb} : \texttt{refund_authorized}()).
\end{aligned}$$

Reasonable Status Sets: Now let us consider the sets $S_1 = \{\mathbf{F}\texttt{send_refund}(\text{John Doe})\}$ and $S_2 = \{\mathbf{Do}\texttt{send_refund}(\text{John Doe})\}$. Assume that the only individual we are interested in for our program P is John Doe. Consider S_2. The reduct of P w.r.t. S_2 consists of the rules:

$$\begin{aligned}
\mathbf{F}\texttt{send_refund}(\text{John Doe}) &\leftarrow \mathbf{Do}\texttt{run_audit}(\text{John Doe}),\\
&\quad \neg\texttt{in}(\text{John Doe}, \texttt{taxdb} : \texttt{refund_authorized}()).\\
\mathbf{Do}\texttt{send_refund}(\text{John Doe}) &\leftarrow \texttt{in}(\text{John Doe}, \texttt{taxdb} : \texttt{refund_authorized}()).
\end{aligned}$$

It is easy to see that the reduct has a unique rational status set, viz. S_2 itself.

Now consider S_1. The reduct of P w.r.t. S_1 consists of just the first rule above, from which we cannot derive $\mathbf{F}\texttt{send_refund}(\text{John Doe})$) because the body of that rule is not true w.r.t. the agent state. Thus, in contrast to the rational status set semantics, the reasonable status set semantics eliminates this unintuitive rational status set, sending John Doe his (well deserved) refund check. □

A more simplistic example is presented below.

Example 5.10 *For the program $\mathcal{P}$:*

$$\mathbf{Do}\beta \leftarrow \neg\mathbf{Do}\alpha,$$

the reduct of $\mathcal{P}$ w.r.t. $S = \{\mathbf{Do}\beta, \mathbf{P}\beta\}$ on agent state $\mathcal{O}_\mathcal{S}$ is the program

$$\mathbf{Do}\beta \leftarrow .$$

Clearly, S is the unique reasonable status set of $red^S(\mathcal{P}, \mathcal{O}_\mathcal{S})$, and hence S is a reasonable status set of $\mathcal{P}$. □

The use of reasonable status sets has also some benefits with respect to knowledge representation. For example, the rule

$$\mathbf{Do}\alpha \leftarrow \neg\mathbf{F}\alpha \tag{6}$$

intuitively expresses that action α is executed by default, unless it is explicitly forbidden (provided, of course, that its precondition succeeds). This default representation is possible because under the reasonable status set approach, the rule itself can not be used to derive $\mathbf{F}\alpha$, which is inappropriate for a default rule.

This benefit does not accrue when using rational status sets because the single rule has two rational status sets: $S_1 = \{\mathbf{Do}\alpha, \mathbf{P}\alpha\}$ and $S_2 = \{\mathbf{F}\alpha\}$. If we adopt reasonable status sets, however, then only S_1 remains and α is executed. If rational status sets are used, then the decision about whether α is executed depends on the choice between S_1 and S_2. (Notice that if the agent would execute those actions α such that $\mathbf{Do}\alpha$ appears in all rational status sets, then no action is taken here. However, such an approach is not meaningful in general, and will lead to conflicts with integrity constraints.)

The definition of reasonable status sets does not introduce a completely orthogonal type of status set. Rather, it prunes among the rational status sets. This is shown by the following property.

Proposition 5.6 *Let $\mathcal{P}$ be an agent program and $\mathcal{O}_\mathcal{S}$ an agent state. Then, every reasonable status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ is a rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$.*

In a follow-up paper [31], we are developing implementation techniques for agent programs that implement a syntactically restricted class of agent programs called *regular agent programs* that are guaranteed to have at least one reasonable status set. Existence of reasonable status sets cannot always be guaranteed because (as we have seen) some programs may have no feasible status sets.

5.4 Violating obligations: weak rational status sets

So far, we have adopted a semantics of agent programs which followed the principle that actions which the agent is obliged to take are actually executed, i.e., the rule

If $\mathbf{O}\alpha$ is true, then $\mathbf{Do}\alpha$ is true

is strictly obeyed. This is known as *regimentation* [54], and reflects the ideal behavior of an agent in a normative system.

However, the essence of deontism is in capturing what *should* be done in a specific situation, rather than what *finally is to be* done under any circumstances [3, 70, 47]. Taking this view, the operator $\mathbf{O}\alpha$ is a suggestion for what should be done; it may be well the case, that in a situation an obligation $\mathbf{O}\alpha$ is true, but α is not executed as it would be impossible (due to a violation of some action constraints), or lead to inconsistency. Such a behavior, e.g. in the legal domain, is a violation of a normative codex, which will be sanctioned in some way.

Example 5.11 (conflicting obligations) Suppose an agent A is obliged to serve requests of other agents A_1 and A_2, represented by facts $\mathbf{O}(\mathtt{serve}(A_1))$ and $\mathbf{O}(\mathtt{serve}(A_2))$, respectively, but there is an action constraint which states that no two service requests can be satisfied simultaneously. This scenario is described by the program $\mathcal{P}$:

$$\begin{aligned}\mathbf{O}(\mathtt{serve}(A_1)) &\leftarrow\\ \mathbf{O}(\mathtt{serve}(A_2)) &\leftarrow\end{aligned}$$

and the action constraint

$$AC: \quad \{\mathtt{serve}(A_1), \mathtt{serve}(A_2)\} \hookleftarrow \mathrm{true}.$$

The program $\mathcal{P}$ has no rational status set (and even no feasible status set exists). The reason is that not both obligations can be followed without raising an inconsistency, given by a violation of the action constraint AC. □

Thus, in the above example, the program is inconsistent and the agent does not take any action. In reality, however, we would expect that the agent serves at least one of the requests, thus only violating one of the obligations. The issue of which request the agent should select for service may depend on additional information – e.g., priority information, or penalties for each of the requests. In absence of any further directives, however, the agent may arbitrarily choose one of the requests.

This example and the sketched desired behavior of the agent prompts us to introduce another generalization of our approach, to a semantics for agent programs which takes into account possible violations of the rule

If $\mathbf{O}\alpha$ *is true, then* $\mathbf{Do}\alpha$ *is true*

in order to reach a consistent status set. An important issue at this point is which obligations an agent may violate, and how to proceed if different alternatives exist. We assume in the following that no additional information about obligations and their violations is given, and develop our approach on this basis. Weak rational status sets introduced below allow obligations to be "dropped" when conflicts arise. Later, Section 5.7 discusses how to build more complex structures involving cost/benefit information on top of weak rational status sets.

Our intent is to generalize the rational status set approach gracefully, and similarly the reasonable status set approach. That is, in the case where a program $\mathcal{P}$ has a rational status set on an agent state $\mathcal{O}_\mathcal{S}$, then this status set (resp., the collection of all such status sets) should be the meaning of the program. On the other hand, if no rational status set exists, then we are looking for possible violations of obligations which make it possible to have such a status set. In this step, we apply Occam's Razor and violate the set of obligations as little as possible; i.e., we adopt a status set S which is rational, if a set Ob of rules $\mathbf{O}\alpha \Rightarrow \mathbf{Do}\alpha$ is disposed, and such that no similar status set S' for some disposal set Ob' exists which is a proper subset of Ob. We formalize this intuition next in the concept of weak rational (resp., reasonable) status set.

Definition 5.9 (relativized action closure) Let S be a status set, and let A be a set of ground actions. Then, the action closure of S under regimentation relativized to A, denoted $ACl_A(S)$, is the closure of S under the rules

$\mathbf{O}\alpha \in S \Rightarrow \mathbf{Do}\alpha \in S$, for any ground action $\alpha \in A$

$\mathbf{Do}\beta \in S \Rightarrow \mathbf{P}\beta \in S$, for any ground action β.

A set S is action closed under regimentation relativized to A, if $S = ACl_A(S)$ holds. □

The following example illustrates this definition.

Example 5.12 Suppose we have:

$$\begin{aligned} A_1 &= \{\alpha, \gamma\} \\ A_2 &= \{\beta\} \\ S &= \{\mathbf{O}\alpha, \mathbf{O}\beta, \mathbf{Do}\gamma\}. \end{aligned}$$

Then the action closure of S under regimentation relativized to A_1 is given by:

$$ACl_{A_1}(S) = S \cup \{\mathbf{Do}\alpha, \mathbf{P}\alpha, \mathbf{P}\gamma\}.$$

Note that $ACl_{A_1}(S)$ is constructed by only examining obligations of actions in A_1 (in particular, action β is not considered), and closing S under the two closure rules in the preceding definition.

On the other hand, the action closure of S under regimentation relativized to A_2 is given by:

$$ACl_{A_2}(S) = S \cup \{\mathbf{Do}\beta, \mathbf{P}\beta, \mathbf{P}\gamma\}.$$ □

Notice that $ACl = ACl_{GA}$, where GA is the set of all ground actions. Using the concept of relativized action closure, we define weak versions of feasible (resp., rational, reasonable) status sets.

Definition 5.10 (relativized status sets) Let $\mathcal{P}$ be a program, let $\mathcal{O}_S$ be an agent state, and let A be a set of ground actions. Then, a status set S is A-feasible (resp., A-rational, A-reasonable), if S satisfies the condition of feasible (resp., rational, reasonable) status set, where the action closure ACl is replaced by the relativized action closure ACl_A (but DCl remains unchanged). □

Definition 5.11 (weak rational, reasonable status sets) A status set S is weak rational (resp., weak reasonable), if there exists an A such that S is A-rational (resp., A-reasonable) and there are no $A' \neq A$ and S' such that $A \subseteq A'$ and S' is an A'-rational (resp., A'-reasonable) status set. □

Observe that for a positive program, the set S' which witnesses that S is not weak rational (resp., weak reasonable) must be a superset of S (this follows from Theorem 5.9 below). However, S' and S are in general unrelated for arbitrary programs, since due to negation in rule bodies, following more obligations may block the application of previously firing rules.

An immediate consequence of the previous definition is the following.

Corollary 5.7 *Let $\mathcal{P}$ be an agent program. If $\mathcal{P}$ has a rational (resp., reasonable) status set on an agent state $\mathcal{O}_S$, then the weak rational (resp., weak reasonable) status sets of $\mathcal{P}$ on $\mathcal{O}_S$ coincide with the rational (resp., reasonable) status sets of $\mathcal{P}$ on $\mathcal{O}_S$.*

Thus, the concept of weak rational (resp., reasonable) status set is a conservative extension of rational (resp., reasonable) status set as desired.

Example 5.13 (conflicting obligations – continued) The program $\mathcal{P}$ has two weak rational status sets, namely

$W_1 = \{\mathbf{O}(\mathtt{serve}(A_1)),\ \mathbf{O}(\mathtt{serve}(A_2)),\ \mathbf{P}(\mathtt{serve}(A_1)),\ \mathbf{P}(\mathtt{serve}(A_2)),\ \mathbf{Do}(\mathtt{serve}(A_1)\}$,
$W_2 = \{\mathbf{O}(\mathtt{serve}(A_1)),\ \mathbf{O}(\mathtt{serve}(A_2)),\ \mathbf{P}(\mathtt{serve}(A_1)),\ \mathbf{P}(\mathtt{serve}(A_2)),\ \mathbf{Do}(\mathtt{serve}(A_2)\}$.

The set W_1 is a $\{\mathtt{serve}(A_1))\}$-rational status set, while symmetrically W_2 is a $\{\mathtt{serve}(A_2))\}$-rational status set. Both W_1 and W_2 are also weak reasonable status sets of $\mathcal{P}$. □

As the previous example shows, even a positive agent program may have more than one weak rational status set. Moreover, in other scenarios, no weak rational status set exists. To cure the first problem, one could impose a total preference ordering on the weak rational status sets. The second problem needs a more sophisticated treatment which is not straightforward; after all, the presence of some conflicts which can not be avoided by violating obligations indicates that there is a major problem, and we must question whether the agent program $\mathcal{P}$ is properly stated by the individual describing the agent.

5.4.1 Characterization of weak rational status sets

By generalizing the definitions in Section 5.2, it is possible to characterize the weak rational status sets of a positive agent program $\mathcal{P}$ using a fixpoint operator.

Definition 5.12 (operator $T_{\mathcal{P},\mathcal{O}_\mathcal{S},A}$) Suppose $\mathcal{P}$ is an agent program, $\mathcal{O}_\mathcal{S}$ an agent state, and A is a set of ground actions. Then, for any status set S,

$$T_{\mathcal{P},\mathcal{O}_\mathcal{S},A}(S) = App_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S) \cup DCl(S) \cup ACl_A(S).$$ □

Note that with respect to $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S)$, the action closure ACl is replaced by the relatived action closure ACl_A; however, $DCl(S)$ may not be dropped, since $DCl(S) \not\subseteq ACl_A(S)$ in general.

Clearly, also $T_{\mathcal{P},\mathcal{O}_\mathcal{S},A}$ is monotone and continuous if $\mathcal{P}$ is positive, and hence has a least fixpoint $lfp(T_{\mathcal{P},\mathcal{O}_\mathcal{S},A}) = \bigcup_{i=0}^{\infty} T^i_{\mathcal{P},\mathcal{O}_\mathcal{S},A}$ where $T^0_{\mathcal{P},\mathcal{O}_\mathcal{S},A} = \emptyset$ and $T^{i+1}_{\mathcal{P},\mathcal{O}_\mathcal{S},A} = T_{\mathcal{P},\mathcal{O}_\mathcal{S},A}(T^i_{\mathcal{P},\mathcal{O}_\mathcal{S},A})$, for all $i \geq 0$.

The following characterization of A-rational status sets is then obtained.

Theorem 5.8 *Let $\mathcal{P}$ be a positive agent program, let A be a set of ground actions, and let $\mathcal{O}_\mathcal{S}$ be an agent state. Then, a status set S is an A-rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}C$, if and only if $S = lfp(T_{\mathcal{P},\mathcal{O}_\mathcal{S},A})$ and S is an A-feasible status set.*

From the previous theorem, we obtain the following result.

Theorem 5.9 *Let $\mathcal{P}$ be a positive agent program, and let $\mathcal{O}_\mathcal{S}$ be an agent state. Then, a status set S is a weak rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, if and only if $S = lfp(T_{\mathcal{P},\mathcal{O}_\mathcal{S},A})$ and S is A-feasible for some maximal A w.r.t. inclusion.*

In general, this criterion does not enable efficient recognition of a weak rational status set (which is, in fact, intractable). The status set A for S in the theorem is unique, and can be detailed as follows.

Definition 5.13 ($A(S)$) For any status set S, denote $A(S) = \mathbf{Do}(S) \cup \{\alpha \mid \alpha \notin \mathbf{O}(S)\}$. □

Proposition 5.10 *Let $\mathcal{P}$ be any agent program, and let $\mathcal{O}_\mathcal{S}$ be any agent state. Suppose a status set S is A-feasible for some A. Then, S is $A(S)$-feasible, and $A \subseteq A(S)$, i.e., $A(S)$ is the unique maximal set of ground actions A such that S is A-feasible.*

Thus, if S is a weak rational status set, then $A = A(S)$ is the unique maximal set such that S is A-feasible. From Theorem 5.9, we obtain the following corollary.

Corollary 5.11 *Let S_1, S_2 be weak rational status set of a positive agent program $\mathcal{P}$ on an agent state $\mathcal{O}_\mathcal{S}$. Then, $\mathbf{O}(S_1) = \mathbf{O}(S_2)$ implies $S_1 = S_2$.*

As a consequence, for every choice of a maximal set of obligations which can be obeyed, the resulting weak rational status set is uniquely determined, if $\mathcal{P}$ is positive. This means that the commitment to a set of obligations does not introduce further ambiguities about the status of actions, which is a desired feature of the semantics.

It is easy to see that the operator $T_{\mathcal{P},\mathcal{O}_\mathcal{S},A}$ is monotone in A, i.e., enjoys the following property.

Proposition 5.12 *Let $\mathcal{P}$ be a positive agent program, let $\mathcal{O}_\mathcal{S}$ be an agent state, and let A_1, A_2 be sets of ground actions such that $A_1 \subseteq A_2$. Then, for any status set S, $T_{\mathcal{P},\mathcal{O}_\mathcal{S},A_1}(S) \subseteq T_{\mathcal{P},\mathcal{O}_\mathcal{S},A_2}(S)$ holds, and $lfp(T_{\mathcal{P},\mathcal{O}_\mathcal{S},A_1}) \subseteq lfp(T_{\mathcal{P},\mathcal{O}_\mathcal{S},A_2})$.*

For the case where no integrity constraints are present, we obtain the following result from Theorem 5.9 and Proposition 5.10.

Theorem 5.13 *Let $\mathcal{P}$ be a positive agent program, where $\mathcal{IC} = \emptyset$, and let $\mathcal{O}_\mathcal{S}$ be an agent state. Then, a status set S is a weak rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, if and only if (i) $S = lfp(T_{\mathcal{P},\mathcal{O}_\mathcal{S},A})$ and S is A-feasible for $A = A(S)$, and (ii) for each ground action $\alpha \notin A(S)$, the status set $S_{A'} = lfp(T_{\mathcal{P},\mathcal{O}_\mathcal{S},A'})$ is not A'-feasible, where $A' = A(S) \cup \{\alpha\}$.*

For a fixed program $\mathcal{P}$, this criterion implies under suitable assumptions a polynomial time algorithm for the recognition of a weak rational status set in this case. Moreover, deciding whether some weak rational status set exists and actually computing one is then possible in polynomial time. This is considered in Part II of this series of papers.

5.5 Expressing action constraints in an agent program

As we have mentioned above, action constraints do not add to the expressive power of our framework, and provide syntactic sugar which is however convenient. We discuss this now a little more in detail.

For every action constraint of the form

$$\{\alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)\} \hookleftarrow \chi \tag{7}$$

in $\mathcal{AC}$, include in the program $\mathcal{P}$ the clause

$$\mathbf{F}(\mathtt{nil}) \leftarrow \mathbf{Do}(\alpha_1(\vec{X}_1)), \mathbf{Do}(\alpha_2(\vec{X}_2)), \ldots, \mathbf{Do}(\alpha_k(\vec{X}_k)), \chi$$

where `nil` is a distinguished new action which has no preconditions and empty add and delete set; moreover, include in $\mathcal{P}$ the rule

$$\mathbf{P}(\mathtt{nil}) \leftarrow .$$

Let $\mathcal{P}^*(\mathcal{AC})$ be the resulting program. Then, the following property can be established.

Proposition 5.14 *Let $\mathcal{P}$ be an agent program, given an action base $\mathcal{AB}$, action constraints $\mathcal{AC}$, and integrity constraints $\mathcal{IC}$, and let $\mathcal{O}_\mathcal{S}$ be an agent state. Then, the rational (resp., reasonable) status sets of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ correspond 1-1 to the rational (resp., reasonable) status sets of $\mathcal{P}^*(\mathcal{AC})$ on $\mathcal{O}_\mathcal{S}$ for action base $\mathcal{AB}^* = \mathcal{AB} \cup \{\texttt{nil}\}$, $\mathcal{AC}^* = \emptyset$, and $\mathcal{IC}^* = \mathcal{IC}$.*

For feasible sets, a similar correspondence (but not 1-1) exists. Therefore, we can always eliminate action constraints by introducing new rules in the program.

In the definition of weak action execution, a possible overlap of the add set $Add(\alpha)$ and the delete set $Del(\beta)$ of two actions which should be executed is ignored. If the programmer feels unpleasant with this situation, then (s)he may add rules to the program which take care of such a check. Namely, we add the rules

$$\mathbf{F}(\texttt{nil}) \leftarrow CC_\alpha, CC_\beta, \mathbf{Do}\alpha, \mathbf{Do}\beta$$

for all $CC_\alpha \in Add(\alpha)$ and $CC_\beta \in Del(\beta)$, where `nil` is the distinguished action from above. Then, the joint execution of α and β is prohibited. In a similar way, we can add rules for nonground actions $\alpha(\vec{X}_1)$ and $\beta(\vec{X}_2)$.

Having to add such overlap rules in the program comes at the benefit of higher flexibility and better control of overlap checking, which also leads to faster computation in general. We think that this is a big advantage which outweighs the stricter definition.

Analogously, this applies to more complex action constraints than joint action executability. If, for example, execution of an action β requires execution of another action α, then we can add rules

$$\begin{array}{rcl} \mathbf{Do}(\beta) & \leftarrow & \mathbf{Do}\alpha, \\ \mathbf{F}(\texttt{nil}) & \leftarrow & \mathbf{Do}\alpha, \neg\mathbf{Do}\beta. \end{array}$$

Then, for every rational (resp., reasonable) status set S, it holds that $\mathbf{Do}\beta \in S$ iff $\mathbf{Do}\alpha \in S$. In a similar fashion, more complex action constraints can be emulated.

5.6 Preferred and Complete Status Sets

In this section, we study what happens when we consider three classes of rational status sets.

- A rational status set S is F-preferred if there is no other rational status set whose set of forbidden atoms is a strict subset of S's set of forbidden atoms. Intuitively, such status sets are *permissive* – most things are allowed unless explicitly forbidden.

- A rational status set S is P-preferred if there is no other rational status set whose set of permitted atoms is a strict subset of S's set of permitted atoms. Intuitively, such status sets are *dictatorial* – most things are allowed unless explicitly permitted.

- The notion of a status set does not insist that for each action α, either $\mathbf{P}\alpha$ or $\mathbf{F}\alpha$ be in S. However, for any action α, either α must be permitted or must be forbidden. Complete status sets insist that this additional condition be satisfied.

5.6.1 Preference

As we have briefly mentioned in the previous section, it may be desirable to use a preference policy for cutting down of status sets. In particular, the issue whether an action should be considered forbidden or allowed is highly relevant.

It appears that there is no straightforward solution to this problem, and that in fact different approaches to using defaults are plausible. For example, the following two are suggestive:

- (weak preference) The first approach takes the view that an action should, if possible, be considered as being not forbidden. According to this view, action sets are preferred in which the part of forbidden actions is small. Note that due to the three-valued nature of the status of an action in an action set (which can be forbidden, permitted, or neither), this does not necessarily mean that the part of explicitly permitted actions in a preferred action set is large. This policy is thus a weak default about the status of an action.

- (strong preference) Another approach is to enforce a deontic completion about whether actions are permission or forbidden, and to request that in an action set, every action is either forbidden or permitted, and such that permission is preferred over forbiddance. This approach requires a redefinition of the notion of a grounded consistent action set, however (keep the permission and forbidden-parts fixed). It amounts to a kind of strong default rule that actions which are not forbidden are explicitly permitted.

These two approaches aim at treating forbidden actions. Of course, one could foster approaches which symmetrically aim at permitted actions, and implement (weak or strong) default rules about such actions. Likewise, default rules for other status operators may be designed. Which approach is in fact appropriate, or even a mixed use of different default for different actions, may depend on the particular application domain. In the following, we take a closer look to weak defaults rules on forbidden actions.

It would be useful if rules like

$$\mathbf{Do}(\alpha) \leftarrow \neg\mathbf{F}\alpha$$

may be stated in an agent program, with the intuitive reading that action α is executed by default, unless it is explicitly forbidden (provided, of course, that its precondition succeeds).

This single rule has two feasible status sets which are grounded: $A_1 = \{\mathbf{Do}(\alpha), \mathbf{P}\alpha\}$ and $A_2 = \{\mathbf{F}\alpha\}$. Under the policy that by default, actions which are not explicitly forbidden are considered to be permitted, A_1 is preferred over A_2 and α is executed. If no such default policy is taken, then no set is preferred over the other, and it depends on the choice between A_1 and A_2, whether α is executed. (If the agent executes those actions α such that $\mathbf{Do}(\alpha)$ appears in all rational status sets, then no action is taken here.) Adopting the view that actions should not be considered forbidden unless explicitly stated motivates the following definition.

Definition 5.14 (F-preference) A set S of action status atoms is *F-preferred*, if S is a rational status set, and there exists no other rational status set S' which has a smaller forbidden part than S, i.e., $\mathbf{F}(S') \subset \mathbf{F}(S)$ holds. □

Example 5.14 For the single rule program

$$\mathbf{Do}(\alpha) \leftarrow \neg\mathbf{F}\alpha$$

from above, the set $A_1 = \{\mathbf{Do}(\alpha), \mathbf{P}\alpha\}$ is the unique F-preferred status set of $\mathcal{P}$.

On the other hand, the rule

$$\mathbf{Do}(\alpha) \leftarrow \mathbf{P}\alpha$$

has a unique F-preferred status set, which is the empty set. Assuming by weak default that α is not forbidden, we can not conclude $\mathbf{Do}(\alpha)$, though, since an assumption $\mathbf{P}\alpha$ is not supported. □

Dual to F-preference, we can define preference for P. Intuitively, F-minimality amounts to a "brave" principle from the view of action permission, while P-minimality amounts to a "cautious" one. Both F- and P-minimality are the extremal instances of a more general preference scheme, which allows to put individual preference on each action α from the action base.

5.6.2 Complete Status Sets

As we have encountered in the examples above, it may happen that a feasible status set leaves the issue of whether some action α is permitted or forbidden open.

It may be desirable, however, that this issue is resolved in a status set which is acceptable for the user; that is, either $\mathbf{P}\alpha$ or $\mathbf{F}\alpha$ is contained in the status set. This may apply to some particular actions α, as well as to all actions in the extremal case.

Our framework for agent programs is rich enough to handle this issue in the language in a natural, simple way. Namely, by including a rule

$$\mathbf{F}\alpha \leftarrow \neg\mathbf{P}\alpha$$

in a program, we can ensure that every feasible (and thus rational) status set includes either $\mathbf{F}\alpha$ or $\mathbf{P}\alpha$; we call this rule the *F/P-completion rule of α*. For an agent program $\mathcal{P}$, we denote by $Comp_{F/P}(\mathcal{P})$ the augmentation of $\mathcal{P}$ by the F/P-completion rules for all actions α in the action base.

Call a status set S *F/P-complete*, if for every ground action α, either $\mathbf{P}\alpha \in S$, or $\mathbf{F}\alpha \in S$.

Then, we have the following immediate property.

Proposition 5.15 *Let $\mathcal{P}$ be an agent program. Then, every feasible status set S of $Comp_{F/P}(\mathcal{P})$ is F/P-complete.*

Example 5.15 The program

$$\begin{aligned} \mathbf{P}\alpha &\leftarrow \\ \mathbf{F}\alpha &\leftarrow \mathbf{P}\beta \\ \mathbf{F}\alpha &\leftarrow \mathbf{F}\beta \end{aligned}$$

has a unique rational status set. However, the program $Comp_{F/P}(\mathcal{P})$ has no feasible status set, and thus also no rational status set.

This is intuitive, if we adopt the view that the status of each action being permitted or forbidden is complete, since there is no way to adopt either $\mathbf{P}\beta$ or $\mathbf{F}\beta$ without raising an inconsistency. □

Example 5.16 Consider the program $\mathcal{P}_1$:

$$\mathbf{Do}(\alpha) \leftarrow \neg\mathbf{F}\alpha.$$

Here, we have two rational status sets, namely $S_1 = \{\mathbf{Do}(\alpha), \mathbf{P}\alpha\}$ and $S_2 = \{\mathbf{F}\alpha\}$. Both are F/P-complete, and are the rational status sets of $Comp_{P/F}$.

On the other hand, the program $\mathcal{P}_2$:

$$\mathbf{Do}(\alpha) \leftarrow \mathbf{P}\alpha,$$

has the unique rational status set $S = \{\}$, while its F/P-completion has the two rational status sets S_1 and S_2 from above. Thus, under F/P-completion semantics, the programs $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent. □

In fact, the following property holds.

Proposition 5.16 *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be ground agent programs and $\mathcal{O}_\mathcal{S}$ a state, such that $\mathcal{P}_2$ results by replacing in $\mathcal{P}_1$ any literals $\pm Op\alpha$ in rules bodies by $\mp\overline{Op}\alpha$, where $Op \in \{P, F\}$ and $\overline{Op}$ is the deontic status opposite to Op. Then, $Comp_{P/F}(\mathcal{P}_1)$ and $Comp_{P/F}(\mathcal{P}_2)$ have the same sets of feasible status sets.*

Hence, under F/P-completion, $\neg F$ amounts to P and similarly $\neg P$ to F.

Further completion rules can be used to reach a complete state on other status information as well. For example, a completion with respect to obligation/waiving can be reached by means of rules

$$\mathbf{W}\alpha \leftarrow \neg\mathbf{O}\alpha$$

for actions α. Such completion rules are in fact necessary, in order to ensure that the rational status sets can be completed to a two-valued deontic "model" of the program. Applying F/P-completion does not suffice for this purpose, as shown by the following example.

Example 5.17 Consider the program $\mathcal{P}$:

$$\begin{aligned} \mathbf{P}\alpha &\leftarrow \\ \mathbf{F}\alpha &\leftarrow \mathbf{O}\beta \\ \mathbf{F}\alpha &\leftarrow \mathbf{W}\beta \end{aligned}$$

The set $S = \{\mathbf{P}\alpha, \mathbf{P}\beta\}$ is a feasible status of $Comp_{F/P}(\mathcal{P})$. However, S can not be completed to a deontic model of $\mathcal{P}$, in which $\mathbf{O}\beta$ and $\mathbf{W}\alpha$ are true or false, respectively, and such that the deontic axiom $\mathbf{W}\alpha \leftrightarrow \neg\mathbf{O}\alpha$ is satisfied. □

5.7 Optimal Status Sets

Thus far, we have discussed the following semantics for agent programs: feasible status sets, rational status sets, reasonable status sets, weak rational status sets, F-preferential status sets, P-preferential status sets, and complete status sets. Let Sem be a variable over any of these semantics. Sem chooses certain feasible status sets in keeping with the philosophical and epistemic principles underlying Sem.

However, in the real world, many choices are made based on the cost of a certain course of action, as well as the benefits gained by adopting that course of action. For example in the case of an automated Purchasing agent for a company, it is quite likely that Suppliers 1 and 2 charge two different prices for the item that the Plant Agent wishes to order. Furthermore, as the two suppliers are likely to be located at different locations, transportation costs are also likely to vary. If one supplier can supply the entire quantity required, the Purchasing Agent will in all likelihood, select the one whose total cost (cost of items plus

transportation) is lower. Note that this cost is being described in terms of the costs of the actions being executed in a status set.

However, yet another parameter that needs to be taken into account is the desirability of the final state that results by executing the **Do**-actions in a Sem-status set. For example, the time at which the supplies will arrive at the company is certainly pertinent, but is not accounted for by the cost parameters listed above. If Supplier 2 will provide the supplies one day before Supplier 1, then the Purchasing Agent may well choose to go with Supplier 1, even if Supplier 2's overall cost is lower.

What the preceding discussion tells us is that we would like to associate with any Sem-status set, a notion of a cost, and that this cost must take into account, the set of **Do**-status atoms in the status set, and the final state that results. This motivates our definition of a cost function.

Definition 5.15 (cost function) Suppose $\mathcal{S} = (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S})$ is a body of software code, and $States$ is the set of all possible states associated with this body of code. Let $\mathcal{AB}$ be the set of all actions. A *cost function*, cf, is a mapping from $(States \times 2^{\mathcal{AB}})$ to the non-negative real numbers such that:

$$[(\forall s_1, s_2)\,(\forall A)\,\mathsf{cf}(s_1, A) = \mathsf{cf}(s_2, A)] \rightarrow [(\forall s)\,(\forall A, A')\,(A \subseteq A' \rightarrow \mathsf{cf}(s, A) \leq \mathsf{cf}(s, A'))]\,. \qquad \square$$

The precondition of the above implication basically reflects state independence. A cost function is *state-independent* iff for any set A of actions, and any two arbitrarily chosen states s_1, s_2, the cost function returns the same value for $\mathsf{cf}(s_1, A)$ and $\mathsf{cf}(s_2, A)$. State-independence implies that the cost function's values are only affected by the actions taken, i.e. by the set of actions A.

The above axiom says that for cf to be a cost function, if it is state-independent, then the values it returns must monotonically increase as the set of actions is enlarged (i.e. as more actions are taken).

One might wonder whether cost functions should satisfy the stronger condition:

$$(*) \quad (\forall s)(\forall A, A').\, A \subseteq A' \rightarrow \mathsf{cf}(s, A) \leq \mathsf{cf}(s, A').$$

The answer is "no" – to see why, consider the situation where executing the actions in A is cheaper than executing the actions in A', but this is offset by the fact that the state obtained by executing the actions in A' is less desirable than the state obtained by executing the actions in A.

Alternatively, one might wonder whether cost functions should satisfy the condition:

$$(**) \quad (\forall s_1, s_2)(\forall A).\, s_1 \subseteq s_2 \rightarrow \mathsf{cf}(s_1, A) \leq \mathsf{cf}(s_2, A).$$

Again, the answer is no. Executing all actions in A in state s_1 may lead to a more desirable state than doing so in state s_2. As an example on the lighter side, consider the action `enter(room)`. State s_1 is empty, state $s_2 = \{$ `in(room,python)` $\}$. Clearly, $s_1 \subseteq s_2$. For most of us, executing the action `enter(room)` in state s_1 is vastly preferable to executing the action `enter(room)` in state s_2.

However, even though not all cost functions should be required to satisfy $(*)$ and $(**)$, there will certainly be applications where either $(*)$ and/or $(**)$ are satisfied. In such cases, it may turn out to be beneficial computationally to take advantage of properties $(*)$ and $(**)$ when computing optimal Sem-status sets defined below.

Definition 5.16 (weak/strong monotonic cost functions) A cost function is said to be weakly monotonic, if it satisfies condition $(*)$ above. It is strongly monotonic, if its satisfies both conditions $(*)$ and $(**)$. $\square$

We are now ready to come to the definition of optimal status sets.

Definition 5.17 (Optimal Sem-status set) Suppose $\mathcal{S} = (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S})$ is a body of software code, and $\mathcal{O}_\mathcal{S}$ is the current state. A Sem-status set X is said to be *optimal with respect to cost function* cf iff there is no other Sem-status set Y such that

$$\mathsf{cf}(\mathcal{O}_\mathcal{S}, \{\mathbf{Do}\alpha \mid \mathbf{Do}\alpha \in Y\}) < \mathsf{cf}(\mathcal{O}_\mathcal{S}, \{\mathbf{Do}\alpha \mid \mathbf{Do}\alpha \in X\}). \quad \square$$

Note that the above definition induces different notions of status set, depending on what Sem is taken to be.

5.8 Feasible status sets and models of logic programs

In this subsection, we describe a transformation AG that takes as input, a logic program P, and produces as output:

- An action base, all of whose actions have an empty precondition, add list and delete set,
- An agent program $\mathsf{AG}(P)$,
- An empty set of action constraints and an empty set of integrity constraints.

As all components other than the agent program produced by $\mathsf{AG}(P)$ are empty, we will abuse notation slightly and use $\mathsf{AG}(P)$ to denote the agent program produced by AG.

For each ground instance of a rule r in P of the form

$$a \leftarrow b_1, \cdots, b_m, \neg c_1, \ldots, \neg c_n$$

insert the rule

$$\mathbf{P}(a) \quad \leftarrow \quad \mathbf{P}(b_1), \cdots, \mathbf{P}(b_m), \neg\mathbf{P}(c_1), \ldots, \neg\mathbf{P}(c_n) \tag{8}$$

in $\mathsf{AG}(P)$. Here, the atoms a, b_i, and c_j of P are viewed as actions with description $(\emptyset, \emptyset, \emptyset)$, i.e., they have no precondition and their add and delete sets are both empty. It is important to note that the only types of status atoms that occur in $\mathsf{AG}(P)$ are of the form $\mathbf{P}(-)$.

Example 5.18 Consider the logic program containing the two rules:

$$\begin{array}{lll} a & \leftarrow & \\ b & \leftarrow & a, \neg c. \end{array}$$

The $\mathsf{AG}(P)$ is the agent program:

$$\begin{array}{lll} \mathbf{P}(a) & \leftarrow & \\ \mathbf{P}(b) & \leftarrow & \mathbf{P}(a), \neg\mathbf{P}(c). \end{array}$$

We observe that the logic program has three models. These are given by:

$$\begin{array}{lll} M_1 & = & \{a, b\} \\ M_2 & = & \{a, c\} \\ M_3 & = & \{a, b, c\} \end{array}$$

$\mathsf{AG}(P)$ happens to have more than three feasible status sets. These are given by:

$$
\begin{aligned}
F_1 &= \{\mathbf{P}(a), \mathbf{P}(b)\}.\\
F_2 &= \{\mathbf{P}(a), \mathbf{P}(b), \mathbf{Do}(a)\}.\\
F_3 &= \{\mathbf{P}(a), \mathbf{P}(b), \mathbf{Do}(b)\}.\\
F_4 &= \{\mathbf{P}(a), \mathbf{P}(b), \mathbf{Do}(a), \mathbf{Do}(b)\}.\\
F_5 &= \{\mathbf{P}(a), \mathbf{P}(c)\}.\\
F_6 &= \{\mathbf{P}(a), \mathbf{P}(c), \mathbf{Do}(a)\}.\\
F_7 &= \{\mathbf{P}(a), \mathbf{P}(c), \mathbf{Do}(c)\}.\\
F_8 &= \{\mathbf{P}(a), \mathbf{P}(c), \mathbf{Do}(a), \mathbf{Do}(c)\}.\\
F_9 &= \{\mathbf{P}(a), \mathbf{P}(b), \mathbf{P}(c)\}.\\
F_{10} &= \{\mathbf{P}(a), \mathbf{P}(b), \mathbf{P}(c), \mathbf{Do}(a)\}.\\
F_{11} &= \{\mathbf{P}(a), \mathbf{P}(b), \mathbf{P}(c), \mathbf{Do}(b)\}.\\
F_{12} &= \{\mathbf{P}(a), \mathbf{P}(b), \mathbf{P}(c), \mathbf{Do}(c)\}.\\
F_{13} &= \{\mathbf{P}(a), \mathbf{P}(b), \mathbf{P}(c), \mathbf{Do}(a), \mathbf{Do}(b)\}.\\
F_{14} &= \{\mathbf{P}(a), \mathbf{P}(b), \mathbf{P}(c), \mathbf{Do}(a), \mathbf{Do}(c)\}.\\
F_{15} &= \{\mathbf{P}(a), \mathbf{P}(b), \mathbf{P}(c), \mathbf{Do}(b), \mathbf{Do}(c)\}.\\
F_{16} &= \{\mathbf{P}(a), \mathbf{P}(b), \mathbf{P}(c), \mathbf{Do}(a), \mathbf{Do}(b), \mathbf{Do}(c)\}.
\end{aligned}
$$

Many further feasible status sets exist, if we take atoms with the other modalities $\mathbf{F}$, $\mathbf{O}$ and $\mathbf{W}$ into account.

However, when we examine the above sixteen and all other feasible status sets, and if we ignore the $\mathbf{Do}$ atoms in them, we find only three feasible status sets, viz. F_1, F_5 and F_9. The reader will easily note that the feasible status sets F_2, F_3, F_4 reflect different ways of determining which actions that are permitted in F_1 should actually be done. The same observation holds with regard to F_5 and the feasible status sets F_6, F_7, F_8. Likewise, the feasible status sets $F_{10}, \ldots, F_{16}$ are derived from F_9 in the same way.

The reader will note that in this example, M_1, M_2, M_3 stand in one one correspondence to the projections of $F_1, \ldots, F_{16}$ with respect to the modality $\mathbf{P}$,i.e. M_1, M_2, M_3 stand in one one correspondence with F_1, F_5 and F_9. □

The following result shows, conclusively, that this is no accident.

Proposition 5.17 *There exists a 1-1 correspondence between the models of P and the $\mathbf{P}$-projection of the feasible status sets of $\mathsf{AG}(P)$, i.e.*

1. *If M is a model of the program P, then $A_M = \{\mathbf{P}(a) \mid a \in M\}$ is a feasible status set of $\mathsf{AG}(P)$.*
2. *If A is a feasible status set of $\mathsf{AG}(P)$, then $M_A = \{a \mid \mathbf{P}(a) \in A$, a occurs in $P\}$ is a model of P.*

5.9 Rational status sets and minimal models of logic programs

If we return to Example 5.18, we will notice that the logic program P shown there has two minimal Herbrand models, corresponding to M_1, M_2 respectively, and the feasible status sets, F_1, F_5 correspond to the rational status sets of $\mathsf{AG}(P)$. Intuitively, minimal Herbrand models of a logic program select models of

P that are inclusion-minimal, while rational status sets select feasible status sets that are also inclusion-minimal. As there is a 1-1 correspondence between models of P and the $\mathbf{P}$-parts of the feasible status sets of $\mathsf{AG}(P)$, it follows immediately that the inclusion minimal elements should also be in 1-1 correspondence. The following result is in fact an immediate corollary of Proposition 5.17 and establishes this 1-1 correspondence.

Proposition 5.18 *There exists a 1-1 correspondence between the minimal models of P and the rational status sets of $\mathsf{AG}(P)$, i.e.*

1. *If M is a minimal model of the program P, then $A_M = \{\mathbf{P}(a) \mid a \in M\}$ is a rational status set of $\mathcal{P}(P)$.*

2. *If A is a rational status set of $\mathsf{AG}(P)$, then $M_A = \{a \mid \mathbf{P}(a) \in A$, a occurs in $P\}$ is a minimal model of P.*

When taken in conjunction with results of Lobo and Subrahmanian [59], the above result implies that there exists a translation T (given in [59]) such that the rational status sets of $\mathsf{AG}(P)$ correspond exactly to the extensions of a pre-requisite free normal default theory $T(P)$.

5.10 Reasonable status sets and stable semantics

In this section, we show that the reasonable status sets of $\mathsf{AG}(P)$ correspond to the stable models of P. Before stating this main result formally, let us return to the case of Example 5.18.

Example 5.19 It is easy to see that the logic program P of Example 5.18 has exactly one stable model, viz. M_1. It is easy to see that $\mathsf{AG}(P)$ program has a unique reasonable status set, viz. $RS_1 = \{\mathbf{P}(a), \mathbf{P}(b)\}$. As Proposition 5.19 below will show, this is not an accident. □

The following result explicitly states this.

Proposition 5.19 *There exists a 1-1 correspondence between the stable models of P and the reasonable status sets of $\mathsf{AG}(P)$, i.e.*

1. *If M is a stable model of the program P, then $A_M = \{\mathbf{P}(a) \mid a \in M\}$ is a reasonable status set of $\mathsf{AG}(P)$.*

2. *If A is a reasonable status set of $\mathcal{P}(P)$, then $M_A = \{a \mid \mathbf{P}(a) \in A$, a occurs in $P\}$ is a stable model of P.*

It is important to observe that by this correspondence, we obtain some complexity results on reasonable status sets. This is because the complexity results known for non-monotonic logic programs with stable model semantics [44, 65] imply lower bounds for agent programs using reasonable status sets.

5.11 Discussion

Thus far, in this section, we have shown that given any logic program P, we can convert P into an agent program , $\mathsf{AG}(P)$, (together with associated action base and empty sets of integrity constraints and action constraints) such that:

1. The $\mathbf{P}$-parts of feasible status sets are in 1-1 correspondence with the models of P;
2. Rational status sets are in 1-1 correspondence with the minimal models of P;
3. Reasonable status sets are in 1-1 correspondence with the stable models of P.

The above results, when taken in conjunction with known results linking logic programs and nonmonotonic reasoning, provide connections with well known nonmonotonic logics as well. For example, the following results are well known:

- Marek and Truszczynski [65] prove 1-1 correspondences between stable models of logic programs and extensions of default logic theories.
- Marek and Subrahmanian [64] and Marek and Truszczynski [65] prove 1-1 correspondences between stable models of logic programs and appropriate types of expansions of auto-epistemic theories.
- Lobo and Subrahmanian [59] prove 1-1 correspondences between minimal models of logic programs, and extensions of prerequisite-free normal default logic theories.
- Ben-Eliyahu and Dechter [16] have proved that stable models and minimal models of logic programs may be viewed as models of a suitable logical theory.

An important topic that we have not addressed (due to space restrictions) is whether there exists a transformation $\wp$ that takes as input, an agent state, action base, an agent program, a set of integrity constraints, and a set of action constraints, and produces as output a logic program such that the above equivalences hold. This is somewhat complicated to do because the use of arbitrary agent states over arbitrary data structures means that classical model semantics, minimal model semantics, and stable semantics cannot be used directly. Rather, the notion of models over arbitrary data structures introduced by Lu *et al.* [61] must be used. For this reason, we defer this to further work.

However, we remark that for feasible and rational status sets, no 1-1 correspondence to the models and minimal models, respectively, of a polynomial-time constructible logic program similar as above is possible in general: An agent program may lack a feasible or rational status set (even in absence of integrity constraints), while a logic program always has some model and minimal model; existence of a feasible as well as a rational status set for an agent program is NP-hard, even for agent programs without integrity constraints (see Part II of this series of papers). Furthermore, since computing a model (resp., minimal model) of a logic program is in FNP (resp., $\mathrm{FNP}/\!/\log$), it is not possible to polynomially reduce the $\mathrm{F}\Sigma_2^P$-hard computation of a rational status set of a general agent program to the computation of a model (resp., minimal model) of a polynomial time-constructible logic program, unless the polynomial hierarchy collapses. In particular, no polynomial-time constructible logic program exists whose minimal models correspond 1-1 to the rational status sets of a general agent program. Observe that from the complexity side, a 1-1 correspondency between reasonable status sets of an agent program and the stable models of a polynomial-time constructible logic program is not excluded; in fact, a rather natural translation seems feasible.

6 Related Work

During the last few years, there has been an explosion in the area of agent based research. This section is divided into four parts — in the first part, we provide a detailed comparison of our work with Shoham's AGENT-0 language [91]. Then, in Section 6.2, we provide a detailed comparison between our framework and languages such as the branching time temporal logic based languages advocated by Rao and Georgeff[76]. In Section 6.3, we provide a detailed comparison between our framework and the deontic logic based framework for agent based programming proposed by Meyer's group [48, 28, 98], and compare it in Section 6.4 to approaches in the area of deontic logic programming [83, 99, 68]. Finally, in Section 6.5, we provide a comparison with other proposals for agent programming and outline directions for future work.

6.1 Shoham's AGENT-0

Shoham [91] was perhaps the first to propose an explicit programming language for agents, based on object oriented concepts, and based on the concept of an agent state. As recalled in Section 4, Shoham [91, Section 3] states that a complete AOP ("agent oriented programming") system will have three components.

1. a restricted formal language for describing mental state;
2. an interpreted agent programming language with primitive commands such as `REQUEST` and `INFORM`; and,
3. an "agentifier".

We have provided in Section 4 a detailed description of how the framework described in this paper supports agentification, which Shoham did not address. This is a significant contribution of our paper, over and above Shoham's important advances.

Shoham's paper concentrated on the second problem listed above, and he developed a language called AGENT-0 to support this. In this section, we restrict our comments to the differences between our framework and AGENT-0.

In AGENT-0, actions are either *private* or *communicative*. Modalities used in AGENT-0 are `DO` and `REFRAIN`, while `INFORM` and `REQUEST` are communicative actions. The mechanism that the agent uses to act in AGENT-0 is through rules having the form

$$(\mathbf{If}\ \ \langle \text{mental-condition} \rangle\ \ \text{action})$$

where $\langle$mental-condition$\rangle$ is a Boolean combination of formulas "$\mathbf{B}\,fact$" and "$\mathbf{CMT}\ a\ action$." Here, $\mathbf{CMT}$ intuitively stands for "commit" which has the same intended meaning [91, p. 72] as our "obligatory" or $\mathbf{O}$ modality. In addition, specific syntax is used to handle messages, in much the same way as the `msgbox` domain described in this paper.

An agent programmed within AGENT-0 can be intuitively transformed into an agent program as described in this paper, through the following mechanism.

1. As all mental conditions of the agent are evaluated over the agent's mental state, create a body of software code (in the strict sense of this paper) called `mental`. Add appropriate code-calls to this mental state manager.

2. Now, every rule of the form "($\mathbf{If}$ $\langle$mental-condition$\rangle$ $\mathbf{M}\alpha$)" where α is an action and $\mathbf{M}$ is one of DO, REFRAIN, INFORM, REQUEST, can be converted into an agent program rule as follows:

$$\begin{aligned}
\mathbf{Do}\alpha &\leftarrow \text{mental-condition}, \quad \text{if } \mathbf{M} = \texttt{DO};\\
\mathbf{F}\alpha &\leftarrow \text{mental-condition}, \quad \text{if } \mathbf{M} = \texttt{REFRAIN};\\
\mathbf{Do}send(a, b, '\texttt{request}\alpha') &\leftarrow \text{mental-condition}, \quad \text{if } \mathbf{M} = \texttt{REQUEST};\\
\mathbf{Do}send(a, b, '\texttt{inform}\alpha') &\leftarrow \text{mental-condition}, \quad \text{if } \mathbf{M} = \texttt{INFORM}.
\end{aligned}$$

It is important to note that in AGENT-0 , the only modalities that can appear in the head of a rule are $\mathbf{F}$ and $\mathbf{Do}$. In contrast, agent programs allow several other modalities in the heads of rules. Furthermore, agent programs allow all the modalities $\mathbf{Do}, \mathbf{F}, \mathbf{P}, \mathbf{O}$ to appear in rule bodies — positively or negatively — which is not the case with AGENT-0 programs.

In view of the above informal embedding of AGENT-0 programs into Agent Programs as introduced in this paper, we may say that our paper builds upon Shoham's important ideas in the following ways. First, we provide a formal semantics (actually, several semantics) for agent programs, and by virtue of the above embedding, our paper also provides several alternative semantics for Shoham's AGENT-0 language which has hitherto not had a formal semantics. Second, we allow arbitrary notions of concurrency to be associated with any agent. Third, we allow integrity constraints and action constraints, thus building further upon languages like AGENT-0. Fourth, we provide a set of algorithms and complexity results for such programs in Part II of this series of papers. Fifth, we provide a concrete syntax for building agents on top of arbitrary data structure and Shoham's language of mental states can be extended easily with this syntax. In contrast, Shoham's elegant characterization of mental states may be used to define a body of software code that can be attached to any agent program as defined in this paper. Shoham also provides a specification of an interpreter for agent programs that we benefit from.

6.2 BDI Framework

It has been well recognized in AI that agents reason based on beliefs (B), their goals or desires (D), and their intentions (I). Intuitively, according to Bratman [18], intentions are plans (partial or total) that an agent has committed to perform so as to accomplish its goals or desires. An architecture centering on these themes called the *BDI-architecture* for agents was proposed by Bratman [18, 19] and by Cohen and Levesque [23]) and further developed by Rao and Georgeff [76].

The fact that the world evolves over time led BDI-researchers to model the evolution of the world through a tree. Nodes at level i in the tree reflect the different possible states of the world at time i. At any given point t in time, the agent knows its current state, but does not know how the world will evolve in the future. Thus, the reasoning of the agent is modeled through a branching time temporal logic such as Computation Tree Logic (CTL) [32]. These logics include specialized operators such as "eventually p" and "always p." In initial work on CTL and its variants, researchers assume that each node represents a state which is "objective" (I.e. represents a set of ground atoms true in the current world). Extensions of these logics have been developed to handle probabilistic data [6].

Cohen and Levesque [23] extend the generic branching time temporal logic framework as follows. Instead of assuming (as in [32]) that each state is a set of "objective" (i.e. modality free) atoms, they assume that each state is itself a time-line consisting of a sequence of events both in the past and in the future. In effect, suppose we are at node N and suppose the time-line associated with node N is a function ℓ_N. The

$\ell_N(i)$ specifies what is believed by the agent to be true at time i, given that the agent is currently at node N. This formulation allows the agent's beliefs to change over time.

Rao and Georgeff[76] generalize this further. They allow each node to contain a time-tree, instead of a time line. Thus, given a node N, they have an associated tree, $tree_N$, which specifies what the agent believes (at node N) will hold in the future. Each timetree allows the agent, once it is in the belief-state captured by node N, to reason about different possible futures, and make a choice about which of these different possible futures it wants to try to steer towards, based on its goals (desires). Rao and Georgeff [76] develop a syntax and a model theory based on this intuition, and define a set of axioms that may be used by a proof procedure for their logic. These axioms are sound, but completeness is not established.

Philosophically, our paper is very different from the BDI architecture in that we do not, in this paper, deal with planning and/or plan objects. Instead, in our paper, we provide a mechanism (Agent Programs) through which the agent can decide what actions to perform now, based on the declarative specification of the agent's behavior stated in the agent program. Thus, our framework can be used to *implicitly specify* a tree of the kinds described above in the following way: Given an agent program $\mathcal{P}$, a set $\mathcal{IC}$ of integrity constraints, a set $\mathcal{AC}$ of action constraints, and an initial agent state s_0, define a tree in the following way:

- Each node in the tree is labeled with a pair (m, s) where m is a set of messages and s is a state;
- The root of the tree is labeled with $(\{\}, s_0)$;
- Each edge in the tree is labeled with a set of actions;
- If there is an edge in the tree of the form $(m, s) \stackrel{\mathcal{A}}{\Longrightarrow} (m', s')$ then
  - there exists a Sem-status set S of $\mathcal{P}$ w.r.t. the state $s^\star$ which is just like s except that the messages m' replace m and $\mathcal{A} = \{\alpha | \mathbf{Do}\alpha \in S\}$ (where Sem is the selected semantics of agent programs);
  - the concurrent execution of the actions in $\mathcal{A}$ in state $s^\star$ leads to the state s'.

However, we do not specify how to reason about the global properties of such trees — this is well done in CTL-like languages, and we have not specified, as is done by Rao and Georgeff [76] and Cohen and Leveseque [23] how the structure of this tree may be generalized.

6.3 Hindriks' *et al.*'s Agent Programs

Hindriks *et al.* [48] also have a proposal similar to ours in spirit. In their framework, an agent's mental state consists of a set of goals and a set of beliefs. An *agent program* in their framework consists of a quadruple $(\mathcal{T}, \Pi_0, \sigma_0, \Gamma)$ where $\mathcal{T}$ is a transition function specificying the effects of basic actions, Π_0 is an initial set of goals, σ_0 is an initial set of beliefs, and Γ is a set of rules of the form

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

In general, $Head$ is a (potentially) complex formula describing a goal. The syntax of goals supported by Hindriks *et al.*[48] allows goals to be elementary actions, but also includes sequential compositions of actions, disjunctive goals, and/or conjunctive goals. The $Guard$ is a logical formula, while the $Body$ has the same structure as the head.

- The language of [48] is richer than ours in some respects — this is particularly true because complex actions can appear in the heads of rules. In contrast, in our paper, no such complex actions are considered.

- In addition, as stated in [48, p.219], a request to achieve a goal G can be encoded through a rule with G in the head and whose body is a "recipe" to accomplish G. There is no direct analog for this in our framework.
- In contrast, our language is richer than that of [48] in some other ways.
  - First, we provide a well defined, and practically realized and tested method for building agents on top of heterogeneous data structures. (For example, Schafer *et al.* [85] describe an application of our theory to build a Virtual Operations Center for the US Army's Logistics Integration Agency that computes joins across heterogeneous data sources involving *billions* of tuples.) This same syntax can be incorporated within the guards of the [48] framework to extend their approach.
  - Second, in our language, we explicitly allow deontic modalities to appear in Agent Programs, which is not allowed in the [48] framework. As a consequence, we can write rules specifying the operational principles underlying an agent's behavior and this allows an agent program to nondeterministically choose what actions to execute.
  - Third, our framework contains explicit constructs for integrity constraints and action constraints that limit the set of what states are allowed to exist, and what actions are concurrently executable. These are new contributions.
  - Fourth, once an agent has selected a status set of the appropriate type, we allow any notion of concurrency to be used to execute these actions. This could include sequential composition or parallel composition as allowed in [48]. But it could also include composing a set of actions to produce a completely "new" action not drawn from the set that reflects the amalgam of these actions being composed.
  - Fifth, by building our framework in a generic way, we have developed a hierarchy of successively more complex semantics that an agent developer can pick and choose from for his application. These semantics may be polynomial for some applications, while they may be more complex for other applications. The application should determine which semantics should be used. These complexity results are discussed in detail in Part II of this paper. In addition, some of our semantics (e.g. the optimal status set semantics) are very natural to use for many applications, and have no obvious counterparts in [48].

Our framework can benefit from that of [48]. In particular, the important idea of allowing complex actions to be directly programmed into Agent Program rules is something we are currently exploring, through the use of Hierarchical Task Network actions.

6.4 Deontic Logic Programming

Several formalizations of deontic action languages were proposed using techniques from logic programming. We consider here the approaches described in [83, 99, 68].

Ryu and Lee [83] develop a deontic system on top of Nute's [74] defeasible reasoning framework. Their language has rules

$$\phi \leftarrow \psi_1, \ldots, \psi_n \qquad\qquad \phi \Leftarrow \psi_1, \ldots, \psi_n$$

for strict and defeasible rules, respectively, where ϕ and the ψ_i are literals. Deontic operators "oblig(α)" and "forbid(α)" are applicable to actions α; taking an action is expressed by "perform(α)". Further operators on actions are available. The semantics of these operators is captured by a set of rules, which are added to the

program. Queries posed to the system are answered by evaluating the augmented program under Nute's SLD resolution-style algorithm [74]. Negation is treated by considering negative literals as defeasibly obtained, i.e., true by default.

Clearly, the system in [83] is fundamentelly different from ours. It is a PROLOG-like, goal-oriented query-answering system, rather than a pure declarative action specification language as ours. Furthermore, concurrent execution of actions, including different execution ontologies, and integrity/action constraints are not adressed.

In [99], van der Meyden proposes a system which applies techniques of logic programming to a deontic specification language. It has action and propositional constants, which can be combined to more complex formulas using standard connectives and deontic operators $\mathbf{P}$ and π, as well as further temporal operators for statements about the future to which action execution results. Intuitively, $\mathbf{P}(\alpha)$ can be read as "it is not prohibited to bring it about that α", while $\pi(\alpha)$ reads "it is permitted to do α" (i.e., executing α and further actions implied results in a permitted state). Programs consist of Horn clauses in the language. E.g.,

$$\mathbf{P}(\mathit{owns}(y,z)) \Rightarrow \mathit{transferable}(z,y)$$

expresses the predicate "z is legally transferable to y", which holds just when some action resulting in y's ownership of z is permitted [99]. For the Horn clause fragment, a Kripke-style semantics is given which using a monotone operator T singles out the least model of a program in the class of Tree Kripke models. A tableaux proof theory, sketched in [99], may be used to build a top-down interpreter in the style of SLD-resolution.

Compared to our agent programs, the framework in [99] allows for more complex statements about actions (including nested modalities), and temporal operators (resorting to dynamic logic) are availale. On the other hand, not all deontic operators are available; most noticably, van der Meyden's language lacks a concept of obligation. The system has thus the flavor of a process logic tailored for sequences of possible actions. Our status set semantics is of a different nature, aiming at instantaneous decision taking. This may involve nondeterministic choice and compositionality of actions — which is excluded in [99] but represented in our framework. Furthermore, the language of [99] disallows negation in rule bodies [99]. A simple extension by negation as failure may be possible [99], but will hardly assign a satisfactory semantics to all programs.

McCarty [68] describes a deontic action language, combining the action language in [69] and his pioneering deontic language [67], which emerged from the TAXMAN project [66]. It has deontic operators similar as ours. McCarty develops an *intuitionistic* logic for this language based on *events*, which paired with an agent represent actions. Informally, an event $\alpha(t_1, t_2)$ is a predicate which is brought about to hold between points t_1 and t_2 in time. A modal atom $\mathbf{P}\langle\alpha, c\rangle$ intuitively expresses that action α is permitted to agent c as a "free choice permission", which means that all ways to perform α are permitted to her. The reading of $\mathbf{O}\langle\alpha, c\rangle$ is as usual, while the reading of $\mathbf{F}\langle\alpha, c\rangle$ is that all ways of performing α are forbidden. An atom $\mathbf{DO}\langle\alpha, c\rangle$ expresses that α is somehow carried out by the agent c. Deontic rules take the form of Horn clauses; e.g.,

$$\mathbf{O}(\mathit{TransferDollars}(c,b,d,t), c) \Leftarrow \mathit{Month}(t)$$

informally may express that on the beginning of a month t, agent c (representing a corporation) must transfer d dollars (e.g., interests) to a bank b [68]. McCarty develops a Kripke-style semantics for his language guided by [67, 69], and explores the use of intuitionistic proof procedures for fragments of the language; a proof theory for the full language is announced for a forthcoming paper.

McCarty's language is richer than ours. It is a multi-agent modal language, allows to relate actions through logical connectives, and incoporates time. The DO operator has similar properties to ours: obligation implies (under strict regimentation) action execution, and doing an action while it is forbidden is inconsistent. However, similar as in van der Meyden's language, negation in rule bodies is treated as negation by failure, and ontologies for concurrent actions are not considered. Moreover, preference-based decision making is not discussed. Compared to our status sets semantics, the one in [68] is much more involved. For a full grasp by the agent language user, a proof theory seems indispensable. Note that, by the results established in Sections 5.8–5.10, status sets are related to other nonmonotonic languages which are well-understood [65].

6.5 Other approaches

In this section, we will briefly compare our work with wide variety of agent research.

Agent Decision Making. There has been a significant amount of work on agent decision making. Rosenschein [80] was perhaps the first to say that agents act according to states, and which actions they take are determined by rules of the form "When P is true of the state of the environment, then the agent should take action A." As the reader can easily see, our framework builds upon this intuitive idea, though (i) our notion of state is defined very generally and (ii) agent programs have a richer set of rules than those listed above. Rosenschein and Kaelbling [81] extend this framework to provide a basis for such actions in terms of situated automata theory.

Bratman *et al.* [19] define the IRMA system which uses similar ideas to generate plans. In their framework, different possible courses of actions (Plans) are generated, based on the agent's intentions. These plans are then evaluated to determine which ones are consistent and optimal with respect to achieving these intentions.

Verharen *et al.* [101] present a language-action approach to agent decision making, which has some similarities to our effort. However, they do not develop any formal semantics for their work, and their language for agent programs uses a linguistic rather than a logical approach. Schoppers and Shapiro [86] describe techniques to design agents that optimize objective functions - such objective functions are similar to the cost functions we have described.

One effort that is close to ours is Singh's approach [92]. Like us, he is concerned about heterogeneity in agents, and he develops a theory of agent interactions through workflow diagrams. Intuitively, in this framework, an agent is viewed as a finite state automaton – as is well known, finite state automata can be easily encoded in logic. This makes our framework somewhat more general than Singh's, instead of explicitly encoding automata (hard to do when an agent has hundreds of *ground* actions it can take). Sycara and Zeng [96] provide a coordinated search methodology for multiple agents. Haddadi [46] develops a declarative theory of interactions, as do Rao and Georgeff [76], and Coradeschi and Karlson [25] who build agents for air traffic simulation.

There has been extensive work on negotiation in multiagent systems, based on the initial idea of contract nets, due to Smith and Davis [93]. In this paradigm, an agent seeking a service invites bids from other agents, and selects the bid that most closely matches its own. Schwartz and Kraus [87] present a model of agent decision making where one agent invites bids (this is an action !) and others evaluate the bids (another action) and respond; this kind of behavior is encodable through agent programs together with underlying data structures. This body of work is complementary to ours: an agent negotiates by taking certain actions in accordance with its negotiation strategy, while we provide the "hooks" to include such actions within our framework, but do not explicitly study how the negotiation actions are performed, as this has been well done

by others [93, 87].

Coalition formation mechanisms where agents dynamically team up with other agents has been intensely studied by many researchers [88, 84, 106]. Determining which agents to team with is a sort of decision making capability. Inverno *et al.* [29] present a framework for dMARS based on the BDI model. Like us, they assume a state space, and the fact that actions cause state transitions. Labrou and Finin [57] develop the semantics of KQML, but do not explicitly present an action language.

Reasoning About Actions. Several works [39, 9, 10, 11] have addressed the problem of modeling the logic of actions by means of logic programming languages. In this section, we briefly address these, one by one. Gelfond and Lifschitz[39] propose a logic programming language called A using which, users may express knowledge about actions and their effects. This framework was later extended by Baral, Gelfond and others in a series of elegant papers [10, 11, 8, 7]. The language A allows users to make statements of the form

$$\begin{array}{l} f \quad \textbf{after} \quad a_1, \ldots, a_m \\ \quad \textbf{initially}\ f \\ a \ \ \textbf{causes} \ \ f \ \ \textbf{if} \ \ p_1, \ldots, p_n. \end{array}$$

Intuitively, the first statement says that executing actions $a_1, \ldots, a_m$ makes f true (afterwards). Likewise, the second statement says f was true in the initial state, and the third statement describes the effect of a on f if certain preconditions are satisfied.

The key differences between our approach and this genre of work are the following. (1) First and foremost, our approach applies to heterogeneous data sources, while this body of work assumes all data is stored in the form of logical atoms. (2) Second, the modalities for determining what is permitted, what is forbidden, what is obligatory, what is done, are not treated in the above body of work. (2) Third, in our approach, we use the semantics to determine which set of firable actions (in a state) must actually be fired, and this policy of *choosing* such sets of actions in accordance with the policies expressed in an agent program and the underlying integrity constraints is different from what is done in [39, 10, 11, 8, 7].

Collaborative Problem Solving. There has also been extensive work on collaborative problem solving and negotiation in multiagent systems (e.g., [24, 50, 53, 79, 102]). As our approach allows arbitrary decisions, and as negotiation is one form of decision making, our work provides a framework within which various negotiation strategies described in the literature can be encoded. Agents can collaborate if they wish, but again, collaboration is an explicit action, and the rules governing such collaborations can be encoded as rules within agent programs.

Agent Architectures. For an excellent anthology of classic works on agent systems, see [49]. There have been numerous proposals for agentization in the literature (e.g., [35, 43, 17]) which have been broadly classified by Genesereth and Ketchpel [40] into four categories: in the first category, each agent has an associated "transducer" that converts all incoming messages and requests into a form that is intelligible to the agent. This is clearly not what happens in IMPACT – as noted in [40], the transducer has to anticipate what other agents will send us and translate that – something which is clearly difficult to do. The second approach is based on wrappers which "inject code into a program to allow it to communicate" [40, p.51]. The IMPACT architecture provides a language (the service description language) for expressing such wrappers, together with accompanying algorithms. The third approach described in [40] is to completely rewrite the code implementing an agent which is obviously a very expensive alternative. Last but not least, there is the *mediation* approach proposed by Wiederhold [103], which assumes that all agents will communicate with a

mediator which in turn may send messages to other agents. In contrast, our framework allows point to point communication between agents without having to go through a mediator. Of course, none of these efforts explicitly address agent decision making in heterogeneous environments, which is the focus of our effort.

Matchmaking. First, there has been substantial work on matchmaking, in which agents advertise their services, and matchmakers match an agent requesting a service with one (or more) that provides it. Kuokka and Harada [55] present the SHADE and COINS systems for matchmaking. Decker, Sycara, and Williamson [27] present matchmakers that store capability advertisements of different agents. Arisha *et al.* [5] developed a mechanism for matchmaking based on a similarity metric between a stored capability description, and a requested capability. In [75], Papakonstantinou and Vassalos provided an extremely expressive, Datalog-style language for describing the capabilities of different data sources and agents. Recently, Sycara *et al.* [95] have designed a system called LARKS that merges, and extends the ideas of similarity based inexact retrieval, and capability description languages. This paper, in contrast, merely focuses on how an agent makes decisions, rather than determining how one agent "matches" up with another.

Relationship to Heterogeneous Data Integration in the Database Community. There is now a great deal of work in mediated systems techniques. In this paragraph, we merely explain the relationship between code call conditions and existing work on data and software integration.

For example, there have been several efforts to integrate multiple relational DBMSs [26, 72] and relational DBMSs, object-oriented DBMSs and/or file systems [36, 52, 89]. However, to date, the semantics of mediators that take actions has not been explored. The work in this paper builds upon mediation efforts reported upon in our HERMES effort described previously in [20, 61, 61, 94, 63]. The Stanford TSIMMIS project [22] effort aimed at integrating a wide variety of heterogeneous databases, together with a free text indexing system. In contrast, HERMES integrated arbitrary software packages such as an Army Terrain Route Planning System, Jim Hendler's UM Nonlin planning system, a face recognition system, a video reasoning system, and various mathematical programming software packages are integrated currently into Hermes. As a consequence, TSIMMIS was able to take advantage of its focus on integrating databases to perform some optimizations which HERMES was unable to incorporate, but conversely, HERMES was able to access many data sources that TSIMMIS could not. Query optimization methods applicable to both TSIMMIS and HERMES were studied in [1]. The SIMS system [4] at USC uses a LISP-like syntax to integrate multiple databases as well. It is closely related to the HERMES effort. HERMES used minimalistic versions of logic to integrate data and software, while SIMS used a somewhat richer language. As a consequence, HERMES was able to take advantage of very efficient caching and query optimization methods [60, 1], but may have not been able to easily express some of the more sophisticated reasoning tasks desired by the authors of SIMS. Other important later directions on mediation include the InfoSleuth effort [13] at MCC.

7 Conclusions and Future Work

In this paper, we have argued the following two simple points:

(I) Agents in the real world manipulate not just logical formulas, but complex data types, that vary from one application to another.

(II) Agents must be able to act in accordance with a specific, declarative action policy that governs their actions. It must be possible to build such a *declarative* policy on top of the existing data structures

that the agent's implemented *imperative* software code manipulates.

Towards this end, we have developed the concept of an agent state, that can consist of instantiations of arbitrary data structures. We then develop the concept of an agent program, building on top of work in deontic logic. Agent programs allow the designer of an agent to specify how an agent should act, and take into account the following aspects. What is the agent obliged to do? What is the agent permitted to do? What is the agent forbidden from doing? and so on. We have developed a theoretical framework within which agent programs can be built on top of arbitrary pieces of software code, and we have developed a series of successively more refined declarative semantics for agent programs. As the declarative semantics for agent programs become intuitively more appealing, they (with some exceptions) also become computationally more complex. We have further established results showing the relationship between the declarative semantics and computational complexity, which are described in Part II of this series of papers. Our complexity results also, for the most part, include algorithms to compute the relevant semantical structures.

Our semantical results are closely related to other research in the field of artificial intelligence. In particular, we have demonstrated that three well-known semantics for logic programs, namely the model semantics, minimal model semantics, and the stable model semantics, are captured within our agent program framework. These semantics are well-known to correspond to certain fragments of advanced knowledge representation frameworks such as default logic and circumscription.

Last, but not least, we have developed a simulation of the working of our agent framework in the area of supply chain management. In the simulation, we have built several agents on top of legacy commercial software including Microsoft Access database agents, and ESRI MapObject agents as well.

Our current and ongoing efforts focus on the following subjects:

- We are extending the semantical framework described here to accommodate the following types of reasoning not currently included: (a) reasoning about uncertain agent states, (b) reasoning about temporal actions, where an agent makes decisions on taking an action in the future, (c) reasoning about other agent's reasoning. All of these modes of reasoning are well recognized in the AI community, and expanding our semantical framework to accommodate these modes of reasoning is an important semantical issue.

- We are exploring alternative possibilities to the STRIPS-like action model. In particular, a replacement of STRIPS style actions with Hierarchical Task Network actions might be desirable, since this allows for representing complex actions. However, the resulting theory is much more complex.

- We are currently developing a compiler for agent programs – in particular, in a future paper [31], we will report upon a class of agent programs called regular agent programs that are guaranteed to possess reasonable status sets – regularity of agent programs is a syntactically easily verifiable property, and regular agent programs possess many nice computational properties. The paper [31] will report upon several experiments evaluating the ease of computing the several diverse semantics described in this paper. This implementation builds upon our existing HERMES Heterogeneous Reasoning and Mediator System, reported on in [20, 63, 94].

- We are studying the problem of whether all agent programs (under different semantics) can be embedded into logic programs. In other words, is there a translation $\wp$ that takes as input, an agent program $\mathcal{P}$, and produces as output, a logic program $\wp(\mathcal{P})$ such that there is a one one correspondence between appropriate status sets of $\mathcal{P}$ and appropriate models of $\wp(\mathcal{P})$?

- Suppose we choose to use Sem-status set semantics for agent programs, where Sem is any of the semantics introduced in this paper. At any given point t in time, the agent program has a Sem-status set, S_t that it acts on. When new events occur (e.g. new messages arrive), these events may be viewed as updates to the current agent state. We would like to incrementally compute a new status set S_{t+1} from S_t, the object state immediately after the **Do**-actions in S_t are executed, and the updates. We are developing algorithms for this task.

Acknowledgements

We wish to thank Alex Dekhtyar, Jürgen Dix, Bob Kowalski, Sarit Kraus, Munindar Singh, Terrence Swift, and the referees for a very close reading of (versions of) this manuscript, and for the numerous detailed comments they made. These commends have significantly improved the quality of this paper. We have also benefited from discussions with Swati Allen, Piero Bonatti, Steve Choy, Keith Clark, Phil Emmerman, Jim Hendler, Bob Kowalski, Dana Nau, Anil Nerode, Dino Pedreschi, Jose Salinas and Leon Sterling.

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, and by a DAAD grant.

References

[1] S. Adali, K. Candan, Y. Papakonstantinou, and V. Subrahmanian. Query Processing in Distributed Mediated Systems. In *ACM SIGMOD Conference on Management of Data*, Montreal, Canada, June 1996.

[2] K. Apt. Logic Programming. In van Leeuwen [100], chapter 10, pages 493–574.

[3] L. Åquist. Deontic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Vol.II*, chapter II.11, pages 605–714. D. Reidel Publishing Company, 1984.

[4] Y. Arens, C. Chee, C. Hsu, and C. Knoblock. Retrieving and Integrating Data From Multiple Information Sources. *International Journal of Intelligent Cooperative Information Systems*, 2:127–158, 1993.

[5] K. Arisha, S. Kraus, F. Ozcan, R. Ross, and V.S.Subrahmanian. IMPACT: The Interactive Maryland Platform for Agents Collaborating Together. Submitted for publication, November 1997.

[6] C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic Model Checking for Probabilistic Processes. In *Proceedings ICALP 97*, number 1256 in LNCS, pages 430–440, 1997.

[7] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. A Modal Programming Language for Representing Complex Actions. manuscript, 1998.

[8] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. An Abductive Proof Procedure for Reasoning about Actions in Modal Logic Programming. In *Workshop on Non Monotonic Extensions of Logic Programming at ICLP '96*, volume 1216 of *Lecture Notes in AI*, pages 132–150. Springer, 1998.

[9] C. Baral and M. Gelfond. Representing Concurrent Actions in Extended Logic Programming. In *13th Intl. Joint Conf. on Artificial Intelligence*, pages 866–871, 1993.

[10] C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[11] C. Baral, M. Gelfond, and A. Provetti. Representing Actions: Laws, Observations, and Hypothesis. *Journal of Logic Programming*, 31:201–243, 1997.

[12] C. Baral and J. Lobo. Formal Characterization of Active Databases. In D. Pedreschi and C. Zaniolo, editors, *Workshop of Logic on Databases (LID '96)*, volume 1154 of *Lecture Notes in Computer Science*, pages 175–195, San Miniato, Italy, 1996.

[13] R. Bayardo *et al.* Infosleuth: Agent-based Semantic Integration of Information in Open and Dynamic Environments. In *ACM SIGMOD Conf. on Management of Data*, pages 195–206, Arizona, USA, 1997.

[14] C. Bell, A. Nerode, R. Ng, and V. Subrahmanian. Mixed Integer Programming Methods for Computing Non-Monotonic Deductive Databases. *Journal of the ACM*, 41(6):1178–1215, November 1994.

[15] C. Bell, A. Nerode, R. Ng, and V. Subrahmanian. Implementing Deductive Databases by Mixed Integer Programming. *ACM Trans. on Database Syst.*, 21(2):238–269, 1996.

[16] R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[17] W. P. Birmingham, E. H. Durfee, T. Mullen, and M. P. Wellman. The Distributed Agent Architecture Of The University of Michigan Digital Library (UMDL). In *AAAI Spring Symposium Series on Software Agents*, 1995.

[18] M. Bratman. *Intentions, Plans and Practical Reason*. Harvard Univ. Press, 1987.

[19] M. Bratman, D. Israel, and M. Pollack. Plans and Resource-Bounded Practical Reasoning. *Computational Intelligence*, 4(4):349–355, 1988.

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

[21] R. Cattell *et al*. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1997.

[22] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proceedings of the 10th Meeting of the Information Processing Society of Japan (IPSJ)*, pages 7–28, Tokyo, Japan, October 1994. Also available via anonymous FTP from host db.stanford.edu, file `/pub/chawathe/1994/tsimmis-overview.ps`.

[23] P. Cohen and H. Levesque. Intention is Choice with Commitment. *Artificial Intelligence*, 42:263–310, 1990.

[24] S. Conry, K. Kuwabara, V. Lesser, and R. Meyer. Multistage Negotiation for Distributed Satisfaction. *IEEE Transactions on Systems, Man, and Cybernetics, Special Issue on Distributed Artificial Intelligence*, 21(6):1462–1477, 1991.

[25] S. Coradeschi and L. Karlsson. A Behavior-based Decision Mechanism for Agents Coordinating using Roles. In Wooldridge and Jennings [107], pages 100–105.

[26] U. Dayal and H. Hwang. View Definition and Generalization for Database Integration in a Multi-Database System. *IEEE Transactions on Software Engineering*, 10(6):628–644, 1984.

[27] K. Decker, K. Sycara, and M. Williamson. Middle Agents for the Internet. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 578–583, Nagoya, Japan, 1997.

[28] F. Dignum and R. Conte. Intentional Agents and Goal Formation. In Wooldridge and Jennings [107], pages 219–231.

[29] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A Formal Specification of dMARS. In Wooldridge and Jennings [107], pages 146–166.

[30] J. Dix, V. Subrahmanian, and G. Pick. Meta-Agent Programs. *Journal of Logic Programming*, to appear.

[31] T. Eiter and V. Subrahmanian. Regular Agent Programs and their Implementation. In preparation, 1998.

[32] E. Emerson. Temporal and Modal Logics. In van Leeuwen [100], chapter 16.

[33] O. Etzioni and D. Weld. A Softbot-Based Interface to the Internet. *Communications of the ACM*, 37(7):72–76, 1994.

[34] M. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[35] L. Gasser and T. Ishida. A Dynamic Organizational Architecture For Adaptive Problem Solving. In *Proceedings AAAI-91*, pages 185–190, 1991.

[36] N. Gehani, H. Jagadish, and W. Roome. OdeFS: A File System Interface to an Object-Oriented Database. In *Int'l Conference on Very Large Databases (VLDB)*, pages 249–260, 1994.

[37] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[38] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[39] M. Gelfond and V. Lifschitz. Representing Actions and Change by Logic Programs. *The Journal of Logic Programming*, 17(2):301–323, 1993.

[40] M. Genesereth and S. Ketchpel. Software Agents. *Communications of the ACM*, 37(7):49–53, 1994.

[41] M. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, 1987.

[42] G. D. Giacomo, Y. Lesperance, and H. Levesque. Reasoning about Concurrent Execution, Prioritized Interrupts, and Exogenous Actions in the Situation Calculus. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1221–1226, Nagoya, Japan, 1997.

[43] L. Glicoe, R. Staats, and M. Huhns. A Multi-Agent Environment for Department of Defense Distribution. In *IJCAI 95 Workshop on Intelligent Systems*, 1995.

[44] G. Gottlob. Complexity Results for Nonmonotonic Logics. *Journal of Logic and Computation*, 2(3):397–425, June 1992.

[45] A. Gupta and I. Mumick, editors. *Materialized Views*. MIT Press, 1998. to appear.

[46] A. Haddadi. Towards a Pragmatic Theory of Interactions. In *Intl Conf. on Multi-Agent Systems*, pages 133–139, 1995.

[47] S. Hansson. Review of Deontic Logic in Computer Science: Normative System Specification. *Bulletin of the IGPL*, 2(2):249–250, 1994.

[48] K. Hindriks, F. de Boer, W. van der Hoek, and J. Meyer. Formal Semantics of an Abstract Agent Programming Language. In Wooldridge and Jennings [107], pages 204–218.

[49] M. Huhns and M. Singh, editors. *Readings in Agents*. Morgan Kaufmann Press, 1997.

[50] N. R. Jennings. Controlling Cooperative Problem Solving in Industrial Multi-Agent Systems Using Joint Intentions. *Artificial Intelligence*, 75(2):1–46, 1995.

[51] S. Kanger. Law and Logic. *Theoria*, 38, 1972.

[52] A. Kemper, C. Kilger, and G. Moerkotte. Function Materialization in Object Bases: Design, Realization, and Evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):587–608, 1994.

[53] S. Kraus. Negotiation and Cooperation in Multi-Agent Environments. *Artificial Intelligence, Special Issue on Economic Principles of Multi-Agent Systems*, 94(1):79–97, 1997.

[54] C. Krogh. Obligations in Multi-Agent Systems. In Aamodt, Agnar, and Komorowski, editors, *Proceedings of the Fifth Scandinavian Conference on Artificial Intelligence (SCAI '95)*, pages 19–30, Trondheim, Norway, 1995. ISO Press.

[55] D. Kuokka and L. Harada. Integrating Information via Matchmaking. *Journal of Intelligent Informations Systems*, 6(3):261–279, 1996.

[56] Y. Labrou and T. Finin. A Semantics Approach for KQML – A General Purpose Communications Language for Software Agents. In *Intl Conf. on Information and Knowledge Management*, pages 447–455, 1994.

[57] Y. Labrou and T. Finin. Semantics for an Agent Communication Language. In Wooldridge and Jennings [107], pages 199–203.

[58] J. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1984, 1987.

[59] J. Lobo and V. Subrahmanian. Relating Minimal Models and Pre-Requisite-Free Normal Defaults. *Information Processing Letters*, 44:129–133, 1992.

[60] J. Lu, G. Moerkotte, J. Schue, and V. Subrahmanian. Efficient Maintenance of Materialized Mediated Views. In M. Carey and D. A. Schneider, editors, *ACM SIGMOD Conf. on Management of Data*, pages 340–351, San Jose, CA, May 1995.

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

[62] P. Maes. Agents that Reduce Work and Information Overload. *Communications of the ACM*, 37(7):31–40, 1994.

[63] S. Marcus and V. Subrahmanian. Foundations of Multimedia Database Systems. *Journal of the ACM*, 43(3):474–523, 1996.

[64] W. Marek and V. Subrahmanian. The Relationship Between Stable, Supported, Default and Auto-Epistemic Semantics for General Logic Programs. *Theoretical Computer Science*, 103:365–386, 1992.

[65] W. Marek and M. Truszczyński. *Nonmonotonic Logics – Context-Dependent Reasoning*. Springer, 1993.

[66] L. McCarty. Reflections on TAXMAN: An Experiment in Artificial Intelligence and Legal Reasoning. *Harvard Law Review*, 90:837–893, 1977.

[67] L. McCarty. Permissions and Obligations. In *Proceedings IJCAI '83*, pages 287–294, 1983.

[68] L. McCarty. Modalities Over Actions, I. Model Theory. In *Proceedings KR '96*, pages 437–448, 1996.

[69] L. McCarty and van der Meyden. Reasoning About Indefinite Actions. In *Proceedings KR '92*, pages 59–70, 1992.

[70] J.-J. C. Meyer and R. Wieringa, editors. *Deontic Logic in Computer Science*. Wiley & Sons, Chichester et al, 1993.

[71] R. Moore. Semantical Considerations on Nonmonotonic Logics. *Artificial Intelligence*, 25:75–94, 1985.

[72] A. Motro. Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering*, 13(7):785–798, 1987.

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

[74] D. Nute. A Non-Monotonic Logic Based on Conditional Logic. Technical Report TR 01–0007, Artificial Intelligence Programs, The University of Georgia, 1985.

[75] Y. Papakonstantinou and V. Vassalos. Expressive Capabilities of Description Languages and Query Rewriting Algorithms. *Journal of Logic Programming*, to appear.

[76] A. Rao and M. Georgeff. Modeling Rational Agents within a BDI-Architecture. In J. F. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the International Conference on Knowledge Representation and Reasoning*, pages 473–484, Cambridge, MA, 1991. Morgan Kaufmann.

[77] R. Reiter. On Closed-World Databases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

[78] R. Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[79] J. S. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers*. MIT Press, Boston, 1994.

[80] S. Rosenschein. Formal Theories of Knowledge in AI and Robotics. *New Generation Computing*, 3(4):345–357, 1985.

[81] S. Rosenschein and L. Kaelbling. A Situated View of Representation and Control. *Artificial Intelligence*, 73:149–173, 1995.

[82] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[83] Y. Ryu and R. Lee. Defeasible Deontic Reasoning: A Logic Programming Model. In Meyer and Wieringa [70], chapter 9, pages 225–241.

[84] T. Sandholm and V. Lesser. Coalition Formation Amongst Bounded Rational Agents. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 662–669, Montreal, Canada, 1995. Morgan Kaufmann.

[85] J. Schafer, T. Rogers, and J. Marin. Networked Visualization of US Army War Reserve Readiness Data. In *Proceedings International Workshop on Multimedia Information Systems*, number 1508 in LNCS, pages 136–147, 1998.

[86] M. Schoppers and D. Shapiro. Designing Embedded Agents to Optimize End-User Objectives. In Wooldridge and Jennings [107], pages 2–12.

[87] R. Schwartz and S. Kraus. Bidding Mechanisms for Data Allocation in Multi-Agent Environments. In Wooldridge and Jennings [107], pages 56–70.

[88] O. Shehory, K. Sycara, and S. Jha. Multi-Agent Coordination through Coalition Formation. In Wooldridge and Jennings [107], pages 135–146.

[89] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[90] J. Shoenfield. *Mathematical Logic*. Addison Wesley, 1967.

[91] Y. Shoham. Agent Oriented Programming. *Artificial Intelligence*, 60:51–92, 1993.

[92] M. Singh. A Customizable Coordination Service for Autonomous Agents. In Wooldridge and Jennings [107], pages 86–99.

[93] R. Smith and R. Davis. Negotiation as a Metaphor for Distributed Problem Solving. *Artificial Intelligence*, 20:63–109, 1983.

[94] V. Subrahmanian. Amalgamating Knowledge Bases. *ACM Trans. on Database Syst.*, 19(2):291–331, 1994.

[95] K. Sycara, J. Lu, and M. Klusch. Interoperability among Heterogeneous Software Agents on the Internet. Technical Report CMU-RI-TR-98-22, Carnegie Mellon University, October 1998.

[96] K. Sycara and D. Zeng. Multi-Agent Integration of Information Gathering and Decision Support. In *European Conf. on Artificial Intelligence (ECAI '96)*, 1996.

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

[98] W. van der Hoek, B. van Linder, and J.-J. Meyer. A Logic of Capabilities. In A. Nerode and Y. Matiyasevich, editors, *3rd Intl Symposium on the Logical Foundations of Computer Science (LFCS 94)*, pages 366–378. Springer Verlag, 1994.

[99] R. van der Meyden. A Clausal Logic for Deontic Action Specification. In *Proceedings ILPS 91*, pages 221–238, 1991.

[100] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*. Elsevier Science Publishers B.V. (North-Holland), 1990.

[101] E. Verharen, F. Dignum, and S. Bos. Implementation of a Cooperative Agent Architecture Based on the Language-Action Perspective. In Wooldridge and Jennings [107], pages 26–39.

[102] M. Wellman. A Market-Oriented Programming Environment and its Application to Distributed Multicommodity Flow Problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

[103] G. Wiederhold. Intelligent Integration of Information. In *ACM SIGMOD Conf. on Management of Data*, pages 434–437, 1993.

[104] F. Wilder. *A Guide to the TCP/IP Protocol Suite*. Artech House, 1993.

[105] M. Wooldridge and N. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Reviews*, 10(2), 1995.

[106] M. Wooldridge and N. Jennings. Formalizing the Cooperative Problem Solving Proces. In M. Huhns and M. Singh, editors, *Readings in Agents*, pages 430–440. Morgan Kaufmann Press, 1997.

[107] M. Wooldridge and N. Jennings, editors. *Intelligent Agents III: Int'l Workshop on Agent Theories, Architectures, and Languages*, number 1237 in LNCS, Providence, RI, 1997. Springer.

A Appendix: Complexity of S- and F-Concurrent Executability

The following two theorems, which state the complexity of S-concurrent and F-concurrent action executability, together with the polynomiality of weakly-concurrent execution establish Theorem 3.1.

Theorem A.1 *Let $AS = \{(\alpha_1, \theta_1, \gamma_1), \ldots, (\alpha_n, \theta_n, \gamma_n)\}$ be a given set of feasible execution triples $(\alpha_i, \theta_i, \gamma_i)$ on a given agent state $\mathcal{O}_\mathcal{S}$. Then, under the assumptions in Theorem 3.1, testing whether AS is S-concurrently executable is* NP*-complete.*

Proof. The problem is NP, since we can guess an appropriate permutation π and check in polynomial time whether AS is π-feasible. Indeed, by our assumptions we can always evaluate the precondition $Pre(\alpha(\vec{X}_{\pi(i)})\theta_{\pi(i)}\gamma_{\pi(i)})$ in polynomial time on $\mathcal{O}^i_\mathcal{S}$, and we can construct the state $\mathcal{O}^{i+1}_\mathcal{S}$ in polynomial time from $\mathcal{O}^i_\mathcal{S}$, for all $i = 0, \ldots, n-1$; overall, this is possible in polynomial time.

To show NP-hardness, we provide a reduction from a restriction of the satisfiability problem (SAT). Recall that SAT is the following problem: Given a formula $\phi = \bigwedge_{i=1}^m C_i$ which is the conjunction of propositional clauses C_i such that each clause C_i is a disjunction $C_i = L_{i,1} \vee \cdots \vee L_{i,k_i}$ of literals over propositional atoms $X = \{x_1, \ldots, x_n\}$, decide whether ϕ is satisfiable. It is well-known that SAT is a NP-complete problem. This remains true if we assume that each clause C_i contains three literals, and either the literals $L_{i,1}$, $L_{i,2}$, $L_{i,3}$ are all positive, or they are all negative; this restriction is known as monotone 3SAT (M3SAT) [34].

For the reduction to our problem, we suppose that the software code $\mathcal{S}$ provides access to a relational database $\mathcal{DB}$ and that an agent state $\mathcal{O}_\mathcal{S}$ is a relational database instance D.

Let I be an instance of M3SAT, consisting of clauses $C_1, \ldots, C_m$ over variables $x_1, \ldots, x_n$.

The database $\mathcal{DB}$ has four relations: VAL(Var, BV), which stores a Boolean value for each variable; SV(Var), which intuitively holds the variables which have assigned a value; SAT(C) which intuitively stores the clauses which are satisfied; and, the 0-ary relation INIT.

The initial database DB holds all possible tuples; in particular, both tuples $(x_i, 0)$ and $(x_i, 1)$ are in VAL for every atom x_i. This will ensure that every execution triple in AS is feasible.

The execution triples in AS are designed such that a feasible schedule must have the following phases:

***Initialization*.** An action `init` must be executed here, which clears all relations except VAL.

***Choice*.** In this phase, for each atom x_i a truth value is chosen by removing from VAL either $(x_i, 0)$ (which sets x_i to 1), or $(x_i, 1)$ (which sets x_i to 0).

***Checking*.** In this phase, it is checked for every single clause C_i independently whether C_i is satisfied.

***Success*.** In this phase, the single tests for the C_i are combined; if the result is positive, i.e., the assignment selected in the choice phase satisfies every C_i, then a success action sat is executed which enables to gracefully execute the remaining actions.

***Clearance*.** In this phase, which is entered only if the Success phase had a positive result, all remaining actions which have not been executed so far are taken. Moreover, we add an action which completely clears the database, such that every feasible permutation π leads to the empty database.

The actions and their descriptions are given in the following table:

Phase	*Action*	*Precondition*
Init	`init`	INIT
Choice	$\texttt{set}_1(X)$	$\text{VAL}(X,1) \wedge \text{VAL}(X,0)$
	$\texttt{set}_0(X)$	$\text{VAL}(X,1) \wedge \text{VAL}(X,0)$
Checking	$\texttt{check}_{i,j}$	$\text{SV}(x_1) \wedge \cdots \wedge \text{SV}(x_n) \wedge At_{i,j}$
Success	`sat`	$\text{SAT}(c_1) \wedge \cdots \wedge \text{SAT}(c_m)$
Clearance	`clear`	$true$

Phase	*Action*	*Add Set*	*Delete Set*
Init	`init`	$\emptyset$	{ SV(V), SAT(C), INIT }
Choice	$\texttt{set}_1(X)$	{ SV(X) }	{ INIT, VAL($X,0$) }
	$\texttt{set}_0(X)$	{ SV(X) }	{ INIT, VAL($X,1$) }
Checking	$\texttt{check}_{i,j}$	{ SAT(c_i) }	{ INIT }
Success	`sat`	{ VAL($x_i,0$), VAL($x_i,1$) $\mid i = 1,\ldots,n$}	{ INIT }
Clearance	`clear`	$\emptyset$	all relations

Here $At_{i,j} = \text{VAL}(x_k, 1)$ if the j-th literal of clause C_i is x_k, and $At_{i,j} = \text{VAL}(x_k, 0)$ if it is $\neg x_k$.

Observe that all variables in the preconditions of the above actions α are action parameters. Thus, γ is void in every solution of $Pre(\alpha(\vec{X}\theta))$, and thus for every $\alpha(\vec{X})$ and θ at most one feasible triple $(\alpha(\vec{X}), \theta, \gamma)$ may exist, in which γ is void; we hence write simply $\alpha(\vec{X}\theta)$ for this triple. Let the set AS be as follows:

$$\begin{aligned} AS \;=\; & \{\texttt{set}_1(x_i), \texttt{set}_0(x_i) \mid 1 \le i \le n\} \cup \\ & \{\texttt{check}_{i,j} \mid 1 \le i \le m, 1 \le j \le 3\} \cup \{\texttt{init}, \texttt{clear}, \texttt{sat}\}. \end{aligned}$$

Notice the following observations on a feasible permutation π for AS:

- `init` must be executed first, i.e., $\alpha_{\pi(1)} = \texttt{init}$, and `clear` must be executed as the last action.
- Clause checking can only be done after some choice action $\texttt{set}_v(x_i)$, $v \in \{0,1\}$, has occurred, for every $i = 1,\ldots,n$.
- In the choice phase, execution of at most one of the actions $\texttt{set}_1(x_i)$, $\texttt{set}_0(x_i)$ is possible, for every $i = 1,\ldots,n$.
- Success is only possible, if for each clause C_i at least one action $\texttt{check}_{i,j}$ has been executed.
- After the Success phase, first every remaining action $\texttt{set}_v(x_i)$ can be executed, then all remaining actions $\texttt{check}_{i,j}$ are possible, and finally `clear` can be done.

It holds that AS is S-concurrently executable, i.e., there exists some permutation π such that AS is π-feasible, if and only if I is a Yes-Instance of M3SAT.

Remark. The construction can be extended so to obtain a fixed action base $\mathcal{AB}$ such that the set AS in the construction is an instance of $\mathcal{AB}$, by adding further relations to the schema describing the clauses in I. Moreover, strict typing of the values occurring in the relations is possible. ∎

Theorem A.2 *Let $AS = \{(\alpha_1, \theta_1, \gamma_1), \ldots, (\alpha_n, \theta_n, \gamma_n)\}$ be a given set of feasible execution triples $(\alpha_i, \theta_i, \gamma_i)$ on a given agent state $\mathcal{O}_\mathcal{S}$. Then, under the assumptions in Theorem 3.1, testing whether AS is F-concurrently executable is* co-NP*-complete.*

Proof. The problem is in co-NP, since we can guess permutations π and π' such that either $AS[\pi] = \alpha_{\pi(1)}, \ldots, \alpha_{\pi(n)}$ or $AS[\pi'] = \alpha_{\pi'(1)}, \ldots, \alpha_{\pi'(n)}$ is not feasible, or $AS[\pi]$ and $AS[\pi']$ yield a different result. By the assumptions, the guess for Π and Π' can be verified in polynomial time (cf. proof of Theorem A.1).

To show that the problem is co-NP-hard, we consider as in the proof of Theorem A.1 the case where $\mathcal{S}$ is a relational database $\mathcal{D}$ and an agent state $\mathcal{O}_\mathcal{S}$ is a relational database instance D.

We reduce the complement of M3SAT to F-concurrent executability checking. Let I be an instance of M3SAT, i.e., a CNF $\phi = \bigwedge_i C_i$ in variables $x_1, \ldots, x_n$. Without loss of generality, ϕ has at least one positive clause and at least one negative clause.

Let I' be the instance of I which results if every positive clause is made negative and vice versa, and if every atom x_i is replaced by x_{n+i}. Clearly, I' is a Yes-instance if and only if I is a Yes-instance, if and only if $I \cup I'$ is satisfied by some truth assignment to $x_1, \ldots, x_{2n}$ in which x_i has the opposite value to the value of x_{n+i}, for every $i = 1, \ldots, n$.

The relational database $\mathcal{D}$ we construct has four relations: $\text{POS}(V_1, V_2, V_3)$ and $\text{NEG}(V_1, V_2, V_3)$, which serve for storing the positive and negative clauses of $I \cup I'$, respectively; $\text{VAL}(Var, BV)$, which stores a truth value assignment to the variables, such that variable x_i is true if $(x_i, 1) \in \text{VAL}$, and x_i is false if $(x_i, 0) \in \text{VAL}$, for every $i = 1, \ldots, 2n$; and, a 0-ary relation UNSAT.

The initial database D contains the relations POS and NEG storing the clauses of $I \cup I'$, the relation VAL which holds the tuples $(x_i, 0)$ and $(x_{n+i}, 1)$, for every $i = 1, \ldots, n$ and the relation UNSAT is empty.

The action base contains the actions $\texttt{switch}(X, Y)$ and $\texttt{eval}$, where

$$
\begin{array}{lll}
\texttt{switch}: & Pre(\texttt{switch}(X,Y)) = true, & \\
& Add(\texttt{switch}(X,Y)) = \{\, \text{VAL}(X,1), \text{VAL}(Y,0)\}, & \\
& Del(\texttt{switch}(X,Y)) = \{\text{VAL}(X,0), \text{VAL}(X,1)\}; & \\
\texttt{eval}: & Pre(\texttt{eval}) = \left\{ \begin{array}{l} \exists V_1,V_2,V_3.\text{POS}(V_1,V_2,V_3) \wedge \text{VAL}(V_1,0) \wedge \text{VAL}(V_2,0) \wedge \text{VAL}(V_3,0) \wedge \\ \exists V_1,V_2,V_3.\text{NEG}(V_1,V_2,V_3) \wedge \text{VAL}(V_1,1) \wedge \text{VAL}(V_2,1) \wedge \text{VAL}(V_3,1) \end{array} \right\}, & \\
& Add(\texttt{eval}) = \{\, \text{UNSAT}\,\}, & \\
& Del(\texttt{eval}) = \emptyset. &
\end{array}
$$

Observe that all variables in preconditions of actions are bound; we thus write $(\alpha(\vec{X}\theta))$ for $(\alpha(\vec{X}), \theta, \gamma)$ where γ is void.

The set AS of execution triples is

$$AS = \{\texttt{switch}(x_i, x_{n+i}) \mid 1 \leq i \leq n\} \cup \{\texttt{eval}\}.$$

Intuitively, a switch action $\texttt{switch}(x_i, x_{n+i})$ flips the value of x_i from 0 to 1 and the value from x_{n+i} from 1 to 0. The $\texttt{eval}$ action checks whether for the truth assignment to $x_1, \ldots, x_{2n}$ given in the database, there is some positive clause and some negative clause which are both violated.

For any permutation π on AS, the actions $\alpha_{\pi(j)}$ scheduled before $\alpha_{\pi(i)} = \texttt{eval}$ flip the values of some variables; notice that flipping x_i is simultaneously done with flipping x_{n+i}. The precondition $Pre(\texttt{eval})$ is true, precisely if there exists some positive clause P and some negative clause N in $I \cup I'$ which are both violated; this is equivalent to the property that the current assignment σ stored in D does not satisfy I.

To see this, if P is from I, then I is not satisfied by σ, and if P is from I', then there is a corresponding negative clause $N(P)$ in I such that $N(P)$ is not satisfied. On the other hand, if σ does not satisfy I, then

there exists either a positive clause $P \in I$ or a negative clause $N \in I$ which is not satisfied, and thus the corresponding negative clause $N(P) \in I'$ (resp. positive clause $P(N) \in I'$) is not satisfied by σ; this means $Pre(\texttt{eval})$ is true.

Clearly, all actions in AS are executable on the initial database DB, and every feasible permutation $AS[\pi]$ yields the same resulting database. Hence, it follows that AS is F-concurrently executable, if and only if I is a Yes-Instance. This proves the result.

Note that we can derive AS from a simple fixed program, if we store the pairs x_i, x_{n+i} in a separate relation. The result then immediately extends to the data complexity of a collection of feasible execution triples. ∎

Remark A.1 The F-concurrent execution problem in the above database setting is polynomial, if the precondition is a conjunction of literals and there are no free (existential) variables in it. Then, the condition amounts to the following property. Let AS be a set of action execution triples, and denote by $Pre^+(\alpha)$ (resp., $Pre^-(\alpha)$) the positive (resp., negated) ground atoms in the precondition of α. Moreover, let $Add\downarrow(\alpha)$ and $Del\downarrow(\alpha)$ be the ground instances of the atoms in $Add(\alpha)$ and $Del\downarrow(\alpha)$ over the database, respectively.

(i) $Add\downarrow(\alpha) \cap Pre^-(\beta) = \emptyset$, for every $\alpha \neq \beta \in AS$;

(ii) $Del\downarrow(\alpha) \cap (Pre^+(\beta) \cup Add\downarrow(\beta)) = \emptyset$, for every $\alpha \neq \beta \in AS$;

(iii) $Add\downarrow(\alpha) \cap Del\downarrow(\alpha) = \emptyset$, for every $\alpha \in AS$.

(Condition (iii) is actually not needed, but avoids philosophical problems.) An alternative, less conservative approach would be to limit change by α to the atoms not in $Add\downarrow(\alpha) \cap Del\downarrow(\alpha)$. ∎

B Appendix: Proofs of Other Results

Proof of Theorem 5.3.
($\Rightarrow$) Suppose $S = \mathit{lfp}(T_{\mathcal{P},\mathcal{O}_\mathcal{S}})$ a rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$. Then, S is feasible by definition of rational status set. By Lemma 5.2, S is a pre-fixpoint of $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}$. Since $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}$ is monotone, it has by the Knaster-Tarski Theorem a least pre-fixpoint, which coincides with $\mathit{lfp}(T_{\mathcal{P},\mathcal{O}_\mathcal{S}})$ (cf. [2, 58]). Thus, $\mathit{lfp}(T_{\mathcal{P},\mathcal{O}_\mathcal{S}}) \subseteq S$. Clearly, $\mathit{lfp}(T_{\mathcal{P},\mathcal{O}_\mathcal{S}})$ satisfies $(S1)$ and $(S3)$; moreover, $\mathit{lfp}(T_{\mathcal{P},\mathcal{O}_\mathcal{S}})$ satisfies $(S2)$, as S satisfies $(S2)$ and this property is hereditary. By the definition of rational status set, it follows $\mathit{lfp}(T_{\mathcal{P},\mathcal{O}_\mathcal{S}}) = S$.

($\Leftarrow$) Suppose $S = \mathit{lfp}(T_{\mathcal{P},\mathcal{O}_\mathcal{S}})$ is a feasible status set. Since every status set S' which satisfies $(S1)$–$(S3)$ is a pre-fixpoint of $T_{\mathcal{P},\mathcal{O}_\mathcal{S}}$ and $\mathit{lfp}(T_{\mathcal{P},\mathcal{O}_\mathcal{S}})$ is the least prefix point, $S' \subseteq S$ implies $S = S'$. It follows that S is rational.

Proof of Proposition 5.6.
In order to show that a reasonable status set S of $\mathcal{P}$ is a rational status of $\mathcal{P}$, we have to verify (1) that S is a feasible status set and (2) that S is grounded.

Since S is a reasonable status set of $\mathcal{P}$, it is a rational status set of $\mathcal{P}' = red^S(\mathcal{P}, \mathcal{O}_\mathcal{S})$, i.e., a feasible and grounded status set of $\mathcal{P}'$. Since the conditions $(S2)$–$(S4)$ of the definition of feasible status set depend only on S and $\mathcal{O}_\mathcal{S}$ but not on the program, this means that for showing (1) it remains to check that $(S1)$ (closure under the program rules) is satisfied.

Let thus r be a ground instance of a rule from $\mathcal{P}$. Suppose the body $B(r)$ of r satisfies the conditions 1.–4. of $(S1)$. Then, by the definition of $red^S(\mathcal{P}, \mathcal{O}_S)$, we have that the reduct of the rule r, obtained by removing all literals of $B^-_{as}(r)$ from the body, is in $\mathcal{P}'$. Since S is closed under the rules of $\mathcal{P}'$, we have $H(r) \in S$. Thus, S is closed under the rules of $\mathcal{P}$, and hence $(S1)$ is satisfied. As a consequence, (1) holds.

For (2), we suppose S is not grounded, i.e., that some smaller $S' \subset S$ satisfies $(S1)$–$(S3)$ for $\mathcal{P}$, and derive a contradiction. If S' satisfies $(S1)$ for $\mathcal{P}$, then S' satisfies $(S1)$ for $\mathcal{P}'$. For, if r is a rule from $\mathcal{P}'$ such that 1.–4. of $(S1)$ hold, then there is a ground rule r' of $\mathcal{P}$ such that r is obtained from r' in the construction of $red^S(\mathcal{P}, \mathcal{O}_S)$ and, as easily seen, 1.–4. of $(S1)$ hold. Since S' satisfies $(S1)$ for $\mathcal{P}$, we have $H(r) \in S$. It follows that S' satisfies $(S1)$ for $\mathcal{P}'$. Furthermore, since $(S2)$ and $(S3)$ do no depend on the program, also $(S2)$ and $(S3)$ are satisfied for S' w.r.t. $\mathcal{P}'$. This means that S is not a rational status set of $\mathcal{P}'$, which is the desired contradictiono.

Thus, (1) and (2) hold, which proves the result.

Proof of Theorem 5.8
The proof is analogous to the proof of Theorem 5.3; observe that any status set S' which satisfies the conditions $(S1)$ and $(S3)$ of A-relativized feasibility, is a pre-fixpoint of $T_{\mathcal{P},\mathcal{O}_S,A}$.

Proof of Theorem 5.9.
S is weak rational, if and only if S is A-rational for some A such that for every $A' \neq A$ such that $A \subseteq A'$, no A'-rational status set exists. This is equivalent to the fact that A is a maximal set of ground actions such that some A-rational status sets exist. By Theorem 5.8, a status set S is A-rational iff $S = \mathit{lfp}(T_{\mathcal{P},\mathcal{O}_S,A})$ and S is A-feasible; the result follows.

Proof of Proposition 5.10.
Clearly, S is $A(S)$-feasible. Suppose that $A(S)$ is not the unique maximal set A such that S is A-feasible. Then, there exists a set $A' \neq A(S)$ and a ground action $\alpha \in A' \setminus A(S)$ such that S is A'-feasible. From the definition of $A(S)$, it follows $\mathbf{O}\alpha \in S$ and $\mathbf{Do}\alpha \notin S$; since the rule $\mathbf{O}\alpha \Rightarrow \mathbf{Do}\alpha$ applies w.r.t. A', it follows $\mathbf{Do}\alpha \in S$, which is a contradiction.

Proof of Theorem 5.13.
$(\Rightarrow)$ If S is weak rational, then (i) follows from Theorem 5.9 and Proposition 5.10. Suppose for some A' in (ii), $S_{A'} = \mathit{lfp}(T_{\mathcal{P},\mathcal{O}_S,A'})$ is A'-feasible. Then, by Theorem 5.8, the set $S_{A'}$ is A'-rational, which contradicts that S is a weak rational status set.

$(\Leftarrow)$ Suppose (i) and (ii) hold. Then, by Theorem 5.8, S is A-rational. Suppose S is not a weak rational status set; hence some $A' \neq A$ exists, $A \subseteq A'$, for which some A'-rational status set S' exists. Since property $(S2)$ of the feasibility condition is hereditary, it follows from Proposition 5.12 that for every $A'' \subseteq A$ the status set $S_{A''} = \mathit{lfp}(T_{\mathcal{P},\mathcal{O}_S,A''})$ satisfies $(S2)$. Moreover, $S_{A''}$ satisfies $(S1)$ and $(S3)$. Since $\mathcal{IC} = \emptyset$, we have that $S_{A''}$ is A''-feasible. Let $\alpha \in A' \setminus A$ and set $A'' = A \cup \{\alpha\}$. This raises a contradiction to (ii). Consequently, an A' as hypothesized does not exist, which proves that S is weak rational.

Proof of Proposition 5.17.
(1) Suppose M is a model of the program P. To show that A_M is a feasible status set of $\mathcal{P}(P)$, we need to show that A_M satisfies conditions $(S1)$–$(S4)$ in the definition of a feasible status set.

$(S1)$ Suppose r is a ground instance of a rule in $\mathsf{AG}(P)$ whose body is true w.r.t. A_M. Rule r must be one of the form $\mathbf{P}a \leftarrow \mathbf{P}b_1, \ldots, \mathbf{P}b_m, \neg\mathbf{P}c_1, \ldots, \mathbf{P}c_m$. Then $\{\mathbf{P}(b_1), \cdots, \mathbf{P}(b_m)\} \subseteq A_M$ and $\{\mathbf{P}(c_1), \ldots, \mathbf{P}(c_n)\} \cap A_M = \emptyset$. By definition of A_M, it follows that $\{a_1, \ldots, a_m\} \subseteq M$. By

definition of A_M we have $\{c_1, \ldots, c_n\} \cap M = \emptyset$. As M is a model of P, $a \in M$, and hence, by definition of A_M, $\mathbf{P}(a) \in A_M$.

Thus, A_M satisfies condition $(S1)$ in the definition of feasible status set.

$(S2)$ It is easy to see that the conditions defining deontic and action consistency (Definition 5.2) are satisfied. The reason is that by definition, A_M only contains atoms of the form $\mathbf{P}(-)$ and hence, the first two bullets of Definition 5.2 are immediately true. The third bullet of Definition 5.2 is satisfied because all actions in $\mathsf{AG}(P)$ have an empty precondition, and hence, the consequent of the implication in the third bullet is immediately true. The action consistency requirement is satisfied trivially as $\mathsf{AG}(P)$ contains no action constraints.

$(S3)$ The deontic and action closure requirements stated in Definition 5.3 are trivially satisfied because A_M contains no status atoms of the form $\mathbf{O}(-)$ or $\mathbf{Do}(-)$.

$(S4)$ As $\mathsf{AG}(P)$ contains no integrity constraints, it follows immediately that the state consistency requirement is satisfied by A_M.

This completes the proof of (1) of the theorem.
(2) Suppose A is a feasible status set of $\mathsf{AG}(P)$ and M_A satisfies the body of a ground instance, r, of a rule in P. Let rule r be of the form

$$a \leftarrow b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_n.$$

As $\{b_1, \ldots, b_m\} \subseteq M_A$, we must, by definition, have $\{\mathbf{P}(b_1), \ldots, \mathbf{P}(b_m)\} \subseteq M_A$. As $\{c_1, \ldots, c_n\} \cap M_A = \emptyset$, we must, by definition, have $A \cap \{\mathbf{P}(c_1), \ldots, \mathbf{P}(c_n)\} = \emptyset$. By construction of $\mathsf{AG}(P)$, we have the rule

$$\mathbf{P}(a) \leftarrow \mathbf{P}(b_1), \ldots, \mathbf{P}(b_m), \neg\mathbf{P}(c_1), \ldots, \neg\mathbf{P}(c_n)$$

in $\mathsf{AG}(P)$. As A is a feasible status set, it must satisfy axiom $(S1)$. Hence, $\mathbf{P}(a) \in A$, which implies that $a \in M_A$. This completes the proof.

Proof of Proposition 5.19.
We show part (1). Part (2) is proved by an analogous (and somewhat simpler) reasoning. Suppose M is a stable model of P. Then let $Q = red^{A_M}(\mathsf{AG}(P), \emptyset)$ be the agent program obtained as the reduct of $\mathsf{AG}(P)$ w.r.t. A_M and the empty agent state. To show that A_M is a reasonable status set of $\mathsf{AG}(P)$, we need to show that A_M is a rational status set of Q. For this we need to show that each of conditions $(S1)$–$(S4)$ is true for A_M with respect to Q, and that A_M is an inclusion-minimal set satisfying this condition.

$(S1)$ Consider a rule in Q having a ground instance, r, of the form

$$\mathbf{P}(a) \leftarrow \mathbf{P}(b_1), \ldots, \mathbf{P}(b_m)$$

such that $\{\mathbf{P}(b_1), \ldots, \mathbf{P}(b_m)\} \subseteq A_M$. By definition, $\{b_1, \ldots, b_m\} \subseteq M$. As $r \in Q$, there must exist a rule in $\mathsf{AG}(P)$ having a ground instance, r', of the form

$$\mathbf{P}(a) \leftarrow \mathbf{P}(b_1), \ldots, \mathbf{P}(b_m), \neg\mathbf{P}(c_1), \ldots, \neg\mathbf{P}(c_n)$$

such that $\{\mathbf{P}(c_1), \ldots, \mathbf{P}(c_n)\} \cap A_M = \emptyset$. This means that there is a rule in P having a ground instance, $r^\star$, of the form

$$a \leftarrow b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_m$$

such that $\{c_1, \ldots, c_n\} \cap M = \emptyset$. Thus as M satisfies the body of $r^\star$ and as M is a stable model of P (and hence a model of P), $a \in M$ which implies that $\mathbf{P}(a) \in A_M$ and this concludes this part of our proof.

$(S2)$ The first bullet in the definition of deontic consistency is immediately satisfied as A_M contains no status atoms of the form $\mathbf{W}(-)$. The second bullet in the definition of deontic consistency is immediately satisfied as A_M contains no status atoms of the form $\mathbf{F}(-)$. The third bullet in the definition of deontic consistency is immediately satisfied as all actions have an empty precondition, which is immediately satisfied. The action consistency requirement is immediately satisfied as the set $\mathcal{AC}$ of action constraints produced by AG is empty.

$(S3)$ A_M is deontically closed because, by definition, A_M contains no status atoms of the form $\mathbf{O}(-)$. A_M is action-closed because A_M contains no status atoms of the form $\mathbf{O}(-), \mathbf{Do}(-)$.

$(S4)$ A_M satisfies the state consistency property because the set $\mathcal{IC}$ of integrity constraints produced by AG is empty.

At this point, we have shown that A_M is a feasible status set of Q. To establish that it is a rational status set of Q, we need to show that it is inclusion-minimal. Suppose not. Then there exists a set $S \subset A_M$ such that S is a feasible status set of Q. It is straightforward to show that $S^\star = \mathbf{P}(S)$ is a stable model of P. But then $S^\star \subset M$, which is a contradiction, as no stable model of any logic program can be a strict subset of another stable model [64].

C Appendix: Table of Notation Used in the Paper

In the following tables, all numbers refer to Definition numbers, unless explicitly stated otherwise.

Notation	Location	Description
$\mathcal{S}$	Sec. 1 beginning	Software Code
$\mathcal{T}_{\mathcal{S}}$	Sec. 1 beginning	Set of data types for software code $\mathcal{S}$
$\mathcal{F}_{\mathcal{S}}$	Sec. 1 beginning	Set of predefined functions for software code $\mathcal{S}$
$\mathcal{O}_{\mathcal{S}}$	Sec. 1 beginning	Agent State
$\tau \in \mathcal{T}_{\mathcal{S}}$	Sec. 1 beginning	Object type
$Var(\tau)$	Sec. 2.1 beginning	Variable symbols ranging over τ
$\mathtt{X.f_i.g}$	Sec. 2.1 beginning	Path variable
$\mathcal{S} : \mathtt{f(d_1, \ldots, d_n)}$	2.2	Code call
$\mathtt{cc}$	2.2	Code call
$\mathtt{in(X,cc)}$	2.2	Code call atom
$<, >, \leq, \geq, =, \&$	2.3	Code call condition operators
χ	2.3	Code call condition
$Sol(\chi)_{\mathcal{T}_{\mathcal{S}},\mathcal{O}_{\mathcal{S}}}$	2.6	Set of code call solutions
$\mathcal{O}_Sol(\chi)_{\mathcal{T}_{\mathcal{S}},\mathcal{O}_{\mathcal{S}}}$	2.6	Set of all objects in a code call solution
$\mathsf{ins}_{\mathcal{S}}$	After 2.6	Inserts objects in a state
$\mathsf{del}_{\mathcal{S}}$	After 2.6	Deletes objects from a state
$\psi \Rightarrow \chi_a$	2.7	Integrity constraint
$\mathcal{O}_{\mathcal{S}} \models IC$	2.8	Integrity constraint satisfaction
$\mathcal{IC}$	2.8	Finite collection of integrity constraints
α	3.1	Action
$(\tau_1, \ldots, \tau_n)$	3.1	Action schema
$Pre(\alpha)$	3.1	Precondition for action α
$Add(\alpha)$	3.1	Add list for action α
$Del(\alpha)$	3.1	Delete list for action α
(θ, γ)-Executability	3.3	Action execution under substitutions
$FE(\alpha(\vec{X}), \mathcal{O}_{\mathcal{S}})$	3.3	Set of all θ, γ making an action $\alpha(\vec{X})$ executable
$apply(A, \mathcal{O}_{\mathcal{S}})$	3.5	Weak-concurrent execution
S-concurrently executable	3.6	Sequential-concurrent execution
F-concurrently executable	3.7	Full-concurrent execution
AC	3.8	Action constraint
$S, \mathcal{O}_{\mathcal{S}} \models AC$	3.9	Action constraint satisfaction
$\mathbf{P}\alpha$	3.10	Agent is permitted to take action α
$\mathbf{F}\alpha$	3.10	Agent is forbidden to take action α
$\mathbf{O}\alpha$	3.10	Agent is obliged to take action α
$\mathbf{W}\alpha$	3.10	Obligation to take action α is waived
$\mathbf{Do}\alpha$	3.10	Agent does take action α
$A \leftarrow L_1, \ldots, L_n$	3	Action rule
$\mathcal{P}$	3.12	Agent program

(continued next page)

Notation	Location	Description
$H(r)$	para before Ex. 3.7	Head of rule r
$B(r)$	para before Ex. 3.7	Body of rule r
$B^-(r)$	para before Ex. 3.7	Negative literals in the body of rule r
$B^+(r)$	para before Ex. 3.7	Positive literals in the body of rule r
$\neg.B^-(r)$	para before Ex. 3.7	Atoms of the negative literals of rule r
$B(r)_{as}, B(r)_{cc}$	para before Ex. 3.7	Body of rule r restricted to action status atoms
$B(r)_{cc}$	para before Ex. 3.7	Body of rule r restricted to code call atoms
$B^-(r)_{as}$	para before Ex. 3.7	Negative literals in the body of rule r restricted to action status atoms
$B^-(r)_{cc}$	para before Ex. 3.7	Negative literals in the body of rule r restricted to code call atoms
$B^+(r)_{as}$	para before Ex. 3.7	Positive literals in the body of rule r restricted to action status atoms
$B^+(r)_{cc}$	para before Ex. 3.7	Positive literals in the body of rule r restricted to code call atoms
$\neg.B^-(r)_{as}$	para before Ex. 3.7	Negations of atoms in $B^-(r)_{as}$
$\neg.B^-(r)_{cc}$	para before Ex. 3.7	Negations of atoms in $B^-(r)_{cc}$
$DCl(S)$	5.3	Deontic closure
$ACl(S)$	5.3	Action closure
$App_{\mathcal{P},\mathcal{O}_S}(S)$	5.4	Application of program rules
$T_{\mathcal{P},\mathcal{O}_S}$	5.7	Fixpoint operator
$ACl_A(S)$	5.9	Relativized action closure
A-feasible, A-rational, A-reasonable	5.10	Relativized status sets
$T_{\mathcal{P},\mathcal{O}_S,A}$	5.12	Relativized fixpoint operator
$A(S)$	5.13	
$A(S)$-feasible	5.10	
F-preferred	Sec.5.6 beginning	Preferred set
F/P-completion	5.6.2	F/P-completion rule of α
$Comp_{F/P}(\mathcal{P})$	Sec. 5.6.2 beginning	Augmentation of $\mathcal{P}$
F/P-complete	5.6.2	
Sem	5.7	Semantics variable
cf	5.15	Cost function