Heterogeneous Active Agents

Thomas Eiter* V.S. Subrahmanian† George Pick‡

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, and study the computation price to be paid (in terms of complexity) for such epistemic desiderata. 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. Last, but not least, we have built a simulation of a Supply Chain application in terms of our theory, building on top of commercial software systems such as Microsoft Access and ESRI's MapObject.

*Institut für Informatik, Universität Gießen, Arndtstraße 2, 35392 Gießen, Germany, and TU Vienna, Austria. E-mail: eiter@informatik.uni-giessen.de, eiter@dbai.tuwien.ac.at

†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

‡Department of Computer Science, University of Maryland, College Park, Maryland 20742. E-mail: george@cs.umd.edu

Contents

1 Introduction

Over the last few years, there has been intense work in the area of intelligent agents [57, 110]. Applications of such agent technology have ranged from intelligent news and mail filtering programs [75], to agents that monitor the state of the stock market and detect trends in stock prices, to intelligent web search agents [36], to the digital battlefield where agent technology closely monitors and merges information gathered from multiple heterogeneous information sources [4, 67, 68, 100, 108].

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, and the University of Gießen (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, joint, 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 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

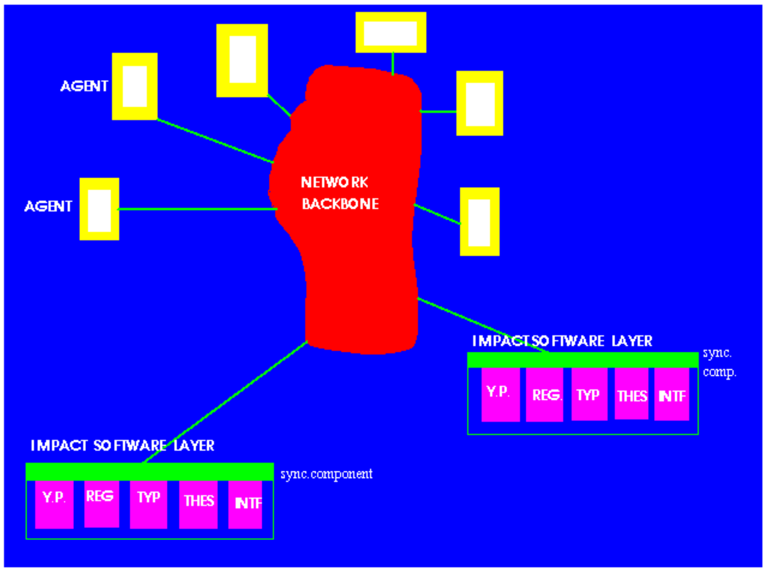


Figure 1: **IMPACT** Architecture

by an *agent program* developed in a logical agent programming language that we introduce in this paper. We are not the first to propose agent programming paradigms – several others, notably [97, 55, 45], 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 develop results on the computational cost of these alternative semantics, giving rise to a hierarchy of increasing complexity.

5. We will establish relationships between some of these semantics and existing semantical characterizations of non-monotonic logic programs.

6. We will report on an application we have built based on a simulation of our Agent Program Language and experimental results based on this implementation. This application involves specifically building agents on top of commercial software packages including Microsoft ACCESS and ESRI's MapObjects.

The organization of this paper is as follows. Section 2 presents a brief example of a multiagent system used for automated ordering of supplies by a company. A simulation of this system, adhering to the principles described in this paper, has been implemented by us. Section 3 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 4 specifies the language within which agents' actions are specified, and the language of agent programs which specifies the conditions governing an agent's behavior. 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. We discuss the advantages and disadvantages of these semantics. Section 6 introduces the reader to the assumptions under which our complexity results (presented in detail in Section 7 and 8) are obtained, and also present an overview of these results. The reader who is not interested in details of the complexity results may safely skip Sections 7 and 8, but still understand the gist of the results by reading Section 6. However, Sections 7 and 8 do contain descriptions of algorithms to compute the semantics, which the reader who is not interested in complexity may wish to read. More efficient and sophisticated algorithms that we have developed and we are currently implementing in IMPACT will be reported on in a companion paper currently in preparation [35]. Section 9 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.

Assumption. 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.

2 Motivating Examples

In this section, we present two very simple multiagent scenarios – the first can be used for automated supply chain management. The second may be used by a tax agency to take relevant actions on which tax returns should be audited, how these audits will be conducted, etc. As we go through the paper, we will revisit these examples many times.

2.1 Supply Chain Example

Supply chain management [17] is one of the most important activities in any major production company. Most production companies like to keep their production lines busy and on schedule. To ensure this, they must constantly monitor their inventory to ensure that components and items needed for creating their product are available in adequate numbers.

For instance, an automobile company is likely to want to guarantee that they always have an adequate number of tires and spark plugs in their local inventory. When the supply of tires or spark plugs drops to a certain predetermined level, the company in question must ensure that new supplies are promptly ordered. This may be done through the following steps.

- In most large corporations, the company has "standing" contracts with producers of different parts (also referred to as an "open" purchase order). When a shortfall occurs, the company contacts suppliers to see which of them can supply the desired quantity of the item(s) in question within the desired time frame. Based on the responses received from the suppliers, one or more purchase orders may be generated.
- The company may also have an existing purchase order with a large transportation provider or providers. The company may then choose to determine whether the items ordered should

be: (a) delivered entirely by truck, or (b) delivered by a combination of truck and airplane.

This scenario can be made significantly more sophisticated than the above description. For example, the company may request bids from multiple potential suppliers, the company may use methods to identify alternative substitute parts if the ones being ordered are not available, etc. We have chosen to keep the scenario relatively simple for pedagogical purposes.

The above automated purchasing procedure may be facilitated by using an architecture such as that shown in Figure 2. In this architecture, we have an Inventory-Agent that monitors the available inventory at the company's manufacturing plant. We have shown two suppliers, each of which have associated agents that monitor two databases:

- An ACCESS database specifying how much uncommitted stock the supplier has. For example, if the tuple `(widget-50,9000)` is in this relation, then this means that the supplier has 9000 pieces of widget-50 that haven't yet been committed to a consumer.
- An ACCESS database specifying how much committed stock the supplier has. For example, if the tuple `(widget-50,1000,company-A)` is in the relation, this means that the supplier has 1000 pieces of widget-50 that have been committed to company-A.

Thus, if company-B were to request 2000 pieces of widget-50, we would update the first relation, by replacing the tuple `(widget-50,9000)` by the tuple `(widget-50,7000)` and adding the tuple `(widget-50,2000,company-B)` to the latter relation – assuming that company-B did not already have widget-50 on order.

Once the Plant Agent places orders with the suppliers, it must ensure that the transportation vendors can deliver the items to the company's location. For this, it consults a Shipping Agent, which in turn consults a Truck-Agent (that provides and manages truck schedules using routing algorithms) and an Airplane-Agent (that provides and manages airplane freight cargo). As described later in Example 4.1, the Truck Agent may in fact control a set of other agents, one located on each truck. The Truck Agent we have built is constructed by building on top of ESRI's MapObject system for route mapping. These databases can be made more realistic by adding other fields – again for the sake of simplicity, we have chosen not to do so.

In this paper, we will work out exactly how the behavior of these different agents can be represented, and how they communicate with each other through messaging (though messaging is not discussed in detail in this paper). Rather, this paper focuses on how an agent *takes decisions* when it receives messages from another agent. The theory in this paper has been simulated through a prototype implementation of this supply chain example. (In fact, for space reasons, we have chosen not to describe several features of this implementation as just a few of them are enough to illustrate the concepts provided in this paper).

2.2 Tax Auditing Example

An alternative example involves a situation that may be used by 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.).

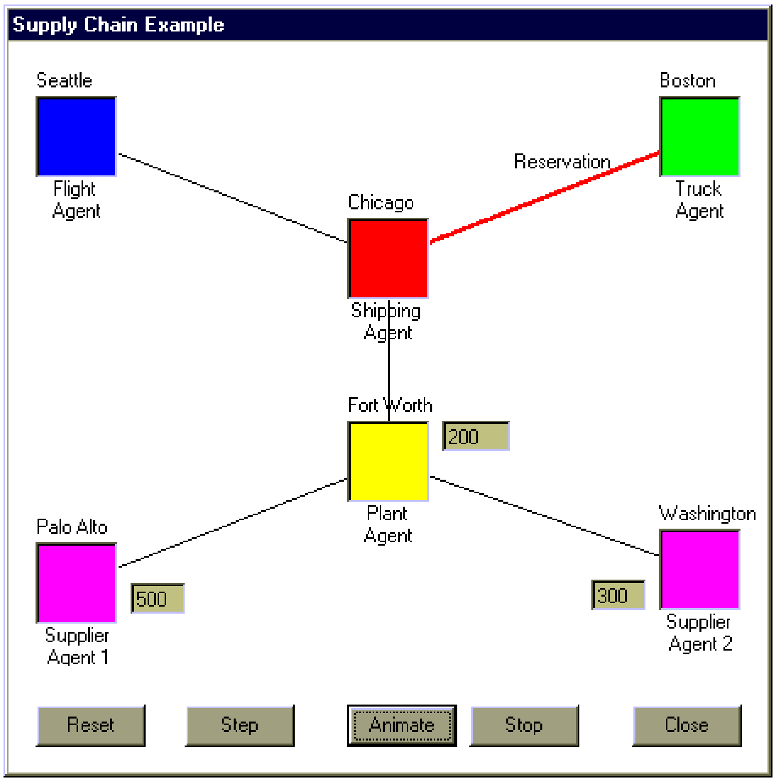


Figure 2: Agents in Supply Chain Example

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 tax-payers.
- The second relation, called `employer_declarations`, 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.).

As in the case of the Supply Chain example, this scenario is a simplified scenario, that permits pedagogical clarity when it is used to illustrate the theories and definitions introduced in this paper. In fact, we will use this example extensively throughout this paper.

3 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 existing industry standards that are consistent with this specification.

3.1 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 3.1 is not new work. It builds upon a previous effort called HERMES by one of the authors on heterogeneous data and software integration [1, 19, 74, 73, 76]. 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 $\mathtt{X}$ a *root-variable*, and the variables $\mathtt{X.f_i}$, $\mathtt{X.f_i.g}$, etc. *path-variables*. For any path variable $\mathtt{Y}$ of the form $\mathtt{X.path}$, where $\mathtt{X}$ is a root variable, we refer to $\mathtt{X}$ as the root of $\mathtt{Y}$, denoted by $root(\mathtt{Y})$); for technical convenience, $root(\mathtt{X})$, where $\mathtt{X}$ 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 3.1 (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 $\mathtt{f}$. 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:

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

- $\mathtt{face:match(mugshotdb, queryface)}$.
  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.

- $\mathtt{terrain:planroute(map1, 97, 97, 102, 103)}$.
  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 $(\mathtt{102}, \mathtt{103})$ 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 3.2 (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 $\mathtt{in(X, cc)}$ 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 [96]). 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.

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

- $\mathtt{in(X, face:match(mugshotdb, queryface))}$.
  This code call atom would succeed, instantiating `X` to any record R (having a `file` and `name` fields) whose R.`file` image matches the `queryface` image as determined by the image processing code implementing the `match` operation.

- $\mathtt{in(X, terrain:planroute(map1, 97, 97, 102, 103))}$.
  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 3.3 (code call condition) A *code call condition* is defined as follows:

1. Every code call atom is a code call condition.

2. If `s,t` are either variables or objects, then $\mathtt{s} = \mathtt{t}$ is a code call condition.

3. If `s,t` are either integers/real valued objects, or are variables over the integers/reals, then $\mathtt{s} < \mathtt{t}, \mathtt{s} > \mathtt{t}, \mathtt{s} \geq \mathtt{t}, \mathtt{s} \leq \mathtt{t}$ are code call conditions.

4. If χ_1, χ_2 are code call conditions, then $\chi_1 \ \& \ \chi_2$ is a code call condition.

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

An example of a code call condition is:

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

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. Thus, any time we attempt to actually execute a code call, the code call must be fully instantiated.

Definition 3.4 (safe code call) A code call $\mathcal{S} : \mathtt{f}(\mathtt{d_1}, \ldots, \mathtt{d_n})$ is *safe* iff each $\mathtt{d_i}$ is ground. A code call condition $\chi_1 \ \& \ldots \& \ \chi_n$, $n \geq 1$ is *safe*, 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 $\chi_{\pi(i)}$ has the form $\mathtt{s} = \mathtt{t}$ or $\mathtt{s} < \mathtt{t}$, $\mathtt{s} \leq \mathtt{t}$, $\mathtt{s} > \mathtt{t}$, $\mathtt{s} \geq \mathtt{t}$, then one of $\mathtt{s}, \mathtt{t}$ (or both) is either a constant or one of the $\mathtt{X}_{\pi(\mathtt{j})}$'s for $\mathtt{j} < \mathtt{i}$; let $\mathtt{X}_{\pi(\mathtt{i})}$ denote a possible new variable;
2. If $\chi_{\pi(i)}$ is a code call atom $\mathtt{in}(\mathtt{X}_{\pi(\mathtt{i})}, \mathtt{cc}_{\pi(\mathtt{i})})$, then for each variable $\mathtt{Y}$ occurring in $\mathtt{cc}_{\pi(i)}$, $root(\mathtt{Y})$ is from the set $\{root(\mathtt{X}_{\pi(\mathtt{j})}) \mid j < i\}$. □

For example, 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.

Definition 3.5 (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, $\mathsf{ins}_{\texttt{quadtree}}(\chi[X])$ where $\chi[X]$ 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 $\chi[X]$ is true w.r.t. the current agent state when $X = o$." 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 $\chi[X]$ of the above type), $\mathsf{ins}_a(\chi[X], \mathcal{O}_{\mathcal{S}})$ 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].

3.2 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:

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

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 3.6 (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 3.7 (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 $IC \in \mathcal{IC}$. □

4 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.

4.1 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 [82, 44, 90] and logical approaches to action [11], 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 4.1 (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* $\mathtt{X}_1,\ldots,\mathtt{X}_n$. This means that $Pre(\alpha)$ is a safe code-call condition if every variable $\mathtt{Y}$ in $Pre(\alpha)$ such that $root(\mathtt{Y}) \in \{\mathtt{X}_i \mid 1 \leq i \leq n\}$ were considered as an instantiated object (constant) from the domain. Furthermore, every code-call condition χ in $Add(\alpha) \cup Del(\alpha)$ must be safe modulo the union of $\mathtt{X}_1,\ldots,\mathtt{X}_n$ and the root variables $\mathtt{Y}_1, \ldots \mathtt{Y}_m$ occurring in $Pre(\alpha)$, i.e., it is safe if every variable $\mathtt{Y}$ in χ such that $root(\mathtt{Y}) \in \{\mathtt{X}_1 \ldots, \mathtt{X}_n, \mathtt{Y}_1, \ldots \mathtt{Y}_m\}$ were considered as though it were a constant.

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 action and their associated descriptions in specific domains.

Example 4.1 (Routing Agent Example) Consider the Truck Agent described in Section 2. This agent may coordinate the route plans and routes of multiple trucks. For each truck, it may have an intended route (sequence of points). Typically, this route will be a schedule (and hence will have temporal attributes), but we shall ignore that for the sake of simplicity. This agent may have several associated actions one of which is shown below.

$$\texttt{Replace}(\texttt{Truck1}, \texttt{Loc1}, \texttt{Truck2}, \texttt{Loc2}).$$

This action might be executed when a truck has suffered a breakdown. The truck gets replaced by a new truck. This action may be specified as follows.

- Name: $\texttt{Replace}(\texttt{Truck1}, \texttt{Loc1}, \texttt{Truck2}, \texttt{Loc2})$
- Schema: $(\texttt{Truck}, \texttt{Place}, \texttt{Truck}, \texttt{Place})$ where `Place` is a pair of integers.
- Pre:
```
in(T1,oracle:select(truckstatus,trucki_id,=,Truck1)) &
=(T1.location,Loc1) &
in(T2,oracle:select(truckstatus,trucki_id,=,Truck2)) &
=(T2.location,Loc2) &
=(T1.status,down(Loc1)) &
=(T2.status,free).
```
- Add:
```
=(T3.truckstatus,enroute(Loc2,Loc1)) &
=(T3.location,T2.location) &
=(T3.truck_id,T2.truck_id).
```
- Del:
```
in(T2,oracle:select(status,trucki_id,=,Truck2)) &
=(T2.location,Loc2) &
=(T2.status,free).
```

In the above, we are assuming that all trucks have an associated status – *free* or *enroute* or *down*. Furthermore, we are assuming that the `truckstatus` relation has only three fields – `truck_id`, `status` and `location`. If it had additional fields, then for each additional field f, we would need to add a conjunct of the form $= (\texttt{T2.f}, \texttt{T3.f})$ in the "Add" list above. Intuitively, in the above add-list, T3 is identical to T2 in all respects except for the `status` field. □

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 4.2 (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). The message box is a buffer that may be filled (when it sends a message) or flushed (when it reads the message) by the agent. In addition, we assume the existence of an operating-systems level messaging protocol (e.g. sockets or TCP/IP [109]) 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 [68] 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 $tup.\texttt{Timeop}Val$ 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 4.3 (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:**

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

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 4.2 (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 4.1, we will show that it is possible using a simple syntax to be introduced in Section 4.1 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 [82]. 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 4.3 ((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 $\Theta\Gamma(\alpha(\vec{X}),\mathcal{O}_\mathcal{S})$ 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 4.4 (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 $(\alpha(\vec{)}, \theta, \gamma)$ 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 4.5 (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 $\{(\alpha(\vec{X}), \theta, \gamma) \mid \alpha(\vec{t}) \in AS,\ \alpha(\vec{X})\theta = \alpha(\vec{t})\theta \text{ ground},\ (\theta, \gamma) \in \Theta\Gamma(\alpha(\vec{X}))\}$ of all feasible execution triples stemming from some grounded action in AS, 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 involve no “conflicts”, they are problematic when the actions in A compete for resources.

Example 4.4 (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 `_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! $\square$

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 4.6 (sequential-concurrent execution) Suppose we have a set $AS = \{(\alpha_i(\vec{X}_i, \theta_i, \gamma_i)) \mid 1 \leq i \leq n\}$ 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 $\mathcal{O}_{\mathcal{S}}^0, \ldots, \mathcal{O}_{\mathcal{S}}^n$ such that:

- $\mathcal{O}_{\mathcal{S}}^0 = \mathcal{O}_{\mathcal{S}}$ 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 $\mathcal{O}_{\mathcal{S}}^{i-1}$, and $\mathcal{O}_{\mathcal{S}}^i = apply((\vec{X}_{\pi(i)}, \theta_{\pi(i)}, \gamma_{\pi(i)}), \mathcal{O}_{\mathcal{S}}^{i-1})$.

In this case, AS is said to be π*-executable*, and $\mathcal{O}_{\mathcal{S}}^n$ 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 $\{(\alpha(\vec{X}), \theta, \gamma) \mid \alpha(\vec{t}) \in AS, \alpha(\vec{X})\theta = \alpha(\vec{t})\theta \text{ ground}, (\theta, \gamma) \in \Theta\Gamma(\alpha(\vec{X}))\}$ is S-concurrently executable on $\mathcal{O}_{\mathcal{S}}$. $\square$

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 already in rather 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 4.7 (full-concurrent execution) Suppose we have a set $AS = \{(\alpha_i(\vec{X}_i, theta_i, \gamma_i)) \mid 1 \leq i \leq n\}$ 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 $\{(\alpha(\vec{X}), \theta, \gamma) \mid \alpha(\vec{t}) \in AS, \alpha(\vec{X})\theta = \alpha(\vec{t})\theta$ ground, $(\theta, \gamma) \in \Theta\Gamma(\alpha(\vec{X}))\}$ 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 any object that may be involved in the state 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.

Theorem 4.1 *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 established 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.

4.2 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 4.8 (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 4.5 (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 4.9 (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.

4.3 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 4.10 (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. $\square$

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 [79, 3]. Moreover, the operator $\mathbf{Do}$ is in the spirit of the "praxiological" operator $E_a A$ [62], which informally means that "agent a sees to it that A is the case" [79, 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, 79]). We discuss the relationship between our approach and deontic logic in detail in Section 11.

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 4.11 (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 [102]).

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 4.12 (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 4.6 (Simple Tax Audit Agent) Here are some examples of a simple agent program P, that may be used by a tax agency. As described in Section 2, 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_declarations` 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:

O(`run_audit`(Person)) ← in(R,taxdb:select(returns,name,=,Person)),
is(AllRecs,taxdb:select(employer_declarations,name,=,person)),
is(TotalInc,taxdb:sum(AllRecs,amount)),
$R.amount \leq 0.7\times$ TotalInc.
P(`run_audit`(Person)) ← in(R,taxdb:select(returns,name,=,Person)),
is(AllRecs,taxdb:select(employer_declarations,name,=,person)),
is(TotalInc,taxdb:sum(AllRecs,amount)),
$R.amount \leq 0.8\times$ TotalInc.
F(`run_audit`(Person)) ← in(R,taxdb:select(returns,name,=,Person)),
is(AllRecs,taxdb:select(employer_declarations,name,=,person)),
is(TotalInc,taxdb:sum(AllRecs,amount)),

$R.amount \geq 0.8\times$ TotalInc.

Do(`run_audit`(Person)) ← **P**(`run_audit`(Person)) ,
in(R,taxdb:select(returns,name,=,Person)) ,
is(AllRecs,taxdb:select(employer_declarations,name,=,person)) ,
is(TotalInc,taxdb:sum(AllRecs,amount)) ,
$TotalInc \geq 200,000.$

O(send_notification(Person)) ← **Do**(`run_audit`(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 rule r of the form (3), 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 4.7 For rule r

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

we have $H(r) = \mathbf{Do}\alpha$, $B(r) = B^+(r) \cup B^-(r)$ where $B^+(r) = \{\mathbf{P}\alpha, p(X,Y)\}$, $B^-(r) = \{\neg\mathbf{Do}\gamma, \neg s(Y)\}$. Furthermore, $B_{as} = B^+_{as} \cup B^-_{as}$, where $B^+_{as} = \{\mathbf{P}\alpha\}$, $B^-_{as} = \{\neg\mathbf{Do}\gamma\}$ and $B_{cc} = B^+_{cc} \cup B^-_{cc}$, where $B^+_{cc} = \{p(X,Y)\}$ and $B^-_{cc} = \{\neg s(Y)\}$. 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.

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 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

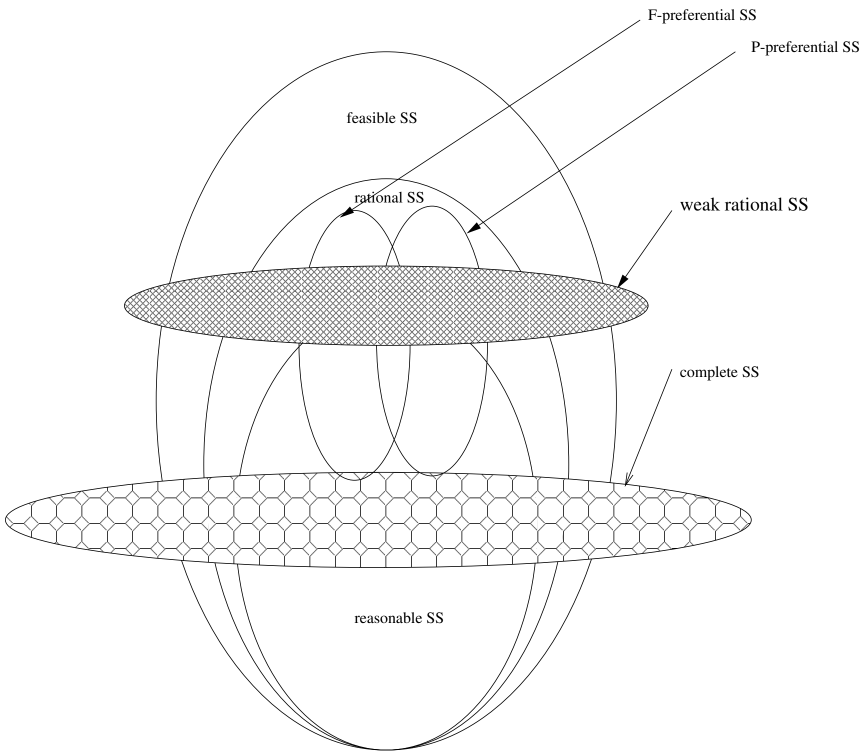


Figure 3: Relationship between different Status Sets (SS)

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 [85] (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 3 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 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. $\Box$

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. The following definition specifies how we may "close" up a status set under the rules expressed by an agent program P.

Definition 5.4 (Operator $App_{\mathcal{P},\mathcal{O}_S}(S)$**)** Suppose $\mathcal{P}$ is an agent program, and $\mathcal{O}_{\mathcal{S}}$ is an agent state. Then, $App_{\mathcal{P},\mathcal{O}_S}(S)$ is defined to be the set of all ground action status atoms A such that there exists a rule in 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 $\mathcal{O}_{\mathcal{S}}$. $\Box$

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.

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) $App_{\mathcal{P},\mathcal{O}_S}(S) \subseteq S$;

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

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

$(S4)$ (state consistency) $\mathcal{O}'_{\mathcal{S}} \models \mathcal{IC}$, 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 $\mathcal{O}_{\mathcal{S}}$. $\Box$

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 5.1 (Tax Audit Agent Revisited) Let us return to the Tax Example (cf. Example 4.6). Let us consider the case where the tax database system, `taxdb`, contains the following relations:

Relation `returns`

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

Relation `employer_declarations`

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 4.6, then $\mathbf{O}\texttt{run\_audit}(\text{John Smith})$ is implied by the program. In the same vein, $\mathbf{P}\texttt{run\_audit}(\text{Jane Shady})$ is true. Last but not least, $\mathbf{F}\texttt{run\_audit}(\text{Denis Rumble})$ 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:

$$\begin{aligned} FSS_1 \;=\; \{ \;& \mathbf{O}\texttt{run\_audit}(\text{John Smith}), \mathbf{Do}\texttt{run\_audit}(\text{John Smith}), \\ & \mathbf{P}\texttt{run\_audit}(\text{John Smith}), \mathbf{F}\texttt{run\_audit}(\text{Denis Rumble}), \\ & \mathbf{P}\texttt{run\_audit}(\text{Jane Shady})\}, \\ FSS_2 \;=\; \{ \;& \mathbf{O}\texttt{run\_audit}(\text{John Smith}), \mathbf{Do}\texttt{run\_audit}(\text{John Smith}), \\ & \mathbf{P}\texttt{run\_audit}(\text{John Smith}), \mathbf{F}\texttt{run\_audit}(\text{Denis Rumble}), \\ & \mathbf{P}\texttt{run\_audit}(\text{Jane Shady}), \mathbf{Do}\texttt{run\_audit}(\text{Jane Shady}) \}. \end{aligned}$$

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

Example 5.2 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 5.3 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 5.4 (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 4.6.

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

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 $\mathbf{P}\alpha$).

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

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" ? $\Box$

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. $\Box$

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 5.5 (Tax Example Continued) The program $\mathcal{P}$ of Example 5.4 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. $\Box$

Example 5.6 (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 former is empty, while the precondition of the latter is $free(Lane)$. The agent program contains the following rules:

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

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 $\mathbf{P}(\texttt{go\_right})$. If the right lane is free, then it contains $\mathbf{Do}(\texttt{drive}(\texttt{right\_lane}))$, and if the right lane is blocked but the left one is free, then it contains $\mathbf{Do}(\texttt{drive}(\texttt{left\_lane}))$. Only in the case where both lanes are blocked, no **Do**-atom with action `drive` is in the rational status set. This 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 5.7 Suppose that in the Tax Audit Agent example, the table `employee_declarations` 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 `add_ed`(P, C, A) and `del_ed`(P, C) for adding and deleting a record from `employee_declarations`, respectively, are available, and an agent program has the simple rule:

$$\mathbf{Do}\texttt{add\_ed}(\mathrm{JohnSmith}, \mathrm{ABC\ Inc.}, 30,000) \leftarrow \ .$$

However, in the state reported in Example 5.1 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

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

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 [51]. Therefore, we do not delve into the intricated and complex more general problem here. ∎

The fact that the program in Example 5.6 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 [70, 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,

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

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 property of feasible status sets is easily seen.

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 pre-fixpoint of $T_{\mathcal{P},\mathcal{O}_{\mathcal{S}}}$, i.e., $T_{\mathcal{P},\mathcal{O}_{\mathcal{S}}}(S) \subseteq S$.*

Clearly, 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. [70, 2]). We then have the following result.

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.*

Proof. $(\Rightarrow)$ Suppose $S = \mathit{lfp}(T_{\mathcal{P},\mathcal{O}_S})$ a rational status set of $\mathcal{P}$ on $\mathcal{O}_S$. Then, S is feasible by definition of rational status set. By Lemma 5.2, S is a pre-fixpoint of $T_{\mathcal{P},\mathcal{O}_S}$. Since $T_{\mathcal{P},\mathcal{O}_S}$ is monotone, it has by the Knaster-Tarski Theorem a least pre-fixpoint, which coincides with $\mathit{lfp}(T_{\mathcal{P},\mathcal{O}_S})$ (cf. [2, 70]). Thus, $\mathit{lfp}(T_{\mathcal{P},\mathcal{O}_S}) \subseteq S$. Clearly, $\mathit{lfp}(T_{\mathcal{P},\mathcal{O}_S})$ satisfies $(S1)$ and $(S3)$; moreover, $\mathit{lfp}(T_{\mathcal{P},\mathcal{O}_S})$ satisfies $(S2)$, as S satisfies $(S2)$ and this property is hereditary. By the definition of rational status set, it follows $\mathit{lfp}(T_{\mathcal{P},\mathcal{O}_S}) = S$.

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

Notice that in case of a positive program, $\mathit{lfp}(T_{\mathcal{P},\mathcal{O}_S})$ 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 $\mathcal{O}_S$, 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 $\mathcal{O}_S$, then $S = S'$.*

As shown by Example 5.4, 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, if we adopt reasonable underlying assumptions (see Section 7.1).

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 4.3, 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 <u>derivable</u> 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

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

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 non-monotonic 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:

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

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:

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

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 [40] which in turn was shown to have strong equivalences with the classical nonmonotonic reasoning paradigms such as default logic [86] and auto-epistemic logic[80] (see [41, 78]), as well as numerical reasoning paradigms such as linear programming and integer programming [13, 14].

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 $\mathcal{O}_{\mathcal{S}}$. $\Box$

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

Example 5.8 (Tax Audit Agent Example, Revisited) Suppose we reconsider the tax audit agent example, as described in Examples 4.6 and 5.1. However, we merely consider the two rules listed below.

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

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:

$$\begin{aligned}\mathbf{F}\mathtt{send\_refund}(Person) \quad \leftarrow \quad & \neg\mathbf{Do}\mathtt{send\_refund}(Person), \\ & in(Person, taxdb : refund\_authorized()).\end{aligned}$$

Reasonable Status Sets: Now consider the sets $S_1 = \{\mathbf{F}\mathtt{send\_refund}(\text{John Doe})\}$ and $S_2 = \{\mathbf{Do}\mathtt{send\_refund}(\text{John Doe})\}$. 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:

$$\begin{aligned}\mathbf{F}\mathtt{send\_refund}(\text{John Doe})) \quad \leftarrow \quad & \mathbf{Do}\mathtt{run\_audit}(\text{John Doe})), \\ & \neg in(\text{John Doe}), taxdb : refund\_authorized()). \\ \mathbf{Do}\mathtt{send\_refund}(\text{John Doe})) \quad \leftarrow \quad & in(\text{John Doe}), taxdb : refund\_authorized()).\end{aligned}$$

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 $\mathbf{F}\mathtt{send\_refund}(\text{John Doe}))$ 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 5.9 *For the program $\mathcal{P}$:*

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

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

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

Clearly, S is the unique reasonable status set of $red^S(\mathcal{P}, \mathcal{O}_S)$, 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

$$\mathbf{Do}\alpha \leftarrow \neg\mathbf{F}\alpha \tag{4}$$

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 $\mathcal{O}_{\mathcal{S}}$.*

Proof. 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 $red^S(\mathcal{P}, \mathcal{O}_{\mathcal{S}})$, 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 $red^S(\mathcal{P}, \mathcal{O}_{\mathcal{S}})$ 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 contradiction.

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

In a follow-up paper [35], 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* [65], 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, 79, 54]. 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 5.10 (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 is scenario is described by the program $\mathcal{P}$:

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

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. $\Box$

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. $\Box$

The following example illustrates this definition.

Example 5.11 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:

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

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 $\mathcal{O}_{\mathcal{S}}$ 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. □

An immediate consequence of this 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 $\mathcal{O}_{\mathcal{S}}$, then the weak rational (resp., weak reasonable) status sets of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$ coincide with the rational (resp., reasonable) status sets of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$.*

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

Example 5.12 (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 S 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 $T_{\mathcal{P},\mathcal{O}_S}(S)$, 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 $T_{\mathcal{P},\mathcal{O}_S,A}$ is monotone and continuous if $\mathcal{P}$ is positive, and hence has a least fixpoint $lfp(T_{\mathcal{P},\mathcal{O}_S,A}) = \bigcup_{i=0}^{\infty} T^i_{\mathcal{P},\mathcal{O}_S,A}$ where $T^0_{\mathcal{P},\mathcal{O}_S,A} = \emptyset$ and $T^{i+1}_{\mathcal{P},\mathcal{O}_S,A} = T_{\mathcal{P},\mathcal{O}_S,A}(T^i_{\mathcal{P},\mathcal{O}_S,A})$, 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 $\mathcal{O}_S$ be an agent state. Then, a status set S is an A-rational status set of $\mathcal{P}$ on $\mathcal{O}_S C$, if and only if $S = lfp(T_{\mathcal{P},\mathcal{O}_S,A})$ and S is an A-feasible status set.*

Proof. 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 $T_{\mathcal{P},\mathcal{O}_S,A}$. ∎

From the previous theorem, we obtain the following result.

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

Proof. 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 $S = lfp(T_{\mathcal{P},\mathcal{O}_S,A})$ and S is A-feasible; the result follows. ∎

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 $\mathcal{O}_S$ 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.*

Proof. 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. ∎

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 $\mathcal{O}_S$. 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 $T_{\mathcal{P},\mathcal{O}_S,A}$ 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, $T_{\mathcal{P},\mathcal{O}_S,A_1}(S) \subseteq T_{\mathcal{P},\mathcal{O}_S,A_2}(S)$ holds, and $lfp(T_{\mathcal{P},\mathcal{O}_S,A_1}) \subseteq lfp(T_{\mathcal{P},\mathcal{O}_S,A_2})$.*

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) $S = lfp(T_{\mathcal{P},\mathcal{O}_S,A})$ and S is A-feasible for $A = A(S)$, and (ii) for each ground action $\alpha \notin A(S)$, the status set $S_{A'} = lfp(T_{\mathcal{P},\mathcal{O}_S,A'})$ is not A'-feasible, where $A' = A(S) \cup \{\alpha\}$.*

Proof. $(\Rightarrow)$ If S is weak rational, then (i) follows from Theorem 5.9 and Proposition 5.10. Suppose for some A' in (ii), $S_{A'} = lfp(T_{\mathcal{P},\mathcal{O}_S,A'})$ 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 $S_{A''} = lfp(T_{\mathcal{P},\mathcal{O}_S,A''})$ 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. ∎

For a fixed program $\mathcal{P}$, this criterion implies 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 (see Section 7.1.1).

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

$$\{\alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)\} \hookleftarrow \chi \tag{5}$$

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 $\texttt{nil}$ 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. $\square$

Example 5.13 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. $\Box$

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 5.14 The program

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

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. $\Box$

Example 5.15 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. $\Box$

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

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

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. $\Box$

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. E.g., if we consider the Supply Chain Example, 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 Plant 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 Plant 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:

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

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)` 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.

6 Algorithms and Complexity Issues

In this section, we address the computational complexity of agent programs. We assume that the reader is familiar with the basic concepts of complexity theory, in particular with NP-completeness and the polynomial hierarchy, and refer to [37, 61, 83] for background material on this subject and for concepts and notation that we use in the remainder of this paper.

Our aim is a sharp characterization of the complexity of different computational tasks which arise in the context of agent programs. Such a characterization is useful in many respects. First of all, it tells us whether certain problems are tractable or intractable in the worst case. However, beyond such a coarse classification, the precise complexity gives us a hint of which type of algorithm is appropriate for implementing solutions to a problem, and how many sources of complexity have to be eliminated in order to ensure tractability.

In the rest of this section, we state the assumptions that we make for our analysis, and we present an overview and a discussion of the results that we derive. The results are then established in Sections 7 and 8.

6.1 Underlying assumptions

Any reasonable measurement of the computational complexity of agent programs must be based on underlying assumptions. We consider here the evaluation of a fixed agent program $\mathcal{P}$ in the context of software code $\mathcal{S}$, an action base $\mathcal{AB}$, action constraints $\mathcal{AC}$, and integrity constraints $\mathcal{IC}$, each of which is fixed, over varying states $\mathcal{O}_{\mathcal{S}}$. This corresponds to what is commonly called the *data complexity* of a program [104]. If we consider varying programs where the agent state is fixed (resp., varying), we would have *expression (or program) complexity and combined complexity*, which are typically one exponential higher than data complexity. This also applies in many cases to the results that we derive below; such results can be established using the complexity upgrading techniques for expression complexity described in [50].

Of course, if we use software packages $\mathcal{S} = (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S})$ with high intrinsic complexity, then the evaluation of agent programs will also be time consuming, and leaves us no chance to build efficient algorithms. We therefore have to make some general assumptions about the software package used such that polynomial time algorithms are not a priori excluded.

We adopt a generalized *active domain* assumption on objects, in the spirit of domain closure; all objects considered for grounding the program rules, evaluation of the action preconditions, the conditions of the actions constraints and the integrity constraints must be from $\mathcal{O}_\mathcal{S}$, or they must be constructible from objects therein by operations from a fixed (finite) set in a number of steps which is bounded by some constant, and such that each operation is efficiently executable (i.e., in polynomial time) and involves only a number of objects bounded by some other constant. Notice that the active domain assumption is often applied in the domain of relational databases, and similar domain closure in the context of knowledge bases. In our framework, creation and use of tuples of bounded arity from values existing in a database would be a feasible object construction process, while creation of an arbitrary relation (as an object that amounts to a set of tuples) would be not.

Under this assumption, the number of objects which may be relevant to a fixed agent program $\mathcal{P}$ on a given state $\mathcal{O}_\mathcal{S}$ is bounded by a polynomial in the number of objects in $\mathcal{O}_\mathcal{S}$, and each such object can be generated in polynomial time. In particular, this also means that the number of ground rules of $\mathcal{P}$ which are relevant is polynomial in the size of $\mathcal{O}_\mathcal{S}$, measured by the number of objects that it contains.

Let us further assume that the evaluation time of code condition calls χ over a state $\mathcal{O}_\mathcal{S}$, for any particular legal assignment of objects, is bounded by a polynomial in the size of $\mathcal{O}_\mathcal{S}$. Moreover, we assume that given an agent state $\mathcal{O}_\mathcal{S}$ and a set of ground actions $\mathcal{A}$, the state $\mathcal{O}'_\mathcal{S}$ which results under weakly-concurrent execution of $\mathcal{A}$ on $\mathcal{O}_\mathcal{S}$ (see Definition 4.5) is constructible in polynomial time.

As a consequence of these assumptions, the action and integrity constraints are evaluable on an agent state $\mathcal{O}_\mathcal{S}$ under the generalized active domain semantics in polynomial time, and the integrity constraints on the agent state $\mathcal{O}'_\mathcal{S}$ resulting from the execution of a set of actions A grounded in the active domain, are checkable in polynomial time in the size of $\mathcal{O}_\mathcal{S}$.[1]

Notice that these assumptions will be met in many software packages which support the use of integrity constraints (e.g., a relational database). If evaluation of the code condition calls or constraints were not polynomial, then the evaluation of the agent program would not be either.

6.2 Problems Whose Complexity is Studied / Overview of Complexity Results

The complexity results we derive may be broken up into two parts. In the first part, we assume that fixed *positive* agent programs are considered. In the second part, this assumption is relaxed, and general, i.e., not necessarily positive agent programs are considered.

In this paper, we study four types of complexity problems. For each concrete semantics introduced in the paper, we study the complexity of these four problems. This leads to Tables 1 and 2 which summarize the results, under different assumptions on the syntax of the agent programs considered. The computational problems that we study are listed below. Let Sem be any kind of status sets.

[1]This would remain true if the integrity constraints where arbitrary fixed first-order formulas (evaluated under active domain semantics).

(consistency) Deciding the consistency of the program on a given agent state, i.e., existence of an Sem-status set;

(recognition) the recognition of a Sem-status set;

(computation) the computation of an arbitrary Sem-status set; and

(action reasoning) reasoning about whether the agent takes an action α under Sem-status sets, both under the

- possibility variant (decide whether α is executed according to some Sem-status set), and the
- certainty variant (decide whether α is executed according to every Sem-status set).

The range of Sem for which these problems have been analyzed is listed in the leftmost column of Tables 1 and 2. Particular attention has been paid to the presence or absence of integrity constraints.

Table 1 specifies the complexity of the four problems that we study when positive agent programs are considered, while Table 2 specifies their complexity when arbitrary agent programs are considered. As Tables 1 and 2 contain many complexity classes that the casual AI researcher may not be familiar with, we present in Figure 4 a graphical representation of these classes – the reader interested in a more detailed discussion of these classes will find them in Section 6.3. Generally speaking, in Figure 4, we draw an arc from complexity class C1 to complexity class C2, if the hardest problems in C_1 are "easier" to solve than the hardest problems in C2 (assuming that the classes in question do not collapse). Thus, for instance, the existence of an edge from the class P to the class NP indicates that the hardest problems in P, which are those complete for P, are easier to solve as compared to the hardest problems in NP, i.e., the NP-complete problems (unless $\mathrm{P} = \mathrm{NP}$).

Note on Tables: The entries for decision problems in Tables 1 and 2 stand for completeness for the respective complexity classes. In case of P, hardness may implicitly be present with costly object construction operations. However, we remark that for all problems except recognition of a feasible status set, hardness holds even if no new objects are introduced and the agent state consists merely of a relational database. Proof of these results are not difficult, using the well-known result that inference from a datalog program (Horn logic program) is P-complete, cf. [28].

The entries for the computation problem are the classes $\mathrm{F}\mathcal{C}$ from the literature considered e.g. in [83, 23, 24] (i.e., compute any arbitrary solution to a problem instance; see subsection 6.3 for more details). Unless stated otherwise, they stand for completeness under an appropriate notion of polynomial time reduction as used in [83, 23]. Observe that we have aimed at characterizing in this paper the complexity of agent programs in terms of existing classes from the literature, rather than introducing new classes to precisely assess the complexity of some problems.

6.2.1 Bottom Line for the Computation Problem

Of all the four problems described above, from the point of view of the IMPACT system (and in general, for any system that attempts to determine which actions an agent must take), the most important problem, by far, is the problem of *Computation* – given an agent program, a current agent state, a set of integrity constraints and action constraints, determine a set of actions that the agent must take. This task forms the single most important task that an agent must take, over and over again.

$\mathcal{IC} = \emptyset \mid \mathcal{IC}$ arbitrary	consistency	computation	recognition	action reasoning possible	certain
feasible	P $\mid$ NP	FP $\mid$ FNP	P	NP	co-NP
rational ≡ reasonable ≡ F-pref. rational ≡ F-pref. reasonable	P	FP	P	P	P
weak rational ≡ weak reasonable	P $\mid$ NP	FP $\mid$ $\mathrm{FNP//log}$ *	P $\mid$ co-NP	NP	co-NP $\mid$ Π_2^P

* … hard for both FNP and $\mathrm{FP}_{\parallel}^{\mathrm{NP}}$

Table 1: Complexity of fixed positive agent programs

$\mathcal{IC} = \emptyset \mid \mathcal{IC}$ arbitrary	consistency	computation	recognition	action reasoning possible	certain
feasible	NP	FNP	P	NP	co-NP
rational	NP $\mid$ Σ_2^P	$\mathrm{FNP//log}$ * $\mid$ $\mathrm{F}\Sigma_2^P$	co-NP	Σ_2^P	co-NP $\mid$ Π_2^P
reasonable	NP	FNP	P	NP	co-NP
F-pref. rational	NP $\mid$ Σ_2^P	$\mathrm{FNP//log} \mid \mathrm{FP}^{\Sigma_2^P} \cap \mathrm{RP} \cdot \mathrm{FP}_{\parallel}^{\Sigma_2^P}$ †	co-NP $\mid$ Π_2^P	$\Sigma_2^P \mid \Sigma_3^P$	$\Pi_2^P \mid \Pi_3^P$
F-pref. reasonable	NP	$\mathrm{FNP//log}$	co-NP	Σ_2^P	Π_2^P
weak rational	NP $\mid$ Σ_2^P	$\mathrm{FNP//log} \mid \mathrm{FP}^{\Sigma_2^P} \cap \mathrm{RP} \cdot \mathrm{FP}_{\parallel}^{\Sigma_2^P}$ †	co-NP $\mid$ Π_2^P	$\Sigma_2^P \mid \Sigma_3^P$	$\Pi_2^P \mid \Pi_3^P$
weak reasonable	NP	$\mathrm{FNP//log}$	co-NP	Σ_2^P	Π_2^P

* … hard for both FNP and $\mathrm{FP}_{\parallel}^{\mathrm{NP}}$

† … hard for both $\mathrm{F}\Sigma_2^P$ and $\mathrm{FP}_{\parallel}^{\Sigma_2^P}$

Table 2: Complexity of fixed agent programs with negation

When considering the different semantics for agent-programs, we easily notice (by examining the column "computation" in both Tables 1 and 2), that the easiest semantics to compute are given as follows:

- When **positive agent programs with no integrity constraints** are considered, the rational, weak rational, reasonable, weak reasonable, F-preferential, and P-preferential semantics are the easiest to compute, all falling into the same complexity class. The other semantics are harder to compute. Thus, in this case, we have some flexibility in choosing that out of the rational, weak rational, reasonable, weak reasonable, F-preferential, and P-preferential, that best meets the agent's epistemic needs. Note that different agents in IMPACT can use different semantics.
- When **positive agent programs with integrity constraints** are considered, the best semantics, from the point of view of computational complexity, are the rational, reasonable, F-preferential, and P-preferential semantics. Note that unlike the previous case, the weak rational and weak reasonable semantics are harder to compute when integrity constraints are present.
- **When arbitrary agent programs with no integrity constraints** are considered, then the easiest semantics to compute are the feasible set semantics and the reasonable status set semantics. All other semantics are harder to compute.
- **When arbitrary agent programs with integrity constraints** are considered, the same continues to be true.

In general, when considering how to compute a kind of status set, the reasonable status set semantics is generally the easiest to compute, irrespective of whether agent programs are positive or not, and irrespective of whether integrity constraints are present or not. As we have argued earlier on in the paper, reasonable status sets have many nice properties which might make them epistemologically preferable to feasible status sets and rational status sets.

6.3 Different Complexity Classes

In this subsection, we briefly describe the various complexity classes encountered earlier in this section. We also analyze the causes for the various complexity trends we notice.

The classes that we use in our characterizations are summarized in Figure 4. An edge directed from class C_1 to class C_2 indicates that all problems in C_1 can be efficiently transformed into some problem in C_2, and that it is strongly believed that a reduction in the other direction is not possible; i.e., the hardest problems in C_2 are more difficult than the problems in C_1.

The decision classes are from the polynomial hierarchy, which is built on top of the classes P and NP (=Σ_1^P), by allowing the use of an oracle (i.e., a subprogram) for deciding problems instantaneously. The class C to which this oracle must belong is denoted in a superscript; e.g., P^{NP} (resp., $\mathrm{NP}^{\mathrm{NP}}$) is the class of problems solvable in polynomial time on a deterministic (resp., nondeterministic) Turing machine, if an oracle for a problem in NP may be used. For the decisional classes, the arcs in Figure 4 actually denote inclusions, i.e., the transformation of problems in C_1 to problems in C_2 is by means of the identity.

The classes for search problems, which are often also called function classes, can be found in [83, 24]. A search problem is a generalization of a decision problem, in which for every instance

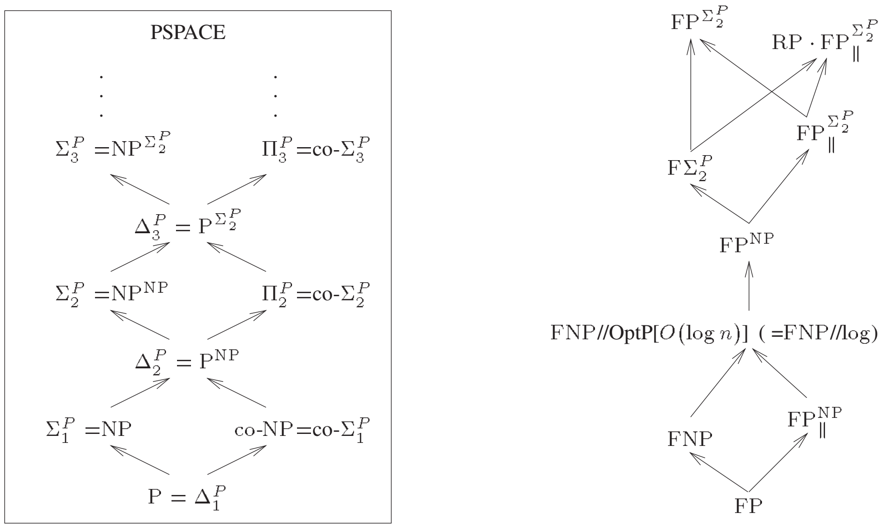


Figure 4: Decision (left) and search (right) problem complexity classes

of the problem a (possibly empty) set of solutions exists. To solve such a problem, an algorithm must (possibly nondeterministically) compute an arbitrary solution out of this set, if it is not empty. Decision problems can be viewed as particular search problems, in which the solution set is either empty or the singleton set $\{$yes$\}$. More formally, search problems in the classes from Figure 4 are solved by transducers, i.e., Turing machines equipped with an output tape. If the machine halt in an accepting state, then the contents of the output tape is the result of the computation. Observe that a nondeterministic machine computes a (partial) multi-valued function. Thus, not all arcs in Figure 4 mean inclusion, i.e., trivial reducibility by the identity.

The concept of reduction among search problems Π_1 and Π_2 is also obtained by a proper generalization of the respective concept for decision problems. Π_1 is (polynomial-time) reducible to Π_2, if (i) from every instance I of Π_1, an instance $f(I)$ of Π_2 is constructible in polynomial time, such that $f(I)$ has some solution precisely if I has; and (ii) from every solution S of $f(I)$, a solution of I can be constructed in time polynomial in the size of S and I.

For both decision and search problem classes C, a problem is complete for C, if it belongs to C and is hard for C, i.e., every problem in C reduces to it.

The search problem counterparts of the classes C in the polynomial hierarchy are often denoted by a prefixed "F"; some of them appear in Figure 4. The other classes $\mathrm{FP}_{\|}^{\mathrm{NP}}$ and $\mathrm{FP}_{\|}^{\Sigma_2^P}$ are the search problem counterparts of the classes $\mathrm{P}_{\|}^{\mathrm{NP}}$ and $\mathrm{P}_{\|}^{\Sigma_2^P}$, respectively, which are not shown in the figure. These classes contain the problems which can be solved in polynomial time on a deterministic Turing machine which has access to an oracle in NP (resp., Σ_2^P), but where all queries to the oracle must be prepared before issuing the first oracle call. Thus, the oracle calls are nonadaptive and must essentially take place in parallel; it is commonly believed that this restricts computational power.

$\mathrm{FNP//log}$ is short for the class FNP//OptP[$O(\log n)$] [24], which is intuitively the class of problems for which a solution can be nondeterministically found in polynomial time for an instance I, if the optimal value $opt(I)$ of an NP optimization problem on I is known, where $opt(I)$ must have $O(\log |I|)$ bits. E.g., computing a clique of largest size in a given graph is a problem in $\mathrm{FNP//log}$. The class $\mathrm{FNP//log}$ reduces to $\mathrm{FP}^{\mathrm{NP}}$ and roughly amounts to a randomized version of $\mathrm{FP}_{\|}^{\mathrm{NP}}$. Due to its nondeterministic nature, it contains problems which are not known to be solvable in $\mathrm{FP}^{\mathrm{NP}}_{\|}$. The most prominent of these problems is the computation of an arbitrary model of a propositional formula [59], which is the prototypical problem complete for the class FNP. Few natural $\mathrm{FNP}//\log$-complete problems are known to date; our analysis contributes some new such problems, which are from the realm of practice rather than artificially designed.

In the context of action programs, computing a weak rational status set for a positive program is in $\mathrm{FNP}//\log$, since if we know the maximum size s of a set A of ground actions such that an A-rational status S set exists, then we can nondeterministically generate such an S in polynomial time. The computation of s amounts to an NP optimization problem, and thus the overall algorithm places the problem in FNP//$\log$.

The class $\mathrm{RP} \cdot \mathrm{FP}^{\Sigma_2^P}_{\|}$ [24] contains, intuitively speaking, those problems for which a solution on input I can be found by a random polynomial time algorithm with very high probability, by using a problem in $\mathrm{FP}^{\Sigma_2^P}_{\|}$ as single call subroutine. This class is above $\mathrm{FP}^{\Sigma_2^P}_{\|}$. Chen and Toda have shown that many optimization problems belong to this class whose solutions are the maximal solutions of an associated problem for which solution recognition is in co-NP. Computation of an F-preferred rational and a weak rational status set matches this scheme, which means that the problems are in $\mathrm{RP} \cdot \mathrm{FP}^{\Sigma_2^P}_{\|}$.

The results show that the complexity of agent programs varies from polynomial up to the third level of the polynomial hierarchy. Observe that in some cases, there are considerable complexity gaps between positive agent programs and agent programs which use negation (e.g., for F-preferred rational status sets).

The reason for this gap are three sources of complexity, which lift the complexity of positive agent programs from P up to Σ_3^P and Π_3^P, respectively (in the cases of F-preferred and weak rational status sets):

1. an (in general) exponential number of candidates for a feasible (resp., weak feasible) status set;
2. a difficult recognition test, which involves groundedness; and
3. an exponential number of preferable candidates, in terms of F-preference or maximal obedience to obligations.

These three sources of complexity act in a way orthogonally to each other; all of them have to be eliminated to gain tractability.

For the canonical semantics of positive agent programs, the rational status set semantics, all computational problems are polynomial. This contrasts with feasible status sets, for which except recognition, all problems are intractable. On the other hand, under the weak status set semantics, the problems apart from action reasoning are polynomial, if no integrity constraints are present; intractability, however, is incurred in all problems as soon as integrity constraints may be used.

It is interesting to observe that for programs with negation, rational status sets are more expensive to compute than reasonable status sets in general, and this is true if no integrity constraints are present, except for consistency checking and cautious action reasoning. A similar observation applies to the F-preferred and weak variants of rational and reasonable status sets in the general case; here, the rational variants are always more complex than the reasonable ones. However, somewhat surprisingly, if no integrity constraints are present, then the complexities of the rational and reasonable variants coincide! This is intuitively explained by the fact that in absence of integrity constraints, the expensive groundedness check for rational status sets can be surpassed in many places, by exploiting the property that in this case, every feasible status set must contain some rational status set.

Another interesting observation is that for programs with negation, the preferential and weak variant of rational status sets have the same complexity characteristics, and similar for reasonable status sets. These semantics have a dual computational nature; preference effects minimization of the $\mathbf{F}$-part of the status set, while weakness effects maximization of the $\mathbf{Do}$-part.

Presence of integrity constraints, even of the simplest nature which is common in practice (e.g., functional dependencies [102] in a database), can have a detrimental effect on (variants of) rational status sets and raises the complexity by one level in the polynomial hierarchy. However, the complexity of reasonable status sets and their variants is immune to integrity constraints except for the weak reasonable status sets on positive programs. Intuitively, this is explained by the fact that the refutation of a candidate for a reasonable status set basically reduces to the computation of the rational status set of a positive agent program, and there integrity constraints do not increase the complexity. In the case of weak reasonable status sets for positive programs, we have an increase since the weakness condition may create an exponential number of candidates if the program is inconsistent.

We finally remark that we have omitted here an analysis of the complexity of optimal status sets as proposed in Section 5.7, in order to avoid an abundance of complexity results. Based on the results presented above, coarse bounds are staightforward. The sources [23, 24, 59] and references therein provide suitable complexity classes for a more accurate assessment.

7 Complexity Results and Algorithms for Agent Programs: Basic Results

This section contains the first part of the derivation of the complexity results which have been presented in Section 6. The focus in this section is on the base case, in which we have programs without integrity constraints (though cases where results on integrity constraints follow as immediate extensions of the no-integrity-constraint case are also included). As the Table 1 and 2 show, in general the presence of integrity constraints has an effect on the complexity of some problems, while it has not for others. For the latter problems, we discuss this effect in detail in the next section. In this section, as complexity results are discussed, we also develop algorithms for various status set computations.

7.1 Positive programs

The most natural question is whether a feasible status set exists for program $\mathcal{P}$ on a given state $\mathcal{O}_\mathcal{S}$. As we have seen, this is not always the case. However, for fixed positive programs, we can always efficiently find a feasible status set (so one exists), and moreover, even a rational status set, measured in the size of the input $\mathcal{O}_\mathcal{S}$. This is possible using the algorithm COMPUTE-P-RSS below, where the program $\mathcal{P}$ and possibly integrity and action constraints are in the background.

Theorem 7.1 *Let $\mathcal{P}$ be a fixed positive agent program. Then, given an agent state $\mathcal{O}_\mathcal{S}$, the unique rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ (so it exists) is computed by* COMPUTE-P-RSS *in polynomial time.*

Algorithm COMPUTE-P-RSS

Input: agent state $\mathcal{O}_{\mathcal{S}}$ (fixed positive agent program $\mathcal{P}$);

Output: the unique rational status set of $\mathcal{P}$, if it exists; "No", otherwise.

Method

Step 1. Compute $S = \mathit{lfp}(T_{\mathcal{P},\mathcal{O}_{\mathcal{S}}})$;

Step 2. Check whether S satisfies conditions $(S2)$ and $(S4)$ of a feasible status set;

Step 3. If S satisfies $(S2)$ and $(S4)$, then output S; otherwise, output "No". Halt.

Moreover, if $\mathcal{IC} = \emptyset$, then deciding whether $\mathcal{P}$ has some feasible status set on $\mathcal{O}_{\mathcal{S}}$ as well as computing any such status set, is possible in polynomial time using COMPUTE-P-RSS.

Proof. By Theorem 5.3 and the fact that S satisfies $(S1)$ and $(S3)$, algorithm correctly computes the unique rational status set of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$.

By the assumptions that we made at the beginning of this section, Step 1 can be done in polynomial time, since a fixed $\mathcal{P}$ amounts to a ground instance which is polynomial in the size of $\mathcal{O}_{\mathcal{S}}$, and we can compute $S = \mathit{lfp}(T_{\mathcal{P},\mathcal{O}_{\mathcal{S}}})$ bottom up by evaluating the sequence $T^i_{\mathcal{P},\mathcal{O}_{\mathcal{S}}}$, $i \geq 0$, until the fixpoint is reached.

Observe that, of course, checking $(S2)$ (action and deontic consistency) –or part of this criterion– in algorithm COMPUTE-P-RSS can be done at any time while computing the sequence $T^i_{\mathcal{P},\mathcal{O}_{\mathcal{S}}}$, and the computation can be stopped as soon as an inconsistency is detected.

Step 2, i.e., checking whether S satisfies the conditions $(S2)$ and $(S4)$ is, by our assumptions, possible in polynomial time. Therefore, for fixed $\mathcal{P}$ (and tacitly assumed fixed action and integrity constraints in the background), algorithm COMPUTE-POS-RATIONAL-SS runs in polynomial time.

If $\mathcal{IC} = \emptyset$, then by Proposition 5.5, $\mathcal{P}$ has a feasible status set on $\mathcal{O}_{\mathcal{S}}$ iff S is a feasible status set. Therefore, deciding the existence of a feasible status set (and computing one) can be done by using COMPUTE-P-RSS in polynomial time $\mathcal{IC} = \emptyset$. ∎

Corollary 7.2 *Let $\mathcal{P}$ be a fixed positive agent program. Then, given an agent state $\mathcal{O}_{\mathcal{S}}$ and a status set S, deciding whether S is a rational status set of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$ is polynomial.*

Since for every positive agent program $\mathcal{P}$, the rational status set, the reasonable status set, and the preferred one among them coincide, we have the following easy corollary.

Corollary 7.3 *Let $\mathcal{P}$ be a fixed positive agent program. Given an agent state $\mathcal{O}_{\mathcal{S}}$, the unique F-preferred rational (resp., reasonable) status set of $\mathcal{P}$ (so it exists) can be computed and recognized in polynomial time.*

7.1.1 Weak rational status sets

In this subsection, we address the problem of computing a weak rational status set for a positive program. As we have mentioned in Section 5.4, for a fixed positive agent program $\mathcal{P}$, it is possible to compute a weak rational status set on a given agent state $\mathcal{O}_{\mathcal{S}}$ in polynomial time, provided that the no integrity constraints are present.

In fact, this is possible by using the following algorithm.

Algorithm COMPUTE-P-WEAK-RSS

Input: agent state $\mathcal{O}_{\mathcal{S}}$ (fixed positive agent program $\mathcal{P}$; $\mathcal{IC} = \emptyset$)

Output: a weak rational status set of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$, if one exists; "No", otherwise.

Method

Step 1. Set $A := \emptyset$, $GA :=$ set of all ground actions, and compute $S := \mathit{lfp}(T_{\mathcal{P},\mathcal{O}_{\mathcal{S}},A})$;

Step 2. If S is not A-feasible, then output "No" and halt; otherwise, set $A := A(S)$ and $GA := GA \setminus A(S)$;

Step 3. If $GA = \emptyset$, then output S and halt;

Step 4. Choose some ground action $\alpha \in GA$, and set $A' := A \cup \{\alpha\}$;

Step 5. If $S' := \mathit{lfp}(T_{\mathcal{P},\mathcal{O}_{\mathcal{S}},A'})$ is A'-feasible, then set $A := A(S')$, $GA := GA \setminus A(S')$, and $S := S'$; continue at Step 3.

The next result states that this algorithm is correct and works in polynomial time.

Theorem 7.4 *Given a positive program $\mathcal{P}$ and an agent state $\mathcal{O}_{\mathcal{S}}$, algorithm* COMPUTE-P-WEAK-RSS *correctly outputs a weak rational status set of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$ (so one exists) if $\mathcal{IC} = \emptyset$. Moreover, for fixed $\mathcal{P}$,* COMPUTE-P-WEAK-RSS *runs in polynomial time.*

We remark that this simple algorithm can be speeded up by exploiting some further properties. In Step 5 of the algorithm, the computation of S' can be done by least fixpoint iteration starting from S rather than from the empty set (cf. Proposition 5.12).

The previous result shows that we can compute an arbitrary weak rational status set in polynomial time under the asserted restrictions. However, this does not mean that we can decide efficiently whether a given status set is a weak rational status set. The next result shows that this is in fact possible.

Theorem 7.5 *Let $\mathcal{P}$ be a fixed positive agent program, and suppose $\mathcal{IC} = \emptyset$. Let $\mathcal{O}_{\mathcal{S}}$ be an agent state, and let S be a status set. Then, deciding whether S is a weak rational status set of $\mathcal{P}$ is polynomial.*

Proof. By Proposition 5.10, S must be $A(S)$-feasible if it is a weak rational status set; since for any set of ground actions A, testing A-feasibility is not harder than testing feasibility, by Proposition 7.7 we obtain that this condition can be tested in polynomial time.

If S is $A(S)$-feasible, then, since $\mathcal{P}$ is positive and $\mathcal{IC} = \emptyset$, by Theorem 5.13 S is a weak rational status set, iff for every ground action $\alpha \notin A(S)$, the status set $S' = lfp(T_{\mathcal{P},\mathcal{O}_S,A'})$ is not A'-feasible, where $A' = A \cup \{\alpha\}$. For each such α, this condition can be checked in polynomial time, and there are only polynomially many such α; hence, the required time is polynomial. Consequently, the overall recognition test is polynomial.

We remark that algorithm COMPUTE-P-WEAK-RSS can be modified to implement the recognition test; we omit the details, however. ∎

Since in general, multiple weak rational status sets may exist, it appears important to know whether some action status atom A belongs to all (resp., some) weak rational status set. This corresponds to what is known as certainty (resp., possibility) reasoning in databases [102], and to cautious (resp., brave) reasoning in the area of knowledge representation [47]. In particular, this question is important for an atom $\mathbf{Do}(\alpha)$, since it tells us whether α is possibly executed by the agent (if (s)he picks nondeterministically some weak rational status set), or executed for sure (regardless of which action set is chosen). Unfortunately, these problems are intractable.

Theorem 7.6 *Let $\mathcal{P}$ be a fixed positive agent program $\mathcal{P}$, and suppose $\mathcal{IC} = \emptyset$. Let $\mathcal{O}_S$ be a given agent state and let α be a given ground action atom. Then, deciding whether $\alpha \in \mathbf{Do}(S)$ holds for (i) some weak rational status set (resp., (ii) every weak rational status set) of $\mathcal{P}$ on $\mathcal{O}_S$ is* NP-*complete (resp.,* co-NP-*complete).*

Proof. For (i), observe that algorithm COMPUTE-P-WEAK-RSS is nondeterministically complete, i.e., every weak rational status set S is produced upon proper choices in Step 4. Therefore, by checking $\mathbf{Do}(\alpha) \in S$ (resp., $\mathbf{Do}(\alpha) \notin S$) before termination, we obtain membership in NP (resp., co-NP).

For the hardness part, we present a reduction from the complement of a restricted version of the satisfiability problem (SAT) for (ii); a similar reduction for (i) is left as an exercise to the reader.

Suppose that $\phi = \bigwedge_{i=1}^{m} C_i$ is conjunction of propositional clauses such that each clause C_i is a disjunction $C_i = L_{i,1} \vee L_{i,2} \vee L_{i,3}$ of three literals over atoms $X = \{x_1, \ldots, x_n\}$. Then, deciding whether ϕ is satisfiable is a well-known NP-complete problem. This remains true if we assume that for each clause C_i, either all literals in it are positive or all are negative; this restriction is known as monotone 3SAT (M3SAT).

In our reduction, we store the formula ϕ in a database $\mathcal{D}$. For this purpose, $\mathcal{D}$ is supposed to have two relations POS(V_1, V_2, V_3) and NEG(V_1, V_2, V_3), in which the positive and negative clauses of ϕ are stored, and a relation VAR(V) which contains all variables. For each positive clause C_i, there exists a tuple with the variables of C_i in POS, e.g., for $x_1 \vee x_4 \vee x_2$ the tuple (x_1, x_4, x_2), and likewise for the negative clauses a tuple with the variables in NEG, e.g., for $\neg x_3 \vee \neg x_1 \vee \neg x_2$ the tuple (x_3, x_1, x_2).

The action base $\mathcal{AB}$ contains the actions $\mathtt{set}_0(X)$, $\mathtt{set}_1(X)$, and α; here, we assume that every action has empty precondition and empty Add and Del-Set. Define now the program $\mathcal{P}$ as follows.

$$
\begin{array}{rcl}
\mathbf{O}(\mathtt{set}_0(X)) & \leftarrow & \mathrm{VAR}(X) \\
\mathbf{O}(\mathtt{set}_1(X)) & \leftarrow & \mathrm{VAR}(X) \\
\mathbf{Do}\alpha & \leftarrow & \mathbf{Do}(\mathtt{set}_0(X_1)), \mathbf{Do}(\mathtt{set}_0(X_2)), \mathbf{Do}(\mathtt{set}_0(X_3)), \mathrm{POS}(X_1, X_2, X_3) \\
\mathbf{Do}\alpha & \leftarrow & \mathbf{Do}(\mathtt{set}_1(X_1)), \mathbf{Do}(\mathtt{set}_1(X_2)), \mathbf{Do}(\mathtt{set}_1(X_3)), \mathrm{NEG}(X_1, X_2, X_3)
\end{array}
$$

Here, and throughout the rest of this paper, code call atoms accessing a relation in a database are for simplicity represented by the facts to be accessed.

On this program, we impose the following action constraint

$$AC: \quad \{\mathtt{set}_0(X), \mathtt{set}_1(X)\} \hookleftarrow \mathrm{VAR}(X).$$

Thus, $\mathcal{AC} = \{AC\}$. Let furthermore be $\mathcal{IC} = \emptyset$.

Then, for a given database instance D describing a formula ϕ, it is easily seen that every weak rational status set of $\mathcal{P}$ on D contains $\mathbf{Do}\alpha$, if and only if the corresponding M3SAT instance ϕ is a No-instance. Since $\mathcal{P}$ is easily constructed, the result is proved. ∎

Before closing this subsection, we remark that tractability of both problems can be asserted, if a total prioritization on the weak rational status sets is used, which technically is derived from a total ordering $\alpha_1 < \alpha_2 < \cdots < \alpha_n$ on the set of all ground actions. In this case, a positive agent program $\mathcal{P}$ has a unique weak rational status set S (if one exists). This set S can be constructed by selecting in Step 4 of algorithm COMPUTE-WEAK-RSS always the least action from GA with respect to the ordering $<$. Thus, in the absence of integrity constraints, the unique weak rational status set can be computed in polynomial time in this case.

7.2 Programs with negation

If we allow unrestricted occurrence of negated status atoms in the rule bodies, then the complexity of evaluating agents programs increases. This is not very surprising, since this way, we can express logical disjunction of positive facts. For example, the rule

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

leads to two rational status sets: $S_1 = \{\mathbf{P}\alpha\}$ and $S_2 = \{\mathbf{F}\alpha\}$. Informally, this clause expresses under rational status semantics the disjunction $\mathbf{F}\alpha \vee \mathbf{P}\alpha$. Notice that under the reasonable status semantics, the above rule has only a single reasonable status set, namely S_1. However, if we add its contrapositive

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

then the resulting program has the two reasonable status sets S_1 and S_2. Thus, in the general case, both rational and reasonable status set semantics allow for expressing disjunction, and are for this reason inherently complex. We now analyze the precise complexity of these semantics.

7.2.1 Feasible status sets

A relevant computational problem, for any of the semantics defined above, is the test whether a given status set is among those which are acceptable under the chosen semantics. This problem corresponds to the task of model checking in the area of knowledge representation and reasoning, which has been addressed e.g. in [53, 20, 69].

We note here that for feasible status sets, the recognition problem is tractable under the assumptions that we made; this can be easily seen.

Proposition 7.7 *Let $\mathcal{P}$ be a fixed agent program (where $\mathcal{IC}$ may be nonempty), let $\mathcal{O}_\mathcal{S}$ be a given state, and let S be a status set. Then, deciding whether S is a feasible status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ is possible in polynomial time.*

However, as the following result shows, the search for feasible status sets is intractable in the general case.

Theorem 7.8 *Let $\mathcal{P}$ be a fixed agent program. Then, given an agent state $\mathcal{O}_{\mathcal{S}}$, deciding whether $\mathcal{P}$ has a feasible status set is* NP*-complete, and computing some feasible status set is complete for* FNP*. Hardness holds even if* $\mathcal{IC} = \emptyset$*.*

Proof. By Proposition 7.7, we can guess and check a feasible status set of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$ in polynomial time. Hence, the existence problem is in NP, and the computation problem is in FNP.

To show that the existence problem is NP-hard, we describe a reduction from M3SAT. The reduction is similar to the one in the proof of Theorem 7.6. As there, we suppose that an M3SAT instance ϕ on variables $x_i \in X$ is stored in relations POS (positive clauses) and NEG (negative clauses), and we assume that all variables x_i are stored in VAR. Moreover, we assume that $\mathcal{D}$ has a relation AUX(Var, Val), which contains in the initial database D all tuples $(x_i, 0)$, for all variables x_i.

Now construct the following agent program $\mathcal{P}$:

$$
\begin{array}{rcl}
\mathbf{P}\beta & \leftarrow & \\
\mathbf{F}\beta & \leftarrow & \mathbf{F}\alpha(X_1), \mathbf{F}\alpha(X_2), \mathbf{F}\alpha(X_3), \mathrm{POS}(X_1, X_2, X_3) \\
\mathbf{F}\beta & \leftarrow & \mathbf{P}\alpha(X_1), \mathbf{P}\alpha(X_2), \mathbf{P}\alpha(X_3), \mathrm{NEG}(X_1, X_2, X_3) \\
\mathbf{P}\alpha(X_1) & \leftarrow & \neg\mathbf{F}\alpha(X_1), \mathrm{VAR}(X_1)
\end{array}
$$

The action base $\mathcal{AB}$ contains two actions α and β, which have both empty preconditions and empty add and delete sets. Thus, these actions do not have any effect on the state of the database. The sets $\mathcal{AC}$ and $\mathcal{IC}$ of action and integrity constraints, respectively, are both assumed to be empty.

Then, it is easy to see that $\mathcal{P}$ possesses a feasible status set over $\mathcal{O}_{\mathcal{S}}$, if and only if the formula ϕ is satisfiable; the satisfying truth assignments of ϕ correspond naturally (but not 1-1) to the feasible status sets of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$. (Observe that every feasible status set must either contain $\mathbf{P}\alpha(x_i)$ or contain $\mathbf{F}\alpha(x_i)$, for every x_i, but not both; intuitively, $\mathbf{P}\alpha(x_i)$ represents that x_i is true, while $\mathbf{F}\alpha(x_i)$ represents that x_i is false.) Since for a given formula ϕ the database instance D of $\mathcal{D}$ is clearly constructible in polynomial time, it follows that the decision problem is NP-hard. Moreover, by the correspondence between feasible sets status of $\mathcal{P}$ and the satisfying assignments of ϕ, it follows immediately that the feasible status set computation problem is hard for FNP.

Observe that we can replace in the construction the positive atoms $\mathbf{F}\alpha(X_i)$ in the rule with $\mathbf{F}\beta$ in the head by $\neg\mathbf{P}\alpha(X_i)$, and we would get the same feasible status sets; moreover, the last rule could then also be removed, and still a feasible status exists iff ϕ is satisfiable. ∎

This negative result raises the issue of how we can achieve tractability of programs. There are different possibilities.

One possibility is that we identify syntactic constraints under which programs are guaranteed to be tractable. However, as the form of the program in the proof of the previous theorem indicates, rather strict conditions on negation must be imposed, in order to exclude possible inconsistencies. Still, a number of different feasible and rational status sets may exist, due to the inherent logical disjunction. In particular, the reduction in the proof of Theorem 7.8 works for rational status sets as well.

Another possibility is that we use an alternative semantics which is more amenable to cutting disjunctive cases. In particular, under *reasonable* status set semantics, the program in the proof of Theorem 7.8 either has no reasonable status set, or a unique such status set, which can be efficiently determined in polynomial time; notice that $\mathbf{F}\alpha(a_i)$ is not contained in any reasonable status set, since there is no possibility for deriving $\mathbf{F}\alpha(a_i)$ by means of the head of a program rule or by deontic closure. However, if we add the rule

$$\mathbf{F}\alpha(X_1) \leftarrow \neg\mathbf{P}\alpha(X_1)$$

to the program, then the reasonable status sets of the resulting program $\mathcal{P}'$ coincide with the rational status sets of $\mathcal{P}$. Hence, also the computation of a reasonable status set is intractable in general. We will deal with reasonable status set in detail in Subsection 7.2.3.

From Theorem 7.8, the following result on action reasoning on the feasible status sets is easily derived.

Theorem 7.9 *Let $\mathcal{P}$ be a fixed agent program. Then, given an agent state $\mathcal{O}_\mathcal{S}$ and a ground action α, deciding whether $\alpha \in \mathbf{Do}(S)$ for (i) every (resp., (ii) some) feasible status set S of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, is* co-NP-*complete (resp.,* NP-*complete).*

7.2.2 Rational status sets

For the existence problem, we obtain from Proposition 5.5 and Theorem 7.8 immediately the following result.

Theorem 7.10 *Let $\mathcal{P}$ be a fixed agent program, and suppose $\mathcal{IC} = \emptyset$. Then, given an agent state $\mathcal{O}_\mathcal{S}$, deciding whether $\mathcal{P}$ has a rational status set on $\mathcal{O}_\mathcal{S}$ is NP-complete.*

The condition that a feasible status set is grounded requires a minimality check. It turns out that this minimality check is, in general, an expensive operation. In fact, the following holds.

Theorem 7.11 *Let $\mathcal{P}$ be a fixed agent program, and suppose $\mathcal{IC} = \emptyset$. Then, given an agent state $\mathcal{O}_\mathcal{S}$ and a feasible status set S for $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, deciding whether S is grounded is* co-NP-*complete.*

Proof. In order to refute that S is grounded, we can guess a status set S' such that $S' \subset S$ and verify in polynomial time that S' satisfies the conditions $(S1)$–$(S3)$ of a feasible status set.

To show that the problem is co-NP-hard, we use a variant of the construction in the proof of Theorem 7.8. For the CNF formula ϕ there, we set up the following program $\mathcal{P}$:

$$\begin{array}{rcl}
\mathbf{P}\beta & \leftarrow & \\
\mathbf{F}\beta & \leftarrow & \neg\mathbf{P}\gamma, \neg\mathbf{P}\alpha(X_1), \neg\mathbf{P}\alpha(X_2), \neg\mathbf{P}\alpha(X_3), \mathrm{POS}(X_1, X_2, X_3) \\
\mathbf{F}\beta & \leftarrow & \neg\mathbf{P}\gamma, \mathbf{P}\alpha(X_1), \mathbf{P}\alpha(X_2), \mathbf{P}\alpha(X_3), \mathrm{NEG}(X_1, X_2, X_3) \\
\mathbf{P}\alpha(X_1) & \leftarrow & \mathbf{P}\gamma, \mathrm{VAR}(X_1)
\end{array}$$

Here, γ is a new action of the same type as α and β, i.e., it has empty precondition and empty add and delete sets.

It is easily seen that $S = \{\mathbf{P}\beta, \mathbf{P}\gamma\} \cup \{\mathbf{P}\alpha(a_i) \mid i = 1, \ldots, n\}$ is a feasible status set of $\mathcal{P}$. Moreover, S is grounded, if and only if formula ϕ is not satisfiable. This proves co-NP-hardness.

The reduction even allows to derive another result. In fact, observe that any rational status set of $\mathcal{P}$ is contained in S: if $\mathbf{P}\gamma \in S'$ for a status set S' which satisfies $(S1)$–$(S3)$, then clearly $S' \supseteq S$ holds; otherwise, if $\mathbf{P}\gamma \notin S'$, then $S' \subset S$ must hold.

Assume w.l.o.g. that either ϕ is unsatisfiable, or it has at least two satisfying assignments. Then, S is the unique rational set of $\mathcal{P}$, iff ϕ is unsatisfiable. As a consequence, deciding whether a non-positive agent program has a unique rational status set is co-NP-hard as well. ∎

The complexity of the recognition problem is an immediate consequence of the previous theorem and Proposition 7.7.

Corollary 7.12 *Let $\mathcal{P}$ be a fixed agent program, and suppose $\mathcal{IC} = \emptyset$. Then, given an agent state $\mathcal{O}_{\mathcal{S}}$ and a status set S, deciding whether S is a rational status set for $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$ is* co-NP*-complete.*

In the absence of integrity constraints, the rational status sets coincide with the minimal feasible status sets. Using an NP oracle, we can compute a rational status set as follows.

Algorithm COMPUTE-RATIONAL-SS

Input: agent state $\mathcal{O}_{\mathcal{S}}$ (fixed agent program $\mathcal{P}$, $\mathcal{IC} = \emptyset$);

Output: a rational status set of $\mathcal{P}$, if one exists; “No”, otherwise.

Method

Step 1. Set $S := \emptyset$ and At := set of all ground action status atoms.

Step 2. Check if S is a feasible status set; if true, then output S and halt.

Step 3. If $At = \emptyset$, then output and halt.

Step 4. Choose some atom $A \in At$ and query the oracle whether a feasible status set S' exists such that $S \subseteq S' \subseteq S \cup (At \setminus \{A\})$; If the answer is “no”, then $S := S \cup \{A\}$;

Step 5. set $At := At \setminus \{A\}$ and continue at Step 2.

This algorithm correctly outputs a rational status set (so one exists) in polynomial time modulo calls to the oracle. Hence, the problem is in $\mathrm{FP}^{\mathrm{NP}}$. This upper bound can be improved to $\mathrm{FNP}//\mathrm{log}$, since we can nondeterministically compute a rational status set as follows.

1. Compute the smallest size s of a feasible status set S;
2. nondeterministically generate, i.e., guess and check a feasible status set S such that $|S| = s$, and output it.

Step 1 amounts to an NP optimization problem, since s can be computed in a binary search on the range of possible values, and s has in binary $O(\log |I|)$ many bits, where I is the instance size.

The time for Step 2 is also polynomial time (cf. Proposition 7.7).

Hence the overall algorithm proves that computing a rational status set is in $\mathrm{FNP}//\mathrm{log}$, if $\mathcal{IC} = \emptyset$. We obtain the following result.

Theorem 7.13 *Let $\mathcal{P}$ be a fixed agent program, and suppose $\mathcal{IC} = \emptyset$. Given an agent state $\mathcal{O}_\mathcal{S}$, computing any rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ is in* FNP//$\log$ *and hard for both* FNP *and* $\mathrm{FP}_{\|}^{\mathrm{NP}}$.

Proof. By the previous discussion, it follows that the problem is in FNP//$\log$. Hardness for FNP follows from the proof of Theorem 7.8.

Thus, it remains to show hardness for $\mathrm{FP}_{\|}^{\mathrm{NP}}$. We establish this by a reduction of computing a minimal model of a propositional CNF formula ϕ, i.e., find a model M (satisfying truth assignment to the variables), such that no model M' exists with $M' \subset M$, where a model is identified with the set of variables which are true in it. $\mathrm{FP}_{\|}^{\mathrm{NP}}$-hardness of this problem, even if all clauses in ϕ have at most three literals, follows easily from the results in [24] (Lemma 4.7).

The reduction is an extension of the one in the proof of Theorem 7.11 (note the observations on rational status sets of the program $\mathcal{P}$ there, and that a rational status set always exists).

We use six further 3-ary relations $C_1, \ldots, C_6$ for storing the clauses which are neither positive nor negative, and add respective rules deriving $\mathbf{F}\beta$. More precisely, if we set C_0 = NEG and C_7 = POS, then the relation C_i stores the clauses $C = L_1 \vee L_2 \vee L_3$ such that the string $p(L_1)p(L_2)p(L_3)$ of the polarities of the literals yields i in binary, where $p(L) = 1$ if L is positive, and $p(L) = 0$, if L is negative; thus, e.g. the clause $x_1 \vee x_5 \vee \neg x_3$ is stored as tuple (x_1, x_5, x_3) in the relation C_6, since $p(x_1)p(x_5)p(\neg x_3) = 110$.

Then, the rational status set of the resulting program $\mathcal{P}'$ on the database for ϕ correspond 1-1 to the minimal models of ϕ, if ϕ is satisfiable, and the set S from there is the unique rational status set iff ϕ is unsatisfiable. Moreover, from any rational status set, the corresponding minimal model M is easily computed.

Hence, the computation of a minimal model reduces to the computation of a rational status set. This implies $\mathrm{FP}_{\|}^{\mathrm{NP}}$-hardness, and the theorem is proved. ∎

An improvement of these bounds, in particular completeness for FNP//$\log$, seems to be difficult to achieve. In fact, it can be shown that computing a rational status set is polynomial time equivalent to computing a minimal model of a CNF formula, which is not known to be complete for FNP//$\log$, cf. [24].

Action reasoning becomes harder in the brave variant if we use rational status sets instead of feasible status sets. The reason is that we have to check groundedness of a status set, which is a source of complexity and adds another level in the polynomial hierarchy. However, for the cautious variant, there is no complexity increase.

Theorem 7.14 *Let $\mathcal{P}$ be a fixed agent program $\mathcal{P}$, and suppose $\mathcal{IC} = \emptyset$. Let $\mathcal{O}_\mathcal{S}$ be a given agent state and let α be a given ground action atom. Then, deciding whether $\alpha \in \mathbf{Do}(S)$ holds for (i) every (resp., (ii) some) rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ is* co-NP*-complete (resp., Σ_2^P-complete).*

Proof. For (i), observe that to disprove $\alpha \in \mathbf{Do}(S)$ for every rational status set S, we can guess a feasible status set S such that $\alpha \notin S$ and verify the guess in polynomial time by Proposition 7.7. Hence, the problem is in co-NP. Hardness follows from the reduction in the proof of Theorem 7.8; there, $\mathbf{Do}(\beta)$ belongs to every rational status set of the constructed program $\mathcal{P}$, if and only if $\mathcal{P}$ has no feasible status set.

(ii). The membership part is easy: A guess for a rational status set S such that $\alpha \in Do(S)$ can be verified by Proposition 7.7 and Theorem 7.11 in polynomial time with an NP oracle.

The hardness part is shown by a reduction from a syntactical fragment of quantified Boolean formulas which is Σ_2^P-hard, and combines the reductions in the proofs of Theorems 7.8 and 7.11 in a suitable way.

Telling whether a quantified Boolean formula (QBF) $\forall X \exists Y.\phi$, where $\phi = \bigvee_{j=1}^{m} C_j$ is a CNF formula of clauses $C_j = L_{j,1} \vee L_{j,2} \vee L_{j,3}$ whose literals $L_{j,k}$ are over propositional variables (atoms) $X \cup Y$, is not true is a well-known Σ_2^P-complete problem [37]. This remains true even if each clause C_j is either positive or negative (i.e., ϕ is a M3SAT instance).

We extend the database $\mathcal{D}$ from the proofs of Theorems 7.8 and 7.11, by adding two further relations XVAR and YVAR for storing the variables of X and Y, respectively. Construct a program $\mathcal{P}$, using the actions α, β, and γ from above as follows.

$$
\begin{array}{rcl}
\mathbf{P}\beta & \leftarrow & \\
\mathbf{F}\beta & \leftarrow & \neg\mathbf{P}\gamma, \neg\mathbf{P}\alpha(X_1), \neg\mathbf{P}\alpha(X_2), \neg\mathbf{P}\alpha(X_3), \text{POS}(X_1, X_2, X_3) \\
\mathbf{F}\beta & \leftarrow & \neg\mathbf{P}\gamma, \mathbf{P}\alpha(X_1), \mathbf{P}\alpha(X_2), \mathbf{P}\alpha(X_3), \text{NEG}(X_1, X_2, X_3) \\
\mathbf{P}\alpha(X_1) & \leftarrow & \neg\mathbf{F}\alpha(X_1), \text{XVAR}(X_1) \\
\mathbf{P}\alpha(X_1) & \leftarrow & \mathbf{P}\gamma, \text{YVAR}(X_1) \\
\mathbf{Do}\gamma & \leftarrow & \mathbf{P}\gamma
\end{array}
$$

Clearly, every feasible status set S must contain either $\mathbf{P}\alpha(x)$ or $\mathbf{F}\alpha(x)$ (but not both), for every $x \in X$. Moreover, if $\mathbf{P}\gamma \in S$, then $\mathbf{Do}\gamma \in S$ and for all $y \in Y$, we have $\mathbf{P}\alpha(y) \in S$.

Let χ be a choice among the atoms $\mathbf{P}\alpha(x)$ and $\mathbf{F}\alpha(x)$, for all $x \in X$. Then, define

$$
S_\chi = \chi \cup \{\mathbf{P}\beta, \mathbf{P}\gamma, \mathbf{Do}\gamma\} \cup \{\mathbf{P}\alpha(y) \mid y \in Y\}.
$$

It is easy to see that S_χ is a feasible status set, for every choice χ. We claim that every rational status set S of $\mathcal{P}$ must be contained in some of the S_χ.

To see this, notice that no atoms with status $\mathbf{W}$ or $\mathbf{O}$ can be in S, since there is no possibility to derive such an atom. For the same reason, no atoms $\mathbf{Do}\alpha(v)$, $\mathbf{Do}\beta$, $\mathbf{F}\gamma$ and $\mathbf{F}\alpha(y)$ can be in S, for every $v \in X \cup Y$ and $y \in Y$. Hence, by the observation on $\mathbf{P}(\alpha(x))$ and $\mathbf{F}(\alpha(x))$ from above, S must be a subset of some S_χ.

It is easy to set that S_χ is not grounded, if and only if $\mathbf{P}\gamma$ can be removed from it, such that $S_\chi \setminus \{\mathbf{P}\gamma, \mathbf{Do}\gamma\}$ contains a feasible status set. This happens to be the case if the formula $\exists Y.\phi[\chi]$ is true, where $\phi[\chi]$ is ϕ with $x \in X$ being replaced by *true*, if $\mathbf{P}\alpha(x) \in S_\chi$, and by *false*, if $\mathbf{F}\alpha(x) \in S_\chi$, for all $x \in X$.

Thus, it follows that some rational status set of $\mathcal{P}$ contains $\mathbf{Do}\gamma$, if and only if S_χ is a rational status set of $\mathcal{P}$ for some χ, if and only if for some χ the formula $\phi[\chi]$ is unsatisfiable, if and only if $\forall X \exists Y.\phi$ is not true. ∎

Of course, for positive agent programs, action reasoning is easier. In fact, in this case it is polynomial for both (i) and (ii) since a rational status set, if it exists, is unique.

7.2.3 Reasonable status sets

Our first result on reasonable status sets is positive: the recognition problem, even in the general setting where we have negation and integrity constraints, is tractable.

Theorem 7.15 *Let $\mathcal{P}$ be a fixed agent program (where $\mathcal{IC}$ is not necessarily empty). Then, given an agent state $\mathcal{O}_\mathcal{S}$ and a status set S, deciding whether S is a reasonable status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ is possible in polynomial time.*

Proof. Indeed, by our assumptions, the ground instance of $\mathcal{P}$ over the agent state is constructible in polynomial time, and, moreover, the reduct $red^S(\mathcal{P}, \mathcal{O}_\mathcal{S})$ is computable in polynomial time. By Theorem 7.1, the unique rational status set S' of $red^S(\mathcal{P}, \mathcal{O}_\mathcal{S})$ is computable in polynomial time, and it remains by Theorem 5.3 and the definition of a reasonable status set to check whether $S = S'$ (so S' exists). Overall, this yields a polynomial-time algorithm. ∎

Computing a reasonable status set, however, is clearly intractable in the general case, even in the absence of integrity constraints. We note for completeness sake the complexities of deciding the existence of a reasonable status set and computing one, which are immediate from Proposition 7.7 and the discussion after Theorem 7.8.

Theorem 7.16 *Let $\mathcal{P}$ be a fixed agent program (where $\mathcal{IC}$ is not necessarily empty). Then, given an agent state $\mathcal{O}_\mathcal{S}$, deciding whether $\mathcal{P}$ has a reasonable status set on $\mathcal{O}_\mathcal{S}$ is* NP*-complete, and computing some reasonable status set S of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ is complete for* FNP. *Hardness holds even if $\mathcal{IC} = \emptyset$.*

It is clear in the light of this result that for nonpositive programs with no integrity constraints, action reasoning on the reasonable status sets is intractable. However, compared to the rational status sets, the complexity of the brave variant is lower; this is explained by the absence of an expensive groundedness test for reasonable status sets, which allows for an efficient recognition.

Theorem 7.17 *Let $\mathcal{P}$ be a fixed agent program $\mathcal{P}$ (where $\mathcal{IC}$ is not necessarily empty). Let $\mathcal{O}_\mathcal{S}$ be a given agent state and let α be a given ground action atom. Then, deciding whether $\alpha \in \mathbf{Do}(S)$ holds for (i) every (resp., (ii) some) reasonable status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ is* co-NP*-complete (resp.,* NP*-complete). Hardness holds even if $\mathcal{IC} = \emptyset$.*

Proof. For (i) (resp., (ii)), we can guess a reasonable status set S of $\mathcal{P}$ such that α in $\mathbf{Do}(S)$ (resp., $\alpha \notin \neg\mathbf{Do}(\alpha)$) and verify the guess in polynomial time (Proposition 7.15).

Hardness for (i) and (ii) can be easily shown by modifying the reduction in the proof of Theorem 7.8 as discussed (add $\mathbf{F}(\alpha(X_1)) \leftarrow \mathbf{P}(\alpha(X_1))$) questioning about β, where we add in (ii) the rule $\mathbf{Do}(\beta) \leftarrow$. ∎

7.2.4 Weak status sets

In Subsection 7.1.1, we have already considered the computation of weak rational (resp., reasonable) status sets for positive programs. In the presence of negation, the concepts of weak rational status sets and weak reasonable status set do no longer coincide. Also, the complexities of the different concepts of status sets are different.

Compared to rational (resp., reasonable) status sets, we have here to deal with relativized action closure ACl_A, where A is a set of ground actions; recall that A-feasibility, A-rationality etc is defined like A-feasibility, with the only only difference that action closure is fixed to actions in A, rather than all ground actions. The relativization to A does not affect the complexity.

Proposition 7.18 *Let $\mathcal{P}$ be any program, and let $\mathcal{O}_\mathcal{S}$ be an agent state. Then, given S and A, testing A-feasibility of S (resp., A-rationality, A-reasonability), has the same complexity as testing feasibility (resp., rationality, reasonability) of S.*

Since under our assumptions, a weak rational (resp., weak reasonable) status set exists if and only if an A-rational (resp., A-reasonable) status set exists for some A, we easily obtain from the proofs of Theorems 7.8 and 7.16 the following result.

Theorem 7.19 *Let $\mathcal{P}$ be a fixed agent program, and suppose $\mathcal{IC} = \emptyset$. Then, given an agent state $\mathcal{O}_\mathcal{S}$, deciding whether $\mathcal{P}$ has a weak rational (resp., reasonable) status set on $\mathcal{O}_\mathcal{S}$ is* NP*-complete.*

The computation of any weak rational status set can be accomplished using the algorithm COMP-WEAK-RATIONAL-SS described below.

Algorithm COMP-WEAK-RATIONAL-SS

Input: agent state $\mathcal{O}_\mathcal{S}$, (fixed agent program $\mathcal{P}, \mathcal{IC} = \emptyset$)

Output: a weak rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, if one exists.

Method

1. Compute the maximum size s of a set A such that $\mathcal{P}$ has an A-feasible status set on $\mathcal{O}_\mathcal{S}$;
2. Compute a set A such that $|A| = s$ and some A-feasible status set exists;
3. Compute the smallest size s' of any A-feasible status set S;
4. Compute an A-feasible status set S such that $|S| = s'$, and output S.

The steps 1.-4. can be done in polynomial time with the help of an NP oracle. Therefore, computing a weak rational status set is in $\mathrm{FP}^{\mathrm{NP}}$ in the absence of integrity constraints. Notice by Proposition 5.10, Steps 1 and 2 can be combined by computing a status set S which is $A(S)$-feasible and such that $|A(S)|$ is maximal.

For weak reasonable status sets, we can apply an adapted version of COMP-WEAK-RSS, in which "A-feasible" is replaced by "A-reasonable"; Notice that existence problem for A-feasible and A-reasonable status sets has the same complexity.

Thus, for both kinds of status sets, the computation problem is polynomial if an NP oracle may be consulted. We can improve on this upper bound and give an exact characterization of the problem in terms of the complexity class FNP//$\log$, which comprises computation problems with an adjunct NP optimization problem (see Section 6.3 and [24]).

In our case, this NP optimization problem consists in the computation of the numbers s and s', respectively. It is possible to combine these two steps into a single NP optimization problem, such that we can generate, given its solution, nondeterministically in polynomial time a weak rational (resp., reasonable) status set.

Theorem 7.20 *Let $\mathcal{P}$ be a fixed agent program and suppose that $\mathcal{IC} = \emptyset$. Then, computing any weak rational (resp., weak reasonable) status set of $\mathcal{P}$ on a given agent state $\mathcal{O}_\mathcal{S}$ is complete for* FNP//$\log$.

Proof. Let GA be the set of all ground action atoms. Associate with every status set S the tuple $t_S = \langle |A(S)|, |GA| - |S| \rangle$, if S is $A(S)$-rational, and $t_s = \langle -1, 0 \rangle$ otherwise, and impose on the tuples t_S the usual lexicographic order. Then, the following holds: Any status set S such that t_S is maximal is a weak rational status set, if and only if $t_S \neq \langle -1, 0 \rangle$.

Given a maximal tuple t_S, it is clearly possible to generate a weak rational status set S nondeterministically in polynomial time, so one exists. Moreover, the tuples t_S can be easily encoded by polynomial size numbers $z(t_S)$, such that $z(t_S) > z(t_{S'})$ iff $t_S > t_{S'}$; e.g., define $z(\langle i, j \rangle) = (|GA| + 1)i + j$. Computing the maximum $z(t_S)$ is an NP optimization problem, and from any $z(t_S)$, the tuple t_S is easily computed. Hence, it follows that computing a weak rational status set is in $\mathrm{FNP}//\log$.

It remains to show hardness for this class. For this purpose, we reduce the computation of an X-maximal model [23, 24] to this problem. This problem is, given a propositional CNF formula ϕ and a subset X of the atoms, compute the X-part of a model M of ϕ such that $M \cap X$ is maximal, i.e., no model M' of ϕ exists such that $M' \cap X \supset M \cap X$, where M is identified with the set of atoms true in it. Hardness of this problem for $\mathrm{FNP}//\log$ is shown in [23, 24].[2]

The reduction is as follows. Without loss of generality, we assume that ϕ is an M3SAT instance. Indeed, we may split larger clauses by introducing new variables, and exchange positive (resp., negative) literals in clauses by using for each variable x a new variable $\hat{x}$ which is made equivalent to $\neg x$. (All new variables do not belong to the set X.)

The reduction is similar to the one in the proof of Theorem 7.6. We use the action base and database from there, and introduce a further relation XVAR for storing the variables in X. Consider the following program $\mathcal{P}$:

$$
\begin{array}{rcl}
\mathbf{O}(\mathtt{set_1}(X)) & \leftarrow & \mathrm{XVAR}(X) \\
\mathbf{Do}(\mathtt{set_0}(X)) & \leftarrow & \neg\mathbf{Do}(\mathtt{set_1}(X)), \mathrm{VAR}(X) \\
\mathbf{P}\alpha & \leftarrow & \\
\mathbf{F}\alpha & \leftarrow & \mathbf{Do}(\mathtt{set_0}(X_1)), \mathbf{Do}(\mathtt{set_0}(X_2)), \mathbf{Do}(\mathtt{set_0}(X_3)), \mathrm{POS}(X_1, X_2, X_3) \\
\mathbf{F}\alpha & \leftarrow & \mathbf{Do}(\mathtt{set_1}(X_1)), \mathbf{Do}(\mathtt{set_1}(X_2)), \mathbf{Do}(\mathtt{set_1}(X_3)), \mathrm{NEG}(X_1, X_2, X_3)
\end{array}
$$

and impose on it the action constraint AC:

$$AC: \quad \{\mathtt{set_0}(X), \mathtt{set_1}(X)\} \hookleftarrow \mathrm{VAR}(X).$$

The first rule states that every variable in X should be set to true, and the second rule together with AC effects that x_i is either set to true or to false, but not both.

It is easily seen that the weak rational status sets S of $\mathcal{P}$ on the input database D for an M3SAT instance ϕ correspond 1-1 to the X-maximal models of ϕ, and from every such S, the X-part of the corresponding X-maximal model is easily obtained.

Since D is efficiently constructed from ϕ in polynomial time, it follows that computing a weak rational status set is hard for FNP//$\log$.

The proof of hardness for computing a weak reasonable status set is similar (use an additional clause $\mathbf{Do}(\mathtt{set_1}(X)) \leftarrow \neg\mathbf{Do}(\mathtt{set_0}(X)), \mathrm{VAR}(X)$). This proves the result. ∎

[2]In fact, the authors use in [24] a slightly stronger form of reduction among maximization problems than in [23], which requires that the transformed instance must always have solutions; our proofs of $\mathrm{FNP}//\log$ hardness can be easily adapted to this stronger reduction.

Like in the case of positive programs, recognition of a weak rational status set S is no harder than computation, even if programs are nonpositive. The recognition problem is solved by the following algorithm.

Algorithm REC-WEAK-RATIONAL

Input: status set S on agent state $\mathcal{O}_\mathcal{S}$ (fixed agent program $\mathcal{P}, \mathcal{IC} = \emptyset$)

Output: "Yes", if S is a weak rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, "No" otherwise.

Method

1. Check whether S is $A(S)$-feasible;
2. Check whether there is no $A(S)$-feasible status set S' such that $S' \subset S$;
3. Check whether there is no S' such that S' is $A(S')$-feasible and $A(S) \subset A(S')$.

The correctness of this algorithm follows from Proposition 5.10. However, it is not clear how to implement it in polynomial time. The next theorem establishes that such an implementation is unlikely to exist, nor that any polynomial time algorithm for this problem is known.

Theorem 7.21 *Let $\mathcal{P}$ be a fixed agent program and suppose that $\mathcal{IC} = \emptyset$. Then, given an agent state $\mathcal{O}_\mathcal{S}$ and a status set S, deciding whether S is a weak rational status set of $\mathcal{P}$ on $\mathcal{P}$ is* co-NP-*complete.*

Proof. Algorithm REC-WEAK-RATIONAL can be easily rewritten as a nondeterministic polynomial time algorithm for refuting that S is a weak rational status set. Hardness is immediate from the proof of Theorem 7.11. ∎

A weak reasonable status set can be recognized in a similar way; see algorithm REC-WEAK-REASONABLE below. The correctness of this algorithm follows from Proposition 5.10. We obtain the following result.

Algorithm REC-WEAK-REASONABLE

Input: agent state $\mathcal{O}_\mathcal{S}$, status set S (fixed agent program $\mathcal{P}$)

Output: "Yes", if S is a weak reasonable status set of $\mathcal{P}$, "No" otherwise.

Method

1. check whether S is $A(S)$-reasonable, and output "No" if not;
2. Check whether there is no S' such that S' is $A(S')$-reasonable and $A(S) \subset A(S')$.

Theorem 7.22 *Let $\mathcal{P}$ be a fixed agent program (where $\mathcal{IC}$ is arbitrary). Then, given an agent state $\mathcal{O}_\mathcal{S}$ and a status set S, deciding whether S is a weak reasonable status set is* co-NP-*complete. Hardness holds even if $\mathcal{IC} = \emptyset$.*

Proof. Clearly, algorithm REC-WEAK-REASONABLE can be turned into a NP-algorithm for showing that S is not a weak rational status set.

The hardness part follows by an easy modification to the proof of Theorem 7.8. Add the rules $\mathbf{F}\alpha(X_1) \leftarrow \neg\mathbf{P}\alpha(X_1)$, VAR$(X_1)$, and replace the rule $\mathbf{P}\beta \leftarrow$ by the rule $\mathbf{O}\beta$. Moreover, assume w.l.o.g. that the assignment in which all variables x_i have value false does not satisfy ϕ.

Then, $S = \{\mathbf{F}(x_1) \mid x_i \in X\} \cup \{\mathbf{F}\beta, \mathbf{O}\beta\}$ is $A(S)$-reasonable. It is easily seen that S is a weak reasonable status set, if and only if ϕ is not satisfied by any assignment which sets some variable true. (If such an assignment exists, then the obligation $\mathbf{O}\beta$, which is violated in S, can be obeyed, and thus a reasonable status set exists). ∎

When we switch from rational (resp., reasonable) status sets to weak versions, the complexity of action reasoning is partially affected in the absence of integrity constraints.

It is easy to see that for the brave variant, the complexity for the weak and the ordinary version of rational status sets is the same. In both cases, the straightforward Guess and Check algorithm yields the same upper bound, and the result for brave rational action reasoning has been derived without involving obligations.

For the cautious variant, we find a complexity increase, even if the complexity of the recognition problem has not changed. The reason is that the beneficial monotonicity property of finding just some feasible status set which does not contain the action α in question as a proof that α does not occur in all rational status sets, can (in a suitable adaptation) no longer be exploited.

Theorem 7.23 *Let $\mathcal{P}$ be a fixed agent program $\mathcal{P}$, and suppose $\mathcal{IC} = \emptyset$. Let $\mathcal{O}_\mathcal{S}$ be a given agent state and let α be a given ground action atom. Then, deciding whether $\alpha \in \mathbf{Do}(S)$ holds for (i) every (resp., (ii) some) weak rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ is Π_2^P-complete (resp., Σ_2^P-complete).*

Proof. The proof for the brave variant is in the discussion above.

For the cautious variant, observe that a weak rational status set S such that $\alpha \notin \mathbf{Do}(S)$ can be guessed and checked with an NP oracle in polynomial time.

For the hardness part, we adapt the construction in the proof of Theorem 7.14 for a reduction from QBF formulas $\exists X \forall Y \phi$, where ϕ is in M3DNF form.

We use the action base $\mathcal{AB}$ from there and extend it with another action β of the same type as α. Moreover, we use the relations POS and NEG for storing the disjuncts of ϕ as described in the proof of Theorem 8.2, and replace VAR by the relations XVAR and YVAR for storing the variables in X and Y, respectively.

Then, we set up the following program:

$$
\begin{array}{rcl}
\mathbf{O}(\mathtt{set}_0(X)) & \leftarrow & \text{XVAR}(X) \\
\mathbf{O}(\mathtt{set}_1(X)) & \leftarrow & \text{XVAR}(X) \\
\mathbf{Do}(\mathtt{set}_0(X)) & \leftarrow & \neg\mathbf{Do}(\mathtt{set}_1(X)), \text{XVAR}(X) \\
\mathbf{F}\beta & \leftarrow & \mathbf{Do}(\mathtt{set}_0(X_1)), \mathbf{Do}(\mathtt{set}_0(X_2)), \mathbf{Do}(\mathtt{set}_0(X_3)), \text{POS}(X_1, X_2, X_3) \\
\mathbf{F}\beta & \leftarrow & \mathbf{Do}(\mathtt{set}_1(X_1)), \mathbf{Do}(\mathtt{set}_1(X_2)), \mathbf{Do}(\mathtt{set}_1(X_3)), \text{NEG}(X_1, X_2, X_3) \\
\mathbf{O}(\alpha) & \leftarrow & \\
\mathbf{P}(\beta) & \leftarrow & \mathbf{Do}(\alpha)
\end{array}
$$

We modify the action constraint AC to

$$AC' : \quad \{\mathtt{set}_0(X), \mathtt{set}_1(X)\} \hookleftarrow \text{XVAR}(X).$$

In the above program, the agent is informally obliged by the first two clauses to set every variable $x \in X$ to both true and false, which is prohibited by AC'. The next clause forces him/her to assign each variable in $X \cup Y$ a truth value. By the maximality of weak rational status sets, then agent follows one of the two obligations for each variable in X, which creates an exponential number of possibilities.

The next two clauses check whether the formula ϕ is satisfied. If so, then $\mathbf{F}\beta$ is derived. By the next clause, the agent should take α, but if $\mathbf{F}\beta$ is derived, s/he cannot execute α; hence, s/he must violate this obligation in that case. Thus, if for a choice χ from $\mathbf{O}(\mathtt{set}_0(x))$, $\mathbf{O}(\mathtt{set}_1(x))$, for all $x \in X$, the formula $\forall Y\,\phi[X = \chi]$ is true, then there exists a weak rational status set S such that $\mathbf{Do}\alpha \notin S$; conversely, if there exists such a status set, then a truth assignment χ to X exists such that $\forall Y\,\phi[X = \chi]$ is true.

Consequently, deciding whether $\alpha \in \mathbf{Do}(S)$ for every weak rational status set of $\mathcal{P}$ on the database D for $\exists X \forall Y\,\phi$ is Π_2^P-hard.

We remark that Σ_2^P-hardness of the brave variant can be obtained by adding a rule

$$\mathbf{Do}\gamma \leftarrow \neg\mathbf{Do}\alpha$$

and querying about γ. ∎

For action reasoning with weak reasonable status sets, we obtain similar complexity results.

Theorem 7.24 *Let $\mathcal{P}$ be a fixed agent program. Let $\mathcal{O}_\mathcal{S}$ be a given agent state and let α be a given ground action atom. Then, deciding whether $\alpha \in \mathbf{Do}(S)$ holds for (i) every (resp., (ii) some) weak reasonable status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ is Π_2^P-complete (resp., Σ_2^P-complete). Hardness holds even if $\mathcal{IC} = \emptyset$.*

Proof. A weak reasonable status set S such that $\alpha \notin \mathbf{Do}(S)$ (resp., $\alpha \in \mathbf{Do}(S)$) can be guessed and checked in polynomial time with an NP oracle by Theorem 7.22.

Hardness follows for both problems by a slight extension of the construction in the proof of Theorem 7.23. Add to the program $\mathcal{P}$ there the clause

$$\mathbf{Do}(\mathtt{set}_1(X)) \quad \leftarrow \quad \neg\mathbf{Do}(\mathtt{set}_0(X)), \mathrm{YVAR}(X)$$

Then, the weak reasonable status sets of the resulting program $\mathcal{P}'$ coincide with the weak rational status sets of $\mathcal{P}'$, which coincide with the weak rational status sets of $\mathcal{P}$. This proves the result for (ii). For (i), add the rule $\mathbf{Do}\gamma \leftarrow \mathbf{Do}\alpha$ and query about γ. ∎

7.3 Preferred status sets

Intuitively, adding a preference on rational or reasonable status sets does increase the complexity of the semantics. Even if we have checked that a status set S is rational (resp., reasonable), then we still have to verify that there is no other rational (resp., reasonable) status set S' which is preferred over S. This check appears to be expensive, since we have to explore an exponential candidate space of preferred rational (resp., reasonable) status sets S', and the test whether S' is in fact rational is expensive as well.

However, as it turns out, for rational status sets, preference does not lead to an increase in the complexity of action reasoning if no integrity constraints are present. On the other hand, preference does increase the complexity of action reasoning for reasonable status set. This is explained by the fact that for rational status sets, an increase in complexity is avoided since for deciding preference, it is sufficient to consider feasible status sets for ruling out a candidate S for a preferred rational set, and feasible status sets have lower complexity. For reasonable status sets, a similar property does not apply, and we enface the situation of being obliged to use reasonable status sets for eliminating a candidate.

In the rest of this paper, we focus on F-preferred status sets. Similar results can be derived for the dual P-preferred status sets by dualizing proofs.

Theorem 7.25 *Let $\mathcal{P}$ be a fixed agent program, and suppose $\mathcal{IC} = \emptyset$. Then, given an agent state $\mathcal{O}_{\mathcal{S}}$ and a status set S, deciding whether S is a F-preferred rational status sets of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$ is* co-NP-*complete.*

Proof. By Proposition 7.7, one can decide in polynomial time whether S is a feasible status set. Now we exploit the following property: A feasible status set S of $\mathcal{P}$ is not a F-preferred rational status set of $\mathcal{P}$, if and only if there exists a feasible status set S' of $\mathcal{P}$ such that $\mathbf{F}(S') \subset \mathbf{F}(S)$ holds.

Therefore, we can refute that S is a F-preferred rational status set by guessing a status set S' and checking in polynomial time whether either S is not a feasible set, or whether S' is feasible and satisfies $\mathbf{F}(S') \subset \mathbf{F}(S)$. Hence, the problem is in $\mathrm{co\text{-}NP}$.

Hardness is an immediate consequence of the proof of Theorem 7.11, as the candidate set S defined there satisfies $\mathbf{F}(S) = \emptyset$, and is thus F-preferred, if and only if it is grounded. ∎

The computation of an F-preferred rational status set is possible using a variant of the algorithm COMPUTE-RATIONAL-SS as follows. After Step 1, compute in a binary search the size s of the smallest possible F-part over all feasible status sets of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$; then, in the remaining steps of the algorithm, constrain the oracle query to existence of a feasible status set S', $S \subseteq S' \subseteq S \cup (At \setminus \{A\})$, such that $|\mathbf{F}(S')| = s$. This is a polynomial algorithm using an NP oracle, and hence the problem is in $\mathrm{FP}^{\mathrm{NP}}$.

A refined analysis unveils that the complexity of this problem is, like the one of computing a weak rational status set, captured by the class FNP//$\log$.

Theorem 7.26 *Let $\mathcal{P}$ be a fixed agent program, and suppose $\mathcal{IC} = \emptyset$. Then, given an agent state $\mathcal{O}_{\mathcal{S}}$, computing an arbitrary F-preferred rational status set of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$ is complete for* FNP//$\log$.

Proof. The proof of membership is similar to the one of computing a weak rational status set. Indeed, the F-preferred status sets are those status sets S for which the tuple $t'_S = \langle |\mathbf{F}(S)|, |S| \rangle$ is minimal under lexicographic ordering, where infeasible status sets S have associated the tuple $t'_S = \langle |GA| + 1, |GA| \rangle$, where GA is the set of all ground action status atoms. From a minimal t'_S, a F-preferred rational status set can be nondeterministically constructed in polynomial time. Hence, the problem is in FNP//$\log$.

The proof of hardness is by a reduction from the problem X-maximal model in the proof of Theorem 7.20, which is w.l.o.g. in M3SAT form.

We modify the program in the proof of Theorem 7.8 to the following program $\mathcal{P}'$:

$$\mathbf{P}\beta \leftarrow$$

$$\begin{array}{rcl}
\mathbf{F}\beta & \leftarrow & \neg\mathbf{P}\alpha(X_1), \neg\mathbf{P}\alpha(X_2), \neg\mathbf{P}\alpha(X_3), \mathrm{POS}(X_1, X_2, X_3) \\
\mathbf{F}\beta & \leftarrow & \mathbf{P}\alpha(X_1), \mathbf{P}\alpha(X_2), \mathbf{P}\alpha(X_3), \mathrm{NEG}(X_1, X_2, X_3) \\
\mathbf{F}\alpha(X_1) & \leftarrow & \neg\mathbf{P}\alpha(X_1), \mathrm{XVAR}(X_1)
\end{array}$$

Here, XVAR stores the variables in X. The rational status sets of $\mathcal{P}'$ on the database D for ϕ correspond 1-1 to the models M of ϕ such that for the X-part fixed to $X \cap M$, the part of the remaining variables is minimal, i.e., to the models M such that no M' exists such that $M' \cap X = M \cap X$ and $M' \subset M$.

It is not hard to see that for every F-preferred rational status set S of $\mathcal{P}$ on D, the corresponding model M of ϕ is X-maximal. (Observe also that for every X-maximal model ϕ, there exists some F-preferred rational status set of $\mathcal{P}$ such that the corresponding model M' of ϕ satisfies $M' \cap X = M \cap X$.) Moreover, M is easily constructed from S. It follows that computing an arbitrary F-preferred rational status set is hard for $\mathrm{FNP}//\log$. ∎

For F-preferred reasonable status sets, we obtain similar results. However, we may allow the presence of integrity constraints without a change in the complexity.

Theorem 7.27 *Let $\mathcal{P}$ be a fixed agent program. Then, given an agent state $\mathcal{O}_{\mathcal{S}}$ and a status set S, deciding whether S is a F-preferred reasonable status set of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$ is* co-NP-*complete. Hardness holds even if $\mathcal{IC} = \emptyset$.*

Proof. By Proposition 7.15, we can decide in polynomial time whether S is a reasonable status set, and check that there is no reasonable status set S' such that $\mathbf{F}(S') \subset \mathbf{F}(S)$ with the help of an NP oracle.

Hardness is shown by a proper modification of the program $\mathcal{P}$ in the proof of Theorem 7.11. Indeed, replace the clause $\mathbf{P}\alpha(X_1) \leftarrow \mathbf{P}\gamma$ with the clause $\mathbf{F}\alpha(X_1) \leftarrow \mathbf{F}\gamma$, replace $\neg\mathbf{P}\gamma$ with $\neg\mathbf{F}\gamma$ in the other clauses, and add the following clauses:

$$\begin{array}{rclcrl}
\mathbf{P}\alpha(X_1) & \leftarrow & \neg\mathbf{F}\alpha(X_1) & \qquad & \mathbf{F}\gamma & \leftarrow \neg\mathbf{P}\gamma \\
\mathbf{F}\alpha(X_1) & \leftarrow & \neg\mathbf{P}\alpha(X_1) & \qquad & \mathbf{P}\gamma & \leftarrow \neg\mathbf{F}\gamma
\end{array}$$

Then, the set $S = \{\mathbf{F}\alpha(x_i) \mid x_i \in X\} \cup \{\mathbf{F}\gamma\}$ is a reasonable status set of the new program $\mathcal{P}'$ on the database D. It is the (unique) F-preferred reasonable status set, if and only if the formula ϕ is not satisfiable. Hence, deciding whether S is a F-preferred reasonable status set is co-NP-hard. ∎

An F-preferred reasonable status set can be computed applying an algorithm analogous to the one used for computing a F-preferred rational status set. First, compute the minimum size s of the F-part $\mathbf{F}(S)$ over all reasonable status sets S, and then construct a reasonable status set S such that $|\mathbf{F}(S)| = s$.

This matches the solution scheme for problems in FNP//$\log$; we obtain the following result.

Theorem 7.28 *Let $\mathcal{P}$ be a fixed agent program (where $\mathcal{IC}$ is not necessarily empty). Then, given an agent state $\mathcal{O}_{\mathcal{S}}$, computing any F-preferred reasonable status set of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$ is complete for* FNP//$\log$. *Hardness holds even if $\mathcal{IC} = \emptyset$.*

Proof. The membership part is in the preceding discussion.

The hardness part can be shown by a modification of the reduction in the proof of Theorem 7.26. Add to the program from there the following rules:

$$
\begin{aligned}
\mathbf{P}\alpha(X_1) &\leftarrow \neg\mathbf{F}\alpha(X_1), \mathrm{XVAR}(X_1)\\
\mathbf{P}\alpha(X_1) &\leftarrow \neg\mathbf{P}\gamma(X_1), \mathrm{YVAR}(X_1)\\
\mathbf{P}\gamma(X_1) &\leftarrow \neg\mathbf{P}\alpha(X_1), \mathrm{YVAR}(X_1)
\end{aligned}
$$

here, YVAR is a relation which stores the variables which are not in X, and γ is a new action of the same type as α.

It holds that the reasonable status sets of the described program $\mathcal{P}$ on the database D for ϕ correspond 1-1 to the models of ϕ; moreover, the F-preferred reasonable status sets S correspond 1-1 to the X-maximal models M of ϕ. Since M is easily computed from S, it follows that computing an arbitrary F-preferred reasonable status set is $\mathrm{FNP}//\log$-hard. ∎

7.3.1 Action reasoning

Theorem 7.29 *Let $\mathcal{P}$ be a fixed program, and suppose that $\mathcal{IC} = \emptyset$. Then, given an agent status $\mathcal{O}_\mathcal{S}$ and an action status atom A, deciding whether A belongs to (i) every (resp., (ii)some) F-preferred rational status sets is Π_2^P-complete (resp., Σ_2^P-complete).*

Proof. For the membership part, observe that a guess for a F-preferred rational status set S such that $A \notin S$ (resp., $A \in S$), can be verified by checking that F is feasible, F is grounded, and that no feasible status set S' exists such that $\mathbf{F}(S') \subset \mathbf{F}(S)$. By Proposition 7.7, and Theorem 7.11, it follows that these tests can be done in polynomial time with an NP oracle. Hence, the problem is in Π_2^P (resp., Σ_2^P).

To show hardness, we employ a slight modification of the construction in the proof of case (ii) of Theorem 7.14. Add to the program $\mathcal{P}$ from there the clauses

$$
\begin{aligned}
\mathbf{P}\alpha^*(X_1) &\leftarrow \mathbf{F}\alpha(X_1), \mathrm{XVAR}(X_1)\\
\mathbf{F}\alpha^*(X_1) &\leftarrow \mathbf{P}\alpha(X_1), \mathrm{XVAR}(X_1)
\end{aligned}
$$

where α^* is a new action of the same type as α. The effect of these clauses is to include $\mathbf{P}\alpha^*(x_i)$ in a rational status set, if $\mathbf{F}\alpha(x_i)$ belongs to it, and symmetrically to include $\mathbf{F}\alpha^*(x_i)$, if $\mathbf{P}\alpha(x_i)$ occurs in it. This way, the extended candidate set

$$S_\chi^* = S_\chi \cup \{\mathbf{F}\alpha^*(x) \mid \mathbf{P}\alpha(x) \in S_\chi, x \in X\} \cup \{\mathbf{P}\alpha^*(x) \mid \mathbf{F}\alpha(x) \in S_\chi, x \in X\}$$

is *not* a F-preferred rational status set, if and only if the formula $\exists Y.\phi[\chi]$ is true.

Since any rational status set S' which is F-preferred over S_χ^* must not contain $\mathbf{Do}\gamma$, it follows that $\mathbf{Do}\gamma$ is contained in some F-preferred rational status set, if and only if the formula $\forall X \exists Y.\phi$ is false. This proves Σ_2^P-hardness of (ii).

For (i), we simply add a clause $\mathbf{Do}\delta \leftarrow \neg\mathbf{P}\gamma$ to the above program, where δ is another action of the type of α. Then, every rational status set S' which is F-preferred to S_χ^* contains $\mathbf{Do}\delta$, while S_χ^* does not contain $\mathbf{Do}\delta$. Consequently, $\mathbf{Do}\gamma$ occurs in all F-preferred rational status sets for $\mathcal{P}$, if and only if $\forall X \exists Y.\phi$ is true. This proves Π_2^P-hardness. ∎

As discussed above, for reasonable status sets F-preference leads to a complexity increase, and raises it to the level of rational status sets. However, as with the other problems on reasonable status sets, this increase is independent of whether integrity constraints are present or not. For rational status sets, this is not the case, and the complexity there is higher in the general case, as we will see in Section 8.5.

Theorem 7.30 *Let $\mathcal{P}$ be a fixed program (where $\mathcal{IC}$ may be nonempty). Then, given an agent state $\mathcal{O}_{\mathcal{S}}$ and an action status atom A, deciding whether A belongs to (i) every (resp., (ii)some) F-preferred reasonable status set is Π_2^P-complete (resp., Σ_2^P-complete). Hardness holds even if $\mathcal{IC} = \emptyset$.*

Proof. A F-preferred reasonable status set S such that $A \notin S$ (resp., $A \in S$) can be guessed and checked in polynomial time with the help of an NP oracle (Proposition 7.15, Theorem 7.27).

For the hardness we employ a reduction from a variant of Quantified Booleans Formulas, described in Lemma 8.14: decide whether a formula $\Phi = \forall X \exists Y \neq \emptyset.\, \phi$ is true, knowing that for every X, the assignment $\chi = \emptyset$ satisfies the formula.

We extend the reduction in the proof of Theorem 7.8 in the same way as we have extended the one in the proof of Theorem 7.14 for the proof of Theorem 7.29. We then have the program $\mathcal{P}$:

$$
\begin{array}{rcl}
\mathbf{P}\beta & \leftarrow & \\
\mathbf{F}\beta & \leftarrow & \mathbf{F}\alpha(X_1), \mathbf{F}\alpha(X_2), \mathbf{F}\alpha(X_3), \text{POS}(X_1, X_2, X_3) \\
\mathbf{F}\beta & \leftarrow & \mathbf{P}\alpha(X_1), \mathbf{P}\alpha(X_2), \mathbf{P}\alpha(X_3), \text{NEG}(X_1, X_2, X_3) \\
\mathbf{P}\alpha(X_1) & \leftarrow & \neg\mathbf{F}\alpha(X_1) \\
\mathbf{F}\alpha(X_1) & \leftarrow & \neg\mathbf{P}\alpha(X_1) \\
\mathbf{P}\alpha^*(X_1) & \leftarrow & \mathbf{F}\alpha(X_1), \text{XVAR}(X_1) \\
\mathbf{F}\alpha^*(X_1) & \leftarrow & \mathbf{P}\alpha(X_1), \text{XVAR}(X_1)
\end{array}
$$

The asserted property of Φ implies that for each choice from $\mathbf{F}(\alpha(x_i))$ and $\mathbf{P}(\alpha(x_i))$, for all $x_i \in X$, we have a status set S_χ^+ containing all atoms $\mathbf{F}(\alpha(y_i))$, for all $y_i \in Y$, such that S_χ^+ is a reasonable status set of $\mathcal{P}$.

Now, if we add the rule

$$\mathbf{Do}(\gamma) \leftarrow \neg\mathbf{F}\alpha(X_1), \text{YVAR}(X_1)$$

where γ is a fresh action of the same type as β, then γ is contained in every F-preferred reasonable status set of $\mathcal{P}$, if and only if $\forall X \exists Y \neq \emptyset.\, \phi$ is true. This proves Π_2^P-hardness of (i). For (ii), add another rule

$$\mathbf{Do}(\delta) \leftarrow \neg\mathbf{Do}(\gamma)$$

where δ is another fresh action of the type of β. Then, δ belongs to some F-preferred reasonable status set of $\mathcal{P}$, if and only if $\forall X \exists Y \neq \emptyset.\, \phi$ is false. This proves the theorem. ∎

8 Complexity Impact of Integrity Constraints

So far, we have focused in our complexity analysis mainly on agent programs where in the background no integrity constraints were present. We say mainly, since for positive programs and reasonable status sets, most results that have been derived in Section 6 do allow for integrity constraints, and fortunately establish tractability for a number of important computation problems.

However, in the presence of negation, we have excluded integrity constraints. The reason is that in some cases, the presence or absence of integrity constraints makes a difference to the intrinsic complexity of a problem, while in other cases, there is no difference. A systematic treatment of this issue is suggestive; therefore, we analyze in this section the effects of integrity constraints on the complexity of agent programs. An overview of the effects and a discussion is given in Section 6.2. In the rest of this section, we develop the technical results.

8.1 Feasible status sets

As shown in the previous section, finding a rational or feasible status set of a positive agent program is polynomial, if no integrity constraints are present. While adding integrity constraints preserves polynomial time computability of rational status sets, it leads to intractability for feasible status sets. In fact, already for a software package $\mathcal{S} = (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}})$ which is a simple relational database $\mathcal{D}$ in which tuples may be inserted or deleted from tables, we face intractability if the integrity constraints include functional dependencies (FDs for short) on the tables. Notice that FDs are one of the most basic and important type of dependencies in databases [102].[3]

Theorem 8.1 *Let $\mathcal{P}$ be a fixed agent program, where $\mathcal{IC}$ ma y be nonempty. Then, deciding whether $\mathcal{P}$ has a feasible status set on a given agent state $\mathcal{O}_{\mathcal{S}}$ is NP-complete, and computing an arbitrary feasible status set is FNP-complete. Hardness holds even if $\mathcal{P}$ is positive and $\mathcal{IC}$ holds functional dependencies of a relational database $\mathcal{D}$.*

Proof. The problem is in NP, since a feasible status set S can be guessed and checked in polynomial time, according to our assumptions (cf. Proposition 7.7).

We show the hardness part for the particular restriction by a reduction from the set splitting problem [37]. Given a collection $\mathcal{S} = \{S_1, \ldots, S_m\}$ of sets over a finite set U, decide whether there exists a partitioning (or *coloring*) (C_1, C_2) of U such that every $S_i \in \mathcal{S}$, $i = 1, \ldots, m$, meets each of C_1 and C_2 in at least one element.

We construct from $\mathcal{S}$ an instance of the feasible status set test as follows. The database $\mathcal{D}$ has four relations: COLL(Set, El), SPLIT$(El, Color)$, A1(Set, El, Tag) and A2(Set, El, Tag). Intuitively, the collection $\mathcal{S}$ is stored in COLL by tuples (i, e) for every $e \in S_i$ and $S_i \in \mathcal{S}$; the table SPLIT is used for placing each element $e \in U$ in C_1 or C_2 (i.e., coloring it), which is indicated by tuples $(e, 1)$ and $(e_2, 2)$; the tables A1 and A2 hold the occurrences of elements in sets, where each set has some label.

The action base $\mathcal{AB}$ contains $\texttt{assign}(X, Y)$ and $\texttt{trigger}(X, Y)$, which have empty precondition and the following Add- and Del-Sets:

$$
\begin{array}{ll}
\texttt{assign}: & Add(\texttt{assign}(X,Y)) = \{\ \text{SPLIT}(X,Y)\},\\
 & Del(\texttt{assign}(X,Y)) = \{\ \text{A1}(S,X,Y),\ \text{A2}(S,X,Y)\ \};\\
\texttt{trigger}: & Add(\texttt{trigger}(X,Y)) = \{\ \text{A1}(X,Y,0), \text{A2}(X,Y,0)\ \},\ Del(\texttt{trigger}(X,Y)) = \emptyset.
\end{array}
$$

[3] A functional dependency is a constraint $C : X \rightarrow A$ on a relation r, where A is an argument of r and $X = \{X_1, \ldots, X_n\}$ is a subset of arguments of r; it holds, if any two tuples in r which agree on the arguments in X agree also on A. In our framework, C can be expressed as an integrity constraint e.g. as follows: $\mathtt{in(T1,db:select(r)\&in(T2,db:select(r))\&T_1.X_1 = T_2.X_1\& \cdots \&T_1.X_n = T_2.X_n \Rightarrow T1.A = T2.A}$.

The program $\mathcal{P}$ has the single rule

$$\mathbf{Do}(\mathtt{trigger}(X,Y)) \quad \leftarrow \quad \mathrm{COLL}(X,Y)$$

Let D be the database instance such that COLL contains the collection $\mathcal{S}$, SPLIT is empty, and A1 (resp. A2) holds for each tuple (s,e) in COLL a tuple $(s,e,1)$ (resp. $(s,e,2)$). Moreover, suppose that the integrity constraints $\mathcal{IC}$ on $\mathcal{D}$ consist of the following FDs: the FD $El \to Color$ on ASSIGN, and the FD $Set \to Tag$ on A1 and A2.

Intuitively, the program forces the agent to add for every occurrence of an element in a set $S_i \in \mathcal{S}$, represented by a tuple (i,e) in COLL, a tuple $(i,e,0)$ to both A1 and A2. This triggers a violation of the FD $Set \to Tag$ on A1 and A2. This violation must be cured by executing $\mathtt{assign}(e_1,0)$ and $\mathtt{assign}(e_2,1)$ actions for some e_1, e_2 which occur in the set S_i; by the FD $El \to Color$ on SPLIT, e_1 must be different from e_2.

Hence, it is easy to see that $\mathcal{P}$ has a feasible status set on D, if and only if $\mathcal{S}$ is colorable by some coloring (C_1, C_2). Since a coloring (C_1, C_2) is easily constructed from any feasible status set S, the result follows. ∎

This result is quite negative, since it tells that already for very simple programs and very basic constraints, computing a feasible set is a hard problem. The reason is that the agent program $\mathcal{P}$ we have constructed in the reduction does not say anything about how and when to use the `assign` action, which does not show up in the program. If we had rules which tell the agent under which conditions a particular `assign` action should be taken or must not be taken, such a situation would hardly arise. However, since the program is underconstrained in that respect, an exponentiality of possibilities exists which must be explored by the agent.

The previous theorem shows that we benefit from using rational status sets instead of feasible status sets on positive programs in different respects. First, on the semantical side, we have a unique rational status set (if one exists) compared to a possible exponential number of feasible status sets, and second, on the computational side, we can compute the unique rational status set on an agent state in polynomial time, compared to the intractability of computing any feasible status set. Unfortunately, in the presence of negation, like on the semantical side, also on the computational side the appealing properties of rational status sets vanish.

8.2 Rational status sets

In the presence of integrity constraints, the complexity of rational status sets increases. The reason is that due to the integrity constraints $\mathcal{IC}$, a feasible set S may no longer necessarily contain a rational status set; deciding this problem is intractable.

Theorem 8.2 *Let $\mathcal{P}$ be a fixed agent program (not necessarily positive), suppose $\mathcal{IC}$ holds functional dependencies on a relational database $\mathcal{D}$. Let $\mathcal{O}_{\mathcal{S}}$ be an agent state, and let S be a feasible status set for $\mathcal{P}$ on $\mathcal{O}_{S}C$. Then, deciding whether S contains some rational status set (resp., S is rational) is* co-NP*-hard, even if $\mathcal{IC}$ contains a single FD.*

Proof. We prove this by a reduction from the M3DNF problem, which is a restriction of the DNF TAUTOLOGY problem (cf. [37]): Given a propositional formula $\phi = \bigvee_{i=1}^{m} D_i$ in DNF, where the D_i's are conjunctions of literals on a set of propositional variables $X = \{x_1, \ldots, x_n\}$, decide whether ϕ is a tautology. M3DNF is the restriction in which each D_i has three literals, and such that the literals of D_i are either all positive or all negative. For convenience, we allow repetition of the same literal in the same disjunct.

The database $\mathcal{D}$ contains three relations: POS(V_1, V_2, V_3) and NEG(V_1, V_2, V_3) for storing the positive and the negative disjuncts of ϕ, respectively, and a relation VAR$(Var, Value, Tag)$, which contains for each pair of variable a $x \in X$ and a value $v \in \{0, 1\}$ precisely one tuple. That is, the FD $Var, Value \to Tag$ is a constraint on VAR.

The initial database D contains the following tuples. For each positive disjunct $D_i = x_{i_1} \wedge x_{i_2} \wedge x_{i_3}$ from ϕ, the tuple $(x_{i_1}, x_{i_2}, x_{i_3})$ is in POS, and for each negative disjunct $D_i = \neg x_{i_1} \wedge \neg x_{i_2} \wedge \neg x_{i_3}$ the tuple $(x_{i_1}, x_{i_2}, x_{i_3})$ is in NEG. Moreover, for each propositional variables $x_i \in X$, the tuples $(x_i, 0, 0)$ and $(x_i, 1, 0)$ are in VAR.

The action base contains the three actions `all`, $\mathtt{set}(X, Y)$ and $\mathtt{addto\_var}(X, Y, Z)$, which have empty preconditions and the following add and delete sets:

`all`:	$Add(\mathtt{all}) = Del(\mathtt{all}) = \emptyset$;
$\mathtt{set}(X, Y)$:	$Add(\mathtt{set}(X, Y)) = \emptyset$, $Del(\mathtt{set}(X, Y)) = \{\text{VAR}(X, Y, 0)\}$;
$\mathtt{addto\_var}(X, Y, Z)$:	$Add(\mathtt{addto\_var}(X, Y, Z)) = \{\text{VAR}(X, Y, Z)\}$, $Del(X, Y, Z) = \emptyset$.

The program $\mathcal{P}$ is as follows:

$$
\begin{aligned}
\mathbf{Do}(\mathtt{set}(X, Y)) &\leftarrow \mathbf{Do}(\mathtt{all}), \text{VAR}(X, Y, Z).\\
\mathbf{Do}(\mathtt{all}) &\leftarrow \mathbf{Do}(\mathtt{set}(X, 0)), \mathbf{Do}(\mathtt{set}(X, 1)), \text{VAR}(X, Y, Z).\\
\mathbf{Do}(\mathtt{all}) &\leftarrow \neg\mathbf{Do}(\mathtt{set}(X, 0)), \neg\mathbf{Do}(\mathtt{set}(X, 1)), \text{VAR}(X, Y, Z).\\
\mathbf{Do}(\mathtt{all}) &\leftarrow \mathbf{Do}(\mathtt{set}(X, 0)), \mathbf{Do}(\mathtt{set}(Y, 0)), \mathbf{Do}(\mathtt{set}(Z, 0)), \text{POS}(X, Y, Z).\\
\mathbf{Do}(\mathtt{all}) &\leftarrow \mathbf{Do}(\mathtt{set}(X, 1)), \mathbf{Do}(\mathtt{set}(Y, 1)), \mathbf{Do}(\mathtt{set}(Z, 1)), \text{NEG}(X, Y, Z).\\
\mathbf{Do}(\mathtt{addto\_var}(X, Y, 1)) &\leftarrow \text{VAR}(X, Y, Z).
\end{aligned}
$$

Suppose that $\mathcal{IC}$ holds the single FD $Var, Value \to Tag$ on VAR. Let S be the smallest status set S which is closed under DCl and ACl and has the Do-Set

$$\mathbf{Do}(S) = \{\mathtt{all}\} \cup \{\mathtt{set}(x_i, v), \mathtt{addto\_var}(x_i, v, 1) \mid x_i \in X, v \in \{0, 1\}\}.$$

Then, it can be checked that S is a feasible status set of $\mathcal{P}$ on the initial database D.

We note that any feasible status set $S' \subset S$ must not contain $\mathbf{Do}(\mathtt{all})$, and must contain exactly one of $\mathbf{Do}(\mathtt{set}(x_i, 0))$,$\mathbf{Do}(\mathtt{set}(x_i, 1))$, for every $x_i \in X$; but, any such S' does never satisfy the FD $Var, Value \to Tag$ on VAR, since either the tuples $(x_i, 1, 0)$,$(x_i, 1, 1)$ are in VAR, or the tuples $(x_i, 0, 0)$, $(x_i, 0, 1)$, and therefore the FD $Var, Value \to Tag$ is violated on VAR.

It holds that S contains some rational status set (resp., that S is rational), if and only if formula ϕ is a tautology. The result follows. ∎

For the recognition problem, we thus obtain by Theorem 8.2 and Proposition 7.7 easily the following result.

Theorem 8.3 *Let $\mathcal{P}$ be a fixed agent program. Then, given an agent state $\mathcal{O}_\mathcal{S}$ and a status set S, deciding whether S is a rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, is* co-NP-*complete. Hardness holds even if $\mathcal{P}$ is positive.*

The computation problem for rational status sets is harder than for feasible status sets, and is beyond the polynomial time closure of NP.

Theorem 8.4 *Let $\mathcal{P}$ be a fixed agent program. Then, given an agent state $\mathcal{O}_\mathcal{S}$, deciding whether $\mathcal{P}$ has a rational status set on $\mathcal{O}_\mathcal{S}$ is Σ_2^P-complete, and computing any rational status set is $\mathrm{F}\Sigma_2^P$-complete.*

Proof. The problems are in Σ_2^P and $\mathrm{F}\Sigma_2^P$, respectively, since a rational status set S can be guessed an verified in polynomial time with the help of an NP oracle (cf. Theorem 8.3).

To show that the problems are hard for Σ_2^P and $\mathrm{F}\Sigma_2^P$, respectively, we extend the construction in the proof of Theorem 8.2, such that we encode the problem of computing, given a QBF $\exists Y \forall X.\phi$, an assignment χ to the Y-variables such that $\forall X.\phi[Y = \chi]$ is valid, where $\phi[Y = \chi]$ is the application of χ to the Y-variables in ϕ.

We use an additional relation YVAR for storing the Y-variables, and add the rule

$$\mathbf{Do}(\mathtt{set}(Y, 1)) \leftarrow \neg\mathbf{Do}(\mathtt{set}(Y, 0))$$

This rule enforces a choice between $\mathbf{Do}(\mathtt{set}(y_j, 0))$ and $\mathbf{Do}(\mathtt{set}(y_j, 1))$, for all $y_j \in Y$; each such choice χ generates a candidate S_χ for a rational status set.

It holds that every rational status set of $\mathcal{P}$ on D must be of the form S_χ, for some choice χ; moreover, the rational status sets of $\mathcal{P}$ on D correspond to the sets S_χ such that the formula $\forall X.\phi[Y = \chi]$ is valid. Therefore, deciding whether $\mathcal{P}$ has a rational status set on D is Σ_2^P-hard, and computing any rational status set is hard for $\mathrm{F}\Sigma_2^P$. The result follows. ∎

For action reasoning, we obtain from the preceding theorem easily the following result.

Theorem 8.5 *Let $\mathcal{P}$ be a fixed agent program. Then, given an agent state $\mathcal{O}_\mathcal{S}$ and a ground action α, deciding $\alpha \in \mathbf{Do}(S)$ holds for (i) every (resp., (ii) some) rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ is (i) Π_2^P-complete (resp., (ii) Σ_2^P-complete).*

Proof. Membership is immediate from Theorem 8.3: A guess for a rational status set S such that $\alpha \notin \mathbf{Do}(S)$ (resp., $\alpha \in \mathbf{Do}(S)$) can be verified with an NP oracle.

For the hardness parts, observe that $\mathtt{all} \in \mathbf{Do}(S)$ for every rational status set of the program $\mathcal{P}$ in the proof of Theorem 8.4; thus, by querying about $\mathtt{all}$, hardness for (i) holds. For (ii), query about α, where α is a fresh action which does not occur in $\mathcal{P}$. ∎

8.3 Reasonable status sets

For reasonable status sets, we find in all cases better computational properties than for rational status sets. This is explained by the fact that the criterion for a reasonable status set is much stronger than the one for a rational status set.

Indeed, this criterion is so strong, such that the presence of integrity constraints has no effect on tractability vs intractability issue of recognizing a reasonable status set. In both cases, a reasonable status set can be recognized in polynomial time (Proposition 7.15). Therefore, the same complexity results hold for programs with and without integrity constraints. (see Section 7.2.3).

8.4 Weak status sets

8.4.1 Positive programs

The recognition problem is no longer known to be polynomial if no integrity constraints are present in general. This is a consequence of the proof of the previous theorem.

Theorem 8.6 *Let $\mathcal{P}$ be a fixed positive agent program. Let $\mathcal{O}_{\mathcal{S}}$ be an agent state, and let S be a status set. Then, deciding whether S is a weak rational status set of $\mathcal{P}$ is* co-NP*-complete.*

Proof. To show that S is not a weak rational status set, we can proceed as follows. Check whether S is not $A(S)$-rational; if this is not the case (i.e., S is $A(S)$-rational), then guess some status set S' such that S' is $A(S')$-rational and $A(S') \supset A(S)$. Since checking A-rationality is polynomial if $\mathcal{P}$ is positive, it follows membership in $\mathrm{co\text{-}NP}$.

For the hardness part, we can establish a reduction from the M3SAT problem similar to the one in the proof of Theorem 7.6.

As there, we suppose that an M3SAT instance ϕ is stored in relations POS (positive clauses) and NEG (negative clauses), and we assume that all variables are stored in VAR. Moreover, there is a relation AUX(Var, Val), which contains in the initial database D all tuples $(x_i, 0)$, for all variables x_i.

The action base $\mathcal{AB}$ contains $\mathtt{set_0}(X)$ and $\mathtt{set_1}(X)$, which have both empty precondition, and both the add sets $\{\text{AUX}(Y, 1)\}$ and the delete sets $\{\text{AUX}(X, 0)\}$. Further, $\mathcal{AB}$ contains an action α with empty precondition and empty add and delete sets. Define the program $\mathcal{P}$ as follows.

$$
\begin{array}{rcl}
\mathbf{O}(\mathtt{set_0}(X)) & \leftarrow & \mathrm{VAR}(X) \\
\mathbf{O}(\mathtt{set_1}(X)) & \leftarrow & \mathrm{VAR}(X) \\
\mathbf{F}\alpha & \leftarrow & \mathbf{Do}(\mathtt{set_0}(X_1)), \mathbf{Do}(\mathtt{set_0}(X_2)), \mathbf{Do}(\mathtt{set_0}(X_3)), \text{POS}(X_1, X_2, X_3) \\
\mathbf{F}\alpha & \leftarrow & \mathbf{Do}(\mathtt{set_1}(X_1)), \mathbf{Do}(\mathtt{set_1}(X_2)), \mathbf{Do}(\mathtt{set_1}(X_3)), \text{NEG}(X_1, X_2, X_3) \\
\mathbf{P}\alpha & \leftarrow &
\end{array}
$$

In addition, we have the action constraint

$$AC: \quad \{\mathtt{set_0}(X), \mathtt{set_1}(X)\} \hookleftarrow \mathrm{VAR}(X)$$

(this action constraint can be easily represented by additional positive rules in $\mathcal{P}$; see Section 5.5). Moreover, $\mathcal{IC}$ contains the FD $Var \to Val$ on AUX. Clearly, the initial D satisfies this FD.

Then, for a given initial database D, the status set

$$S = \{\mathbf{O}(\mathtt{set_0}(x_i)), \mathbf{O}(\mathtt{set_1}(x_i)), \mathbf{P}(\mathtt{set_0}(x_i)), \mathbf{P}(\mathtt{set_1}(x_i)) \mid x_i \in X\} \cup \{\mathbf{P}\alpha\}$$

is an $\{\alpha\}$-set feasible status set. Moreover, it holds that S is a weak rational status set, if and only if there exists no status set S' such that S' is $A(S')$-feasible and $A(S') \supset \{\alpha\}$. Observe that any such S' must contain $\mathbf{Do}(\mathtt{set_0}(x_i))$ or $\mathbf{Do}(\mathtt{set_1}(x_i))$, for every $x_i \in X$, and thus corresponds to a truth assignment; indeed, taking $\mathtt{set\_0}(x_i)$ or $\mathtt{set\_1}(x_i)$ for any x_i adds the tuples $(x_j, 1)$ to AUX1, for all variables x_j.

Thus, S is a weak rational status set, iff ϕ is a No-instance; Since the database D is easily constructed from ϕ, the result follows. ∎

As we have seen in Section 7.1.1, a weak rational (resp., reasonable) status set of a fixed positive agent program can be computed in polynomial time using the algorithm COMPUTE-WEAK-RSS. Unfortunately, a similar polynomial algorithm in the presence of integrity constraints is unlikely to exist. This is a consequence of the following result.

Theorem 8.7 *Let $\mathcal{P}$ be a fixed positive agent program. Given an agent state $\mathcal{O}_\mathcal{S}$, deciding whether $\mathcal{P}$ has a weak rational status set on $\mathcal{O}_\mathcal{S}$ is* NP*-complete.*

Proof. The problem is in NP, since we can guess a set A of ground actions, compute $S = \mathit{lfp}(T_{\mathcal{P},\mathcal{O}_\mathcal{S},A})$ and check whether S is A-feasible in polynomial time. If such a set A exists, then $\mathcal{P}$ has some weak rational status set on $\mathcal{O}_\mathcal{S}$.

NP-hardness can be shown by a slight extension to the reduction in the proof of Theorem 8.6. Without loss of generality, the M3SAT formula ϕ from the reduction there is only satisfiable if a designated variable x_1 is set to true. Thus, if we add the rule $\mathbf{Do}\mathtt{set_1}(x_1) \leftarrow$ to the program $\mathcal{P}$, then the resulting program has a weak rational status set if and only if ϕ is satisfiable. ∎

For the computation problem, we have the algorithm COMPUTE-PIC-WEAK-RATIONAL-SS below, which makes use of an NP oracle.

Algorithm COMPUTE-PIC-WEAK-RATIONAL-SS

Input: agent state $\mathcal{O}_\mathcal{S}$ (fixed positive agent program $\mathcal{P}$; $\mathcal{IC}$ is arbitrary)

Output: a weak rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, if one exists; "No", otherwise.

Method

Step 1. Set $A_{new} := \emptyset$, GA := set of all ground actions.

Step 2. Query the oracle whether some $A \supseteq A_{new}$ exists such that $S' = \mathit{lfp}(T_{\mathcal{P},\mathcal{O}_\mathcal{S},A})$ is $A(S')$-feasible.

Step 3. If the answer is "yes", then let $S := \mathit{lfp}(T_{\mathcal{P},\mathcal{O}_\mathcal{S},A_{new}})$ and set $A_{old} := A(S)$, $GA := GA \setminus A_{old}$; otherwise, if $A_{new} = \emptyset$, then output "No" and halt.

Step 4. If $GA = \emptyset$, then output S and halt.

Step 5. choose some $\alpha \in GA$, and set $A_{new} := A_{old} \cup \{\alpha\}$, $GA := GA \setminus \{\alpha\}$; continue at Step 2.

This algorithm computes a weak rational status set in polynomial time modulo calls to the NP oracle. Therefore, the problem is in $\mathrm{FP}^{\mathrm{NP}}$. Observe that in case $\mathcal{IC} = \emptyset$, the NP-oracle can be replaced by a polynomial time algorithm, such that we obtain an overall polynomial algorithm similar to COMPUTE-WEAK-RSS.

Like in other cases, the $\mathrm{FP}^{\mathrm{NP}}$ upper bound for the computation problem can also be lowered to $\mathrm{FNP}//\log$ in this case.

Theorem 8.8 *Let $\mathcal{P}$ be a fixed positive agent program. Then, computing a weak rational status set is in* $\mathrm{FNP}//\log$ *and hard for both* FNP *and* $\mathrm{FP}^{\mathrm{NP}}_{\|}$.

Proof. For computing a weak rational status set, we can proceed as follows. First, compute the maximum size $s = |A(S)|$ over all status sets S such that S is $A(S)$-rational; then, generate nondeterministically a status set S which is $A(S)$-rational and such that $|A(S)| = s$, and output this set (so one exists).

The correctness of this algorithm follows from Proposition 5.10. Moreover, checking whether S is $A(S)$-rational is polynomial if $\mathcal{P}$ is positive, as follows from Propositions 7.18 and 7.2. Consequently, computing a weak rational status set is in $\mathrm{FNP}//\log$ in this case. FNP-hardness follows from the proof of Theorem 8.7; the weak rational status sets of the program from the proof of this theorem correspond to the satisfying assignments of an M3SAT instance, whose computation is easily seen to be FNP-complete by the FNP-completeness of SAT.

For the proof of $\mathrm{FP}_{\|}^{\mathrm{NP}}$-hardness, we use the fact that given instances $I_1, \ldots, I_n$ of any arbitrary fixed co-NP-complete problem Π, computing the binary string $B = b_1 \cdots b_n$ where $b_i = 1$ if I_i is a Yes-instance of Π and $b_i = 0$ otherwise, is $\mathrm{FP}_{\|}^{\mathrm{NP}}$-hard (this is easily seen).

We choose for this problem the recognition of a weak rational status set S of a fixed positive agent program $\mathcal{P}$, which is co-NP-complete by Theorem 8.6.

We may assume that $\mathcal{P}$ is the program from the proof of this result, and S the status set constructed over the database D constructed for a formula ϕ. We observe that $\mathcal{P}$ has weak rational status set on $\mathcal{P}$, and that S is the unique weak rational status set, iff the formula ϕ is unsatisfiable. Thus, from any arbitrary weak rational status set S' of $\mathcal{P}$ over D, it is immediate whether S is weak rational or not.

Consequently, computing weak rational status sets $S_1, \ldots, S_n$ of $\mathcal{P}$ over given databases $D_1, \ldots,$ D_n is $\mathrm{FP}_{\|}^{\mathrm{NP}}$-hard.

It remains to show that the computation of $S_1, \ldots, S_n$ can be reduced to the computation of a single weak rational status set S of a fixed program $\mathcal{P}'$ over a database D'.

For thus purpose, we merge the databases D_i into a single database. This is accomplished by tagging each tuple in D_i with i, i.e., add a new attribute A in each relation, and each tuple obtains value i on it; A is added on the left hand side of each functional dependency. Moreover, an additional argument T for the tag is introduced in each action, and all literals in a rule have the same fresh variable T in the tag position.

Then, the resulting program $\mathcal{P}'$ has some weak rational status set S on the union D' of the tagged D_i's, and from any such S weak rational status sets $S_1, \ldots, S_n$ of $\mathcal{P}$ on each D_i are easily obtained in polynomial time. On the other hand, D' is polynomial-time constructible from $D_1, \ldots, D_n$. ∎

We finally address the problem of action reasoning.

Theorem 8.9 *Let $\mathcal{P}$ be a fixed positive agent program $\mathcal{P}$. Let $\mathcal{O}_{\mathcal{S}}$ be a given agent state and let α be a given ground action atom. Then, deciding whether $\alpha \in \mathbf{Do}(S)$ holds for (i) every (resp., (ii) some) weak rational status set of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$ is Π_2^P-complete (resp., NP-complete).*

Proof. The membership part of (ii) is easy from Theorem 8.6. A guess for a weak rational status set S such that $A \notin \mathbf{Do}(S)$ can be verified with an NP oracle in polynomial time.

For the membership part of (i), observe that under the assumptions, if S is an A rational status set, then any A'-rational status set S' such that $A' \supseteq A$ satisfies $S' \supseteq S$. Therefore, it suffices to guess a status set S such that S is $A(S)$-rational and $A \in \mathbf{Do}(S)$; the guess can be verified in polynomial time.

Hardness for (i) follows from Theorem 7.6. The hardness part for (ii) can be shown by a suitable extension of the construction in the proof of Theorem 8.6, such that validity of a quantified Boolean formula $\forall Y \exists X \phi$, can be decided, where ϕ is in M3SAT form.

Assume without loss of generality that no clause of ϕ has all its variables from Y, and that ϕ can only be satisfied if a particular variable $x_1 \in X$ is set to true. We introduce two new relations YVAR for storing the variables in Y (VAR stores $X \cup Y$) and AUX1(Var, Val), on which also the FD $Var \rightarrow Val$ applies. Store in the database D for each variable $x_i \in X$ a tuple $(x_i, 0)$ in AUX, and for each y_i a tuple $(y_i, 0)$ in AUX1.

The delete sets of $\mathtt{set_0}(X)$ and $\mathtt{set_1}(X)$ are augmented by AUX$(X, 0)$. Moreover, we introduce a new action add_to_aux1(Y), which has empty precondition, empty delete set, and add set $\{$ AUX1$(Y, 1)$ $\}$.

Finally, add to the program $\mathcal{P}$ in the proof of Theorem 8.6 the following rule:

$$\mathbf{Do}(\mathtt{add\_to\_aux1}(Y)) \quad \leftarrow \quad \text{YVAR}(Y)$$

These modifications have the following effect. The added rules enforces that for each variable $y_i \in Y$, the tuple $(y_i, 1)$ is added to AUX1, and either $\mathtt{set_0}(y_i)$ or $\mathtt{set_1}(y_i)$ must be executed, in order to maintain the FD $Var \rightarrow Val$ on AUX1. Thus, every weak rational status set of the constructed program on D contains either $\mathbf{Do}(\mathtt{set_0})(y_i)$ or $\mathbf{Do}(\mathtt{set_1})(y_i)$, for each $y_i \in Y$.

On the other hand, for each such choice χ, which embodies a truth assignment to Y, by the assumption on ϕ a weak rational status set exists; if all obligations $\mathtt{set_0}(x_i)$, $\mathtt{set_1}(x_i)$, $x_i \in X$ are violated, then we obtain a respective A-feasible status set S_χ, and therefore, since the program is positive, a weak rational status set $S' \supseteq S_\chi$ exists. It holds that S_χ is weak rational, iff $\exists X \phi[Y = \chi]$ is unsatisfiable.

By our assumptions, it follows that $\mathtt{set_1}(x_1) \in \mathbf{Do}(S)$ for every weak rational status set S of the program on D, iff formula ϕ is valid. This proves the hardness part for (ii) and the result. ∎

8.4.2 Programs with negation

Let us now consider programs with negation. In this case, weak rational and weak reasonable status sets are no longer identical in all cases.

Weak reasonable status sets. For weak reasonable status sets, we can observe that integrity constraints do not add on the complexity; under both presence and absence of integrity constraints, we obtain the same results for the worst case complexity. In particular, all the general upper bounds that we have derived for weak reasonable status sets in Section 7.2.4, apply to programs with integrity constraints as well. It thus remains to consider weak rational status sets.

Weak rational status sets. The existence problem of an A-rational status set has the same complexity as the existence problem of a rational status set (Proposition 7.18). Since a weak rational status set exists if and only if an A-rational status set exists for some A, we obtain from Theorem 8.4 resp. its proof the following result.

Theorem 8.10 *Let $\mathcal{P}$ be a fixed agent program. Then, given an agent state $\mathcal{O}_\mathcal{S}$, deciding whether $\mathcal{P}$ has a weak rational status set on $\mathcal{P}$ is Σ_2^P-complete.*

For the computation of a weak rational status set, we can use a modified version of the algorithm COMPUTE-WEAK-RATIONAL-SS, by replacing A-feasible sets with A-rational sets. This increases the complexity, as we have to replace the NP oracle by a Σ_2^P oracle. We thus obtain that the problem belongs to $\mathrm{FP}^{\Sigma_2^P}$. A slightly better upper bound can be given.

Theorem 8.11 *Let $\mathcal{P}$ be a fixed agent program. Then, computing any weak rational status set of $\mathcal{P}$ on a given agent state $\mathcal{O}_{\mathcal{S}}$ is in $\mathrm{FP}^{\Sigma_2^P} \cap \mathrm{RP} \cdot \mathrm{FP}_{\|}^{\Sigma_2^P}$ and hard for both $\mathrm{F}\Sigma_2^P$ and $\mathrm{FP}_{\|}^{\Sigma_2^P}$.*

Proof. Membership in $\mathrm{FP}^{\Sigma_2^P}$ was discussed above. Membership in $\mathrm{RP} \cdot \mathrm{FP}_{\|}^{\Sigma_2^P}$ can be established using results from [24]. In fact, the computation of a weak rational status set in the most general setting can be easily expressed as a maximization problem (MAXP) as defined in [24], such that the instance-solution relation is co-NP-decidable; for such problems, $\mathrm{RP} \cdot \mathrm{FP}_{\|}^{\Sigma_2^P}$ is an upper bound.

Hardness for $\mathrm{F}\Sigma_2^P$ is immediate from the proof of Theorem 8.4 (existence of a rational status set), since **O** does not occur in the program constructed. Hardness for $\mathrm{FP}_{\|}^{\Sigma_2^P}$ can be established as follows. Let Π be any Σ_2^P-complete problem. Then, computing, given instances $I_1, \ldots, I_n$ of Π, the binary string $B = b_1 \cdots b_n$ where $b_i = 1$ if I_i is a Yest-instance and $b_i = 0$ otherwise, is easily seen to be hard for $\mathrm{FP}_{\|}^{\Sigma_2^P}$.

From the proof of Theorem 8.4, we know that deciding whether a fixed agent program $\mathcal{P}$, in which **O** does not occur, has a rational status set on a given database D is Σ_2^P complete. Thus, for given databases $D_1, \ldots, D_n$, computing the string B is $\mathrm{FP}_{\|}^{\Sigma_2^P}$-hard.

The different instances can be combined into a single instance of a new fixed program as follows. Take a fresh action α, which does not occur in $\mathcal{P}$ such that $Pre(\alpha)$ is void and $Add(\alpha) = Del(\alpha) = \emptyset$. Add the atom $\mathbf{Do}\alpha$ in the body of each rule in $\mathcal{P}$, and add the rule $\mathbf{O}\alpha \leftarrow$. Then the resulting program $\mathcal{P}_0$ has some weak rational status set S on each D_i, and for any such S it holds $\alpha \in \mathbf{Do}(S)$ iff $\mathcal{P}_0$ has a rational status set on D_i.

The databases D_i can be merged into a single database D' for a new fixed program $\mathcal{P}'$, in the same way as described in the proof of Theorem 8.8, by tagging the databases D_i with i and taking their union. This program $\mathcal{P}'$ has some weak rational status set S on D'; moreover, for every such S, it holds that $\alpha(i) \in \mathbf{Do}(S)$ iff $\mathcal{P}$ has a rational status set on D_i; thus, from an S the string binary string B is easily computed.

Since the database D' is polynomial-time constructible from $D_1, \ldots, D_n$, it follows that computing a weak rational status set is hard for $\mathrm{FP}_{\|}^{\Sigma_2^P}$. ∎

Next we consider the recognition problem. Here, the complexity increases if integrity constraints are allowed; the benign property that a A-feasible status set is A-rational, if no smaller A-feasible status set exists is no longer valid.

Theorem 8.12 *Let $\mathcal{P}$ be a fixed agent program. Then, given an agent state $\mathcal{O}_{\mathcal{S}}$ and a status set S, deciding whether S is a weak rational status set of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$ is Π_2^P-complete.*

Proof. For the membership part, consider the following algorithm for showing that S is not a weak rational status set. First, check whether S is not an $A(S)$-rational status set. If S is found $A(S)$-rational, then guess $A' \supset A(S)$ and S' and check whether S' is A'-rational. Since Checking A-rationality of S is in co-NP, this is an Σ_2^P algorithm for refuting S as a weak rational status set; this proves the membership.

For the hardness part, we adapt the construction in the proof of Theorem 8.2 for QBF formulas $\exists Y \forall X \phi$, by adding the $\exists Y$ quantifier block.

We use the database $\mathcal{D}$, the actions base $\mathcal{AB}$, and the integrity constraints as there, but add to $\mathcal{D}$ another relation YVAR for storing the Y-variables (the X-variables are in VAR) and introduce another action α, which has empty precondition and empty add and delete sets.

We add the following clauses in the program:

$$\begin{array}{rcl}
\mathbf{O}(\alpha) & \leftarrow & \\
\mathbf{O}(\mathtt{set}(Y,0)) & \leftarrow & \text{YVAR}(Y) \\
\mathbf{O}(\mathtt{set}(Y,1)) & \leftarrow & \text{YVAR}(Y) \\
\mathbf{Do}(\mathtt{set}(Y,0)), & \leftarrow & \mathbf{Do}(\alpha), \neg\mathbf{Do}(\mathtt{set}(Y,1)), \text{YVAR}(Y) \\
\mathbf{Do}(\alpha) & \leftarrow & \mathbf{Do}(\mathtt{set}(Y,0)), \text{YVAR}(Y) \\
\mathbf{Do}(\alpha) & \leftarrow & \mathbf{Do}(\mathtt{set}(Y,1)), \text{YVAR}(Y)
\end{array}$$

and we set up the action constraint

$$AC: \quad \{\mathtt{set}(Y,0), \mathtt{set}(Y,1)\} \hookleftarrow \text{YVAR}(Y).$$

(The use of AC can be surpassed, but is convenient.) Let the resulting program be $\mathcal{P}'$.

These rules state that the agent is obliged to execute α and to set every variable $y_i \in Y$ to true and false, which however is prohibited by AC. Moreover, each y_i must have assigned a value if α is executed, and if some variable receives a value, then α is executed; consequently. if α is executed, then every y_i gets precisely one value, and if α is not executed, then no y_i gets a value.

Let S_0 be the status sets defined by

$$S_0 = S \cup \{\mathbf{O}\alpha, \mathbf{P}\alpha\} \cup \mathbf{O}(\mathtt{set}(y,v)), \mathbf{P}(\mathtt{set}(y,v)) \mid y \in Y, v \in \{0,1\}\},$$

where S is the status set from the proof of Theorem 8.2.

Then, S_0 is an $A(S)$-rational status set, in which all the obligations from the newly added rules are violated.

It holds that S_0 is the (unique) weak rational status set of $\mathcal{P}'$ iff $\forall Y \exists X \neg\phi$ is true.

($\Rightarrow$) Suppose S_0 is weak rational. Then, it is impossible for any choice χ from $\mathtt{set}(y,0), \mathtt{set}(y,1)$, $y \in Y$, to find an A-rational status set where the obligation followed in A correspond to χ.

In particular, the status set

$$S_\chi = S_0 \cup \{\mathbf{Do}(\mathtt{set}(y,v)) \mid \mathtt{set}(y,v) \in \chi\} \cup \{\mathbf{Do}\alpha\}$$

is not weak rational. It holds that S_χ is $A(S_\chi)$-feasible; hence, there must exist some truth assignment τ to X such that $\phi[\tau(X), \sigma(Y)]$ is false, where σ is the truth assignment to Y such that $\sigma(y) \equiv \mathtt{set}(y,1) \in \chi$. Hence, $\forall X \exists Y \neg\phi$ is true.

($\Leftarrow$) Suppose $\forall Y \exists X \neg\phi$ is true. Consider any weak rational status set S of $\mathcal{P}'$. Then, either (i) $A(S) = A(S_0)$, or (ii) $A(S)$ defines a choice χ from $\mathtt{set}(y,0), \mathtt{set}(y,1)$, $y \in Y$, and $\alpha \in S$.

Assume (ii) and consider the following two cases:

(1) $\mathbf{Do}(\mathtt{all}) \notin S$. Then, exactly on of the actions $\mathtt{set}(x,0)$, $\mathtt{set}(x,1)$ must be in S, for every $x \in X$. But then, executing $\mathbf{Do}(S)$ violates the integrity constraints $\mathcal{IC}$, which contradicts that S is a weak rational status set.

(2) $\mathbf{Do}(\texttt{all}) \in S$. Then, for some truth assignment τ to X, we have that $\phi[\tau(X), \sigma(Y)]$ is false, for σ defined as previously. Consequently, there is an $A(S)$-feasible status set S' such that $S' \subset S$, which contradicts $A(S)$-rationality of S.

Hence, case (ii) is impossible, and case (i) applies to S. Consequently, S_0 is a weak rational status set. It can be seen that $S = S_0$ must hold. This proves the result. ∎

The last result that we turn to in this subsection is action reasoning under weak rational status sets. Here we face the full complexity of all conditions that we have imposed on acceptable status sets.

Theorem 8.13 *Let $\mathcal{P}$ be a fixed agent program $\mathcal{P}$. Let $\mathcal{O}_\mathcal{S}$ be a given agent state and let α be a given ground action atom. Then, deciding whether $\alpha \in \mathbf{Do}(S)$ holds for (i) every (resp., (ii) some) weak rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ is Π_3^P-complete (resp., Σ_3^P-complete).*

Proof. The membership part is routine: A guess for a weak rational status set S such that $\alpha \notin S$ (resp., $\alpha \in S$) can be verified with an Σ_2^P oracle in polynomial time (Theorem 8.12).

For the hardness part, we extend the construction in the proof of Theorem 8.12 to QBF formulas $\forall Z \exists Y \forall X \, \phi$, by adding another quantifier block.

For that, we introduce a new relation ZVAR for storing the variables in Z, and add the following clauses to the program:

$$\begin{aligned} \mathbf{O}(\texttt{set}(Z,0)) &\leftarrow \text{ZVAR}(Z), \\ \mathbf{O}(\texttt{set}(Z,1)) &\leftarrow \text{ZVAR}(Z), \\ \mathbf{Do}(\texttt{set}(Y,0)), &\leftarrow \neg\mathbf{Do}(\texttt{set}(Y,1)), \text{ZVAR}(X); \end{aligned}$$

denote this program by $\mathcal{P}''$. Moreover, we add another action constraint

$$AC': \quad \{\texttt{set}(Z,0), \texttt{set}(Z,1)\} \hookleftarrow \text{ZVAR}(Z).$$

Similar as the rules for the variables in Y, these rules force the agent to make a choice χ from $\mathbf{Do}(\texttt{set}(z,0)$, $\mathbf{Do}(\texttt{set}(z,1))$, for all $z \in Z$, in every weak rational status set. Upon such a choice, the program $\mathcal{P}''$ behaves like the program $\mathcal{P}'$.

For any such χ, it holds that a weak rational status set S implementing this χ satisfies $\mathbf{Do}\alpha \in S$, iff $\exists Y \forall X \, \phi[\sigma(Z)]$ is true, where σ is the truth assignment to the Z-variables according to χ.

It holds that $\alpha \in \mathbf{Do}(S)$ for every weak rational status set of $\mathcal{P}''$, iff $\forall Z \exists Y \forall X \, \phi$ is true.

This proves Π_3^P-hardness of (i). For (ii), we add the rule

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

in the program, where β is a fresh action.

Then, it holds a status set S such that $\beta \in \mathbf{Do}(S)$ is a weak rational status set of the resulting program $\mathcal{P}^*$, iff $S' = S \setminus \{\mathbf{Do}\alpha, \mathbf{P}\alpha\}$ is a weak rational status set of $\mathcal{P}'''$.

Hence, $\beta \in \mathbf{Do}(S)$ for some weak rational status set of $\mathcal{P}^*$, iff α is not a cautious consequence of $\mathcal{P}'''$. This proves Σ_3^P-hardness of (ii), and completes the proof of the theorem. ∎

8.5 Preferred status sets

Let us now consider the effect of integrity constraints on F-preferred status sets. It appears that for rational status sets, we have a complexity increase, while for reasonable status sets, the complexity remains unchanged (see Section 7.3).

We start with the recognition problem for F-preferred rational status sets. In the presence of integrity constraints, this problem migrates to the next level of the polynomial hierarchy. For the proof of this result, we use the following convenient lemma.

Notation. Let ϕ be propositional formula, and let $\chi \in 2^Y$ be a choice from the subsets of a set Y of variables. Then, $\phi[Y = \chi]$ denotes the formula obtained by substituting in ϕ the value true for every $y \in Y$ which is in χ, and the value false for every $y \in Y$ which is not in χ.

Lemma 8.14 *Let $\Phi' = \exists Y' \forall X' \phi'$ be a QBF such that ϕ' is in DNF. Then, a formula $\Phi = \exists Y \forall X \phi$, where ϕ is in M3DNF (see proof of Theorem 8.2 for M3DNF) can be constructed in polynomial time, such that*

(1) *for $Y = \emptyset$, the formula $\forall X \phi[Y = \emptyset]$ is true;*

(2) $\Phi' \longleftrightarrow (\exists Y \neq \emptyset)(\forall X)\phi$ *holds.*

Proof. See Appendix B. ∎

Theorem 8.15 *Let $\mathcal{P}$ be a fixed agent program. Then, given an agent state $\mathcal{O}_\mathcal{S}$ and a status set S, deciding whether S is a F-preferred rational status set of $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$ is Π^P_2-complete. Hardness holds even if $\mathcal{IC}$ holds functional dependencies on a relational database.*

Proof. Checking whether S is a rational status set can be done with a call to an NP oracle (Theorem 8.3), and a rational status set S' such that $\mathbf{F}(S') \subset \mathbf{F}(S)$ can be guessed an checked in polynomial time with an NP oracle; hence, showing that S is not a F-preferred rational status set is in Σ^P_2.

The proof of hardness is an extension to the proof of Theorem 8.2. We encode the $\exists Y \forall X \phi$ QBF problem, by adding an alternate block of quantifiers in the construction.

For convenience, we may start from the formula $\exists Y \forall X.\phi$ as in Lemma 8.14, and encode the failure of condition (2) of it; this is a Π^P_2-hard problem.

As in the construction of the proof of Theorem 8.2, we assume that database tables POS($Var1$, $Var2$, $Var3$) and NEG($Var1, Var2, Var3$) contain the positive and negative disjuncts of ϕ', respectively, and that the table VAR($Var, Value, Tag, Tag1$) contains all variables, but now has an additional tag ($Tag1$), which indicates whether a variable is from X (value 0) or from Y (value 1).

Thus, $\mathcal{IC}$ holds two FDs on VAR: $Var, Value \rightarrow Tag$ and $Var \rightarrow Tag1$.

The action base $\mathcal{AB}$ is the same, with the only differences that `addto_var` has a fourth parameter W, and that occurrence of VAR(X, Y, Z) has to be replaced by VAR(X, Y, Z, W).

Modify and extend the program $\mathcal{P}$ to a program $\mathcal{P}'$ as follows. First, the occurrences of every atom VAR(X, Y, Z) have to be replaced by VAR($X, Y, Z, 0$), and `addto_var`($X, Y, 1$) is uniformly replaced by `addto_var`($X, Y, 1, 0$). Then, add the rules

$$
\begin{array}{lrcl}
(1) & \mathbf{F}(\mathtt{set}(X,1)) & \leftarrow & \mathbf{P}(\mathtt{set}(X,0)), \mathrm{VAR}(X,Y,Z,1) \\
(2) & \mathbf{Do}(\mathtt{set}(X,1)) & \leftarrow & \neg\mathbf{Do}(\mathtt{set}(X,0)), \mathrm{VAR}(X,Y,Z,1)
\end{array}
$$

For each choice $\chi \in 2^Y$ from the subsets of Y, we obtain a candidate S_χ which is a feasible status set of $\mathcal{P}'$. This candidate is the deontic and action closure of the set

$$\{\, \mathbf{Do}(\mathtt{set}(y_i, 1) \mid y_i \in \chi \,\} \cup \{\, \mathbf{Do}(\mathtt{set}(y_i, 0)), \mathbf{F}(\mathtt{set}(y_i, 1)) \mid y_i \in Y \setminus \chi \,\},$$

where S is the feasible status set from the construction in the proof of Theorem 8.2.

For the assignment $Y = \emptyset$, S_χ has a "maximal" F-part on the $\mathbf{F}(\mathtt{set}(X, 1))$ atoms over all χ; this is also the maximal F-part possible for any rational status set of $\mathcal{P}'$. Moreover, by Lemma 8.14 and the construction of $\mathcal{P}'$, this S_{χ_0} is a rational status set.

We claim that S_{χ_0} is F-preferred, if and only if $\exists Y \neq \emptyset\, \forall X\, \phi$ is false.

($\Rightarrow$) Suppose S_{χ_0} is F-preferred. Then, no rational status set S' exists such that $\mathbf{F}(S') \subset \mathbf{F}(S)$. In particular, no candidate set S_χ for some $\chi \neq \chi_0$ can amount to a rational status set. Similar as in the proof of Theorem 8.2, we conclude that the formula $\forall X.\phi[Y = \chi]$ must be false. Hence, the formula $\exists Y \neq \emptyset \forall X.\phi$ is false.

($\Leftarrow$) Suppose S_{χ_0} is not F-preferred. Then, there exists a rational set S' such that $\mathbf{F}(S') \subset \mathbf{F}(S)$.

The clause (1) ensures that there is at most one of $\mathbf{Do}(\mathtt{set}(y, 0))$ and $\mathbf{Do}(\mathtt{set}(y, 1))$ in S', for every $y \in Y$; moreover, by the clause (2), precisely one of them is in S'. This S' amounts to a choice χ from the subsets of Y such that $\chi \neq \emptyset$, given by $y \in \chi$ iff $\mathbf{Do}(\mathtt{set}(y, 1)) \in S'$, for every $y \in Y$.

The program $\mathcal{P}'$, under S' and choice χ, basically reduces to a program $\mathcal{P}$ for a formula $\forall.\phi[Y = \chi]$. Notice that by the rationality of S', no atoms for status $\mathbf{O}$ or $\mathbf{W}$ are in S', and neither atoms $\mathbf{F}\alpha$ for irrelevant actions α. Moreover, rationality of S' implies that the formula $\forall.\phi[Y = \chi]$ must evaluate to true. This means that $\exists Y \neq \emptyset\, \forall X.\phi$ is true. ∎

Since the recognition of F-preferred rational status sets is gets more complex in the presence of integrity constraints, also the complexity of computing such a status set increases; the increase is one level in the polynomial hierarchy.

Theorem 8.16 *Let $\mathcal{P}$ be a fixed agent program. Then, given an agent state $\mathcal{O}_{\mathcal{S}}$, computing any F-preferred rational status set of $\mathcal{P}$ on $\mathcal{O}_{\mathcal{S}}$ (so one exists), is in* $\mathrm{FP}^{\Sigma_2^P} \cap \mathrm{RP} \cdot \mathrm{FP}_{\|}^{\Sigma_2^P}$ *and hard for both* $\mathrm{F}\Sigma_2^P$ *and* $\mathrm{FP}_{\|}^{\Sigma_2^P}$.

Proof. We can use the same algorithm as for the computation where $\mathcal{IC} = \emptyset$, in which we have to replace the NP-oracle by a Σ_2^P-oracle. This proves membership in $\mathrm{FP}^{\Sigma_2^P}$. Membership in $\mathrm{RP} \cdot \mathrm{FP}_{\|}^{\Sigma_2^P}$ follows from the fact that computing a F-preferred rational status set can be easily expressed as a maximization problem (MAXP) as defined in [24], whose instance-solution relation is co-NP-decidable; as mentioned previously, $\mathrm{RP} \cdot \mathrm{FP}_{\|}^{\Sigma_2^P}$ is an upper bound for such problems.

Hardness for $\mathrm{F}\Sigma_2^P$ is immediate from the proof of Theorem 8.4 (compute a rational status set), since each rational status set of the program $\mathcal{P}$ constructed there is F-preferred. The lower bound of hardness for $\mathrm{FP}_{\|}^{\Sigma_2^P}$ can be shown following the line of the proof of Theorem 8.11, where we reduce the computation of the binary string B for instances $I_1, \ldots, I_n$ of the Σ_2^P-complete complement of the recognition problem for F-preferred rational status sets.

We may suppose that the program is $\mathcal{P}'$ from the proof of Theorem 8.15, and that the set S to check over database D is the set S_{χ_0}, which is rational and has the maximal F-part over all rational status sets. Similar as in the proof of Theorem 8.11, we tag databases D_i by introducing a new column

T in each table which is added on the left hand side of FDs, and we add for the tag a variable T to each action scheme. Then, the F-preferred rational status sets S of the obtained program $\mathcal{P}''$ over the union $\hat{D}$ of all tagged databases D_i, are given by the union of the F-preferred rational status sets S_i of $\mathcal{P}''$ over each tagged database D_i. Hence, from any F-preferred status set S of $\mathcal{P}''$, the desired string B can be efficiently computed. Moreover, $\hat{D}$ is constructible in polynomial time from $I_1, \ldots, I_n$. It follows that computing a F-preferred rational status set is $\mathrm{FP}_{\|}^{\Sigma_2^P}$-hard, which proves the result. ∎

The last result of this subsection concerns action reasoning for F-preferred rational status sets. It shows that this task has the highest complexity of all the problems considered, and is located at the third level of the polynomial hierarchy.

Theorem 8.17 *Let $\mathcal{P}$ be a fixed program. Then, deciding whether an action status atom A belongs to some (resp., all) F-preferred rational status sets is Σ_3^P-complete (resp., Π_3^P-complete).*

Proof. The membership part is similar as in the case where no integrity are present (Theorem 7.29), with the difference that we need a Σ_2^P oracle instead of a NP oracle for checking whether an status set is an F-preferred rational status set.

The hardness part is an extension of the construction in the proof of Theorem 8.15.

We add another block of quantifiers $\forall Z$ in front of the formula Φ; Lemma 8.14 generalizes to the case where free variables occur in Φ'.

The action base and the database is the same, and field $Tag1$ has value 2 for identifying the Z variables. We add to the program $\mathcal{P}'$ the following clauses:

$$\begin{aligned}
\mathbf{F}(\mathtt{set}(X,1)) &\leftarrow \mathbf{P}(\mathtt{set}(X,0)), \mathrm{VAR}(X,Y,Z,2)\\
\mathbf{F}(\mathtt{set}(X,0)) &\leftarrow \mathbf{P}(\mathtt{set}(X,1)), \mathrm{VAR}(X,Y,Z,2)\\
\mathbf{Do}(\mathtt{set}(X,1)) &\leftarrow \neg\mathbf{Do}(\mathtt{set}(X,0)), \mathrm{VAR}(X,Y,Z,2)
\end{aligned}$$

These clauses effect a choice χ of a subset of Z, which is passed to the rest of the program, similar as the rules for a choice from the subsets of Y; however, here the choice entails that the F-parts of candidates corresponding to different choices χ and χ' are incomparable.

Furthermore, if we add to the program a rule

$$\mathbf{Do}(\alpha) \leftarrow \mathbf{P}(\mathtt{set}(X,0)), \mathrm{VAR}(X,Y,Z,1),$$

where α is some new action without effects, then $\mathbf{Do}(\alpha)$ is contained in every F-preferred rational status set of the resulting program $\mathcal{P}''$, if and only if the formula $(\forall Z)(\exists Y \neq \emptyset)(\forall X)\phi$ is true. To see this, notice that this rule can be applied in a F-preferred rational status set S, if and only if S does not contain for all variables $y \in Y$ the atoms $\mathbf{F}(set(y,1))$. $\mathbf{Do}(\alpha)$ is contained in all candidate F-preferred rational status sets, and is dispensable if and only if the formula $\forall Z \exists Y \neq \emptyset \forall X.\phi$ is true.

If we add a rule

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

to $\mathcal{P}''$, where β is an action of the same type as α, then $\mathbf{Do}(\beta)$ belongs to some F-preferred status set of the obtained program $\mathcal{P}'''$, if and only if the formula $\forall Z \exists Y \neq \emptyset \forall X.\phi$ is false. Indeed, it is not hard to see that a status set S such that $\mathbf{Do}(\beta), \mathbf{P}(\beta) \in S$ is a F-preferred status set of $\mathcal{P}'''$, if and only if the set $S' = S \setminus \{\mathbf{Do}(\beta), \mathbf{P}(\beta)\}$ is a F-preferred status set of $\mathcal{P}''$ such that $\mathbf{Do}(\alpha) \notin S$.

This implies Π_3^P-hardness (resp. Σ_3^P-hardness), and completes the proof of the theorem. ∎

9 Relation to Logic Programming

Thus far in this paper, we have introduced several semantics for agent programs. In this section, we will show that these semantics for agent programs are specifically tied to well known semantics for logic programs. In particular, we will show that three major semantics for logic programs may be "embedded" within the concept of agent programs.

- First, we will exhibit a transformation, called AG, that takes an arbitrary logic program P as input, and produces as output, an agent program, and an empty set of action constraints and an empty set of integrity constraints. We will show that the (Herbrand) models of P are in a 1-1 correspondence with the feasible status sets of $\mathsf{AG}(P)$, if they are projected to their $\mathcal{P}$-parts.
- Second, we will show that the minimal Herbrand models of P are in a 1-1 correspondence with the rational status sets of $\mathsf{AG}(P)$. This automatically implies, by results of Marek and Subrahmanian [77], the existence of a 1-1 correspondence between supported models of P, rational status sets of $\mathsf{AG}(P)$, weak extensions of a default theory associated with P as defined by [77], and expansions of an auto-epistemic theory associated with P [77]. Similar equivalences also exist between rational status sets and disjunctive logic programs [71].
- Third, we show that the stable models of P are in a 1-1 correspondence with the reasonable status sets of $\mathsf{AG}(P)$. As a consequence of known results due to Marek and Truszczyński [78], it follows immediately that there is a 1-1 correspondence between reasonable status sets and extensions of default logic theories associated with P.

Throughout this section, we assume the reader is familiar with standard logic program terminology as described by Lloyd [70] and nonmonotonic logic programming terminology [78].

9.1 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

$$\mathbf{P}(a) \leftarrow \mathbf{P}(b_1), \cdots, \mathbf{P}(b_m), \neg\mathbf{P}(c_1), \ldots, \neg\mathbf{P}(c_n) \tag{6}$$

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 9.1 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:

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

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 $F_{10}, \ldots, F_{17}$ 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. $\Box$

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

Proposition 9.1 *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.*

Proof. (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

$$a \quad \leftarrow \quad b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_n.$$

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

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

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. ∎

9.2 Rational Status Sets and Minimal Models of Logic Programs

If we return to Example 9.1, 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 9.1 and establishes this 1-1 correspondence.

Proposition 9.2 *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 [72], the above result implies that there exists a translation T (given in [72]) such that the rational status sets of $\mathsf{AG}(P)$ correspond exactly to the extensions of a pre-requisite free normal default theory $T(P)$.

9.3 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 9.1.

Example 9.2 It is easy to see that the logic program P of Example 9.1 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 9.3 below will show, this is not an accident. □

The following result explicitly states this.

Proposition 9.3 *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.*

Proof. 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 object 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

$$\mathbf{P}(a) \quad \leftarrow \quad \mathbf{P}(b_1), \ldots, \mathbf{P}(b_m)$$

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

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

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

$$a \quad \leftarrow \quad b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_m$$

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 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. Let $S^\star = \{a \mid \mathbf{P}(a) \in S\}$. It is straightforward to show that $S^\star$ 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 [77]. ∎

It is important to observe that by this correspondence, we obtain alternative proofs for the complexity results on reasonable status sets in the previous section. This is because the complexity results known for non-monotonic logic programs with stable model semantics [47, 48, 49] directly imply the above results.

9.4 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 non-monotonic reasoning, provide connections with well known nonmonotonic logics as well. For example, the following results are well known:

- Marek and Truszczynski [78] prove 1-1 correspondences between stable models of logic programs and extensions of default logic theories.
- Marek and Subrahmanian [77] and Marek and Truszczynski [78] prove 1-1 correspondences between stable models of logic programs and appropriate types of expansions of auto-epistemic theories.
- Lobo and Subrahmanian [72] prove 1-1 correspondences between minimal models of logic programs, and extensions of prerequisite-free normal default logic theories.
- Ben-Eliyahu and Dechter [15] 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.* [74] 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; recall that existence of a feasible as well as a rational status set for an agent program was shown to be NP-hard, even for agent programs without integrity constraints. 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.

10 Supply Chain Example, Revisited

In this section, we briefly revisit the supply chain example introduced towards the beginning of this paper, and see how this example works.

- The Plant Agent, located at a factor in Forth Worth, monitors the status of its inventory. For example, Figure 1 shows that the plant has in its inventory, 200 copies of Item1. Likewise, there are two supplier agents, associated with vendors who supply parts to the factory. These supply agents are located in Palo Alto and Washington, respectively. Their inventories show 500 and 300 copies of Item1, respectively. The two supplier agents and the plant agent are all built on top of Microsoft Access.
- When the inventory for the plant agent falls below a threshold, the plant agent initiates an attempt to procure the relevant parts. In the example, it attempts to procure Item1. It is important to note that dropping of inventory levels triggers this *action*.
- The first action taken by the Plant Agent is to send messages to the two supply agents. Figure 5 shows various messages that are interchanged during the evolution of the supply chain example. The reader should note that sending a message is an action. Determining what message to send is also an action. For example, the first two rows of Figure 5 shows the Plant Agent sending the message `QuantityRequest(Item1)` to both supplier agent 1 and 2.
- The supplier agents, on receiving this message, take actions. They determine the amount of Item1 they have in stock by executing a query on the Access database, and send a response such as `QuantityAvailable(SupplierAgent1,Item1,500)`.
- Once the Plant Agent has received affirmative responses from the Supply Agents, it contacts the Shipping Agent in order to schedule shipment of the items.
- The shipping agent comprises the flight agent and the truck agent. The task of the flight agent is to determine if there is an airport "near" the origin and destination of the shipment. In our implementation, "near" is defined as a range query on top of ESRI's MapObject geographic information system. If the origin and the destination are very "near" then we merely truck the shipment. Otherwise, if an airport exists "near" both the origin and the destination, then we merely truck it to the airport, fly it to an airport near the destination, and truck it from there to

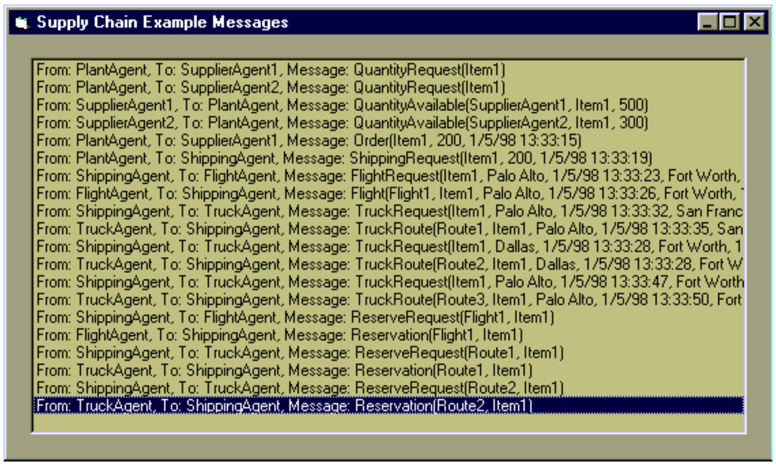


Figure 5: Messages Exchanged by Agents in Supply Chain Example

the destination. Otherwise, the shipping agent decides to truck it all the way from the origin to the destination. (Note that more sophisticated shipping strategies are possible, but in fact, this simple strategy is the one used by some of the large transportation logistics companies).

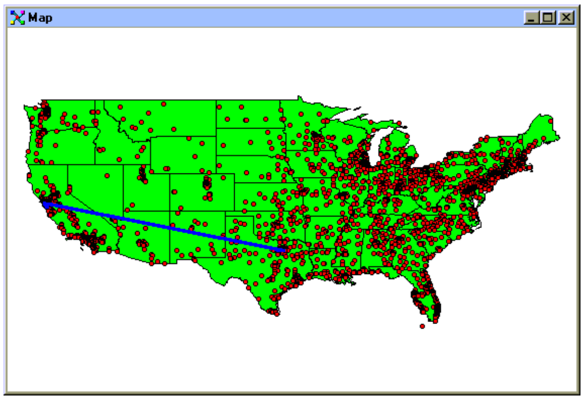
- Figure 5 shows several messages exchanged between the shipping agent, the flight agent, and the truck agent, coordinating this task.

- At the end of this process, the shipping agent produces a simple map, visualizing the shipment of the items from the Supplier location to the destination. Figure 6(a),(b),(c) shows such a visualization. In particular, Figure 6(a) shows the original flight path from San Francisco to Dallas. Figure 6(b) shows how our demonstration allows zooming, so that the truck route from Palo Alto to San Francisco airport is visualized, and Figure 6(c) shows the truck route from Dallas to the final destination, Forth Worth. This visualization is produced by the shipping agent by invoking appropriate code calls in the ESRI MapObjects system.

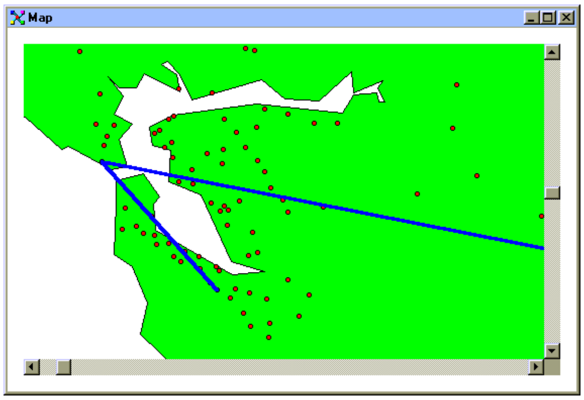
11 Related Work

During the last few years, there has been an explosion in the area of agent based research. Of this plethora of research, the work reported in this paper is perhaps closest to that of the group in CWI, Amsterdam, working on deontic logics for agent based programming [55, 31, 56] Below, we review the work on agents in a variety of arenas.

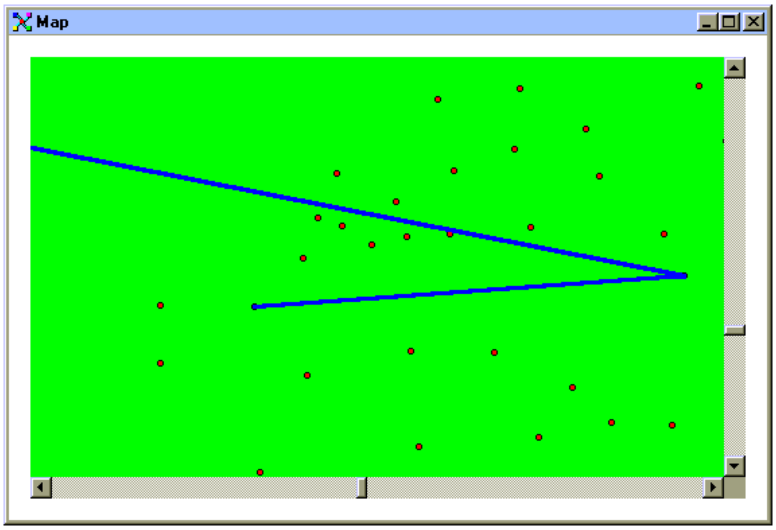
Agent Programming. Shoham [97] was perhaps the first to propose an explicit programming language for agents, based on object oriented concepts, and based on the concept of an agent state. In Shoham's approach, an "agent is an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments" [97]. He proposes a language, Agent-0, for agent programming, that provides a mechanism to express actions, time, and obligations. Agent-0 is a simple, yet powerful language. There are several differences between our approach and Shoham's. First, our language builds on top of arbitrary data structures, whereas Shoham's language is more or less logical (though it uses a LISP-like syntax). For us, states are instantiations of data structures



(a) Flight from San Francisco to Dallas



(b) Truck route from Palo Alto to San Francisco



(c) Truck route from Dallas to Forth Worth

Figure 6: Maps Produced by Shipping Agent in Supply Chain Example

managed by the program code associated with agents, while for Shoham, the agent state consists of beliefs, capabilities, choices and commitments. This allows Shoham to focus on reasoning about beliefs (e.g. agent A knows that agent B knows that agent A has no money), whereas our focus is on decision making on top of arbitrary data structures. Clearly both paradigms are needed for successfully building an agent.

Closely related to Shoham's work is that of Hindriks *et al.* [55] where an agent programming language based on BDI-agents is presented (BDI stands for "Belief, Desires, Intentionality"). They proceed upon the assumptions that an agent language must have the ability for updating beliefs, goals and for practical reasoning. (finding means to achieve goals). Hindriks *et al.* [55] argue that "Now, to program an agent is to specify its initial mental state, the semantics of the basic actions the agent can perform, and to write a set of practical reasoning rules" [55, p.211].

In our framework, as decision layers can (in principle) be embedded on top of arbitrary pieces of software code, representations of beliefs, goals, such as those developed by researchers in "reasoning about beliefs" can be easily incorporated as modules into those data structures, though we have not focused on this part. We do not *insist* that all agents have an a priori goal. For instance, consider an ACCESS database agent. This agent has no real goal that requires AI planning, unless one considers the fact that it should respond to user queries as a goal. Practical reasoning is achieved in our system because each agent processes an explicit call through a *method* used to process that call. Hindriks *et al.* [55] argue that "Now, to program an agent is to specify its initial mental state, the semantics of the basic actions the agent can perform, and to write a set of practical reasoning rules" [55, p.211]. In contrast to their framework, for us, an initial state is any set of (instantiated) data types – they assume this is a set of logical atoms. Likewise, practical reasoning rules for us are implemented as methods (or code-calls), but the decision about which of these actions is to be taken is represented through rules.

ConGolog [45] is a logic programming language for concurrent execution of actions. ConGolog creates static plans from a set of goals and primitive actions. ConGolog is built on the framework of Cohen and Levesque [25] who develop a logic of rational agents based intentionality using speech acts as a starting point. Their work has subsequently been used for a variety of other multiagent frameworks – we do not go into these extensions here, as they are not directly related to our effort.

In general, the approach in this paper builds upon the approaches of [97] and [55] in the following sense: first, we agree with these earlier approaches that the behavior of agents should be encoded through an agent program, and second, that actions taken by agents should modify agent states. However, we differ from these approaches in the following sense. First, our notion of an agent state is built on top of arbitrary data structures, rather than on top of states represented in logic. As a consequence, our approach complements the work of [97, 55] where they focus on logical representations of agent state, describing beliefs, capabilities, commitments, and goals. In addition, [97] describes temporal action scheduling which our language does not currently support, though ongoing work will extend it to do so [35]. If these modes of reasoning can be expressed as data structures, then the notion of agent proposed in our paper can benefit from the contributions in [97, 55]. Second, we propose a set of increasingly more satisfying declarative (epistemic) formal semantics for our work – [55] proposes an elegant proof theoretic operational semantics. Our semantics has the advantage of being neatly related to existing well-understood semantics for logic programs. Third, we analyze the tradeoffs between the adopting an epistemically satisfying semantics, and the computational complexity of these semantics. The complexity results also contain algorithmic approaches to computing these semantics.

Deontic Logic. In many applications (e.g. the tax application, and various legal applications), the administrators of an application have certain legal obligations (that is, they are required to take certain actions), as well as certain restrictions (that is, they are forbidden to take certain actions) if certain conditions are true. However, not all actions are either forbidden or obligatory. The vast majority of actions fall within a "gray" area – they are permitted, but neither obligatory or forbidden. To date, no active database system has provided a formal semantics for obligatory, permitted, and forbidden actions. In this paper, we have done so, building on top of classical deontic logic syntax [3, 79].

We have added to deontic logic as well in several ways: first, we have introduced the **Do** operator which standard deontic logic does not contain. Second, classical deontic logic does not account for interference between multiple actions (i.e., do actions α, β have mutually inconsistent effects, or can actions α, β be simultaneously executed), while our framework takes into account, both effects of actions, and provides different notions of concurrent executability. Third, our framework also allows nonmonotonic inference through the negation operator in rule bodies – this nonmonotonic negation operator does not occur in classical deontic logic model theory. The need for non-monotonic operators has been well argued by Reiter [86]. Last, but not least, the semantics of classical deontic logic is given in terms of a classical Hintikka-Kripke style model theory. Due to the introduction of the new features described above, and due to the fact that most deontic logic model theory leads to one or another deontic paradox, we chose to develop an alternative semantics that incorporates nonmonotonicity, concurrent actions, and the $\mathbf{Do}$ operator proposing the concepts of feasible, rational and reasonable status sets, and their variants, through which many desirable deontic desiderata (e.g. regimentation, relaxing obligations when the cannot be satisfied)) can be incorporated. Precisely how various other deontic assumptions can be captured within our semantics remains to be worked out.

The approach of Hindriks *et al.* [55] is based on such logics and has already been discussed earlier. Dignum and Conte [31] have used deontic logic extensively to develop methods for goal formation – in our framework, goal formation is one of several actions that an agent can take. Thus, we can specifically gain from the work of Dignum and Conte [31], through explicitly plugging-in such a framework as an action called `form-goals` implemented through the elegant work they report.

Agent Decision Making. There has been a significant amount of work on agent decision making. Rosenschein [88] 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 [89] extend this framework to provide a basis for such actions in terms of situated automata theory.

Bratman *et al.* [18] 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.* [105] 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 [92] 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 [98]. 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 [101] provide a coordinated search methodology for multiple agents. Haddadi [52] develops a declarative theory of interactions, as do Rao and Georgeff [84], and Coradeschi and Karlson [27] 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 [99]. 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 [93] 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 [99, 93].

Coalition formation mechanisms where agents dynamically team up with other agents has been intensely studied by many researchers [94, 91, 111]. Determining which agents to team with is a sort of decision making capability. Inverno *et al.* [58] 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 [68] develop the semantics of KQML, but do not explicitly present an action language.

Reasoning About Actions. Several works [42, 8, 9, 10] 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[42] 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 [9, 10, 6, 7]. The language A allows users to make statements of the form

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

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 [42, 9, 10, 6, 7].

Collaborative Problem Solving. There has also been extensive work on collaborative problem solving and negotiation in multiagent systems (e.g., [26, 60, 64, 87, 107]). 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 [57]. There have been numerous proposals for agentization in the literature (e.g., [38, 46, 16]) which have been broadly classified by Genesereth and Ketchpel [43] 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 [43], 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" [43, 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 [43] 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 [108], 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 [66] present the SHADE and COINS systems for matchmaking. Decker, Sycara, and Williamson [30] present matchmakers that store capability advertisements of different agents. Arisha *et al*. [5] present a theoretical foundation for matchmaking as well. 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 [29, 81] and relational DBMSs, object-oriented DBMSs and/or file systems [39, 63, 95]. 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 [19, 74, 74, 100, 76]. 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 nonlinear 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 [73, 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 [12] system, at MCC.

12 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 developed results showing the relationship between the declarative semantics and computational complexity. 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 currently developing a compiler for agent programs – in particular, in a future paper [35], 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 [35] 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 [19, 76, 100].

- 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 $\mathbf{Do}$-actions in S_t are executed, and the updates. We are developing algorithms for this task.

Acknowledgements

We wish to thank Alex Dekhtyar, Juergen Dix, Sarit Kraus, Munindar Singh, and Terrence Swift, for a very close reading 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, 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.

References

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

[2] K. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 10. Elsevier Science Publishers B.V. (North-Holland), 1990.

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

[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, Nov. 1997.

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

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

[8] C. Baral and M. Gelfond. Representing Concurrent Actions in Extended Logic Programming. In: *Proc. 13th Intl. Joint Conf. on Artificial Intelligence*, pp. 866-871, 1993.

[9] C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[10] C. Baral, M. Gelfond and A. Provetti. Representing Actions I: Laws, Observations, and Hypothesis. In: *Proc. AAAI '95 Spring Symposium on Extending Theories of Action*, 1995.

[11] C. Baral and J. Lobo. *Formal Characterization of Active Databases*. In: *Proc. Workshop of Logic on Databases (LID '96)*, D. Pedreschi and C. Zaniolo (eds), San Miniato, Italy, LNCS 1154, 1996.

[12] R. Bayardo, et. al. *Infosleuth: Agent-based Semantic Integration of Information in Open and Dynamic Environments*. In: *Proc. ACM SIGMOD Conf. on Management of Data*, 1997.

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

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

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

[16] 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: *Proc. AAAI Spring Symposium Series on Software Agent*, 1995.

[17] D.J. Bowersox, D.J. Closs and O.K. Helferich. *Logistical Management: A Systems Integration of Physical Distribution, Manufacturing Support, and Materials Procurement*. 3rd ed., Macmillan, New York, 1986.

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

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

[20] M. Cadoli, *The Complexity of Model Checking for Circumscriptive Formulae*. Information Processing Letters, 44:113–118, 1992.

[21] R.G.G. Cattell *et al.* (eds.) *The Object Database Standard: ODMG-97*. 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: *Proc. IPSJ Conference, Tokyo, Japan*, October 1994. (Also available via anonymous FTP from host db.stanford.edu, file /pub/chawathe/1994/tsimmis-overview.ps.)

[23] Z.-Z. Chen and S. Toda. *The Complexity of Selecting Maximal Solutions*. In: *Proc. 8th IEEE Structure in Complexity Theory Conference*, pp. 313–325, 1993.

[24] Z.-Z. Chen and S. Toda. *The Complexity of Selecting Maximal Solutions*. Information and Computation, 119:231–239, 1995.

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

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

[27] S. Coradeschi and L. Karlsson. *A Behavior-based Decision Mechanism for Agents Coordinating using Roles*. In: *Proc. 1997 Intl. Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp 100-105, 1997.

[28] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. *Complexity and Expressive Power of Logic Programming*. In: *Proc. Twelfth IEEE International Conference on Computational Complexity (CCC '97)*, pp 82–101, 1997.

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

[30] K. Decker, K. Sycara and M. Williamson. *Middle Agents for the Internet*. In: *Proc. IJCAI 97, Nagoya, Japan,* pp 578–583, 1997.

[31] F. Dignum and R. Conte. *Intentional Agents and Goal Formation*, In: *Proc. 1997 Intl. Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp 219–231, 1997.

[32] T. Eiter and G. Gottlob. *The Complexity of Logic-Based Abduction*. Journal of the ACM, 42(1):3–42, 1995.

[33] T. Eiter, G. Gottlob, and N. Leone. *On the Indiscernibility of Individuals in Logic Programming*. Journal of Logic and Computation, 7(6):805–824, 1997.

[34] T. Eiter, G. Gottlob, and H. Mannila. *Disjunctive Datalog*. ACM Transactions on Database Systems, 22(3):315–363, 1997.

[35] T. Eiter, V.S. Subrahmanian, and M. Tikir. *Regular Agent Programs and their Implementation*, in preparation.

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

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

[38] L. Gasser and T. Ishida. *A Dynamic Organizational Architecture For Adaptive Problem Solving*. In: *Proc. AAAI '91*, pp 185–190, 1991.

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

[40] 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.

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

[42] M. Gelfond and V. Lifschitz. *Representing Actions and Change by Logic Programs*. Journal of Logic Programming, 17(2-4):301–323, 1993.

[43] M.R. Genesereth and S.P. Ketchpel. *Software Agents*. Communications of the ACM, 37(7), 1994.

[44] M.R. Genesereth and N.J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufman Pub., 1987.

[45] G. De Giacomo, Y. Lesperance and H.J. Levesque. *Reasoning about Concurrent Execution, Prioritized Interrupts, and Exogenous Actions in the Situation Calculus*. In: *Proc. IJCAI 97, Nagoya, Japan,* 1997.

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

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

[48] G. Gottlob. *The Complexity of Default Reasoning Under the Stationary Fixed Point Semantics*. Information and Computation, 121(1):81-92, 1995.

[49] G. Gottlob. *Translating default logic into standard autoepistemic logic*, Journal of the ACM, 42(4):711–740, 1995.

[50] G. Gottlob, N. Leone, and H. Veith. Second-Order Logic and the Weak Exponential Hierarchies. In: *Proc. 20th Conference on Mathematical Foundations of Computer Science (MFCS '95)*, LNCS 969, pages 66–81, 1995. Full paper CD-TR 95/80, Christian Doppler Lab for Expert Systems, TU Wien.

[51] A. Gupta and I.S. Mumick (eds). *Materialized Views*. MIT Press, 1998, to appear.

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

[53] J.Y. Halpern and M.Y. Vardi. *Model Checking vs. Theorem Proving: A Manifesto*. In: *Proc. Intl Conf. on Knowledge Representation and Reasoning (KR 91)*, pp 325–334, 1991.

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

[55] K.V. Hindriks, F.S. de Boer, W. van der Hoek and J.-J.Ch. Meyer. *Formal Semantics of an Abstract Agent Programming Language*. In: *Proc. 1997 Intl. Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp 204–218, 1997.

[56] W.van der Hoek, B. van Linder and J.-J.Ch.Meyer. *A Logic of Capabilities*. In: *Proc. 3rd Intl Symposium on the Logical Foundations of Computer Science (LFCS 94)*, A. Nerode and Y.V. Matiyasevich (eds), pp 366-378, Springer Verlag, 1994.

[57] M. Huhns and M. Singh (eds). *Readings in Agents*. Morgan Kaufmann Press, 1997.

[58] M. d'Inverno, D. Kinny, M. Luck and M. Wooldridge. *A Formal Specification of dMARS*. In: *Proc. 1997 Intl. Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp 146-166, 1997.

[59] B. Jenner and J. Toran. *The Complexity of Obtaining Solutions for Problems in NP and NL*. In: A. Selman (ed), Complexity Theory: A Retrospective II, to appear.

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

[61] D. S. Johnson. A Catalog of Complexity Classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. A, chapter 2. 1990.

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

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

[64] S. Kraus. *Negotiation and Cooperation in Multi-Agent Environments*. Artificial Intelligence, Special Issue on Economic Principles of Multi-Agent Systems. 94(1-2):79-98, 1997.

[65] C. Krogh. Obligations in Multi-Agent Systems. In: Aamodt, Agnar, and Komorowski (eds), *Proc. Fifth Scandinavian Conference on Artificial Intelligence (SCAI '95)*, pp 19–30, Trondheim, Norway, 1995. ISO Press.

[66] D. Kuokka and L. Harada. *Integrating Information via Matchmaking*. Journal of Intelligent Informations Systems, 6(2/3):261–279, 1996.

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

[68] Y. Labrou and T. Finin. *Semantics for an Agent Communication Language*, In: *Proc. 1997 Intl. Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp 199–203, 1997.

[69] P. Liberatore and M. Schaerf. *The Complexity of Model Checking for Belief Revision and Update*. Ib: *Proc. AAAI-96*, pp 556–561, 1996.

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

[71] J. Lobo, J. Minker and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. MIT Press, 1992.

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

[73] J. Lu, G. Moerkotte, J. Schue, and V.S. Subrahmanian. *Efficient Maintenance of Materialized Mediated Views*. In: *Proc. 1995 ACM SIGMOD Conf. on Management of Data*, San Jose, CA, May 1995.

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

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

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

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

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

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

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

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

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

[83] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[84] A.S. Rao and M. Georgeff. *Modeling Rational Agents within a BDI-Architecture*. In: *Proc. Intl Conf. on Knowledge Representation and Reasoning (KR 91)*, pp 473–484, 1991.

[85] R. Reiter. *On Closed-World Databases*. In: H. Gallaire and J. Minker (eds.), *Logic and Data Bases*, pp 55–76. Plenum Press, New York, 1978.

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

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

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

[89] S.J. Rosenschein and L.P. Kaelbling. *A Situated View of Representation and Control*. Artificial Intelligence, 73, 1995.

[90] S.J. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[91] T. Sandholm and V. Lesser. *Coalition Formation Amongst Bounded Rational Agents*. In: *Proc. IJCAI 1995*, pp 662–669, Montreal, Canada, 1995.

[92] M. Schoppers and D. Shapiro. *Designing Embedded Agents to Optimize End-User Objectives*. In: *Proc. 1997 Intl Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp 2–12, 1997.

[93] R. Schwartz and S. Kraus. *Bidding Mechanisms for Data Allocation in Multi-Agent Environments*, In : *Proc. 1997 Intl. Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp 56–70, 1997.

[94] O. Shehory, K. Sycara and S. Jha. *Multi-Agent Coordination through Coalition Formation*. In: *Proc. 1997 Intl. Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp 135–146, 1997.

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

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

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

[98] M.P. Singh. *A Customizable Coordination Service for Autonomous Agents*. In: *Proc. 1997 Intl Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp 86–99, 1997.

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

[100] V.S. Subrahmanian. *Amalgamating Knowledge Bases*. ACM Transactions on Database Systems, 19(2):291–331, 1994.

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

[102] J.D. Ullman *Principles of Database and Knowledge-base Systems*. Computer Science Press, 1989.

[103] R. van der Meyden. *The Dynamic Logic of Permission*. In: *Proc. Fifth Annual IEEE Symposium on Logic in Computer Science (LICS '90)*, pages 72–78, 1990.

[104] M. Vardi. Complexity of Relational Query Languages. In: *Proc. 14th ACM Symposium on the Theory of Computing (STOC '82)*, pp 137–146, San Francisco, 1982.

[105] E. Verharen, F. Dignum and S. Bos. *Implementation of a Cooperative Agent Architecture Based on the Language-Action Perspective*, In: *Proc. 1997 Intl Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp 26–39, 1997.

[106] K. Wagner. *Bounded Query Classes*. SIAM Journal of Computing, 19(5):833–846, 1990.

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

[108] G. Wiederhold. *Intelligent Integration of Information*. In: *Proc. 1993 ACM SIGMOD Conf. on Management of Data*, pp 434–437, 1993.

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

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

[111] M. Wooldridge and N.R. Jennings. *Formalizing the Cooperative Problem Solving Process*, In: M. Huhns and M. Singh (eds.), *Readings in Agents*, pp 430–440. Morgan Kaufmann, 1997.

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

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 any object that may be involved in the state 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 relational database maintained under active domain semantics.

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 previous assumptions, 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 monotone 3SAT (M3SAT) [37] to the S-concurrent execution problem, for a setting where the software code $\mathcal{S}$ provides access to a relational database $\mathcal{DB}$ and 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 $x_1, \ldots, x_n$, such that each C_i is either positive or negative.

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; 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	$\mathtt{set_1}(X)$	VAL$(X,1)\wedge$ VAL$(X,0)$
	$\mathtt{set_0}(X)$	VAL$(X,1)\wedge$ VAL$(X,0)$
Checking	$\mathtt{check_{i,j}}$	SV$(x_1)\wedge\cdots\wedge$ SV$(x_n)\wedge At_{i,j}$
Success	sat	SAT$(c_1)\wedge\cdots\wedge$ SAT(c_m)
Clearance	clear	$\emptyset$

Phase	Action	Add Set	Delete Set
Init	init	$\emptyset$	{ SV(V), SAT(C), INIT }
Choice	$\mathtt{set_1}(X)$	{ SV(X) }	{ INIT, VAL$(X,0)$ }
	$\mathtt{set_0}(X)$	{ SV(X) }	{ INIT, VAL$(X,1)$ }
Checking	$\mathtt{check_{i,j}}$	{ 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 $At_{i,j} = VAL(x_k,1)$ if the j-th literal of clause C_i is x_k, and $At_{i,j} = VAL(x_k,0)$ 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 hene write simply $\alpha(\vec{X})\theta$ for this triple. Let the set AS be as follows:

$$\begin{aligned} AS &= \{\mathtt{set_1}(x_i),\mathtt{set_0}(x_i)\mid 1\leq i\leq n\}\cup\\ &\quad\ \{\mathtt{check_{i,j}}\mid 1\leq i\leq m, 1\leq j\leq 3\}\cup\{\mathtt{init},\mathtt{clear},\mathtt{sat}\}. \end{aligned}$$

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

- init must be executed first, i.e., $\alpha_{\pi(1)} = \mathtt{init}$, and clear must be executed as the last action.
- Clause checking can only be done after some choice action $\mathtt{set_v}(x_i)$, $v\in\{0,1\}$, has occurred, for every $i=1,\ldots,n$.
- In the choice phase, execution of at most one of the actions $\mathtt{set_1}(x_i)$, $\mathtt{set_0}(x_i)$ is possible, for every $i=1,\ldots,n$.
- Success is only possible, if for each clause C_i at least one action $\mathtt{check_{i,j}}$ has been executed.

- After the Success phase, first every remaining action $\mathtt{set_v}(x_i)$ can be executed, then all remaining actions $\mathtt{check_{i,j}}$ 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 previous assumptions, 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 $AS[\pi'] = \alpha_{\pi'(1)}, \ldots, \alpha_{\pi'(n)}$ not feasible, or $AS[\pi]$ and $AS[\pi']$ yield a different result. By our 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 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, consisting of at least one positive clause and at least one negative clause. Here, each clause is supposed to have three (not necessarily different) literals.

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: POS(V_1, V_2, V_3) and NEG(V_1, V_2, V_3), which serve for storing the positive and negative clauses of $I \cup I'$, respectively; VAL(Var, BV), which stores a truth value assignment to the variables, s.t. variable x_i is true if $(x_i, 1) \in$ VAL, and x_i is false if $(x_i, 0) \in$ 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 $\mathtt{switch}(X, Y)$ and `eval`, where

$$
\begin{array}{ll}
\mathtt{switch:} & Pre(\mathtt{switch}(X,Y)) = \emptyset, \\
& Add(\mathtt{switch}(X,Y)) = \{\ \mathrm{VAL}(X,1), \mathrm{VAL}(Y,0)\}, \\
& Del(\mathtt{switch}(X,Y)) = \{\mathrm{VAL}(X,0), \mathrm{VAL}(X,1)\};
\end{array}
$$

$$
\begin{array}{ll}
\mathtt{eval:} & Pre(\mathtt{eval}) = \{\ \exists V_1, V_2, V_3. \mathrm{POS}(V_1,V_2,V_3) \wedge \mathrm{VAL}(V_1,0) \wedge \mathrm{VAL}(V_2,0) \wedge \mathrm{VAL}(V_3,0) \wedge \\
& \qquad\qquad \mathrm{NEG}(V_1,V_2,V_3) \wedge \mathrm{VAL}(V_1,1) \wedge \mathrm{VAL}(V_2,1) \wedge \mathrm{VAL}(V_3,1)\ \}, \\
& Add(\mathtt{eval}) = \{\ \mathrm{UNSAT}\ \}, \\
& Del(\mathtt{eval}) = \emptyset.
\end{array}
$$

Observe that like in the proof of Theorem A.1, all unbound variables in preconditions of actions are action parameters; we thus write analogously $(\alpha(\vec{X})\theta)$ for $(\alpha(\vec{X}), \theta, \gamma)$ where γ is void.

The set AS of execution triples is

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

Intuitively, a switch action $\mathtt{switch}(x_i, x_{n+i})$ flips the value of x_i from 0 to 1 and the value from x_{n+i} from 1 to 0. The 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 $\alpha_{\pi(i)} = \mathtt{eval}$ flip the values of some variables; notice that flipping x_i is simultaneously done with flipping x_{n+i}. The precondition $Pre(\mathtt{eval})$ 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 $Pre(\mathtt{eval})$ 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.

Remark. We can derive AS from a simple fixed program, if we store the pairs x_i, x_{n+i} in a separate relation. result extends to the data complexity of an tion set. ∎

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: QBF

Lemma 8.14 *Let $\Phi' = \exists Y' \forall X' \phi'$ be a QBF such that ϕ' is in DNF. Then, a formula $\Phi = \exists Y \forall X \phi$, where ϕ is in M3DNF (see proof of Theorem 8.2 for M3DNF) can be constructed in polynomial time, such that*

(1) *for* $Y = \emptyset$*, the formula* $\forall X\,\phi[Y = \emptyset]$ *is true;*

(2) $\Phi' \longleftrightarrow (\exists Y \neq \emptyset)(\forall X)\phi$ *holds.*

(As a remark to the interested reader, $\exists Y \neq \emptyset$ is, strictly speaking, a second-order generalized quantifier.)

Proof. Without loss of generality, Φ' is already monotone. Suppose $Y' = \{y_1, y_2, \ldots, y_n\}$ and let $Y = Y' \cup \{y_0\}$. Consider the formula

$$\exists Y' \neq \emptyset \forall Y.((\neg y_1 \wedge \cdots \wedge \neg y_n) \vee \phi). \tag{7}$$

This formula is clearly equivalent to Φ'. Construct next the formula

$$\begin{gathered}\exists Y' \forall X[\forall Z.(\neg y_0 \wedge z_1) \vee (\neg z_1 \wedge \neg y_1 \wedge z_2) \vee (\neg z_2 \wedge \neg y_2 \wedge z_3) \vee \\ (\neg z_{n-1} \wedge y_{n-1} \wedge z_n) \vee (\neg z_n \wedge \neg y_n)] \vee \phi,\end{gathered} \tag{8}$$

where $Z = \{z_1, \ldots, z_n\}$. The formula (8) is equivalent to (7). Indeed, observe that the subformula $[\forall Z. \ldots]$ of (8), denoted by $\forall Z.\psi$, is equivalent to $\neg y_1 \wedge \cdots \wedge \neg y_n$. To see this, suppose first $\forall Z.\psi$ is true. Then, for every $i = 0, \ldots, n$ consider a value assignment χ_i to Z such that $z_j = 0$, for every $j < i$, and $z_j = 1$, for all $j \geq i$. Then, $\psi[Z] \leftrightarrow \neg x_i$. Hence, the only-if direction holds. Suppose now that $\neg y_0 \wedge \cdots \wedge \neg y_n$ is true. Towards a contradiction, suppose $\psi[Z]$ is false for some value assignment χ to Z. Hence, by the first disjunct in ψ z_1 is false in χ, which means by the second that z_2 is false in χ, …, that z_n is false in χ. However, the last disjunct in ψ is $\neg z_n \wedge \neg x_n$. Thus, this disjunct is true, which is a contradiction.

By elementary quantifier pulling, formula (8) is equivalent to the formula

$$\exists Y \neq \emptyset \forall X Z(\psi \vee \phi), \tag{9}$$

and $\forall X Z(\psi \vee \phi)[Y = \emptyset]$ is true. By using further universally quantified variables $Z' = \{z'_1, \ldots, z'_n\}$, we obtain

$$\exists Y \neq \emptyset \forall Y Z Z'[\bigvee_i (z_i \wedge z_i \wedge z'_i) \vee \bigvee_i (\neg z_i \wedge \neg z_i \wedge \neg z'_i) \vee \psi[Z/\neg Z] \vee \phi] \tag{10}$$

where $\phi[Z/\neg Z']$ means the obvious substitution of Z' literals for Z literals such that the formula is monotone (observe that z_i resp. $\neg z_i$ are for convenience replicated in disjuncts). Clearly, (10) is equivalent to (9). This proves the lemma. ∎

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. 3.1 beginning	Variable symbols ranging over τ
$\mathtt{X.f_i.g}$	Sec. 3.1 beginning	Path variable
$\mathcal{S}:\mathtt{f}(\mathtt{d_1},\ldots,\mathtt{d_n})$	3.2	Code call
`cc`	3.2	Code call
`in(X,cc)`	3.2	Code call atom
$<,>,\leq,\geq,=,\&$	3.3	Code call condition operators
χ	3.3	Code call condition
$Sol(\chi)_{\mathcal{T}_\mathcal{S},\mathcal{O}_\mathcal{S}}$	3.5	Set of code call solutions
$\mathcal{O}\_Sol(\chi)_{\mathcal{T}_\mathcal{S},\mathcal{O}_\mathcal{S}}$	3.5	Set of all objects in a code call solution
$\mathsf{ins}_\mathcal{S}$	After 3.5	Inserts objects in a state
$\mathsf{del}_\mathcal{S}$	After 3.5	Deletes objects from a state
$\psi \Rightarrow \chi_a$	3.6	Integrity constraint
$\mathcal{O}_\mathcal{S} \models IC$	3.7	Integrity constraint satisfaction
$\mathcal{IC}$	3.7	Finite collection of integrity constraints
α	4.1	Action
$(\tau_1,\ldots,\tau_n)$	4.1	Action schema
$Pre(\alpha)$	4.1	Precondition for action α
$Add(\alpha)$	4.1	Add list for action α
$Del(\alpha)$	4.1	Delete list for action α
(θ,γ)-Executability	4.3	Action execution under substitutions
$\Theta\Gamma(\alpha(\vec{X}),\mathcal{O}_\mathcal{S})$	4.3	Set of all θ,γ making an action $\alpha(\vec{X})$ executable
$apply(A,\mathcal{O}_\mathcal{S})$	4.5	Weak-concurrent execution
S-concurrently executable	4.6	Sequential-concurrent execution
F-concurrently executable	4.7	Full-concurrent execution
AC	4.8	Action constraint
$S,\mathcal{O}_\mathcal{S} \models AC$	4.9	Action constraint satisfaction
$\mathbf{P}\alpha$	4.10	Agent is permitted to take action α
$\mathbf{F}\alpha$	4.10	Agent is forbidden to take action α
$\mathbf{O}\alpha$	4.10	Agent is obliged to take action α
$\mathbf{W}\alpha$	4.10	Obligation to take action α is waived
$\mathbf{Do}\alpha$	4.10	Agent does take action α
$A \leftarrow L_1,\ldots,L_n$	3	Action rule
$\mathcal{P}$	4.12	Agent program
(continued next page)		

Notation	Location	Description
$H(r)$	para before Ex. 4.7	Head of rule r
$B(r)$	para before Ex. 4.7	Body of rule r
$B^-(r)$	para before Ex. 4.7	Negative literals in the body of rule r
$B^+(r)$	para before Ex. 4.7	Positive literals in the body of rule r
$\neg.B^-(r)$	para before Ex. 4.7	Atoms of the negative literals of rule r
$B(r)_{as}, B(r)_{cc}$	para before Ex. 4.7	Body of rule r restricted to action status atoms
$B(r)_{cc}$	para before Ex. 4.7	Body of rule r restricted to code call atoms
$B^-(r)_{as}$	para before Ex. 4.7	Negative literals in the body of rule r restricted to action status atoms
$B^-(r)_{cc}$	para before Ex. 4.7	Negative literals in the body of rule r restricted to code call atoms
$B^+(r)_{as}$	para before Ex. 4.7	Positive literals in the body of rule r restricted to action status atoms
$B^+(r)_{cc}$	para before Ex. 4.7	Positive literals in the body of rule r restricted to code call atoms
$\neg.B^-(r)_{as}$	para before Ex. 4.7	Negations of atoms in $B^-(r)_{as}$
$\neg.B^-(r)_{cc}$	para before Ex. 4.7	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