# Heterogeneous Agent Systems

**V.S. Subrahmanian**
University of Maryland

**Piero Bonatti**
Universita di Milano

**Jürgen Dix**
Universität Koblenz-Landau

**Thomas Eiter**
Technische Universität Wien

**Sarit Kraus**
Bar-Ilan University

**Fatma Özcan**
University of Maryland

**Robert Ross**
University of Maryland

**This is only a draft. Please have a look at**
**`http://mitpress.edu/book-home.tcl?isbn=0262194244`.**
**The book will come out in early 2000.**

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In past decades, software developers created massive, monolithic software programs that often performed a wide variety of tasks. During the past few years, however, there has been a shift from the development of massive programs containing millions of lines of code, to smaller, *modular*, pieces of code, where each module performs a well defined, focused task (or a small set of tasks), rather than thousands of different tasks, as used to be the case with old legacy systems. Software agents are the latest innovation in this trend towards splitting complex software systems into components. Roughly speaking, a software agent is a body of software that:

- provides one or more useful *services* that other agents may use under specified conditions,
- includes a *description* of the services offered by the software, which may be accessed and understood by other agents,
- includes the ability to *act autonomously* without requiring explicit direction from a human being,
- includes the ability to succinctly and declaratively describe how an agent determines *what actions to take* even though this description may be kept hidden from other agents,
- includes the ability to *interact* with other agents—including humans—either in a cooperative, or in an adversarial manner, as appropriate.

Note that not all software agents have to have the above properties—however any software agent programming paradigm must have the ability to create agents with some or all of these properties. In addition, agents will have a variety of other properties not covered in the above list, which will be spelled out in full technical detail as we go through this book.

With the proliferation of the Internet, there is now a huge body of data stored in a vast array of diverse, heterogeneous data sources, which is directly accessible to anyone with a network connection. This has led to the need for several agent based capabilities.

**Data Integration Agents:** Techniques to *mix and match*, *query*, *manipulate*, and *merge* such data together have gained increasing attention. Agents that can access heterogeneous data sources, and mix and match such data are increasingly important. Several agent based techniques for such data integration have been developed (Bayardo, R., et al. 1997; Arens, Chee, Hsu, and Knoblock 1993; Brink, Marcus, and Subrahmanian 1995; Lu, Nerode, and Subrahmanian 1996; Chawathe, S., et al. 1994).

**Mobile Agents:** The rapid evolution of the Java programming language (Horstmann and Cornell 1997) and the ability of Java applets to "move" across the network, executing bytecode at remote sites, has led to a new class of "mobile" agents (Rus, Gray, and Kotz 1997; Lande and Osjima 1998; Vigna 1998b; White 1997). If such agents are to autonomously form teams with other agents to cooperatively solve a problem, it is necessary that various techniques will be needed, such as techniques for describing agent services, for comprehending agent services, and for indexing and retrieving agent services, as well as techniques to facilitate interoperability between multiple agents.

**Software Interoperability Agents:** As the number of Java *applets* and other freely available and usable software deployed on the web increases, the ability to pipe data from one data source directly into one of these programs, and pipe the result into yet another program becomes more and more important. There is a growing body of research on agents that facilitate software interoperability (Patil, Fikes, Patel-Schneider, McKay, Finin, Gruber, and Neches 1997).

**Personalized Visualization:** Some years ago, the Internet was dominated by computer scientists. That situation has experienced a dramatic change and over the years, the vast majority of Internet users will view the Internet as a tool that supports their interests, which, in most cases, will not be computational. This brings with it a need for visualization and presentation of the results of a computation. As the results of a computation may depend upon the interests of a user, different visualization techniques may be needed to best present these results to the user (Candan, Prabhakaran, and Subrahmanian 1996; Ishizaki 1997).

**Monitoring Interestingness:** As the body of network accessible data gets ever larger, the need to identify what is of interest to users increases. Users do not want to obtain data that is "boring" or not relevant to their interests. Over the years, programs to monitor user interests have been built—for example, (Goldberg, Nichols, Oki, and Terry 1992; Foltz and Dumais 1992; Sta 1993; Sheth and Maes 1993) presents systems for monitoring newspaper articles, and several intelligent mail-handlers prioritize user's email buffers. Techniques to identify user-dependent *interesting* data are growing increasingly important.

The above list merely provides a few simple examples of so-called *agent applications*. Yet, despite the growing interest in agents, and the growing deployment of programs that are billed as being "agents" several basic scientific questions have to be adequately answered.

***(Q1)*** *What is an agent?*
Intuitively, any definition of agenthood is a predicate, `isagent`, that takes as input a program $P$ in any programming language. Program $P$ is considered an agent *if, by definition,* `isagent`$(P)$ is true. Clearly, the `isagent` predicate may be defined in many different ways. For example, many of the proponents of Java believe that `isagent`$(P)$ is true *if and only if* $P$ is a Java program—a definition that some might consider restrictive.

***(Q2)*** *If program P is not considered to be an agent according to some specified definition of agenthood, is there a suite of tools that can help in "agentizing" P?*
Intuitively, if a definition of agenthood is mandated by a standards body, then it is reasonable for the designer of a program $P$ which does not comply with the definition of agenthood, to want tools that allow program $P$ to be reconfigured as an agent. Efforts towards a definition of agenthood include ongoing agent standardization activities such as those of *FIPA* (the Foundation for Intelligent Physical Agents).

*(Q3)* *What kind of software infrastructure, is required for multiple agents to interact with one another once a specific definition of agenthood is chosen, and what kinds of basic services should such an infrastructure provide?*
For example, suppose agents are programs that have (among other things) an associated service description language in which each agent is required to describe its services. Then, yellow pages facilities which an agent might access are needed when the agent needs to find another agent that provides a service that it requires. Such a yellow pages service is an example of an infrastructural service.

The above questions allow a multiplicity of answers. For every possible definition of agenthood, we will require different agentization tools and infrastructural capabilities. The main aim of this book is to study what properties *any* definition of agenthood should satisfy. In the course of this, we will specifically make the following contributions.

- We will provide a concrete definition of agenthood that satisfies the requirements alluded to above, and compare this with alternative possible definitions of agenthood;
- We will provide an architecture and algorithms for agentizing programs that are deemed not to be agents according to the given definition;
- We will provide an architecture and algorithms for creating and deploying software agents that respect the above definition;
- We will provide a description of the infrastructural requirements needed to support such agents, and the algorithms that make this possible.

The rest of this chapter is organized as follows. We will first provide three motivating example applications in Sections 1.1, 1.2, and 1.3, respectively. These three examples will each illustrate different features required of agent infrastructures and different capabilities required of individual agents. Furthermore, these examples will be revisited over and over throughout this entire book to illustrate basic concepts. In short, these examples form a common thread throughout this whole book. Later, in Section 1.4, we will a brief overview of existing research on software agents, and specify how these different existing paradigms address one or more of the basic questions raised by these three motivating examples. Section 1.4 will also explain what the shortcomings of these existing approaches are. In Section 1.5, we describe some general desiderata that agent theories and architectures should satisfy. Finally, in Section 1.6, we will provide a quick glimpse into the organization of this book, and provide a birdseye view of how (and where) the shortcomings pointed out in Section 1.4 are addressed by the framework described in the rest of this book.

## 1.1 A Personalized Department Store Application (STORE)

Let us consider the case of a large department store that has a web-based marketing site. Today, the Internet contains a whole host of such sites, offering on-line shopping services.

**Today's Department Store:** In most existing web sites today, interaction is initiated by a user who contacts the department store web site, and requests information on one or more consumer products he is interested in. For example, the user may ask for information on "leather shoes." The advanced systems deployed today access an underlying database and bring back relevant information on leather shoes. Such relevant information typically includes a picture of a shoe, a price, available colors and sizes, and perhaps a button that allows the user to place an order.

The electronic department store of today is characterized by two properties: first, it assumes that users will come to the department store, and second, it does nothing more than simply retrieving data from a database and displaying it to the user.

**Tomorrow's (Reactive) Department Store:** In contrast, the department store of tomorrow will take explicit actions so that the department store goes to the customer, announcing items deemed to be of interest to the customer, rather than waiting for the customer to come to the store. This is because the department store's ultimate goal is to maximize profit (current as well as future), and in particular, it will accomplish this through the following means: It would like to ensure that a customer who visits it is presented items that maximize its expected profit as well as the likelihood of making a sale (e.g., they may not want to lose a sale by getting too greedy.) In particular, the department store would like to ensure that the items it presents a user (whether she visited the site of her own volition, or whether the presentation is a directed mailing), are items that are likely to be of maximal interest to the user—there is no point in mailing information about $100-dollar ties to a person who has always bought clothing at lower prices.

Intelligent agent technology may be used to accomplish these goals through a simple architecture, as shown in Figure 1.1 on the facing page. This architecture involves the following agents:

1. **A Credit Database Agent:** This agent does nothing more sophisticated than providing access to a credit database. In the United States, many department stores issue their own credit cards, and as a consequence, they automatically have access to (at least some) credit data for many customers. The credit database agent may in fact access a variety of databases, not just one. Open source credit data is (unfortunately) readily available to paying customers.

2. **Product Database Agent:** This agent provides access to one or more product databases reflecting the merchandise that the department store sells. Given a desired product description (e.g., "leather shoes"), this agent may be used to retrieve tuples associated with this product description. For example, a department store may carry 100 different types of leather shoes, and in this case, the product database may return a list of 100 records, one associated with each type of leather shoe.

3. **A Profiling Agent:** This agent takes as input the identity of a user (who is interacting with the Department Store Interface agent described below). It then requests the credit database agent for information on this user's credit history, and analyses the credit data. Credit information typically contains detailed information about an individual's spending habits. The profiling agent may then classify the user as a "high" spender, an "average" spender, or a "low" spender. Of course, more detailed classifications are possible; it may classify the user as a "high" spender on clothing, but a "low" spender on appliances, indicating that the person cares more about personal appearance than on electrical appliances in his home.

   As we go through this book, we will see that the Profiling agent can be made much more complex—if a user's credit history is relatively small (as would be the case with someone who pays cash for most purchases), it could well be that the Profiling agent analyzes other information (e.g., the person's home address) to determine his profile and/or it might contact other agents outside the department store that sell profiles of customers.

4. **A Content Determination Agent:** This agent tries to determine what to show the user. It takes as input, the user's request, and the classification of the agent as determined by the Profiling

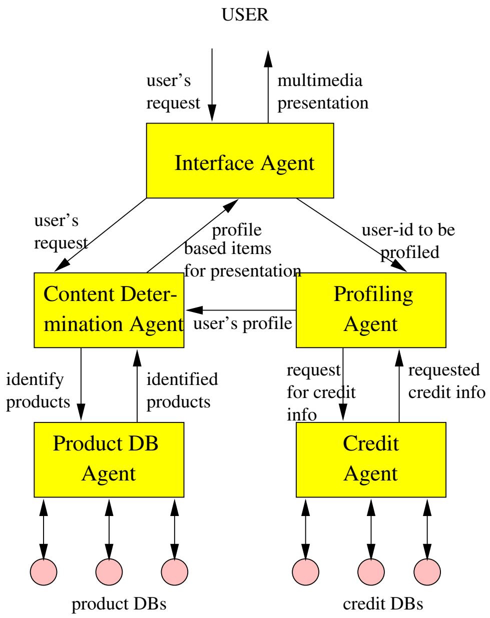



Figure 1.1: Interactions between Agents in STORE Example

Agent. It executes a query to the product database agent, which provides it a set of tuples (e.g., the 100 different types of leather shoes). It then uses the user classification provided by the profiling agent to filter these 100 leather shoes. For example, if the user is classified as a "high spender," it may select the 10 most expensive leather shoes. In addition, the content determination agent may decide that when it presents these 10 leather shoes to the user, it will run advertisements on the bottom of the screen, showing other items that "fit" this user's high-spending profile.

**5. Interface Agent:** This agent takes the objects identified by the Content Determination Agent and weaves together a multimedia presentation (perhaps accompanied with music to the user's taste if it has information on music CDs previously purchased by the user!) containing these objects, together with any focused advertising information.

Thus far, we have presented how a department store might deploy a multiagent system. However, a human user may wish to have a personalized agent that finds an online store that provides a given service. For example, one of the authors was recently interested in finding wine distributors who sell *1990 Chateau Tayac* wines. An agent that found such a distributor would have been invaluable. In addition to finding a list of such distributors, the user might want to have these distributors ranked in descending order of the per bottle sales—the scenario can be made even more complex by wanting to have distributors ranked in descending order of the total (cost plus shipping) price for a dozen bottles.

**Active Department Store of Tomorrow:** Thus far, we have assumed that our department store agent is *reactive*. However, in reality, a department store system could be *proactive* in the following sense. As we all know, department stores regularly have sales. When a sale occurs, the department store could have a *Sale-Notification Agent* that performs the following task. For every individual $I$ in the department store's database, the department store could:

- identify the user's profile,

- determine which items going on sale "fit" the user's profile, and
- take an appropriate action—such an action could email the user a list of items "fitting" his profile. Alternatively, the action may be to create a *personalized sale flyer* specifying for each user, a set of sale item descriptions to be physically mailed to him.

In addition, the Sale-Notification agent may *schedule future actions* based on its *uncertain beliefs about the users*. For example, statistical analysis of John Doe's shopping habits at the store may indicate the following distribution:

| **Day** | **Percentage Spent** |
|---|---|
| Monday | 2% |
| Tuesday | 3% |
| Wednesday | 3% |
| Thursday | 2% |
| Friday | 27% |
| Saturday | 50% |
| Sunday | 13% |

In the above table, the tuple, $\langle \texttt{Monday}, 2\% \rangle$ means that of all the money that John Doe is known to have spent at this store, 2% of the money was spent on Mondays.

The Sale-Notification agent may now reason as follows: 90% of John Doe's dollars spent at this store are spent during the Friday-Saturday-Sunday period. Therefore, I will mail John Doe promotional material on sales so as to reach him on Thursday evening.

However, there may be uncertainty in postal services. For example, the bulk mailing system provided by the US Postal Service may have statistical data showing that 13% of such mailings reach the customer within 1 day of shipping, 79% in 2 days, and and the remaining 8% take over 2 days. Thus, the Sale-Notification agent may mail the sales brochures to John Doe on Tuesday.

When we examine the above department store example, we notice that:

1. The department store example may be viewed as a multiagent system where the interactions between the agents involved are clear and well defined.

2. Each agent has an associated body of data structures and algorithms that it maintains. The content of these data structures may be updated independently of the application as a whole (e.g., user's credit data may change in the above example without affecting the Product-Database agent).

3. Each agent is capable of performing a small, but well defined set of actions/tasks.

4. The actual actions executed (from the set of actions an agent is capable of performing) may vary depending upon the circumstances involved. For example, the Credit agent may provide credit information in the above example only to the Profiling Agent, but may refuse to respond to credit requests from other agents.

5. Each agent may reason with beliefs about the behavior of other agents, and each agent not only decides what actions to perform, but also when to perform them. Uncertainty may be present in the beliefs the agent holds about other agents.

## 1.2 The Controlled Flight into Terrain Application (CFIT)

According to the *Washington Post (Feb. 12, 1998, page A-11)* 2,708 out of 7,496 airline fatalities during the 1987-1996 period did not happen due to pilot error (as is commonly suspected), but due to a phenomenon called *controlled flight into terrain* (CFIT). Intuitively, a CFIT error occurs when a plane is proceeding along an Auto-Pilot (not human) controlled trajectory, but literally crashes into the ground. CFIT errors occur because of malfunctioning sensors and because the autopilot program has an *incorrect belief* about the actual location of the plane. CFIT is the number one cause of airline deaths in the world. The CFIT problem is highlighted by two major plane crashes during recent years:

- The December 1995 crash of an American Airlines plane in Cali, Colombia, killing over 250 people including Paris Kanellakis, a prominent computer scientist;
- the crash of a US military plane near Dubrovnik, Yugoslavia in 1996, killing the US Commerce Secretary, Ron Brown.

We have developed a solution to the CFIT to develop a solution to the CFIT problem, and have developed a working prototype of a multi-agent solution to the CFIT problem. *BOEING Aerospace* has expressed interest in our solution. The solution involves the following agents:

**Auto-Pilot Agent:** The Auto-Pilot agent ensures that the plane stays on its allocated flight path. Most civilian flights in the world fly along certain prescribed flight corridors that are assigned to each flight by air traffic controllers. The task of the Auto-Pilot agent is to ensure that the plane stays on-course, and make appropriate adjustments (by perhaps using AI planning or 3-dimensional path planning techniques) when the physical dynamics of the plane cause it to veer off course. Techniques for agent based solutions to flight planning and air traffic control problems have been studied in the agents community by Tambe, Johnson, and Shen (1997).

**Satellite Agents:** We assume the existence of a set of satellite agents that will monitor the position of several planes simultaneously. Every $\Delta t$ units of time, each satellite agent broadcasts a report that may be read by the location agent. Thus, if $\Delta t = 10$ and the first report is read at time 0, then this means that all the satellite agents send reports at times $0, 10, 20, \ldots$ and so on. Each satellite agent specifies where it believes the plane is at that point in time.

**GPS Agent:** This agent takes reports from multiple satellite agents above and merges them together. Multiplexing satellite agents together enhances reliability—if one satellite agent fails, the others will still provide a report. Merging techniques may include methods of eliminating outliers—e.g., if 9 of 10 satellite agents tell the plane it is at location $A$ and the 10th agent tells the plane it is at location $B$, the last report can be eliminated. The GPS agent then feeds the GPS-based location of the plane to the Auto-Pilot agent, which consults the Terrain agent below before taking corrective action.

**Terrain Agent:** The Terrain agent takes a coordinate in the globe, and generates a terrain map for the region. In the case of our CFIT example, a special kind of terrain map is retrieved called a *Digital Terrain Elevation Data* (*DTED*) map. Our implementation currently includes DTED data for the whole of the continental USA, but not for the world. Given any $(x, y)$ location which falls within this map, the elevation of that $(x, y)$ location can then be retrieved from the DTED map by the Terrain agent. The Terrain agent provides to the Auto-Pilot agent a set of "no-go" areas. Using this set, the Auto-Pilot agent can check if its current heading will cause

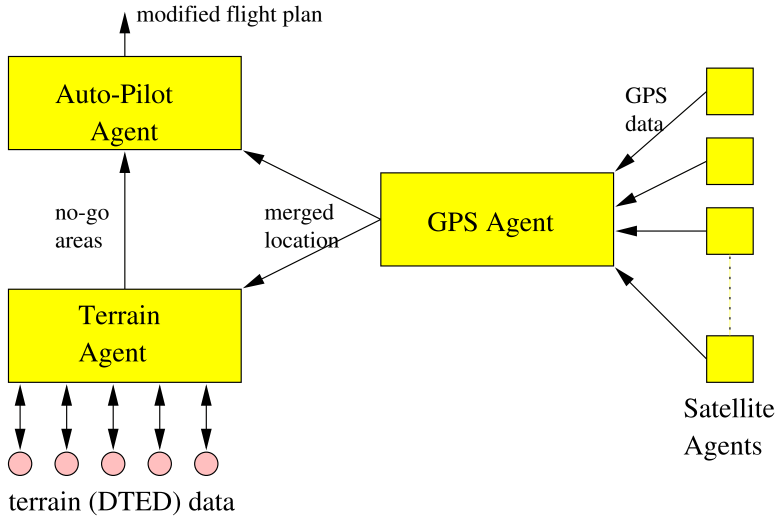



Figure 1.2: Interactions between Agents in CFIT Example

it to fly into a mountain (as happened with the American Airlines crash of 1996), and in such cases, it can replan to ensure that the plane avoids these no-go areas.

Figure 1.2 shows a schematic diagram of the different agents involved in this example.

The reader will readily note that there are some similarities, as well as some differences, between this CFIT example and the preceding STORE example. The example is similar to the department store example in the following ways:

- Like the STORE application, the CFIT application may be viewed as a multiagent system where the agents interact with one another in clearly defined ways.
- In both examples, each agent manages a well defined body of data structures and associated algorithms, but these data structures may be updated autonomously and vary from one agent to another.
- As in the case of the STORE example, each agent performs a set of well defined tasks.
- As in the case of the STORE example, agents may take different actions, based on the circumstances. For example, some satellite agents may send updates to one plane every 5 seconds, but only at every 50 seconds for another plane.

In addition, the following attributes (which also appear in the department store example) play an important role in the CFIT example:

**Reasoning about Beliefs:** The Auto-Pilot agent reasons with *Beliefs*. At any given point $t$ in time, the Auto-Pilot agent believes that it is at a given location $\ell_t$. However, its belief about its location, and the location it is really at, may be different. The task of the GPS agent in the CFIT application is to alert the Auto-Pilot agent to its incorrect beliefs, which may then be appropriately corrected by the Auto-Pilot agent.

In this example, the Auto-Pilot agent believes the correction it receives from the satellite agents. However, it is conceivable that if our plane is a military aircraft, then an enemy might attempt to masquerade as a legitimate satellite agent, and falsely inform the Auto-Pilot agent that it is at location $\ell_t^\star$, with the express intent of making the plane go off-course. However, agents must make decisions on how to act when requests/information are received from other agents. It is important to note that which actions an agent decides to execute depends upon background information that the agent has. Thus, if an agent suspects that a satellite agent message is not reliable, then it might choose to ignore information it receives from that agent or it may choose to seek clarification from another source. On the other hand, if it believes that the satellite agent's message is "legitimate," then it may take the information provided into consideration when making decisions. In general, agents decide how to act, based upon (i) the background knowledge that the agent has, and (ii) the beliefs that the agent currently holds.

**Delayed Actions:** Yet another difference with the STORE example is that the Auto-Pilot agent may choose to *delay* taking actions. In other words, the Auto-Pilot agent may know at time $t$ that it is off-course. It could choose to create a plan at time $t$ (creation of a plan is an explicit action) that commits the Auto-Pilot agent to take other actions at later points in time, e.g., "Execute a climb action by 50 feet per second between time $(t+5)$ and time $(t+10)$."

**Uncertainty:** If the Auto-Pilot agent receives frequent information from the Location agent, stating that it is off-course, it might suspect that some of its on-board sensors or actuators are malfunctioning. Depending upon its knowledge of these sensors and actuators, it might have different beliefs about which sensor/actuator is malfunctioning. This belief may be accompanied with a *probability* or *certainty* that the belief is in fact true. Based on these certainties, the Auto-Pilot may take one of several actions that could include returning the plane to manual control, switching off a sensor and/or switching on an alternative sensor. In general, in extended versions of our CFIT example, Auto-Pilot agents may need to reason with uncertainty when making decisions.

## 1.3 A Supply Chain Example (CHAIN)

Supply chain management (Bowersox, Closs, and Helferich 1986) is one of the most important activities in any major production company. Most such 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 products 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 with a group of 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. For pedagogical purposes, we have chosen to keep the scenario relatively simple.

The above automated purchasing procedure may be facilitated by using an architecture such as that shown in Figure 1.3 on the facing page. 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 has an associated agent that monitors two databases:

- An ACCESS database specifying how much uncommitted stock the supplier has. For example, if the tuple $\langle \texttt{widget50}, \texttt{9000} \rangle$ is in this relation, then this means that the supplier has 9000 pieces of widget50 that haven't yet been committed to a consumer.
- An ACCESS database specifying how much committed stock the supplier has. For example, if the triple (`widget50,1000,companyA`) is in the relation, this means that the supplier has 1000 pieces of widget50 that have been committed to company A.

Thus, if company-B were to request 2000 pieces of widget50, we would update the first relation, by replacing the tuple $\langle \texttt{widget50}, \texttt{9000} \rangle$ by the tuple $\langle \texttt{widget50}, \texttt{7000} \rangle$ and adding the tuple $(\texttt{widget50}, \texttt{2000}, \texttt{companyB})$ to the latter relation—assuming that company B did not already have widget50 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). 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.

As in the previous two examples, the Plant agent may make decisions based on a more sophisticated reasoning process. For example:

**Reasoning about Uncertainty:** The Plant agent may have some historical data about the ability of the supplier agents to deliver on time. For example, it may have a table of the form:

| Supplier | Item | Days Late | Percentage |
|---|---|---|---|
| supplier1 | widget1 | -3 | 5 |
| | | -1 | 10 |
| | | 0 | 55 |
| | | 1 | 20 |
| | | 2 | 10 |
| ... | ... | ... | ... |
| ... | ... | ... | ... |

In this table, the first tuple says that in cases where `supplier1` promised to deliver widget1, he supplied it 3 days early in 5% of the cases. The last entry above likewise says that when `supplier1` promised to deliver widget1, he supplied it 2 days late in 10% of the cases. Using this table, the Plant agent may make decisions about the probability that placing an order with supplier1 will in fact result in the order being delivered within the desired deadline.

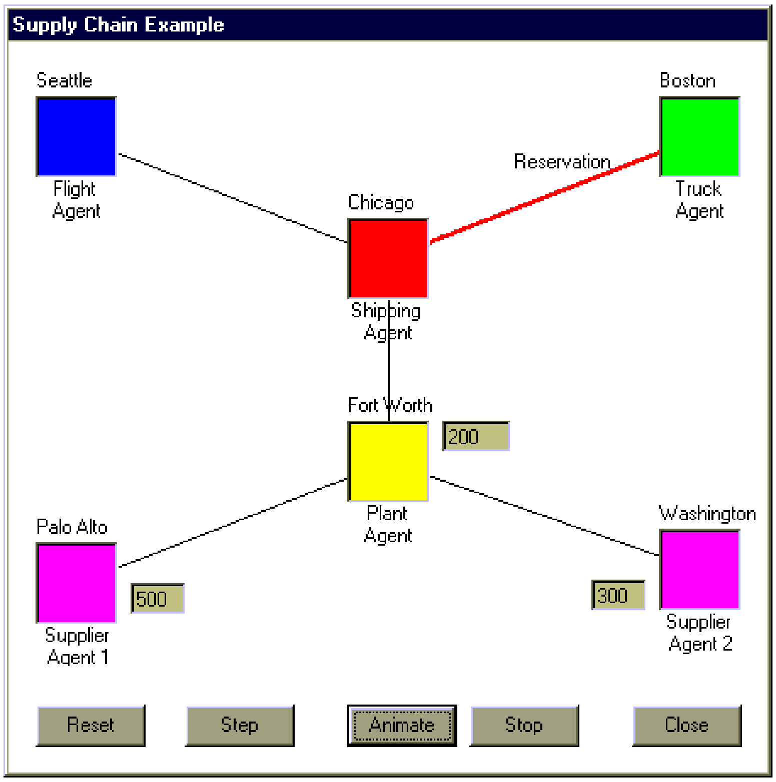



Figure 1.3: Agents in CHAIN Example

**Delayed Actions:** When placing an order with `supplier1`, the Plant agent `plant` may want to retain the option of cutting the contract to `supplier1` if adequate progress has not been made. Thus, the Plant agent may inform `supplier1` up front that 10 days after placement of the order, it will inspect the status of the supplier's performance on that order (such inspections will of course be based on reasonable and precisely stated evaluation conditions). If the performance does not meet certain conditions, it might cancel part of the contract.

**Reasoning about Beliefs:** As in the case of the CFIT agent, the Plant agent may make decisions based on its *beliefs* about the suppliers ability to deliver, or the transportation companies ability to ship products. For example, if the Plant agent believes that a Transportation agent is likely to have a strike, it might choose to place its transportation order with another company.

The CHAIN example, like the other examples, may be viewed as a multiagent system where the interactions between agents are clearly specified, and each agent manages a set of data structures that can be autonomously updated by the agent. Furthermore, different agents may manage different data structures.

However, a distinct difference occurs when the Plant agent realizes that neither of its Supplier agents can supply the item that is required within the given time frame. In such a case, the Plant agent may need to *dynamically find* another agent that supplies the desired item. This requires that the Plant agent has access to some kind of yellow pages facility that keeps track of the services offered by different agents. Later, in Chapters 2 and 3, we will define detailed yellow pages service mechanisms to support the need of finding agents that provide a service, when the identity of such agents is not known *a priori*.

## 1.4 Brief Overview of Related Research on Agents

In this section, we provide a brief overview of existing work on agents, and explain their advantages and disadvantages with respect to the three motivating examples introduced above.

As we have already observed, the three examples above all share a common structure:

- Each agent has an associated set of data structures.
- Each agent has an associated set of low-level operations to manipulate those data structures.
- Each agent has an associated set of high-level actions that "weave" together the low level operations above that it performs.
- Each agent has a policy that it uses to determine which of its associated high level actions to execute in response to requests and/or events (e.g., receipt of data from another agent).

There are various other parameters associated with any single agent that we will discuss in greater detail in later chapters, but for now, these are the most salient features of practical implemented agents. In addition, a platform to support multi-agent interactions must provide a set of common services including, but not limited to:

1. *Registration* services, through which an agent can register the services it provides.
2. *Yellow pages* services that allow an agent to find another agent offering a service *similar to* a service sought by the agent.
3. *Thesauri* and *dictionary* services that allow agents to determine what words mean.
4. More sophisticated *ontological* services that allow an agent to determine what another agent might mean when it uses a term or expression.
5. *Security* services that allow an agent to look up the security classification of another agent (perhaps under some restricted conditions).

Different parts of various technical problems raised by the need to create multiagent systems have been addressed in many different scientific communities, ranging from the database community, the AI community, the distributed objects community, and the programming languages community, to name a few. In this section, we will briefly skim some of the major approaches to these technical problems—a detailed and much more comprehensive overview is contained in Chapter 13.

### 1.4.1 Heterogeneous Data/Software Integration

One of the important aspects of agent systems is the ability to uniformly access heterogeneous data sources. In particular, if agent decision making is based on the content of arbitrary data structures managed by the agent, then there must be some unified way of accessing those data structures. Many formalisms have been proposed to integrate heterogeneous data structures. These formalisms fall into three categories:

**Logical Languages:** One of the first logical languages to integrate heterogeneous data sources was the *SIMS* system (Arens, Chee, Hsu, and Knoblock 1993) at USC which uses a LISP-like syntax to integrate multiple databases as well. More or less at the same time as *SIMS*, a Datalog-extension to access heterogeneous data sources was proposed in the *HERMES* Heterogeneous Reasoning and Mediator System Project in June 1993 (Lu, Nerode, and Subrahmanian 1996; Subrahmanian 1994; Brink, Marcus, and Subrahmanian 1995; Marcus and Subrahmanian 1996; Adali, Candan, Papakonstantinou, and Subrahmanian 1996; Lu, Moerkotte, Schue, and Subrahmanian 1995). Shortly thereafter, the IBM-Stanford *TSIMMIS* effort (Chawathe, S., et al. 1994) proposed logical extensions of Datalog as well. These approaches differed in their expressive power—for instance, *TSIMMIS* was largely successful on relational databases, but also accessed some non-relational data sources such as bibliographic data. *SIMS* accessed a wide variety of AI knowledge representation schemes, as well as traditional relational databases. 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.

**SQL Extensions:** SQL has long had a mechanism to make "foreign function" calls whereby an SQL query can embed a subquery to an external data source. The problem with most existing implementations of SQL is that even though they can access these external data sources, they make assumptions on the format of the outputs returned by such foreign function calls. Thus, if the foreign functions return answers that are not within certain prescribed formats, then they cannot be processed by standard SQL interpreters. Extensions of SQL to access heterogeneous relational databases such as the Object Database Connectivity (ODBC) standard (Creamer, Stegman, and Signore 1995) have received wide acceptance in industry.

**OQL Extensions:** Under the aegis of the US Department of Defense, a standard for data integration was proposed by a group of approximately 11 researchers selected by *DARPA* (including the first author of this book). The standard is well summarized in the report of this working group (Buneman, Ullman, Raschid, Abiteboul, Levy, Maier, Qian, Ramakrishnan, Subrahmanian, Tannen, and Zdonik 1996). The approach advocated by the *DARPA* working group was to built a *minimal core language* based on the Object Definition Language and the Object Query Language put forth earlier by the industry wide Object Data Management Group (*ODMG*) (Cattell, R. G. G., et al. 1997). The basic idea was that the core part be a restricted version of OQL, and all extensions to the core would handle complex data types with methods.

Another important later direction on mediation includes the *InfoSleuth* effort (Bayardo, R., et al. 1997) system, at MCC—this will be discussed in detail later in Chapter 4.

Implementations of all the three frameworks listed above were completed in the 1993-1996 time frame, and many of these are available, either free of charge or for a licensing fee (Brink, Marcus, and Subrahmanian 1995; Adali, Candan, Papakonstantinou, and Subrahmanian 1996; Lu, Nerode, and Subrahmanian 1996; Chawathe, S., et al. 1994; Arens, Chee, Hsu, and Knoblock 1993). *Any of the frameworks listed above could constitute a valid language, by using which access is provided to arbitrary data structures.*

### 1.4.2 Agent Decision Making

There has been a significant amount of work on agent decision making. Rosenschein (1985) 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."* Rosenschein and Kaelbling (1995) extend this framework to provide a basis for such actions in terms of situated automata theory. For example, in the case of the department store example, the Profiling Agent may use a rule of the form *"If the credit data on person P shows that she spends over $ 200 per month (on the average) at our store, then classify P as a high spender."* Using this rule, the Sales agent may take another action of the form *"If the Profiling agent classifies person P as a high spender, then send P material M by email."*

Bratman, Israel, and Pollack (1988) 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. This is useful when applied to agents which have intentions that might require planning (though there might be agents that do not have any intentions or plans such as a GPS receiver in the CFIT example). Certainly, the Auto-Pilot agent in the CFIT example has an *intention*—namely to stay on course, as specified by the flight plan filed by the plane, and it may need to *replan* when it is notified by the GPS agent that it has veered off course.

The *Procedural Reasoning System* (*PRS*) is one of the best known multiagent construction system that implements BDI agents (BDI stands for *Belief, Desires, Intentionality*) (d'Inverno, Kinny, Luck, and Wooldridge 1997). This framework has led to several interesting applications including a practical, deployed application called *OASIS* for air traffic control in Sydney, Australia. The theory of *PRS* is captured through a logic based development, in Rao and Georgeff (1991).

Singh (1997) 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. Agent states are viewed as states of the automaton, and agent actions are viewed as transitions on these states. This is certainly consistent with the three motivating examples—for instance, in the CHAIN example, when the `Supplier1` agent executes an action (such as shipping supplies), this may certainly be viewed as a state transition, causing the available quantity of the supply item in question at `Supplier1`'s location to drop.

### 1.4.3 Specific Interaction Mechanisms for Multiagent Systems

There has been extensive work in AI on specific protocols for multiagent interactions. Two such mechanisms are worth mentioning here:

**Bidding Mechanisms:** Let us return to the CHAIN example and assume that neither of the two approved suppliers (with existing contracts to funnel the purchase through) can deliver the supplies required by the Plant agent. In this case, the Plant agent needs to find another agent (one for which no contract is currently in force). The Plant agent needs to *negotiate* with the new agent, arriving at a mutually agreeable arrangement. There has been extensive work on negotiation in multiagent systems, based on the initial idea of contract nets, due to Smith and Davis (1983). 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 (1997) 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. Other forms of negotiation have also been studied and will be discussed in detail in Chapter 14.

**Coalition Formation:** A second kind of interaction between agents is coalition formation. Consider an expanded version of the CFIT example, in a military setting. Here, a Tank agent `tank` may have a mission, but as it proceeds toward execution of the mission, it encounters heavier resistance than expected. In this case, it may dynamically team with a helicopter gunship whose Auto-Pilot and control mechanisms are implemented using the CFIT example. Here, the tank is forming a *coalition dynamically* in order to accomplish a given goal. Coalition formation mechanisms where agents dynamically team up with other agents has been intensely studied by many researchers (Shehory, Sycara, and Jha 1997; Sandholm and Lesser 1995; Wooldridge and Jennings 1997). Determining which agents to team with is a sort of decision making capability.

### 1.4.4 Agent Programming

Shoham (1993) 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."*

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.

Closely related to Shoham's work is that of Hindriks, de Boer, van der Hoek, and Meyer (1997) where an agent programming language based on BDI-agents is presented. They proceed upon the assumption that an agent language must have the ability to update its beliefs and its goals, and it must have a practical reasoning method (which will find a way to achieve goals). Hindriks, de Boer, van der Hoek, and Meyer (1997, p. 211) 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."*

When compared to Singh's approach described earlier in this chapter, these approaches provide a compact way of representing a massive finite state automaton (only the initial state is explicit) and transitions are specified through actions and rules governing the actions. This is very appealing, and the semantics is very clean.

However, both approaches assume that all the reasoning done by agents is implemented in one form of logic or another, and that all agents involved manipulate logical data. While logic is a reasonable *abstraction* of data, it remains a fact of life that the vast majority of data available today is in the form of non-logical data structures that vary widely.

The second assumption made is that *all* reasoning done by agents is encoded through logical rules. While this is also reasonable as an abstraction, it is rarely true in practice. For example, consider the planning performed by the Auto-Pilot agent in the CFIT example, or the profiling performed by the Profiling agent in the department store example, or the route planning performed by the Truck agent in the in the CHAIN example. These three activities will, in all likelihood, be programmed using imperative code, and mechanisms such as those alluded to above must be able to meaningfully reason on top of such legacy code.

### 1.4.5 Agent Architectures

An architecture for the creation and deployment of multiagent applications must satisfy three goals:

1. First and foremost, it must provide an architecture for designing software agents.

2. It must provide the underlying software infrastructure that provides a common set of services that agents will need.

3. It must provide mechanisms for interactions between clients and the underlying agent infrastructure.

As we have already discussed the first point earlier in this section, we will confine ourselves to related work on the latter two components.

With respect to agent architectures, there have been numerous proposals in the literature, e.g., (Gasser and Ishida 1991; Glicoe, Staats, and Huhns 1995; Birmingham, Durfee, Mullen, and Wellman 1995), which have been broadly classified by Genesereth and Ketchpel (1994) into four categories:

1. 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. In the context of our CFIT example introduced this means that each agent in the example must have the ability to understand messages sent to it by other agents. However, the CFIT example shows only a small microcosm of the functioning of the Auto-Pilot agent. In reality, the Auto-Pilot needs to interact with agents associated with hundreds of sensors and actuators, and to require that the transducers anticipate what other agents will send and translate it is clearly a complex problem. In general, in an $n$-agent system, we may need $O(n^2)$ transducers, which is clearly not desirable.

2. The second approach is based on wrappers which *"inject code into a program to allow it to communicate"* (Genesereth and Ketchpel 1994, p. 51). This idea is based on the principle that each agent has an associated body of code that is expressed in a common language used by other agents (or is expressed in one of a very small number of such languages). This means that in the case of the CFIT example, each agent is built around a body of software code, and this software code has an associated body of program code (expressed perhaps in a different language) expressing some information about the program.

3. The third approach described in (Genesereth and Ketchpel 1994) is to completely rewrite the code implementing an agent, which is obviously a very expensive alternative.

4. Last but not least, there is the *mediation* approach proposed by Wiederhold (1993), which assumes that all agents will communicate with a mediator which in turn may send messages to other agents. The mediation approach has been extensively studied (Arens, Chee, Hsu, and Knoblock 1993; Brink, Marcus, and Subrahmanian 1995; Chawathe, S., et al. 1994; Bayardo, R., et al. 1997). However, it suffers from a problem. Suppose all communications in the CFIT example had to go through such a mediator. Then if the mediator malfunctions or "goes down," the system as a whole is liable to collapse, leaving the plane in a precarious position. In an agent based system, we should allow point to point communication between agents without having to go through a mediator. This increases reliability of the entire multiagent system as a whole and often avoids inefficiency by avoiding huge workloads on certain agents or servers or network nodes.

### 1.4.6 Match-making Services

As stated before, one of the infrastructural tasks to be provided is yellow pages services whereby agents may advertise services they offer (via the yellow pages) and the infrastructure layer allows for identifying agents $A$ that provide a service *similar* to a service requested by agent $B$. For instance, in the CHAIN example, the plant agent may need to contact such a yellow pages service in order to find agents that can provide the supply item needed. The yellow pages agent must attempt to identify agents that provide either the exact supply item required, or something similar to the requested item. Kuokka and Harada (1996) present the *SHADE* and *COINS* systems for matchmaking. *SHADE* uses logical rules to support matchmaking—the logic used is a subset of *KIF* and is very expressive. In contrast, *COINS* assumes that a message is a document (represented by a weighted term vector) and retrieves the *most similar* advertised services using the *SMART* algorithm of Salton and McGill (1983). Decker, Sycara, and Williamson (1997) present matchmakers that store capability advertisements of different agents. They look for *exact* matches between requested services and retrieved services, and concentrate their efforts on architectures that support load balancing and protection of privacy of different agents.

## 1.5 Ten Desiderata for an Agent Infrastructure

In this book, we will describe advances in the construction of agents, as well as multiagent systems. Our intent is to provide a rich formal theory of agent construction and agent interaction that is *practically* implementable and realizable. *IMPACT* (*Interactive Maryland Platform for Agents Collaborating Together*) is a software platform for the creation and deployment of agents, and agent based systems. In this book, we will provide one set of answers to the following questions raised at the beginning of this chapter:

**(Q1)** *What is an agent?*

**(Q2)** *If program P is not considered to be an agent according to some specified definition of agenthood, is there a suite of tools that can help in "agentizing" P?*

**(Q3)** *Once a specific definition of agenthood is chosen, what kind of software infrastructure is required to support interactions between such agents, and what core set of services must be provided by such an infrastructure?*

In particular, any solution to the above questions must (to our mind) satisfy the following important desiderata:

**(D1)** Agents are for everyone: anybody who has a software program *P*, either custom designed to be an agent, or an existing legacy program, must be able to *agentize* their program and *plug* it into the provided solution. In particular, in the case of the CFIT example, this means that if a new Satellite agent becomes available, or a better flight planning Auto-Pilot agent is designed, plugging it in should be simple. Similarly, if a new Supplier agent is identified in the CHAIN example, we should be able to access it easily and incorporate it into the existing multiagent CHAIN example. Any theory of agents must encompass the above diversity.

**(D2)** No theory of agents is likely to be of much practical value if it does not recognize the fact that data is stored in a wide variety of data structures, and data is manipulated by an existing corpus of algorithms. If this is not taken into account in a theory of agents, then that theory is not likely to be particularly useful.

**(D3)** A theory of agents must *not* depend upon the set of actions that the agent performs. Rather, the set of actions that the agent performs must be a *parameter* that is taken into account in the semantics. Furthermore, any proposed action framework must allow actions to have effects on arbitrary agent data structures, and must be capable of being built seamlessly on top of such existing applications.

**(D4)** Every agent should execute actions based on some *clearly articulated* decision policy. While this policy need not be disclosed to other agents, such a specification is invaluable when the agent is later modified. We will argue that a *declarative* framework for articulating decision policies of agents is imperative.

**(D5)** Any agent construction framework must allow agents to perform the following types of reasoning:

- Reasoning about its beliefs about other agents.
- Reasoning about uncertainty in its beliefs about the world and about its beliefs about other agents.
- Reasoning about time.

These capabilities should be viewed as *extensions* to a core agent action language, that may be "switched" on or off, depending upon the reasoning needs of an agent. The reason for this is that different agents need to reason at different levels of sophistication. However, increasingly sophisticated reasoning comes at a computational price, viz. an increase in complexity (as we will see in the book).

Thus, it is wise to have a *base language* together with a *hierarchy* of extensions of the base language reflecting increased expressive power. Depending on which language within this hierarchy the agent wishes to use, the computational price to be paid by the agent should be clearly defined. This also requires that we have a *hierarchy* of compilers/interpreters mirroring the language hierarchy. It is in general computationally unwise to use a solver for a language "high" in the hierarchy if an agent is using a language "low" in the hierarchy (e.g., using a solver for a PSPACE-complete problem on a polynomial instance of the problem is usually not wise).

**(D6)** Any infrastructure to support multiagent interactions *must* provide two important types of security—security on the agent side, to ensure that an agent (if it wishes) can protect some of its information and services, and security on the infrastructural side so that one agent cannot masquerade as another, thus acquiring access to data/services that is not authorized to receive.

**(D7)** While the efficiency of the code underlying a software agent cannot be guaranteed (as it will vary from one application to another), guarantees are needed that provide information on the performance of an agent relative to an oracle that supports calls to underlying software code. Such guarantees must come in two forms—results on *worst case complexity* as well as accompanying *experimental results*. Both these types of results are useful, because in many cases, worst case complexity results do not take into account specific patterns of data requests that become apparent only after running experiments. Conversely, using experimental results alone is not adequate, because in many cases, we do want to know worst case running times, and experimental data may hide such information.

**(D8)** Efficiency of an implementation of the theory is critical in the development of a multiagent system. We must identify efficiently computable *fragments* of the general hierarchy of languages alluded to above, and our implementations must take advantage of the specific structure of such language fragments. A system built in this way must be accompanied by a suite of software tools that helps the developer build sophisticated multiagent systems.

**(D9)** A critical point is *reliability*—there is no point in a highly efficient implementation, if all agents deployed in the implementation come to a grinding halt when the agent "infrastructure" crashes.

**(D10)** The only way of testing the applicability of any theory is to build a software system based on the theory, to deploy a set of applications based on the theory, and to report on experiments based on those applications. Thus, an implementation *must be validated* by a set of deployed applications.

## 1.6 A Birdseye View of this Book

This book is organized as follows.

Chapter 2 introduces the reader to the overall architecture of the proposed *IMPACT* framework. It explains the issues involved in designing the architecture, what alternative architectures could have been used, and why certain design choices were made. It explains the architecture of individual agents, as well as the architecture of the agent infrastructure, and how the two "fit" together, using the STORE, CFIT, and CHAIN examples to illustrate the concepts.

Chapter 3 explains the *IMPACT* Service Description language using which an agent may specify the set of services that it offers. We describe the syntax of the language and specify the services offered by various agents in the STORE, CFIT, and CHAIN examples using this syntax. We further specify how requests for *similar* services are handled within this framework. We explain existing alternative approaches, and describe the advantages of disadvantages of these approaches when compared to ours.

Chapter 4 shows how agents may be built on top of legacy data, using a basic mechanism called a *code call condition*. We will show how such access methods may be efficiently implemented and we will describe our implementation efforts to date to do so. As in other cases, the STORE, CFIT, and CHAIN examples will be revisited here.

Chapter 5 describes the implementation of *IMPACT* Servers. These are programs that provide the "infrastructural" services needed for multiple agents to interact, including yellow pages and other services described in detail in Chapter 3. This chapter explains how to access these servers, how they were implemented, and how agents may interact to them. A theorist may wish to skip this chapter, but an individual seeking to implement an agent system may find this chapter very useful.

Chapter 6 builds on top of Chapter 4 and shows how an agent's action policies may be declaratively specified. Such declarative policies must encode what the agent is permitted to do, what it is forbidden from doing, what it is obliged to do, and what in fact, it does, given that the agent's data structures reflect a "current state" of the world. We show that the problem of determining how to "act" (which an agent must make continuously) in a given agent state may be viewed as computing certain kinds of objects called "status sets." In this chapter, we assume a frozen instant of time, and make very few assumptions about "states." These concepts are illustrated though the STORE, CFIT, and CHAIN examples.

In Chapter 7 we argue that an agent's state may (but does not have to!) contain some information about the agent's beliefs about other agents. This is particularly useful in adversarial situations where agent $a$ might want to reason about what agent $b$'s state before deciding what to do. The theory of Chapter 4 is extended to handle such meta-reasoning. There is also another example introduced, the RAMP example, which was particularly designed agents reasoning about beliefs.

In Chapter 8, we extend the theory of Chapter 6 in yet another direction—previously, a "frozen" instant of time was assumed. Of course, this is not valid—we all make decisions today on what we will do tomorrow, or day after, or next month. We create schedules for ourselves, and agents are no different. This chapter describes an extension of the theory of Chapter 7 to handle such temporal reasoning.

In Chapter 9, we add a further twist, increasing the complexity of both Chapter 7 and 8, by assuming that an agent may be *uncertain* both about its beliefs (about the state of the world, as well as its beliefs about other agents). The theory developed in previous chapters is extended to handle this case.

In Chapter 10, we revert to our definition of states and actions, and examine specific data structures that an agent must maintain in order to preserve security, and specific actions it can take (relative to such data structures) that allow it to preserve security. We further explore the relationship between actions taken by individual agents and the data structures/algorithms built into the common agent infrastructure, with a view to maintaining security.

In Chapter 11, we develop a body of complexity results, describing the overall complexity of the different languages developed in preceding chapters. The chapter starts out with a succinct summary and interpretation of the results—a reader interested in the "bottom line" may skip the rest of this chapter.

In Chapter 12, we identify efficiently computable fragments of agent programs, and provide polynomial algorithms to compute them. We explain what can, and what cannot, be expressed in these fragments. We then describe *IADE*—the *IMPACT* Agent Development Environment, that interested users can use to directly build agents in *IMPACT*, as well as build multiagent systems in *IMPACT*. We will report on experiments we have conducted with *IMPACT*, and analyze the performance results we obtain.

In Chapter 13, we will describe in detail, an integrated logistics application we have built within the *IMPACT* framework for the US Army.

Finally, in Chapter 14, we will revisit the basic goals of this book—as described in Chapters 1 and 2, and explain how we have accomplished them. We identify the strengths of our work, as well as shortcomings that pave the way for future research by us, and by other researchers.

## 1.7 Selected Commercial Systems

There has been an increase in the number of agent applications and agent infrastructures available on the Internet. In this section, we briefly mention some of these commercial systems.

Agents Technologies Corp.'s *Copernic 98* (`http://www.copernic.com/`) integrates information from more than 110 information sources.

Dartmouth College's *D'Agents* project (`http://www.cs.dartmouth.edu/~agent/`) supports applications that require the retrieval, organization, and presentation of distributed information in arbitrary networks.

Firefly's *Catalog Navigator* (`http://www.firefly.net/company/keyproducts.fly`) allows users to add preference and general interest-level information to each customer's personal profile, hence providing more personalized service.

General Magic's *Odyssey* (`http://www.genmagic.com/agents/`) provides class libraries which enable people to easily develop their own mobile agent applications in Java. It also includes third party libraries for accessing remote *CORBA* objects or for manipulating relational databases via JDBC.

IBM's *Aglets* provide a framework for development and management of mobile agents. (`http://www.trl.ibm.co.jp/aglets/`). An aglet is a Java object having mobility and persistence and its own thread of execution. Aglets can move from one Internet host to another in the middle of execution, (Lande and Osjima 1998). Whenever an aglet moves, it takes along its program code and data. Aglets are hosted by an Aglet server, as Java applets are hosted by a Web browser.

Microelectronics and Computer Technology Corporation's *Distributed Communicating Agents* (DCA) for the Carnot Project (`http://www.mcc.com/projects/carnot/DCA.html`) enables the development and use of distributed, knowledge-based, communicating agents. Here, agents are expert systems that communicate and cooperate with human agents and with each other.

Mitsubishi Electric ITA Horizon Systems Laboratory's *Concordia* (`http://www.meitca.com/HSL/Projects/Concordia/`) is a full-fledged framework for development and management of network-efficient mobile agent applications for accessing information anytime, anywhere, and on any device supporting Java. A key asset is that it helps abstract away the specific computing or communication devices being used to access this data.

ObjectSpace's *Voyager* (`http://www.objectspace.com/voyager/`) allows Java programmers to easily construct remote objects, send them messages, and move objects between programs. It combines the power of mobile autonomous agents and remote method invocation with CORBA support and distributed services.

Oracle's *Mobile Agents* (`http://www.oracle.com/products/networking/mobile_agents/html/index.html`) is networking middleware designed to facilitate connectivity over low bandwidth, high latency, occasionally unreliable, connections. It may be used to help provide seamless data synchronization between mobile and corporate databases.

Softbot (software robot) programs (`http://www.cs.washington.edu/research/projects/softbots/www/projects.html`) are intelligent agents that use software tools and services on a person's behalf. They allow a user to communicate what they want accomplished and then dynamically determine how and where to satisfy these requests.

Stanford's *Agent Programs* (`http://www-ksl.stanford.edu/knowledge-sharing/agents.html`) provide several useful agent-related utilities such as a content-based router (for agent messages), a matchmaker (w.r.t. agent interests), and many more. They follow the KIF/KQML protocols (Neches, Fikes, Finin, Gruber, Patil, Senator, and Swarton 1991; Genesereth and Fikes 1992; Labrou and Finin 1997a; Finin, Fritzon, McKay, and McEntire 1994; Mayfield, Labrou, and Finin 1996; Finin, T., et al. 1993) for knowledge sharing.

UMBC's *Agent Projects* (`http://www.cs.umbc.edu/agents/projects/`) include several applications such as Magenta (for the development of agent-based telecommunication applications), AARIA (for autonomous agent based factory scheduler at the Rock Island Arsenal), etc. UMBC also maintains descriptions of several projects using *KQML* (`http://www.csee.umbc.edu/kqml/software/`), a Knowledge Query and Manipulation Language for information exchange.

Some other sites of interest include the *Agent Society* (`http://www.agent.org/`) and a site `http://csvax.cs.caltech.edu/~kiniry/projects/papers/IEEE_Agent/agent_paper/agent_paper.html`), which surveys *Java Mobile Agent Technologies*

# Chapter 2

# *IMPACT* Architecture

In order to describe an architecture that supports the dynamic interaction of multiple software agents, three fundamental questions need to be answered.

1. What does it mean for a program $P$ written in some arbitrary programming language to be considered an agent (how does a suitable `isagent` predicate look like)?

2. Once such a definition of the `isagent` predicate is provided, what underlying infrastructural capabilities are needed in order to allow these agents to interact meaningfully with each other?

3. How will multiple software agents communicate with one another, and how will agents and the infrastructure communicate with one another?

This chapter *sketches* out solutions to all these problems. The rest of this book will precisely describe the mathematics underlying these solutions, and go into details about algorithms for computing different problems within this architecture.

## 2.1 Overview of Architecture

In this section, we provide a general overview of the *IMPACT* architecture. In *IMPACT*, we have two kinds of entities:

**Agents,** which are software programs (legacy or new) that are augmented with several new interacting components constituting a *wrapper*. Agents may be created by either arbitrary human beings or by other software agents (under some restrictions).

***IMPACT* Servers,** which are programs that provide a range of infrastructural services used by agents. *IMPACT* Servers are created by the authors of this book, rather than by arbitrary individuals.

Figure 2.1 on the next page provides a brief high level description of the *IMPACT* system architecture. According to this architecture, *IMPACT* agents may be scattered across the network. *IMPACT* servers may, likewise, be replicated an/or mirrored, and also located at disparate points on the network. Figure 2.1 on the following page illustrates the following:

- *Agent to agent connectivity is allowed*, which facilitates interactions such as

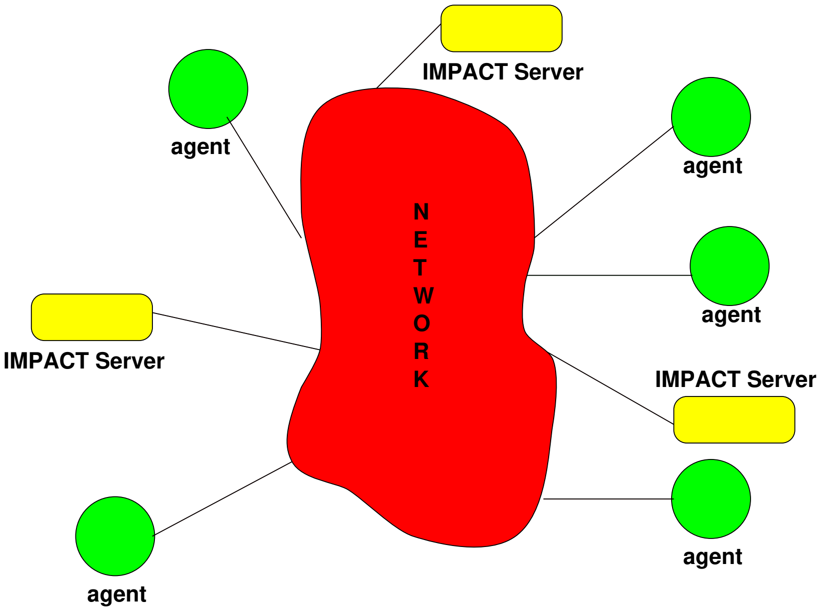



Figure 2.1: Overall *IMPACT* Architecture

- Agent $a$ requests agent $b$ to provide a service (e.g., in the CHAIN example, the `plant` agent may request a `supplier` agent to specify by when it can provide 500 items of *widget-50*.)
- Agent $b$ sends agent $a$ the answer to agent $a$'s request.

• *Agent to server connectivity is allowed* which facilitates interactions such as

- Agent $a$ requests the server to identify all agents that provide a given service (e.g., in the CHAIN example, the `plant` agent may request the *IMPACT* server to identify all agents capable of supplying *widget-50*.
- The server sends agent $a$ a list of agents that the server believes are capable of providing the desired service (possibly with additional accompanying information).
- Agent $a$ requests the server for other descriptors of a word such as `car`.
- The server sends to agent $a$ a list of synonyms of the requested word.

We are now ready to describe the software-level architecture of *IMPACT* agents.

## 2.2 Agent Architecture

An *IMPACT* agent may be built on top of an arbitrary piece of software, defined in any programming language whatsoever. *IMPACT* agents have the following components.

**Application Program Interface:** Each *IMPACT* agent has an associated *application program interface* (*API*) that provides a set of functions which may be used to manipulate the data structures managed by the agent in question. The API of a system consists of a set of procedures that enable external access and utilization of the system, without requiring detailed knowledge of system internals such as the data structures and implementation methods used.

Thus, a remote process can use the system via procedure invocations and gets results back in the form defined by the output of the API procedure.

For instance, in the case of the STORE example, every time the `profiling` agent makes a request to the `credit` agent, one or more functions must be executed on the data structures managed by the `credit` agent. The task of the API is to specify the set of such available functions, together with their signatures (input/output types).

**Service Description:** Each *IMPACT* agent has an associated *service description* that specifies the set of services offered by the agent. Each service has four parts:

- A *name*—for example, the `gps` agent in the CFIT example may provide a service called *provide: location* which specifies the location of a plane at a given instance in time.
- A *set of mandatory inputs* that must be provided in order for the function to be executable: in the STORE example, providing a potential customer's card number might be mandatory before the `credit` agent `credit` provides a credit report.
- A *set of discretionary inputs* that may (but do not have to) be provided—returning to the STORE example, providing a potential customer's name may not be mandatory.
- A *set of outputs* that will be returned by the function.

Of course, type information must be specified for all the above inputs and outputs. Chapter 3 provides a detailed description of our service description language, together with algorithms to manipulate service descriptions, and identify agents that provide a given service.

**Message Manager:** Each agent has an associated module that manages incoming and outgoing messages.

**Actions, Constraints, and Action Policies:** Each agent has a set of actions that it can physically perform. The actions performed by an agent are capable of changing the data structures managed by the agent and/or changing the message queue associated with another agent (if the action is to send a message to another agent). Each agent has an associated *action policy* that states the conditions under which the agent

- *may*,
- *may not*, or
- *must*

do some actions. The actions an agent can take, as well as its action policy, must be clearly stated in some declarative language. Furthermore, there might be constraints stating that certain ways of populating a data structure are "invalid" and that certain actions are not concurrently executable. Chapter 6 provides such a language, and describes its syntax and semantics. Chapters 8 and 9 extend this language to handle reasoning with uncertainty and time.

**Metaknowledge:** Some agents may hold beliefs about other agents, and use these beliefs in specifying action policies. For example, in the case of the CFIT example, the `gps` agent may believe that transmissions from a given `satellite` agent are being jammed by an enemy agent, [1] and in such a case, it may attempt to notify another agent to identify the source of the jamming signal. On the other hand, some agents may not need to reason about other agents or about

[1] We understand Russia currently markets an off the shelf GPS jammer.

the world. In the CHAIN example, the `supplier` agent may do nothing more sophisticated than answering a database retrieval query. Our framework for creating agents must be rich enough to support both possibilities, as well as other intermediate situations. In general, each agent may have certain *metaknowledge* structures that are used for such reasoning. Chapter 7 provides a framework for such metaknowledge structures, and how they may be manipulated.

**Temporal Reasoning:** In some applications, such as the CHAIN example, agents may *schedule* actions to take place in the future. For instance, the supplier agent makes a commitment to deliver certain items at certain fixed points in time in the future. This requires the ability for agents to make future commitments. Chapter 8 extends the theory of Chapter 6 to handle this situation.

**Reasoning with Uncertainty:** The designer of an application needs to take into account the fact that the state of an agent may be uncertain. For example, consider the CFIT example. Here, the `autoPilot` agent may detect an aircraft, but does not know if the detected aircraft is a friend or a foe. Based on its sensors, it may have *uncertain beliefs* about the properties of the aircraft, as well as uncertainty about the *actions* that the enemy aircraft will take. Thus, the `autoPilot` agent needs to reason with this uncertainty in order to make a decision. Chapter 9 extends the theory of Chapter 8 to handle this case.

**Security:** The designer of any agent has the right to enforce any security policies that he or she deems appropriate. Some agents may have significant security components, others may have none at all. Any framework for creating agents must be rich enough to support both extremes, as well as intermediate security strategies. For instance, an agent may treat the same request from two agents differently, based on what it is willing to disclose to other agents. In the case of the CFIT example, the `autoPilot` agent may provide its flight path to the commander of the mission, but may not be willing to provide its flight path to other agents. In general, each agent may have certain *security* related data structures which are used for the purpose of maintaining security. Such security related data structures may well build on top of the existing meta knowledge structures that the agent has. Chapter 10 provides a framework for such security structures, and how they may be manipulated.

Figure 2.2 on the next page provides a pictorial view of how an agent is configured. The components shown in blue denote legacy components (which the developer of an *IMPACT* agent does not have to build), while the components shown in yellow denote components built by the person creating an *IMPACT* agent. The arrows indicate flow of data. Note that only the action policy can cause changes on the data stored in the underlying software's data structures, and such changes are brought about through appropriate function calls.

In order to illustrate the agent architecture described thus far, we now briefly revisit the STORE, CFIT, and CHAIN examples, and illustrate (briefly) how one of the agents in each of these scenarios may be captured within our agent architecture. However, as we have thus far not explicitly described languages for specifying service descriptions, metaknowledge, actions and action policies, etc., these examples will be informally described in English.

### 2.2.1 STORE Revisited

Consider the `profiling` agent used in the STORE example. This agent may have several services, two of which are listed below.

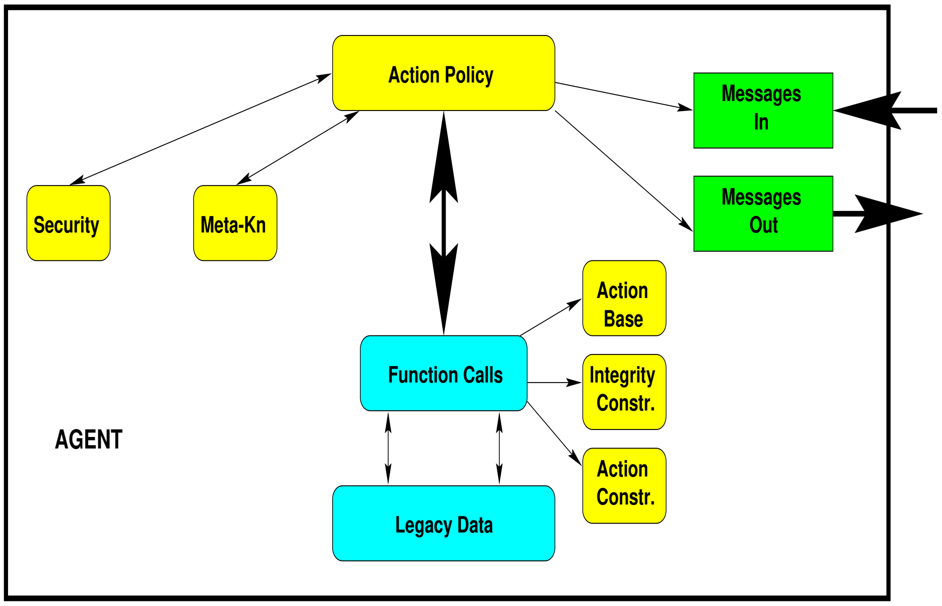



Figure 2.2: Basic Architecture of *IMPACT* Agents

- *classify: user*.
  This service may take as input, the social security number of a user, and provide as output, a classification of the user as a "low," "medium," "high," or "very high" spender.

- *provide: user-profile*.
  This service also takes as input, the social security number of a user, and yields as output, a set of pairs of the form *(clothing,high), (appliances,low), (jewelry,high)*, ... Unlike the service *classify: user*, the service *provide: user-profile* provides a more detailed classification of the user's spending habits.

In addition, the agent may support specific actions such as:

- *update* actions (that update user profiles) when more information becomes available about their purchases and

- *respond* actions that take a request (from another agent or human client) and give an answer to it.

The `profiling` agent may use, as an action policy, the rule that only agents created and owned by the department store may request all its profiling services. In addition, certain other agents not owned by the store, but who pay the store some money, may access the *classify: user* service. Of course, this is closely tied to the security requirements of this agent. The `profiling` agent may have an "empty" metaknowledge component, meaning that this agent does not perform any metareasoning about other agents and their beliefs.

### 2.2.2 CFIT Revisited

Now let us consider the `autoPilot` agent in the CFIT example. This agent may provide the following services;

- maintain: *course*
  This service may take as input the current scheduled flight path of a plane, and try to maintain it throughout the flight, by returning a sequence of actions to be performed. Informally, this function merely examines the current flight path schedule, and "reads off" what actions are scheduled to be done at this point in time, and returns this set of actions.

- adjust: *course*
  This service may take as input, a set of no-go areas from the `terrain` agent, and take appropriate action according to the plane's current location and its allocated flight path. Unlike the preceding function, it does not return a set of actions by reading it from the flight plan—rather, it creates a set of actions (i.e., it constructs a plan) to avoid an unexpected contingency.

- return: *control*
  This service may take as input, the id of the requester and relinquish control of the plane to the requester if he or she is the pilot of the plane. This service may have no output.

- create: *plan(flight)*
  This service may take as input, the GPS-based location of the plane, a set of "no-go" areas, and the plane's allocated flight path, and generate a flight plan for the plane.

Moreover, the `autoPilot` agent might support several specific actions, some of which could be the following;

- *collect* actions that collect GPS data from the on-board sensors and actuators as well as from the `terrain` agent,

- *compute* actions that compute the current location of the plane based on the collected GPS data,

- *climb* actions that climb by some specific amount per second to avoid no-go areas.

In addition, the `autoPilot` agent may have an action policy that determines the conditions under which it executes the above actions. For example, it may execute the *compute* action every 2 minutes (i.e. based on clock information), while it executes course information based on GPS data, terrain data, as well as data from on-board sensors. Moreover, the `autoPilot` agent might have to return control to the pilot whenever he requests so. The agent's security requirements might insist that the agent provides its flight plan and path only to the pilot of the plane. Furthermore, the meta-knowledge component of the `autoPilot` agent may consist of its beliefs about the on-board sensors and actuators and the `terrain` agent. The `autoPilot` agent might reason about the plane's current location based on those beliefs and take appropriate actions. For example, if the `autoPilot` agent frequently receives course adjustment requests from a `satellite` agent, while the `terrain` agent and on-board sensors do not alert the `autoPilot` agent of its incorrect path, it may conclude that the `satellite` agent is an enemy trying to falsify the `autoPilot` agent. As is clear from the preceding discussion, the `autoPilot` agent may have to reason with uncertainty.

### 2.2.3 CHAIN Revisited

Let us consider the CHAIN example. Here, the `supplier` agent may provide a variety of services such as:

- *monitor: available-stock*
  This service may take as input the amount and the name of the requested part, and then checks the ACCESS database to determine if the requested amount can be provided. It either returns the string "amount available" or "amount not available."

- *monitor: committed-stock*
  This service may take as input the name of some part, and check an ACCESS database to see how much of that part is committed, and return as output the amount committed.

- *update: stock*
  This service takes as input the name of a requested part and the amount requested. It first checks to see if the requested amount is available using the *monitor :committed-stock* function above. It then updates the available stock database, reducing the available amount by the amount requested. It also updates a "commitments" database, by adding the amount requested to the committed amount.

In addition, the `supplier` agent may support a set of specific actions, which might include the following:

- *update* actions that update the two ACCESS databases, and

- *respond* actions that take a part request and either confirm or reject the request.

The `supplier` agent's actions may be based on principles such as the following:

1. Parts may be ordered only by agents with whom the `supplier` agent has an existing contract.

2. Orders may be taken from agents that have a large outstanding payment balance, but such orders may trigger a *"We need you to make overdue payments before we ship your order"* message action to be executed.

Unlike other agents listed above, the `supplier` agent may have an empty metaknowledge component, as it may not perform any metareasoning about other agents and their beliefs.

## 2.3 Server Architecture

Consider the CHAIN example. It may well be the case that two "approved" supplier agents cannot provide a requested part that the `plant` agent needs to keep its production line running. In this case, the `plant` agent must locate new sources from which this part can be obtained. In an electronic setting, this means that there must be some *Yellow Pages Services* available. In such a case, the `plant` agent can utilize this service and ask for identities of potential new supplier agents.

Broadly speaking, there are two general mechanisms for identifying service providers:

- In the first mechanism, the agent requesting or providing a service communicates with an *appropriate* yellow pages server. This approach first assumes the existence of a yellow pages server. Second, it assumes that the agent has a mechanism to identify which of several possible yellow pages servers is appropriate for its needs. Third, it assumes that somehow, the yellow pages server has information about agents and their services. Fourth, it assumes that all these agent services are described in some uniform language by the yellow pages server.

- In the second mechanism, the agent requesting or providing a service broadcasts this fact to all *appropriate* agents. This assumes that the agent knows who to broadcast this information to. Second, it assumes that all agents share a common language that supports this interaction.

The first approach places significant demands on the yellow pages server, but efficiently utilizes network resources and reduces load on agents (e.g., agents will not be flooded by unwanted messages). The second approach on the other hand makes fewer assumptions than the former, but freely uses network bandwidth, and hence, participating agents may be overwhelmed by the number of advertisements for services and requests for services sent to them.

In our effort, we have chosen to go with the first approach and we have taken it upon ourselves to provide all the infrastructural facilities that are needed so as to reduce unwanted message traffic and agent workload. This is accomplished by *IMPACT* Servers—in fact, an *IMPACT* Server is actually a collection of the following servers:

**Registration Server:** This server is mainly used by the creator of an agent to specify the services provided by it and who may use those services.

**Yellow Pages Server:** This server processes requests from agents to identify other agents that provide a desired service.

**Thesaurus Server:** This server receives requests when new agent services are being registered as well as when the yellow pages server is searching for agents providing a service.

**Type Server:** This server maintains a set of class hierarchies containing information about different data types used by different agents, and the inclusion relationship(s) between them.

We now describe these services in greater detail.

### 2.3.1 Registration Server

When the creator of an agent wishes to deploy it, he *registers* the agent with an *IMPACT* server, using a *Registration Interface*. Figure 2.3 on the facing page shows our prototype Java-based registration interface for registering a single agent. When registering an agent, the user specifies the services provided by that agent. A wide variety of languages may be used for this purpose—using straight English would be one such (extreme) example. Suppose that SDL (service description language) is the chosen one.

In this case, when an agent $\mathfrak{a}$ wants to find another agent providing a service $q_s$ expressed in SDL, we must *match* $q_s$ with other service descriptions stored in the yellow pages, in order to find appropriate services. Efficiently doing this with free English descriptions and with $q_s$ expressed in free flowing English is currently not feasible.

As a compromise, we have chosen to use *words* from English to describe services, but no free text is allowed. Instead, we assume in our framework that each service is named by a verb and a special structure called a noun-term. However, certain words in English are "similar" to others. For example, in the STORE example, the words *cup* and *mug* are similar, and a customer wanting coffee mugs will probably be interested in agents selling coffee mugs as well as coffee cups. As a consequence, there is a need for data structures and algorithms to support such *similarity based* retrieval operations on service names. The Registration Server creates such data structures, and supports:

- *insertion* of new service names,

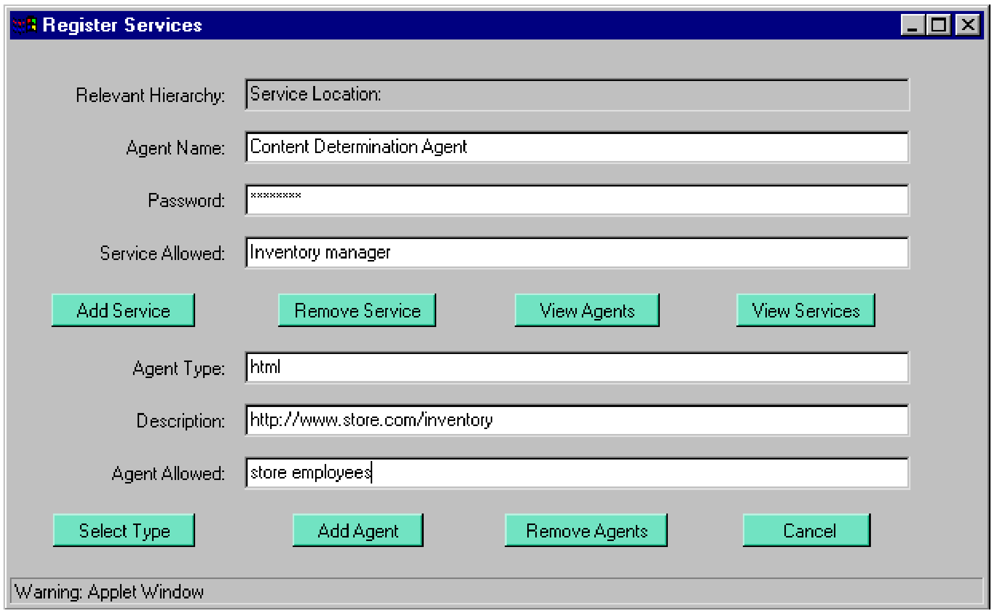



Figure 2.3: Agent/Service Registration Screen Dump

- *insertion* of data specifying that a given agent provides a service (and deletion of such information),
- *browsing* of these data structures.

As searching these data structures for *similarity based* retrieval operations performed by the yellow pages server is to be described in Section 2.3.4, we now proceed to some basic concepts needed before the above mentioned data structures can be described.

### Hierarchies

We assume that all agents use English words (including proper nouns) to express information about services. However, different multiagent applications may only use a small fragment of legitimate English words. For example, the CFIT example may use a vocabulary consisting of various flight and terrain related terminology—in contrast, the STORE example may not have words like *aileron* and *yaw* in its vocabulary.

Suppose Verbs is a set of verbs in English, and Nouns is a set of nouns in English. Note that these sets are not necessarily disjoint. For instance, in the CFIT example, *plan* may be both a verb and a noun. A *noun term* is either a noun or an expression of the form $n_1(n_2)$ where $n_1, n_2$ are both nouns. Thus for instance, if the nouns *flight*, *plan*, *route* are in the vocabulary of the CFIT example, then *flight*, *plan*, *route*, *flight(plan)*, *plan(flight)* are (some) noun terms. We use the notation $\mathsf{nt}(\mathsf{Nouns})$ to denote the set of all syntactically valid noun terms generated by the set Nouns.

**Definition 2.3.1 (Service Name)**
*If* $v \in \mathsf{Verbs}$ *and* $nt \in \mathsf{nt}$*, then* $v\!:\!nt$ *is called a* service name.

If *create* is a verb in the vocabulary of the CFIT example, then *create*: *plan(flight)* is a service name. For that matter, so is *create*: *route*.

| AGENT | SERVICES |
|---|---|
| credit | *provide: information(credit)*<br>*provide: address* |
| profiling | *provide: user-profile*<br>*classify: user* |
| productDB | *provide: description(product)*<br>*identify: product* |
| contentDetermin | *prepare: presentation(product)*<br>*determine: advertisement*<br>*identify: items* |
| interface | *present: presentation(product)*<br>*provide: information(product)*<br>*provide: advertisement* |
| saleNotification | *identify: user-profile*<br>*determine: items*<br>*mail: brochure*<br>*create: mail-list* |

Table 2.1: Service List for the STORE example

It is important to note that once we are given the sets Verbs and Nouns, the space of syntactically valid service names is uniquely determined. Of course, only a few of these service names will make sense for a given multiagent application. In most applications, the sets Verbs and Nouns evolve over time as more and more agents' services are registered. When the creator of an agent registers the agent with the *IMPACT* server, he might introduce new verbs and nouns which would enlarge the sets Verbs and Nouns. Table 2.1, Table 2.2 and Table 2.3 on page 34 give a comprehensive list of the names of services offered by the agents in the STORE,CFIT and CHAIN examples.

| AGENT | SERVICE |
|---|---|
| autoPilot | *maintain: course*<br>*adjust: course*<br>*return: control*<br>*create: plan(flight)* |
| satellite | *broadcast: data(GPS)* |
| gps | *collect: data(GPS)*<br>*merge: data(GPS)*<br>*create: information(GPS)* |
| terrain | *generate: map(terrain)*<br>*determine: area(no-go)* |

Table 2.2: Service List for the CFIT example

Now consider the terrain agent used in the CFIT example. This agent provides a service called *generate: map(terrain)*. In particular, the terrain agent may provide this service to many agents in addition to agents in the CFIT example. Consider the following two situations:

1. Some agent on the network wants to find an agent providing a service *generate: map(ground)*.

The words *ground* and *terrain* are synonyms, and hence, the CFIT `terrain` agent is a potential candidate agent providing the desired service. However, this can only be found by using a thesaurus, which we discuss later in Section 2.3.3.

2. A second possibility is that the above agent wants to find an agent providing a service called *generate: map(area)*. Certainly, the CFIT `terrain` agent above can provide terrain maps of areas. However, here, the word *area* is being specialized to a more specific word, *terrain*. We now specify how this kind of reasoning may be accomplished.

Suppose $\Sigma$ is any set of English words, such that either all words in $\Sigma$ are verbs, or all words in $\Sigma$ are noun-terms. Furthermore, suppose $\sim$ is an arbitrary equivalence relation on $\Sigma$.

**Definition 2.3.2 ($\Sigma$-node)**
*A* $\Sigma$-node *is any subset $N \subseteq \Sigma$ that is closed under $\sim$, i.e.*

1. $x \in N \,\&\, y \in \Sigma \,\&\, y \sim x \Rightarrow y \in N.$

2. $x, y \in N \Rightarrow x \sim y.$

*In other words,* $\Sigma$-nodes *are equivalence classes of* $\Sigma$.

Observe that the empty set is always a trivial $\Sigma$-node, which will be of no interest to us.

Intuitively, as the reader can already see from Tables 2.1–2.3, service names are often specified by using verbs and noun terms. Suppose the $\sim$ relation denotes "semantic equivalence," i.e., if we consider two words $w_1, w_2$, saying that $w_1 \sim w_2$ means that these two words are considered semantically equivalent. If $\Sigma$ is some vocabulary (of verbs or nouns), then a $\Sigma$-node is one that is closed under semantic equivalence. The first condition says that if two words are semantically equivalent, then they must label the same node. The second condition says that if two words label the same node, then they must be semantically equivalent. For example, in the case of the CFIT example, we would have $terrain \sim ground$. However, $terrain \not\sim area$.

It is easy to see, however, that in the context of maps, *terrain* is a specialization of *region*. Certainly, the phrase *terrain* refers to a specific aspect of a region. Terrain maps are maps (of a sort) of regions. This notion of specialization is captured through the definition of a $\Sigma$-Hierarchy given below.

**Definition 2.3.3 ($\Sigma$-Hierarchy)**
*A $\Sigma$-Hierarchy is a weighted, directed acyclic graph $\mathcal{SH} =_{def} (T, E, \wp)$ such that:*

1. *$T$ is set of nonempty $\Sigma$-nodes;*

2. *for $t_1$ and $t_2$ are different $\Sigma$-nodes in $T$, then $t_1$ and $t_2$ are disjoint;*

3. *$\wp$ is a mapping from $E$ to $\mathbb{Z}^+$ indicating a positive distance between two neighboring vertices.*[2]

Figure 2.4 on page 35 and Figure 2.5 on page 36 provide hierarchies describing the three motivating examples in this book. In particular, Figure 2.4 describes a hierarchy on verbs, and Figure 2.5 provides a hierarchy on noun-terms.

[2]We do not require $\wp$ to satisfy any metric axioms at this point in time.

| AGENT | SERVICE |
|---|---|
| plant | monitor: *inventory*<br>update: *inventory*<br>determine: *amount(part)*<br>choose: *supplier*<br>order: *part*<br>monitor: *performance(supplier)*<br>notify: *supplier*<br>cancel: *contract*<br>find: *supplier* |
| supplier | monitor: *available-stock*<br>monitor: *committed-stock*<br>update: *stock* |
| shipping | find: *truck*<br>find: *airplane*<br>prepare: *schedule(shipping)* |
| truck | provide: *schedule(truck)*<br>manage: *freight*<br>ship: *freight* |
| airplane | provide: *freight*<br>manage: *freight*<br>ship: *freight* |

Table 2.3: Service List for the CHAIN example

**Distances**

Given a $\Sigma$-Hierarchy $\mathcal{SH} =_{def} (T, E, \wp)$, the distance between two terms, $w_1, w_2 \in T$, is defined as follows:

$$d_{\mathcal{SH}}(w_1, w_2) =_{def} \begin{cases} 0, & \text{if some } t \in T \text{ exists such that } w_1, w_2 \in t; \\ cost(p_{min}), & \text{if there is an undirected path in } \mathcal{SH} \text{ between} \\ & w_1, w_2 \text{ and } p_{min} \text{ is the least cost such path;} \\ \infty, & \text{otherwise.} \end{cases}$$

It is easy to see that given any $\Sigma$-hierarchy, $\mathcal{SH} =_{def} (T, E, \wp)$, the distance function, $d_{\mathcal{SH}}$ induced by it is well defined and satisfies the triangle inequality.

**Example 2.3.1 (Distances in Verb and Noun-Term Hierarchies)**
*Consider the verb and noun-term hierarchies in Figures 2.4 on the facing page and 2.5 on page 36 respectively. In these figures, the weights are shown on the arcs connecting nodes in the hierarchy. All edges with no explicitly marked weights are assumed to have 1 as their weight. Consider the terms* compute *and* adjust *in the verb hierarchy of Figure 2.4 on the facing page. The distance between these two terms is given by*

$$d_{\mathcal{SH}} = cost(\text{compute}, \text{change}, \text{adjust}) = 2.$$

*Here, (*compute,change,adjust*) is the unique optimal path between* compute *and* adjust*. As another example, consider the terms* product *and* shoes(leather) *in the noun-term hierarchy of Figure 2.5.*

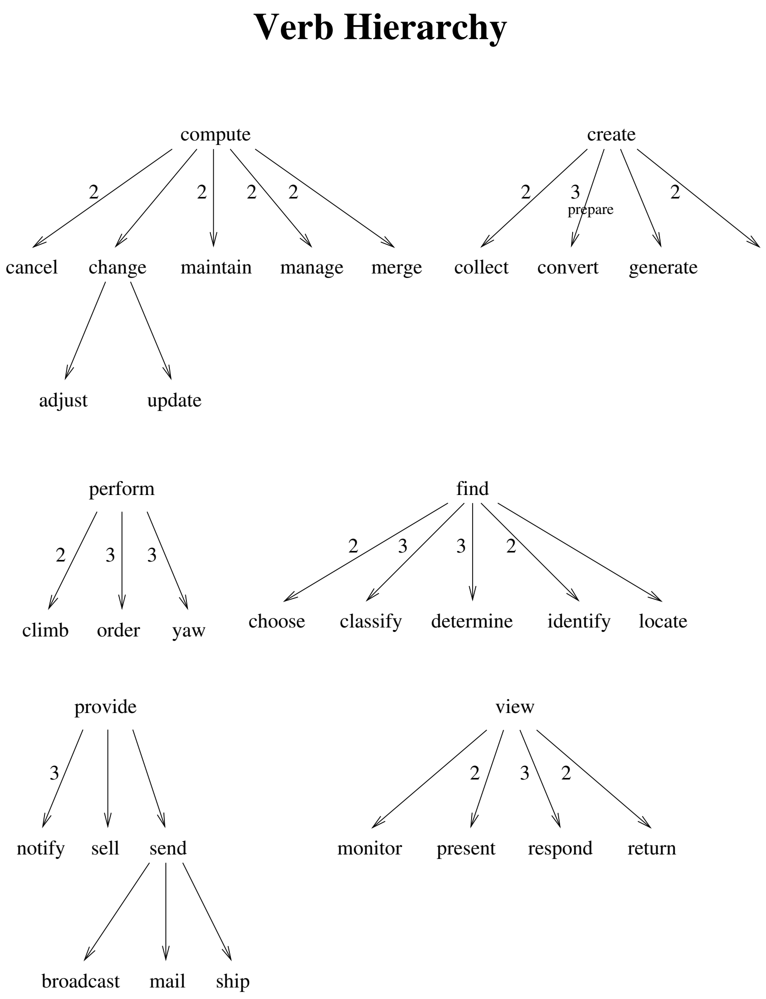



Figure 2.4: Example Verb Hierarchy (Missing Edge Labels are 1)

*The distance between these two terms is given by*

$$d_{\mathcal{SH}} = cost(product, clothing, shoes, shoes(leather)) = 5.$$

### 2.3.2 Steps in Registering an Agent

When the creator of an agent registers the agent, he interacts with the *IMPACT* server's *GUI* interface. Our current *IMPACT* server interface is a Java interface (Horstmann and Cornell 1997) which means that creators of agents can register their agents from any Java-enabled web browser. Figure 2.3 on page 31 shows a screen shot of this interface.

Information about agents is maintained in an AgentTable, which contains the following fields:

**agentName:** Name of the agent.

**passwd:** Password used when registering the agent. Agent information can not be modified unless the correct password is supplied.

**type:** The type of information contained in the *descr* field (described below). Some valid types include `HTML` and `English`.

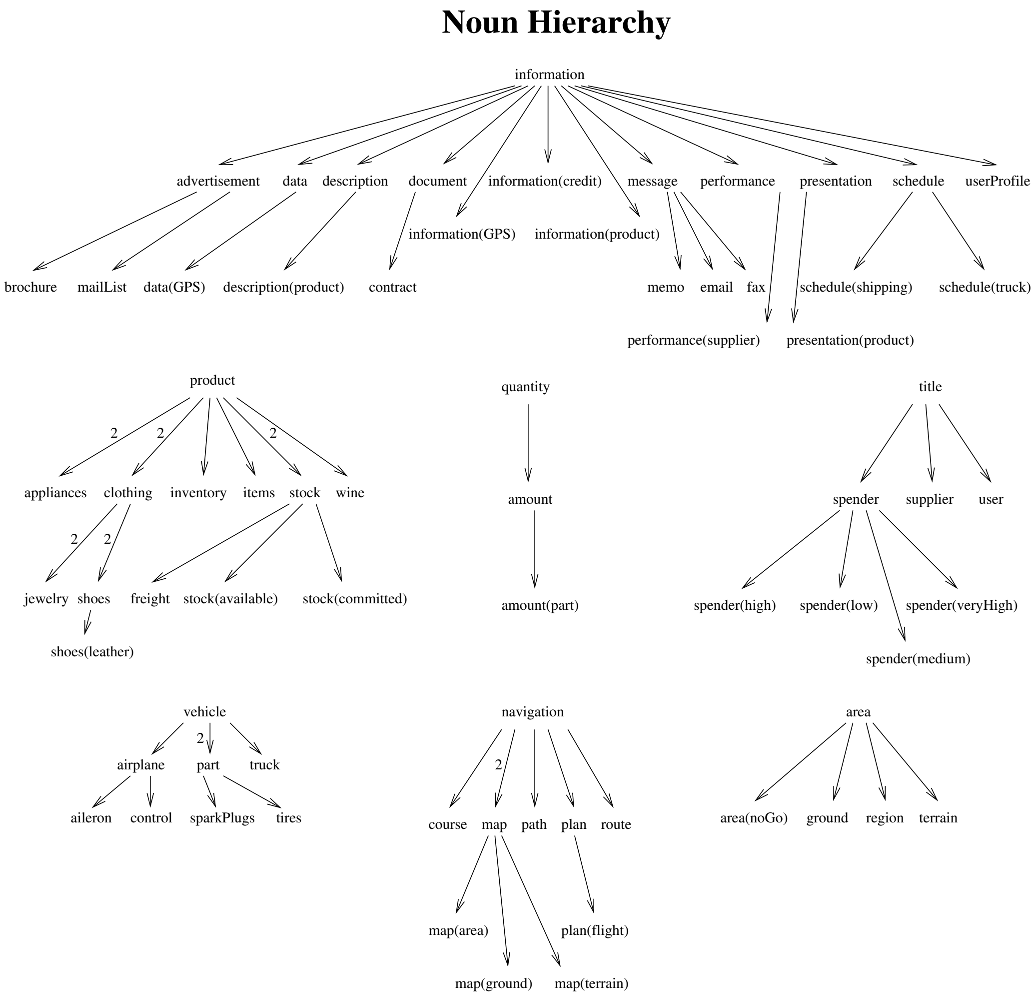



Figure 2.5: Example Noun-term Hierarchy

**descr:** A description of the agent. The format of this description depends on the *type* field mentioned above. For instance, if the type is `HTML`, this field's data will contain a URL. Alternatively, if the type is `English`, this field's data will contain a description of the agent in English, which can be displayed to users but which should not be parsed.

**allowed:** Indicates which groups of users are allowed to see information about the agent.

An agent designer goes through the following steps to add an agent:

1. *Specifying the name*: First, the agent designer must choose the name for his agent, as well as information on how other agents may connect to the agent. The designer may click the *View Agents* button to see descriptions for all agents currently in use. If the *Agent Name* or *Password* fields are given some values before clicking on *View Agents*, only agents matching the given name or using the given password will be displayed. Figure 2.3 on page 31 shows this interface.

2. *Security Conditions*: Once the agent designer has chosen a name, he should enter the desired values for *Agent Name*, *Password*, *Agent Type*, *Description*, and *Allowed*. These directly

correspond to the fields of the AgentTable described above. For convenience, designers can use the *Select Type* button to choose an existing type from a pull-down menu.

3. *Registering*: Now, to register the agent, the designer should click on the *Add Agent* button. If the agent name is unique, the new agent is added to AgentTable. If the agent name was already in use and the supplied password matches the one in the AgentTable, the type, description, and allowed fields are updated.

Each agent in an AgentTable may provide a number of services. These are maintained in a ServiceTable which contains the following fields:

**agentName:** Name of the agent which provides a service. This name must match some name in AgentTable.

**verbId:** The unique identifier for a node in the verb hierarchy.

**nounId:** The unique identifier for a node in the noun-term hierarchy.

**allowed:** Indicates which groups of users are allowed to see information about the service. Thus, a service is only accessible to users who satisfy the conditions in both `AgentTable.allowed` and `ServiceTable.allowed`.

This design ensures that each service's verb and noun-term appear in their relevant hierarchies. An agent designer goes through the following steps when adding a service for her agent.

1. First, the agent designer must specify the name of the service. To do so, she may click the *View Services* button to see which services are currently in use. If the *Agent Name* field is given some value before clicking this button, only services for the agent matching the given name are returned.

   Alternatively, she may browse the verb and noun-term hierarchies maintained by the *IMPACT* server, and/or query the hierarchy. For instance, if the agent designer wants to declare a service whose name is *update*: *stock*, she may want to check if one or more terms similar to *stock* are already in use. The noun term hierarchy may already contain terms such as *inventory* and in a case like this, the agent creator may wish to use this term instead of the term stock. Figure 2.6 on the next page shows the agent browsing a hierarchy.

   Hierarchies can be visualized by relating them to UNIX directory structures. For instance, "/a/b/c" indicates that node *c* is a child of node *b*, and node *b* is a child of root node *a*. In general, all root nodes can be considered children of a special node "/". For the verb hierarchy shown in Figure 2.6 on the following page, the Location field indicates the current node *n* while the listbox below this field gives the names and distances (from *n*) for all children of *n*. Here, if the user clicked on "classify", the Location field would contain "/find/classify" and the listbox would contain the names of "/find/classify"'s children. Alternatively, if the user clicked on "../" instead of "classify", the Location field would contain "/" and the listbox would contain the names of all the verb hierarchy root nodes. The noun-term hierarchy can be traversed in a similar way.

2. For each of the above services, the agent may wish to provide *security* conditions. This information tells the *IMPACT* server that the fact that the agent provides a service may only be disclosed to agents that satisfy the security conditions. For instance, in the CHAIN example, the fact that the `profiling` agent provides a *classify*: *user* service may be something that should be disclosed only to other agents owned by the department store in question and to employees of the department store.

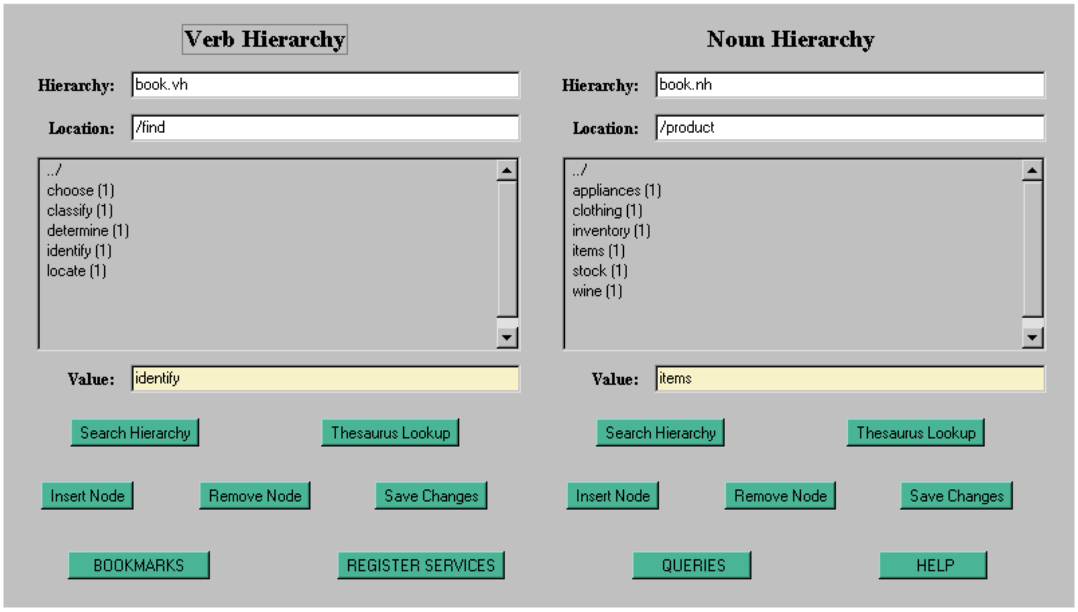



Figure 2.6: Hierarchy Browsing Screen Dump

3. Finally, to add the service, the designer should fill in the *Agent Name* field and click on the *Add Service* button. The *IMPACT* server adds the new service to ServiceTable.

Note that *IMPACT* also allows services and agents to be removed if the user enters the appropriate values into the *Agent Name* and *Password* fields. When an agent is removed from AgentTable, all of its services are also removed from ServiceTable. To help prevent accidental erasure, a confirmation box is used to inform the user of all services which will be removed along with their agent.

### 2.3.3 Thesaurus Server

We have built a thesaurus server on top of a commercial thesaurus system (the *ThesDB* Thesaurus Engine from Wintertree Software). The thesaurus server allows the owner of a new agent to browse a thesaurus and find words similar to the ones he is using to describe services. This server supports only one type of operation invoked by external clients. The client provides a word as input, and requests all synonyms as output. The thesaurus server, in addition to providing synonyms as output, "marks" those synonyms that appear in one of the two hierarchies (verb, noun-term). See Figure 2.7 on the next page for an example.

The thesaurus server can be accessed directly or through the registration server—in the latter case, a graphical user interface is available for human users.

### 2.3.4 Yellow Pages

At any given point in time, the *IMPACT* server receives zero, one or many requests from agents. We have described above the steps to be followed in registering an agent with the *IMPACT* server. The data structures created and managed by the registration server are used by the yellow pages server to provide *two* types of services.

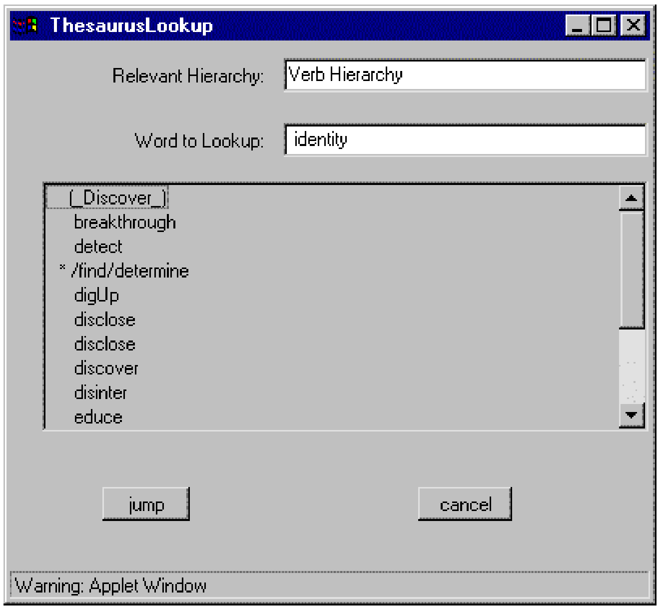



Figure 2.7: Thesaurus Screen Dump

**Nearest Neighbor Retrievals:** An agent $a$ might send a request to the Yellow Pages Server requesting information on the agents that provide services that most "closely" match a service $s_{req}$ that agent $a$ is seeking. Furthermore, agent $a$ might want the $k$ "best" matches. If there is some underlying *metric* on the space of service descriptions, then we are interested in finding the agents that provide the $k$-nearest neighboring services with respect to the requested service, $s_{req}$. In Chapter 3, we will propose a formal service description language, and show that the notion of distance on hierarchies described in this chapter may be extended to a metric on service descriptions.

**Range Retrievals:** Alternatively, agent $a$ might send a request to the Yellow Pages server requesting information on the agents that provide services that are within some "distance" of the requested service, $s_{req}$. In Chapter 3, we will show how this may be accomplished.

### 2.3.5 Synchronization Component

One problem with the use of *IMPACT* servers is that they may become a performance bottleneck. In order to avoid this, we allow multiple, mirrored copies of an *IMPACT* server to be deployed at different network sites. This solves the bottleneck problem, but raises the problem of consistency across the mirrored servers. The problem of replicated data management has been addressed by many researchers in the database community (Silberschatz, Korth, and Sudarshan 1997; Date 1995; Abbadi, Skeen, and Cristian 1985; Thomas 1979; Breibart and Korth 1997; Gray, Helland, O'Neil, and Shasha 1996; Holler 1981). Numerous algorithms have been proposed including primary copy, timestamping, majority voting, and quorum consensus. Except for timestamping algorithms, the others are based on distributed locking protocols and guarantee one-copy serializability (Abbadi, Skeen, and Cristian 1985). The number of messages exchanged in these algorithms is considerable. Moreover, one-copy serializability is not required in our system. As a result, we decided to deploy a version of the timestamping algorithms.

To ensure that all servers are accessing the same data, we have introduced a synchronization module. Users and agents do not access the synchronization module. Every time one copy of data structures maintained by an *IMPACT* server is updated, these updates are time-stamped and propagated to all the other servers. Each server incorporates the updates according to the time-stamps. If a server performs a local update before it should have incorporated a remote update, a rollback is performed as in classical databases (Silberschatz, Korth, and Sudarshan 1997).

Notice that the data structures of the *IMPACT* server are only updated when a new agent (or a new service) is added to an existing agent's service repertoire. As the use of existing agents and interactions between existing agents is typically much more frequent than such new agent/service introductions, this is not expected to place much burden on the system.

## 2.4 Related Work

During the last few years, there have been several attempts to define what an agent is — (Russell and Norvig 1995)[p. 33]; (Wooldridge and Jennings 1995; Franklin and Graesser 1997; Hayes-Roth 1995; Etzioni and Weld 1995; Moulin and Chaib-Draa 1996; Foner 1993). For example, Oren Etzioni (Etzioni and Weld 1995) provide the following characterization of agents (Etzioni and Weld 1995):

**Autonomy:** An agent must be able to take initiative and exercise a non-trivial degree of control over its own actions. It needs to be goal-oriented, collaborative, and flexible, and to decide by itself when to act.

**Temporal continuity:** An agent is a continuously running process.

**Communicability:** An agent is able to engage in complex communication with other agents, including people.

**Adaptivity:** An agent automatically customizes itself to the preferences of its user and to changes in the environment.

We agree that the above characterizations are useful in describing *intelligent* agent. However, our definition of an agent is wider, and allows agents to have a wide range of intelligence—agents can be dumb (e.g., sensor agents), a bit more intelligent (e.g., databases and other data retrieval agents), or smarter (e.g., agents that learn and/or adapt, etc.), or even smarter (e.g. agents that perform a cycle of sophisticated learning and planning activities). For example, consider a database agent that provides valuable access to a database system. Such an agent may not satisfy the autonomy criterion listed above, but yet provides a useful service. Furthermore, we may have Java applets that perform a useful function—yet, such agents may not be adaptive. They may do one thing well, but may not really adapt much to the user(s) involved and/or the environment. In addition, we specify how a program $P$ can be agentized. We believe our approach is the right way to go—requiring that all agents be "intelligent" according to the above criteria is too restrictive, and eliminates an extremely large percentage of useful programs in the world today. However, all agent infrastructures must be capable of deploying agents with the four properties mentioned above, and *IMPACT*'s architecture will support the creation of such smart agents.

Moulin and Chaib-Draa (1996) distinguish between artificial agents (software modules) and human agents (users). They propose that artificial agents should ideally possess several abilities: perception and interpretation of incoming data and messages, reasoning based upon their beliefs, decision making (goal selection, solving goal interactions, reasoning on intentions), planning, and the ability to execute plans including message passing. In this book we only discuss artificial agents and provide formal theories and software for building agents with the above capabilities. We also study additional agent capabilities such as service description and enforcing security policies.

There are two aspects to the development of agent architectures: what is the architecture of each agent and how do these architectures interconnect to form an overall multiagent framework? There are many approaches to the development of a single agent. These approaches were divided by Wooldrige and Jennings into three main categories (Wooldridge and Jennings 1995) (see also the discussion in Section 1.4):

**Deliberative:** A *deliberative agent architecture* is one which contains an explicitly represented, symbolic model of the world, and in which decisions (for example about what actions to perform) are made via logical (or at least pseudo-logical) reasoning, based on pattern matching and symbolic manipulations. Examples of such architecture includes the Intelligent Resource-bounded Machine Architecture (*IRMA*) (Bratman, Israel, and Pollack 1988), *HOMER* (Vere and Bickmore 1990), Etzioni softbots for the *UNIX* environments (Etzioni, Lesh, and Segal 1994), Twok and Weld's information gathering agents (Twok and Weld 1996) and many others. The main criticism of this approach is that the computational complexity of symbol manipulation is very high and some key problems appear to be intractable.

**Reactive:** Such architectures are usually defined as those that do not include any kind of central symbolic world model and do not use any complex symbolic reasoning. One of the first architectures of this type is Brooks's *subsumption architecture* (Brooks 1986). Others include Rosenschein and Kaelbling's situated automata (Rosenschein 1985) and Maes's Agent Network Architecture (Maes 1989). These types of agents work efficiently when they are faced with "routine" activities.

**Hybrid:** Several researchers have suggested that neither a completely deliberative nor a completely reactive approach is suitable for building agents. They use *hybrid* systems which attempt to combine the deliberate and the reactive approaches. Some examples include the *PRS* architecture (Georgeff and Lansky 1987), *TouringMachine* (Ferguson 1992) and *AIS* (Hayes-Roth 1995).

*IMPACT* is populated with different agents, possibly having different architectures of different types. The agent architecture proposed in this book is a hybrid architecture. As agents in *IMPACT* can be built on top of arbitrary pieces of code, it follows immediately that agents in *IMPACT* can be built on top of agents in other agent frameworks such as *PRS* architecture (Georgeff and Lansky 1987), *TouringMachine* (Ferguson 1992) and *AIS* (Hayes-Roth 1995).

The second aspect of developing agent architecture—how do agents interconnect to form an overall multiagent framework—has also been studied extensively. Bond and Gasser (Bond and Gasser 1988) divide multiagent systems into two main categories:

1. DPS (Distributed Problem Solving): This category considers how the work of solving a particular problem can be divided among several agents. Each agent is intelligent, but they have all a common goal and common preferences. DPS systems are described, for example, in (Smith and Davis 1983; Durfee 1988; Shehory and Kraus 1998).

2. MAS (Multi-Agent systems): coordinates intelligent behavior among a collection of autonomous intelligent agents. Each agent may have different goals and different interests, which may conflict with the interests of other agents in the system. MASs are discussed, for example, in (Sycara 1987; Rosenschein and Zlotkin 1994; Kraus, Wilkenfeld, and Zlotkin 1995).

These classes represent two extreme poles of the spectrum in multiagent research. Our research falls closer to the MAS pole, as we consider autonomous agents, possibly developed by different programmers or organizations. However, in our framework, sub-groups of agents (e.g., the agents in the supply chain example) may cooperate and form a DPS (sub)-system. A similar approach is taken in the *RETZINA* project (Sycara and Zeng 1996a) which is also a generic infrastructure for agent systems. However, in all current implementations of *RETZINA* agents are assumed to be cooperative.

# Chapter 3

# Service Description Language

When an agent wishes to use a service, one of two situations may exist. In the first case, the agent knows which agent provides the desired service. The second case occurs when an agent does not know which agents, if any, provide the service it needs. In both cases, the agent needs to know what inputs the potential service provider agent needs in order to provide the service, and what outputs the service provider returns. This chapter presents *IMPACT*'s HTML-like Service Description Language (SDL), which is used by the agents to describe their services. It also describes in detail the *IMPACT* Yellow Pages Server, which provides matchmaking services to *IMPACT* agents.

## 3.1 Agent Service Description Language

The specification of a single service consists of the following components:

**Service Name:** This is a *verb: noun(noun)* expression describing the service which is defined in Definition 2.3.1 on page 31. For example, as discussed in Section 2.2.3, *monitor: available-stock* is the name of a service provided by the `supplier` agent in the CHAIN example.

**Inputs:** Services assume that the users of the service will provide zero or more inputs. The service description must include a specification of what inputs are expected and which of these inputs are mandatory. This specification must provide an "English" name for each input, as well as a semantic type for that input. For example, *Amount:* `Integer` specifies that we have an input called *Amount* of type `Integer` and *Part:* `PartName` specifies that we have an input called *Part* of type `PartName` (which could be an enumerated type).

**Outputs:** Each service must specify the outputs that it provides and each output is specified in the same way as an input.

**Attributes:** In addition, services may have attributes associated with them. Examples of such attributes include cost (for using the service), average response time for requests to that service, etc.

When an agent wishes to find agents that offer a desired service, it obtains the service name and identity of the agent involved from the yellow pages server. However, the rest of the description can be obtained directly from the agent that provides the service. This strategy is efficient as it reduces the load on the yellow pages server by distributing the workload to the agents.

Before defining input specifications, we need to formally define types.

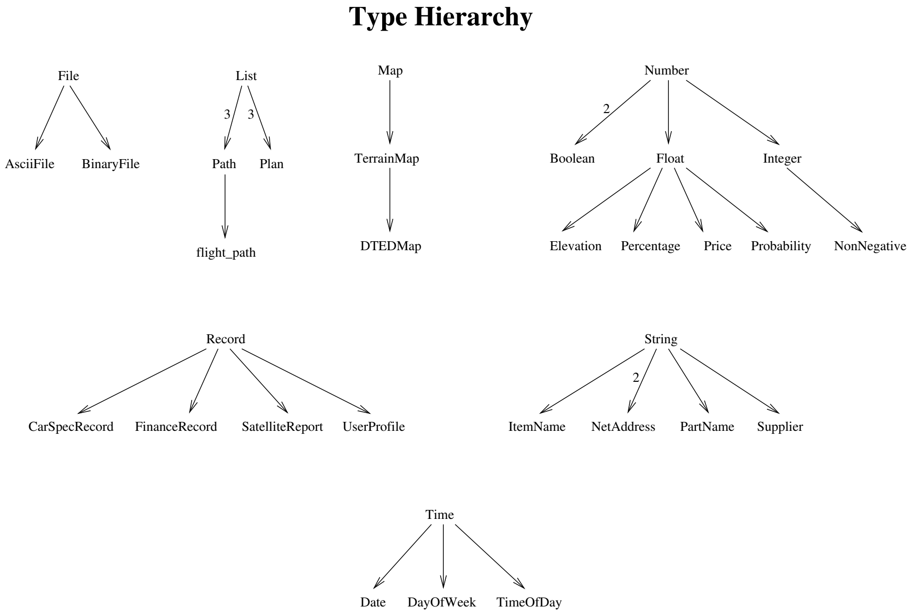



Figure 3.1: Example Type Hierarchy

**Definition 3.1.1 (Type/Type Hierarchy $(\mathcal{T}, \leq)$)**
*A* type $\tau$ *is a set whose elements are called "values" of* $\tau$. *The pair* $(\mathcal{T}, \leq)$ *is called a* type hierarchy *if* $\mathcal{T}$ *is a set of types and* $\leq$ *is a partial ordering on* $\mathcal{T}$.

Figure 3.1 provides a hierarchy associated with the three motivating examples.

**Definition 3.1.2 (Set of Type Variables $V_{\mathcal{T}}$)**
*Associated with any type hierarchy* $(\mathcal{T}, \leq)$, *is a set* $V_{\mathcal{T}}$ *of symbols called* type variables.

Intuitively, a type variable ranges over the values of a given type. For instance, `PartName` may be a type variable ranging over strings. When specifying the inputs required to invoke a service, we need to specify variables and their associated types. This is done in the usual way, as defined below.

**Definition 3.1.3 (Items $s\!:\!\tau$)**
*If* $s$ *is a variable ranging over objects of type* $\tau$, *then* $s\!:\!\tau$ *is called an* item.

For example, *Part:*`String`, *Document:*`AsciiFile`, and *Addr:*`NetAddress` are all valid items if one assumes that the types `String`, `AsciiFile` and `NetAddress` are all well defined. As is common in most imperative programming languages, the syntactic object $s\!:\!\tau$ may be read as saying *"the variable s may assume values drawn from the type* $\tau$*"*.

Each service requires zero, one, or more inputs. Some of these inputs are mandatory (i.e. the service cannot be provided if these inputs are not specified), while others are discretionary (they are not required for the service to be provided, but their provision may either increase the efficiency of the provided service, or the quality of the service). For example, the service *create:plan(flight)*, defined in the CFIT example, may require that the *location* and the *path* fields be filled, but may not require a *no_go* field to be filled in. This is captured in the following definition.

**Definition 3.1.4 (Item Atom)**
*If* $s\!:\!\tau$ *is an item, then* $\langle \mathtt{I} \rangle s\!:\!\tau \langle \backslash \mathtt{I} \rangle$ *(resp.* $\langle \mathtt{MI} \rangle s\!:\!\tau \langle \backslash \mathtt{MI} \rangle$*) is called an* input *(resp.* mandatory input*) item atom, and* $\langle \mathtt{O} \rangle s\!:\!\tau \langle \backslash \mathtt{O} \rangle$ *is called an* output *item atom.*

# Chapter 4

# Accessing Legacy Data and Software

The design of data structures for an application depends fundamentally on the kind of data manipulated by that application, by the kinds of operations to be performed on the data, by the relative frequency of those operations, and by user communities' expectations on the performance of these different operations. Computer scientists design data structures and algorithms based on the above criteria, so as to efficiently support operations on the data structures. Under no circumstances should a definition of agents limit the choice of data structures and algorithms that an application designer must use.

Thus, the ability to build agents on top of arbitrary pieces of code is critical to the agent enterprise as a whole. For instance, in the CHAIN example, we might wish to build the supplier agents on top of an existing commercial relational DBMS system. Likewise, in the case of the terrain agent in the CFIT example, we might wish to build this agent on top of existing US military terrain reasoning software. In addition, there may be agents that are not built on top of a single piece of existing software, but which access a set of software packages. For instance, the Product Database agent productDB in the CHAIN example may access some file structures, as well as some databases.

In this chapter, we will define a single unified front-end to a set of disparate heterogeneous data sources and software packages, on top of which agents may be built. We will first start with an abstract definition of software code, and show how most packages may be viewed as instances of this abstract definition. An agent may be built on top of one or more such pieces of software code. We will introduce the concept of an agent state that describes what the agent's data structures are populated with at a given instant of time. Then we will introduce the concept of a code call condition which provides a generic query language that can span multiple abstractions of software code. We then introduce a specific code domain called a msgbox that can be used by different agents for messaging. Integrity constraints are then introduced as constraints that limit what the agent state might look like. We then show how the concept of a service description defined in Chapter 3 may be implemented via a concept called a service description program that uses the above concepts.

## 4.1 Software Code Abstractions

In this section, we focus on the *internal* data managed by the software code underlying an agent.

**Definition 4.1.1 (Software Code $\mathcal{S} = (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$)**
*We may characterize the code on top of which an agent is built as a triple $\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ where:*

1. *$\mathcal{T}_{\mathcal{S}}$ is the set of all data types managed by $\mathcal{S}$,*

2. *$\mathcal{F}_{\mathcal{S}}$ is a set of predefined functions which makes access to the data objects managed by the agent available to external processes, and*

3. *$\mathcal{C}_{\mathcal{S}}$ is a set of type composition operations. A type composition operator is a partial $n$-ary function $c$ which takes as input types $\tau_1,\ldots,\tau_n$ and yields as a result a type $c(\tau_1,\ldots,\tau_n)$. As $c$ is a partial function, $c$ may only be defined for certain arguments $\tau_1,\ldots,\tau_n$, i.e., $c$ is not necessarily applicable on arbitrary types.*

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* (Cattell, R. G. G., et al. 1997). Almost all existing servers used in real systems, as well as most commercial packages available on the market are instances of the above definition. The same is true of commercial standards such as the Object Data Management Group's *ODMG* standard (Cattell, R. G. G., et al. 1997), the *CORBA* framework (OMG 1998a; Siegal 1996) and Microsoft's COM/OLE (Microsoft 1999) (http://www.microsoft.com/com/comPapers.asp).

Intuitively, $\mathcal{T}_{\mathcal{S}}$ is the set of all data types that are managed by the agent. $\mathcal{F}_{\mathcal{S}}$ intuitively represents the set of all function calls supported by the package $\mathcal{S}$'s application programmer interface (*API*). $\mathcal{C}_{\mathcal{S}}$ the set of ways of creating new data types from existing data types.

Given a software package $\mathcal{S}$, we use the notation $\mathcal{T}_{\mathcal{S}}{}^{\star}$ to denote the *closure* of $\mathcal{T}_{\mathcal{S}}$ under the operations in $\mathcal{C}_{\mathcal{S}}$. In order to formally define this notion, we introduce the following definition.

**Definition 4.1.2** **($\mathcal{C}_{\mathcal{S}}(\mathcal{T})$ and $\mathcal{T}_{\mathcal{S}}{}^{\star}$)**
**a)** *Given a set $\mathcal{T}$ of types, we define*

$$\mathcal{C}_{\mathcal{S}}(\mathcal{T}) =_{def} \mathcal{T} \;\cup\; \{\tau : \text{ there exists an } n\text{-ary composition operator } c \in \mathcal{C}_{\mathcal{S}} \text{ and types } \tau_1,\ldots,\tau_n \in \mathcal{T} \text{ such that } c(\tau_1,\ldots,\tau_n) = \tau\}.$$

**b)** *We define $\mathcal{T}_{\mathcal{S}}{}^{\star}$ as follows:*

$$\begin{array}{lll} \mathcal{T}_{\mathcal{S}}{}^{0} & =_{def} & \mathcal{T}_{\mathcal{S}}, \\ \mathcal{T}_{\mathcal{S}}{}^{i+1} & =_{def} & \mathcal{C}_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}{}^{i}), \\ \mathcal{T}_{\mathcal{S}}{}^{\star} & =_{def} & \bigcup_{i\in\mathbb{N}} \mathcal{T}_{\mathcal{S}}{}^{i}. \end{array}$$

Intuitively, $\mathcal{T}_{\mathcal{S}}{}^{\star}$ represents the set of all possible types that can be produced by repeatedly applying the composition operations in $\mathcal{C}_{\mathcal{S}}$ on the base types in $\mathcal{T}$. Let us return to the CFIT, CHAIN and STORE examples to see how some of the software packages within them may be captured by this definition.

### 4.1.1 CHAIN Revisited

Consider the two Supplier agents in the CHAIN example. Each of these agents may manage the set

$$\mathcal{T}_{\mathcal{S}} =_{def} \{\texttt{Integer}, \texttt{Location}, \texttt{String}, \texttt{Date}, \texttt{OrderLog}, \texttt{Stock}\}$$

of types. Here, `OrderLog` is a relation having the schema

$$(\mathit{client}/\texttt{string},\ \mathit{amount}/\texttt{Integer},\ \mathit{part\_id}/\texttt{String},\ \mathit{method}/\texttt{String},\ \mathit{src}/\texttt{Location},\ \mathit{dest}/\texttt{Location},\ \mathit{pickup\_st}/\texttt{date},\ \mathit{pickup\_et}/\texttt{date}),$$

while `Stock` is a relation having the schema (*amount*/**Integer**, *part_id*/**String**). `Location` is an enumerated type containing city names. In addition, $\mathcal{F}_\mathcal{S}$ might consist of the functions:

- *monitorStock*(*Amount*/**Integer**, *Part_id*/**String**) of type `String`.
  This function returns either `amount_available` or `amount_not_available`, which are status strings having the obvious meaning.

- *shipFreight*(*Amount*/**Integer**, *Part_id*/**String**, *method*/**String**, *Src*/**Location**, *Dest*/**Location**).
  This function, when executed, updates the order log and logs information about the order, together with information on (i) the earliest time the order will be ready for shipping, and (ii) the latest time by which the order must be picked up by the shipping vendor. Notice that this does *not* mean that the shipment will in fact be picked up by the `airplane` agent at that time. It just means that these are the constraints that will be used by the `supplier` agent in its negotiations with the `shipping` agent.

- *updateStock*(*Amount*/**Integer**, *Part_id*/**String**).
  This function, when executed, updates the inventory of the Supplier. For example, if the supplier had 500 items of a particular part on hand, then the fact that 200 of these parts have been committed means that the stock has dropped to 300.

Finally, in this case, $\mathcal{C}_\mathcal{S}$ might consist of the operations of projection of attributes of a relation and cartesian products of relations. Thus, $\mathcal{T}_\mathcal{S}{}^\star$ consists not only of all the above data types, but also of

1. sub-records of the above data types such as a relation having the schema (*amount*/**Integer**, *dest*/**Location**) derived from the `OrderLog` relation,
2. cartesian products of types, and
3. mixes involving the above two operations.

### 4.1.2 STORE Revisited

Consider the `profiling` agent in the STORE example. Here, the data types might consist of the set

$$\mathcal{T}_\mathcal{S} =_{def} \{\texttt{String}, \texttt{UserProfile}\},$$

while, $\mathcal{F}_\mathcal{S}$ might contain the following function:

- *classifyUser*(*Ssn*/**String**) of type `UserProfile`.
  This function takes as input, a person's social security number, and returns as output a value such as `spender(high)`. The output type of this function is called `UserProfile`, which is an enumerated type consisting of `spender(high)` and similar strings.

Moreover, the `credit` agent might have the following data types;

$$\mathcal{T}_\mathcal{S} =_{def} \{\texttt{FinanceRecord}, \texttt{String}\},$$

and $\mathcal{F}_\mathcal{S}$ might contain the following function:

- *provideCreditInfo*(*Ssn*/**String**, *DetailLevel*/**String**) of type `FinanceRecord`.
  This function takes as input a person's social security number and a string `high`, `medium`, or `low`). It returns credit information about person *ssn* in a record of Type `FinanceRecord`, whose fields depend on the detail level requested.

Finally, let $\mathcal{C}_\mathcal{S}$ be

$$\{\pi_{f_1,\dots,f_k}(\texttt{FinanceRecord}) : \ f_i, f_j \text{ are fields of } \texttt{FinanceRecord} \wedge i \neq j \Rightarrow f_i \neq f_j\}.$$

Thus, in this example, $\mathcal{T}_\mathcal{S}{}^\star$ contains every type in $\mathcal{T}_\mathcal{S}$ plus the `FinanceRecord` type which consists of all possible projections on the fields of type `FinanceRecord`. The type `FinanceRecord` is assumed to be a named type among these projections.

### 4.1.3 CFIT Revisited

Let us now consider the autoPilot agent and the gps agent in the CFIT example. Let the set of types by

$$\mathcal{T}_\mathcal{S} =_{def} \{\texttt{Map}, \texttt{Path}, \texttt{Plan}, \texttt{SatelliteReport}\}.$$

Here, the maps in question are a special class of maps called *DTED Digital Terrain Elevation Data* that specify the elevations of different regions of the world. (For financial reasons, our implementation only uses *DTED* data for the continental United States—obtaining such data for the whole world is extremely expensive.) The type `SatelliteReport` may itself be a complex record type containing fields: `height` (height of ground), `sat_id` (identity of satellite providing the report), `dist` (current distance between the plane and the ground), and `2dloc` (current `x,y` location) which in turn has fields `x` and `y`.

Suppose the autoPilot agent's associated set of functions $\mathcal{F}_\mathcal{S}$ contains:

- $\mathit{createFlightPlan}(\mathit{Location}/_{\texttt{Map}}, \mathit{Flight\_route}/_{\texttt{Path}}, \mathit{Nogo}/_{\texttt{Map}})$ of type `Plan`.
  Intuitively, this function takes as input, the *actual* location of the plane, the flight path of the plane, and a map depicting *no_go* volumes. Intuitively, *no_go* volumes are regions of space where the plane is not supposed to fly. For example, a mountain, surrounded by an envelope of 1000 feet is an example of such a *no_go* volume. This functions returns as output a modified flight plan that avoids the *no_go* volumes.

Moreover, the $\mathcal{F}_\mathcal{S}$ of the gps might contain the following function:

- $\mathit{mergeGPSData}(\mathit{Data1}/_{\texttt{SatelliteReport}}, \mathit{Data2}/_{\texttt{SatelliteReport}})$ of type `SatelliteReport`.
  This function takes as input satellite report data from two or more satellites, and merges them into a single report that can be used to pinpoint the location of the plane. Typically, this function would be iteratively used by the gps agent for this pinpointing to occur.

Let $\mathcal{C}_\mathcal{S}$ be the singleton set

$$\{\texttt{list of SatelliteReport}\}.$$

Thus, $\mathcal{T}_\mathcal{S}{}^\star$ contains every type in $\mathcal{T}_\mathcal{S}$, plus the type which consists of all lists of type `SatelliteReport`; for future reference, let us name this type `SatelliteReport`.

### 4.1.4 State of an Agent

At any given point $t$ in time, the *state of an agent* will refer to a set $O_\mathcal{S}(t)$ 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. Throughout this book we will assume that except for appending messages to an agent $\mathsf{a}$'s mailbox, another agent $\mathsf{b}$ cannot directly change $\mathsf{a}$'s state. However, it might do so indirectly by shipping the other agent a message issuing a change request. The precise definitions of messages and message management will be given in Section 4.3, while details of actions and action management will be described in Chapter 6.

## 4.2 Code Call Conditions

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

Intuitively, code call conditions are logical expressions that access the data of heterogeneous software sources using the pre-existing external *API* (application program interface) function calls provided by the software package in question. In other words, the language of code-call conditions is layered on top of the physical data structures and implementation within a specific package.

### 4.2.1 Variables

Suppose we consider a body $\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ of software code. Given any type $\tau \in \mathcal{T}_{\mathcal{S}}$, we will assume that there is a set *root*$(\tau)$ of "root" variable symbols ranging over $\tau$. Such "root" variables will be used in the construction of code calls.

However, consider a complex type $\tau$, and suppose $\tau$ is a complex record type having fields $\mathtt{f_1}, \ldots, \mathtt{f_n}$. Then, for every variable of type $\tau$, we require that $\mathtt{X.f_i}$ be a variable of type $\tau_i$ where $\tau_i$ is the type of field $\mathtt{f_i}$. In the same vein, if $\mathtt{f_i}$ itself has a sub-field $\mathtt{g}$ of type $\gamma$, then $\mathtt{X.f_i.g}$ is a variable of type $\gamma$, and so on. The variables, $\mathtt{X.f_i}$, $\mathtt{X.f_i.g}$, etc. are called *path variables*. For any path variable `Y` of the form `X.path`, where `X` is a root variable, we refer to `X` as the root of `Y`, denoted by *root*(`Y`); for technical convenience, *root*(`X`), where `X` is a root variable, refers to itself. To see the distinction between root variables and path variables, let us return to the CFIT example.

**Example 4.2.1 (CFIT Revisited)**
*Let* `X` *be a (root) variable of type* `SatelliteReport` *denoting the current location of an airplane. Then* `X.2dloc`, `X.2dloc.x`, `X.2dloc.y`, `X.height`, *and* `X.dist` *are path variables. For each of the path variables* `Y`, *root*(`Y`) = `X`. *Here,* `X.2dloc.x`, `X.2dloc.y`, *and* `X.height` *are of type* `Integer`, `X.2dloc`*'s type is a record of two* `Integer` *s, and* `X.dist` *is of type* `NonNegative`.

**Definition 4.2.1 (Variable Assignment)**
*An* assignment of objects to 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.*

We now return to the CFIT example to see an example assignment.

**Example 4.2.2 (CFIT Revisited)**
*A legal assignment may be*

$$(\mathtt{X.height} := 50, \mathtt{X.sat\_id} := \mathtt{iridium\_17}, \mathtt{X.dist} := 25, \mathtt{X.2dloc.x} := 3, \mathtt{X.2dloc.y} := -4).$$

*If the record is ordered as shown here, then we may abbreviate this assignment as (50,* `iridium_17`, *25,* $\langle 3, -4\rangle$*). Note however that*

$$(\mathtt{X.height} := 50, \mathtt{X.sat\_id} := \mathtt{iridium\_17}, \mathtt{X.dist} := -25, \mathtt{X.2dloc.x} := 3, \mathtt{X.2dloc.y} := -4)$$

# Chapter 5

# *IMPACT* Server Implementation

In Chapter 3, we have described the functions performed by an *IMPACT* server. In this chapter, we will provide a detailed description of how the *IMPACT* server is implemented. Readers who are not interested in the implementation can skip this entire chapter. Readers who only want a brief overview of the implementation can skip Section 5.2.

The architecture of the *IMPACT* server is shown in Figure 5.1. It contains two major components: the *adminImpact* component, and the *dbImpact* component. The *adminImpact* and *dbImpact* components use Unix sockets and the *TCP/IP* protocol (Wilder 1993) to communicate with clients. For *dbImpact*, the client is usually *adminImpact*. For *adminImpact*, clients can be graphical user interfaces (*GUI*s) or agents requesting services from an *IMPACT* server. Users indirectly communicate with *adminImpact* by using a *Tcl/Tk* or *Java* based *GUI*. These *GUI*s hide client related details (like id numbers) from the user.

Intuitively, *dbImpact* contains several classes. Each class is responsible for providing a set of related services. For instance, the *hierClass* can be used to query/modify the *verb*, *noun-term*, or *dataType* hierarchies, the *thesClass* allows queries to a *thesaurus*, and the *tableClass* provides services for querying/updating an *agentTable* or *serviceTable*.

Before clients can use hierarchies, thesauri, or agent/service tables, these entities must be initialized. After invoking a class initialization service such as *hier_init*, *thes_init*, or *db_init*, *dbImpact* responds with an identification (id) number. This number can be used as a *handle* for accessing these resources. Information such as *type* and *owner* is associated with each handle.

This helps prevent clients from using the wrong type of handle in the wrong place or from illegally using another client's handles.

Clients can directly communicate with *dbImpact* servers. Alternatively, if a client wants to avoid the complexity of maintaining the id numbers discussed above, the client can indirectly communicate with a *dbImpact* server by talking to an *adminImpact* server. Here, client requests are processed in the following way:

1. First, a client sends a request to an *adminImpact* server. This request can use names (like `NASA_verb_hierarchy`) instead of id numbers (like "3") to refer to resources.

2. *adminImpact* translates these names into ids and sends the request to a *dbImpact* server.

3. After processing this request, the *dbImpact* server sends responses to the *adminImpact* server.

4. The *adminImpact* server reads this response and updates its internal data structures if necessary.

5. Finally, the *adminImpact* server sends a (simplified) response to the client.

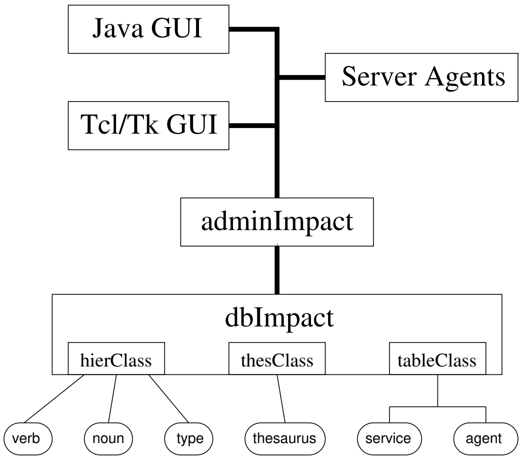



Figure 5.1: *IMPACT* server architecture

When an *adminImpact* or *dbImpact* server is started, a port is selected. Clients can then establish a connection by specifying the host name and port of a running server. Servers can handle multiple remote clients. Resource allocation to each client is carefully maintained so that memory can be freed if a connection is unexpectedly dropped.

Clients communicate with servers by sending request strings. The first word of these strings is always the name of the desired service. The remainder of these strings contains a space-separated list of arguments. After processing a request string, servers return a response string which begins with either `Ok` (indicating success), `Error` (indicating some error condition), or `Exit` (indicating that the connection should be dropped). If desired, servers can log all of these request/response strings.

When multiple requests arrive, servers put the strings for each request on a queue. These requests are then processed serially (one at a time). This simplifies the implementation of our operators.

For the remainder of this chapter, we shall begin by giving a brief description of the services offered by *dbImpact*. We shall then give the longer, more technical description for each service which includes information such as input parameters, output format, and examples. Finally, we shall consider how these services may be used to handle requests by an agent or a user.

## 5.1 Overview of *dbImpact* Services

This section provides a summary of the services offered by *dbImpact*. Note that all of these services are also provided by *adminImpact*.

1. Connection related services:
   (a) *tcp_echo* echos back to the client every word it sent in its request. This service can be used for testing connections.
   (b) *tcp_exit* informs the server that the client wants to disconnect. The server will respond with the `Exit` message.
2. Hierarchy creation, initialization, and termination (for using verb, noun-term, or dataType hierarchies):

(a) *hier_create* creates a new, blank hierarchy.
(b) *hier_init* initializes a hierarchy by reading its contents from a file.
(c) *hier_quit* frees memory associated with a hierarchy.

3. Hierarchy browsing/traversal (note that in our current implementation, hierarchies are stored as a forest of trees instead of a true DAG):

   (a) *hier_firstId* returns the lowest numbered *nodeId* in a hierarchy. In the implementation, each node in a hierarchy is assigned a unique *nodeId* number.
   (b) *hier_emptyId* returns **true** iff *nodeId* is empty (i.e., if the target node has been deleted).
   (c) *hier_lastId* returns the highest numbered *nodeId* in a hierarchy. Clients can quickly scan every valid *nodeId* by processing all non-empty nodes between the first and last *nodeId*.
   (d) *hier_getRoots* returns the *nodeId*s of all root nodes in a hierarchy.
   (e) *hier_getKids* returns the *nodeId*s of all (immediate) children of a given node. It also returns the costs/distances to each of these nodes.
   (f) *hier_getParent* returns the *nodeId* for the given node's parent (i.e., its immediate predecessor) or zero if the given node is a root node.
   (g) *hier_search* returns the *nodeId*s for every node that matches a given search string. Wild card searches are supported.

4. Id/path conversions:

   (a) *hier_getNames* returns the names associated with the given node. Usually, nodes will only have one name but we allow multiple names nonetheless.
   (b) *hier_getPath* returns the *path* (from a root node) associated with the given node. For instance, when using our sample noun-term hierarchy, the *path* to the node named `memo` would be `/information/message/memo`.
   (c) *hier_getNodeId* returns the *nodeId* associated with the given *path*. In other words, this service is the inverse of *hier_getPath*.

5. Hierarchy modification:

   (a) *hier_insert* inserts a new node into a hierarchy. The new node becomes the child of a given node $n$ (or a root node). For each child node $n'$ of $n$ (or for every root $n'$), the client can specify whether or not $n'$ should become a child of the new node. Note that the client may also specify the distances to each node that will be adjacent to the newly inserted node.
   (b) *hier_setCosts* allows clients to specify the distances between a node $n$ and all nodes adjacent to $n$. By using this function, we can change edge weights without deleting and reinserting nodes.
   (c) *hier_remove* removes a node $n$ (but not its children) from a hierarchy. This service also ensures that for each $n_1, n_2 \neq n$ in the hierarchy, the distance between $n_1$ and $n_2$ is not changed.
   (d) *hier_flush* flushes a hierarchy to disk. This is important because *hier_insert*, *hier_setCosts*, and *hier_remove* only modify the copy of the hierarchy that is located in memory.

6. Thesaurus functions:

(a) *thes_init* initializes a thesaurus so that it will use the given thesaurus file.

(b) *thes_quit* frees memory associated with a thesaurus.

(c) *thes_getCategories* returns the verb or noun categories associated with a given word $w$. Intuitively, each category represents a different meaning for $w$. Different categories will be returned when a client specifies different parts of speech.

(d) *thes_getSynonyms* returns a list of synonyms associated with the given category. Thus to get all synonyms of a word $w$, clients would first invoke *thes_getCategories* and then, for each answer returned, invoke *thes_getSynonyms*.

7. Db creation, initialization, and termination (for using agent/service tables):

(a) *db_create* creates new, blank, agents and service tables.

(b) *db_init* initializes a database connection.

(c) *db_quit* frees memory associated with a database connection.

8. Hier-Db queries:

(a) *query_distance* returns the shortest distance between two nodes.

(b) *query_nn* runs the $\mathit{find\_nn}(v, nt, k)$ algorithm where $v$ and $nt$ are nodes in a verb and noun-term hierarchy. Here, the $k$ nearest services to $\langle v, nt \rangle$ are returned.

(c) *query_range* runs the $\mathit{range\_nn}(v, nt, D)$ algorithm where $v$ and $nt$ are nodes in a verb and noun-term hierarchy. Here, all services whose composite distance from $\langle v, nt \rangle$ is less than or equal to $D$ are returned.

(d) *query_findSource* takes a verb $v$ and a noun-term $nt$ as input and returns as output the pair $\langle v, nt \rangle$ which most closely matches $\langle \mathsf{verb}, \mathit{noun\text{-}term} \rangle$. We also require $v$ and $nt$ to appear in their respective hierarchies. Note that in order to satisfy these constraints, *query_findSource* may have to consult other services such as *hier_search*, *thes_getSynonyms*, *query_distance*, and so on. Once we determine $v$ and $nt$, clients can use these as inputs to *query_nn* or *query_range*.

9. Db queries (viewing lists of *agent types*, *agents*, or *services*):

(a) *db_getAgentTypes* returns a list of all *agent types* which occur in an agent table. An *agent type* determines the format of an agent description <*agent_descr*>. Some sample *agent types* include `HTML`, `Hermes`, and `English`. Here, <*agent_descr*> may be a URL, a function within a Hermes mediator, or an English description of an agent.

(b) *db_getAgents* returns information for all agents using a given password.
If this password is `ALL`, information for all agents is returned. Here, an agent's information consists of its name, its allowed field, its agent type, and its agent descr.

(c) *db_getAgentInfo* returns information for the agent which matches a given name. This information has a format that is similar to the one of the output from *db_getAgents*.

(d) *db_getServices* returns information about all services which are offered by a given agent. This function can also be used to return all services offered by all agents. Service information consists of an agent name and *nodeId*s for the verb and noun-term used when registering the service.

# Chapter 6

# Agent Programs

In Chapter 4, we described a *code call* mechanism that allows us to build agents "on top" of arbitrary data structures. This is because the vast majority of useful software applications out there in the market were developed prior to the ongoing interest in agents. Furthermore, even in the future, different data structures will prove to be necessary for different applications, and for such packages to meaningfully interoperate with other such packages, we need to *agentize* them.

In this chapter, we will set up the basic infrastructure that decides how an agent will or should *act*. For instance, in the CFIT example, the satellite agent "wakes up" every $\Delta t$ units of time, broadcasts a location report, and goes back to "sleep" for another $\Delta t$ seconds. In contrast, in the case of the truck agent in the CHAIN example, the agent's work is initiated by receipt of a message from another agent that requires truck resources. In the STORE example, the credit agent may treat different agents differently, providing appropriate responses to authorized agents, and denying service to other agents. In each of these cases, the agent is making a decision on how to respond to changes in its environment (e.g., receipt of a message, ticking of the clock, etc.). Different agents use different policies to make such decisions. Yet, the creator or administrator of an agent will want his agent to adhere to certain principles—he must set up *behavioral guidelines* that his agent must obey. These guidelines will, in all likelihood, state that the agent is obliged to take certain actions under certain circumstances, and that it is forbidden from taking certain actions under other circumstances. In some cases, it may have discretion about what actions to take. The main aim of this chapter is to define a language called *Agent Programs*, by using which the individual deploying an agent may specify what actions the agent must take, and the rules governing the execution of these actions.

## 6.1 Agent Decision Architecture

The basic decisionmaking structures used by an agent are shown in Figure 6.1. Later, in Chapters 7, 9, and 10, we will expand this significantly to include *beliefs* (about other agents), *uncertainty*, and *security* concerns. These basic decisionmaking structures include the following components:

**Underlying Software Code:** This consists of the basic set of data structures and legacy code on top of which the agent is built. It is accessed through the code-call mechanism formalized in Chapter 4. At any given point $t$ of time, there is a finite set of objects associated with each data type managed by the agent. The set of all such objects, across all the data types managed by the software code, is called the *state of the agent* at time $t$. Clearly, the state of the agent varies with time. Without loss of generality, we will assume that each agent's legacy code includes the "message box" described in Section 4.3.

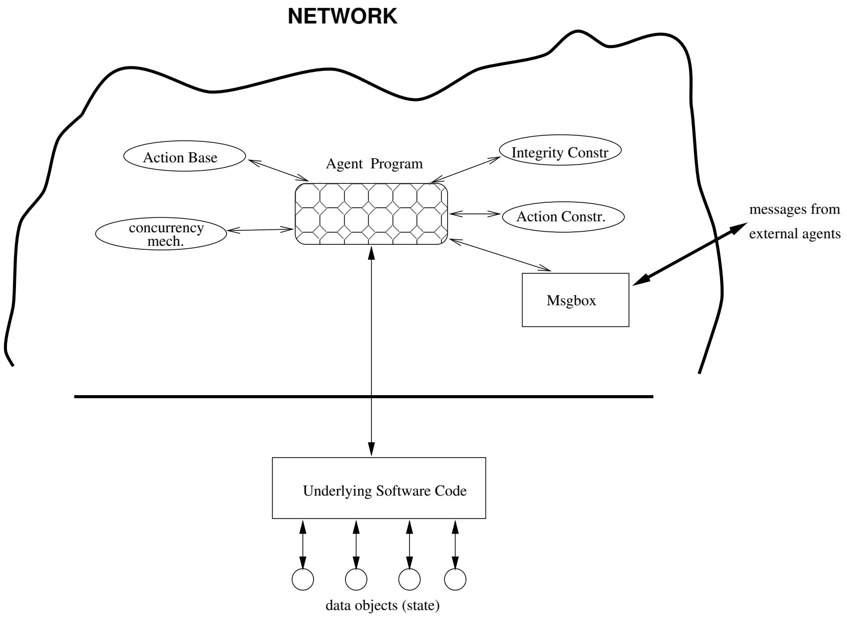



Figure 6.1: Agent Decision Architecture

**Integrity Constraints:** The agent has an associated finite set, $\mathcal{IC}$, These integrity constraints reflect the expectations, on the part of the designer of the agent, that the state of the agent must satisfy.

**Actions:** Each agent has an associated set of *actions*. An action is implemented by a body of code implemented in any suitable imperative (or declarative) programming language. The agent reasons about actions via a set of preconditions and effects defining the conditions an agent state must satisfy for the action to be considered executable, and the new state that results from such an execution. We assume that the preconditions and effects associated with an action correctly specify the behavior of the code implementing the action. The syntax and informal semantics of actions, as well as their sequential and concurrent execution, is described in Section 6.2.

**Action Constraints:** In certain cases, the creator of the agent may wish to prevent the agent from concurrently executing certain actions even though it may be feasible for the agent to take them. Action constraints are user constrained specifications stating the conditions under which actions may not be concurrently executed. Action constraints are described in Section 6.3.

**Agent Programs:** Finally, an agent program is a set of rules, in a language to be defined in Section 6.4, that an agent's creator might use to specify the principles according to which the agent behaves, and the policies governing what actions the agent takes, from among a possible plethora of possible actions. In short, the agent program associated with an agent encodes the "do's and dont's" of the agent.

## 6.2 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 (Nilsson 1980; Genesereth and Nilsson 1987; Russell and Norvig 1995) and logical approaches to action (Baral and Lobo 1996), 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 $\mathcal{S} = (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ is described by the set $O_{\mathcal{S}}$. The data objects in $O_{\mathcal{S}}$ *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 6.2.1 (Action; Action Atom)**
*An* action $\alpha$ *consists of six components:*

**Name:** *A name, usually written* $\alpha(X_1, \ldots, X_n)$*, where the* $X_i$*'s are root variables.*

**Schema:** *A schema, usually written as* $(\tau_1, \ldots, \tau_n)$*, of types. Intuitively, this says that the variable* $X_i$ *must be of type* $\tau_i$*, for all* $1 \leq i \leq n$.

**Action Code:** *This is a body of code that executes the action.*

**Pre:** *A code-call condition* $\chi$*, called the* precondition *of the action, denoted by* $Pre(\alpha)$ *(*$Pre(\alpha)$ *must be* safe modulo the variables $X_1, \ldots, X_n$*);*

**Add:** *a set* $Add(\alpha)$ *of code-call conditions;*

**Del:** *a set* $Del(\alpha)$ *of code-call conditions.*

*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* $\tau_i$*, for all* $i = 1, \ldots, n$.

It is important to note that there is a big distinction between our definition of an action, and the classical definition of an action in AI (Genesereth and Nilsson 1987; Nilsson 1980). Here are the differences.

| Item | Classical AI | Our framework |
|---|---|---|
| Agent State | Set of logical atoms | Arbitrary data structures |
| Precondition | Logical formula | Code call condition |
| Add/delete list | set of ground atoms | Code call condition |
| Action Implementation | Via add list and delete list | Via arbitrary program code |
| Action Reasoning | Via add list and delete list | Via add list and delete list |

A more subtle difference is that in classical AI, states are *physically modified* by union-ing the current state with items in the add list, and then deleting items in the delete list. In contrast, the add-list and the delete-list in our framework plays no role whatsoever in the physical implementation of the action. The action is implemented by its associated action code. The agent uses the preconditions, add list, and the delete list to reason about what is true/false in the new state.

Given any action, we may associate with it an automaton modeling a transition system as follows. The state space of the automaton is the set of all possible (perhaps infinitely many) states of the agent in question. The states in which the precondition of the action are false have no outgoing edges in the automaton. There is an edge (transition) from one state to another if the first state satisfies the precondition of the action, and the second state results from the execution of the action in the first state.

**Note 5** *Throughout this book, we assume that the precondition, add and delete lists associated with an action, correctly describe the behavior of the action code associated with the action.*

Let us now consider some examples of actions and their associated descriptions in specific domains. Obviously, the code implementing these actions is not listed below.

**Example 6.2.1 (CHAIN Revisited)**
*Suppose the* supplier *agent of the CHAIN example has an action called update_stockDB which is used to process orders placed by other agents.*

**Name:** $\mathit{update\_stockDB}(\mathtt{Part\_id}, \mathtt{Amount}, \mathtt{Company})$

**Schema:** (String, Integer, String)

**Pre:** $\mathbf{in}(\mathtt{X}, \mathtt{supplier}{:}\mathit{select}('\mathit{uncommitted}', \mathtt{id}, =, \mathtt{Part\_id}))\ \&\ \mathtt{X.amount} > \mathtt{Amount}.$

**Del:** $\mathbf{in}(\mathtt{X}, \mathtt{supplier}{:}\mathit{select}('\mathit{uncommitted}', \mathtt{id}, =, \mathtt{Part\_id}))\ \&$
$\mathbf{in}(\mathtt{Y}, \mathtt{supplier}{:}\mathit{select}('\mathit{committed}', \mathtt{id}, =, \mathtt{Part\_id}))$

**Add:** $\mathbf{in}(\langle \mathtt{part\_id}, \mathtt{X.amount} - \mathtt{Amount}\rangle, \mathtt{supplier}{:}\mathit{select}('\mathit{uncommitted}', \mathtt{id}, =, \mathtt{Part\_id}))\ \&$
$\mathbf{in}(\langle \mathtt{part\_id}, \mathtt{Y.amount} + \mathtt{Amount}\rangle, \mathtt{supplier}{:}\mathit{select}('\mathit{committed}', \mathtt{id}, =, \mathtt{Part\_id}))$

*This action updates the two ACCESS databases for* uncommitted *and* committed *stock. The* supplier *agent should first make sure that the amount requested is available by consulting the* uncommitted *stock database. Then, the* supplier *agent updates the* uncommitted *stock database to reduce the amount requested and then adds a new entry to the* committed *stock database for the requesting company.*

**Example 6.2.2 (CFIT Revisited)**
*Suppose the* autoPilot *agent in the CFIT example has the following action for computing the current location of the plane:*

***Name:*** $\mathit{compute\_currentLocation}(\mathtt{Report})$

***Schema:*** (SatelliteReport)

***Pre:*** $\mathbf{in}(\mathtt{Report}, \mathtt{msgbox}{:}\mathit{getVar}(\mathtt{Msg.Id}, \texttt{"}\mathit{Report}\texttt{"}))$

***Del:*** $\mathbf{in}(\mathtt{OldLocation}, \mathtt{autoPilot}{:}\mathit{location}()).$

***Add:*** $\mathbf{in}(\mathtt{OldLocation}, \mathtt{autoPilot}{:}\mathit{location}())\&$
$\mathbf{in}(\mathtt{FlightRoute}, \mathtt{autoPilot}{:}\mathit{getFlightRoute}())\&$
$\mathbf{in}(\mathtt{Velocity}, \mathtt{autoPilot}{:}\mathit{velocity}())\&$
$\mathbf{in}(\mathtt{NewLocation}, \mathtt{autoPilot}{:}\mathit{calculateLocation}(\mathtt{OldLocation}, \mathtt{FlightRoute}, \mathtt{Velocity}))$

*This action requires a satellite report which is produced by the* gps *agent by merging the GPS Data. Then, it computes the current location of the plane based on this report as well as the allocated flight route of the plane.*

**Example 6.2.3 (STORE Example Revisited)**
*The* profiling *agent might have the following action:*

**Name:** $\textit{update\_highProfile}(\texttt{Ssn},\texttt{Name},\texttt{Profile})$

**Schema:** *(*String, String, UserProfile*)*

**Pre:** $\textbf{in}(\texttt{spender}(\texttt{high}),\mathtt{profiling}:\textit{classifyUser}(\texttt{Ssn}))$

**Del:** $\textbf{in}(\langle\texttt{Ssn},\texttt{Name},\texttt{OldProfile}\rangle,\mathtt{profiling}:\textit{all}('\textit{highProfile}'))$

**Add:** $\textbf{in}(\langle\texttt{Ssn},\texttt{Name},\texttt{Profile}\rangle,\mathtt{profiling}:\textit{all}('\textit{highProfile}'))$

*This action updates the user profiles of those users who are high spenders. In order to determine the high spenders, it first invokes the classifyUser code call. After obtaining the target list of users, it updates entries of those users in the profile database. The* profiling *agent may also have similar actions for low and medium spenders.*

In our framework, we assume that any explicit state change initiated by an agent is an action. For example, *sending messages* and *reading messages* are actions. 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 6.2.4 (Java Agents)**
*In today's world, the word "agent" is often considered (in certain non-AI communities) to be synonymous with 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* agentID, *may be captured within our framework as follows.*

**Name:** $\textit{exec}(\texttt{Op},\texttt{Host},\texttt{Target},\texttt{ArgumentList})$, *which says* "Execute the operation Op on the list ArgumentList of arguments located at the Target address by moving there from the Host address."

**Pre:** $\textbf{in}(\texttt{Host},\mathtt{java}:\textit{location}(\texttt{Agent-id}))$ &
$\textbf{in}(\texttt{ok},\mathtt{security}:\textit{authorize}(\texttt{Agent-id},\texttt{Op},\texttt{Target},\texttt{ArgumentList}))$.

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

**Add/Del:** *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 6.2.2 (Action Base)**
*An* action base, $\mathcal{AB}$, *is any finite collection of actions.*

The following definition shows what it means to execute an action in a given state.

**Definition 6.2.3 ($(\theta,\gamma)$-Executability)**
*Let $\alpha(\vec{X})$ be an action, and let $\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ 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 $O_{\mathcal{S}}$, if, by definition, there exists a solution $\gamma$ of $Pre(\alpha(\vec{X}))\theta$ w.r.t. $O_{\mathcal{S}}$. In this case, $\alpha(\vec{X})$ is said to be* $(\theta,\gamma)$-executable *in state $O_{\mathcal{S}}$, and $(\alpha(\vec{X}),\theta,\gamma)$ is a feasible* execution triple *for $O_{\mathcal{S}}$. By $\Theta\Gamma(\alpha(\vec{X}), O_{\mathcal{S}})$ we denote the set of all pairs $(\theta,\gamma)$ such that $(\alpha(\vec{X}),\theta,\gamma)$ is a feasible execution triple in state $O_{\mathcal{S}}$.*

Intuitively, in $\alpha(\vec{X})$, the substitution $\theta$ causes all variables in $\vec{X}$ to be grounded. However, it is entirely possible that the precondition of $\alpha$ has occurrences of other free variables not occurring 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 $O_{\mathcal{S}}$. 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 $(\theta,\gamma)$-execution is.

**Definition 6.2.4 (Action Execution)**
*Suppose $(\alpha(\vec{X}),\theta,\gamma)$ is a feasible execution triple in state $O_{\mathcal{S}}$. Then the* result *of executing $\alpha(\vec{X})$ w.r.t. $(\theta,\gamma)$ is given by the state*

$$apply((\alpha(\vec{X}),\theta,\gamma), O_{\mathcal{S}}) = \textbf{ins}(O_{add}, \textbf{del}(O_{del}, O_{\mathcal{S}})),$$

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

We reiterate here the fact that the above definition assumes that the execution of the action code leads to a state which is described precisely by $\text{apply}((\alpha(\vec{X}),\theta,\gamma), O_{\mathcal{S}})$—otherwise, the specification of the action code provided by its associated precondition and add lists are incorrect.

Furthermore, observe that in the above definition, we do not pay attention to integrity constraints. Possible violation of such constraints owing to the execution of an action will be handled later (Section 6.5.1) in the definition of the semantics of agent programs that we are going to develop, and will of course prevent 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 $O_{\mathcal{S}}$, 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 that 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. For example, if only one from many solutions $\gamma$ is selected—in a well-defined way—such that $(\alpha(\vec{X}),\theta,\gamma)$ is feasible, for a grounded action $\alpha(\vec{X})\theta$; we do not discuss this any further here.

### 6.2.1 Concurrent Execution of Actions

Suppose then we wish to simultaneously execute a set of (not necessarily all) feasible execution triples $AS$. There are many ways to define this.

**Definition 6.2.5 (Concurrency Notion)**
*A* notion of concurrency *is a function,* **conc**, *that takes as input, an object state, $O_{\mathcal{S}}$, and a set of execution triples AS, and returns as output, a single new execution triple such that:*

1. *if $AS = \{\alpha\}$ is a singleton action, then $\textbf{conc}(O_{\mathcal{S}}, AS_i) = \alpha$.*

# Chapter 7

# Meta Agent Programs

In this chapter we extend our framework considerably by allowing agents to *reason* about other agents based on the beliefs they hold. We introduce certain *belief data structures* that an agent needs to maintain and introduce *meta agent program* as an extension of agent programs introduced in the previous chapter. We also extend the semantics of *agent programs* to semantics of *meta agent programs*. Finally, we show how meta agent programs can be implemented via agent programs by *encoding beliefs into extended code calls*.

In contrast to the previous chapters we do not consider all three examples CFIT, STORE and CHAIN. The reason is that adding meta-reasoning capabilities to the STORE or CHAIN example would seem rather artificial. Instead, we extend our CFIT example and use this scenario extensively throughout this chapter.

## 7.1 Extending CFIT by Route and Maneuver Planning

We extend our CFIT example by replacing the original plane by a helicopter that operates together with other helicopters in the air and certain tanks at the surface. Thus we consider tasks that are important in route and maneuver planning over free terrain.

A simplified version of such an application that deals with meta-reasoning by agents is shown in Figure 7.1 and is described below. This example, referred to as the CFIT* example (because it extends our earlier CFIT example) will provide a unifying theme throughout this chapter, and will be used to illustrate the various definitions we introduce.

Our application involves tracking enemy vehicles on the battlefield, and attempting to predict what these enemy agents are likely to do in the future, based on metaknowledge that we have about them.

**A set of enemy vehicle agents:** These agents (mostly tanks) move across free terrain, and their movements are determined by a program that the other agents listed below do not have access to (though they may have beliefs about this program). A detailed description is given in Subsection A.4.1.

**A terrain route planning agent,** `terrain`**,** which was already introduced in Chapter 1 (see Table 2.2). Here we extend the `terrain` agent so that it also provides a flight path computation service for helicopters, through which it plans a flight, given an origin, a destination, and a set of constraints specifying the height at which the helicopters wish to fly. The terrain route planning agent is built on top of an existing *US ARMY Route planning software package* de-

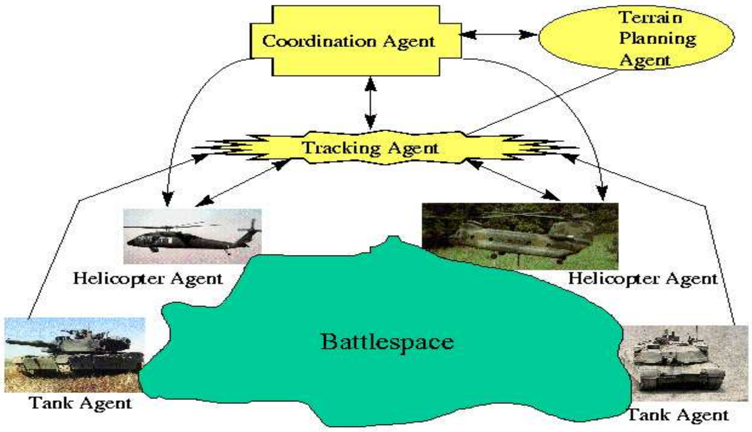



Figure 7.1: Agents in of CFIT* Example

veloped at the *Topographic and Engineering Center* (Benton and Subrahmanian 1994). The code calls and actions associated with `terrain` are described in Subsection A.4.2.

**A tracking agent,** which takes as input, a *DTED* (Digital Terrain Elevation Data) map, an `id` assigned to an enemy agent, and a time point. It produces as output, the location of the enemy agent at the given point in time (if known) as well as its best guess of what kind of enemy the agent is. Section A.4.3 provides full details.

**A coordination agent,** that keeps track of current friendly assets. This agent receives input and ships requests to the other agents with a view to determining exactly what target(s) the enemy columns may be attempting to strike, as well as determining how to nullify the oncoming convoy. The situation is complicated by the fact that the agent may have a hard time determining what the intended attack target is. It may be further complicated by uncertainty about what kind of vehicle the enemy is using—depending upon the type of vehicle used, different routes may be designated as optimal by the terrain route planning agent. Section A.4.4 provides a detailed description.

**A set of helicopter agents,** that may receive instructions from the coordination agent about when and where to attack the enemy vehicles. When such instructions are received, the helicopter agents contact the terrain route planning agent, and request a flight path. Such a flight path uses terrain elevation information (to ensure that the helicopter does not fly into the side of a mountain). We refer the reader to Subsection A.4.5 for a complete description.

The aim of all agents above (except for the enemy agents) is to attack and nullify the enemy attacking force. To do this, the `coordination` agent sends requests for information and analyses to the other friendly agents, as well as instructions to them specifying actions they must take. It is important to note that the `coordination` agent's actions are based on its *beliefs* about what the enemy is likely to do. These beliefs include:

- *Beliefs about the type of enemy vehicle.* Each enemy vehicle has an associated type—for example, one vehicle may be a T-80 tank, the other may be a T-72 tank. However, the coordination agent may not precisely know the type of a given enemy vehicle, because of inaccurate and/or uncertain identification made by the sensing agent. At any point in time, it holds some beliefs about the identity of enemy vehicle.
- *Beliefs about intentions of enemy vehicle.* The coordination agent must try to guess what the enemy's target is. Suppose the tracking agent starts tracking a given enemy agent at time $t_0$, and the current time is $t_{now}$. Then the tracking agent can provide information about the location of this agent at each instant between time $t_0$ and time $t_{now}$. Let $\ell_i$ denote the location of one such enemy agent at time $t_i$, $0 \leq i \leq now$. The coordination agent believes that the enemy agent is trying to target one of its assets $A_1, \dots, A_k$, but does not know which one. It may ask the terrain agent to plan a route from $\ell_0$ to each of the locations of $A_1, \dots, A_k$, and may decide that the intended target is the location whose associated route most closely matches the observed initial route taken by the enemy agent between times $t_0$ and $t_{now}$.
- *Changing beliefs with time.* As the enemy agent continues along its route, the coordination agent may be forced to revise its beliefs, as it becomes apparent that the actual route being taken by the enemy vehicle is inconsistent with the expected route. Furthermore, as time proceeds, sensing data provided by the tracking agent may cause the coordination agent to revise its beliefs about the enemy vehicle type. As the terrain agent plans routes based on the type of enemy vehicle being considered, this may cause changes in the predictions made by the terrain agent.
- *Beliefs about the enemy agent's reasoning.* The coordination agent may also hold some beliefs about the enemy agents' reasoning capabilities (see the *Belief-Semantics Table* in Definition 7.2.4 on page 178). For instance, with a relatively unsophisticated and disorganized enemy whose command and control facilities have been destroyed, it may believe that the enemy does not know what moves friendly forces are making. However, in the case of an enemy with viable/strong operational command and control facilities, it may believe that the enemy does have information on the moves made by friendly forces—in this case, additional actions to mislead the enemy may be required.

A detailed description of all agents and their actions will be given in the Section A.4.

## 7.2 Belief Language and Data Structures

In this section, we introduce the important notion of a *belief atom*. Belief atoms express the beliefs of one agent a about what holds in another agent's, say b's, state. They will be used later in Definition 7.2.7 on page 184 to define the notion of a *meta agent program*, which is central to this chapter. When an agent a reasons about another agent b, it must have some beliefs about b's underlying action base (*what actions can* b *take?*), b's action program (*how will* b *reason?*) etc. These beliefs will be discussed later in more depth.

In this section, we will describe the belief language that is used by *IMPACT* agents. In particular, our definitions proceed as follows:

1. We first describe in Subsection 7.2.1 a hierarchy of belief languages of increasing complexity as we go "up" the hierarchy.

2. We then define in Subsection 7.2.2 an intermediate structure called a *basic belief table*. Intuitively, a basic belief table maintained by agent $a$ contains information about the beliefs $a$ has about the states of other agents, as well as $a$ itself. It also includes $a$'s belief about action status atoms that are adopted by other agents.

3. Each agent also has some beliefs about how other agents reason about beliefs. As the same syntactic language fragment can admit many different semantics, the agent maintains a *Belief Semantics Table*, describing its perceptions of the semantics used by other agents to reason about beliefs (Subsection 7.2.3).

4. We then extend in Subsection 7.2.4 the concept of a basic belief table to a *belief table*. Intuitively, a belief table is obtained by adding an extra column to the basic belief table—the reason for separating these two definitions is that the new column may refer to conditions on the columns of basic belief tables. Intuitively, belief tables contain statements of the form *If condition* $\phi$ *is true, then agent* $a$ *believes* $\psi$ where $\psi$ is a condition about some agent $b$'s state, or about the actions that agent $b$ might take.

It is important to note that assuming additional datatypes as part of our underlying software package has strong implications on the possible code calls as introduced in Definition 4.2.2 on page 66: the more datatypes we have, the more types of code calls can be formulated in our language. We will introduce in Definition 7.3.5 on page 188 a precise notion of the set of *extended code calls*.

### 7.2.1 Belief Language Hierarchy

We are now ready to start defining the beliefs that agent $a$ may hold about the code calls agent $b$ can perform. These code calls determine the code call conditions that may or may not hold in agent $b$'s state. Let us denote this by the belief atom

$$\mathcal{B}_a(b,\chi)$$

which represents one of the beliefs of agent $a$ about what holds in the state of agent $b$. In that case, agent $a$ must have beliefs about agent $b$'s software package $\mathcal{S}^b$: the code call condition $\chi$ has to be contained in $\mathcal{S}^b$. We will collect all the beliefs that an agent $a$ has about another agent $b$ in a set $\Gamma^a(b)$ (see Definition 7.3.4 on page 188).

From now on we will refer to code call conditions satisfying the latter property as *compatible code call conditions* . We will use the same term for action atoms: *compatible action atoms* of agent $a$ with respect to agent $b$, are those action atoms in the action base that $a$ believes agent $b$ holds. We also assume that the structure of such an action contained in $b$'s base (as believed by $a$) is defined in $\Gamma^a(b)$. This means that the *schema*, the *set of preconditions*, the *add-list* and the *delete-list* are uniquely determined.

**Definition 7.2.1 (Belief Atom/Literal, $\mathcal{B}\mathrm{At}_1(a,b)$, $\mathcal{B}\mathrm{Lit}_1(a,\mathbf{A})$)**
*Let $a,b$ be agents in $\mathbf{A}$. Then we define the set $\mathcal{B}\mathrm{At}_1(a,b)$ of $a$-belief atoms about $b$ of level 1 as follows:*

1. *If $\chi$ is a compatible code call condition of $a$ with respect to $b$, then $\mathcal{B}_a(b,\chi)$ is a* belief atom.

2. *For $Op \in \{\mathbf{O},\mathbf{W},\mathbf{P},\mathbf{F},\mathbf{Do}\}$: if $\alpha(\vec{t})$ is a compatible action atom of agent $a$ with respect to $b$, then $\mathcal{B}_a(b, Op\,\alpha(\vec{t}))$ is a* belief atom.

*If* $\mathcal{B}_{\mathtt{a}}(\mathtt{b},\chi)$ *is a belief atom, then* $\mathcal{B}_{\mathtt{a}}(\mathtt{b},\chi)$ *and* $\neg\mathcal{B}_{\mathtt{a}}(\mathtt{b},\chi)$ *are called* belief literals of level 1, *the corresponding set is denoted by* $\mathcal{B}\mathrm{Lit}_1(\mathtt{a},\mathtt{b})$.
*Let*

$$\mathcal{B}\mathrm{At}_1(\mathtt{a},\mathtt{A}) =_{def} \bigcup_{b\in \mathtt{A}} \mathcal{B}\mathrm{At}_1(\mathtt{a},\mathtt{b}) \;\; \text{and} \;\; \mathcal{B}\mathrm{Lit}_1(\mathtt{a},\mathtt{A}) =_{def} \bigcup_{b\in \mathtt{A}} \mathcal{B}\mathrm{Lit}_1(\mathtt{a},\mathtt{b})$$

*be the set of all* $\mathtt{a}$*-belief atoms (resp. belief literals) relative to* $\mathtt{A}$*. This reflects the idea that agent* $\mathtt{a}$ *can have beliefs about many agents in* $\mathtt{A}$*, not just about a single one.*

Here are a couple of belief atoms from our CFIT* example:

**Example 7.2.1 (Belief Atoms In CFIT*)**

- $\mathcal{B}_{\mathtt{heli1}}(\mathtt{tank1},\mathbf{in}(\mathtt{pos1},\mathtt{tank1}:\mathit{getPos}()))$
  *This belief atom says that the agent,* heli1 *believes that agent* tank1*'s current state indicates that* tank1*'s current position is* pos1.
- $\mathcal{B}_{\mathtt{heli1}}(\mathtt{tank1},\mathbf{F}\mathit{attack}(\mathtt{pos1},\mathtt{pos2}))$
  *This belief atom says that the agent,* heli1 *believes that agent* tank1*'s current state indicates that it is forbidden for* tank1 *to attack from* pos1 *to* pos2.
- $\mathcal{B}_{\mathtt{heli3}}(\mathtt{tank1},\mathbf{O}\mathit{drive}(\mathtt{pos1},\mathtt{pos2},35))$
  *This belief atom says that the agent,* heli3 *believes that agent* tank1*'s current state makes it obligatory for* tank1 *to drive from location* pos1 *to* pos2 *at* 35 *miles per hour.*

It is important to note that these are *beliefs* held by agents heli1 and heli3, respectively. Any of them could be an incorrect belief.

Thus far, we have not allowed for nested beliefs. The language $\mathcal{B}\mathrm{Lit}_1(\mathtt{a},\mathtt{A})$ does not allow agent $\mathtt{a}$ to have beliefs of the form "*Agent* b *believes that agent* c*'s state contains code call condition* $\chi$," i.e., agent $\mathtt{a}$ cannot express beliefs it has about the beliefs of another agent.

The next definition introduces nested beliefs and also a general *belief language*. We introduce the following notation: for a given set $X$ of formulae we denote by $\mathbf{Cl}_{\{\&,\neg\}}(X)$ the set of all conjunctions consisting of elements of $X$ or their negations: $x_1 \wedge \neg x_2 \wedge \ldots \wedge x_n$, where $x_i \in X$. We emphasize that this does not correspond to the usual closure of $X$ under & and $\neg$: in particular, it does not allow us to formulate disjunctions, if $X$ consists of atoms.

**Definition 7.2.2 (Nested Beliefs $\mathcal{B}\mathrm{Lit}_i(\mathtt{a},\mathtt{b})$, Belief language $\mathcal{BL}_i^{\mathtt{a}}$)**
*In the following let* $\mathtt{a},\mathtt{b} \in \mathtt{A}$*. We want to define* $\mathcal{BL}_i^{\mathtt{a}}$*, the* belief language of agent $\mathtt{a}$ of level i. *This is done recursively as follows.*

**i ≤ 1:** *In accordance with Definition 7.2.1 on the preceding page (where we already defined* $\mathcal{B}\mathrm{At}_1(\mathtt{a},\mathtt{b})$*) we denote by* $\mathcal{B}\mathrm{At}_0(\mathtt{a},\mathtt{b})$ *as well as by* $\mathcal{B}\mathrm{Lit}_0(\mathtt{a},\mathtt{b})$

$$\{\phi \mid \phi \text{ is a compatible code call condition or action atom}\}$$

*the flat set of code call conditions or action atoms—no belief atoms are allowed. Furthermore, we define*

$$\mathcal{BL}_0(\mathtt{a},\mathtt{b}) =_{def} \mathcal{B}\mathrm{At}_0(\mathtt{a},\mathtt{b})$$
$$\mathcal{BL}_1(\mathtt{a},\mathtt{b}) =_{def} \mathbf{Cl}_{\{\&,\neg\}}(\mathcal{B}\mathrm{At}_1(\mathtt{a},\mathtt{b})),$$

*i.e., the set of formulae* $\mathcal{B}\mathrm{At}_0(\mathtt{a},\mathtt{b})$*, resp. the of all conjunctions of belief literals from* $\mathcal{B}\mathrm{At}_1(\mathtt{a},\mathtt{b})$.

# Chapter 8

# Temporal Agent Programs

In Chapter 6, we have described the important concept of an *agent program* and provided a set of semantics for agent programs based on the concept of a *status set* semantics. Once the designer of an agent has selected the type Sem of semantics he would like his agent to use, the agent continuously executes a cycle as shown in Figure 8 (cf. Algorithm 6.4.1):

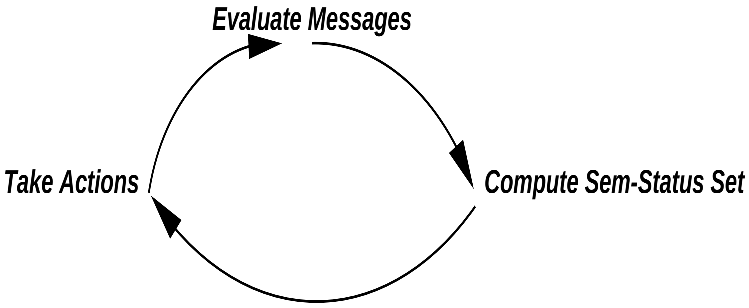



Figure 8.1: Cycle for Temporal Agents

However, the reader will notice that once an agent finishes computing a status set $S$, it *immediately* executes $\mathbf{conc}(DoSet)$ where $DoSet$ is the set of all actions of the form $\{\alpha \mid \mathbf{Do}\,\alpha \in S\}$. This may not always be desirable—for instance, the agent may want to make commitments *now* to perform certain actions in the *future*.

The *syntax* of agent programs described in Chapter 6 suffers from three major shortcomings which we describe below, and these shortcomings also extend to the *semantics* of agent programs.

**Temporal Extents of Actions:** In practice, actions have a *temporal extent*. To see why actions may have temporal extents, consider the CHAIN and CFIT examples. In the CHAIN example, the truck agent may execute the $drive(\mathtt{boston}, \mathtt{chicago})$ action. Clearly, this action has a nontrivial temporal extent during which many other events can occur. Similarly, in the case of the autoPilot agent in the CFIT example, when the action $adjustAltitude(\mathtt{35000})$ requires the plane to adjust its altitude to 35,000 feet, this action takes some time to perform.

**Scheduling Actions:** In addition, the designer of an agent may wish to schedule actions to be executed in the *future*. To see why, consider the STORE example. It may be a legal requirement that every time the credit agent provides a credit report on a customer, that customer must be notified within 10 days. Furthermore, all customers must receive annual reports from the credit agent about his/her credit summary for the year. These annual reports are required to be mailed by February 15 of every year.

**Reasoning about the Past:** Last, but not least, the designer of an agent may wish to take actions (or schedule actions) based on *what has happened in the past*. For example, the credit agent in the STORE example may execute the action *terminateCredit* for a customer who has not responded to three previous actions taken by the credit agent asking him to make an overdue payment.

In order to address these three problems, we will extend the notion of an agent program to that of a *temporal agent program* $\mathcal{TP}$ (tap for short). A tap allows the designer of an agent to specify temporal aspects of actions and states. For simplicity, we assume in this chapter that time points are represented by non-negative integers and the time line extends infinitely far into the future.[1]

The organization of this chapter is as follows. In Section 8.1, we will describe the important concept of a *temporal action*. Then, in Section 8.2, we will define the syntax of taps. In Section 8.3, we will extend the notion of feasible status sets, rational status sets, and reasonable status sets introduced in Chapter 6, to taps. In Section refc8-nego:sec we provide an application of taps to handle strategic negotiations described in the literature. In Section 8.8, we will describe how taps are related to other existing formalisms for temporal agent reasoning.

**Remark 4** *Throughout this chapter, we will assume that the structure of time is modeled by the set of natural numbers* $\mathbb{N}$. *Every agent has an initial "start" time* 0, *which usually denotes the time of deployment of the agent. We assume a distinguished integer valued variable* $\mathtt{X}_{now}$ *which is instantiated to the current time (this can be done by adding the code call* $\mathbf{in}(\mathtt{X}_{now}, \mathtt{agent}{:}\mathit{current\_time}())$*).*

*We also use* $\mathtt{t}_{now}$ *as a metavariable over time (natural numbers). The reader should note that although the concept of a temporal agent program does not depend on a particular time instance, the semantics of such a program, the temporal feasible status set, will be computed later* ***at a fixed instant of time****. We use* $\mathtt{t}_{now}$ *as an index for all concepts that depend on this particular time. This reflects the fact that the state of an agent, O, is considered a snapshot at a particular time instance. The semantics of a program must change over time, because, as time goes by, the agent state changes and therefore rules of the program may apply later that did not before.*

*There is a wide variety of literature, such as (Koubarakis 1994; Ladkin 1986; Ladkin 1987; Leban, McDonald, and Forster 1986; Niezette and Stevenne 1992) showing how time units (e.g., weeks, months, years, decades, etc.) may be represented and manipulated when the natural numbers are assumed to represent time. Hence, in this book, when we refer to a time point such as* Jan. 11, 1998*, we will assume the existence of a mapping from such a representation of time, to the natural numbers, and vice versa.*

## 8.1 Actions with Temporal Duration

Recall, from Definition 6.2.1 of Chapter 6, that an action has five components. These components include the *name* of the action, the *schema* of the action, the *preconditions* required to execute the action, an *add-list* for the action, and *delete-list* for the action.

We would like to provide, in this chapter, an extension of this general definition, to handle the possibility that an action has a *duration* or *temporal extent*. Let us return to our initial example of the autoPilot, drawn from the CFIT example. Suppose the current altitude of the plane is 20,000

[1] The time unit depends on the specific application, e.g., minutes, hours, days, weeks, and we will not discuss it here.

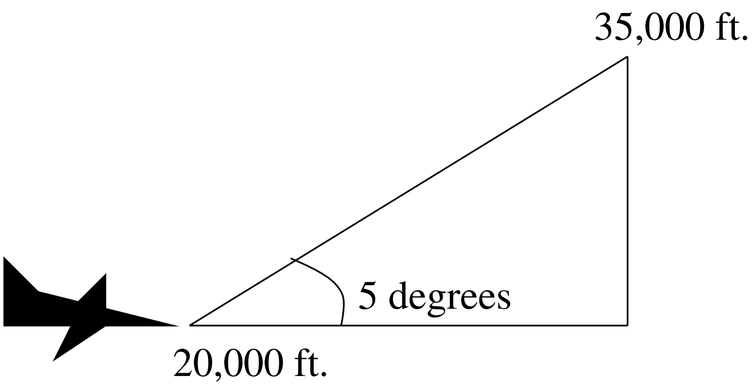



Figure 8.2: Airplane's "Climb" Action

feet, the current velocity of the plane is 1000 feet per minute, and the plane executes the action *adjustAltitude*(`35000`) specifying that it is going to adjust the altitude to 35,000 feet starting now. If the airplane's climb angle is 5 degrees, then it is easy to see (by elementary geometry, see Figure 8.2) that in one minute, the plane will gain $1000\sin(5)$ feet in height. Thus, if our lowest measure of temporal granularity is "minute" then the plane will reach the altitude of 35,000 feet in

$$\lceil\frac{35000-20000}{1000\sin(5)}\rceil$$

minutes. In general, at the end of minute $i$, where

$$0 \le i \le \lceil\frac{35000-20000}{1000\sin(5)}\rceil$$

the altitude of the plane will be given by the formula

$$20000 + (1000\sin(5) \times i).$$

While this reasoning is trivial from a geometric point of view, it does provide some important guidance on what a definition of a timed-action must look like.

1. First, the definition of a timed action must specify the total amount of time it takes for the action to be "completed."

2. Second, the definition of a timed action must specify exactly how the state of the agent changes *while* the action is being executed. Most traditional AI planning frameworks (Nilsson 1980) assume that an action's effects are realized only *after* the entire action is successfully executed.

A further complication arises when we consider the `truck` agent in the CHAIN example, when the `truck` agent is executing the action *drive*(`boston`, `chicago`, `i90`). Unlike the case of the `autoPilot` agent, there may be no easy "formula" that allows us to specify where the truck is at a given instant of time, and furthermore, there *may* be no need to know that the truck has moved one mile further west along Interstate I-90 since the last report. The designer of the `truck` agent may be satisfied with knowing the location of the truck every 30 minutes.

Thus, the notion of a *timed action* should allow the designer of an agent to specify the preconditions of an action, as well as *intermediate effects* that the action has prior to completion. Thus, a

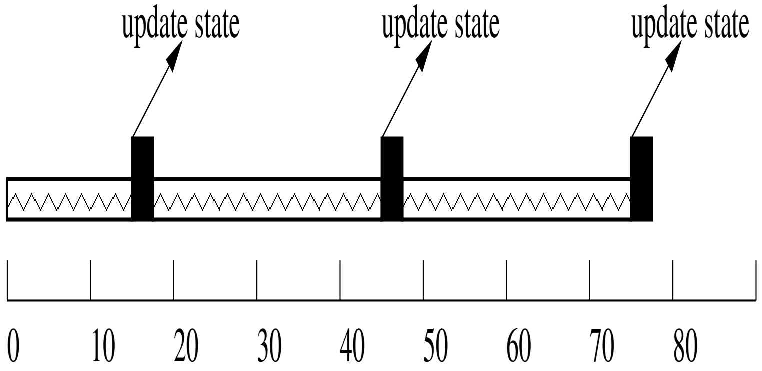



Figure 8.3: Checkpoints of an Action

*timed action* should have some *checkpoints* and intermediate effects could be incorporated within the agent state at these *checkpoints*. For example, Figure 8.3 shows a simple case where an action has a duration of 75 units of time and the action starts being taken at time 0. The action causes the state of the world to continuously change during this time interval. However, the designer of the agent executing this action has specified (using the mechanism described below) that the effects of this action need to be incorporated as updates to the state at times 15, 45, and 75.

### 8.1.1 Checkpoints

It is important to note that it is the *agent designer's responsibility* to specify checkpoints in a manner that satisfies his application's needs. If he needs to incorporate intermediate effects on a millisecond by millisecond basis, his checkpoints should be spaced out at each millisecond (assuming the time unit is not larger than a millisecond). If on the other hand, the designer of the truck agent feels that checkpoints are needed on an hourly basis (assuming the time unit of the time line is not larger than an hour), then he has implicitly decided that incorporating the effects of the *drive* action on an hourly basis is good enough for his application—thus, the decision of what checkpoints to use is entirely made on an application by application basis, and we would like our definition of a timed action to support such checkpoint articulation by an agent designer.

In addition, an agent designer may specify checkpoints by referring to *absolute* time, or he may specify checkpoints by referring to *relative* time. For example, in the STORE example, an absolute checkpoint may say that the action *check_credit* must be executed at 3 am every morning. This is an absolute checkpoint. In contrast, an agent designer associated with the *climb* action of a plane may specify a *relative* checkpoint, requiring that the hight of the plane be updated every 30 seconds after it has started to climb.

**Definition 8.1.1 (Checkpoint Expressions** $\mathsf{rel}:\{\mathtt{X} \mid \chi\}, \mathsf{abs}:\{\mathtt{X} \mid \chi\}$**)**
*A* checkpoint expression *is defined as follows:*

- *If $i \in \mathbb{N}$ is a positive integer, then $\mathsf{rel}:\{i\}$ and $\mathsf{abs}:\{i\}$ are checkpoint expressions.*

- *If $\chi$ is a code call condition involving a non-negative, integer-valued variable* X*, then $\mathsf{rel}:\{\mathtt{X} \mid \chi\}$ and $\mathsf{abs}:\{\mathtt{X} \mid \chi\}$ are* checkpoint expressions.

*We distinguish between checkpoint expressions and actual checkpoints: the latter are time points specified by checkpoint expressions (see below).*

*We will also use $\{chk\}$ as a metavariable for arbitrary checkpoint expressions, both relative and absolute ones.*

A designer of an agent can use absolute time points and relative time points to specify the checkpoints. $\mathsf{abs}:\{i\}$ and $\mathsf{abs}:\{\mathtt{X} \mid \chi\}$ specify *absolute* time points. Intuitively, when we associate the checkpoint expression $\mathsf{abs}:\{\mathtt{X} \mid \chi\}$ with an action $\alpha$, then this says that *every* member of the set

$$\{\mathtt{X}\theta \mid \theta \text{ is a solution of } \chi \text{ w.r.t. the current object state } \mathcal{O}_\mathcal{S}\}$$

is a checkpoint for $\alpha$. When we associate $\mathsf{abs}:\{i\}$ with an action then this says that $i$ itself (viewed as an absolute time point) is a checkpoint.

Alternatively, associating the checkpoint expression $\mathsf{rel}:\{i\}$ with action $\alpha$ says that checkpointing must be done every $i$ units of time *from the start time of* $\alpha$. If $\mathsf{rel}:\{\mathtt{X} \mid \chi\}$ is associated with an action, then this says that for *every* member $d$ of the set

$$\{\mathtt{X}\theta \mid \theta \text{ is a solution of } \chi \text{ w.r.t. the current object state } \mathcal{O}_\mathcal{S}\}$$

checkpointing must be done every $d$ units of time *from the start time of* $\alpha$ *on*. The following are simple checkpoint expressions.

- $\mathsf{rel}:\{100\}$.
  This says that a checkpoint occurs at the time of the start of the action, 100 units later, 200 units later, and so on.
- $\mathsf{abs}:\{\mathtt{T} \mid \mathbf{in}(\mathtt{T}, \mathtt{clock}:\mathit{time}())\,\&\,\mathbf{in}(\mathtt{0}, \mathtt{math}:\mathit{remainder}(\mathtt{T}, \mathtt{100}))\,\&\,\mathtt{T} > 5000\}$.
  This says that a checkpoint occurs at absolute times 5000, 5100, 5200, and so on.

Note that by this definition, checkpoints associated with an action $\alpha$ that are just integers with a prefix "rel" denote *relative* times but not absolute time points. So we have to distinguish between the time point "100" (which can occur in a nontrivial checkpoint expression) and the relative time "100" denoting a sequence of time points of the form

$$t^{\alpha}_{\text{start}},\ t^{\alpha}_{\text{start}} + 100,\ t^{\alpha}_{\text{start}} + 200,\ \ldots$$

where $t^{\alpha}_{\text{start}}$ is the starting time of performing $\alpha$.

**Example 8.1.1**
*The following are some example checkpoint expressions from our STORE, CFIT, and CHAIN examples;*

- *The* `autoPilot` *agent in the CFIT example may use the following checkpoint expression;* $\mathsf{rel}:\{30\}$, *to create checkpoints every 30 seconds.*
- *The* `credit` *agent of the STORE example may use the following checkpoint expression:*

  $$\begin{array}{lllll} \mathsf{abs}:\{ & \mathtt{X}_{now} \mid \mathbf{in}(\mathtt{X}_{now}, \mathtt{clock}:\mathit{time}()) & \& & \mathbf{in}(\mathtt{Overdue}, \mathtt{credit}:\mathit{checkCredit}(\mathtt{Ssn}, \mathtt{Name}))\,\& \\ & > (\mathtt{Overdue}, 0) & \& & \mathbf{in}(\mathtt{0}, \mathtt{math}:\mathit{remainder}(\mathtt{X}_{now}, \mathtt{10})) \\ & \}. & & \end{array}$$

  *This checkpoint expression tells that checkpoints occur at the time* $\mathtt{X}_{now}$ *when there is a customer with an overdue payment credit.*
- *The* `truck` *agent in the CHAIN example may use the checkpoint expression* $\mathsf{rel}:\{60\}$, *to create checkpoints every hour, assuming that the time unit is a minute.*

### 8.1.2 Timed Actions

**Definition 8.1.2 (Timed Effect Triple $\langle\{chk\},Add,Del\rangle$)**
*A* timed effect triple *is a triple of the form $\langle\{chk\},Add,Del\rangle$ where $\{chk\}$ is a checkpoint expression, and Add and Del are add lists and delete lists.*

Intuitively, when we associate a triple of the form $\langle\{chk\},Add,Del\rangle$ with an action $\alpha$, we are effectively saying that the contents of the *Add*- and *Del*- lists are used to update the state of the agent at every time point specified in $\{chk\}$.

A couple of simple timed effect triples are shown below.

- $\langle \mathsf{rel}:\{100\},Add_1,Del_1\rangle$ where $Add_1$ and $Del_1$ are add and delete lists. This timed effect triples says that every 100 units of time, the state should be updated by incorporating the code calls in $Add_1$ and $Del_1$.
- $\langle \mathsf{abs}:\{\mathtt{X}_{\mathsf{now}} \mid \mathbf{in}(\mathtt{X}_{\mathsf{now}},\mathtt{clock}:time())\,\&\,\mathbf{in}(\mathtt{0},\mathtt{math}:rem.(\mathtt{X}_{\mathsf{now}},\mathtt{100}))\,\&\,\mathtt{X}_{\mathsf{now}}>5000\},Add_2,Del_2\rangle$ says that at times 5000, 5100, 5200, and so on, the state should be updated by incorporating the code calls in $Add_2$ and $Del_2$.

**Example 8.1.2 (Timed Effect Triples)**
*The following are some example timed effect triples associated with our STORE, CFIT, and CHAIN examples;*

- *The* autoPilot *agent may employ the following timed effect triple to update the altitude of the plane every 30 seconds;*

$$\begin{array}{lll}
\textit{1st arg}: & & \mathsf{rel}:\{30\} \\
\textit{2nd arg}: & \{ & \mathbf{in}(\mathtt{NewAltitude},\mathtt{plane}:altitude(\mathtt{X}_{now}))\,\} \\
\textit{3rd arg}: & \{ & \mathbf{in}(\mathtt{OldAltitute},\mathtt{plane}:altitude(\mathtt{X}_{now}-\mathtt{30}))\,\}
\end{array}$$

- *The* credit *agent may use the following timed effect triple to notify a customer whose credit has an overdue payment every 10 days;*

$$\begin{array}{llll}
\textit{1st arg}: & \mathsf{rel}: & \{ & \mathtt{X}_{now} \mid \mathbf{in}(\mathtt{X}_{now},\mathtt{clock}:time())\,\& \\
& & & \mathbf{in}(\mathtt{Overdue},\mathtt{credit}:checkCredit(\mathtt{Ssn},\mathtt{Name}))\& \\
& & & >(\mathtt{Overdue},0)\,\& \\
& & & \mathbf{in}(\mathtt{0},\mathtt{math}:remainder(\mathtt{X}_{now},\mathtt{10})) \\
& & \} & \\
\textit{2nd arg}: & & \{ & \mathbf{in}(\langle\mathtt{Name},\mathtt{Ssn},\mathtt{X}_{now}\rangle,\mathtt{credit}:customer_to_be_notified())\& \\
& & & \mathbf{in}(\mathtt{X}_{now},\mathtt{clock}:time())\} \\
& & \} & \\
\textit{3rd arg}: & & \{\} &
\end{array}$$

*The* truck *agent may employ the following timed effect triple to update its current location every hour, assuming that the time unit is a minute;*

$$\begin{array}{lll}
\textit{1st arg}: & & \mathsf{rel}:\{60\} \\
\textit{2nd arg}: & \{ & \mathbf{in}(\mathtt{NewPosition},\mathtt{truck}:location(\mathtt{X}_{now}))\,\} \\
\textit{3rd arg}: & \{ & \mathbf{in}(\mathtt{OldPosition},\mathtt{truck}:location(\mathtt{X}_{now}-\mathtt{60}))\,\}
\end{array}$$

# Chapter 9

# Probabilistic Agent Programs

Thus far, we have assumed that all agents reason with a complete and certain view of the world. However, in most real world applications, agents have only a partial, and often uncertain view of what is true in the world.

For example, consider the CFIT* example described in Chapter 7. In this example, the tracking agent is keeping track of the locations of enemy vehicles over time. Any such endeavor is fraught with uncertainty—the tracking agent may not have the ability to conduct surveillance on a specific enemy vehicle when, for instance, the vehicle's location is occluded from the tracking agent's sensors. This is an example of *positional* uncertainty. Likewise, in the case of the CHAIN example, the plant agent may know that a certain purchase was sent sometime during the first week of June 1998, but is not sure about the exact date on which it was sent—this *temporal uncertainty* may affect the planning performed by this agent.

In general, uncertainty in an agent's reasoning occurs due to the following *basic* phenomena:

- *The agent is uncertain about its state.*
  Throughout this book, we have assumed that agents are certain about their state, i.e. if a code-call of the form $\mathtt{a}:f(\mathtt{d}_1,\ldots,\mathtt{d}_\mathtt{n})$ is executed, a *definite answer* results. If the set $\{o_1,\ldots,o_k\}$ of objects is returned, then each of these objects is definitely in the result. However, consider our tracking agent in the CFIT* example—when identifying an object through visual imagery, it may return the fact that object $o$ is a T-72 tank with 70–80% probability and a T-80 tank with 20–30% probability. Thus, if we were to execute a code-call of the form $\mathtt{tracking}:\mathit{findobjects}(\mathtt{image1})$, the answer described above is difficult to express in our current framework—returning a set containing triples of the form $\{(t72,0.7,0.8),(t80,0.2,0.3)\}$ would be incorrect because this does not capture the intuition that the object is either a T-72 or a T-80, but not both.

- *The agent is uncertain about when some of its actions will have effects.*
  In Chapter 8, we have provided a detailed definition of how actions can have effects over time, and how such delayed effects can be modeled through the mechanism of checkpoints. However, there are a wide range of applications in which we cannot be sure of when an action's effects will be realized. For instance, consider the case where an action of the form $\mathit{Fly}(\mathtt{boston},\mathtt{chicago},\mathtt{flightnum})$ is executed at time $t$. Even if we know the arrival and departure times of the flight in question, there is some uncertainty about exactly when this action will be completed. The airline in question may have statistical data showing a probability distribution over a possible space of completion times.

- *The agent is uncertain about its beliefs about another agent's state.*

This kind of uncertainty arises, for instance, in the CFIT* example where, as seen in Example 7.2.2 on page 177, we may have a situation where heli1 is not sure where agent tank1 is currently located—here, heli1 may believe that agent tank1 is located at some point along a stretch of highway with a certain probability distribution. In this case, heli1 needs to take this uncertainty into account when determining whether to fire at the enemy tank.

- *The agent is uncertain about its beliefs about another agent's actions.*
  This kind of uncertainty arises when one agent is unsure about what another agent will do—what are the other agent's obligations, permissions, etc. For example, the heli1 agent may not be certain about the speed at which a given enemy vehicle can move over a certain kind of terrain. Thus, it may hypothesize that the enemy tank1 agent will execute one of the actions **Do***Drive*(pos1,pos2,35), ..., **Do***Drive*(pos1,pos2,50), with an associated probability distribution over these potential actions.

This is not intended to be a complete and comprehensive list of why agents may need to deal with uncertainty—rather, it represents a small set of core reasons for needing to deal with uncertainty.

In this chapter, we will comprehensively address the first kind of uncertainty described above, and we will briefly indicate how we can deal with the other types of uncertainty.

Before going into our technical development, a further note is in order. Uncertainty has been modeled in many different ways. Fuzzy sets (Zadeh 1965; Baldwin 1987; Dubois and Prade 1988; Dubois and Prade 1989), Bayesian networks (Pearl 1988), possibilistic logic (Dubois and Prade 1991; Dubois, Land, and Prade 1991; Dubois, Lang, and Prade 1994; Dubois and Prade 1995) and probabilities (Nilsson 1986; Emden 1986; Fagin and Halpern 1989; Fagin, Halpern, and Megiddo 1990; Guntzer, Kiessling, and Thone 1991; Kiessling, Thone, and Guntzer 1992; Ng and Subrahmanian 1993b; Ng and Subrahmanian 1993a; Lakshmanan and Sadri 1994a; Lakshmanan and Sadri 1994b; Ng and Subrahmanian 1995; Zaniolo, Ceri, Faloutsos, Snodgrass, Subrahmanian, and Zicari 1997; Lakshmanan and Shiri 1999) are four leading candidates for reasoning about uncertain domains. Of all these, there is little doubt that probability theory remains the most widely studied. As a consequence, we have chosen to develop a *probabilistic* theory of agent reasoning in uncertain domains. The others represent rich alternative avenues for exploration in the future.

## 9.1 Probabilistic Code Calls

Consider a code call of the form $\mathtt{a}:_{\mathbf{RV}} f(\mathtt{d}_1,\ldots,\mathtt{d}_n)$. This code call returns as output, some set of objects $o_1,\ldots,o_k$ each of type $\tau$ where $\tau$ is the output type of $f$. By returning $o_i$ as an output object, we are declaring that in an agent state $O$, $o_i \in \mathtt{a}:_{\mathbf{RV}} f(\mathtt{d}_1,\ldots,\mathtt{d}_n)$. Uncertainty arises when we do not know what objects are in the set $\mathtt{a}:_{\mathbf{RV}} f(\mathtt{d}_1,\ldots,\mathtt{d}_n)$.

For instance, in the CFIT* example, the tracking agent, when invoked with the code call tracking:*findobjects*(image1), may wish to report that a T-72 tank is definitely in the image, and another tank, either a T-72 (70–80% probability) or a T-80 (20–30% probability), is in the image. The current output type of the code call tracking:*findobjects*(image1) does not allow this to be returned. The problem is that instead of returning a set of objects, in this case, we need to return a set of *random variables* (see (Ross 1997)) in the strict sense of probability theory. Furthermore, these random variables need to have the same type as the code call's output type.

**Definition 9.1.1 (Random Variable of Type $\tau$)**
*A* random variable *of type $\tau$ is a finite set* **RV** *of objects of type $\tau$, together with a probability distribution $\wp$ that assigns real numbers in the unit interval $[0,1]$ to members of* **RV** *such that* $\Sigma_{o\in\mathbf{RV}}\wp(o) \leq 1$.


Figure 9.1: Example of Random Variable in CFIT* Example

It is important to note that in classical probability theory (Ross 1997), random variables satisfy a stronger requirement that $\Sigma_{o \in \mathbf{RV}} \wp(o) = 1$. However, in many real life situations, a probability distribution may have missing pieces, which explains why we have chosen a weaker definition.

Let us see how this notion of a random variable is pertinent to our CFIT* example.

**Example 9.1.1 (CFIT* Example Revisited)**
*For example, consider the CFIT* example, and suppose we have a tank as shown in Figure 9.1. In this case, the* `tracking` *agent may not know the precise location of the tank because some time may have elapsed since the last surveillance report at which this tank was observed. Based on its projections, it may know that the tank is somewhere between markers 11 and 15 on a particular road. It may assume that the probability of exactly which point it is at is uniformly distributed over these points. Hence, given any of these five points, there is a 20% probability that the tank is at that point.*

**Definition 9.1.2 (Probabilistic Code Call $\mathtt{a}:_{\mathbf{RV}} f(\mathtt{d_1},\ldots,\mathtt{d_n})$)**
*Suppose* $\mathtt{a}:f(\mathtt{d_1},\ldots,\mathtt{d_n})$ *is a code call where* $f$*'s output type is* $\tau$. *A* probabilistic code call *associated with* $\mathtt{a}:f(\mathtt{d_1},\ldots,\mathtt{d_n})$ *when executed on state* $O$ *returns as output, a set of random variables of type* $\tau$. *To distinguish this code call from the original code call, we denote it by* $\mathtt{a}:_{\mathbf{RV}} f(\mathtt{d_1},\ldots,\mathtt{d_n})$.

The following example illustrates the use of probabilistic code calls.

**Example 9.1.2 (CFIT* Example Revisited)**
*Let us extend Example 9.1.1 to the situation shown in Figure 9.2 on the following page. Here, two vehicles are moving on two intersecting roads. The traffic circle is at location 12 on both roads. Suppose we know that the tank on road 1 (*`tank1`*) is somewhere between locations 1 and 10 (with a uniform distribution) and* `tank2` *on road 2 is somewhere between locations 1 and 8 (with a uniform distribution as well).*

*Suppose we can execute a code call that answers the query "Find all vehicles within 6 units of the traffic circle." Clearly both* `tank1` *and* `tank2` may *be within 6 units of the circle. The probability that* `tank1` *is within 6 units of the circle is the probability that* `tank1` *is at one of locations 6,7,8,9,10, which equals* 0.5. *Similarly, the probability that* `tank2` *is within 6 units of the circle is the probability that* `tank2` *is at one of locations 6,7,8 which is* 0.375.

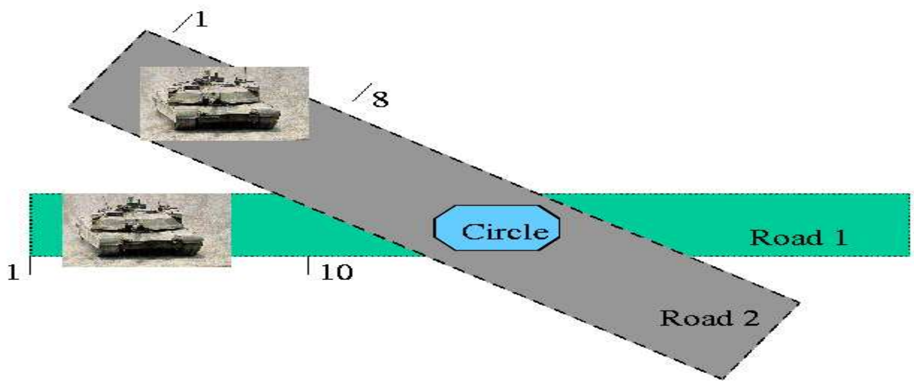



Figure 9.2: Example of Probabilistic code calls in CFIT* Example

*Therefore, the answer to this probabilistic code call should contain two random variables*[1]

$$\langle\{\mathtt{tank1}\},0.5\rangle,\langle\{\mathtt{tank2}\},0.375\rangle.$$

It is important to note that probabilistic code calls and ordinary code calls have the same syntax—however, the results they return may be different. The former returns a set of random variables of type $\tau$, while the latter returns a set of objects of type $\tau$.

Let us see how the above definition of a probabilistic code call may be extended to probabilistic code call atoms. Syntactically, a probabilistic code call atom is exactly like a code call atom—however, as a probabilistic code call returns a set of *random variables*, probabilistic code call atoms are true or false with some probability. Let us consider some simple examples before providing formal definitions.

**Example 9.1.3 (CFIT* Example Revisited)**
*Let us return to the case of Example 9.1.2 on the page before. Consider the code-call atom* $\mathbf{in}(\mathtt{X},\chi)$ *where* $\chi$ *is the code call "Find all vehicles within 6 units of the traffic circle" described in Example 9.1.2 on the preceding page. Clearly,* $\mathbf{in}(\mathtt{tank1},\chi)$ *should be true with 50% probability and* $\mathbf{in}(\mathtt{tank2},\chi)$ *should be true with 37.5% probability because of the reasoning in Example 9.1.2.*

However, as the following example shows, this kind of reasoning may very quickly lead to problems.

**Example 9.1.4 (Combining Probabilities I)**
*Suppose we consider a code call* $\chi$ *containing the following two random variables.*

$$\begin{aligned}\mathbf{RV}_1 &= \langle\{a,b\},\wp_1\rangle\\ \mathbf{RV}_2 &= \langle\{b,c\},\wp_2\rangle\end{aligned}$$

*Suppose* $\wp_1(a)=0.9,\wp_1(b)=0.1,\wp_2(b)=0.8,\wp_2(c)=0.1$. *What is the probability that* $b$ *is in the result of the code call* $\chi$*?*

*Answering this question is problematic. The reason is that we are told that there are at most two objects returned by* $\chi$. *One of these objects is either* $a$ *or* $b$, *and the other is either* $b$ *or* $c$. *This leads to four possibilities, depending on which of these is true. The situation is further complicated because in some cases, knowing that the first object is* $b$ *may preclude the second object from being* $b$*—this would occur, for instance, if* $\chi$ *examines photographs each containing two different people*

[1]Distinguish this from the following *one* random variable: $\langle\{\mathtt{tank1},\mathtt{tank2}\},\wp\rangle$.

*and provides identifications for each. $a$, $b$ and $c$ may be potential id's of such people returned by the image processing program. In such cases, the same person can never be pictured with himself or herself.*

*Of course, in other cases, there may be no reason to believe that knowing the value of one of two objects tells us anything about the value of the second object. For example if we replace people with colored cubes (with $a$ denoting amber cubes, $b$ black, and $c$ cyan), there is no reason to believe that two identical black cubes cannot be pictured next to each other.*

The source of the problem above is that of *disjunctive information*. The object $b$ could be in the result of executing the code call $\chi$in one of two ways—because of the first random variable, or because of the second.

There are two ways around this problem. Thus far in this book, we have assumed that all code calls return *sets* of objects, not multisets. Under this interpretation, the scenario in Example 9.1.4 on the preceding page has another *hidden* constraint which says that if the first random variable is known to have a value, then the other random variable cannot have the same value.

The other scenario would be to argue that the reasoning in Example 9.1.4 is incorrect—that if two objects are completely identical, then they must be the same. This means that if we have two distinct black cubes, then these two black cubes must be *distinguishable* from one another via some property such as their location in the photo, or *Id*s assigned to them must be distinct. This is in fact quite reasonable: it is the *extensionality principle* which dates back to Leibniz.

In either of these two cases, it is reasonable to assume that every code call returns a set of random variables that have no overlap. This is formalized in the next definition.

**Definition 9.1.3 (Coherent Probabilistic Code Call)**
*Consider a probabilistic code call that returns a set of random variables of type $\tau$. This probabilistic code call is said to be* coherent *if, by definition, whenever $\langle X_1, \wp_1\rangle, \langle X_2, \wp_2\rangle$ are distinct random variables in the set of output random variables, then $X_1 \cap X_2 = \emptyset$.*

Throughout the rest of this chapter, we will assume that only coherent probabilistic code calls are considered. Thus, the expression "probabilistic code call" will in fact denote "coherent probabilistic code call."

**Definition 9.1.4 (Probabilistic Code Call Atom)**
*Suppose $\mathtt{a}:_{\mathbf{RV}} f(\mathtt{d}_1,\dots,\mathtt{d}_n)$ is a ground probabilistic code call and suppose $o$ is an object of the output type of this code call w.r.t. agent state $O$. Suppose $[\ell,u]$ is a closed subinterval of the unit interval $[0,1]$. We define below, what it means for $o$ to* probabilistically satisfy *a code call atom.*

- $o \models_O^{[\ell,u]} \mathbf{in}(\mathtt{X}, \mathtt{a}:_{\mathbf{RV}} f(\mathtt{d}_1,\dots,\mathtt{d}_n))$
*if, by definition, $(Y,\wp)$ is a random variable contained in $\mathtt{a}:_{\mathbf{RV}} f(\mathtt{d}_1,\dots,\mathtt{d}_n)$ when evaluated w.r.t. $O$ and $o \in Y$ and $\ell \le \wp(o) \le u$.*

- $o \models_O^{[\ell,u]} \mathbf{not\_in}(\mathtt{X}, \mathtt{a}:_{\mathbf{RV}} f(\mathtt{d}_1,\dots,\mathtt{d}_n))$
*if, by definition, for all random variables $(Y,\wp)$ contained in $\mathtt{a}:_{\mathbf{RV}} f(\mathtt{d}_1,\dots,\mathtt{d}_n)$ when evaluated w.r.t. $O$, either $o \notin Y$ or $\wp(o) \notin [\ell,u]$.*

As in Definition 4.2.4 on page 68, we define the important notion of probabilistic code call conditions.

**Definition 9.1.5 (Probabilistic Code Call Condition)**
*A* probabilistic code call condition *is defined as follows:*

# Chapter 10

# Secure Agent Programs

As more and more agent applications are built and deployed, and as access to data and services is increasingly provided by such agents, the need to develop techniques to enforce security become greater and greater. For example, in the STORE example, the `credit` agent provides access to sensitive credit data and credit rating services which should only be accessible to users or agents authorized to make such accesses. Likewise, in the CFIT example, the current location of a Stealth `autoPilot` agent during a mission is a piece of classified information that should not be disclosed to arbitrary agents. In addition, as agents operate in an environment involving a variety of host systems and other agents, tools should be available that allow the agent developer to configure his agent in a way that ensures that it will not crash host computers and/or maliciously attack other agents. In general, there is a wide range of security problems that arise in an agent environment such as (but not restricted to) the following:

- Agents often communicate through messages; it may be necessary to *encrypt* such messages to prevent unauthorized agents from reading them.

- Some agents may be willing to provide certain services only to specifically authorized agents; this implies that reliable *authentication mechanisms* are needed (to check that the "client" agent is not pretending to be somebody else).

- Mobile agent *hosts* should be protected from being misused—or even crashed—by malicious and/or misfunctioning agents.

- Symmetrically, mobile agents' integrity should be protected from malicious and/or misfunctioning hosts, which—for example—might attempt to read agents' private data, and modify their code.

As research into authentication mechanisms and prevention of network "sniffers" is extensive and can be directly incorporated within agents, in this chapter we focus only on how agents can support the following two major principles of security:

**Data security principle** For each data-object in an agent's state, there may be restrictions on which other agents may read, write, or otherwise manipulate that data.

**Action security principle** For each action in an agent's repertoire, there may be restrictions on which other agents may utilize to those actions.

**Example 10.0.1 (CFIT)**
*The current location of a Stealth plane agent during a mission is a piece of classified information that should not be disclosed to arbitrary agents. According to the data security principle, the plane's* `autoPilot` *agent should answer a current location request (thereby disclosing part of its data structures) only if the request comes from an authorized military agent.*

*To see an application of the action security principle, suppose that an* `autoPilot` *agent is asked to change the plane's altitude. Such requests should be obeyed only when they come from certified traffic control agents. Interference from other agents would turn air traffic into chaos.*

The above example shows that through service requests, agents may obtain (part of) other agents' data; similarly, through service requests agents can make other agents execute actions. Therefore, in order to enforce the basic security principles described above, there must be a mechanism to ensure that while servicing incoming requests, agents will never improperly disclose any piece of sensitive or secret information, nor will they execute any undesirable actions.

To some extent, data security is analogous to database security. The oldest approaches to database security restrict query processing in such a way that no secret tuples are returned with the answer. This very simple form of filtering— called *surface security*— does not take into account the ability to infer new information from the information disclosed by the database. However, database query answers can be enriched by users through background knowledge, other data sources (e.g., databases or humans), and so on. Thus, users may be able to infer a secret indirectly from query results, even if such results contain no explicit secret. In this respect, software agents are not different: they are general computer programs, with enormous computational potentials, which may combine data obtained from different agents and derive secret information, not explicitly provided by any individual data source.

**Example 10.0.2 (CFIT)**
*Agents need not be extremely intelligent to infer secrets. Suppose the current position $\vec{p}_{now}$ of a military plane is a secret. An air traffic control agent* `ground_control` *may compute $\vec{p}_{now}$ from the current velocity $\vec{v}$ of the plane and its position $\vec{p}_t$ at the last velocity change, using the formula: $\vec{p}_{now} = \vec{p}_t + \vec{v} \cdot (now - t)$ , that involves only straightforward numeric calculations.*

In light of the above discussion, a stronger notion of security is needed—when an agent is responding to a request from a client agent, it must ensure that the client agent does not *derive* secrets that it wants to keep hidden from the client agent.

As agents do not know much about the inferential abilities and knowledge sources of other agents, how can they determine what information can be safely disclosed? For example, modeling human agents is a hopeless task. Humans are frequently unable to explain even their own inferences. Software agents are somewhat easier to model. For instance, if agents `a` and `b` are produced by the same company, then the developers of `a` may be able to encode a precise model of `b` inside agent `a` because they have access to `b`'s code, which determines both the possible knowledge sources and the possible inferences of `b`. Nonetheless, even in this fortunate case, `a` might not know what knowledge has been gathered by `b` at arbitrary points in time, because this depends on which agents have been contacted by `b`, which in turn depends on the state of the network, and other factors that are difficult to model precisely. Thus, agents have to preserve data security using incomplete and imprecise knowledge about other agents.

In this chapter, we make the following contributions:

- First, in Section 10.1, we introduce a *completely logical agent model* that enables us to discuss agent security mechanisms.

- Second, in Section 10.2, we propose an abstract definition of what it means for an abstract agent to preserve data and action security. This apparently straightforward task turns out to be extremely complex, and involves several subtleties. It turns out that preserving the *exact* abstract notion of security described here is basically impossible, because it requires the agent in question to have a vast body of knowledge that it usually will not have.

- We attack this problem head on in Section 10.3 where we introduce a methodology for designing safe data security checks using incomplete and imprecise knowledge about other agents—these checks will be called *approximate security checks*. The methodology is developed using the abstract agent model. This has the advantage that attention is focused on the logical aspects of maintaining security in the absence of implementation choices made in *IMPACT*. We introduce two types of security checks—*static* security checks which, if checked at compile-time, guarantee that the agent will always be secure, and dynamic security checks that allow the agent to dynamically adapt to preserve security. Approximate security checks are compatible both with static security verification and with dynamic (run-time) security verification.

- In Section 10.4, we show that the problem of *exact* static security verification as well as various other related problems are undecidable. We study the different sources of complexity, and provide the good news that if we are willing to live with some constraints, then security can be guaranteed.

- *IMPACT*'s architecture for implementing of secure services and security related tools is illustrated in Section 10.5. The underlying model is based on the notion of action security introduced in Section 10.2 and on the methodology for approximate data security checks of Section 10.3.

- Related work is discussed in Section 10.6.

## 10.1 An Abstract Logical Agent Model

In this section, we will impose a logical model of agents on top of the framework described thus far in this book. Every agent has an associated *logical* state generalizing that of Chapter 6. In addition, at any given point in time, each agent has an associated *history* of interactions with other agents which play a role in shaping agent $\mathtt{a}$'s beliefs about agent $\mathtt{b}$'s beliefs. Each agent has an associated *inference mechanism* or *logical consequence notion* that it uses to infer data from a given body of data. Of course, it is possible that some agents use a degenerate form of inference (e.g., membership in a set of facts), while others may use first order logical reasoning, or yet other logical systems. Finally, in response to a request, each agent evaluates that request via an abstract *service evaluation function* which specifies which other agents will be contacted, and which queries/operations will be executed by the agent. In this section, we study each of these four parameters *without* concerning ourselves about security. Section 10.2 will then explain how this abstract agent model may be modified to accommodate security needs.

### 10.1.1 Logical Agent States

The state of an agent may be represented as a set of ground logical facts. In other words, the state of an agent may be *represented* as the set of all ground code call atoms $\mathbf{in}(\mathtt{o}, \mathcal{S}:f(\mathtt{a_1}, \ldots, \mathtt{a_n}))$ which are true in the state, where $\mathcal{S}$ is the name of a data structure manipulated by the agent, and $f$ is one of the functions defined on this data structure. The following examples show how this may be accomplished.

**Example 10.1.1 (CFIT)**
*Consider a* `ground_control` *agent written in C. This kind of agent is likely to store the current position of planes in data structures of the following type:*

```
struct 3DPoint {
    float latitude;
    float longitude;
    float altitude; }
```

*The current value of the the above three fields can be represented by the atomic code call condition:*

$$\mathbf{in}(\mathtt{X}, \mathtt{plane}{:}\mathit{current\_loc}())$$

*where* `X` *is an object of type* `3DPoint` *with three fields:* `X.latitude`, `X.longitude` *and* `X.altitude`*. The same fact-based representation can be used for the instances of a Java class such as*

```
class 3DPoint {
    float latitude;
    float longitude;
    float altitude;
    ...  }
```

*Example 10.1.5 on page 278 will show that facts are also suitable for representing class methods.*

Formally, we associate with each agent $a$, a language that determines the syntactic structure of facts.

**Definition 10.1.1 (Fact Language $\mathcal{L}_a$)**
*Each agent $a$ has an associated language $\mathcal{L}_a$ (a denumerable set), such that for all states $O$ of $a$, $O \subseteq \mathcal{L}_a$.*

To tie this definition to *IMPACT*, we note that an *IMPACT* agent $a$ may have as its associated fact language, the set of all ground code call atoms expressible by it.

**Remark 5** *Two states that satisfy the same code call conditions are identical in the abstract framework. This is a reasonable assumption, as the behavior of IMPACT agents depends only on the value of code call conditions ("internal" differences which do not affect the value of code call conditions may be ignored).*

### 10.1.2 Abstract Behavior: Histories

As mentioned earlier, there are two types of *events* that may affect agent states.

**action events,** denoted by the corresponding action names, (we shall use for this purpose the meta-variable $\alpha$, possibly with subscripts or superscripts) represent the actions that an agent has taken, either autonomously or in response to a request made by another agent;

**message events** represented by triples $\langle \mathit{sender}, \mathit{receiver}, \mathit{body} \rangle$, where *sender* and *receiver* are agents, $\mathit{sender} \neq \mathit{receiver}$, and *body* is either a service request $\rho$ or an *answer*, that is, a set of ground facts $\mathit{Ans} = \{f_1, f_2, \ldots\}$.

Formally, we need no assumptions on the syntactic structure of service requests—our results do not depend on it. However, for the purpose of writing some examples, we shall adopt service requests of the form $sn(i_1,\dots,i_k,mi_1,\dots,mi_m)$, where $sn$ is a service name, $i_1,\dots,i_k$ are its inputs, and $mi_1,\dots,mi_m$ are its mandatory inputs, while answers will be sets of facts of the form $sn(i_1,\dots,i_k,mi_1,\dots,mi_m,o_1,\dots,o_n)$, where $o_1,\dots,o_n$ are the service outputs (see Definition 4.6.1 on page 79).

We are now ready to define the basic notion of a history, as a sequence of events.

**Definition 10.1.2 (Histories)**
*A* history *is a possibly infinite sequence of events, such as $\langle e_1, e_2, \dots\rangle$. We say that a history $h$ is a* history for $\mathtt{a}$ *if each action in $h$ can be executed by $\mathtt{a}$, and for all messages $\langle s,r,m\rangle$ in $h$, either $s = \mathtt{a}$ or $r = \mathtt{a}$.*

*The* concatenation *of two histories $h_1$ and $h_2$ will be denoted by $h_1 \cdot h_2$. With a slight abuse of notation, we shall sometimes write $h \cdot e$, where $e$ is an event, as an abbreviation for the concatenation $h \cdot \langle e\rangle$.*

The notion of a history for $\mathtt{a}$ keeps track of a set of messages that $\mathtt{a}$ has interchanged with other agents, and a set of actions that $\mathtt{a}$ has performed. It is important to note that a history need not be complete—an agent may or may not choose to explicitly keep all information about events in its history.

**Example 10.1.2 (CFIT)**
*A history for an $\mathtt{autoPilot}$ agent may have the form $\langle \dots e_1, e_2, e_3, e_4 \dots\rangle$, where:*

$$\begin{aligned}
e_1 &= \langle \mathtt{ground_control}, \mathtt{autoPilot}, \mathtt{set{:}altitude}(\mathit{new_alt})\rangle,\\
e_2 &= \mathit{climb}(\mathit{15sec}),\\
e_3 &= \langle \mathtt{ground_control}, \mathtt{autoPilot}, \mathtt{location}()\rangle,\\
e_4 &= \langle \mathtt{autoPilot}, \mathtt{ground_control}, \{\mathtt{location}((50,20,40))\}\rangle.
\end{aligned}$$

*Here $e_1$, $e_3$ are request messages, $e_2$ is an action event, and $e_4$ is an answer message. Intuitively, the ground control asks the $\mathtt{autoPilot}$ to change the plane's altitude, then asks for the new position. Events $e_2$ and $e_4$ model the $\mathtt{autoPilot}$'s reactions to those requests.*

As mentioned at the beginning of this section, the events in an agent's history determine the agent's current state. Accordingly, we adopt the following notation.

**Definition 10.1.3 (Agent State at $h$: $O_{\mathtt{a}}(h)$)**
*For all agents $\mathtt{a}$ and all histories $h$ for $\mathtt{a}$, we denote by $O_{\mathtt{a}}(h)$, the state of $\mathtt{a}$ immediately after the sequence of events $h$. The initial state of $\mathtt{a}$ (i.e. the state of $\mathtt{a}$ when it was initially deployed) is denoted by $O_{\mathtt{a}}(\langle\rangle)$.*

The following example illustrates the above definition.

**Example 10.1.3**
*If $h$ has the form $\langle \dots e_4\rangle$, where $e_4$ is the event described in Example 10.1.2, and $F$ is the fact $\mathbf{in}((50,20,40), \mathtt{autoPilot}{:}\mathit{location}())$, then $O_{\mathtt{autoPilot}}(h)$ may contain the facts $F$ and*

$$\mathbf{in}(F, \mathbf{BT}^{\mathbf{a}} : \text{proj-select}(\text{agent}, =, \mathtt{ground\_control})), \tag{10.1}$$

*(recall that (10.1) means that $\mathtt{autoPilot}$ believes that $\mathtt{ground\_control}$ believes that $\mathtt{autoPilot}$'s current position is $(50,20,40)$).*

The notion of history for $\mathfrak{a}$ captures histories that are *syntactically* correct. However, not every history for $\mathfrak{a}$ describes a possible behavior of $\mathfrak{a}$. For instance, some histories are impossible because $\mathfrak{a}$'s code will never lead to that sequence of events. Some others are impossible because they contain messages coming from agents that will never want to talk to $\mathfrak{a}$. The definition below models the set of histories that might actually happen.

**Definition 10.1.4 (Possible Histories)**
*The set of* possible histories *for $\mathfrak{a}$ is denoted by $pos\mathcal{H}_{\mathfrak{a}}$. It is a subset of the set of all histories for $\mathfrak{a}$.*

### 10.1.3 Agent Consequence Relation

In principle, "intelligent" agents can derive new facts from the information explicitly stored in their state. Different agents have different reasoning capabilities. Some agents may perform no reasoning on the data they store, some may derive new information using numeric calculations (as in Example 10.0.2 on page 274); others may have sophisticated inference procedures.

**Example 10.1.4**
*Given any agent $\mathfrak{a}$, we may regard the* $\mathbf{in}(.,.)$ *predicate and $=, >, <, \geq, \leq$ as standard predicates of first order logic. Then $\mathfrak{a}$'s state can be viewed as a set of first-order formulas (the code call conditions which are true in the state), from which $\mathfrak{a}$ may be able to infer (some) logical consequences, using the standard inferences of first-order logic.*

The notion of agent consequence introduced below will model the process by which agents draw inferences based upon their state. The result of those inferences will be modeled by sets of facts.

**Example 10.1.5 (CFIT)**
*Let us reconsider Example 10.1.1 on page 276. If the* `ground_control` *agent were written in Java, then it would probably represent planes with a Java class* `Plane`*, one of whose methods would be* `current_loc()`*. In this case, the private variables of the class* `Plane` *need not explicitly represent the plane's current location. For instance, by analogy with Example 10.0.2 on page 274, such variables may encode the plane's position and velocity at the last time $t$ when its velocity changed, and the method* `current_loc` *may compute the current position with the formula $\vec{p}_{now} = \vec{p}_t + \vec{v} \cdot (now - t)$. The method* `location` *can be represented by the code call*

$$\mathbf{in}(\mathtt{X}, \mathtt{plane}\!:\!current\_loc())\,.$$

**Remark 6** *The same code call can be used to model a structured variable (see Example 10.1.1 on page 276); the fact language constitutes a uniform way of expressing an agent's information, independently of whether it is represented explicitly or implicitly in the agent's state.*

**Definition 10.1.5 (Agent Consequence Relation)**
*We assume that each agent $\mathfrak{a}$ has an associated consequence relation $Cn_{\mathfrak{a}}$, that takes as input, a set of ground facts belonging to $\mathcal{L}_{\mathfrak{a}}$ and returns as output, a set of ground facts belonging to $\mathcal{L}_{\mathfrak{a}}$. $Cn_{\mathfrak{a}}(F)$ returns as output, all facts implied by the input set $F$, according to the notion of consequence adopted by $\mathfrak{a}$. $Cn_{\mathfrak{a}}$ is required to satisfy the following general axioms:*

1. $Cn_{\mathfrak{a}}(X) \supseteq X$ ;
2. $Cn_{\mathfrak{a}}(Cn_{\mathfrak{a}}(X)) = Cn_{\mathfrak{a}}(X)$ .

*We say that $Cn_{\mathfrak{a}}$ is a* strong *consequence relation if it further satisfies the condition that $Cn_{\mathfrak{a}}^{\mathtt{fol}}(X) \supseteq Cn_{\mathfrak{a}}(X)$ for all $X$, where $Cn_{\mathfrak{a}}^{\mathtt{fol}}(F)$ denotes the set of all consequences of $F$ using the standard consequence notion of first order logic.*

Our definition of agent consequence builds upon the classical notion of an abstract consequence relation, originally proposed by Tarski (1981). Almost all standard provability relations, $\vdash$, for different proof systems ranging from classical logic to modal logics to multivalued logics, induce a function $\mathrm{Cn}^{\vdash}$ as follows:

$$\mathrm{Cn}^{\vdash}(X) \quad =_{def} \quad \{\psi \mid X \vdash \psi\}\,.$$

Conversely, each abstract consequence relation $\mathrm{Cn}_{\mathfrak{b}}$ induces a provability relation

$$S \vdash_{\mathfrak{b}} \psi \quad \textit{if, by definition,} \quad \forall X : S \subseteq X \subseteq \mathcal{L}_{\mathfrak{b}},\ \psi \in \mathrm{Cn}_{\mathfrak{b}}(X)\,.$$

Note a subtle difference between $\vdash_{\mathfrak{b}}$ and $\mathrm{Cn}_{\mathfrak{b}}$: in $S \vdash_{\mathfrak{b}} \phi$, $S$ is treated as a *partial* description of a state, while the argument $X$ of $\mathrm{Cn}_{\mathfrak{b}}$ is taken as a *complete* description of $\mathfrak{b}$'s state.

The notion of strong consequence requires agents to be “rational” in the sense that they do not make inferences that are unsound with respect to classical logic. This is not the only possible form of rationality; some agents may make decisions on the basis of conditions that *normally* or *plausibly* hold; the consequence relation of such agents is not strong. Drawing conclusions requires resources; some agents may want to infer all valid conclusions from their state, while others may only draw inferences obtainable through a bounded number of inferences. This explains why consequence relations need not satisfy $\mathrm{Cn}_{\mathfrak{a}}^{\mathtt{fol}}(X) \subseteq \mathrm{Cn}_{\mathfrak{a}}(X)$.

### 10.1.4 Abstract Service Request Handling

The abstract model of histories described thus far is completely general. It can model agents that respond to many requests simultaneously, by interleaving the corresponding actions; it can also model agents that respond to service requests with arbitrarily complex plans. However, for the purposes of this chapter, we shall make a few simplifying assumptions about service request handling in order to improve readability:

**Assumption 1** Each agent serves one request at a time, and does not interleave its autonomous plans with the evaluation of any service requests. This assumption makes it easier to relate events to the request that triggered them, because the request itself and the sequence of triggered events are contiguous. Each event $e$ is triggered by the last incoming request before $e$. This assumption can be easily eliminated by assigning an “id” to each event, and then ensuring that a response made by agent $\mathfrak{a}$ to agent $\mathfrak{b}$ contains information about the id of the request being responded to.

**Assumption 2** Each request service follows a precise pattern of activities. First, if necessary, the server agent $\mathfrak{a}$ contacts other agents (e.g., to get some missing information, to make a reservation, etc.); second, their answers are waited for; then, the requested actions “internal” to $\mathfrak{a}$ (i.e., not involving communications) are executed, and finally, an answer is returned to the client (if the request specifies some output variables). This pattern makes it easier to identify the different parts of the request service relevant to security enforcement, such as the sequence of actions triggered by a request and the answer to a data request.

The above assumptions correspond to the following formal definition.

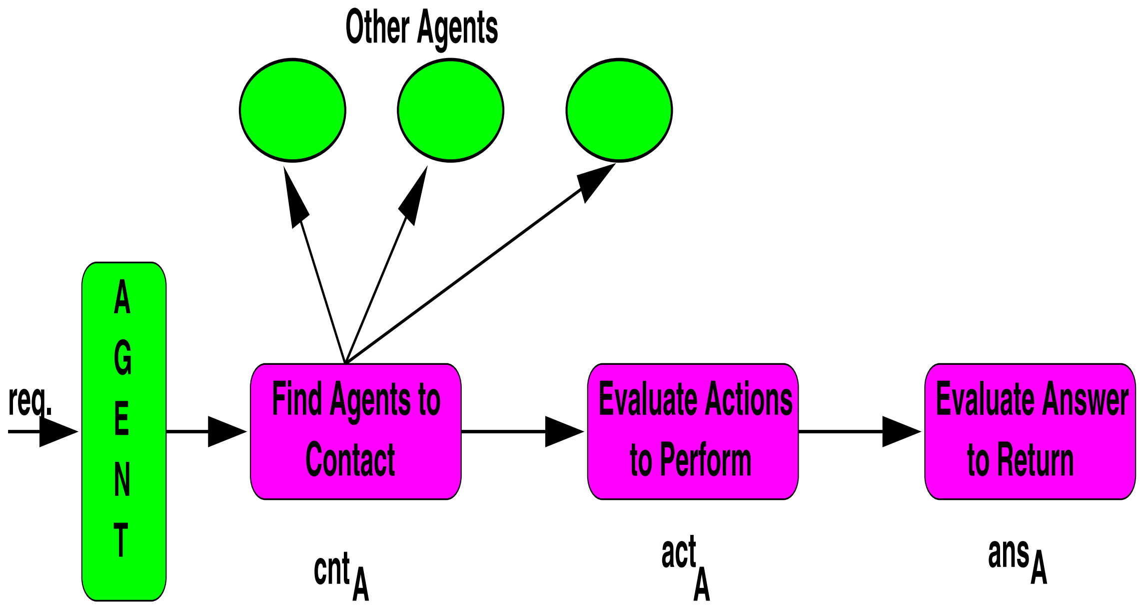



Figure 10.1: Agent Service Evaluation Procedure

**Definition 10.1.6 (Service Request Evaluation Functions)**
*Suppose* $\mathsf{a}$ *is an agent, and* $h$ *ranges over those histories for* $\mathsf{a}$ *whose last event is a request message* $\langle \mathsf{b}_h, \mathsf{a}, \rho_h \rangle$, *where* $\rho_h$ *is a service request.*

- *A* contact evaluation function used by agent $\mathsf{a}$ *is a mapping* $\mathit{cnt}_{\mathsf{a}}$ *that takes* $h$ *as input and returns a sequence of outgoing messages* $\mathit{cnt}_{\mathsf{a}}(h) = \langle \langle \mathsf{a}, \mathsf{c}_1, \rho_1 \rangle, \ldots, \langle \mathsf{a}, \mathsf{c}_n, \rho_n \rangle \rangle$, $(\mathsf{c}_i \neq B)$.
- *An* action evaluation function *is a mapping* $\mathit{act}_{\mathsf{a}}$ *that takes* $h$ *and a sequence of incoming messages* $Resp(\mathit{cnt}_{\mathsf{a}}(h)) = \langle \langle \mathsf{c}_1, A, Ans_1 \rangle, \ldots, \langle \mathsf{c}_n, A, Ans_n \rangle \rangle$ *(corresponding to the response received from other agents to the outgoing requests in* $\mathit{cnt}_{\mathsf{a}}(h)$*) as input, and returns a sequence of actions executable by* $\mathsf{a}$.
- *A* data evaluation function *is a mapping* $\mathit{ans}_{\mathsf{a}}$ *that takes* $h$ *and* $Resp(\mathit{cnt}_{\mathsf{a}}(h))$ *as described above as inputs, and returns an answer* $\{f_1, \ldots, f_m\}$.

*Finally, the* (global) service evaluation function *is*

$$\zeta_{\mathsf{a}}(h) = h \cdot \mathit{cnt}_{\mathsf{a}}(h) \cdot Resp(\mathit{cnt}_{\mathsf{a}}(h)) \cdot \mathit{act}_{\mathsf{a}}(h, Resp(\mathit{cnt}_{\mathsf{a}}(h))) \cdot \langle \mathsf{a}, \mathsf{b}_h, \mathit{ans}_{\mathsf{a}}(h, Resp(\mathit{cnt}(h))) \rangle .$$

*The triple* $\langle \mathit{cnt}_{\mathsf{a}}, \mathit{act}_{\mathsf{a}}, \mathit{ans}_{\mathsf{a}} \rangle$ *will be referred to as a* service request evaluation policy. *With a slight abuse of notation, the service request evaluation policy will be sometimes be identified with the global service evaluation function* $\zeta_{\mathsf{a}}$.

Figure 10.1 shows a diagrammatic view of how an agent evaluates a service request. Intuitively, $\zeta_{\mathsf{a}}(h)$ extends $h$ with the events triggered by the most recent request $\langle \mathsf{b}_h, \mathsf{a}, \rho_h \rangle$ (which may be found at the end of $h$); the action part and the data part are determined on the basis of the inputs modeled by $Resp(\mathsf{cnt}_{\mathsf{a}}(h))$. The answer $\mathsf{ans}_{\mathsf{a}}(h, Resp(\mathsf{cnt}_{\mathsf{a}}(h)))$ is the data returned by $\mathsf{a}$.

In *IMPACT* agents, $\mathsf{cnt}_{\mathsf{a}}$ and $\mathsf{act}_{\mathsf{a}}$ would be defined through *agent programs* (see Chapter 6), while $\mathsf{ans}_{\mathsf{a}}$ would be implemented as a set of *service rules* (Definition 4.6.1 on page 79).

# Chapter 11

# Complexity Results

In the previous chapters, we have described our approach to a *software agent system*, which involves a number of different components to make an agent work in practice. As we have seen, a large number of different tasks must be or should be handled by an agent, including *location of external services*, *decision making*, *meta-reasoning* as well as *uncertain* and *temporal reasoning*. All these capabilities require proper conceptualization and formalization, which we have given in this book.

However, to put the *IMPACT* system at work, we need algorithms for solving the particular computational problems which emerge from these various tasks. And, of course, these algorithms should be decent ones, in the sense that the problems are solved in an efficient way. This calls us for the development of such algorithms.

But before we start off developing such algorithms, we have to commit to a clear understanding what "efficient" really means. Commonly,

*computable in polynomial time*

is understood as a synonym of "efficiently computable." Not all problems are solvable in polynomial time, and for many of them, this is provably the case. However, there is a vast body of important problems in computer science, for which it is not formally proven yet that no polynomial time algorithms do exist for solving them, even though there is strong evidence that no such algorithms do exist. An example of such problems are the NP-complete problems, which are solvable by (deterministic) algorithms in exponential time. The issue of whether any such problem can be solved by a polynomial time algorithm amounts to the famous open P = NP question.

Extending the work on NP-complete problems, the field of structural complexity theory has developed a number of techniques and tools to assess the "difficulty" of a computation problem, such that we may find how difficult solving a particular problem is. Moreover, these techniques may help us in judging whether a particular algorithm is to be considered reasonable or, under an appropriate notion, even optimal given the intrinsic problem complexity. Rather than simply classifying a problem as either "solvable in polynomial time" or by some evidence "not solvable in polynomial time, and thus intractable," a closer look on the computational cost of solving a problem should be given, as this might reveal interesting information. Such information may be, for example, which type of algorithm is appropriate for solving a problem, or which related problems do exist to which a problem at hand can be efficiently reduced to.

In this chapter, we turn to this question and address the computational cost of tasks which have to be performed by *IMPACT* agents. An exhaustive study which covers all these tasks would be an ambitious program, however, and would fill yet another book. For this reason, we focus here on a central task of the *IMPACT* architecture, namely, on agent decision making. In fact, this task is at the heart of the *IMPACT* agent system—without the agent decision layer, *IMPACT* agents are deemed to idly wait and not providing any services. Agent programs, which we have described in Chapter 6, along with different notions status sets provide a formal semantics for an agent's behavior. For an implementation of decision making, we need appropriate algorithms to compute these semantics, which suggests to perform a study of the computational complexity of the semantics.

Our plan for this chapter is as follows. As we will encounter some complexity classes which the casual reader is usually not familiar with, we recall some basic concepts of the theory of NP-completeness and beyond in the first section. We then precisely formulate in Section 11.2 the problems whose complexity is investigated, and which underlying assumptions are made in our analysis—this is indispensable for a complexity assessment. After that, we present in Section 11.3 a summary of the results and the bottom line of the conclusions that can be drawn. The reader who is merely interested in such a summary and a general discussion, may safely skip the rest of this chapter, in which the technical results are established. However, she might browse Sections 11.4 and 11.5 if she is interested in algorithms by which various kinds of status sets are computed, since such algorithms are developed in the course of the complexity analysis.

## 11.1 Complexity Classes

We assume at this point that the reader has some basic knowledge about computational complexity; the texts (Garey and Johnson 1979; Johnson 1990; Papadimitriou 1994) are good sources, which the reader who has little experience on complexity may consult for further background.

### 11.1.1 Decision Problems

The attempt to distinguish easy from hard problems has led to the widely accepted notion of *tractability* as the property that a problem can be *solved in polynomial time*, measured in the size of the problem input. The mathematical study of problem solving has for a long time centered around the simplest type of problems, namely *decision problems*.

**Definition 11.1.1 (Decision Problems)**
*Formally, a decision problem* $\Pi$ *consists of a pair* $(D,Y)$ *of a set* $D$ *of* problem instances, *which are strings from a language over a finite alphabet, and a subset* $Y$ *of* $D$ *which are called* "Yes"-instances. *The strings in* $D \setminus Y$ *are called* "No"-instances.[1]

*An algorithm correctly solves a decision problem* $\Pi = (D,Y)$*, if it outputs, given an instance* $I$ *from* $D$*, "Yes" if* $I \in Y$ *holds, and outputs "No" otherwise.*

The *complexity* of a decision problem, in terms of a resource such as computation time or work space, is understood as the cost of an optimal algorithm (assuming some computation model) for solving this problem. Abstracting from this optimal cost, different complexity classes have been defined which allow to roughly characterize the difficulty of a problem in the large.

**Definition 11.1.2 (Deterministic Complexity Classes)**
*The class* P *contains all decision problems* $\Pi$ *that are solvable in polynomial time on a deterministic Turing machine (DTM), i.e., there exists a DTM* $M$*, which solves* $\Pi$ *and such that the running time of* $M$ *is bounded by a polynomial in the length* $|I|$ *of the input* $I$*. Similarly, the class* PSPACE *contains*

[1] Note that decision problems are often defined as formal languages $L$ over a finite alphabet, such that the strings in $L$ are the "Yes"-instances and all other strings correspond to the "No"-instances and the strings not encoding any problem instance. Under the usual assumption that the proper encodings of problem instances are easily recognized, the two formal definitions of problems are for our purposes equivalent.

*all decision problems which are solvable by some DTM in polynomial work space, i.e., the number of tape cells visited by the workhead of $M$ is bounded by a polynomial in the length of the input $I$.*

Further complexity classes, such as EXPTIME and EXPSPACE, are defined by using other bounds on time and space, respectively.

For example, the problem ACYCLIC = $(D,Y)$ where $D$ consists of the encodings of directed graphs (according to some standard encoding) and $Y$ contains the (encodings of) acyclic graphs in $D$, can be solved in polynomial time. Informally, we say that the problem, given a graph, deciding whether the graph is acyclic, is possible in polynomial time, and thus the problem is in P.

A decision problem $\Pi = (D,Y)$ is solved by a nondeterministic Turing machine (NTM), if for every instance $I$ of $\Pi$ there is a run of $M$ such that $M$ outputs "Yes" if and only if $I$ is a yes-instance. Note that no symmetric behavior is required for no-instances–$M$ need even not halt on such instances.

**Definition 11.1.3 (Nondeterministic Turing Machines)**
*A decision problem $\Pi$ is solvable in nondeterministic polynomial time, if $\Pi$ is solved by some NTM $M$ such that a polynomial exists which bounds for each "Yes"-instance the time of the shortest run of $M$ which outputs "Yes." The class* NP *contains all decision problems which are solvable in nondeterministic polynomial time.*

For example, the well-known satisfiability problem SAT = $(D,Y)$, whose problem instances $D$ are propositional formulas $\phi = \bigwedge_{i=1}^{m} C_i$ in conjunctive normal form (CNF), and whose "Yes"-instances are the satisfiable formulas in $D$, is a in NP. Indeed, an assignment $\sigma$ of truth values to the propositional variables $x_1, \ldots, x_n$ occurring in $\phi$ such that it satisfies $\phi$, can be nondeterministically generated by the Turing machine (i.e., "guessed") in $n$ steps, and the verification of the guess (i.e., checking whether $\sigma$ satisfies $\phi$), can be done efficiently.

Clearly, all problems in P are in NP, and as can be seen, all problems in NP are in PSPACE. It is to date unknown, however, whether any of the reverse containments holds, even though it is widely believed that none of them actually is true. Problems that are candidates for lying in the difference between NP and P are the NP-complete problems, which we introduce next.

The notion of completeness of a problem has been introduced in order to characterize the "hardest" problems in a complexity class. This notion involves the concept of *problem reduction*, which is as follows.

**Definition 11.1.4 (Problem Reduction, Hardness, and Completeness)**
*A decision problem $\Pi$ is polynomial-time reducible to a decision problem $\Pi'$, if there is a function $f$ which is computable in polynomial time, such that for every instance $I$ of $\Pi$, $f(I)$ is an instance of $\Pi'$ and $f(I)$ is a "Yes"-instance of $\Pi'$ just if $I$ is a "Yes"-instance of $\Pi$.*

*A problem $\Pi$ is said to be* hard *for a class of problems $C$ (i.e., $C$-*hard*), if every problem in $C$ is polynomial-time reducible to $\Pi$. Furthermore, a problem $\Pi$ is* complete *for $C$, if $\Pi$ is hard for $C$ and $\Pi$ belongs to $C$.*

For example, SAT is a well-known NP-complete problem. Many restrictions of SAT have been shown to be NP-complete as well. One of them is monotone 3SAT (M3SAT), which contains those instances $\phi = \bigwedge_{i=1}^{m} C_i$ of SAT such that each clause $C_i = L_{i,1} \vee L_{i,2} \vee L_{i,3}$ is a disjunction of three literals $L_{i,j}$, and either all literals in $C_i$ are positive atoms or all are negated atoms. The complement of SAT (i.e., deciding unsatisfiability of a propositional CNF $\phi$) is a well-known co-NP-complete problem, as is deciding whether a propositional formula $\phi = \bigvee_{i=1}^{m} D_i$ in disjunctive normal form (DNF) is a tautology. Similarly to SAT, this remains true if each disjunct $D_i = L_{i,1} \wedge L_{i,2} \wedge L_{i,3}$ is a conjunction of three literals $L_{i,j}$, and either all $L_{i,j}$ in $D_i$ are atoms or all are negated atoms. We shall refer to such $\phi$ as *monotone 3DNF* (M3DNF) formulas.

Knowing that a problem is NP-complete or co-NP-complete is quite useful, because this tells us that one should not spend much time for developing a polynomial time algorithm (unless she wants to solve the P = NP problem, and become a famous computer scientist!), and rather look for restrictions on the problem under which a polynomial algorithm is feasible, or for a suitable heuristics which works well in practice.

It appeared that a number of problems reside between NP and PSPACE, and that there are proper, natural models of computation which express that computational nature of these problems. The most important such problems are the ones which reside in the polynomial hierarchy, which has been built upon the classes P and NP (= $\Sigma_1^{\mathrm{P}}$), by allowing the use of an oracle (i.e., a subprogram) for deciding problems instantaneously.

**Definition 11.1.5 (Oracle Turing Machine)**
*An oracle Turing machine (OTM) is a Turing machine equipped with a query tape, on which queries to a fixed problem can be posed. Any such query is answered upon entering a designated query state in one step.*

*For a complexity class $C$, we denote by $\mathrm{P}^C$ (resp., $\mathrm{NP}^C$) the class of all decision problems which can be solved by a deterministic (resp., nondeterministic) OTM wich the help of an oracle in $C$.*

In particular, $\mathrm{P}^{\mathrm{NP}}$ (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 oracle classes $C$ which we will encounter, we may also say that an oracle for solving any problem in $C$ (a $C$-oracle, for short) is available, rather than only a fixed one. The reason is that all instances of problems in $C$ can be efficiently transformed into instances of a single (fixed) $C$-complete problem.

The classes of the polynomial hierarchy, which is embedded in PSPACE, are shown in the left part of Figure 11.1 on the next page. In this figure, 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., any hardest problem in $C_2$ is more difficult than each single problem in $C_1$. For the decisional classes, the arcs in Figure 11.1 on the facing page actually denote inclusions, i.e., the transformation of problems in $C_1$ to problems in $C_2$ is by means of the identity.

All classes of the polynomial hierarchy have complete problems, including canonical complete problems in terms of evaluating quantified Boolean formulas. A *quantified Boolean formula* (QBF) is a generalized propositional formula, in which each propositional variable $x_i$ ranges over $\{$ *true*, *false* $\}$ and is governed either by an existential ($\exists$) or a universal ($\forall$) quantifier. The truth value of such a formula is obtained by eliminating all quantifiers in the obvious way and evaluating the resulting variable-free formula. For example, $\forall y_1, y_2 \exists x_1, x_2 (x_1 \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge y_1)$ is a QBF. It evaluates to *false*, since on assigning e.g. *false* to both $y_1$ and $y_2$, the remaining formula is unsatisfiable.

Evaluating a given QBF $\Phi$ is a classical PSPACE-complete problem. Syntactic restrictions on $\Phi$ provide problems complete for the $\Sigma_k^{\mathrm{P}}$ and $\Pi_k^{\mathrm{P}}$ classes of the polynomial hierarchy. In particular, deciding whether a QBF of the form $\exists Y^1 \forall Y^2 \cdots Q_k Y^k \phi$, where the $Y^i$ are sets of variables and the quantifiers $Q_i$ in front of them alternate, evaluates to *true* is a well-known $\Sigma_k^{\mathrm{P}}$-complete problem. Dually, deciding whether a QBF of form $\forall Y^1 \exists Y^2 \cdots Q_k Y^k \phi$ evaluates to true is $\Pi_k^{\mathrm{P}}$-complete.

# Chapter 12

# Implementing Agents

We have developed a software environment called the *IMPACT* Agent Development Environment (*IADE* for short). *IADE* provides a set of software tools using which, an agent developer may configure his or her agent. As we have already seen in Chapter 11, the diverse semantics supported by *IMPACT* have widely varying complexities. For positive agent programs, the rational, reasonable, F-preferential rational, and F-preferential reasonable status set semantics are polynomially computable even when integrity constraints are present. However, as we have already seen via the STORE, CHAIN and CFIT examples, encoding agents naturally seems to require negation. In the presence of negation, Chapter 11 tells us that the "cheapest" semantics to compute are the feasible status set semantics and the reasonable status set semantics, as computing such status sets falls within the class FNP when integrity constraints are present. Though this is intractable, of all the other semantics, this is the most tractable. Furthermore, we have argued in Chapter 6 that the reasonable status set semantics has epistemically more desirable qualities than the feasible status set semantics. As a consequence, the current version of the *IMPACT* Agent Development Environment supports the computation of the reasonable status set semantics.

Notwithstanding this, as remarked above, computing reasonable status sets of arbitrary agent programs with negation falls in the class FNP. As agents need to compute reasonable status sets on a continuous basis, the *IADE* implements a class of agents for which computing reasonable status sets is polynomial in the presence of integrity constraints, and under certain conditions on the integrity constraints, action constraints, and code calls. Such agents are called *regular agents*, and their associated agent programs, integrity constraints and action constraints are called regular agent programs, regular integrity constraints, and regular action constraints, respectively.

The main aim of this chapter is to:

- First define (in Section 12.1), a class of agents called *weak regular agents* that serve as a stepping stone to later defining regular agents.
- Derive various theoretical properties of weak regular agents that make the design of a computation procedure to compute regular agents polynomial. This is done in Section 12.2.
- Extend the definition of weak regular agents to define regular agents—the central contribution of this chapter. This is done in Section 12.3.
- Describe two algorithms used when a regular agent is compiled, and show that these algorithms are correct. This is done in Section 12.4.
- Describe a special computation package we have built called the *Query Maintenance Package* (or *QMP* for short). This package specifies

  - Data structures used by *IADE* to ensure that status sets are succinctly (implicitly) represented in a state independent compact data structure and
  - Define functions that manipulate the above *QMP* data structures to explicitly compute a status set and/or explicitly compute a new set of actions to be done when the agent's state is changed through the receipt of messages from other agents.

  These contributions are described in Section 12.5.

- Describe in detail our implementation of *IADE* and how an agent developer might use it to develop agents. This is done in Section 12.6.

- Provide experimental results that describe the behavior of our algorithms. This is done in Section 12.7.

## 12.1 Weakly Regular Agents

In this section, we will first identify a class of agent programs called *weak regular* agent programs (*WRAPs* for short)—later, these are extended to *regular agent programs* (*RAPs* for short) that are guaranteed to possess polynomially computable reasonable status sets. *WRAPs* are characterized by three basic properties:

1. *Strong Safety.* In addition to the safety requirement on rules introduced in Chapter 4 (Definition 4.2.5), code call conditions are required to satisfy some additional conditions which ensure that they always return finite answers.

2. *Conflict-Freedom.* The set of rules in a *WRAP* should not lead to conflicts—for example, the rules must not force an agent to do something it is forbidden to do.

3. *Deontic Stratifiability.* This is a property in the spirit of stratification in logic programs (Apt, Blair, and Walker 1988), which prevents problems with negation in rule bodies. However, as we will see, deontic stratification is more complex than ordinary stratification due to (i) the presence of deontic modalities in rule bodies, and (ii) the fact that rules can be inconsistent due to conflicting modalities in rule heads.

### 12.1.1 Strong Safety

As described in Chapter 4, for a code-call to be executable at run-time, its arguments must be instantiated. Safety is a *compile-time* check that ensures that all code calls generated at *run-time* have instantiated parameters. However, executability of a code call condition does not depend solely on safety. For example, consider the simple code call condition

$$\mathbf{in}(\mathtt{X}, \mathtt{math}{:}geq(\mathtt{25})).$$

This code call condition attempts to execute a function that computes all integers greater than or equal to 25. Though this code call condition is safe, it leads to an infinite set of possible answers, leading to non-termination. In fact, the problem is even more insidious. Consider, for instance, the code call condition

$$\mathbf{in}(\mathtt{X}, \mathtt{math}{:}geq(\mathtt{25}))\,\&\,\mathbf{in}(\mathtt{Y}, \mathtt{math}{:}square(\mathtt{X}))\,\&\,Y \leq 2000.$$

This code call condition may find all numbers that are less than 2000 and that are squares of an integer greater than or equal to 25. Clearly, over the integers there are only finitely many ground substitutions that cause this code call condition to be true. Furthermore, this code call condition is safe. However, its evaluation may never terminate. The reason for this is that safety requires that we first compute the set of all integers that are greater than 25, leading to an infinite computation. This means that in general, we must impose some restrictions on code call conditions to ensure that they are finitely evaluable.

Suppose the developer of an agent examines the code calls supported by a given data structure and specifies which of them are finite and which are not. As is well known, determining whether a function is finite or not is undecidable (Rogers Jr. 1967), and hence, input from the agent developer is imperative.

**Definition 12.1.1 (Binding Pattern)**
*Suppose we consider a code call $\mathcal{S}:f(\mathtt{a_1},\ldots,\mathtt{a_n})$ where each $\mathtt{a_i}$ is of type $\tau_i$. A* binding pattern *for $\mathcal{S}:f(\mathtt{a_1},\ldots,\mathtt{a_n})$ is an n-tuple $(bt_1,\ldots,bt_n)$ where each $bt_i$ (called a* binding term*) is either:*

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

We require that the agent developer must specify a *finiteness* predicate that may be defined via a *finiteness table* having two columns—the first column is the name of the code call, while the second column is a binding pattern for the function in question.[1] Intuitively, suppose we have a row of the form

$$\langle \mathcal{S}:f(\mathtt{a_1},\mathtt{a_2},\mathtt{a_3}),(\flat,5,\flat)\rangle$$

in the finiteness table. Then this row says that the answer returned by any code call of the form $\mathcal{S}:f(-,5,-)$ is finite. In other words, as long as the second argument of this code call is 5, the answer returned is finite, irrespective of the values of the first and third arguments. Clearly, the same code call may occur many times in a finiteness table with different binding patterns.

**Example 12.1.1 (Finiteness Table for AutoPilot Agent in CFIT Example)**
*An example of a finiteness table is given below.*

| ***Code Call*** | ***Binding Pattern*** |
|---|---|
| $\mathtt{autoPilot}:\mathit{readGPSData}(\mathtt{SensorId})$ | $(\flat)$ |
| $\mathtt{autoPilot}:\mathit{calculateLocation}(\mathtt{Location},\mathtt{FlightRoute},\mathtt{Speed})$ | $(\flat,\flat,\flat)$ |
| $\mathtt{autoPilot}:\mathit{calculateNFlightRoutes}(\mathtt{CurrentLocation},\mathtt{No\_go},\mathtt{N})$ | $(\flat,\flat,1)$ |
| $\mathtt{autoPilot}:\mathit{calculateNFlightRoutes}(\mathtt{CurrentLocation},\mathtt{No\_go},\mathtt{N})$ | $(\flat,\flat,2)$ |
| $\mathtt{autoPilot}:\mathit{calculateNFlightRoutes}(\mathtt{CurrentLocation},\mathtt{No\_go},\mathtt{N})$ | $(\flat,\flat,3)$ |

*This indicates that $\mathtt{autoPilot}:\mathit{readGPSData}()$ and $\mathtt{autoPilot}:\mathit{calculateLocation}()$ always return a finite number of answers. The code call*

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

*autoPilot:calculateNFlightRoutes(CurrentLocation,No_go,N) returns up to N flight routes when $N \neq 0$. If $N = 0$, then an infinite number of flight routes (which start at CurrentLocation and avoid the given No_go areas) may be returned. Our finiteness table above indicates that when $1 \leq N \leq 3$, autoPilot:calculateNFlightRoutes() will only return a finite number of answers. Notice that this table is incomplete since it does not indicate that a finite number of answers will be returned when $N > 3$.*

From the fact that any code call of the form $\mathcal{S}:f(-,5,-)$ has a finite answer, we should certainly be able to infer that the code call $\mathcal{S}:f(20,5,17)$ has a finite answer. In order to make this kind of inference, we need to associate an ordering on binding patterns. We say that $\flat \leq val$ for all values, and take the reflexive closure. We may now extend this $\leq$ ordering to binding patterns.

**Definition 12.1.2 (Ordering on Binding Patterns)**
*We say a binding pattern $(bt_1,\ldots,bt_n)$ is* equally or less informative *than another binding pattern $(bt'_1,\ldots,bt'_n)$ if, by definition, for all $1 \leq i \leq n$, $bt_i \leq bt'_i$.*

We will say $(bt_1,\ldots,bt_n)$ is *less informative* than $(bt'_1,\ldots,bt'_n)$ *if and only if* it is equally or less informative than $(bt'_1,\ldots,bt'_n)$ and $(bt'_1,\ldots,bt'_n)$ is not equally or less informative than $(bt_1,\ldots,bt_n)$. If $(bt'_1,\ldots,bt'_n)$ is less informative than $(bt_1,\ldots,bt_n)$, then we will say that $(bt_1,\ldots,bt_n)$ is *more informative* than $(bt'_1,\ldots,bt'_n)$.

Suppose now that the developer of an agent specifies a finiteness table FINTAB. The following definition specifies what it means for a specific code call atom to be considered finite w.r.t. FINTAB.

**Definition 12.1.3 (Finiteness)**
*Suppose FINTAB is a finite finiteness table , and $(bt_1,\ldots,bt_n)$ is a binding pattern associated with the code call $\mathcal{S}:f(\cdots)$. Then FINTAB is said to* entail the finiteness of $\mathcal{S}:f(\mathtt{bt_1},\ldots,\mathtt{bt_n})$ *if, by definition, there exists an entry of the form $\langle \mathcal{S}:f(\ldots),(bt'_1,\ldots,bt'_n)\rangle$ in FINTAB such that $(bt_1,\ldots,bt_n)$ is more informative than $(bt'_1,\ldots,bt'_n)$.*

Below, we show how the finiteness table introduced for the autoPilot agent entails the finiteness of some simple code calls.

**Example 12.1.2 (Finiteness Table)**
*Let FINTAB be the finiteness table given in Example 12.1.1 on the page before. Then FINTAB entails the finiteness of autoPilot:readGPSData(5) and autoPilot:calculateNFlightRoutes($\langle$221,379,433$\rangle$,$\emptyset$,2) but it does not entail the finiteness of autoPilot:calculateNFlightRoutes($\langle$221,379,433$\rangle$,$\emptyset$,0) (since this may have an infinite number of answers) or autoPilot:calculateNFlightRoutes($\langle$221,379,433$\rangle$,$\emptyset$,5) (since FINTAB is not complete).*

According to the above definition, when we know that FINTAB entails the finiteness of the code call $\mathcal{S}:f(\mathtt{bt_1},\ldots,\mathtt{bt_n})$, then we know that every code call of the form $\mathcal{S}:f(\ldots)$ whose arguments satisfy the binding requirements are guaranteed to yield finite answers. However, defining strong safety of a code call condition is more complex. For instance, even if we know that $\mathcal{S}:f(\mathtt{t_1},\ldots,\mathtt{t_n})$ is finite, the code call atom **not_in**$(\mathtt{X},\mathcal{S}:f(\mathtt{t_1},\ldots,\mathtt{t_n}))$ may have an infinite answer. Likewise, comparison conditions such as $\mathtt{s} > \mathtt{t}$ may have finite answers in some cases and infinite answers in other cases, depending upon whether we are evaluating variables over the reals, the integers, the positive reals, the positive integers, etc. In the sequel, we make two simplifying assumptions, though both of them can be easily modified to handle other cases:

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

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

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

   It is easy to see that the positive integers have this property, as do the set of all strings ordered by the standard lexicographic ordering. (Later, we will show how this property may be relaxed to accommodate the reals, the negative integers, and so on.)

We are now ready to define strong safety.

**Definition 12.1.4 (Strong Safety)**
*] A safe code call condition $\chi = \chi_1 \& \ldots \& \chi_n$ is* strongly safe *w.r.t. a list $\vec{X}$ of root variables if, by definition, there is a permutation $\pi$ witnessing the safety of $\chi$ modulo $\vec{X}$ such that for each $1 \leq i \leq n$, $\chi_{\pi(i)}$ is strongly safe modulo $\vec{X}$, where strong safety of $\chi_{\pi(i)}$ is defined as follows:*

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

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

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

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

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

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

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

Algorithm **safe_ccc** defined in Chapter 4 may easily be modified to handle a strong safety check, by replacing the test "select all $\chi_{i_1}, \ldots, \chi_{i_m}$ from $L$ such that $\chi_{i_j}$ is safe modulo $\vec{X}$" in step (4) of that algorithm by the test "select all $\chi_{i_1}, \ldots, \chi_{i_m}$ from $L$ such that $\chi_{i_j}$ is strongly safe modulo $\vec{X}$." As a consequence, it is not hard to see that strong safety can be checked in time proportional to the product of the time taken to check safety and the time to look up items in FINTAB. The former is quadratic (using appropriate data structures, even linear) in the length of the code call condition, and the latter is linear in the number of entries in FINTAB.

**Definition 12.1.5 (Strongly Safe Agent Program)**
*A rule $r$ is* strongly safe *if, by definition, it is safe, and $B_{cc}(r)$ is a strongly safe code call condition. An agent program is* strongly safe *if, by definition, all rules in it are strongly safe.*

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

### 12.1.2 Conflict-Freedom

In the preceding section, we have argued that for an agent program to be considered a *WRAP*, it must be strongly safe w.r.t. the finiteness table FINTAB specified by the agent developer. In this section, we specify another condition for being a *WRAP*—namely that the agent program must be guaranteed to never encounter a conflict.

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

**Definition 12.1.6 (Conflicting Modalities)**
*Given two action modalities $Op, Op' \in \{\mathbf{P}, \mathbf{F}, \mathbf{O}, \mathbf{Do}, \mathbf{W}\}$ we say that $Op$* conflicts with *$Op'$ if, by definition, there is an entry "$\times$" in the following table at row $Op$ and column $Op'$:*

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

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

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

**Definition 12.1.7 (Conflicting Action Status Literals)**
*Suppose $L_i, L_j$ are two action status literals. $L_i$ is said to* conflict with *$L_j$ if, by definition,*

- *$L_i, L_j$ are unifiable and their modalities conflict, or*

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

| $\mathit{Op} \setminus \neg \mathit{Op}'$ | $\neg$**P** | $\neg$**F** | $\neg$**O** | $\neg$**W** | $\neg$**Do** |
|---|---|---|---|---|---|
| **P** | × | | | | |
| **F** | | × | | | |
| **O** | × | | × | | × |
| **W** | | | | × | |
| **Do** | × | | | | × |

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

**Definition 12.1.8 (Conflicting Rules w.r.t. a State)**
*Consider two rules* $r_i, r_j$ *(whose variables are standardized apart) having the form*

$$\begin{array}{rcl} r_i : \mathit{Op}_i(\alpha(\vec{t})) & \leftarrow & B(r_i) \\ r_j : \mathit{Op}_j(\beta(\vec{t'})) & \leftarrow & B(r_j) \end{array}$$

*We say that* $r_i$ *and* $r_j$ conflict *w.r.t. an agent state* $O_{\mathcal{S}}$ *if, by definition,* $\mathit{Op}_i$ *conflicts with* $\mathit{Op}_j$*, and there is a substitution* $\theta$ *such that:*

- $\alpha(\vec{t}\theta) = \beta(\vec{t'}\theta)$ *and*
- $(B_{cc}(r_i) \wedge B_{cc}(r_j))\theta\gamma$ *is true in* $O_{\mathcal{S}}$ *for some substitution* $\gamma$ *that causes* $(B_{cc}(r_i) \wedge Bcc(r_j))\theta$ *to become ground and*
- *If* $\mathit{Op}_i \in \{\mathbf{P}, \mathbf{Do}, \mathbf{O}\}$ *(resp.,* $\mathit{Op}_j \in \{\mathbf{P}, \mathbf{Do}, \mathbf{O}\}$*) then* $\alpha(\vec{t}\theta)$ *(resp.,* $\beta(\vec{t'}\theta)$*) is executable in* $O_{\mathcal{S}}$*, and*
- $(B_{as}(r_i) \cup B_{as}(r_j))\theta$ *contains no pair of conflicting action status literals.*

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

**Definition 12.1.9 (Conflict Free)**
*An agent program,* $\mathcal{P}$*, is said to be* conflict free *if and only if it satisfies two conditions:*

1. *For every possible agent state* $O_{\mathcal{S}}$*, there is no pair* $r_i, r_j$ *of conflicting rules in* $\mathcal{P}$*.*
2. *For any rule* $\mathit{Op}_i(\alpha(\vec{t})) \leftarrow \ldots, (\neg)\mathit{Op}_j(\vec{t'}), \ldots$ *in* $\mathcal{P}$*,* $\mathit{Op}_i(\alpha(\vec{t}))$ *and* $(\neg)\mathit{Op}_j(\alpha(\vec{t'}))$ *do not conflict.*

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

**Theorem 12.1.1 (Undecidability of Conflict Freedom Checking)**
*The problem of deciding whether an input agent program $\mathcal{P}$ satisfies the first condition of conflict-freedom is undecidable. Hence, the problem of deciding whether an input agent program $\mathcal{P}$ is conflict free is undecidable.*

**Proof:** The undecidability of this problem is inherited from the undecidability of the problem whether a function $\mathtt{f} \in \mathcal{F}$ from a software code $\mathcal{S} = (\mathcal{T}, \mathcal{F})$ returns a particular value on at least one agent state $O_{\mathcal{S}}$. We may choose for $\mathcal{S}$ a standard relational database package, and let $\mathtt{f}$ be a Boolean query on the database written in SQL. Then, it is undecidable whether $\mathtt{f}$ evaluates to true over some (finite) database, i.e., agent state $O_{\mathcal{S}}$; this follows from well-known results in the area of database theory. In particular, relational calculus is undecidable, and every query in relational calculus an be expressed in SQL; see (Abiteboul, Hull, and Vianu 1995). Now, define the rules:

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

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

The ability to check whether an agent program is conflict free is very important. When an agent developer builds an agent, in general, s/he cannot possibly anticipate all the future states of the agent. Thus, the developer must build guarantees into the agent which ensure that no conflicts can possibly arise in the future. However, in general, as checking conflict freedom of agent programs is undecidable, we cannot hope for an effective algorithm to check conflict freedom. However, there are many possible ways to define *sufficient* conditions on agent programs that guarantee conflict freedom. If an agent developer encodes his agent program in a way that satisfies these sufficient conditions, then he is guaranteed that his agent is going to be conflict free. The concept of a conflict freedom implementation defined below provides such a mechanism.

**Definition 12.1.10 (Conflict-Freedom Test)**
*A* conflict-freedom test *is a function* cft *that takes as input any two rules $r_1, r_2$, and provides a boolean output such that: if $\mathsf{cft}(r_1, r_2) = \mathbf{true}$, then the pair $r_1, r_2$ satisfies the first condition of conflict freedom.*

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

**Definition 12.1.11 (Conflict-Free Agent Program w.r.t. cft)**
*An agent program $\mathcal{P}$ is* conflict free w.r.t. cft *if and only if for all pairs of distinct rules $r_i, r_j \in \mathcal{P}$, $\mathsf{cft}(r_i, r_j) = \mathbf{true}$, and all rules in $\mathcal{P}$ satisfy the second condition in the definition of conflict free programs.*

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

**Example 12.1.3 (Head-CFT, $\mathsf{cft}_h$)**
*Let $r_i$,$r_j$ be two rules of the form*

$$\begin{aligned} r_i : \mathit{Op}_i(\alpha(\vec{t})) &\leftarrow B(i) \\ r_j : \mathit{Op}_j(\beta(\vec{t'})) &\leftarrow B(j). \end{aligned}$$

*Now let the head conflict-freedom test $\mathit{cft}_h$ be as follows,*

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

**Example 12.1.4 (Body Code Call CFT, $\mathsf{cft}_{bcc}$)**
*Let us continue using the same notation as in Example 12.1.3. Now let the body-code conflict-freedom test $\mathit{cft}_{bcc}$ be as follows,*

$$\mathit{cft}_{bcc}(r_i,r_j) = \begin{cases} \textbf{true}, & \text{if either } \mathit{Op}_i, \mathit{Op}_j \text{ do not conflict, or} \\ & \alpha(\vec{t}) \text{ and } \beta(\vec{t'}) \text{ are not unifiable, or} \\ & \mathit{Op}_i, \mathit{Op}_j \text{ conflict and } \alpha(\vec{t}), \beta(\vec{t'}) \text{ are unifiable via mgu } \theta \text{ and} \\ & \text{there is a pair of contradictory code call atoms in } B_{cc}(r_1\theta), B_{cc}(r_2\theta); \\ \textbf{false} & \text{otherwise.} \end{cases}$$

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

**Example 12.1.5 (Body-Modality-CFT, $\mathsf{cft}_{bm}$)**
*The body-modality conflict-freedom test is similar to the previous one, except that action status atoms are considered instead. Now let $\mathit{cft}_{bm}$ be as follows,*

$$\mathit{cft}_{bcc}(r_i,r_j) = \begin{cases} \textbf{true} & \text{if } \mathit{Op}_i, \mathit{Op}_j \text{ do not conflict or} \\ & \alpha(\vec{t}), \beta(\vec{t'}) \text{ are not unifiable or} \\ & \mathit{Op}_i, \mathit{Op}_j \text{ conflict, and } \alpha(\vec{t}), \beta(\vec{t'}) \text{ are unifiable via mgu } \theta \text{ and} \\ & \text{literals } (\neg)\mathit{Op}_i\alpha(\vec{t''}) \text{ in } B_{as}(r_i\theta) \text{ for } i = 1,2 \text{ exist} \\ & \text{such that } (\neg)\mathit{Op}_1 \text{ and } (\neg)\mathit{Op}_2 \text{ conflict;} \\ \textbf{false} & \text{otherwise.} \end{cases}$$

**Example 12.1.6 (Precondition-CFT, $\mathsf{cft}_{pr}$)**
*Often, we might have action status atoms of the form $\mathbf{P}\alpha, \mathbf{Do}\,\alpha, \mathbf{O}\alpha$ in a rule. For a rule $r_i$ as shown in Example 12.1.3, denote by $r_i^\star$ the new rule obtained by appending to $B(i)$ the precondition of any action status atom of the form $\mathbf{P}\alpha, \mathbf{Do}\,\alpha, \mathbf{O}\alpha$ (appropriately standardized apart) from the head or body of $r_i$. Thus, suppose $r$ is the rule*

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

*Suppose $pre(\alpha(X,Y)) = \mathbf{in}(\mathtt{Y}, \mathtt{d}_1{:}f_1(\mathtt{X}))$ and $pre(\beta) = \mathbf{in}(\mathtt{3}, \mathtt{d}_2{:}f_2())$. Then $r^\star$ is the rule*

$$\begin{aligned} \mathbf{Do}\,\alpha(X,Y) \leftarrow\; & \mathbf{in}(\mathtt{X}, \mathtt{d}{:}f(\mathtt{Y}))\,\&\,\mathbf{in}(\mathtt{Y}, \mathtt{d}_1{:}f_1(\mathtt{X}))\,\&\,\mathbf{in}(\mathtt{3}, \mathtt{d}_2{:}f_2())\,\& \\ & \mathbf{P}\beta\,\&\,\mathbf{F}\gamma(Y). \end{aligned}$$

*We now define* $\mathit{cft}_{pr}$ *as follows.*

$$\mathit{cft}_{pr}(r_i, r_j) = \begin{cases} \textbf{true} & \textit{if } \mathit{cft}_{bcc}(r_i^\star, r_j^\star) = \textbf{true} \\ \textbf{false} & \textit{otherwise.} \end{cases}$$

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

**Theorem 12.1.2**
*Suppose* $r$ *is a rule, and* $\alpha(\vec{X})$ *is an action such that some atom* $Op\,\alpha(\vec{t})$ *appears in* $r$*'s body where* $Op \in \{\mathbf{P}, \mathbf{O}, \mathbf{Do}\}$. *Then:*

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

**Proof:**

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

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

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

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

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

(2) Follows immediately from the strong safety of $\alpha$'s precondition, and part (1) above.

**Note 9** *Throughout the rest of this chapter, we will assume that an arbitrary, but fixed conflict-freedom test is used.*

### 12.1.3 Deontic Stratification

In this section, we define the concept of what it means for an agent program $\mathcal{P}$ to be *deontically stratified*—this definition extends the classical notion of stratification in logic programs introduced by (Apt, Blair, and Walker 1988). The first concept we define is that of a layering function.

**Definition 12.1.12 (Layering Function)**
*Let* $\mathcal{P}$ *be an agent program.* A layering function $\ell$ *is a function* $\ell : \mathcal{P} \to \mathcal{N}$.

A layering function assigns a nonnegative integer to each rule in the program, and in doing so, it groups rules into layers as defined below.

**Definition 12.1.13 (Layers of an Agent Program)**
*If $\mathcal{P}$ is an agent program, and $\ell$ is a layering function over $\mathcal{P}$, then the i-th layer of $\mathcal{P}$ w.r.t. $\ell$, denoted $\mathcal{P}_i^\ell$, is defined as:*

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

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

The following example presents some simple layering functions.

**Example 12.1.7 (Layering Functions)**
*Consider the agent program $\mathcal{P}$ given below.*

$r_1$: **Do** *execute_flight_plan*(`Flight_route`) $\leftarrow$
  **in**(`automated`, `autoPilot`:*pilotStatus*(`pilot_message`)),
  **Do** *create_flight_plan*(`No_go`, `Flight_route`, `Current_location`)

*If the plane is on autopilot and a flight plan has been created, then execute it.*

$r_2$: **O** *create_flight_plan*(`No_go`, `Flight_route`, `Current_location`) $\leftarrow$
  **O** *adjust_course*(`No_go`, `Flight_route`, `Current_location`)

*If our agent is required to adjust the plane's course, then it is also required to create a flight plan.*

$r_3$: **O** *maintain_course(no_go, flight_route, current_location)* $\leftarrow$
  **in**(`automated`, `autoPilot`:*pilotStatus*(`pilot_message`)),
  $\neg$ **O** *adjust_course(no_go, flight_route, current_location)*

*If the plane is on autopilot and our agent is not obliged to adjust the plane's course, then our agent must ensure that the plane maintains its current course.*

$r_4$: **O** *adjust_course(no_go, flight_route, current_location)* $\leftarrow$
  **O** *adjustAltitude*(`Altitude`)

*If our agent must adjust the plane's altitude, this it is obliged to also adjust the plane's flight route as well.*

*Note that for simplicity, these rules use constant valued parameters for maintain_course and adjust_course. A more realistic example may involve using* `autoPilot`:*calculateLocation*() *to determine the plane's next location (i.e., the value for current_location),* `autoPilot`:*calculateFlightRoute*() *to determine a new flight route w.r.t. this value for current_location (i.e., the value for flight_route), etc.*

*Let function $\ell_1$ assign 0 to rule $r_4$, 1 to rules $r_2, r_3$, and 2 to rule $r_1$. Then $\ell_1$ is a layering function which induces the program layers $\mathcal{P}_0^{\ell_1} = \{r_4\}$, $\mathcal{P}_1^{\ell_1} = \{r_2, r_3\}$, and $\mathcal{P}_2^{\ell_1} = \{r_1\}$. Likewise, the function $\ell_2$ which assigns 0 to rule $r_4$ and 1 to the remaining rules is also a layering function. In fact, the function $\ell_3$ which assigns 0 to all rules in $\mathcal{P}$ is also a layering function.*

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

**Definition 12.1.14 (Modality Ordering)**
*The partial ordering "$\leq$" on the set of deontic modalities $M = \{\mathbf{P}, \mathbf{O}, \mathbf{Do}, \mathbf{W}, \mathbf{F}\}$ is defined as follows (see Figure 12.1 on the following page): $\mathbf{O} \leq \mathbf{Do}$, $\mathbf{O} \leq \mathbf{P}$, $\mathbf{Do} \leq \mathbf{P}$, and $Op \leq Op$, for each*

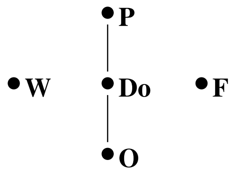



Figure 12.1: Modality ordering

*$Op \in M$. Furthermore, for ground action status atoms $A$ and $B$, we define that $A \leq B$ if, by definition, $A = Op\alpha$, $B = Op'\alpha$, and $Op' \leq Op$ all hold.*

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

**Definition 12.1.15 (Deontically Stratifiable Agent Program)**
*An agent program $\mathcal{P}$ is* deontically stratifiable *if, by definition, there exists a layering function $\ell$ such that:*

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

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

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

**Example 12.1.8 (Deontic Stratifiability)**
*Consider the agent program and layer functions given in Example 12.1.7 on the page before. Then the first condition of deontic stratifiability requires $\ell(r_2) \leq \ell(r_1)$ and $\ell(r_4) \leq \ell(r_2)$. Also, the second condition of deontic stratifiability requires $\ell(r_4) < \ell(r_3)$. Thus, $\ell_1$ and $\ell_2$ (but not $\ell_3$) are witnesses to the stratifiability of $\mathcal{P}$.*

*Note that some agent programs are not deontically stratifiable. For instance, let $\mathcal{P}'$ contain the following rule:*

$r_1'$**:** **Do** *compute_currentLocation*(report) $\leftarrow$
$\neg$ **Do** *compute_currentLocation (report)*

*Here, the author is trying to ensure that a plane's current location is always computed. The problem is that the second condition of deontic stratifiability requires $\ell(r_1') < \ell(r_1')$ which is not possible so $\mathcal{P}'$ is not deontically stratifiable. Note that if we replace $r_1'$ with "***Do** *compute_currentLocation (report) $\leftarrow$ ", then $\mathcal{P}'$ would be deontically stratifiable.*

It is worth noting that if $\mathcal{P}$ is deontically stratifiable, then condition (2) in the definition of a conflict free agent program (Definition 12.1.9 on page 389) is immediately true. Informally speaking, if we have a positive literal in the body of a rule, then the rule can only fire if that literal is derived—this means that heads conflict. Otherwise, if the literal is negative, then the rule must be in a lower layer than itself, which is impossible.

### 12.1.4 Weak Regular Agent Programs

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

**Definition 12.1.16 (Strongly Safe Action)**
*An action $\alpha(\vec{X})$ is said to be* strongly safe *w.r.t. FINTAB if its precondition is strongly safe modulo $\vec{X}$, and each code call from the add list and delete list is strongly safe modulo $\vec{Y}$ where $\vec{Y}$ includes all root variables in $\vec{X}$ as well as in the precondition of $\alpha$.*

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

We can now define a weak regular agent program.

**Definition 12.1.17 (Weak Regular Agent Program)**
*Let $\mathcal{P}$ be an agent program, FINTAB a finiteness table, and cft a conflict-freedom test. Then, $\mathcal{P}$ is called a* weak regular agent program *(WRAP for short) w.r.t. FINTAB and cft, if, by definition, the following three conditions all hold:*

**Strong Safety:** *All rules in $\mathcal{P}$ and actions $\alpha$ in the agent's action base are strongly safe w.r.t. FINTAB.*

**Conflict-Freedom:** *$\mathcal{P}$ is conflict free under cft.*

**Deontic Stratifiability:** *$\mathcal{P}$ is deontically stratifiable.*

The following example presents an example of a *WRAP*, as well as an agent program that is not a *WRAP*.

**Example 12.1.9 (Sample *WRAP*)**
*Let $\mathcal{P}$ be the agent program given in Example 12.1.7 on page 393 and suppose that all actions in $\mathcal{P}$ are strongly safe w.r.t. a finiteness table FINTAB. Consider the conflict freedom test $\mathsf{cft}_h$. Then $\mathcal{P}$ is a WRAP as it is conflict free under $\mathsf{cft}_h$ and as it is deontically stratified according to Example 12.1.8 on the facing page. Now, suppose we add the following rule to $\mathcal{P}$:*

$r_5$: **W** *create_flight_plan(no_go, flight_route, current_location)* ←
  **not_in**(`automated`, autoPilot:*pilotStatus*(`pilot_message`))

*This rule indicates that our agent is not obligated to adjust the plane's course if the plane is not on autopilot. Note that as $\mathsf{cft}_h(r_2, r_5)$ =* **false***, our new version of $\mathcal{P}$ is not conflict free and so $\mathcal{P}$ would no longer be a WRAP.*

### 12.1.5 Weakly Regular Agents

The framework in Chapter 6 specifies that in addition to an agent program, each agent has an associated set $\mathcal{IC}$ of integrity constraints, specifying conditions that an agent state must satisfy, and action constraints $\mathcal{AC}$, which describe conditions under which a certain collection of actions may not be concurrently executed. In order for an agent to evaluate what it must do in a given state, the ability to effectively or even polynomially evaluate the agent program is not enough—effective evaluation of the integrity and action constraints is also required.

**Definition 12.1.18 (Strongly Safe Integrity and Action Constraints)**
*An integrity constraint of the form* $\psi \Rightarrow \chi$ *is* strongly safe *if, by definition,* $\psi$ *is strongly safe and* $\chi$ *is strongly safe modulo the root variables in* $\psi$*. An action constraint* $\{\alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)\} \hookleftarrow \chi$ *is* strongly safe *if and only if* $\chi$ *is strongly safe.*

**Note 10** *We will generally assume that integrity constraints and action constraints do* not *refer to the* msgbox *package. This will become necessary to assume in Section 12.5.1, and does not restrict our framework very much from a practical point of view.*

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

**Example 12.1.10 (Integrity and Action Constraints)**
*Let* $\mathcal{IC}$ *be the following integrity constraint:*

$\mathbf{in}(\mathtt{X}_1, \mathtt{autoPilot}{:}\mathit{pilotStatus}(\mathtt{Pilot\_message}))$ &
$\mathbf{in}(\mathtt{X}_2, \mathtt{autoPilot}{:}\mathit{pilotStatus}(\mathtt{Pilot\_message})) \Rightarrow X_1 \neq X_2$

*This indicates each pilot message can denote at most one pilot status. Here,* $\mathcal{IC}$ *is strongly safe if FINTAB has a row of the form* $\langle \mathtt{autoPilot}{:}\mathit{pilotStatus}(\mathtt{a}_1), (\flat)\rangle$.
*Let* $\mathcal{AC}$ *be the following action constraint:*

$\{\mathit{adjust\_course}(\mathtt{No\_go}, \mathtt{FlightRoute}, \mathtt{CurrentLocation}),$
$\mathit{maintain\_course}(\mathtt{No\_go}, \mathtt{FlightRoute}, \mathtt{CurrentLocation}) \hookleftarrow$

*This indicates that the plane cannot adjust its course and maintain its course at the same time. Here, regardless of FINTAB,* $\mathcal{AC}$ *is strongly safe.*

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

**Definition 12.1.19 (Strongly Safe Notion of Concurrency)**
*A notion of concurrency,* **conc***, is said to be* strongly safe *if, by definition, for every set* $\mathcal{A}$ *of actions, if all members of* $\mathcal{A}$ *are strongly safe, then so is* $\mathbf{conc}(\mathcal{A})$.

The reader may easily verify that the three notions of concurrency proposed in Chapter 6 are all strongly safe.

**Definition 12.1.20 (Weakly Regular Agent)**
*An agent* $\mathtt{a}$ *is* weakly regular *if, by definition, its associated agent program is weakly regular and the action constraints, integrity constraints, and the notion of concurrency in the background are all strongly safe.*

## 12.2 Properties of Weakly Regular Agents

In this section, we will describe some theoretical properties of regular agents that will help us compute their reasonable status sets efficiently (i.e. in polynomial time data complexity).

This section is divided up into the following parts. First, we show that every deontically stratifiable agent program (and hence every *WRAP*) has a so-called "canonical layering". Then we will show that every *WRAP* has an associated fixpoint computation method—the fixpoint computed by this method is the only possible reasonable status set the *WRAP* may have.

### 12.2.1 Canonical Layering

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

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

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

The following example shows the canonical layering associated with the *WRAP* we have encountered earlier on in this chapter.

**Example 12.2.1 (Canonical Layering)**
*Consider the agent program and layer functions given in Example 12.1.7 on page 393. Recall that* $\ell_1 \in \mathsf{wtn}(\mathcal{P})$, $\ell_2 \in \mathsf{wtn}(\mathcal{P})$, *and* $\ell_3 \notin wtn(\mathcal{P})$. *Here, since* $\ell_1$ *requires three layers* $\ell_2$ *requires two layers, and* $\ell_3$ *requires one layer,* $\mathsf{can}^{\mathcal{P}} = \ell_2$.

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

**Proposition 12.2.1**
*Let* $\mathcal{P}$ *be an agent program which is deontically stratifiable. Then* $\mathsf{can}^{\mathcal{P}} \in \mathsf{wtn}(\mathcal{P})$, *i.e.* $\mathsf{can}^{\mathcal{P}}$ *is a witness to the deontic stratifiability of* $\mathcal{P}$.

**Proof:**

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

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

**Note 11** *Throughout this book, whenever we discuss a WRAP, unless stated otherwise we will use its canonical layering.*

### 12.2.2 Fixpoint Operators for *WRAPs*

In Chapter 6, we have shown how we may associate with any agent program $\mathcal{P}$ an operator $\mathbf{T}_{\mathcal{P},O_S}$ which maps a status set $S$ to another status set, and we have characterized the rational status of a positive agent program as the least fixpoint of this operator. We will use the powers of this operator in the characerization of the reasonable status set of a *WRAP*. We introduce the following definition.

**Definition 12.2.1 ($\mathbf{T}^i_{\mathcal{P},O_S}(S)$ and $\mathbf{T}^\omega_{\mathcal{P},O_S}(S)$ Operators)**
*Suppose $\mathcal{P}$ is an agent program, $O_S$ an agent state, and $S$ is a status set. Then, the operators $\mathbf{T}^i_{\mathcal{P},O_S}$, $i \geq 0$, and $\mathbf{T}^\omega_{\mathcal{P},O_S}$ are defined as follows:*

$$
\begin{aligned}
\mathbf{T}^0_{\mathcal{P},O_S}(S) &= S,\\
\mathbf{T}^{(i+1)}_{\mathcal{P},O_S}(S) &= \mathbf{T}_{\mathcal{P},O_S}(\mathbf{T}^i_{\mathcal{P},O_S}(S)),\\
\mathbf{T}^\omega_{\mathcal{P},O_S}(S) &= \bigcup_{i=0}^{\infty} \mathbf{T}^i_{\mathcal{P},O_S}(S).
\end{aligned}
$$

An example of the behavior of the $\mathbf{T}^{\cdot}_{\mathcal{P},O_S}$ operators is given below.

**Example 12.2.2 ($\mathbf{T}^{\cdot}_{\mathcal{P},O_S}$ Operator)**
*Let $\mathcal{P}$ contain rules $r_1, r_2, r_4$ from the agent program given in Example 12.1.7 on page 393, let $O_S$ indicate that the plane is on autopilot, and let $S = \{\mathbf{O}adjustAltitude(5000)\}$. Then*

$\mathbf{T}^0_{\mathcal{P},O_S}$ *(S)* = { **O** *adjustAltitude(5000)* },
$\mathbf{T}^1_{\mathcal{P},O_S}$ *(S)* = { **O** *adjust_course(no_go, flight_route, current_location),*
**Do** *adjustAltitude(5000),* **P** *adjustAltitude(5000)* } $\cup$ $\mathbf{T}^0_{\mathcal{P},O_S}$ *(S),*
$\mathbf{T}^2_{\mathcal{P},O_S}$ *(S)* = { **O** *create_flight_plan(no_go, flight_route, current_location),*
**Do** *adjust_course(no_go, flight_route, current_location),*
**P** *adjust_course(no_go, flight_route, current_location)* } $\cup$ $\mathbf{T}^1_{\mathcal{P},O_S}$ *(S),*
$\mathbf{T}^3_{\mathcal{P},O_S}$ *(S)* = { **Do** *create_flight_plan(no_go, flight_route, current_location),*
**P** *create_flight_plan(no_go, flight_route, current_location)* } $\cup$ $\mathbf{T}^2_{\mathcal{P},O_S}$ *(S),*
$\mathbf{T}^4_{\mathcal{P},O_S}$ *(S)* = { **Do** *execute_flight_plan(flight_route)* } $\cup$ $\mathbf{T}^3_{\mathcal{P},O_S}$ *(S),*
$\mathbf{T}^5_{\mathcal{P},O_S}$ *(S)* = { **P** *execute_flight_plan(flight_route)* } $\cup$ $\mathbf{T}^4_{\mathcal{P},O_S}$ *(S), and*
$\mathbf{T}^6_{\mathcal{P},O_S}$ *(S)* = $\mathbf{T}^5_{\mathcal{P},O_S}$ *(S),* so $\mathbf{T}^\omega_{\mathcal{P},O_S}(S) = \mathbf{T}^5_{\mathcal{P},O_S}(S)$.

*Note that by removing rule $r_3$ from $\mathcal{P}$, we turned $\mathcal{P}$ into a positive WRAP. To see why this was necessary, suppose $\mathcal{P}$ included $r_3$. Then both* **O** *maintain_course(no_go, flight_route, current_location) and* **O** *adjust_course(no_go, flight_route, current_location) would be members of $\mathbf{T}^1_{\mathcal{P},O_S}$ (S). This is not good since the plane cannot maintain and adjust its course at the same time. Later in this chapter, we shall introduce a fixpoint operator for general WRAPs which effectively solves this problem.*

We remark that the increase of the sequence $\mathbf{T}^i_{\mathcal{P},O_S}(S)$ in the previous example is not incidental. In fact, the operator $\mathbf{T}_{\mathcal{P},O_S}$ is *inflationary*, i.e., $S \subseteq \mathbf{T}_{\mathcal{P},O_S}(S)$ always holds. This important property will be exploited below.

### Positive *WRAPs*

Recalling from Chapter 6 that $lfp(\mathbf{T}_{\mathcal{P},O_S}) = \bigcup_{i=0}^{\infty} \mathbf{T}_{\mathcal{P},O_S}{}^i$ is the only candidate for being a rational status set of a positive agent program, using the identity $\mathbf{T}^i_{\mathcal{P},O_S}(\emptyset) = \mathbf{T}_{\mathcal{P},O_S}{}^i$ we may restate Theorem 6.5.1 on page 147 as follows.

**Proposition 12.2.2**
*Let $\mathcal{P}$ be a positive agent program. Then, a status set $S$ is a rational status set of $\mathcal{P}$ if and only if $S = \mathbf{T}^\omega_{\mathcal{P},O_S}(\emptyset)$ and $S$ is a feasible status set of $\mathcal{P}$.*

The preceding result guarantees that *positive* agent programs always have an iteratively computable least fixpoint. This fixpoint is a rational status set, and thus a reasonable status set, if $S$ satisfies deontic consistency as well as the action constraints and integrity constraints. If the program is weakly regular, then we obtain the following result—in the sequel, if $r$ is a rule, $O_S$ is an agent state, $S$ is a status set, and $\theta$ is a substitution, we define a special predicate $\mathrm{AR}(r,\theta,S)$ to be true *if, by definition,*:

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

That is, $\mathrm{AR}(r,\theta,S)$ is true just if the instance $r\theta$ of $r$ „fires" and adds $head(r\theta)$ to $\mathbf{App}_{\mathcal{P},O_S}(S)(S)$.

**Proposition 12.2.3**
*Let $\mathcal{P}$ be a* positive *agent program, and suppose that $\mathcal{P}$ is weakly regular. Then, $\mathcal{P}$ has at most one rational status set on $O_S$, and $S = \mathbf{T}^{\omega}_{\mathcal{P},O_S}(\emptyset)$ is the unique rational status set, if and only if $S$ satisfies the action and the integrity constraints.*

**Proof:** That $\mathcal{P}$ has at most one rational status set follows from Proposition 12.2.2 on the facing page. Since $S$ is deontically and action closed, it remains to verify that $S$ is deontically consistent. Suppose this is not the case. Then, at least one of the three conditions (D1)–(D3) of deontic consistency of $S$ is violated. For each case, we derive a contradiction, which proves the result:

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

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

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

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

Though positive agent programs may appear to be unnecessarily restrictive, they are in fact very useful to express many complex agent applications. For instance, the logistics application described in Chapter 13 is an example of a highly nontrivial positive agent program used for a real world application.

As for arbitrary agent programs, we observe that like for positive agent programs iterating the $\mathbf{T}_{\mathcal{P},O_S}$ operator on $\emptyset$ will eventually lead to a fixpoint $\mathbf{T}^{\omega}_{\mathcal{P},O_S}$ of the $\mathbf{T}_{\mathcal{P},O_S}$ operator; this is true even if we start from an arbitrary set $S$.

**Proposition 12.2.4**
*Suppose $\mathcal{P}$ is any agent program and $O_S$ is any agent state. Then, for every $S$, $\mathbf{T}^{\omega}_{\mathcal{P},O_S}(S)$ is a fixpoint of $\mathbf{T}_{\mathcal{P},O_S}$ and $\mathbf{T}^{\omega}_{\mathcal{P},O_S}(S)$ is action closed.*

**Proof:** Let $X = \mathbf{T}^{\omega}_{\mathcal{P},O_S}(S)$ $(=\bigcup_{i=0}^{\infty}\mathbf{T}^{i}_{\mathcal{P},O_S}(S))$. We have to show that $\mathbf{T}_{\mathcal{P},O_S}(X) = X$ holds. Since, as remarked above, $\mathbf{T}_{\mathcal{P},O_S}$ is inflationary, $X \subseteq \mathbf{T}_{\mathcal{P},O_S}(X)$ holds; it thus remains to show $\mathbf{T}_{\mathcal{P},O_S}(X) \subseteq X$, i.e., that each atom $A \in \mathbf{T}_{\mathcal{P},O_S}(X)$ is in $X$. The are two cases to consider.

(1) $A \in \mathbf{App}_{\mathcal{P}}(X)$. Then, a rule $r \in \mathcal{P}$ and a $\theta$ exist such that $head(r\theta) = A$ and $\mathrm{AR}(r,\theta,X)$ is true, i.e.,

   (a) $r\theta$ is ground;

   (b) $B_{cc}(r\theta)$ is true in $O_S$;

   (c) $B^{+}_{as}(r\theta) \subseteq X$ and

   (d) $\neg.B^{-}_{as}(r\theta) \cap X = \emptyset$;

   (e) For every atom $\mathsf{Op}\alpha \in B^{+}_{as}(r\theta) \cup H(r\theta)$ where $\mathsf{Op} \in \{\mathbf{P},\mathbf{Do},\mathbf{O}\}$, the action $\alpha$ is executable in $O_S$.

   Since $B^{+}_{as}(r\theta)$ is finite and $\mathbf{T}_{\mathcal{P},O_S}$ is inflationary, the third condition implies that $B^{+}_{as}(r\theta) \subseteq \mathbf{T}^{k}_{\mathcal{P},O_S}(S)$ holds for some $k \geq 0$. Furthermore, the fourth condition implies that $\neg.B^{-}_{as}(r\theta) \cap \mathbf{T}^{k}_{\mathcal{P},O_S}(S) = \emptyset$. This means that $\mathrm{AR}(r,\theta,\mathbf{T}^{k}_{\mathcal{P},O_S}(S))$ is true; hence, $A \in \mathbf{T}_{\mathcal{P},O_S}(\mathbf{T}^{k}_{\mathcal{P},O_S}(S)) \subseteq \mathbf{T}^{k+1}_{\mathcal{P},O_S}(S)$, which means $A \in X$.

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

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

### General *WRAPs*

We now extend the fixpoint operator in the preceding subsection to arbitrary *WRAPs*. We will define below an operator $\Gamma^{l}_{\mathcal{P},O_S}\uparrow\omega$ that evaluates (from bottom to top) the layers of a *WRAP* generated by a layering function $\ell$.

The operator $\Gamma^{l}_{\mathcal{P},O_S}\uparrow i$ evaluates the layer $i$ by computing the fixpoint $\mathbf{T}^{\omega}_{\mathcal{P}_i,O_S}$ for the program $\mathcal{P}_i$, starting from the result that has been computed at the previous layer $i-1$. The operator $\Gamma^{l}_{\mathcal{P},O_S}\uparrow\omega$ accumulates the computation of all layers. Formally, the definition is as follows.

**Definition 12.2.2 ($\Gamma^l_{\mathcal{P},O_S}\uparrow i$ and $\Gamma^l_{\mathcal{P},O_S}\uparrow\omega$ Operators)**
*Suppose $\mathcal{P}$ is a WRAP witnessed by layering function $\ell$, and suppose the layers of $\mathcal{P}$ induced by $\ell$ are $\mathcal{P}_0,\ldots,\mathcal{P}_k$. The operators $\Gamma^l_{\mathcal{P},O_S}\uparrow i(S)$ and $\Gamma^l_{\mathcal{P},O_S}\uparrow\omega(S)$ are defined as follows.*

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

We write $\Gamma^l_{\mathcal{P},O_S}\uparrow i$ and $\Gamma^l_{\mathcal{P},O_S}\uparrow\omega$ for $\Gamma^l_{\mathcal{P},O_S}\uparrow i(\emptyset)$ and $\Gamma^l_{\mathcal{P},O_S}\uparrow\omega(\emptyset)$, respectively. The following example illustrates the computation of $\Gamma^l_{\mathcal{P},O_S}\uparrow\omega$.

**Example 12.2.3 ($\Gamma^l_{\mathcal{P},O_S}\uparrow\omega$ Operator)**
*Let $O_S$ indicate that the plane is on autopilot and let $\mathcal{P}$ contain all rules for the agent program given in Example 12.1.7 on page 393. Additionally, let $\mathcal{P}$ contain the following rule:*

$r_0$**: O** *adjustAltitude(5000)* $\leftarrow$

*Then $\mathcal{P}$ is a WRAP which is witnessed by layering function $\ell$ where $\mathcal{P}^\ell_0=\{r_0,r_4\}$ and $\mathcal{P}^\ell_1=\{r_1,r_2,r_3\}$. Here,*

$\Gamma^l_{\mathcal{P},O_S}\uparrow 0=\{$ **O** *adjustAltitude(5000),*
**O** *adjust_course(no_go, flight_route, current_location),*
**Do** *adjustAltitude(5000),* **P** *adjustAltitude(5000),*
**Do** *adjust_course(no_go, flight_route, current_location),*
**P** *adjust_course(no_go, flight_route, current_location)* $\}$,
$\Gamma^l_{\mathcal{P},O_S}\uparrow 1=\{$ **O** *create_flight_plan(no_go, flight_route, current_location),*
**Do** *create_flight_plan(no_go, flight_route, current_location),*
**P** *create_flight_plan(no_go, flight_route, current_location),*
**Do** *execute_flight_plan(flight_route),*
**P** *execute_flight_plan(flight_route)* $\}\cup\Gamma^l_{\mathcal{P},O_S}\uparrow 0$, *and*
$\Gamma^l_{\mathcal{P},O_S}\uparrow 2=\Gamma^l_{\mathcal{P},O_S}\uparrow 1$, so $\Gamma^l_{\mathcal{P},O_S}\uparrow\omega=\Gamma^l_{\mathcal{P},O_S}\uparrow 1$.

*Note that although $r_3$ was included in $\mathcal{P}$, it never had a chance to fire as it had to be assigned to the second layer. Thus, we have some insight into why this fixpoint operator solves the problem mentioned in Example 12.2.2 on page 398.*

The theorem below tells us that for *all* layerings $\ell\in\mathsf{wtn}(\mathcal{P})$, $\Gamma^l_{\mathcal{P},O_S}\uparrow\omega$ is a reasonable status set of any WRAP that has no associated action constraints or integrity constraints. For the proof of this result, we use the following technical lemma, which says that in the computation of $\Gamma^l_{\mathcal{P},O_S}\uparrow\omega$, the applicability of rules is preserved.

Let us denote $S^\ell_{0,j}=\mathbf{T}^j_{\mathcal{P}_0,O_S}(\emptyset)$, for all $j\geq 0$, and $S^\ell_{i+1,j}=\mathbf{T}^j_{\mathcal{P}_{i+1},O_S}(\Gamma^l_{\mathcal{P},O_S}\uparrow i)$, for all $i,j\geq 0$, i.e., $S^\ell_{i,j}$ contains the result of computing $\Gamma^l_{\mathcal{P},O_S}\uparrow\omega$ after step $j$ in level $i$. We shall drop the superscript $\ell$ when it is clear from the context. Note that $S_{i,j}$ monotonically increases, i.e., $S_{i',j'}\subseteq S_{i,j}$ if $(i',j)<(i,j)$ under the standard lexicographic ordering.

**Lemma 12.2.1**
*Suppose $\mathcal{P}$ is a WRAP, and let $\ell\in\mathit{wtn}(\mathcal{P})$. If, for some rule $r\in\mathcal{P}_i$ and stage $S_{i,j}$, it is the case that $\mathrm{AR}(r,\theta,S_{i,j})$ is true, then $\mathrm{AR}(r,\theta,S_{i',j'})$ is true, for every stage $S_{i',j'}$ such that $(i,j)<(i',j')$, and $\mathrm{AR}(r,\theta,S)$ is true where $S=\Gamma^l_{\mathcal{P},O_S}\uparrow\omega$.*

**Proof:** Suppose that $\mathrm{AR}(r,\theta,S_{i,j})$ is true. Thus,

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

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

**Theorem 12.2.1**
*Suppose $\mathcal{P}$ is a WRAP. Let $\ell \in wtn(\mathcal{P})$ be any witness to the regularity of $\mathcal{P}$. If $\mathcal{IC}$ and $\mathcal{AC}$ are both empty, then $\Gamma^{l}_{\mathcal{P},\mathcal{O}_{\mathcal{S}}} \uparrow \omega$ is a reasonable status set of $\mathcal{P}$ w.r.t. $\mathcal{O}_{\mathcal{S}}$.*

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

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

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

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

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

(S2) Since $\mathcal{AC} = \emptyset$, $S$ is trivially action consistent. To see that $S$ is deontically consistent, assume it is not. Thus, it must violate some deontic consistency rule (D1)–(D3). As in the proof of Proposition 12.2.3 on page 399, it can be shown using Lemma 12.2.1 on the preceding page that each such violation raises a contradiction.

(S3) Clearly, $\mathbf{A\text{-}Cl}(S) = \bigcup_{i=0}^{\infty} \mathbf{A\text{-}Cl}(\Gamma^{l}_{\mathcal{P},O_S} \uparrow i)$ holds by definition of $S$ and the fact that $\mathbf{A\text{-}Cl}(X) = \bigcup_{X' \in \mathcal{C}} \mathbf{A\text{-}Cl}(X')$ holds for every collection $\mathcal{C}$ of subsets of an arbitrary status set $X$ such that $\bigcup_{X' \in \mathcal{C}} X' = X$. Applying Proposition 12.2.4 on page 400, it follows $\mathbf{A\text{-}Cl}(S) = S$.

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

At this stage, we have shown that $S$ is a feasible status set of $\mathcal{P}'$. To establish that $S$ is rational, suppose $S' \subset S$ is a status set satisfying conditions (S1)–(S3) for $\mathcal{P}'$. Let $S_{i,j}$ be the first stage in the construction of $S$ such that $S_{i,j} \setminus S' \neq \emptyset$. Let $A \in S_{i,j} \setminus S'$ be any atom. It follows that there exist a rule $r \in \mathcal{P}_i$ and a $\theta$ such that $A = head(r\theta)$ and $\mathrm{AR}(r,\theta,S_{i,j-1})$ is true. By Lemma 12.2.1 on page 401, also $\mathrm{AR}(r,\theta,S)$ is true; item (4) of the definition of $\mathrm{AR}(r,\theta,S)$ implies that the rule $r'$ obtained from $r\theta$ by removing $B^{-}_{as}(r\theta)$ belongs to $\mathcal{P}'$. Furthermore, the minimality of $S_{i,j}$ implies that $B^{+}_{as}(r')$ ($=B^{+}_{as}(r\theta)$) is contained in $S'$. Thus, $\mathrm{AR}(r',\theta,S')$ is true. This implies $A \in S'$, which is a contradiction. Thus, a feasible status set $S' \subset S$ of $\mathcal{P}$ does not exist. This proves the result. ∎

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

**Corollary 16**
*Let $\mathcal{P}$ be a WRAP and let $\ell \in wtn(\mathcal{P})$. Then, $S = \Gamma^{l}_{\mathcal{P},O_S} \uparrow \omega$ is a fixpoint of $\mathbf{T}_{\mathcal{P},O_S}$.*

**Proof:** Theorem 12.2.1 on the preceding page and Lemma 6.5.1 on page 147 imply that $S$ is a pre-fixpoint of $\mathbf{T}_{\mathcal{P},O_S}$, i.e., $\mathbf{T}_{\mathcal{P},O_S}(S) \subseteq S$ holds. Since, as pointed out above, $\mathbf{T}_{\mathcal{P},O_S}$ is inflationary, i.e., $S \subseteq \mathbf{T}_{\mathcal{P},O_S}(S)$ holds, the result follows. ∎

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

**Theorem 12.2.2**
*Suppose $\mathcal{P}$ is a WRAP, $\ell \in wtn(\mathcal{P})$ and $S$ is any reasonable status set of $\mathcal{P}$. Then, $S = \Gamma^{l}_{\mathcal{P},O_S} \uparrow \omega$.*

**Proof:** To prove this result, it is sufficient to show by induction on $i \geq 0$ that for every rule $r \in \mathcal{P}_i$ and ground substitution $\theta$ it is the case that

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

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

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

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

holds. We consider the two directions of this equivalence.

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

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

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

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

The following are immediate corollaries of the above result.

**Corollary 17**
*Suppose $\mathcal{P}$ is a WRAP, and suppose $\ell_1, \ell_2$ are in wtn$(\mathcal{P})$. Then $\Gamma^{\ell_1}_{\mathcal{P},O_S} \uparrow \omega = \Gamma^{\ell_2}_{\mathcal{P},O_S} \uparrow \omega$.*

**Corollary 18**
*Suppose $\mathcal{P}$ is a WRAP and let $\ell \in$ wtn$(\mathcal{P})$ be arbitrary. If $\Gamma^{\ell_1}_{\mathcal{P},O_S} \uparrow \omega$ satisfies the action and integrity constraints $\mathcal{AC}$ and $\mathcal{IC}$, respectively, then $\Gamma^{\ell_1}_{\mathcal{P},O_S} \uparrow \omega$ is the (unique) reasonable status of $\mathcal{P}$ on $O_S$. Otherwise, $\mathcal{P}$ has no reasonable status set on $O_S$.*

This last result will play a *fundamental role* in the design of algorithms to compute the reasonable status set of a *WRAP* (if one exists). All that is required is to iteratively compute $\Gamma^{\ell_1}_{\mathcal{P},O_S} \uparrow \omega$, and then to check if $\Gamma^{\ell_1}_{\mathcal{P},O_S} \uparrow \omega$ satisfies the integrity and action constraints associated with the current state of the agent.

## 12.3 Regular Agent Programs

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

**Definition 12.3.1 (Prerequisite-Free (pf) Constraint)**
*A prerequisite-free (pf) constraint is defined as follows:*

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

**Definition 12.3.2 (Prerequisite-Free Constraint Rule)**
*A prerequisite-free constraint rule is of the form*

$$A \;\leftarrow\; \mathsf{pfc}$$

*where $A$ is an action status atom and $\mathsf{pfc}$ is a pf-constraint.*

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

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

**Definition 12.3.3 (Operator $\mathbf{Coll}_{\mathcal{P}}$)**
*Let $\mathcal{P}$ be an agent program and $R$ be a set of pf-constraint rules which are standardized apart. Suppose $\mathsf{Op} \in \{\mathbf{P}, \mathbf{O}, \mathbf{Do}, \mathbf{W}, \mathbf{F}\}$ and let $\alpha$ be any action name. Then the* collect set, *$\mathrm{Coll}_{\mathcal{P}}(R, \mathsf{Op}, \alpha)$, is defined as the following set of pf-constraints:*

$$\begin{aligned}
\mathrm{Coll}_{\mathcal{P}}(R,\mathsf{Op},\alpha) = \Big\{\gamma \mid\ & \mathsf{Op}\alpha(\vec{X}) \leftarrow \gamma_0 \in R, \\
& \text{there exists a rule } r \in \mathcal{P} \text{ such that } head(r) = \mathsf{Op}'\alpha(\vec{t}), \mathsf{Op}' \leq \mathsf{Op}, \\
& B^+_{as}(r) = \{\mathsf{Op}_1\alpha_1(\vec{t}_1), \ldots, \mathsf{Op}_k\alpha_k(\vec{t}_k)\}, \\
& \mathsf{Op}_i\alpha_i(\vec{X}_i) \leftarrow \gamma_i \in R, i = 1, \ldots, k, \\
& \text{and } \gamma = \gamma_0 \vee \Big[(\vec{X} = \vec{t})\& B_{cc}(r) \& \textstyle\bigwedge_{i=1}^{k} \Big((\vec{X}_i = \vec{t}_i)\&\gamma_i\Big)\& B^-_{as}(r)\Big]\Big\}
\end{aligned}$$

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

What this operator does is the following. It takes a pf-constraint rule from $R$ which defines $\mathsf{Op}\alpha(\vec{X})$ through its body $\gamma_0$, and weakens this constraint $\gamma_0$ (i.e., increases the set of solutions) by taking the disjunction with an unfolded rule $r$ whose head either defines an instance of the action status atom $\mathsf{Op}\alpha(\vec{X})$, or of an action status atom $\mathsf{Op}'\alpha(\vec{X})$ which, by deontic and action closure rules, defines an instance of $\mathsf{Op}\alpha(\vec{X})$. The unfolding of rule $r$ is obtained by replacing each positive action status atom $\mathsf{Op}_i\alpha_i(\vec{t}_i)$ in $r'$s body with a the body $\gamma_i$ of a pf-constraint rule $\mathsf{pfc}_i$ from $R$ which defines $\mathsf{Op}_i\alpha(\vec{X}_i)$.

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

**Note 12** *We will assume that when no pf-constraint rule in R has $Op\alpha(\vec{X})$ in the head, then that R is augment via the insertion of the pf-constraint rule $Op\alpha(\vec{X}) \leftarrow$ **false**. The rest of our treatment is based on this assumption.*

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

The following simple example illustrates the use of the Coll operator.

**Example 12.3.1 (Coll$_{\mathcal{P}}$ Operator)**
*Let $\mathcal{P}$ be the agent program in Example 12.1.7 on page 393 and let*

$$R = \{\, \mathbf{O} \mathit{adjustAltitude}(5000) \leftarrow \mathbf{in}(\mathtt{Alt}, \mathtt{autoPilot}{:}\mathit{getAltitude}())\,\&\,(\mathtt{Alt} < 4000)\,\}.$$

*Then*

$$\begin{aligned} \mathit{Coll}_{\mathcal{P}}(R, \mathbf{O}, \mathit{adjust\_course}(\mathtt{X},\mathtt{Y},\mathtt{Z})) \quad = \{ \quad & (\mathtt{X} = \mathit{no\_go})\,\&\,(\mathtt{Y} = \mathit{flight\_route})\,\& \\ & (\mathtt{Z} = \mathit{current\_location})\,\&\,(\mathtt{Altitude} = 5000)\,\& \\ & \mathbf{in}(\mathtt{Alt}, \mathtt{autoPilot}{:}\mathit{getAltitude}())\,\&\,(\mathtt{Alt} < 4000)\}. \end{aligned}$$

*Note that this expression can be simplified by removing the unused $(\mathtt{Altitude} = 5000)$ conjunct.*

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

**Definition 12.3.4 (Operator Unfold$_{\mathcal{P}}$)**

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

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

The operator Unfold$_{\mathcal{P},Op,\alpha}$ may be iterated; its powers are (as usual) denoted by $\mathrm{Unfold}_{\mathcal{P}}{}^0(\mathsf{Op},\alpha,R) = R$, $\mathrm{Unfold}_{\mathcal{P}}{}^{i+1}(\mathsf{Op},\alpha,R) = \mathrm{Unfold}_{\mathcal{P}}(\mathsf{Op},\alpha,\mathrm{Unfold}_{\mathcal{P}}{}^{i}(\mathsf{Op},\alpha,R))$, $i \geq 0$, and similar with Unfold$_{\mathcal{P}}$. The following simple example illustrates the use of the Unfold operator.

**Example 12.3.2 (Unfold$_{\mathcal{P}}$ Operator)**
*Let $\mathcal{P}$ be the agent program in Example 12.2.3 on page 401 and let $R = \emptyset$. Then*

$\mathit{Unfold}_{\mathcal{P}}{}^0(R) = \emptyset$,
$\mathit{Unfold}_{\mathcal{P}}{}^1(R) = \{$ **O/Do/P** *maintain_course(no_go, flight_route, current_location)* $\leftarrow$
$\quad \mathbf{in}(\mathtt{automated}, \mathtt{autoPilot}{:}\mathit{pilotStatus}(\mathtt{pilot\_message}))$ *&*
$\quad \neg$ **O** *adjust_course(no_go, flight_route, current_location),*
$\quad \mathbf{O}/\mathbf{Do}/\mathbf{P}\mathit{adjustAltitude}(5000) \leftarrow \}$,
$\mathit{Unfold}_{\mathcal{P}}{}^2(R) = \{$ **O/Do/P** *adjust_course(no_go, flight_route, current_location)* $\leftarrow$
$\quad (\mathtt{Altitude} = 5000)\ \} \cup \mathit{Unfold}_{\mathcal{P}}{}^1(R)$,
$\mathit{Unfold}_{\mathcal{P}}{}^3(R) = \{$ **O/Do/P** *create_flight_plan*$(\mathtt{No\_go}, \mathtt{Flight\_route}, \mathtt{Current\_location}) \leftarrow$

$$(\texttt{No\_go} = \textit{no\_go})\,\&\,(\texttt{Flight\_route} = \textit{flight\_route})\,\&$$
$$(\texttt{Current\_location} = \textit{current\_location})\,\&$$
$$(\texttt{Altitude} = 5000)\ \} \cup \textit{Unfold}_{\mathcal{P}}{}^{2}(R),$$
$$\textit{Unfold}_{\mathcal{P}}{}^{4}(R) = \{\ \mathbf{Do/P}\ \textit{execute\_flight\_plan}(\texttt{Flight\_route}) \leftarrow$$
$$\mathbf{in}(\texttt{automated}, \texttt{autoPilot}{:}\textit{pilotStatus}(\texttt{pilot\_message}))\ \&$$
$$(\texttt{No\_go} = \textit{no\_go})\,\&\,(\texttt{Flight\_route} = \textit{flight\_route})\,\&$$
$$(\texttt{Current\_location} = \textit{current\_location})\,\&$$
$$(\texttt{Altitude} = 5000)\ \} \cup \textit{Unfold}_{\mathcal{P}}{}^{3}(R),$$

*and* $\textit{Unfold}_{\mathcal{P}}{}^{5}(R) = \textit{Unfold}_{\mathcal{P}}{}^{4}(R)$.

*Note that in this example,* $\mathbf{O}/\mathbf{Do}/\mathbf{P}\,\alpha(\vec{X}) \in \textit{Unfold}_{\mathcal{P}}{}^{i}(R)$ *(or* $\mathbf{Do}/\mathbf{P}\,\alpha(\vec{X}) \in \textit{Unfold}_{\mathcal{P}}{}^{i}(R)$*) indicates that* $\{\mathbf{O}\,\alpha(\vec{X}), \mathbf{Do}\,\alpha(\vec{X}), \mathbf{P}\,\alpha(\vec{X})\} \subseteq \textit{Unfold}_{\mathcal{P}}{}^{i}(R)$ *(*$\{\mathbf{Do}\,\alpha(\vec{X}), \mathbf{P}\,\alpha(\vec{X})\} \subseteq \textit{Unfold}_{\mathcal{P}}{}^{i}(R)$*).*

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

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

**Definition 12.3.5 (Bistructure)**
*A* bistructure *for an agent program* $\mathcal{P}$ *is a pair* $(O_{\mathcal{S}}, S)$ *where* $O_{\mathcal{S}}$ *is a possible state of the agent in question, and* $S$ *is a status set.*

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

**Definition 12.3.6 (Satisfaction of a Ground pf-constraint by a Bistructure)**
*A bistructure* $(O_{\mathcal{S}}, S)$ satisfies

1. *a ground code call condition* $\chi$ *if, by definition,* $\chi$ *is true in* $O_{\mathcal{S}}$*;*
2. *a ground action status atom* $\neg Op\alpha$ *if, by definition,* $Op\alpha \notin S$*;*
3. *a conjunction* $pfc_1\,\&\,pfc_2$ *if, by definition, it satisfies* $pfc_1$ *and* $pfc_2$*;*
4. *a disjunction* $pfc_1 \vee pfc_2$ *if, by definition, it satisfies either* $pfc_1$ *or* $pfc_2$*.*

**Definition 12.3.7 (Solutions of a pf-constraint w.r.t. a Bistructure)**
*Suppose* $pfc$ *is a pf-constraint involving free variables* $\vec{X}$*. The* solutions *of* $pfc$ *w.r.t. a bistructure* $(O_{\mathcal{S}}, S)$ *is the set of all ground substitutions* $\theta$ *such that* $(O_{\mathcal{S}}, S)$ *satisfies* $pfc\theta$*.*

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

**Definition 12.3.8 ($a$-Equivalent pf-constraints)**
*Suppose* $a$ *is an agent, and* $pfc_1, pfc_2$ *are pf-constraints involving variables* $\vec{X}, \vec{Y}$ *respectively. Let* $\vec{X}', \vec{Y}'$ *be subvectors of* $\vec{X}, \vec{Y}$ *respectively of the same length.* $pfc_1, pfc_2$ *are said to be* $(a, \vec{X}', \vec{Y}')$-equivalent, *denoted* $pfc_1 \sim_{a,\vec{X}',\vec{Y}'} pfc_2$ *if, by definition, for every bistructure* $(O_{\mathcal{S}}, S)$ *such that* $S$ *is a reasonable status set of* $a$*'s agent program w.r.t. state* $O_{\mathcal{S}}$*, it is the case that* $\pi_{\vec{X}'}(\mathsf{Sol}(pfc_1)) = \pi_{\vec{Y}'}(\mathsf{Sol}(pfc_2))$ *where* $\mathsf{Sol}(pfc_i)$ *denotes the set of all solutions of* $pfc_i$ *and* $\pi_{\vec{Z}}(\mathsf{Sol}(pfc_i))$ *denotes the set of projections of solutions of* $pfc_i$ *on the variables in* $\vec{Z}$*.*

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

In general, the problem of checking equivalence of two PFCs is easily seen to be undecidable, and as a consequence, we introduce the notion of a pf-constraint equivalence test below which provides a sufficient condition for two pfc's to be equivalent.

**Definition 12.3.9 (pf-constraint Equivalence Test)**
*A* pf-constraint equivalence check test $eqi_{\alpha,\vec{X}',\vec{Y}'}$ *is a function that takes as input two pf-constraints* $pfc_1, pfc_2$*, such that if* $eqi_{\alpha,\vec{X}',\vec{Y}'}(pfc_1, pfc_2) = \mathbf{true}$ *then* $pfc_1, pfc_2$ *are equivalent w.r.t.* $\mathcal{P}$*.*

We will often write eqi instead of $\text{eqi}_{\alpha,\vec{X}',\vec{Y}'}$ when the parameters $\alpha, \vec{X}', \vec{Y}'$ are clear from context. Note that just as in the case of conflict freedom tests, a pf-constraint equivalence test merely implements a *sufficient* condition to guarantee equivalence of two pf-constraint rules. It may well be the case that $\mathsf{pfc}_1, \mathsf{pfc}_2$ are in fact equivalent on all agent states, but $\text{eqi}(\mathsf{pfc}_1, \mathsf{pfc}_2) = \text{false}$.

Some examples of constraint equivalence tests are given below.

**Example 12.3.3 (Renaming Permutation Equivalence)**
*The function* $eqi^{rp}$ *returns* **true** *on two pf-constraints* $pfc_1, pfc_2$ *whose variables are standardized apart if, by definition, there is a renaming substitution* $\theta$ *such that* $\{C\theta \mid C \in pfc_1^\star\} = \{C'\theta \mid C' \in pfc_2^\star\}$ *where* $pfc_i^\star$ *is a conjunctive normal form representation of* $pfc_i$*.*

**Example 12.3.4 (Rewrite-Based Equivalence)**
*Another way to check equivalence of two pf-constraints is to expect the agent developer to write a set, RW, of rewrite rules of the form*

$$condition \quad \rightarrow \quad pfc_1 = pfc_2$$

*where condition is a code call condition not involving the* $\mathbf{in}(,)$ *predicate, i.e. it only involves comparison operations* $=, <, \leq, >, \geq$*. RW encodes domain knowledge about what equivalences hold in the data structures and actions involved. It may be viewed as an equational theory (Plaisted 1993). Let* $\Upsilon^k(pfc)$ *denote the set of all PFC's that* $pfc$ *can be rewritten to by applying at most* $k$ *rules in RW.*

*We say that* $pfc_1, pfc_2$ *are* $k$*-equivalent w.r.t. RW if and only if* $\Upsilon^k(pfc_1) \cap \Upsilon^k(pfc_2) \neq \emptyset$*.*

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

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

**Definition 12.3.10 (Equivalence of Two Sets of pf-constraint Rules)**
*Two sets* $R_1, R_2$ *of pf-constraint rules are equivalent w.r.t. a pf-constraint equivalence test eqi, denoted* $R_1 \equiv_{eqi} R_2$*, if, by definition, there is a bijection* $\psi : R_1 \rightarrow R_2$ *such that for all* $r_1 \in R_1$*,* $eqi(r_1, \psi(r_1)) = \mathbf{true}$ *and* $r_1, r_2$ *both have heads of the form* $Op\alpha(\cdots)$*, i.e. their heads involve the same action name and the same deontic modality.*

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

**Definition 12.3.11 (*b*-bounded Agent Program)**
*An* agent program $\mathcal{P}$ *is* bounded *w.r.t. an equivalence check test eqi if, by definition, there is an integer b such that* $eqi(Unfold_{\mathcal{P}}{}^b(R), Unfold_{\mathcal{P}}{}^{b+1}(R)) = \mathbf{true}$, *for any set of pf-constraints R. In this case,* $\mathcal{P}$ *is* $(eqi, b)$*-bounded.*

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

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

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

$$\begin{aligned} R_0^\ell &= \emptyset, \\ R_{i+1}^\ell &= \text{Unfold}_{\mathcal{P}_i}{}^b(R_i^\ell), \text{ for all } i \geq 0, \end{aligned}$$

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

**Definition 12.3.12 (*b*-regular Agent Program)**
*Suppose a layering* $\ell$ *and equivalence tests* $eqi^{(i)}$ $(i \geq 0)$ *have been fixed for an agent program* $\mathcal{P}$*. Then,* $\mathcal{P}$ *is said to be a b-*regular agent program *w.r.t.* $\ell$ *and the* $eqi^{(i)}$*, if, by definition,* $\mathcal{P}$ *is a WRAP,* $\ell \in \mathsf{wtn}(\mathcal{P})$*, and each layer* $\mathcal{P}_i$ *of* $\mathcal{P}$ *is* $(eqi^{(i)}, b)$*-bounded.*

**Definition 12.3.13 (Regular Agent)**
*An agent is said to be* regular *w.r.t. a layering* $\ell$ *and a selection of pf-constraint equivalence tests* $eqi^{(i)}$*, if it is weakly regular and its associated agent program is b-regular w.r.t.* $\ell$ *and the* $eqi^{(i)}$*, for some* $b \geq 0$.

In the above definition, an agent's regularity depends on several parameters $\ell$, $\text{eqi}^{(i)}$, and $b$. *IADE* generates a layering of an agent program $\mathcal{P}$, and equivalence tests $\text{eqi}^{(i)}$ are fixed for each layer $\mathcal{P}_i$ with the help of the agent developer. *IADE* then sets $b$ to a default value, and iteratively constructs the sequence $R_0^\ell, R_1^\ell, \dots, R_{k+1}^\ell$; if in some step, the equivalence test

$$\text{eqi}^{(i)}(\text{Unfold}_{\mathcal{P}}{}^b(R), \text{Unfold}_{\mathcal{P}}{}^{b+1}(R))$$

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

## 12.4 Compile-Time Algorithms

In this section, we develop algorithms used in the compilation phase—that is, when the agent developer has built the agent and is either testing it, or is about to deploy it. This phase has two major components—checking if an agent is weakly regular, and computing an "initial" reasonable status set of the agent.

### 12.4.1 Checking Weak Regularity

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

The conflict-freedom conditions can be readily checked, as they do not depend on a layering $\ell$. The function $\mathsf{cft}(r_i, r_j)$ is used to check the first conflict freedom condition, while adapted efficient unification algorithms, e.g., (Paterson and Wegman 1978), may be used to check the second condition. However, the check for deontic stratification conditions is more complex. Different methods can be applied, and we outline here a method which is based on computing the (maximal) strongly connected components of a graph $G = (V, E)$. This method extends similar methods for finding stratifications of logic programs, cf. (Ullman 1989).

A *strongly connected component* (*SCC*) of a directed graph $G = (V, E)$ is a maximal set $C \subseteq V$ of vertices (maximal w.r.t. set inclusion) such that between every pair of vertices $v, v' \in C$, there exists a path from $v$ to $v'$ in $G$ involving only vertices from $C$. For any graph $G$, we can define its *supergraph* $S(G) = (V^*, E^*)$ as the graph whose vertices are the strongly connected components of $G$, and such that there is an edge $C \to C'$ in $E^*$, if there is an edge from some vertex $v \in C$ to some vertex $v' \in C'$ in the graph $G$. Note that the supergraph $S(G)$ is acyclic. Using Tarjan's algorithm, see e.g., (Moret and Shapiro 1991), the SCCs of $G$, and thus the supergraph $S(G)$, is computable in time $(O(|V| + |E|))$, i.e., in linear time from $G$.

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

**Example 12.4.1 (Layering Through Graphs)**
*Let $\mathcal{P}$ be the agent program given in Example 12.2.3 on page 401. Then the first condition of deontic stratifiability requires $\ell(r_0) \leq \ell(r_4) \leq \ell(r_2) \leq \ell(r_1)$. Also, the second condition of deontic stratifiability requires $\ell(r_4) < \ell(r_3)$. Thus, we obtain the following graph $G$:*

$$\begin{array}{ccccccc} r_1 & \longleftrightarrow & r_2 & \longleftrightarrow & r_4 & \longleftrightarrow & r_0 \\ & & & & \uparrow & & \\ & & & & r_3 & & \end{array}$$

*In other words, $E = \{(r_1, r_2), (r_2, r_1), (r_2, r_4), (r_4, r_2), (r_3, r_4), (r_4, r_0), (r_0, r_4)\}$ and $V = \{r_0, r_1, r_2, r_3, r_4\}$. For supergraph $S(G)$, $E^* = \{(v_2^*, v_1^*)\}$ and $V^* = \{v_1^*, v_2^*\}$ where $v_1^* = \{r_0, r_1, r_2, r_4\}$ and $v_2^* = \{r_3\}$. Here, since $r_3$ and $r_4$ are not in the same SCC, $\mathcal{P}$ is deontically stratified. Furthermore, our variant of "reverse" topological sorting on $S(G)$ reveals that $\ell(r_0) = \ell(r_1) = \ell(r_2) = \ell(r_4) = 0$ and $\ell(r_3) = 1$.*

The algorithm, **Check_WRAP**, used to check whether an agent program $\mathcal{P}$ is weakly regular is shown below.

**Algorithm 12.4.1**
***Check_WRAP($\mathcal{P}$)***

*($\star$ input is an agent program $\mathcal{P}$, a conflict-freedom test cft, and a finiteness table FINTAB $\star$)*
*($\star$ output is a layering $\ell \in wtn(\mathcal{P})$, if $\mathcal{P}$ is regular and "no" otherwise $\star$)*

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

The following example shows how the above algorithm works on an example agent program.

**Example 12.4.2 (Algorithm Check_WRAP)**
*Let $\mathcal{P}$ be the agent program given in Example 12.2.3 on page 401. Then* ***Check_WRAP($\mathcal{P}$)*** *begins by ensuring that every rule and action in $\mathcal{P}$ is strongly safe w.r.t. our FINTAB. It also ensures that there are no conflicts between or within $\mathcal{P}$'s rules. If everything is ok, it builds the graph $G$ and supergraph $S(G)$ given in Example 12.4.1 on the preceding page and ensures that $\mathcal{P}$ is deontically stratifiable. If it is not, then $\mathcal{P}$ cannot be a WRAP so an error message is returned and the algorithm halts. Otherwise, since $v_1^*$ has no outgoing edges, each rule $r \in v_1^*$ is assigned to layer $i = 0$, we increment $i$, and we remove $v_1^*$ and $(v_2^*, v_1^*)$ from $S(G)$. Since $V^* = \{v_2^*\} \neq \emptyset$, we continue the loop. Now we assign rule $r_3 \in v_2^*$ to layer $i = 1$, increment $i$, and remove $v_2^*$ from $S(G)$. Finally, $V^* = \emptyset$ so we return our layering $\ell$ as shown in Example 12.4.1 on the facing page.*

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

**Theorem 12.4.1**
*For any agent program $\mathcal{P}$,* ***Check_WRAP($\mathcal{P}$)*** *returns w.r.t. a conflict-freedom test cft and a finiteness table FINTAB, a layering $\ell \in wtn(\mathcal{P})$ if $\mathcal{P}$ is a WRAP, and returns "no" if $\mathcal{P}$ is not regular.*

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

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

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

Thus, extending the assumption for Step 1 by further assuming that atoms and rule bodies have size bounded by a constant—an assumption that is certainly plausible, since the number of literals in a rule body is not expected to exceed 20, say, and each literal will, as a string, hardly occupy more than 1024 characters—we obtain that **Check_WRAP**($\mathcal{P}$) can be executed in $O(|\mathcal{P}|^2 t_{\mathsf{cft}}(\mathcal{P}))$ time. This bound further decreases to $O(|\mathcal{P}| t_{\mathsf{cft}}(\mathcal{P}))$ time if for each action $\alpha$ and modality $\mathsf{Op}$, only a few rules (bounded by a constant) with head $\mathsf{Op}\,\alpha(\cdots)$ exist in $\mathcal{P}$. These assumptions on the "shape" of the rules and the program seem to be reasonable with respect to agent programs in practice. Thus, we may expect that **Check_WRAP** runs in $O(|\mathcal{P}| \cdot t_{\mathsf{cft}}(\mathcal{P}))$ time. In particular, for an efficient implementation of a $\mathsf{cft}$ as in Examples 12.1.3 on page 391– 12.1.5 on page 391, it runs in $O(|\mathcal{P}|)$ time, i.e., in linear time in the number of rules.

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

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

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

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

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

### 12.4.2 Computing Reasonable Status Sets

As we have already remarked previously in this chapter, computing the unique reasonable status set (if one exists) of a regular agent program can be done by first computing the status set $\Gamma^{\ell_1}_{\mathcal{P},O_S} \uparrow \omega$ and then checking if this status set satisfies the integrity and action constraints.

**Algorithm 12.4.2**
***Reasonable-SS***$(\mathcal{P}, \ell, \mathcal{IC}, \mathcal{AC}, O_S)$

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

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

Even though Algorithm **Reasonable_SS** can be executed on weakly regular agent programs, rather than *RAPs*, there is no guarantee of termination in that case. The following theorem states the result that for a regular agent, its reasonable status set on an agent state is effectively computable.

**Theorem 12.4.2**
*If $a$ is a regular agent, then algorithm **Reasonable_SS** computes a reasonable status set (if one exists) in finite time.*

**Proof:** We have to show that each of the steps – 5 of the algorithm can be done in finite time. As for step 1, the boundedness of the agent program $\mathcal{P}$ associated with $a$ ensures that the set $\Gamma^{l}_{\mathcal{P},O_S} \uparrow \omega$ is computable within a bounded number of steps: For computing $\Gamma^{l}_{\mathcal{P},O_S} \uparrow i$, we must compute the operator $\mathbf{T}^{\omega}_{\mathcal{P}_i,O_S}$ from (Def. 12.2.1 on page 397) associated with the layer $\mathcal{P}_i$, which needs to apply the operator $\mathbf{T}_{\mathcal{P}_i,O_S}$ only a bounded number of times. Furthermore, the number of nonempty layers $\mathcal{P}_i$ is bounded as well. An inductive argument shows that in each step $S_0, S_1, S_2, \ldots S_m = \Gamma^{l}_{\mathcal{P},O_S} \uparrow \omega$ of the fixpoint computation, any rule $r$ from the layer $\mathcal{P}_i$ currently considered instantiates due to strong safety only to a finite number of ground rules $r'$ which fire. These $r'$ can be effectively computed from the status set $S_k$ derived so far (where $S_0 = \emptyset$), proceeding as follows. Let $\Theta$ be the set of ground substitutions such that $B^+_{as}(r)$ is true w.r.t. $S_k$. Since, by induction hypothesis, $S_k$ is finite, $\Theta$ can be computed in finite time. Next, for each $\theta \in \Theta$, the set of all ground substitutions $\gamma$ that satisfy $B_{cc}(r\theta)$ is finite and is effectively computable. For each such $\gamma$, it is effectively checkable whether for the ground instance $r' = r\theta\gamma$ of $r$, the part $B^-_{as}(r')$ is true w.r.t. $S_k$ (i.e., $\neg.B^-_{as}(r') \cap S_k = \emptyset$), and whether for each atom $\mathsf{Op}\alpha$ from $B^+_{as}(r') \cup \{head(r')\}$ such that $\mathsf{Op} \in \{\mathbf{O}, \mathbf{Do}, \mathbf{P}\}$ it holds that $\alpha$ is executable in $\mathcal{O}_\mathcal{S}$. The instances of $r$ which fire are precisely all rules of this form. Since this yields only finitely many new action status atoms, also $S_{k+1}$ is finite and effective computable. It follows that $\Gamma^l_{\mathcal{P},\mathcal{O}_\mathcal{S}} \uparrow \omega$ is computed within finite time.

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

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

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

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

**Theorem 12.4.3**
*Suppose $\mathtt{a}$ is a fixed regular agent. Assume that the following holds:*

(1) *Every ground code call $\mathcal{S}:f(\mathtt{d_1}, \ldots, \mathtt{d_n})$, has a polynomial set of solutions, which is computed in polynomial time; and*

(2) *no occurrence of a variable in $\mathtt{a}$'s description loose.*

*Furthermore, assume that assembling and executing $\mathbf{conc}(\mathbf{Do}(S), \mathcal{O}_\mathcal{S})$ is possible in polynomial time in the size of $\mathbf{Do}(S)$ and $\mathcal{O}_\mathcal{S}$. Then, algorithm **Reasonable_SS** computes a reasonable status set (if one exists) on a given agent state $\mathcal{O}_\mathcal{S}$ in polynomial time (in the size of $\mathcal{O}_\mathcal{S}$).*

**Proof:** We have to argue that each of the steps 1–5 can be done in polynomial time, rather than in arbitrary finite time. This can be accomplished by refining the analysis in the proof of Theorem 12.4.2 on the preceding page.

As for Step 1, the cardinality of the set $\Theta$ of substitutions such that $B^+_{as}(r)$ is true w.r.t. the already derived status set $S_k$ is polynomial in the the size of $S_k$. Under the assumptions, this set is computable in polynomial time. Next, for each $\theta \in \Theta$, the assumptions imply that the set $\Gamma_\theta$ contains only polynomially many assignments $\gamma$, each of which is computable in polynomial time. The check whether $B^-_{as}(r')$ where $r' = (r\theta\gamma)$ is true w.r.t. $S_k$ is easy, and the test whether $r'$ is actually fired is feasible in time polynomial in the size of $\mathcal{O}_S$. Overall, the number of instances $r'$ of $r$ which eventually fire is polynomial in the size $S_k$ and the agent state.

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

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

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

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

## 12.5 The Query Maintenance Package

In this section, we will describe how RAPs are implemented within the *IMPACT* architecture via a specialized package called the *Query Maintenance Package.* The basic idea behind this package is simple. Agents need to *continuously* recompute their reasonable status sets, based on the latest set of state changes that have occurred (which in turn are triggered by messages received). We would like to reduce this run-time computation load on the agent as much as possible. We do this by ensuring that when the agent is deployed, a certain data structure called the **QMPtab** defined in Section 12.5 is populated. The **QMPtab** contains a succinct, non ground description of the agent's reasonable status set at *any given point in time*, i.e., it is state independent. With every operator $\mathsf{Op} \in \{\mathbf{O}, \mathbf{P}, \mathbf{Do}, \mathbf{F}, \mathbf{W}\}$, and every action $\alpha(\vec{X})$ with all nonground parameters, it associates a single query, which when evaluated against the current agent state specifies which instances of $\mathsf{Op}\,\alpha(\vec{X})$ are true in the current reasonable status set of the agent. Then, in Section 12.5.1, we describe a set of functions that may be used, both at run-time and compile time, to perform computations based on the **QMPtab**. These operators perform the basic computations needed by any agent.

**The QMPtab Data Structure**

The **QMPtab** is a table having the schema `(Opr,Action,PFC)` where:

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

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

**Example 12.5.1**
*A small example **QMPtab** is shown below.*

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

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

In general, given any **QMPtab**, we may associate with it a unique status set **SS**(**QMPtab**) as follows:

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

Here, truth of the pf-constraint $r.\mathtt{PFC}$ is with respect to the current agent state. As we can see, QMP may be used to succinctly represent status sets. What we would *like* is to ensure that at any given point in time **SS**(**QMPtab**) is in fact the unique reasonable status set of the RAP it is associated with w.r.t. the agent's state at that point in time.

The basic idea is to first compute such a **QMPtab** when the agent is deployed. As changes occur, we would like to use the **QMPtab** to rapidly compute the set of all **Do**-actions that the agent must perform, given the occurrence of the changes. This will be done via the `qmp:update` function described in Section 12.5.1.

### 12.5.1 QMP Function Calls

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

- `qmp:init`(action base $\mathcal{AB}$; software code $\mathcal{S}$)
  This function initializes the **QMPtab** so that **SS**(**QMPtab**) coincides with the unique reasonable status set of the agent program. This is done by first directly computing $\text{Unfold}_{\mathcal{P}}{}^{b}$. If $\mathsf{Op}\,\alpha \leftarrow \chi$ is in $\text{Unfold}_{\mathcal{P}}{}^{b}$, then the triple $(\mathsf{Op}, \alpha, \chi)$ is placed in **QMPtab**. The implementation of `qmp:init` is described in Section 12.5.1.

- `qmp:eval`(action status atom $\mathsf{Op}\,\alpha(\vec{t})$; agent state $O_{\mathcal{S}}$; software code $\mathcal{S}$)
  Given an action status atom $\mathsf{Op}\,\alpha(\vec{t})$, may associate with it a pfc-constraint $\gamma$ by finding the pfc-constraint $\gamma'$ associated with $\mathsf{Op}\,\alpha(\vec{X})$ in **QMPtab** and setting $\gamma = \gamma' \,\&\, \vec{X} = \vec{t}$. We then need to evaluate $\gamma$ over the current agent state and output the set of all solutions to $\gamma$ on $O_{\mathcal{S}}$. The `qmp:eval` function does this, building on top of the existing implementation of the HERMES Heterogeneous Reasoning and Mediator System developed at the University of Maryland (Adali et al. 1996; Brink et al. 1995; Lu et al. 1996).

- `qmp:update`(action rule $r$; software code $\mathcal{S}$)
  At any given point $t$ in time, the agent has a reasonable status set. When the state of the agent changes through the receipt of one or more messages, the agent must determine what new actions must be performed, i.e. it must compute the set of all actions of the form $\mathbf{Do}\,\alpha(\vec{t})$ that are true in the new reasonable status set of the agent (w.r.t. the state of the agent after receipt of the messages). The `qmp:update` algorithm tries to avoid a brute-force computation of this set.

We will now describe the three functions from above in more detail.

### The `qmp:init` Function

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

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

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

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

Figure 12.2 on the next page shows the AND/OR tree associated with the body of this pf-constraint rule. Note that this AND/OR tree is nothing more than a parse tree associated with a pf-constraint. When unfolding is performed, and $\mathsf{pfc}_i$ is updated to a new AND/OR parse tree $\mathsf{pfc}_{i+1}$ the AND/OR parse tree associated with $\mathsf{pfc}_i$ can be straightforwardly extended to one associated with $\mathsf{pfc}_{i+1}$.

It is important to note that the AND/OR tree associated with a pf-constraint may often contain repeated code calls and/or code calls whose answers are subsets of other code calls. In such cases, there are alternative representations (e.g., a directed acyclic graph) and various ways to optimize such trees by modifying standard query optimization methods in databases (Ullman 1989). We will not go into this issue in greater detail.

### The `qmp:eval` Function

Consider any pair $(\mathsf{Op}, \alpha(\vec{X}))$. After unfolding the RAP out completely using the `qmp:init` function, this pair has an associated pf-constraint in **QMPtab** which is represented as an AND/OR tree.

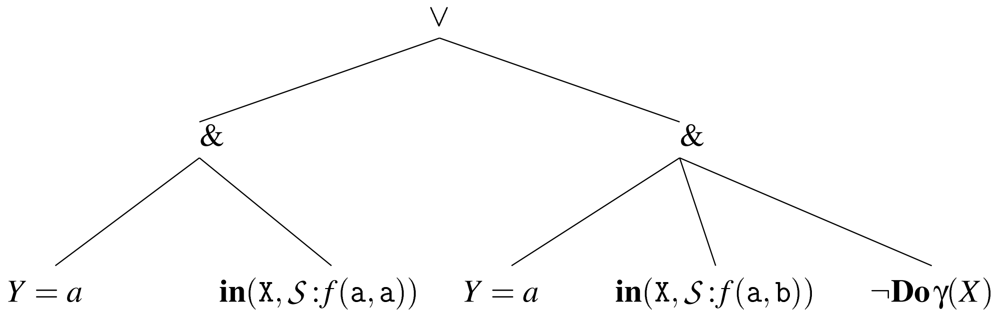



Figure 12.2: An example AND/OR tree associated with a pf-constraint

The `qmp:eval` function evaluates these pf-constraints bottom up (i.e. starting from leaf nodes). It associates a SOL field with each non-leaf node in the AND/OR tree. This field is intended to find the set of all solutions of the pf-constraint rooted at that node, and is initially set to be empty. The following algorithm specifies how to associate a SOL field with a non-leaf node and compute the final answer.

1. **Associating SOL Fields With Interior Nodes.**

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

   (b) *The node is an "∨" Node.* Recursively evaluate all its children. If $C_1, \ldots, C_n$ are its children, and these children have $SOL_1, \ldots, SOL_n$ as their solutions, respectively, let $SOL = \bigcup_{i=1}^{n} SOL_i$.

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

Notice that for the above procedure to work, for each ground negated action status atom $\neg\mathsf{Op}'\beta(\vec{Y}\theta)$, we must already know the set of solutions of $\mathsf{Op}'\beta(\vec{Y})$. This means that we must always invoke the `qmp:eval` function bottom up starting with action status atoms defined in the bottom most layer of the RAP, and work our way up.

Let us illustrate this procedure on the example AND/OR tree shown in Figure 12.2. We assume of course, that $\neg\mathbf{Do}\,\gamma(X)$ has been evaluated prior to the evaluation of this parse tree. The evaluation of the left "&" node is straightforward. Let us suppose this returns two substitutions, $\theta_1 = \{X/a\}$ and $\theta_2 = \{X/b\}$. Now evaluate the right "&" expression. This is done by first executing the code call condition $Y = a\,\&\,\mathbf{in}(\mathtt{X}, \mathcal{S}{:}f(\mathtt{a},\mathtt{b}))$. Let us assume this returns two substitutions, $\sigma_1 = \{X/a, Y/a\}$ and $\sigma_2 = \{X/b, Y/a\}$. For each of these two substitutions, we invoke $\mathbf{Do}\,\gamma(X)$. Suppose $\mathbf{Do}\,\gamma(a)$

fails, but $\mathbf{Do}\,\gamma(b)$ succeeds. Then SOL of the right "&" node consists just of $\sigma_1$. SOL of the root is now $\{\sigma_1, \theta_1, \theta_2\}$. The algorithm restricts these to the variables in $X$, so it now returns one substitution $\{X/a\}$.

### The `qmp:update` Function

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

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

One the other hand, case (ii) above forces the agent to find a new reasonable status set based on the messages received. When the integrity constraints and action constraints of the agent are independent of the $\mathtt{msgbox}$ (I.e. $\mathtt{msgbox}$ is never mentioned in either the integrity or the action constraints of the agent), then the agent is guaranteed to have a new unique reasonable status set. This status set is still $\mathbf{ss}(\mathbf{QMPtab})$! The reason is that when an agent is regular, it can have at most one reasonable status set, viz. $\mathbf{ss}(\mathbf{QMPtab})$. If $\mathbf{ss}(\mathbf{QMPtab})$, when evaluated agains the state of the agent *after* the messages are received, satisfies the integrity and action constraints, then we know that it is reasonable. However, if $\mathtt{msgbox}$ is never mentioned in either the integrity or the action constraints of the agent, then we are guaranteed that $\mathbf{ss}(\mathbf{QMPtab})$, evaluated on the new agent state, provides the new reasonable status set of the agent

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

- Create a graph whose vertices are action status atoms as follows:
  - For each row $r$ in **QMPtab** such that $r$.Opr $= \mathbf{Do}$ and $r$.Action $= \alpha(\vec{X})$ create the node $\mathbf{Do}\,\alpha(\vec{X})$.
  - If $\mathsf{Op}\,\beta(\vec{X})$ is a node in the graph, corresponding to the row $r$ in **QMPtab** with $r$.Opr $=$ $\mathsf{Op}$ and $r$.Action $= \beta(\vec{X})$, and a negated action status atom $\neg\mathsf{Op}'\beta'(\vec{t})$ appears in $r$.PFC, then $\mathsf{Op}'\beta'(\vec{Y})$ is a node in the graph, where $\mathsf{Op}'\beta'(\vec{Y})$ is in **QMPtab**, and there is an edge in the graph from $\mathsf{Op}'\beta'(\vec{Y})$ to $\mathsf{Op}\,\beta(\vec{X})$.

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

- **while** there is an unmarked node left **do**
  - Select an action status atom $\mathsf{Op}\,\beta(\vec{X})$ such that all $B \in pred(\mathsf{Op}\,\beta(\vec{X}))$ are marked.
  - Evaluate $\mathsf{Op}\,\beta(\vec{X})$ using $\mathtt{qmp}:eval(-)$. (By selecting only action status atoms using the previous condition, we know that the negative action status atoms associated with $\mathsf{Op}\,\beta(\vec{X})$'s associated pf-constraint have been fully evaluated).
  - Mark the node labeled $\mathsf{Op}\,\beta(\vec{X})$.

- Let

$$DoSet = \bigcup_{DoNodes} \{\mathbf{Do}\,(\vec{X}\sigma) \mid \sigma \in \mathtt{qmp}:eval(\mathbf{Do}\,(\beta(\vec{x})), O_{\mathcal{S}}, \mathcal{S})\},$$

where *DoNodes* is the set of all nodes in this graph labeled with an action status atom of the form **Do** $(-)$.

- Execute the action **conc**$(DoSet)$.

## 12.6 The *IMPACT* Agent Development Environment (*IADE*)

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

- First, it provides an easy to use, network accessible graphical user interface through which an agent developer can specify the data types, functions, actions, integrity constraints, action constraints, notion of concurrency and agent program associated with his/her agent.

- Second, it provides support for compilation and testing. In particular, *IADE* allows the agent developer to specify various parameters (e.g., conflict freedom test, finiteness table) he wants to use for compilation. It implements the `qmp:init` algorithm and also accesses the `qmp:eval`

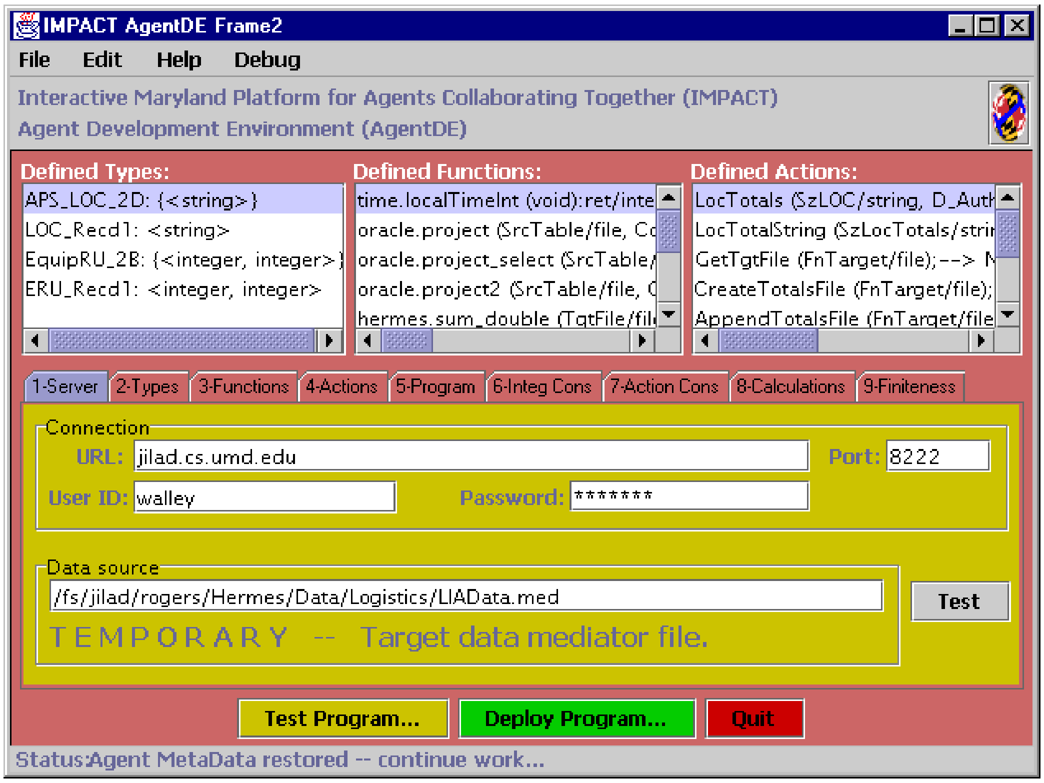



Figure 12.3: Main *IADE* Screen

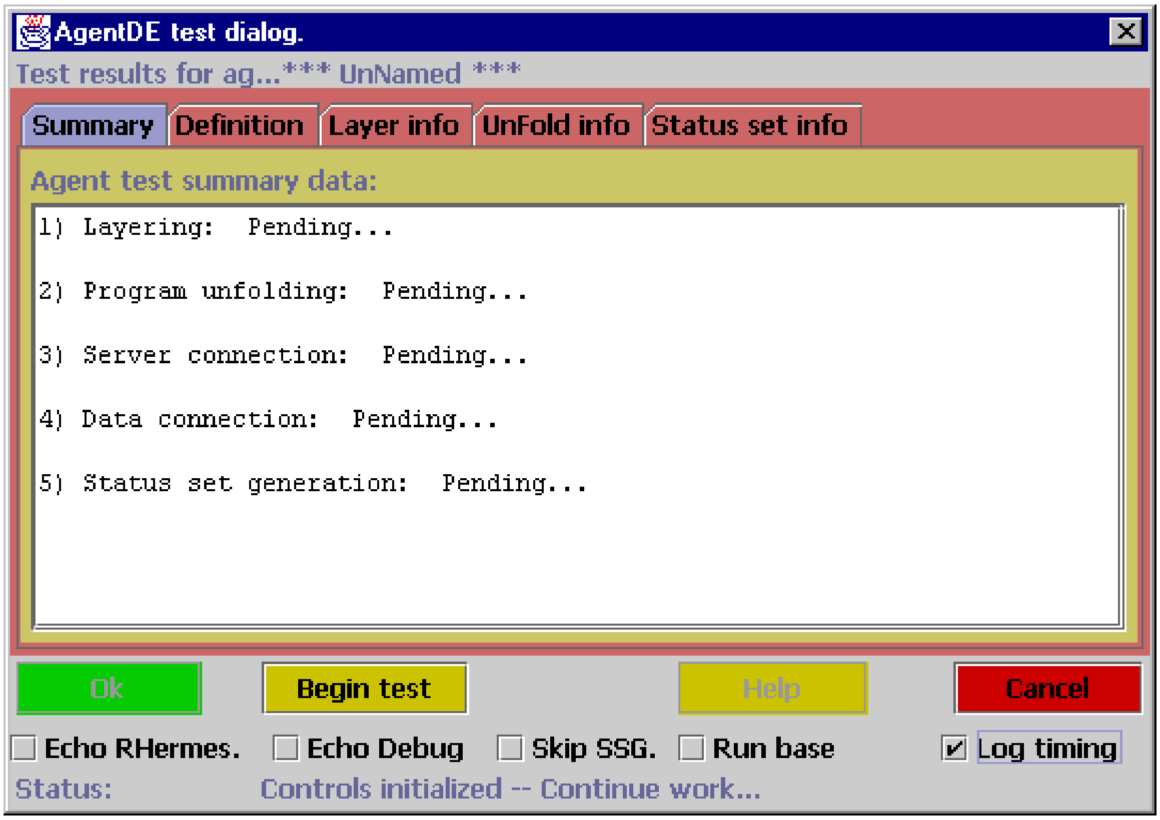



Figure 12.4: *IADE* Test Dialog Screen Prior to Program Testing

algorithm. It allows the agent developer to view the reasonable status set associated with his agent program w.r.t the current state of the agent.

The *IADE* includes the safety, strong safety, conflict freedom algorithms, and the **Check_WRAP** algorithms (the last is slightly modified). The unfold algorithm currently works on positive agent programs—this is being extended to the full fledged case.

Figure 12.3 on the preceding page shows a screendump of *IADE*'s top-level screen. When the agent developer brings up this screen, he may use the "URL" and "PORT" entries in this screen to specify connect information for his agent. He can specify where the data objects associated with this agent are located in the "Data source" entry. The three windows labeled "Defined Types," "Defined Functions," and "Defined Actions" at the top allow the user to browse types, functions and actions defined to date, using the scrollbars shown. The nine tabs below allow the agent developer to specify new types, functions, etc. The tab marked "calculations" allows the user to specify the conflict freedom test he wants to use, and the notion of concurrency he wants to use, while the tab marked "finiteness" allows the user to specify the finiteness table.

Figure 12.4 specifies what happens when the agent developer presses the "Test Program" button in the Figure 12.3 on the preceding page screen. The new dialog that comes up is a debug window that shows that no layering has been done (yet), no unfolding has been done yet, etc.

When the user presses the "Begin Test" button in the screen of Figure 12.4, the window shown in the screen of Figure 12.5 on the next page is brought up. It shows the time taken for program unfolding, the fact that the server connection has been tested and established to work, etc. In the

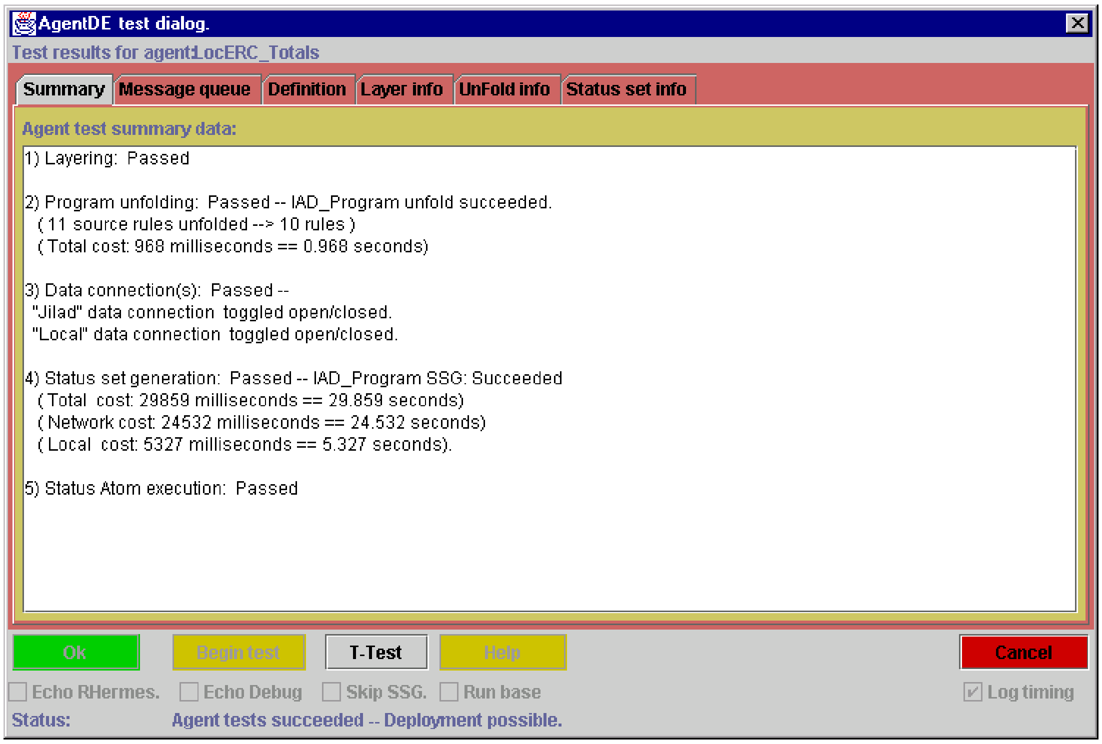



Figure 12.5: *IADE* Test Execution Screen

specific example screen shown in Figure 12.5, the agent program being considered is one used for a live logistics demonstation on a large amount of US Army War Reserves data. This application will be described in further detail in Chapter 13.

Once the status sets have been generated after the test execution phase is completed, the user can press the “Unfold Info” tab (to see the unfolded program) or the “Layer Info” tab (to see the layers of the agent program) or the “Status Set Info” tab (to see status information). Figure 12.6 on the next page shows the results of viewing the unfold information.

When the user selects the “Status set Info” tab, he sees the screen shown in Figure 12.7 on the facing page. Note that this screen has tabs on the right, corresponding to the various deontic modalities. By selecting a modality, the agent developer can see what action status atoms associated with that modality are true in the status set. Figure 12.7 on the next page shows what happens when the user wishes to see all action status atoms of the form **Do** $(\ldots)$ in the status set.

Figure 12.8 on page 424 shows the interface used to specify the “finiteness” table. As mentioned earlier on in this chapter, in the *IMPACT* implementation, we actually represent code calls that are infinite in this table, using some extra syntax. Specifically, the first row of the table shown in Figure 12.8 on page 424 says that when $Q > 3$ and $R > 4$, all code calls of the form $\mathtt{domain}_1 : \mathit{function}_1(\mathtt{Q},\mathtt{R})$ are infinite.

Figure 12.9 on page 424 shows the interface used by the agent developer to specify what notion of concurrency he wishes to use, what conflict freedom implementation he wishes to use and what semantics he wishes to use. Each of the items in the figure have associated drop-down menus (not visible in the picture). The last item titled “Calculation Method” enables us (as developers of *IMPACT*) to test different computation algorithms. It will be removed from the final *IMPACT* release.

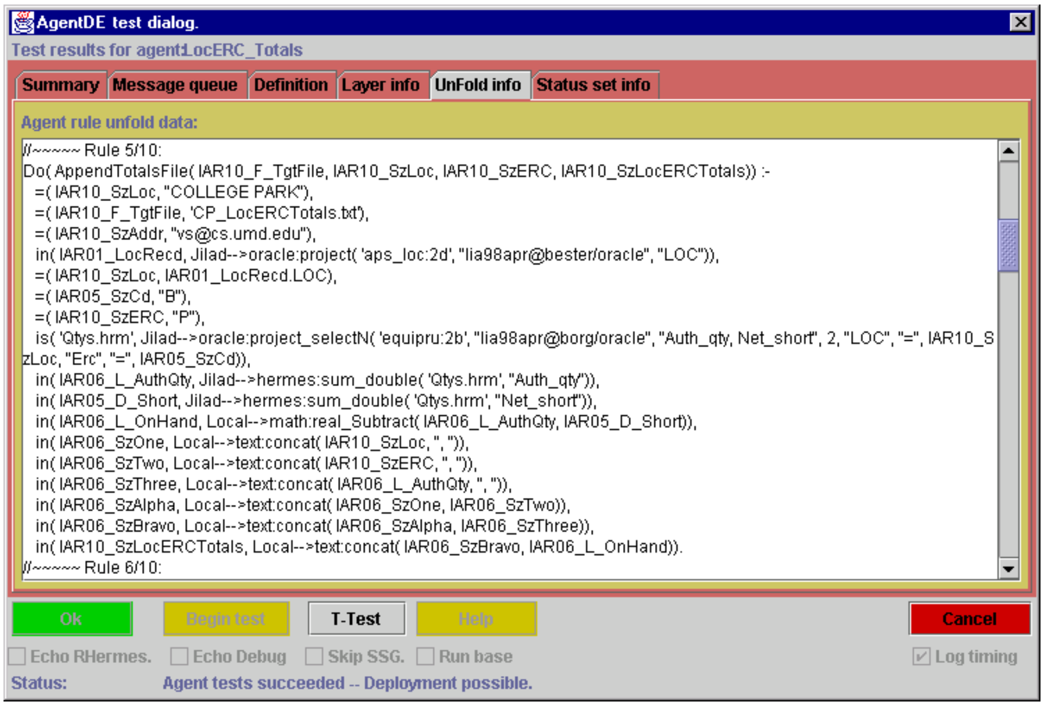



Figure 12.6: *IADE* Unfold Information Screen

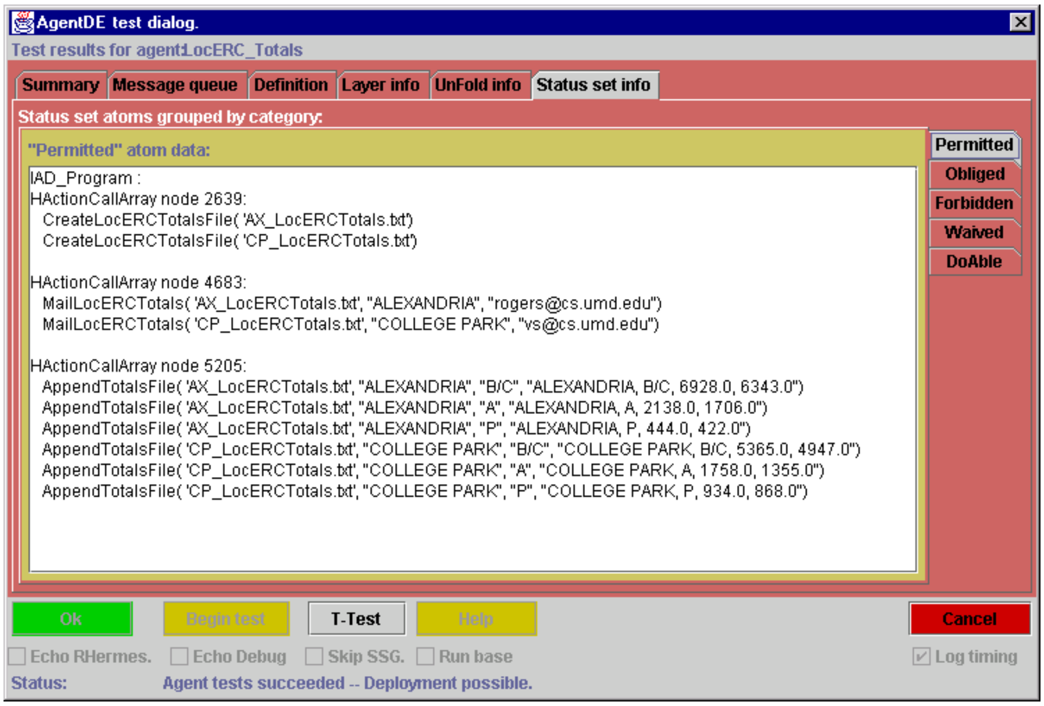



Figure 12.7: *IADE* Status Set Screen

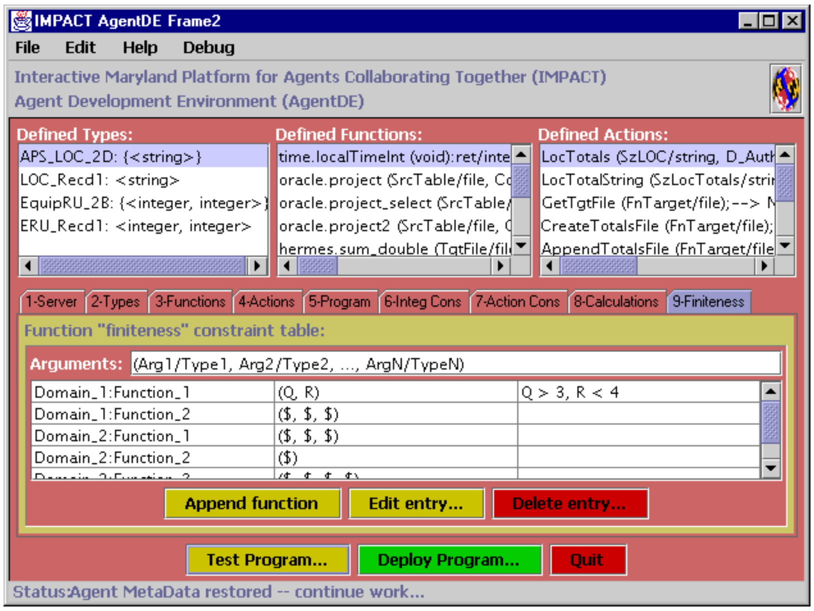



Figure 12.8: *IADE* (In-)Finiteness Table Screen

It is important to note that the above interfaces are only intended for use by the agent developer, and the status set computations shown in Figure 12.7 on the page before are for the agent developer's testing needs. The run-time execution module runs as a background applet and performs the following steps: (i) Monitoring of the agent's message box, (ii) execution of the **Update_Reasonable_SS** algorithm, and (iii) concurrent execution of the actions $\alpha$ such that **Do** $(\alpha)$ is in the updated reasonable status set.

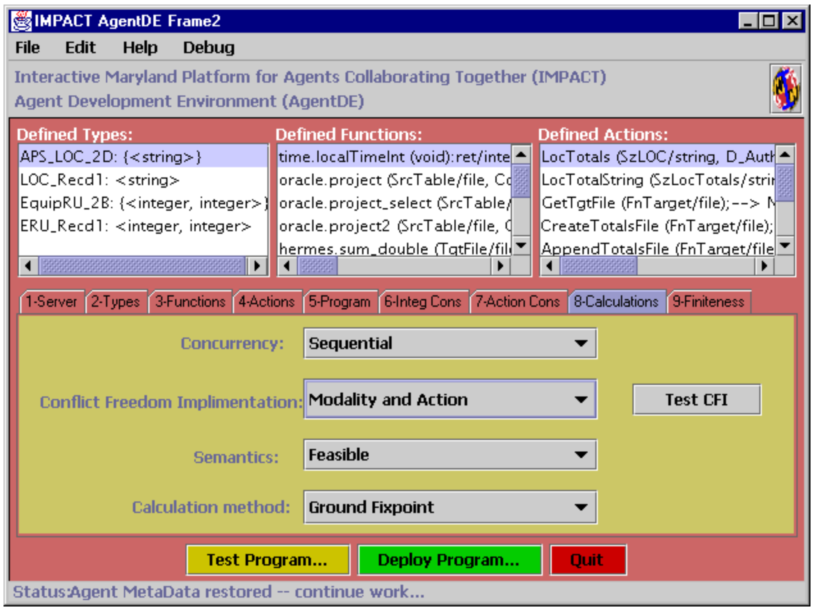



Figure 12.9: *IADE* Option Selection Screen

## 12.7 Experimental Results

In this section, we overview experiments with different aspects of the *IMPACT* Agent Development Environment.

### 12.7.1 Performance of Safety Algorithm

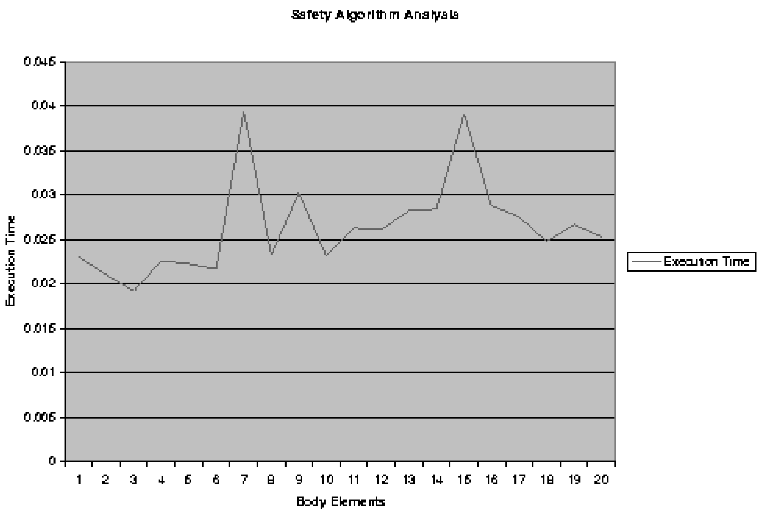


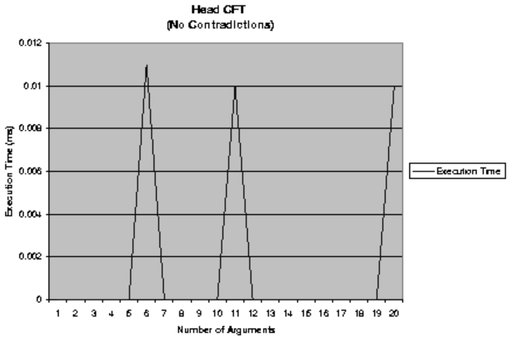

Figure 12.10: Safety Experiment Graphs

Figure 12.10 shows the performance of our implemented safety check algorithm. In this experiment, we varied the number of conjuncts in a code call condition from 1 to 20 in steps of 1. This is shown on the $x$-axis of Figure 12.10. For each $1 \leq x \leq 20$, we executed the **safe_ccc** algorithm 1000 times, varying the number of arguments of each code call from 1 to 10 in steps of 1, and the number of root variables occurring in the code call conditions from 1 to twice the number of conjuncts (i.e., 1 to $2x$). The actual conjuncts were generated randomly once the number of conjuncts, number of arguments, and number of root variables was fixed. For each fixed number $1 \leq i \leq 20$ of conjuncts, the execution time shown on the $y$-axis represents the average over 1000 runs with varying values for number of arguments and number of variables. Times are given in milliseconds. The reader can easily see that algorithm **safe_ccc** is extremely fast, taking between 0.02 milliseconds and 0.04 milliseconds. Thus, checking safety for an agent program with a 1000 rules can probably be done in 20-40 milliseconds.

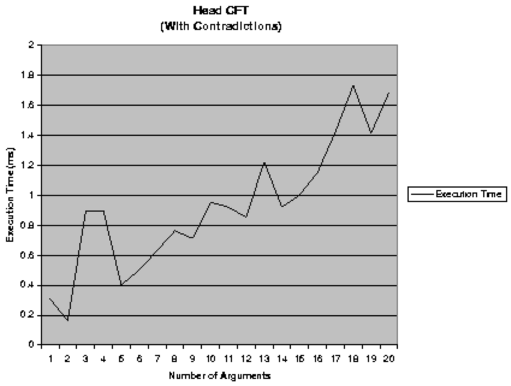
Notice that the bounds used in our experiments are a good reflection of reality—we do not expect to see many agent programs with more than 20 conjuncts in the code call part of a single rule body. This is both difficult for a human being to write, and is difficult to read.

### 12.7.2 Performance of Selected Conflict Freedom Tests

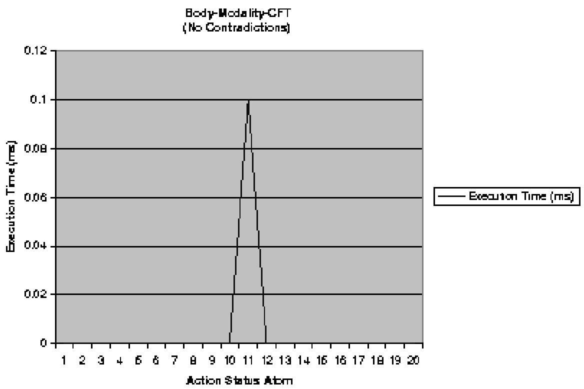
In *IADE*, we have implemented the Head-CFT and Body-Modality-CFT—several other CFTs are being implemented to form a library of CFTs that may be used by agent developers. Figure 12.11 on the following page shows the time taken to execute the Head-CFT and Body-Modality-CFTs. Note that Head-CFT is clearly much faster than Body-Modality-CFT when returning "false"—however,



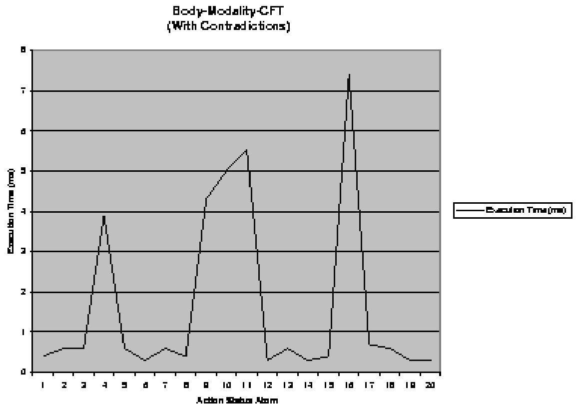

(a) HeadCFT returning “**true**”




(b) HeadCFT returning “**false**”




(c) BodyModalityCFT returning “**true**”




(d) BodyModalityCFT returning “**false**”

Figure 12.11: Performance of Conflict Freedom Tests

this is so because Head-CFT returns “false” on many cases when Body-Modality-CFT does not do so. However, on returns of “**true**,” both mechanisms are very fast, usually taking time on the order of $\frac{1}{100}$ to $\frac{1}{10}$ of a millisecond, with some exceptions. These very small times also explain the “zigzag” nature of the graphs—even small discrepancies (on the order of $\frac{1}{100}$ of a second) appear as large fluctuations in the graph. Even if an agent program contains a 1000 rules (which we expect to be an exceptional case), one would expect the Body-Modality-CFT to only take a matter of seconds to conduct the one-time, compile-time test—a factor that is well worth paying for in our opinion.

### 12.7.3 Performance of Deontic Stratification Algorithm

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

In our experiments, we randomly varied the number of rules from 0 to 200 in steps of 20, and ensured the there were between $V$ and $2V$ edges in the resulting graph, where $V$ is the number of rules (vertices). The precise number was randomly generated. For each such selection, we performed twenty runs of the algorithm. The time taken to generate the graphs was included in these experimental timings. Figures 12.8 on the next page (a) and (b) show the results of our experiments.

Figure 12.8 on the following page(a) shows the time taken to execute all but the safety and conflict freedom tests of the **Check_WRAP** algorithm. The reader will note that the algorithm is very fast, taking only about 260 milliseconds on an agent program with 200 rules. Figure 12.8 on the next page(b) shows the relationship between the number of SCCs in a graph, and the time taken to compute whether the agent program in question is deontically stratified. In this case, we note that as the number of SCCs increases to 200, the time taken goes to about 320 milliseconds. Again, the deontic stratifiability requirement seems to be very efficiently computable.

### 12.7.4 Performance of Unfolding Algorithm

We were unable to conduct detailed experiments on the time taken for unfolding and the time taken to compute status sets as there are no good benchmark agent programs to test against, and no easy way to vary the very large number of parameters associated with an agent. In a sample application shown in Figures 12.6 on page 423 and 12.7 on page 423, we noticed that it took about 1 second to unfold a program containing 11 rules, and to evaluate the status set took about 30 seconds. However, in this application, massive amounts of Army War reserves data resident in Oracle as well as in a multi-record, nested, unindexed flat file were accessed, and the time reported (30 seconds) includes times taken for Oracle and the flat file to do their work, plus network times. Network cosr alone is about 25 seconds. We did not yet implement any optimizations, like caching etc.

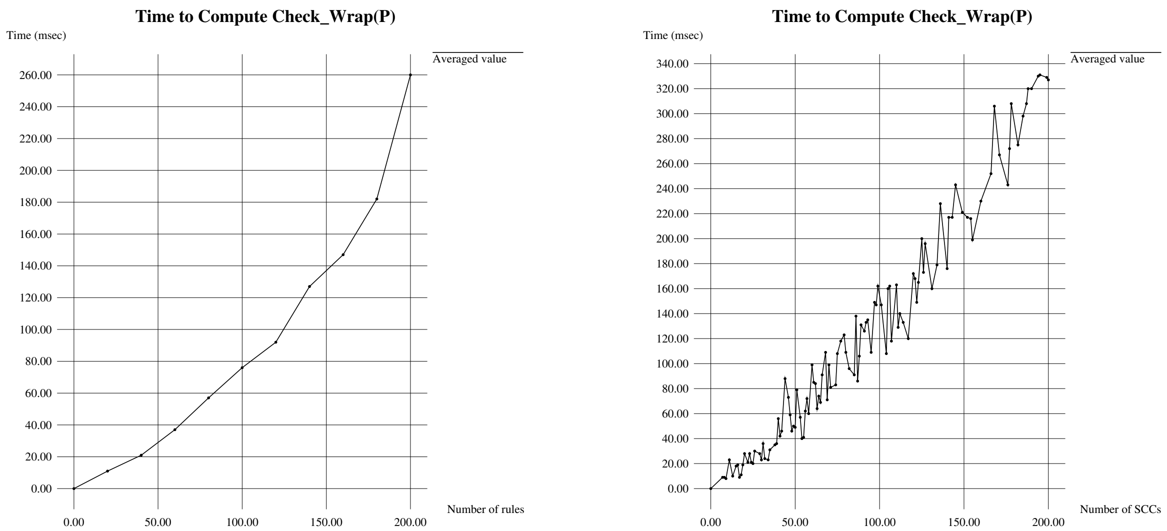
## 12.8 Related Work

There has been relatively little work in defining a formal semantics for agent programming languages: Exceptions include the various pieces of work described in Chapter 6. In this chapter, we have attempted to define a polynomially implementable class of agent programs and described how we implemented this class of programs.

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

- The concept of safety is related to the notion of *mode realizability* in logic programs (Rouzaud and Nguyen-Phoung 1992; Boye and Maluszynski 1995). In order to evaluate the truth or falsity of some atoms in a logic program, certain arguments of that atom may need to be instantiated. This is similar, but not identical to the notion of safety where we have similar conditions on code call conditions. Strong safety requires the important finiteness property in addition to this.

- The concept of conflict freedom has been studied in logic programming when negations are allowed in both the head and the body of a rule. Such logic programs were introduced by (Gelfond and Lifschitz 1991) and contradiction removal in such programs was studied extensively by Pereira's group (Alferes and Pereira 1996). Our work differs from these in the sense that we are looking for syntactic conditions on agent programs (rather than logic programs) that guarantee that under all possible states of the agent, conflicts will not occur. Such a test can be encoded at compile time.




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

Figure 12.12: Performance of Deontic Stratification

- The notion of deontic stratifiability of an agent program, builds directly on top of the concept of a stratified logic program introduced by Apt and Blair (1988). We extend the concept of stratified logic programs to the case of a deontic stratified agent program modulo a conflict freedom test. Checking deontic stratifiability is somewhat more complex than checking ordinary stratifiability, and hence, our algorithms to do this are new.

- The notion of boundedness of an agent program builds upon the well known idea of unfolding (or partial evaluation) in logic programs. This area has been recently studied formally for semantics of (disjunctive) logic programs (wellfounded as well as stable) in (Brass and Dix 1997; Brass and Dix 1998; Brass and Dix 1999). The use of Tarjan's algorithm for computing the well-founded semantics in almost linear time has been explicitly addressed, e.g., in (Berman, Schlipf, and Franco 1995; Dix, Furbach, and Niemela 1999).

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

# Chapter 13

# An Example Application

Based on the theory of agent programs defined in this book, we have developed a significant, highly non-trivial logistics application for the US Army Logistics Integration Agency's War Reserves planning. In this chapter, we will:

- Describe the *War Reserves data set problem* faced by the US Army;
- Describe the architecture used to address this problem;
- Describe our solution to the above problem, using *IMPACT*.

## 13.1 The Army War Reserves (*AWR*) Logistics Problem

At any given point of time, the US Army has a set of ships deployed worldwide containing "prepositioned stocks." Whenever a conflict arises anywhere in the world, one or more of these ships can set sail to that location, and the prepositioned stocks on board those ships can be immediately used to set up a base of operations.

However, this strategy would not be very useful if the stocks on the ship are either (i) insufficient, or (ii) sufficient, but not in proper working order. Readiness of these ships refers to the answer to the question: Does the ship have most of the items it should have on board the ship in proper working order?

As the *AWR* data describing the contents and readiness of the ships in question has evolved over the years, there has been considerable variance in the formats and structures in which data has been stored. Specifically, two data sources are used:

- A body of data is stored in a system called *LOGTAADS* (Logistics – The Army Authorization Document System), which consists of a single-file, multitable structure. In other words, this file consists of a set of distinct (actually four) tables. The *WM_MOC* file contains 68,146 records, no functions were available to access this data—hence, we had to implement our own functions to do so (Schafer, Rogers, and Marin 1998).
- A body of Oracle data. This data contains an *EquipRU* and a *Apr_loc* file comprising of 4,721 and 155 records, respectively.

Logisticians responsible for the readiness of the Army War Reserves need the following types of services in order to successfully accomplish the goal of maintaining high levels of readiness.

1. **Query Services:** They need to be able to execute a variety of queries such as:

   (a) *Find me the "overall status" of all AWR ships?*
   This query may access information on all ships from the `logtaads` and `oracle` data sources, and merge them together, using a crude measure of readiness to define the overall status of a ship. In our implementation, this "crude measure" of readiness merely finds the ratio (termed percentage fill) of the actual number of parts on the ship, to the number of parts that should be on the ship.

   (b) *Find me a breakdown of the fill of the ship Alexandria* [1] *for each supply item class?*
   Supply items on ships are classified into "P" items, "A" items, "B/C" items, "BN TF" items, "BDE CS/CSS" items and "EAD CS/CSS" items. For example, if the logistician finds that the percentage fill mentioned above is too crude an estimate for his requirements, he may pose this more sophisticated query in order to obtain a clearer picture of the state of the different ships.

   (c) *Find me unit level information on the "BN TF" supply items in the Alexandria?*
   This query may be posed when the logistician is still not satisfied with the level of detail—here he wants to know exactly how many of each "BN TF" item are actually on the ship.

2. **Alert Services:** In addition, logisticians need to be able to obtain automatic "alerts" when certain conditions arise. For example, a logistician tracking the Alexandria may wish to obtain an alert by e-mail every time the percentage fill on the Alexandria drops below 80%. Alternatively, another logistician may want to receive an e-mail whenever the percentage fill of "BN TF" items drops below 85%. In such a case, he may want the unit level data as well for the Alexandria to be mailed to him.

3. **Update Services:** Third, multiple logisticians may be working with and manipulating the *AWR* data over time. Whenever an update occurs to the *AWR* data set (either made by the logistician or elsewhere), the effect of these updates need to be incorporated and actions such as the alert services described above might need to be taken.

4. **Analytic Services:** The alert services may be closely coupled to analytic services. For example, when the percentage fill on the Alexandria drops below 70%, the logistician may want an Excel chart to be created, showing a graphical rendering of supply items (x-axis) and percentage fill for that supply item alone (on the y-axis). Instead of mailing the raw data to the appropriate recipients, he may want this chart to be mailed instead. Creating such a chart requires the ability to interoperate with an Excel "agent."

In our current implementation, we have implemented Query services and Alert services, but still have not completed the implementation of update services and analytic services—something we plan to do shortly.

## 13.2 *AWR* Agent Architecture

Figure 13.1 shows the *AWR* architecture we are planning to build for this application. The items shown in yellow are ones we have implemented, while those shown in green are ones still to be implemented.

[1] Owing to the sensitive nature of this application, names of US Army ships have been replaced with fictitious names, and the data shown below is not "real" data.

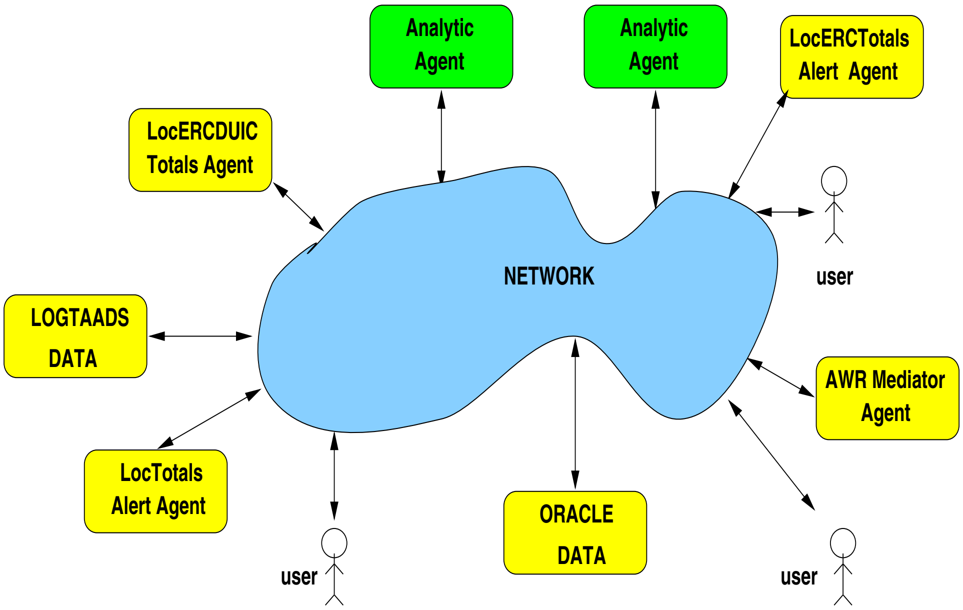



Figure 13.1: Architecture of the multiagent *AWR* system

Both the *LOGTAADS* and Oracle data sets are accessible via agents built by us that support accessing those data sets. In addition, there is a `awrMediator` agent that provides an integrated view across these two data sources.

- The `locTotals` agent provides one and only one service. When requested to provide high level percentage fill data on US Army ships, it returns a table having four attributes—a ship's name, its authorized quantity, its on-hand quantity, and its percentage fill (which is the ration of the on-hand quantity to the authorized quantity).

- The `locERCTotals` agent also provides only one service. Given any ship, it is capable of creating a composite file containing the breakdown by category ("P," "A," "B/C," "BN TF," "BDE CS/CSS," and "EAD CS/CSS") for the ship in question. As different users express interests in different ships, it then e-mails them the ship's ERC totals at a specific time each day.

- The `locERCDUICTotals` agent can provide much more detailed information. Instead of providing aggregate information about a set of items (e.g., all "A" items or all "BN TF" items), it provides on hand quantity, and authorized quantity information for all authorized supply items.

At his stage, we have not implemented the analytic agents shown in Figure 13.1. We are currently working on this.

## 13.3 *AWR* Agent Implementation

In this section, we briefly describe the way we implemented two of the agents described above.

### 13.3.1 The locTotals Agent

We describe below, the various components associated with the locTotals agent. The main aim of this agent is to notify a set of subscribers about the status of all *AWR* ships. Each subscriber specifies a time at which they want to be notified (e.g., at 8 am daily, every Monday at 8 am, etc.), and where (i.e., at what e-mail address) they wish to be notified at those times.

#### Types

This particular agent manipulates four data types:

- `LOC_Recd1`
  which has the schema: (*LOC*/**`String`**).
- `APS_LOC_2D`
  which is a set of `LOC_Recd1` records.
- `ERU_Recd1`
  which has the schema: (*R_Auth_Qty*/**`Integer`**,*R_Net_short*/**`Integer`**).
- `EquipRU_2B`
  which is a set of `ERU_Recd1` records.

#### Functions

The following ten functions are supported by the locTotals agent. Notice that the locTotals agent spans five packages—Oracle, a *HERMES* package, a math package, a time package, and a string manipulation package.

- time:*localTimeInt*() $\rightarrow$ `Integer`.
  This code call returns the current local time.
- oracle:*project*(`SrcTable`, `ConInfo`, `ProjectFlds`) $\rightarrow$ `APS_LOC_2D`.
  This indicates that one of the functions supported by the locTotals agent is a call to Oracle, on the data types listed. The calls returns an object of type *APS_LOC_2D*, i.e., it returns a set of records containing a single string field.
- oracle:*project_select*(`SrcTable`, `ConInfo`, `ProjectFlds`, `ConsField`, `ConsOp`, `Constr`) $\rightarrow$ `EquipRU_2B`.
  This function returns as output, a set of pairs $\langle$*R_Auth_Qty*/**`Integer`**, *R_Net_short*/**`Integer`**$\rangle$.
- oracle:*project2*(`SrcTable`, `ConInfo`, `ProjectFlds`) $\rightarrow$ `polymorphic`.
  This function takes as input an Oracle table located at a specified location and a set of fields, and projects out the appropriate fields. Note the polymorphic nature of this function—the output type depends on the specified fields. The *IADE* can automatically infer the output type.
- In addition, some "workhorse" functions supported are:
  - hermes:*sum_double*(`TgtFile`, `SzTgtField`) $\rightarrow$ `Integer`.
  - math:*subtract*($\mathtt{Val}_1$, $\mathtt{Val}_2$) $\rightarrow$ `Integer`.

– $\mathtt{math}$:*add*($\mathtt{Val}_1$,$\mathtt{Val}_2$) $\rightarrow$ `Integer`.
– $\mathtt{math}$:*multiply*($\mathtt{Val}_1$,$\mathtt{Val}_2$) $\rightarrow$ `Integer`.
– $\mathtt{math}$:*divide*($\mathtt{Val}_1$,$\mathtt{Val}_2$) $\rightarrow$ `Integer`.
– $\mathtt{string}$:*concat*($\mathtt{Val}_1$,$\mathtt{Val}_2$,$\mathtt{Val}_3$,$\mathtt{Val}_4$,$\mathtt{Val}_5$) $\rightarrow$ `String`.

Note that the above math operations represent integer, rather than real-valued arithmetic.

**Actions**

The following nine actions are supported by the locTotals agent. In *IMPACT*, there is no need to specify preconditions, add and delete lists for generic file manipulation actions and message management actions as these are handled automatically. The preconditions, add and delete lists for the other actions are empty.

- *LocTotals*(SzLOC,D_AuthQty,D_OnHand).
  This action takes a ship name (SzLOC) as input and computes its authorized quantity and its OnHand quantity from the *LOGTAADS* and Oracle data.
- *LocTotalString*(SzLocTotals).
  This action is used to convert the answer returned by the *LocTotals* action above into a string.
- *GetTgtFile*(FnTarget).
  This action gets a file.
- *CreateTotalsFile*(FnTarget) .
  This action creates a file.
- *AppendTotalsFile*(FnTarget,SzLocTotals).
  This action takes a target file and a string denoting the totals for a ship, and concatenates the latter to the existing file. The idea is that this action is called once for each ship.
- *LogLocTotals*(FnTarget).
  This action executes the *CreateTotalsFile* action followed by the *AppendTotalsFile* action.
- *ValidateTimeInterval*(L_Hour,L_MinuteSpan).
  This action validates the given time interval.
- *GetEmailAddr*().
  This action gets an e-mail address. For now, this action is hardcoded to just get one e-mail address.
- *MailLocTotalsFile*(R9_TgtFile,R9_SzAddr).
  This action mails the specified file to the specified e-mail address.

**Agent Program**

We list below the 13 rules in this agent program.

**r1: Do** *LocTotals*(SzLOC,D_AuthQty,D_OnHand) $\leftarrow$
**in**(LocRecd, oracle:*project*('*aps_loc* : *2d*', "*lia98apr@bester/oracle*", "LOC")),

**is**('*Qtys.hrm*', oracle:*project*( '*equipru* : *2b*', "*lia98apr@bester/oracle*", "Auth_qty, Net_short", )),
"LOC", "=", *LocRecd.LOC*
=(SzLOC, LocRecd.LOC),
**in**(AuthSum, hermes:*sum_double*('*Qtys.hrm*', "Auth_qty")),
=(D_AuthQty, AuthSum),
**in**(ShortSum, hermes:*sum_double*('*Qtys.hrm*', "Net_short")),
**in**(OnHandSum, math:*subtract*(AuthSum, ShortSum)),
=( D_OnHand, OnHandSum).

This rule executes an action called *LocTotals* that accesses an Oracle relation at a location denoted by *lia98apr@bester* and computes the totals for all ships, together with the percentage fills. It is important that the *LocTotals* action has no effects—so in fact, this rule only serves to say that a set of ground status atoms of the form *LocTotals*(SzLOC, D_AuthQty, D_OnHand) is in any status set of the agent. The following rule takes the set of these ground status atoms and merges them into a massive string.

**r2:** **Do** *LocTotalString*(R2_SzLocTotals) ←
**Do** *LocTotals*(R2_SzLoc, R2_D_AuthQty, R2_D_OnHand),
**in**(R2_T_SzLocTotals, string:*concat*(R2_SzLoc, ",", R2_D_AuthQty, ",", R2_D_OnHand)),
=(R2_SzLocTotals, R2_T_SzLocTotals).

**r3:** **Do** *GetTgtFile*(R3_F_Name) ←
=(R3_F_Name, '*LocTotals.txt*').
This rule gets a file called '*LocTotals.txt*'.

**r4:** **Do** *CreateTotalsFile*(R4_TgtFile) ←
**Do** *GetTgtFile*(R4_TgtFile).
Likewise, this rule creates a file called LocTotals.txt. The meaning of the following rules is more or less self-explanatory.

**r5:** **Do** *AppendTotalsFile*(R5_TgtFile, R5_SzLocTotals) ←
**Do** *GetTgtFile*(R5_TgtFile), **Do** *LocTotalString*(R5_SzLocTotals).

**r6:** **Do** *LogLocTotals*(R6_TgtFile) ←
**Do** *CreateTotalsFile*(R6_TgtFile), **Do** *AppendTotalsFile*(R6_TgtFile, R6_SzLocTotals).

**r7:** **Do** *ValidateTimeInterval*(L_TgtHour, L_TgtMinuteSpan) ←
**in**(L_HermesLocalHM, time:*localTimeInt*()),
**in**(L_TgtH, math:*multiply*(L_TgtHour, 60)),
**in**(L_LoTgtHM, math:*subtract*(L_TgtH, L_TgtMinuteSpan)),
**in**(L_HiTgtHM, math:*add*(L_TgtH, L_TgtMinuteSpan)),
>( L_HermesLocalHM, L_LoTgtHM),
<(L_HermesLocalHM, L_HiTgtHM).
The following rules execute the mail actions to a small group of two subscribers—vs and rogers. Depending upon the time of day, the notifications are sent to different e-mail addresses. For example, rule 8 sends e-mail to vs at *vs@earlynet.com* during one time interval, while rule 9 and 10 send him notifications to other addresses at other time intervals. The three rules after that do likewise for rogers.

**r8:** **Do** *GetEmailAddr*(SzAddr) ←
**Do** *ValidateTimeInterval*(5, 240), =(SzAddr, "*vs@earlynet.com*").

**r9:** **Do** *GetEmailAddr*(SzAddr) ←
**Do** *ValidateTimeInterval*(6,240), =(SzAddr, "*vs@cs.umd.edu*").

**r10:** **Do** *GetEmailAddr*(SzAddr) ←
**Do** *ValidateTimeInterval*(15,240), =( SzAddr, "*vs@latenite.com*").

**r11:** **Do** *GetEmailAddr*(SzAddr) ←
**Do** *ValidateTimeInterval*(5,240), =( SzAddr, "*tjrogers@ricochet.net*").

**r12:** **Do** *GetEmailAddr*(SzAddr) ←
**Do** *ValidateTimeInterval*(6,240), =( SzAddr, "*rogers@cs.umd.edu*").

**r13:** **Do** *GetEmailAddr*(SzAddr) ←
**Do** *ValidateTimeInterval*(15,240), =( SzAddr, "*rogers@interserv.com*").

As the reader will see from Figure 12.5, the time taken for unfolding the locTotals agent is relatively small—1.433 seconds. *IADE* contains an optimization—for actions that have no effects, there is no need to evaluate the associated pf-constraint. This causes several rules in the unfolded program to be thrown away which explains why a set of 13 source rules is replaced by 8 rules after the unfolding process is complete. The total time taken to evaluate these rules over the distributed data sources described in Chapter 12 is 25.681 seconds.

### 13.3.2 The locERCTotals Agent

The locERCTotals agent is similar to the locTotals agent. For each ship, instead of producing aggregate level information, it provides information for the various categories of supply items. We now describe the design of this agent.

#### Types

The locERCTotals agent uses the same four types used by the locTotals agent. This is not a surprise, as the data accessed by this agent is identical to the data accessed by the locTotals agent.

#### Functions

The locERCTotals agent uses the same functions used by the locTotals agent except for the oracle:*project2*() and math:*divide*() functions.

#### Actions

The nine actions implemented in the locERCTotals agent are listed below. Unless mentioned otherwise, all actions have empty preconditions, add lists and delete lists.

- *Loc_Names*(SzLoc).
  This action computes the name of all ships used to store the Army War Reserves.

- *Cd2ERC*(SzCD,SzERC).
  This action generates something called a "pacing" code from the data character passed into it.

- *LocERC_Totals*(SzLoc, SzERC, L_AuthQty, L_OnHand).
  This action is similar to the *LocTotals* action in the locTotals agent—the main difference is that given any ship (SzLoc) and any supply category (SzERC), it finds the totals for that category.
- *LocERC_TotalString*(SzLoc, SzERC, SzLocERCTotals).
  This action is similar to the *LocTotalString*(a)ction described for the locTotals agent. Given any ship (SzLoc) and any supply category (SzERC), it converts the totals for that ship and supply category into a string.
- *ValidateTimeInterval*(L_Hour, L_MinuteSpan).
  This action is identical to the one used by the locTotals agent.
- The following actions are self explanatory:
  - *GetTargets*(F_Name, SzLoc, SzAddr).
  - *CreateLocERCTotalsFile*(F_TgtFile).
    This action executes a body of code which creates such a file.
  - *AppendTotalsFile*(F_TgtFile, SzLoc, SzERC, SzLocERCTotals).
    This function also executes a body of code that performs the desired append action.
  - *LogLocERCTotals*(F_TgtFile, SzLoc, SzERC).
  - *MailLocERCTotals*(F_TgtFile, SzLoc, SzAddr).

### Agent Program

The agent program associated with the locERCTotals agent contains a total of 17 rules and is shown below.

**r1:** **Do** *Loc_Names*(SzLoc) ←
 **in**(LocRecd, oracle : *project*('*aps_loc : 2d*', "*lia98apr@bester/oracle*", "LOC")),
 =( SzLoc, LocRecd.LOC).

**r2:** **Do** *Cd2ERC*(SzCode, SzERC) ←
 =( SzCode, "A"), =( SzERC, "Ship 0 CD code").

**r3:** **Do** *Cd2ERC*(SzCode, SzERC) ←
 =( SzCode, "B"), =( SzERC, "P").

**r4:** **Do** *Cd2ERC*(SzCode, SzERC) ←
 =( SzCode, "C"), =( SzERC, "A").

**r5:** **Do** *Cd2ERC*(SzCode, SzERC) ←
 =( SzCode, "D"), =( SzERC, "B/C").

**r6:** **Do** *LocERC_Totals*(SzLoc, SzERC, L_AuthQty, L_OnHand) ←
 **Do** *Loc_Names*(SzLoc),
 **Do** *Cd2ERC*(SzCd, SzERC),
 **is**('*Qtys.hrm*', oracle : *project_selectN*( '*equipru : 2b*', "*lia98apr@borg/oracle*", "Auth_qty", )),
 "Net_short", 2, "LOC", "=", SzLOC, "Erc", "=", SzCd
 **in**(L_AuthQty, hermes : *sum_double*('*Qtys.hrm*', "Auth_qty")),
 **in**(L_Short, hermes : *sum_double*('*Qtys.hrm*', "Net_short")),
 **in**(LOnHand, hermes : *subtract*(L_AuthQty, L_Short)).

**r7:** **Do** *LocERC_TotalString*(SzLoc, SzERC, SzLocERCTotals) ←
  **Do** *LocERC_Totals*(SzLoc, SzERC, L_AuthQty, L_OnHand),
  **in**(SzLocERCTotals, string:*concat*(SzLoc, ",", SzERC, ",", L_AuthQty, ",", L_OnHand, ",")).

**r8:** **Do** *ValidateTimeInterval*(L_TgtHour, L_TgtMinuteSpan) ←
  **in**(L_HermesLocalHM, time:*localTimeInt*()),
  **in**(L_TgtH, math:*multiply*(L_TgtHour, 60)),
  **in**(L_LoTgtHM, math:*subtract*(L_TgtHour, L_TgtMinuteSpan)),
  **in**(L_HiTgtHM, math:*add*(L_TgtH, L_TgtMinuteSpan)),
  >(L_HermesLocalHM, $L_L$oTgtHM),
  <(L_HermesLocalHM, $L_H$iTgtHM).

**r9:** **Do** *GetTargets*(F_Name, SzLoc, SzAddr) ←
  **Do** *ValidateTimeInterval*(8, 240),
  =(SzLoc, "COLLEGE PARK"),
  =(F_Name, '*CP_LocERCTotals.txt*'),
  =(SzAddr, "*vs@cs.umd.edu*").

**r10:** **Do** *GetTargets*(F_Name, SzLoc, SzAddr) ←
  **Do** *ValidateTimeInterval*(8, 240),
  =(SzLoc, "WEST POINT"),
  =(F_Name, '*WP_LocERCTotals.txt*'),
  =(SzAddr, "*dj4149@exmail.usma.army.mil*").

**r11:** **Do** *GetTargets*(F_Name, SzLoc, SzAddr) ←
  **Do** *ValidateTimeInterval*(2, 240),
  =(SzLoc, "WEST POINT"),
  =(F_Name, '*WP_LocERCTotals.txt*'),
  =(SzAddr, "*jschafer@earlybird.com*").

**r12:** **Do** *GetTargets*(F_Name, SzLoc, SzAddr) ←
  **Do** *ValidateTimeInterval*(15, 240),
  =( SzLoc, "WEST POINT"),
  =( F_Name, '*WP_LocERCTotals.txt*'),
  =( SzAddr, "*jschafer@latenite.com*").

**r13:** **Do** *GetTargets*(F_Name, SzLoc, SzAddr) ←
  **Do** *ValidateTimeInterval*(8, 240),
  =( SzLoc, "ALEXANDRIA"),
  =( F_Name, '*AX_LocERCTotals.txt*'),
  =( SzAddr, "*rogers@cs.umd.edu*").

**r14:** **Do** *GetTargets*(F_Name, SzLoc, SzAddr) ←
  **Do** *ValidateTimeInterval*(15, 240),
  =( SzLoc, "ALEXANDRIA"),
  =( F_Name, '*AX_LocERCTotals.txt*'),
  =( SzAddr, "*rogers@latenite.com*").

**r15:** **Do** *CreateLocERCTotalsFile*(F_TgtFile) ←
  **Do** *GetTargets*(F_TgtFile, SzLoc, szAddr).

**r16:** **Do** *AppendTotalsFile*(`F_TgtFile`, `SzLoc`, `SzERC`, `SzLocERCTotals`) ←
**Do** *GetTargets*(`F_TgtFile`, `SzLoc`, `szAddr`).
**Do** *LocERC_TotalString*(`SzLoc`, `SzERC`, `SzLocERCTotals`).

**r17:** **Do** *MailLocERCTotals*(`F_TgtFile`, `SzLoc`, `SzAddr`) ←
**Do** *GetTargets*(`F_TgtFile`, `SzLoc`, `szAddr`).

The *IMPACT* Agent Development environment takes 150 milliseconds to unfold this agent program, rendering the 17 rules listed above as a set of 30 agent program rules. Computing the status set (which involves computing these 30 pf-constraints associated with the unfolded agent program) takes a total of 35.386 seconds, including all network costs.

# Chapter 14

# Conclusions

Chapter 1 of this book lays out a set of basic goals and desiderata that we believe any program called an "agent" must satisfy, and that we believe an infrastructure supporting interactions between multiple agents must satisfy. Subsequently, in Chapters 2 onward, we have described a wide variety of contributions, including the *IMPACT* architecture, how *IMPACT* servers support a wide variety of generic interoperability needs, how *IMPACT* agents may be built, how these agents can make intelligent decisions, reason about commitments both now and in the future, reason about uncertainty, and so on. The important, and as yet unanswered questions at this stage are:

1. *Do the contributions in Chapters 2–13 successfully address the needs and desiderata described in Chapter 1?*

2. *In addition, do these desiderata support the kinds of agent behavior required by other researchers, e.g., Hayes(1999); Lesser(1999)?*

3. Finally, what are the answers to the three important questions, **(Q1), (Q2), (Q3)** raised in Section 1.5. To refresh the reader's mind, these questions are recapitulated below.

   **(Q1)** *What is an agent?*

   **(Q2)** *If program P is not considered to be an agent according to some specified definition of agenthood, is there a suite of tools that can help in "agentizing" P?*

   **(Q3)** *Once a specific definition of agenthood is chosen, what kind of software infrastructure is required to support interactions between such agents, and what core set of services must be provided by such an infrastructure?*

In this chapter, we attempt to answer these questions.

## 14.1 Progress Toward the Ten Desiderata

Section 1.5 provides ten desiderata that, to our mind, any notion of "agenthood" must satisfy. In this section, we review how our framework supports these ten desiderata, and also assess where it falls short and what research is needed to make up the shortfall.

### 14.1.1 Desiderata (D1) and (D2)

Desideratum **(D1)** says that "anybody who has a software program *P*, either custom designed to be an agent, or an existing legacy program, must be able to *agentize* their program and *plug* it into the provided solution." Similarly, Desideratum **(D2)** says that a theory of agents must take into account that data is stored in a wide variety of data structures and is manipulated by an existing corpus of algorithms.

*IMPACT* supports this very strongly, both in its theory and in its implementation. In fact, Chapter 4 is devoted to describing the following concepts: *code call conditions* provide a syntax using which, queries and requests to arbitrary legacy and/or specialized data structures may be executed. It also describes the *state* of the agent as a set of objects of any set of data types whatsoever, as long as the set of data types used by the agent is specified when the agent is built. It supports the concept of an *integrity constraint* over such diverse data structures. Likewise, in Chapter 6 onward, we notice that agent programs, action constraints, temporal agent programs, probabilistic agent programs, etc., are all built on top of arbitrary data structures accessible via the code call mechanism.

The implementation of *IMPACT* uses a pre-existing software package developed at the University of Maryland called *WebHermes* (Adali, S., et al. 1997) which supports execution of code call conditions over a wide variety of data structures and software packages. These currently include (or have included in the past), relational database management systems (Oracle, Ingres, Dbase, Paradox), an object oriented system (ObjectStore), a multimedia system called MACS (Brink, Marcus, and Subrahmanian 1995), a video information system called AVIS (Adali, Candan, Chen, Erol, and Subrahmanian 1996), a geographic data structure called a PR-quadtree, arbitrary flat files (as long as their schemas are specified), a US Army route planner over free terrain (Benton and Subrahmanian 1994), a variety of US Army logistics data including specialized Oracle and nested multirecord TAADS data (Schafer, Rogers, and Marin 1998), a variety of US Army simulation data from a massive program called JANUS deployed by the Simulation, Training and Instrumentation Command, a face recognition program, and so on.

### 14.1.2 Desideratum (D3)

Desideratum **(D3)** says that "a theory of agents must *not* depend upon the set of actions that the agent performs. Rather, the set of actions that the agent performs must be a *parameter* that is taken into account in the semantics."

This is certainly the case in *IMPACT*. When an agent developer builds an agent, s/he can specify, through the *IMPACT* Agent Development Environment, a set of actions that his or her agent can perform. For each of these actions, he specifies a codebase, and a set of preconditions, add and delete lists. This feature is not only present in the implementation—it is also an essential component of the theory underlying agent programs. As we can see in Chapter 6, an agent's associated semantics (feasible status set, rational status set, reasonable status set, etc.) depend fundamentally upon the agent's associated agent program, as well as upon the agent's actions' effects, which determine if a new state (reached by executing these actions) satisfies the integrity constraints or not.

### 14.1.3 Desideratum (D4)

Desideratum **(D4)** says that "every agent should execute actions based on some *clearly articulated* decision policy."

The goal of Chapters 6, 7, 8, and 9 is to allow the agent developer to define decision policies based on his application needs. An agent program, defined in Chapter 6 specifies the operating principles/decision policy of the agent. It provides a number of alternative semantics that an agent developer may "choose" for his agent to use.

Thus, from a *theoretical* point of view, we have successfully provided a language for articulating decision policies based on just a set of data structures if needed, but also based on sophisticated forms of temporal reasoning, meta-reasoning and prediction, and reasoning with uncertainty. From the point of view of *implementation* methods, however, our current *IMPACT* system handles agent programs (actually *regular* agents as described in Chapter 12).

### 14.1.4 Desideratum (D5)

This desideratum says that any agent construction framework must allow agents to perform "reasoning about beliefs."

Chapter 7 extends the concept of an agent program proposed in Chapter 6 to the case where agents reason about the beliefs of other agents, about what other agents believe about other agents, about what other agents are permitted or obligated to do, and so on. It proposes the concept of a *meta-agent program* and shows that the semantics of meta-agent programs cleanly extends that of agent programs. Furthermore, it shows that meta-agent programs can be directly implemented on top of an implementation of ordinary agent programs. The effect of this startling result is that the *IADE* can be used to implement meta-agent programs if its interface is extended to allow the user to articulate his "meta-reasoning" conditions and his belief atoms (see Chapter 7), his belief tables, his belief semantics tables, and so on. We anticipate extending the *IADE* to accomplish this in the near future.

### 14.1.5 Desideratum (D6)

Desideratum **(D6)** of our agent framework says that "any infrastructure to support multiagent interactions *must* provide two important types of security—security on the agent side, to ensure that an agent (if it wishes) can protect some of its information and services, and security on the infrastructural side so that one agent cannot masquerade as another."

As the reader can see from Chapter 10, we have shown that agent security can be split into two components—*data security* and *action security*. We have shown that maintaining security is undecidable in general. However, we have proposed the notions of *approximate security* and shown that *IMPACT* agents can maintain security by trading off some cooperativeness. In addition, our notions of security may build on top of existing, well understood authentication mechanisms. For instance, checking authentication using a specific authentication program is an *action* that an agent may take!

On the implementation side, however, the *IADE* currently forces the agent developer to explicitly specify actions to handle his agent's security needs. In addition, no mechanisms exist in the current *IADE* implementation to specify the various "approximation" components specified in Chapter 10. Successfully addressing this problem is a major priority of our ongoing effort.

Chapter 8 involves another extension to the case where agents may reason about its past, present and future states, as well as reason about actions it is permitted to do, obliged to do, forbidden from doing, both now, in the past, and in the future. Chapter 9 extends agent programs to allow users to articulate decision policies in uncertain environments.

### 14.1.6 Desideratum (D7) and (D8)

Desideratum (D7) says, in effect, that agents must be efficient, relative to an oracle to handle API calls to the underlying code base on top of which the agent is built. Desideratum (D8) says that implementing polynomial fragments of agent programming languages is critical.

We have carefully studied, in Chapter 11, the complexity of computing appropriate status sets of agent programs. Chapter 11's theoretical complexity results show that for positive agent programs with integrity constraints, the reasonable and feasible status sets can be efficiently (i.e., polynomially) computed. In Chapter 12, we provide concrete implementation mechanisms. In particular, we show via our implementation that if we are willing to pay a price at the time an agent is deployed (a one time cost), then we may end up with huge savings at run time as far as status set computation is concerned.

One extension that we are currently working on is to extend *regular* agent programs to the case of meta-agent programs, temporal agent programs, and probabilistic agent programs, so that polynomial versions of those agent programs are also available.

### 14.1.7 Desideratum (D9)

Desideratum (D9) deals with *reliability*—can we ensure that if the "agent infrastructure" crashes, then all agents' activities will not come to a grinding halt?

In *IMPACT*, the *IMPACT* Servers are the ones that provide the so called "agent infrastructure." Agents need to use the server only when either (i) they do not understand what another agent's terminology or defined types refer to, or (ii) when they need to find an agent that provides a specific service they want. We have provided for reliability of *IMPACT* servers via replication of these servers at distributed nodes, as described in Chapter 2, and by minimizing the dependency of individual agents on *IMPACT* servers. Even if all replicas of the *IMPACT* servers go down, agents in *IMPACT* can still function and interact with one another, as long as the underlying communications medium (wired or wireless network) is up.

### 14.1.8 Desideratum (D10)

Last, but not least, desideratum (D10) says that "the only way of testing the applicability of any theory is to build a software system based on the theory, to deploy a set of applications based on the theory, and to report on experiments based on those applications. Thus, an implementation *must be validated* by a set of deployed applications."

As reported in Chapter 13, we have validated our theory and implementation via two applications: one for the US Army Logistics Integration Agency's Army War Reserves agent, while the other is an agent application for the US Army Simulation, Training and Instrumentation Command (STRICOM).

## 14.2 Agent Desiderata Provided by Other Researchers

Our research supports various requirements about agents specified by other researchers. Prior to this work, most definitions of agents were *behavioral* in the sense that whether a program was considered an agent or not depended upon how the program behaved. In contrast, in this book, we have proposed a *structural* definition of agents in which we specify what software components must be contained within an agent. Shoham (1999) provides a list of parameters that have typically been used to characterize a software agent. This list of parameters is a set of behavioral requirements articulated by various researchers. We outline Shoham's requirements (as well as a few more) below, together with an assessment of how *IMPACT* supports these requirements, and where it falls short.

- *Ongoing execution.* This criterion says that software agents function continuously over time. In fact, as described by the Agent Decision Cycle in Chapter 6, the reader can see that all *IMPACT* agents are continuously functioning.

- *Autonomy.* Autonomy means that agents do not require human supervision. Autonomous agents can easily be built in *IMPACT*. We have shown, in Chapter 6, how clock driven agents can be built, which automatically take certain actions based on occurrence of clock events. Likewise, the credit agent in Chapter 8 may automatically notify a customer whose credit has an overdue payment every 10 days. In general, an autonomous agent is one that takes actions without explicit intervention by the user on behalf of whom the action is being taken. *IMPACT* agent programs contain rules that can trigger actions whenever a state change occurs, and as our notion of a state change is powerful enough to include clock ticks, database updates, random event occurrence, etc., our framework is rich enough to trigger actions based on such events.

- *Adaptiveness.* This parameter is also a behavioral parameter which says that agents should be able to adapt over time to the needs of specific users. It is important to note that adapting is an action. In *IMPACT*, an agent can log all transactions performed by a user into its state (thus maintaining a history of the sort shown in Chapter 10), and then determine what actions to take on a user's behalf, and how to perform these actions, based on the log.

- *Intelligence.* This is an extremely vague requirement that agents be able to perform different types of reasoning such as probabilistic reasoning, planning, etc. In *IMPACT*, as shown in Chapter 9, agents can certainly reason in the presence of uncertainty. Furthermore, in *IMPACT*, agents can plan, by specifically triggering planning actions in the agent decision cycle. The coupling of action to state in *IMPACT* is very generic. Actions in *IMPACT* can include invoking route planners and flight planners (as in the CFIT example), performing statistical data mining style analyses (as in the `credit` agent), and so on.

- *Agent Awareness.* This parameter requires that agents be able to model other agents, reason about them, and so on. As we have shown in Chapter 7, using the map paradigm, *IMPACT* agents may reason about the state, as well as the actions of other agents. Furthermore, agents can participate in auctions and engage in strategic negotiations with other agents (Chapter 8).

- *Mobility.* *IMPACT* agents can be mobile, because mobility is an action that any agent can execute. So is cloning. In addition, as shown in Chapter 6, Java applets may themselves be viewed as *IMPACT* agents.

- *Anthropomorphism.* This parameter says that agents should have the ability to deal with beliefs, obligations, and other such modalities, and also convey "emotions" through graphical or voice actions. Of course, this is straightforward to encode in an *IMPACT* agent. By its very design, an agent program allows an agent to explicitly describe what it may or may not do, what its obligations are, and so forth. In the same vein, if the state of the agent satisfies certain conditions executing an action that displays a "happy face" or a "grumpy face" can be directly encoded as an agent program rule.

- *Reactivity.* Reactive agents are those that can automatically react to events. It is apparent from the preceding paragraph that *IMPACT* agents can react to events, because events are state changes as far as the agent is concerned.

- *Evaluating Courses of Action.* Several researchers, such as Bratman, Israel, and Pollack (1988) and Schoppers and Shapiro (1997), require agents to have the ability to evaluate alternative courses of actions, and adopt one that is optimal w.r.t. some objective function of interest to the agent. In Chapter 6, we have defined an "optimal status set" semantics for agent programs that accomplishes this. First, an agent developer can select a semantics for his agent (feasible, rational, reasonable, etc.). Each status set according to the selected semantics is considered to be a possible course of action. The one that is optimal w.r.t. some cost function specified by the agent developer is then selected by the agent, and the actions in it are concurrently executed.

- *Communication.* Another well accepted requirement of agents is the ability to communicate with other agents and to comprehend the content of these communications. In *IMPACT*, agents communicate with other agents via the `msgbox` package. Sending messages, reading messages, flushing the message box, are all actions that the agent might take. Agents can *comprehend* what messages other agents are sending via the *IMPACT* servers that maintain thesaurus and type information.

- *Planning.* As agents are required to act intelligently on behalf of one or more individuals, agents must be able to plan in order to accomplish these objectives. Here, *IMPACT*'s way of accomplishing planning by agents is quite different from methods traditionally adopted in artificial intelligence and in agent research in particular, (Schoppers and Shapiro 1997), Bratman, Israel, and Pollack (1988). In *IMPACT*, *planning* is an action. An agent can plan for a contingency by explicitly invoke a planning code base, specific to the agent. This makes sense to us. Database retrieval agents, for instance, should not be forced to do AI planning. In the CHAIN example, the `truck` agent may plan its route using a road map, while in the CFIT agent, planning may be done using 3-d models of the world. Both require invocation of different planners. Thus, in our framework, we have provided the ability to "plug in" different planners into *IMPACT* agents.

- *Negotiation.* There has been extensive work on negotiation in multiagent systems, based on the initial idea of contract nets, due to Smith and Davis (1983). In Chapter 8, we have provided a detailed example of how strategic models of negotiation can be encoded into *IMPACT* agents.

# Appendix A

# Code Calls and Actions in the Examples

## A.1 Agents in the CFIT Example

### A.1.1 AutoPilot Agents

The `autoPilot` agent ensures that the plane stays on its allocated flight path.The task of the `autoPilot` agent is to ensure that the plane stays on-course, and make appropriate adjustments when the physical dynamics of the plane cause it to veer off course.

#### Code Calls

1. Determine the current location of the plane.
   `autoPilot`:*location*() → `3DPoint`
2. Determine the pilot status of the plane, ie whether it is *automated, semi-automated, semi-manual* or *manual*.
   `autoPilot`:*pilotStatus*(`Pilot_message`) → `PilotStatusString`
3. Get the current flight route of the plane
   `autoPilot`:*getFlightRoute*() → `Path`
4. Get the next `SensorId`, each time this code call is called the next sensor from the queue is returned. All the sensors are maintained in a circular queue.
   `autoPilot`:*getSensorId*() → `String`
5. Read the GPS Data from the sensor with `SensorId`
   `autoPilot`:*readGPSData*(`SensorId`) → `GPSDataReport`
6. Given the `CurrentLocation`, `CurrentFlightRoute` and a set of `No_go` regions, adjust the flight route of the plane
   `autoPilot`:*adjustFlightRoute*(`CurrentFlightRoute`, `CurrentLocation`, `No_go`) → `Path`
7. Given the `CurrentLocation` of the plane and a set of `No_go` areas, calculate a flight path for the plane
   `autoPilot`:*calculateFlightRoute*(`CurrentLocation`, `No_go`) → `Path`
8. Given the `CurrentLocation` of the plane and a set of `No_go` areas, calculate `N` alternative flight paths for the plane
   `autoPilot`:*calculateNFlightRoutes*(`CurrentLocation`, `No_go`, `N`) → `ListofPaths`

9. Get the current `Altitude` of the plan
   $\mathtt{autoPilot}$:*getAltitude*() $\rightarrow$ `Altitude`

10. Set the altitude of the plane to `Altitude`
    $\mathtt{autoPilot}$:*setAltitude*(`Altitude`) $\rightarrow$ `Boolean`

11. Given the current `Altitude` of the plane calculate the new altitude of the plane
    $\mathtt{autoPilot}$:*calculateNewAltitude*(`Altitude`) $\rightarrow$ `Altitude`

12. Get the current `Velocity` of the plane
    $\mathtt{autoPilot}$:*velocity*() $\rightarrow$ `Velocity`

13. Given the current location, flight route and speed of the plane calculate the next location of the plane
    $\mathtt{autoPilot}$:*calculateLocation*(`Location`,`FlightRoute`,`Velocity`) $\rightarrow$ `3DPoint`

14. Return the list of passangers on the plane
    $\mathtt{autoPilot}$:*passengers*() $\rightarrow$ `ListOfPassengers`

**Actions**

1. Collect GPS Data from the sensor with `SensorId`.

   **Name:** *collectGPSData*(`SensorId`)

   **Pre:** **in**(`SensorId`, $\mathtt{autoPilot}$:*getSensorId*())

   **Del:** {}

   **Add:** **in**(`SensorId`, $\mathtt{autoPilot}$:*getSensorId*())&
   **in**(`GPSData`, $\mathtt{autoPilot}$:*readGPSData*(`SensorId`))

2. Adjust course of the plane

   **Name:** *adjust_course*(`No_go`,`FlightRoute`,`CurrentLocation`)

   **Pre:** **in**(`No_go`, $\mathtt{msgbox}$:*getVar*(`Msg.Id`, `"No_go"`))

   **Del:** **in**(`CurrentFlightRoute`, $\mathtt{autoPilot}$:*getFlightRoute*())

   **Add:** **in**(`CurrentLocation`, $\mathtt{autoPilot}$:*location*())&
   **in**(`CurrentFlightRoute`, $\mathtt{autoPilot}$:*getFlightRoute*())&
   **in**(`FlightRoute`, code call)

   where code call is
   $\mathtt{autoPilot}$:*adjustFlightRoute*(`CurrentFlightRoute`, `CurrentLocation`, `No_go`)

3. Maintain the flight route of the plane

   **Name:** *maintain_course*(`No_go`,`Flight_route`,`Current_location`)

   **Pre:** **in**(`No_go`, $\mathtt{msgbox}$:*getVar*(`Msg.Id`, `"No_go"`))&
   **in**(`CurrentLocation`, $\mathtt{autoPilot}$:*location*())&
   **in**(`CurrentRoute`, $\mathtt{autoPilot}$:*getFlightRoute*())&
   $= (CurrentRoute, allocatedFlightPath)$

   **Del:** {}

   **Add:** {}

4. Return the control of the plane to the pilot

   **Name:** $return\_control()$

   **Pre:** $\mathbf{in}(\mathtt{automated}, \mathrm{autoPilot}: pilotStatus(\mathtt{pilot\_message}))$

   **Del:** $\mathbf{in}(\mathtt{automated}, \mathrm{autoPilot}: pilotStatus(\mathtt{pilot\_message}))$

   **Add:** $\mathbf{in}(\mathtt{manual}, \mathrm{autoPilot}: pilotStatus(\mathtt{pilot\_message}))$

5. Create a flight plan for the plane

   **Name:** $create\_flight\_plan(\mathtt{No\_go}, \mathtt{Flight\_route}, \mathtt{Current\_location})$

   **Pre:** $\mathbf{in}(\mathtt{No\_go}, \mathrm{msgbox}: getVar(\mathtt{Msg.Id}, \mathtt{"No\_go"}))\,\&$
   $\mathbf{in}(\mathtt{CurrentLocation}, \mathrm{autoPilot}: location())$

   **Del:** $\mathbf{in}(\mathtt{CurrentRoute}, \mathrm{autoPilot}: getFlightRoute())$

   **Add:** $\mathbf{in}(\mathtt{No\_go}, \mathrm{msgbox}: getVar(\mathtt{Msg.Id}, \mathtt{"No\_go"}))\,\&$
   $\mathbf{in}(\mathtt{CurrentLocation}, \mathrm{autoPilot}: location())\,\&$
   $\mathbf{in}(\mathtt{FlightRoute}, \mathrm{autoPilot}: calculateFlightRoute(\mathtt{CurrentLocation}, \mathtt{No\_go}))$

6. Execute the flight plan of the plane

   **Name:** $execute\_flight\_plan(\mathtt{Flight\_route})$

   **Pre:** $\mathbf{in}(\mathtt{No\_go}, \mathrm{msgbox}: getVar(\mathtt{Msg.Id}, \mathtt{"No\_go"}))\,\&$
   $\mathbf{in}(\mathtt{CurrentLocation}, \mathrm{autoPilot}: location())$

   **Del:** $\{\}$

   **Add:** $\mathbf{in}(\mathtt{No\_go}, \mathrm{msgbox}: getVar(\mathtt{Msg.Id}, \mathtt{"No\_go"}))\,\&$
   $\mathbf{in}(\mathtt{CurrentLocation}, \mathrm{autoPilot}: location())\,\&$
   $\mathbf{in}(\mathtt{FlightRoute}, \mathrm{autoPilot}: calculateFlightRoute(\mathtt{CurrentLocation}, \mathtt{No\_go}))$

7. Adjust the altitude of the plane

   **Name:** $adjustAltitude(\mathtt{Altitude})$

   **Pre:** $\mathbf{in}(\mathtt{CurrentAltitude}, \mathrm{autoPilot}: getAltitude())$

   **Del:** $\mathbf{in}(\mathtt{CurrentAltitude}, \mathrm{autoPilot}: getAltitude())$

   **Add:** $\mathbf{in}(\mathtt{CurrentAltitude}, \mathrm{autoPilot}: getAltitude())\,\&$
   $\mathbf{in}(\mathtt{Altitude}, \mathrm{autoPilot}: calculateNewAltitude(\mathtt{CurrentAltitude}))$

8. Compute the current location of the plane

   **Name:** $compute\_currentLocation(\mathtt{Report})$

   **Pre:** $\mathbf{in}(\mathtt{Report}, \mathrm{msgbox}: getVar(\mathtt{Msg.Id}, \mathtt{"Report"}))$

   **Del:** $\mathbf{in}(\mathtt{OldLocation}, \mathrm{autoPilot}: location())$

   **Add:** $\mathbf{in}(\mathtt{OldLocation}, \mathrm{autoPilot}: location())\,\&$
   $\mathbf{in}(\mathtt{FlightRoute}, \mathrm{autoPilot}: getFlightRoute())\,\&$
   $\mathbf{in}(\mathtt{Velocity}, \mathrm{autoPilot}: velocity())\,\&$
   $\mathbf{in}(\mathtt{NewLocation}, \text{code call})$

   where code call is
   $\mathrm{autoPilot}: calculateLocation(\mathtt{OldLocation}, \mathtt{FlightRoute}, \mathtt{Velocity})$

### A.1.2 Satellite Agents

These agents monitor the the position of several planes simultaneously. Every $\Delta t$ units of time, each satellite agent broadcasts a report. Each satellite agent specifies where it believes the plane is at that point in time.

#### Code Calls

1. Broadcast GPS data
   $\mathtt{satellite}:\mathit{broadcastGPSData}() \rightarrow \mathtt{SatelliteReport}$

#### Actions

1. Broadcast satellite report

   **Name:** $\mathit{broadcast}(\mathtt{Report})$

   **Pre:** $\{\}$

   **Del:** $\{\}$

   **Add:** $\mathbf{in}(\mathtt{Report}, \mathtt{satellite}:\mathit{broadcastGPSData}())$

### A.1.3 GPS Agents

This agent takes reports from multiple satellite agents and merges them together. The gps agent then feeds the GPS-based location of the plane to the autoPilot agent, which consults the terrain agent below before taking corrective action.

#### Code Calls

1. Merge the two satellite reports into a single satellite report
   $\mathtt{gps}:\mathit{mergeGPSData}(\mathtt{Report1}, \mathtt{Report2}) \rightarrow \mathtt{SatelliteReport}$

2. Return the satellite id of the next satellite in the queue. All available satellites are maintained in a circular queue, each time this code call is invoked the next satellite in the queue is returned
   $\mathtt{gps}:\mathit{getNextSatellite}() \rightarrow \mathtt{String}$

3. Receive the GPS data sent from the Satellite and store it as a satellite report
   $\mathtt{gps}:\mathit{receiveFrom}(\mathtt{Satellite}) \rightarrow \mathtt{SatelliteReport}$

#### Actions

1. Collect GPS data from Satellite

   **Name:** $\mathit{collect\_data}(\mathtt{Satellite})$

   **Pre:** $\mathbf{in}(\mathtt{Satellite}, \mathtt{gps}:\mathit{getNextSatellite}())$

   **Del:** $\{\}$

   **Add:** $\mathbf{in}(\mathtt{Satellite}, \mathtt{gps}:\mathit{getNextSatellite}())\&$
   $\mathbf{in}(\mathtt{Report}, \mathtt{gps}:\mathit{recieveFrom}(\mathtt{Satellite}))$

2. Merge satellites reports of Satellite1 and Satellite2

**Name:** $merge\_data(\mathtt{Satellite1}, \mathtt{Satellite2})$

**Pre:** $\mathbf{in}(\mathtt{Satellite1}, \mathtt{gps}: getNextSatellite())\,\&$
$\mathbf{in}(\mathtt{Satellite2}, \mathtt{gps}: getNextSatellite())$

**Del:** $\{\}$

**Add:** $\mathbf{in}(\mathtt{Satellite1}, \mathtt{gps}: getNextSatellite())\,\&$
$\mathbf{in}(\mathtt{Satellite2}, \mathtt{gps}: getNextSatellite())\,\&$
$\mathbf{in}(\mathtt{Report1}, \mathtt{gps}: recieveFrom(\mathtt{Satellite1}))\,\&$
$\mathbf{in}(\mathtt{Report2}, \mathtt{gps}: recieveFrom(\mathtt{Satellite2}))\,\&$
$\mathbf{in}(\mathtt{Report}, \mathtt{gps}: mergeGPSData(\mathtt{Report1}, \mathtt{Report2}))$

### A.1.4 Terrain Agents

The terrain agent takes a coordinate in the globe, and generates a terrain map for the region. In this example, a special kind of terrain map is retrieved called a *Digital Terrain Elevation Data* (DTED) map. The terrain agent provides to the autoPilot agent a set of "no-go" areas. Using this set, the autoPilot agent can take corrective action, if necessary.

#### Code Calls

1. Given a 2D Point generate the terrain map for that point
   $\mathtt{terrain}: generateTerrainMap(\mathtt{Point}) \rightarrow \mathtt{TerrainMap}$
2. Determine the Nogo areas of a plane given its FlightRoute
   $\mathtt{terrain}: determineNogo(\mathtt{FlightRoute}) \rightarrow \mathtt{TerrainMap}$

#### Actions

1. Given the FlightRoute of a plan determine the set of No_go areas on that flight route

**Name:** $determineNogo(\mathtt{FlightRoute})$

**Pre:** $\mathbf{in}(\mathtt{FlightRoute}, \mathtt{msgbox}: getVar(\mathtt{Msg.Id}, \text{"FlightRoute"}))$

**Del:** $\{\}$

**Add:** $\mathbf{in}(\mathtt{FlightRoute}, \mathtt{msgbox}: getVar(\mathtt{Msg.Id}, \text{"FlightRoute"}))\,\&$
$\mathbf{in}(\mathtt{No\_go}, \mathtt{terrain}: determineNogo(\mathtt{FlightRoute}))$

### A.1.5 Action Constraints

$$\{compute\_currentLocation(\mathtt{Report}),\ adjust\_course(\mathtt{No\_go}, \mathtt{FlightRoute}, \mathtt{CurrentLocation})\} \hookleftarrow$$

This action constraint states that the actions compute_currentLocation and adjust_course may never be executed concurrently. This is because the adjust_course action requires the current location of the plane as input, and the compute_currentLocation action computes the required input.

$$\{collect\_data(\mathtt{Satellite}), merge\_data(\mathtt{Satellite1}, \mathtt{Satellite2})\} \hookleftarrow \mathtt{Satellite} = \mathtt{Satellite1}.$$
$$\{collect\_data(\mathtt{Satellite}), merge\_data(\mathtt{Satellite1}, \mathtt{Satellite2})\} \hookleftarrow \mathtt{Satellite} = \mathtt{Satellite2}.$$

These two action constraints state that the gps agent cannot concurrently execute the action merge_data and collect_data, if the satellite it is collecting data from is one of the satellites whose data it is merging.

## A.2 Agents in the STORE Example

### A.2.1 Credit Agents

This agent does nothing more sophisticated than providing access to a credit database. In the USA, many department stores issue their own credit cards, and as a consequence, they automatically have access to (at least some) credit data for many customers. The credit agent may in fact access a variety of databases, not just one.

#### Code Calls

1. Provide the credit information of the customer with social security number Ssn with the Type of detail
   $\mathtt{credit}:\mathit{provideCreditInfo}(\mathtt{Ssn},\mathtt{Type}) \rightarrow \mathtt{FinanceRecord}$
2. Check the credit records of the customer with name Name and social security number Ssn
   $\mathtt{credit}:\mathit{checkCredit}(\mathtt{Ssn},\mathtt{Name}) \rightarrow \mathtt{Real}$
3. Determine the list of customers to notify about their pending credit debts
   $\mathtt{credit}:\mathit{customer\_to\_be\_notified}() \rightarrow <\mathtt{String},\mathtt{String},\mathtt{Time}>$
4. Return the financial records of the customer with social security number Ssn
   $\mathtt{credit}:\mathit{getFinanceRec}(\mathtt{Ssn}) \rightarrow \mathtt{FinanceRecord}$
5. Send a notice to the customer Name with social security number Ssn
   $\mathtt{credit}:\mathit{sendNotice}(\mathtt{Ssn},\mathtt{Name}) \rightarrow \mathtt{Boolean}$

#### Actions

1. Terminate the credit of the customer with social security number Ssn

   **Name:** $\mathit{terminateCredit}(\mathtt{Ssn})$

   **Pre:** $\mathbf{in}(\mathtt{Ssn},\mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id},\text{"Ssn"}))$

   **Del:** $\mathbf{in}(\mathtt{FinanceRec},\mathtt{credit}:\mathit{getFinanceRec}(\mathtt{Ssn}))$

   **Add:** $\{\}$

2. Notify the customer with name Name and social security number Ssn

   **Name:** $\mathit{notifyCustomer}(\mathtt{Ssn},\mathtt{Name})$

   **Pre:** $\mathbf{in}(\mathtt{Ssn},\mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id},\text{"Ssn"}))\&$
   $\mathbf{in}(\mathtt{Name},\mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id},\text{"Name"}))$

   **Del:** $\mathbf{in}(\mathtt{Ssn},\mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id},\text{"Ssn"}))\&$
   $\mathbf{in}(\mathtt{Name},\mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id},\text{"Name"}))$

   **Add:** $\mathbf{in}(\mathtt{Status},\mathtt{credit}:\mathit{sendNotice}(\mathtt{Ssn},\mathtt{Name}))$

3. Provide credit report of the customer with name `Name` and social security number `Ssn`. Credit reports are prepared in high detail

   **Name:** $\textit{provideCreditReport}(\texttt{Ssn},\texttt{Name})$

   **Pre:** $\textbf{in}(\texttt{Ssn}, \mathrm{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Ssn"}))\,\&$
   $\textbf{in}(\texttt{Name}, \mathrm{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Name"}))$

   **Del:** $\textbf{in}(\texttt{Ssn}, \mathrm{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Ssn"}))\,\&$
   $\textbf{in}(\texttt{Name}, \mathrm{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Name"}))$

   **Add:** $\textbf{in}(\texttt{CreditReport}, \mathrm{credit}:\textit{provideCreditInfo}(\texttt{Ssn},\texttt{high}))$

4. Check the credit record of customer with social security number `Ssn` to see if he has an overdue payment

   **Name:** $\textit{checkCredit}(\texttt{Ssn})$

   **Pre:** $\textbf{in}(\texttt{Ssn}, \mathrm{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Ssn"}))\,\&$
   $\textbf{in}(\texttt{Name}, \mathrm{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Name"}))$

   **Del:** $\textbf{in}(\texttt{Ssn}, \mathrm{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Ssn"}))\,\&$
   $\textbf{in}(\texttt{Name}, \mathrm{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Name"}))$

   **Add:** $\textbf{in}(\texttt{Overdue}, \mathrm{credit}:\textit{checkCredit}(\texttt{Ssn},\texttt{Name}))$

### A.2.2 Profiling Agents

This agent takes as input the identity of a user, it then requests the credit agent for information on this user's credit history, and analyses the credit data. Credit information typically contains detailed information about an individual's spending habits. The profiling agent may then classify the user as a "high" spender, an "average" spender, or a "low" spender.

#### Code Calls

1. Classify users as high, medium or low spenders
   $\mathrm{profiling}:\textit{classifyUser}(\texttt{Ssn}) \rightarrow \texttt{UserProfile}$

2. Select all records of relation `Relation`
   $\mathrm{profiling}:\textit{all}(\texttt{Relation}) \rightarrow \texttt{SetsofRecords}$

3. List the social security numbers of a given `Category` of users
   $\mathrm{profiling}:\textit{listUsers}(\texttt{Category}) \rightarrow \texttt{ListOfStrings}$

#### Actions

1. Update the profiles of customers who are classified as high spenders

   **Name:** $\textit{update\_highProfile}(\texttt{Ssn},\texttt{Name},\texttt{Profile})$

   **Pre:** $\textbf{in}(\texttt{spender(high)}, \mathrm{profiling}:\textit{classifyUser}(\texttt{Ssn}))$

   **Del:** $\{\}$

   **Add:** $\textbf{in}(\langle\texttt{Ssn},\texttt{Name},\texttt{Profile}\rangle, \mathrm{profiling}:\textit{all}('\textit{highProfile}'))$

2. Update the profiles of customers who are medium spenders

**Name:** $\mathit{update\_mediumProfile}(\texttt{Ssn},\texttt{Name},\texttt{Profile})$

**Pre:** $\mathbf{in}(\texttt{spender}(\texttt{medium}),\mathrm{profiling}:\mathit{classifyUser}(\texttt{Ssn}))$

**Del:** $\{\}$

**Add:** $\mathbf{in}(\langle\texttt{Ssn},\texttt{Name},\texttt{Profile}\rangle,\mathrm{profiling}:\mathit{all}('\mathit{mediumProfile}'))$

3. Update the profiles of customers who are low spenders

**Name:** $\mathit{update\_lowProfile}(\texttt{Ssn},\texttt{Name},\texttt{Profile})$

**Pre:** $\mathbf{in}(\texttt{spender}(\texttt{low}),\mathrm{profiling}:\mathit{classifyUser}(\texttt{Ssn}))$

**Del:** $\{\}$

**Add:** $\mathbf{in}(\langle\texttt{Ssn},\texttt{Name},\texttt{Profile}\rangle,\mathrm{profiling}:\mathit{all}('\mathit{lowProfile}'))$

4. Classify the user with name `Name` and social security number `Ssn` as high, medium or low spender

**Name:** $\mathit{classify\_user}(\texttt{Ssn},\texttt{Name})$

**Pre:** $\mathbf{in}(\texttt{Ssn},\mathrm{msgbox}:\mathit{getVar}(\texttt{Msg.Id},\texttt{"}\mathrm{Ssn}\texttt{"}))\,\&$
$\mathbf{in}(\texttt{Name},\mathrm{msgbox}:\mathit{getVar}(\texttt{Msg.Id},\texttt{"}\mathrm{Name}\texttt{"}))$

**Del:** $\mathbf{in}(\texttt{Ssn},\mathrm{msgbox}:\mathit{getVar}(\texttt{Msg.Id},\texttt{"}\mathrm{Ssn}\texttt{"}))\,\&$
$\mathbf{in}(\texttt{Name},\mathrm{msgbox}:\mathit{getVar}(\texttt{Msg.Id},\texttt{"}\mathrm{Name}\texttt{"}))$

**Add:** $\mathbf{in}(\texttt{UserProfile},\mathrm{profiling}:\mathit{classifyUser}(\texttt{Ssn}))$

5. Inform the $\mathrm{saleNotification}$

**Name:** $\mathit{inform\_sale\_notifier}(\texttt{Ssn},\texttt{Name},\texttt{Profile})$

**Pre:** $\mathbf{in}(\texttt{Ssn},\mathrm{msgbox}:\mathit{getVar}(\texttt{Msg.Id},\texttt{"}\mathrm{Ssn}\texttt{"}))\,\&$
$\mathbf{in}(\texttt{Name},\mathrm{msgbox}:\mathit{getVar}(\texttt{Msg.Id},\texttt{"}\mathrm{Name}\texttt{"}))\,\&$
$\mathbf{in}(\texttt{Profile},\mathrm{msgbox}:\mathit{getVar}(\texttt{Msg.Id},\texttt{"}\mathrm{Profile}\texttt{"}))\,\&$
$=(\texttt{Profile},\texttt{riskProfile})$

**Del:** $\{\}$

**Add:** $\mathbf{in}(\texttt{Status},\text{code call})$
where code call is
$\mathrm{msgbox}:\mathit{sendMessage}(\mathrm{profiling},\mathrm{saleNotification},\texttt{"}\mathrm{Ssn,\ Name,\ riskprofile}\texttt{"})$

### A.2.3 ProductDB Agents

This agent provides access to one or more product databases reflecting the merchandise that the department store sells. Given a desired product description, this agent may be used to retrieve tuples associated with this product description.

#### Code Calls

1. Return product description
$\mathrm{productDB}:\mathit{provideDescription}(\texttt{ProductId})\ \to \texttt{ProductDescription}$

### A.2.4 Content-Determination Agents

This agent tries to determine what to show the user. It takes as input, the user's request, and the classification of the agent as determined by the $\mathsf{profiling}$ agent. It executes a query to the $\mathsf{productDB}$ agent, which provides it a set of tuples. It then uses the user classification provided by the $\mathsf{profiling}$ agent to filter these tuples. In addition, the $\mathsf{contentDetermin}$ agent may decide that when it presents the items selected to the user, it will run advertisements on the bottom of the screen, showing other items that "fit" this user's high-spending profile.

#### Code Calls

1. Prepare a presentation for a product
   $\mathsf{contentDetermin}:\mathit{preparePresentation}(\mathtt{ProductId},\mathtt{UserRequest},\mathtt{UserProfile})$

### A.2.5 Interface Agents

This agent takes the objects identified by the $\mathsf{contentDetermin}$ agent and weaves together a multimedia presentation.

### A.2.6 Sale-Notification Agents

This agent identifies a user's profile, determines which of the items going on sale "fits" the user's profile, and takes an appropriate action, such as mailing the user a list of enclosed items determined above.

#### Code Calls

1. Identify a user profile
   $\mathsf{saleNotification}:\mathit{identifyProfile}(\mathtt{Ssn},\mathtt{Name}) \rightarrow \langle\mathtt{UserProfile},\mathtt{UserAddress}\rangle$

2. Determine the items on sale that fit a user's profile
   $\mathsf{saleNotification}:\mathit{determineItems}(\mathtt{ListOfItemsOnSale},\mathtt{Profile}) \rightarrow \mathtt{ListOfItems}$

#### Actions

1. Mail brochures to the customer address `CustomerAddress`, containing the list of items `ListOfItems`

   **Name:** $\mathit{mailBrochure}(\mathtt{CustomerAddress},\mathtt{ListOfItems})$

   **Pre:** $\mathbf{in}(\langle\mathtt{Profile},\mathtt{CustomerAddress}\rangle,\mathsf{saleNotification}:\mathit{identifyProfile}(\mathtt{Ssn},\mathtt{Name}))$
   $\&\ \mathbf{in}(\mathtt{ListOfItems},\text{code call})$

   where code call is
   $\mathsf{saleNotification}:\mathit{determineItems}(\mathtt{ListOfItemsOnSale},\mathtt{Profile})$

   **Del:** $\{\}$

   **Add:** $\{\}$

### A.2.7 Action Constraints

$$\{\,\textit{update\_highProfile}(\texttt{Ssn1},\texttt{Name1},\texttt{profile}),\textit{update\_lowProfile}(\texttt{Ssn2},\texttt{Name2},\texttt{profile})\,\} \hookleftarrow$$
$$\textbf{in}(\texttt{spender}(\texttt{high}),\mathtt{profiling}:\textit{classifyUser}(\texttt{Ssn1}))\ \&$$
$$\texttt{Ssn1} = \texttt{Ssn2}\ \&\ \texttt{Name1} = \texttt{Name2}$$

$$\{\,\textit{update\_userProfile}(\texttt{Ssn1},\texttt{Name1},\texttt{Profile}),\textit{classify\_user}(\texttt{Ssn2},\texttt{Name2})\,\} \hookleftarrow$$
$$\texttt{Ssn1} = \texttt{Ssn2}\ \&\ \texttt{Name1} = \texttt{Name2}$$

The first action states that if the user is classified as a high spender, then the profiling agent cannot execute *update_highProfile* and *update_lowProfile* concurrently. In contrast, the second action constraint states that the profiling agent cannot classify a user profile if it is currently updating the profile of that user.

## A.3 Agents in the CHAIN Example

### A.3.1 Plant Agents

This agent monitors available inventory, and makes sure that inventories does not fall below some determined threshold values. Moreover, the plant agent determines the amount of stock needed, finds out the supplier, and places orders. 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 may consult a truck agent or an airplane agent. It also monitors the performance of suppliers.

#### Code Calls

1. Return the amount available of part Part_id in the inventory
   $\mathtt{plant}:\textit{monitorInventory}(\texttt{Part\_id}) \to \texttt{Integer}$
2. Update inventory to set the amount of Part_id to Amount
   $\mathtt{plant}:\textit{updateInventory}(\texttt{Part\_id},\texttt{Amount}) \to \texttt{Boolean}$
3. Choose a supplier of Part_id
   $\mathtt{plant}:\textit{chooseSupplier}(\texttt{Part\_id}) \to \texttt{String}$
4. Determine the amount of Part_id to order
   $\mathtt{plant}:\textit{determineAmount}(\texttt{Part\_id}) \to \texttt{Integer}$

#### Actions

1. Order Amount of part_id from Supplier when the amount in the inventory falls below the threshold lowInInventory

   **Name:** $\textit{orderPart}(\texttt{part\_id},\texttt{Amount},\texttt{Supplier})$

   **Pre:** $\textbf{in}(\texttt{AmountAvailable},\mathtt{plant}:\textit{monitorInventory}(\texttt{part\_id}))\,\&$
   $\texttt{AmountAvailable} \leq \texttt{lowInInventory}$

   **Del:** $\{\}$

   **Add:** $\textbf{in}(\texttt{Amount},\mathtt{plant}:\textit{determineAmount}(\texttt{part\_id}))\,\&$
   $\textbf{in}(\texttt{Supplier},\mathtt{plant}:\textit{chooseSupplier}(\texttt{part\_id}))$

### A.3.2 Supplier Agents

This agents basically monitors two databases, one for the committed stock the supplier has, and the other for the uncommitted stock the supplier has. When the `plant` agent requests a particular amount of some part, it consults these databases, serves the `plant` agent if it has the available stock, and updates its databases.

#### Code Calls

1. Monitor the stock of `Part_id`, and return either `amount_available` if there is `Amount` of `Part_id`, or `amount_not_available` if there is not `Amount` of `Part_id` available
   `supplier`:*monitorStock*(`Amount`, `Part_id`) $\rightarrow$ `StatusString`
2. Ship the `Amount` of `Part_id` from `Src` to `Dest` by using `Method`
   `supplier`:*shipFreight*(`Amount`, `Part_id`, `Method`, `Src`, `Dest`) $\rightarrow$ `Boolean`
3. Return the constant `too_low_threshold`
   `supplier`:*too_low_threshold*(`Part_id`) $\rightarrow$ `Integer`
4. Return the constant `low_threshold`
   `supplier`:*low_threshold*(`Part_id`) $\rightarrow$ `Integer`
5. Return the product status of part `Part_id`
   `supplier`:*productStatus*(`Part_id`) $\rightarrow$ `ProductStatusString`
6. Determine the amount of part `Part_id` to order
   `supplier`:*determineAmount*(`Part_id`) $\rightarrow$ `Integer`
7. Place an order for part `Part_id`
   `supplier`:*placeOrder*(`Part_id`) $\rightarrow$ `Boolean`
8. Place an order for part `Part_id` by fax
   `supplier`:*placeOrderByFax*(`Part_id`) $\rightarrow$ `Boolean`
9. Handle `Amount` of shipment of part `Part_id` to company `Company`
   `supplier`:*handleShipment*(`Company`, `Part_id`, `Amount`) $\rightarrow$ `Boolean`

#### Actions

1. Respond to part requests of other agents

   **Name:** *respond_request*(`Part_id`, `Amount`, `Company`)

   **Pre:** **in**(`Part_id`, `msgbox`:*getVar*(`Msg.Id`, "Part_id"))&
   **in**(`Amount`, `msgbox`:*getVar*(`Msg.Id`, "Amount"))&
   **in**(`Company`, `msgbox`:*getVar*(`Msg.Id`, "Company"))

   **Del:** **in**(`Part_id`, `msgbox`:*getVar*(`Msg.Id`, "Part_id"))&
   **in**(`Amount`, `msgbox`:*getVar*(`Msg.Id`, "Amount"))&
   **in**(`Company`, `msgbox`:*getVar*(`Msg.Id`, "Company"))

   **Add:** **in**(`varstatus`, `supplier`:*productStatus*(`Part_id`))

2. Update the stock database

**Name:** $\textit{update\_stockDB}(\texttt{Part\_id}, \texttt{Amount}, \texttt{Company})$

**Pre:** $\mathbf{in}(\texttt{Part\_id}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Part\_id"}))\,\&$
$\mathbf{in}(\texttt{Amount}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Amount"}))\,\&$
$\mathbf{in}(\texttt{Company}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Company"}))\,\&$
$\mathbf{in}(\texttt{X}, \mathtt{supplier}:\textit{select}('\textit{uncommitted}', \texttt{id}, =, \texttt{Part\_id}))\,\&$
$\texttt{X.amount} > \texttt{Amount}$

**Del:** $\mathbf{in}(\texttt{X}, \mathtt{supplier}:\textit{select}('\textit{uncommitted}', \texttt{id}, =, \texttt{Part\_id}))\,\&$
$\mathbf{in}(\texttt{Y}, \mathtt{supplier}:\textit{select}('\textit{committed}', \texttt{id}, =, \texttt{Part\_id}))$

**Add:** $\mathbf{in}(\langle \texttt{part\_id}, \texttt{X.amount} - \texttt{Amount}\rangle, \mathtt{supplier}:\textit{select}('\textit{uncommitted}', \texttt{id}, =, \texttt{Part\_id}))\,\&$
$\mathbf{in}(\langle \texttt{part\_id}, \texttt{Y.amount} + \texttt{Amount}\rangle, \mathtt{supplier}:\textit{select}('\textit{committed}', \texttt{id}, =, \texttt{Part\_id}))$

3. Order Amount_to_order units of part Part_id

**Name:** $\textit{order\_part}(\texttt{Part\_id}, \texttt{Amount\_to\_order})$

**Pre:** $\mathbf{in}(\texttt{Part\_id}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Part\_id"}))\,\&$
$\mathbf{in}(\texttt{supplies\_low}, \mathtt{supplier}:\textit{low\_threshold}(\texttt{Part\_id}))\,\&$
$\mathbf{in}(\texttt{amount\_not\_available}, \mathtt{supplier}:\textit{monitorStock}(\texttt{supplies\_low}, \texttt{Part\_id}))$

**Del:** $\{\}$

**Add:** $\mathbf{in}(\texttt{Amount\_to\_order}, \mathtt{supplier}:\textit{determineAmount}(\texttt{Part\_id}))\,\&$
$\mathbf{in}(\texttt{Status}, \mathtt{supplier}:\textit{placeOrder}(\texttt{Part\_id}))$

4. Order Amount_to_order units of part Part_id by fax

**Name:** $\textit{fax\_order}(\texttt{Company}, \texttt{Part\_id}, \texttt{Amount})$

**Pre:** $\mathbf{in}(\texttt{Part\_id}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Part\_id"}))\,\&$
$\mathbf{in}(\texttt{supplies\_low}, \mathtt{supplier}:\textit{low\_threshold}(\texttt{Part\_id}))\,\&$
$\mathbf{in}(\texttt{amount\_not\_available}, \mathtt{supplier}:\textit{monitorStock}(\texttt{supplies\_low}, \texttt{Part}\,_{\texttt{i}}\texttt{d}))$

**Del:** $\{\}$

**Add:** $\mathbf{in}(\texttt{Amount\_to\_order}, \mathtt{supplier}:\textit{determineAmount}(\texttt{Part\_id}))\,\&$
$\mathbf{in}(\texttt{Status}, \mathtt{supplier}:\textit{placeOrderByFax}(\texttt{Part\_id}))$

5. Ship Amount units of part Part_id to Company

**Name:** $\textit{shipped}(\texttt{Company}, \texttt{Part\_id}, \texttt{Amount})$

**Pre:** $\mathbf{in}(\texttt{Part\_id}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Part\_id"}))\,\&$
$\mathbf{in}(\texttt{Amount}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Amount"}))\,\&$
$\mathbf{in}(\texttt{Company}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Company"}))$

**Del:** $\mathbf{in}(\texttt{Part\_id}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Part\_id"}))\,\&$
$\mathbf{in}(\texttt{Amount}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Amount"}))\,\&$
$\mathbf{in}(\texttt{Company}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, \texttt{"Company"}))$

**Add:** $\mathbf{in}(\texttt{Status}, \mathtt{supplier}:\textit{handleShipment}(\texttt{Company}, \texttt{Part\_id}, \texttt{Amount}))$

### A.3.3 Shipping Agents

This agents coordinates the shipping of parts by consulting to truck and airplane agents.

**Code Calls**

1. Prepare shipping schedule
   $\mathtt{shipping}:\mathit{prepareSchedule}(\mathtt{Part\_id}, \mathtt{Amount}, \mathtt{Src}, \mathtt{Dest}) \rightarrow \mathtt{Schedule}$

2. Determine a truck to ship `Amount` of $\mathtt{Part_id}$ from `Src` to `Dest`
   $\mathtt{shipping}:\mathit{determineTruck}(\mathtt{Part\_id}, \mathtt{Amount}, \mathtt{Src}, \mathtt{Dest}) \rightarrow \mathtt{String}$

3. Determine an airplane to ship `Amount` of $\mathtt{Part_id}$ from `Src` to `Dest`
   $\mathtt{shipping}:\mathit{determineAirplane}(\mathtt{Part\_id}, \mathtt{Amount}, \mathtt{Src}, \mathtt{Dest}) \rightarrow \mathtt{String}$

**Action**

1. Find a truck to ship `Amount` of $\mathtt{Part_id}$ from `Src` to `Dest`

   **Name:** $\mathit{findTruck}(\mathtt{Part\_id}, \mathtt{Amount}, \mathtt{Src}, \mathtt{Dest})$

   **Pre:** $\mathbf{in}(\mathtt{Part\_id}, \mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id}, \text{"Part\_id"}))\,\&$
   $\mathbf{in}(\mathtt{Amount}, \mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id}, \text{"Amount"}))\,\&$
   $\mathbf{in}(\mathtt{Src}, \mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id}, \text{"Src"}))\,\&$
   $\mathbf{in}(\mathtt{Dest}, \mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id}, \text{"Dest"}))$

   **Del:** $\{\}$

   **Add:** $\mathbf{in}(\mathtt{TruckId}, \mathtt{shipping}:\mathit{determineTruck}(\mathtt{Part\_id}, \mathtt{Amount}, \mathtt{Src}, \mathtt{Dest}))$

2. Find an airplane to ship `Amount` of `Part_id` from `Src` to `Dest`

   **Name:** $\mathit{findAirplane}(\mathtt{Part\_id}, \mathtt{Amount}, \mathtt{Src}, \mathtt{Dest})$

   **Pre:** $\mathbf{in}(\mathtt{Part\_id}, \mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id}, \text{"Part\_id"}))\,\&$
   $\mathbf{in}(\mathtt{Amount}, \mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id}, \text{"Amount"}))\,\&$
   $\mathbf{in}(\mathtt{Src}, \mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id}, \text{"Src"}))\,\&$
   $\mathbf{in}(\mathtt{Dest}, \mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id}, \text{"Dest"}))$

   **Del:** $\{\}$

   **Add:** $\mathbf{in}(\mathtt{PlaneId}, \mathtt{shipping}:\mathit{determineAirplane}(\mathtt{Part\_id}, \mathtt{Amount}, \mathtt{Src}, \mathtt{Dest}))$

### A.3.4 Truck Agents

This agent provides and manages truck schedules using routing algorithms.

**Code Calls**

1. Return the current location of the truck
   $\mathtt{truck}:\mathit{location}() \rightarrow \mathtt{2DPoint}$

2. Given a `highway` and the current location of the truck calculate the destination of the truck
   $\mathtt{truck}:\mathit{calculateDestination}(\mathtt{From}, \mathtt{Highway})$

### Actions

1. Drive from location From to location To on highway Highway

   **Name:** $\mathit{drive}(\mathtt{From}, \mathtt{To}, \mathtt{highway})$

   **Pre:** $\mathbf{in}(\mathtt{From}, \mathtt{truck}: \mathit{location}())$

   **Del:** $\mathbf{in}(\mathtt{From}, \mathtt{truck}: \mathit{location}())$

   **Add:** $\mathbf{in}(\mathtt{From}, \mathtt{truck}: \mathit{location}())\&$
   $\mathbf{in}(\mathtt{To}, \mathtt{truck}: \mathit{calculateDestination}(\mathtt{From}, \mathtt{highway}))$

### A.3.5 Airplane Agents

This agent provides and manages airplane freight cargo.

#### Code Calls

1. Return the current location of the plane
   $\mathtt{airplane}: \mathit{location}() \rightarrow \mathtt{3DPoint}$

2. Return the current angle of the plane
   $\mathtt{airplane}: \mathit{angle}() \rightarrow \mathtt{Angle}$

3. Return the current speed of the plane
   $\mathtt{airplane}: \mathit{speed}() \rightarrow \mathtt{Speed}$

4. Given the current speed, current location and angle of the plane calculate the next position of the plane
   $\mathtt{airplane}: \mathit{calculateNextLocation}(\mathtt{Location}, \mathtt{Speed}, \mathtt{Angle}) \rightarrow \mathtt{3DPoint}$

#### Actions

1. Fly from location From to location To

   **Name:** $\mathit{fly}(\mathtt{From}, \mathtt{To})$

   **Pre:** $\mathbf{in}(\mathtt{From}, \mathtt{airplane}: \mathit{location}())$

   **Del:** $\mathbf{in}(\mathtt{From}, \mathtt{airplane}: \mathit{location}())$

   **Add:** $\mathbf{in}(\mathtt{From}, \mathtt{airplane}: \mathit{location}())\&$
   $\mathbf{in}(\mathtt{Speed}, \mathtt{airplane}: \mathit{speed}())\&$
   $\mathbf{in}(\mathtt{Angle}, \mathtt{airplane}: \mathit{angle}())\&$
   $\mathbf{in}(\mathtt{To}, \mathtt{airplane}: \mathit{calculateNextLocation}(\mathtt{From}, \mathtt{Speed}, \mathtt{Angle}))$

### A.3.6 Action Constraints

$$
\begin{aligned}
\{\ & \mathit{update\_stockDB}(\mathtt{Part\_id1}, \mathtt{Amount1}, \mathtt{Company1}),\\
& \mathit{update\_stockDB}(\mathtt{Part\_id2}, \mathtt{Amount2}, \mathtt{Company2})\ \} \hookleftarrow \\
& \hfill \mathtt{Part\_id1} = \mathtt{Part\_id2}\ \&\\
& \hfill \mathbf{in}(\mathtt{X}, \mathtt{supplier}: \mathit{select}('\mathit{uncommitted}', \mathtt{id}, =, \mathtt{Part\_id1}))\ \&\\
& \hfill \mathtt{X.amount} < \mathtt{Amount1} + \mathtt{Amount2}\ \&\\
& \hfill \mathtt{Company1} \neq \mathtt{Company2}.
\end{aligned}
$$

$$\{\, \textit{respond\_request}(\texttt{Part\_id1}, \texttt{Amount1}, \texttt{Company1}), \; \textit{respond\_request}(\texttt{Part\_id2}, \texttt{Amount2}, \texttt{Company2}) \,\} \hookleftarrow \texttt{Part\_id1} = \texttt{Part\_id2} \;\&\; \texttt{Company1} \neq \texttt{Company2}.$$

The first constraint states that if the two update_stockDB actions update the same `Part_id` and the total amount available is less than the sum of the requested amounts, then these actions cannot be concurrently executable. The second constraint states that if two companies request the same `Part_id`, then the `supplier` agent does not respond to them concurrently. That is, the `supplier` agent processes requests one at a time.

## A.4 Agents in the CFIT* Example

### A.4.1 Tank Agents

**Code Calls**

1. Drive forward at speed `Speed` (`0` to `Max` speed)
   $\texttt{tank}:\textit{goForward}(\texttt{Speed}) \rightarrow \texttt{Boolean}$

2. Drive backward at speed `Speed` (`0` to `Max` speed)
   $\texttt{tank}:\textit{goBackward}(\texttt{Speed})$

3. Turn left by `Degrees` degrees (`0` to `360`)
   $\texttt{tank}:\textit{turnLeft}(\texttt{Degrees})$

4. Turn right by `Degrees` degrees (`0` to `360`)
   $\texttt{tank}:\textit{turnRight}(\texttt{Degrees})$

5. Determine current position in 2D
   $\texttt{tank}:\textit{getPosition}() \rightarrow \texttt{2DPoint}$

6. Get current heading
   $\texttt{tank}:\textit{getHeading}() \rightarrow \texttt{Heading}$

7. Aim the gun at 3D point `Point`
   $\texttt{tank}:\textit{aim}(\texttt{Point}) \rightarrow \texttt{Boolean}$

8. Fire the gun using the current aim
   $\texttt{tank}:\textit{fire}() \rightarrow \texttt{Boolean}$

9. Compute the distance between two 2D points
   $\texttt{tank}:\textit{computeDistance}(\texttt{X}, \texttt{Y}) \rightarrow \texttt{Distance}$

10. Retrieve the maximum range for the gun
    $\texttt{tank}:\textit{getMaxGunRange}() \rightarrow \texttt{Distance}$

11. Calculate the next position of the tank when driving with `Speed` from `CurrentLocation`
    $\texttt{tank}:\textit{calculateNextPosition}(\texttt{CurrentLocation}, \texttt{Speed}) \rightarrow \texttt{2DPoint}$

12. Find all vehicles within `Distance` units of traffic circle given by (`XCoord`,`YCoord`,varRadius)
    $\texttt{tank}:\textit{FindVehiclesInRange}(\texttt{Distance}, \texttt{XCoord}, \texttt{YCoord}, \texttt{varRadius}) \rightarrow \texttt{ListOfVehicles}$

**Actions**

1. Drive from to 2D point $\mathtt{From}$ to 2D point $\mathtt{To}$ at speed $\mathtt{Speed}$

   **Name:** $\mathit{drive}(\mathtt{From},\mathtt{To},\mathtt{Speed})$

   **Pre:** $\mathbf{in}(\mathtt{From},\mathtt{tank}:\mathit{getPosition}())$

   **Del:** $\mathbf{in}(\mathtt{From},\mathtt{tank}:\mathit{getPosition}())$

   **Add:** $\mathbf{in}(\mathtt{CurrentLocation},\mathtt{tank}:\mathit{getPosition}())\&$
   $\mathbf{in}(\mathtt{To},\mathtt{tank}:\mathit{calculateNextPosition}(\mathtt{CurrentLocation},\mathtt{Speed}))$

2. Drive route $\mathtt{Route}$ given as a sequence of 2D points at speed $\mathtt{Speed}$

   **Name:** $\mathit{driveRoute}(\mathtt{Route},\mathtt{Speed})$

   **Pre:** $\mathbf{in}(\mathtt{Route(0).Position},\mathtt{tank}:\mathit{getPosition}())$

   **Del:** $\mathbf{in}(\mathtt{Route(0).Position},\mathtt{tank}:\mathit{getPosition}())$

   **Add:** $\mathbf{in}(\mathtt{Route(Route.Count).Position},\text{code call})$
   where code call is

   $$\mathtt{tank}:\mathit{calculateNextPosition}(\mathtt{Route(Route.Count}-1\mathtt{).Position},\mathtt{Speed}).$$

3. Attack vehicle at position $\mathtt{Position}$ from position $\mathtt{MyPosition}$

   **Name:** $\mathit{attack}(\mathtt{MyPosition},\mathtt{Position})$

   **Pre:** $\mathbf{in}(\mathtt{MyPosition},\mathtt{tank}:\mathit{getPosition}())\&$
   $\mathbf{in}(\mathtt{Distance},\mathtt{tank}:\mathit{computeDistance}(\mathtt{MyPosition},\mathtt{Position}))\&$
   $\mathbf{in}(\mathtt{maxRange},\mathtt{tank}:\mathit{getMaxGunRange}())\&\mathtt{Distance}<\mathtt{maxRange}$

   **Del:** $\{\}$

   **Add:** $\{\}$

### A.4.2 Terrain Route Planning Agent

**Code Calls**

1. Sets current map to $\mathtt{Map}$
   $\mathtt{route}:\mathit{useMap}(\mathtt{Map})\ \to \mathtt{Boolean}$

2. Compute a route plan on the current map for a vehicle of type $\mathtt{VehicleType}$ from $\mathtt{SourcePoint}$ to $\mathtt{DestinationPoint}$ given in 2D. Returns a route plan as a sequence of points in plane.
   $\mathtt{route}:\mathit{getPlan}(\mathtt{SourcePoint},\mathtt{DestinationPoint},\mathtt{VehicleType})$
   $\to \mathtt{SequenceOf2DPoints}$

3. Given $\mathtt{SourcePoint}$ and $\mathtt{DestinationPoint}$ on the current map, determine the likely routes of a vehicle of type $\mathtt{VehicleType}$ whose initial route segment is $\mathtt{Route}$, given as a sequence of points in the plane It returns a sequence of route-probability pairs.
   $\mathtt{route}:\mathit{groundPlan}(\mathtt{SourcePoint},\mathtt{DestinationPoint},\mathtt{VehicleType},\mathtt{Route})$
   $\to (\mathtt{Route},\mathtt{Probability})$

4. Compute a flight plan on the current map from $\mathtt{SourcePoint}$ to $\mathtt{DestinationPoint}$ given in 3D. Returns a flight plan as a sequence of points in space
   $\mathtt{route}:\mathit{flightPlan}(\mathtt{SourcePoint},\mathtt{DestinationPoint})\ \to \mathtt{SequenceOf3DPoints}$

5. Determines whether two points are visible from each other on the given map. For example if a hill lies between the two points, they are not visible from each other. This is useful to determine whether an agent can see another agent or whether an agent can fire upon another agent.
   $\mathtt{route}:\mathit{visible}(\mathtt{Map},\mathtt{Point1},\mathtt{Point2}) \rightarrow \mathtt{Boolean}$

### Actions

1. Compute a route plan on map `Map` for a vehicle of type `VehicleType` from `SourcePoint` to `DestinationPoint` given in 2D.

   **Name:** $\mathit{planRoute}(\mathtt{Map},\mathtt{SourcePoint},\mathtt{DestinationPoint},\mathtt{VehicleType})$

   **Pre:** $\mathtt{SourcePoint} \neq \mathtt{DestinationPoint}$

   **Del:** $\{\}$

   **Add:** $\mathbf{in}(\mathbf{true},\mathtt{route}:\mathit{useMap}(\mathtt{Map}))\,\&$
   $\mathbf{in}(\mathtt{Plan},\mathtt{route}:\mathit{getPlan}(\mathtt{SourcePoint},\mathtt{DestinationPoint},\mathtt{VehicleType}))$

2. Given `SourcePoint` and `DestinationPoint` on map `Map` determine the likely routes of a vehicle of type `VehicleType` whose initial route segment is `Route`, given as a sequence of points in the plane

   **Name:** $\mathit{evaluateGroundPlan}(\mathtt{Map},\mathtt{SourcePoint},\mathtt{DestinationPoint},\mathtt{VehicleType},\mathtt{Route})$

   **Pre:** $\mathtt{SourcePoint} \neq \mathtt{DestinationPoint}$

   **Del:** $\{\}$

   **Add:** $\mathbf{in}(\mathbf{true},\mathtt{route}:\mathit{useMap}(\mathtt{Map}))\,\&$
   $\mathbf{in}(\mathtt{RP},\mathtt{route}:\mathit{groundPlan}(\mathtt{SourcePoint},\mathtt{DestinationPoint},\mathtt{VehicleType},\mathtt{Route}))$

3. Compute a flight plan on map `Map` from `SourcePoint` to `DestinationPoint` given in 3D.

   **Name:** $\mathit{planFlight}(\mathtt{Map},\mathtt{SourcePoint},\mathtt{DestinationPoint})$

   **Pre:** $\mathtt{SourcePoint} \neq \mathtt{DestinationPoint}$

   **Del:** $\{\}$

   **Add:** $\mathbf{in}(\mathbf{true},\mathtt{route}:\mathit{useMap}(\mathtt{Map}))\,\&$
   $\mathbf{in}(\mathtt{Plan},\mathtt{route}:\mathit{flightPlan}(\mathtt{SourcePoint},\mathtt{DestinationPoint}))$

## A.4.3 Tracking Agent

This agent continuously scans the area for enemy vehicles. It maintains a list of enemy vehicles, assigning each an agent id. It tries to determine the vehicle type for each enemy vehicle. When it detects a new vehicle, it adds it to its list, together with its position. Since the `tracking` agent only keeps track of enemy vehicles which are on the ground, the position is in the plane. This could be for example an *AWACS* plane.

### Code Calls

1. Get position for agent with id `AgentId`
   $\mathtt{tracking}:\mathit{getPosition}(\mathtt{AgentId}) \rightarrow \mathtt{2DPoint}$

2. Get the type of agent for agent with id `AgentId`. It returns the most likely vehicle type together with the probability
   tracking:*getTypeOfAgent*(`AgentId`) $\rightarrow$ (`VehicleType`,`Probability`)

3. Return the list of all agents being tracked
   tracking:*getListOfAgents*() $\rightarrow$ `ListOfAgentIds`

4. Find the list of agents in the given `Image`
   tracking:*findobjects*(`Image`) $\rightarrow$ `ListOfAgentIds`

5. Return the list of neutralized agents
   tracking:*getListOfNeutralizedAgents*() $\rightarrow$ `ListOfAgentIds`

6. Return the marking information in the given `Image`
   tracking:*marking*(`Image`) $\rightarrow$ `MarkingInformation`

7. Return the turrent information in the given `Image`
   tracking:*turrent*(`Image`) $\rightarrow$ `TurrentInformation`

### A.4.4 Coordination Agent

**Code Calls**

1. Determine whether a vehicle of type `VehicleType1` at position `Position1` can attack a vehicle of type `VehicleType2` at position `Position2`. For example a tank is not able to attack a fighter plane unless it is on the ground.
   coord:*canBeAttackedNow*((`VehicleType1`,`Position1`,`VehicleType2`,`Position2`))
   $\rightarrow$ `Boolean`

2. Given an agent id for an enemy vehicle, determine the best position, time and route for an attack to be successful. Also return the estimated probability of success
   coord:*findAttackTimeAndPosition*(`AgentId`) $\rightarrow$ (`Position`,`Time`,`Route`,`Probability`)

3. Given a set of ids for friendly agents, compute a plan for a coordinated attack against the enemy agent with id `EnemyId`. The friendly agents participating in the coordinated attack are taken from the set `SetOfAgentIds`
   coord:*coordinatedAttack*((`SetOfAgentIds`,`EnemyId`)) $\rightarrow$ `AttackPlan`

**Actions**

1. Given a set of ids for friendly agents, compute a plan for a coordinated attack against the enemy agent with id `EnemyId`. The friendly agents participating in the coordinated attack are taken from the set `SetOfAgentIds`.

   **Name:** *attack*(`SetOfAgentIds`,`EnemyId`)

   **Pre:** `SetOfAgentIds` $\neq \emptyset$

   **Del:** $\{\}$

   **Add:** **in**(`AP`, coord:*coordinatedAttack*((`SetOfAgentIds`,`EnemyId`)))

### A.4.5 Helicopter Agents

#### Code Calls

1. Change flying altitude to `Altitude` (`0` to `Maximum` altitude)
   `heli`:*setAltitude*(`Altitude`) $\rightarrow$ `Boolean`

2. Get current altitude
   `heli`:*getAltitude*() $\rightarrow$ `Altitude`

3. Change flying speed to `Speed` (`0` to `Maximum` speed)
   `heli`:*setSpeed*(`Speed`) $\rightarrow$ `Boolean`

4. Get current speed
   `heli`:*getSpeed*() $\rightarrow$ `Speed`

5. Change flying heading to `Heading` (`0` to `360`)
   `heli`:*setHeading*(`Heading`) $\rightarrow$ `Boolean`

6. Get current heading
   `heli`:*getHeading*() $\rightarrow$ `Heading`

7. Aim the gun at the 3D point given by `Position`
   `heli`:*aim*(`Position`) $\rightarrow$ `Boolean`

8. Fire the gun using the current aim
   `heli`:*fire*() $\rightarrow$ `Boolean`

9. Determine the current position in space
   `heli`:*getPosition*() $\rightarrow$ `3DPoint`

10. Compute heading to fly from 2D point `Src` to 2D point `Dst`
    `heli`:*computeHeading*(`Src`,`Dst`) $\rightarrow$ `Heading`)

11. Compute the distance between two 3D points
    `heli`:*computeDistance*(`X`,`Y`) $\rightarrow$ `Distance`

12. Retrieve the maximum range for the gun
    `heli`:*getMaxGunRange*() $\rightarrow$ `Distance`

13. Calculate the next position of the helicopter given its `CurrentPosition`, its `Speed` and flying `Angle`
    `heli`:*calculateNextPosition*(`CurrentPosition`,`Speed`) $\rightarrow$ `3DPoint`

#### Actions

1. *Fly*() from 3D point `From` to 3D point `To` at altitude `Altitude` and with speed `Speed`

   **Name:** *fly*(`From`,`To`,`Altitude`,`Speed`)

   **Pre:** **in**(`From`, `heli`:*getPosition*())

   **Del:** **in**(`From`, `heli`:*getPosition*())

   **Add:** **in**(`CurrentPosition`, `heli`:*getPosition*())&
   **in**(`To`, `heli`:*calculateNextPosition*(`CurrentPosition`,`Speed`))

2. *FlyRoute*$(\texttt{path})$ `Path` given as a sequence of quadruples consisting of: a 3D point, altitude, speed and angle

   **Name:** $\mathit{flyRoute}(\texttt{Path})$

   **Pre:** $\mathbf{in}(\texttt{Path}(0).\texttt{Position}, \mathsf{heli}{:}\mathit{getPosition}())$

   **Del:** $\mathbf{in}(\texttt{Path}(0).\texttt{Position}, \mathsf{heli}{:}\mathit{getPosition}())$

   **Add:** $\mathbf{in}(\texttt{Path}(\texttt{Path.Count}).\texttt{Position}, \text{code call})$, where code call is

   $$\mathsf{heli}{:}\mathit{calculateNextPosition}(\texttt{Path}(\texttt{Path.Count}-1).\texttt{Position}, \texttt{Speed})$$

3. Attack vehicle at position `Position` in space from position `MyPosition`

   **Name:** $\mathit{attack}(\texttt{MyPosition}, \texttt{Position})$

   **Pre:** $\mathbf{in}(\texttt{MyPosition}, \mathsf{heli}{:}\mathit{getPosition}())\&$
   $\mathbf{in}(\texttt{Distance}, \mathsf{heli}{:}\mathit{computeDistance}(\texttt{MyPosition}, \texttt{Position}))\&$
   $\mathbf{in}(\texttt{maxRange}, \mathsf{heli}{:}\mathit{getMaxGunRange}())\&\texttt{Distance} < \texttt{maxRange}$

   **Del:** $\{\}$

   **Add:** $\{\}$

### A.4.6 AutoPilot Agents

**Code Calls**

1. Return the current location of the plane.
   $\mathsf{autoPilot}{:}\mathit{location}() \rightarrow \texttt{3DPoint}$
2. Return the current status of the plane.
   $\mathsf{autoPilot}{:}\mathit{planeStatus}() \rightarrow \texttt{PlaneStatus}$
3. Return the current flight route of the plane $\mathsf{autoPilot}{:}\mathit{getFlightRoute}() \rightarrow \texttt{Path}$
4. Return the current `Velocity` of the plane $\mathsf{autoPilot}{:}\mathit{velocity}() \rightarrow \texttt{Velocity}$
5. Given the current location, flight route and speed of the plane calculate the next location of the plane
   $\mathsf{autoPilot}{:}\mathit{calculateLocation}(\texttt{Location}, \texttt{FlightRoute}, \texttt{Velocity}) \rightarrow \texttt{3DPoint}$
6. Return the current `Altitude` of the plan
   $\mathsf{autoPilot}{:}\mathit{getAltitude}() \rightarrow \texttt{Altitude}$
7. Set the altitude of the plane to `Altitude`
   $\mathsf{autoPilot}{:}\mathit{setAltitude}(\texttt{Altitude}) \rightarrow \texttt{Boolean}$
8. Detect the possible dangerous situations to warn the pilot
   $\mathsf{autoPilot}{:}\mathit{detectWarning}() \rightarrow \texttt{WarningType}$
9. Determine the specific cause and specific information of the warning type
   $\mathsf{autoPilot}{:}\mathit{determineSpecifics}(\texttt{WarningType}) \rightarrow \texttt{SpecificInformation}$
10. Send a warning signal to the pilot
    $\mathsf{autoPilot}{:}\mathit{warnPilot}(\texttt{WarningType}, \texttt{WarningSpecificInfo}) \rightarrow \texttt{SignalType}$
11. Send a warning signal to the base station
    $\mathsf{autoPilot}{:}\mathit{sendSignalToBase}(\texttt{WarningType}, \texttt{WarningSpecificInfo}) \rightarrow \texttt{SignalType}$

**Actions**

1. Compute the current location of the plane

   **Name:** $\mathit{compute\_currentLocation}(\mathtt{Report})$

   **Pre:** $\mathbf{in}(\mathtt{Report}, \mathrm{msgbox}{:}\mathit{getVar}(\mathtt{Msg.Id}, \texttt{"Report"}))$

   **Del:** $\mathbf{in}(\mathtt{OldLocation}, \mathrm{autoPilot}{:}\mathit{location}())$

   **Add:** $\mathbf{in}(\mathtt{OldLocation}, \mathrm{autoPilot}{:}\mathit{location}())\&$
   $\mathbf{in}(\mathtt{FlightRoute}, \mathrm{autoPilot}{:}\mathit{getFlightRoute}())\&$
   $\mathbf{in}(\mathtt{Velocity}, \mathrm{autoPilot}{:}\mathit{velocity}())\&$
   $\mathbf{in}(\mathtt{NewLocation}, \text{code call})$

   where code call is
   $\mathrm{autoPilot}{:}\mathit{calculateLocation}(\mathtt{OldLocation}, \mathtt{FlightRoute}, \mathtt{Velocity})$

2. Warn the pilot with `WarningType` and with specific information `WarningSpecificInfo`

   **Name:** $\mathit{warn\_pilot}(\mathtt{WarningType}, \mathtt{WarningSpecificInfo})$

   **Pre:** $\mathbf{in}(\mathtt{WarningType}, \mathrm{autoPilot}{:}\mathit{detectWarning}())\&$
   $\mathbf{in}(\mathtt{WarningSpecificInfo}, \mathrm{autoPilot}{:}\mathit{determineSpecifics}(\mathtt{WarningType}))$

   **Del:** $\{\}$

   **Add:** $\mathbf{in}(\mathtt{Status}, \mathrm{autoPilot}{:}\mathit{warnPilot}(\mathtt{WarningType}, \mathtt{WarningSpecificInfo}))$

3. Send a signal to the base station to give a warning of `WarningType` with specific information `WarningSpecificInfo`

   **Name:** $\mathit{signal\_base}(\mathtt{WarningType}, \mathtt{WarningSpecificInfo})$

   **Pre:** $\mathbf{in}(\mathtt{WarningType}, \mathrm{autoPilot}{:}\mathit{detectWarning}())\&$
   $\mathbf{in}(\mathtt{WarningSpecificInfo}, \mathrm{autoPilot}{:}\mathit{determineSpecifics}(\mathtt{WarningType}))$

   **Del:** $\{\}$

   **Add:** $\mathbf{in}(\mathtt{Status}, \mathrm{autoPilot}{:}\mathit{sendSignalToBase}(\mathtt{WarningType}, \mathtt{WarningSpecificInfo}))$

4. Decrease the altitude of the plane

   **Name:** $\mathit{decrease\_altitude}(\mathtt{newaltitude})$

   **Pre:** $\{\}$

   **Del:** $\mathbf{in}(\mathtt{OldAltitude}, \mathrm{autoPilot}{:}\mathit{getAltitude}())$

   **Add:** $\mathbf{in}(\mathtt{status}, \mathrm{autoPilot}{:}\mathit{setAltitude}(\mathtt{newaltitude}))$

**Action Constraints**

$\mathbf{in}(\mathtt{S}, \mathrm{heli}{:}\mathit{getSpeed}()) \hookleftarrow S < \mathtt{maxSpeed}$

$\mathbf{in}(\mathtt{A}, \mathrm{heli}{:}\mathit{getAltitude}()) \hookleftarrow A < \mathtt{maxAltitude}$

**Action Constraints**

$$\{\mathit{fly\_plane1}(\mathtt{X1},\mathtt{Y1},\mathtt{A1},\mathtt{S1}),\mathit{fly\_plane2}(\mathtt{X2},\mathtt{Y2},\mathtt{A2},\mathtt{S2})\} \hookleftarrow \mathtt{Y1} = \mathtt{Y2}$$
$$\{\mathit{attack}(\mathtt{MyPos},\mathtt{P})\} \hookleftarrow \mathbf{in}(\mathtt{P},\mathtt{heli1}:\mathit{getPosition}())\ \&\ \mathbf{in}(\mathtt{MyPos},\mathtt{heli}:\mathit{getPosition}())$$
$$\{\mathit{attack}(\mathtt{MyPos},\mathtt{P})\} \hookleftarrow \mathbf{in}(\mathtt{P},\mathtt{heli2}:\mathit{getPosition}())\ \&\ \mathbf{in}(\mathtt{MyPos},\mathtt{heli}:\mathit{getPosition}())$$
$$\{\mathit{attack}(\mathtt{MyPos},\mathtt{P})\} \hookleftarrow \mathbf{in}(\mathtt{P},\mathtt{heli3}:\mathit{getPosition}())\ \&\ \mathbf{in}(\mathtt{MyPos},\mathtt{heli}:\mathit{getPosition}())$$
$$\{\mathit{attack}(\mathtt{MyPos},\mathtt{P})\} \hookleftarrow \mathbf{in}(\mathtt{P},\mathtt{heli4}:\mathit{getPosition}())\ \&\ \mathbf{in}(\mathtt{MyPos},\mathtt{heli}:\mathit{getPosition}())$$

where $\mathtt{heli1}, \ldots, \mathtt{heli4}$ are the friendly agents of the agent in question.

# References


Abbadi, A. E., D. Skeen, and F. Cristian (1985). An Efficient, Fault-tolerant Protocol for Replicated Data Management. In *Proceedings of the 1985 SIGACT-SIGMOD Symposium on Principles of Data Base Systems*, Portland, Oregon, pp. 215–229.

Abiteboul, S., R. Hull, and V. Vianu (1995). *Foundations of Databases*. Addison Wesley.

Adali, S., K. S. Candan, S.-S. Chen, K. Erol, and V. S. Subrahmanian (1996). Advanced Video Information Systems:Data Structures and Query Processing. *Multimedia Systems 4*(4), 172–186.

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

Adali, S., et al. (1997). Web hermes user manual. `http://www.cs.umd.edu/projects/hermes/UserManual/index.html`.

Ahmed, R., et al. (1991, December). The Pegasus Heteregeneous Multidatabase System. *IEEE Computer 24*(12), 19–27.

Alferes, J. J. and L. M. Pereira (1996). Reasoning with Logic Programming. In *Springer-Verlag Lecture Notes in AI*, Volume 1111.

Allen, J. (1984). Towards a General Theory of Action and Time. *Artificial Intelligence 23*(2), 123–144.

Allen, J. F. and G. Ferguson (1994). Actions and Events in Interval Temporal Logic. *Journal of Logic and Computation 4*(5), 531–579.

Alur, R. and T. A. Henzinger (1992, June). Logics and Models of Real Time: A Survey. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg (Eds.), *Proceedings of Real-Time: Theory in Practice*, Volume 600 of *Lecture Notes in Computer Science*, pp. 74–106. Berlin, Germany: Springer-Verlag.

Apt, K. (1990). Logic Programming. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Volume B, Chapter 10, pp. 493–574. Elsevier Science Publishers B.V. (North-Holland).

Apt, K. and H. Blair (1988). Arithmetic Classification of Perfect Models of Stratified Programs. In R. Kowalski and K. Bouwen (Eds.), *Proceedings of the Fifth Joint International Conference and Symposium on Logic Programming (JICSLP-88)*, pp. 766–779. MIT Press.

Apt, K., H. Blair, and A. Walker (1988). Towards a Theory of Declarative Knowledge. In J. Minker (Ed.), *Foundations of Deductive Databases and Logic Programming*, pp. 89–148. Washington DC: Morgan Kaufmann.

Åquist, L. (1984). Deontic Logic. In D. Gabbay and F. Guenthner (Eds.), *Handbook of Philosophical Logic, Volume II*, Chapter II.11, pp. 605–714. D. Reidel Publishing Company.

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

Artale, A. and E. Franconi (1998). A Temporal Description Logic for Reasoning about Actions and Plans. *Journal of Artificial Intelligence Research 9*, 463–506.

Baker, A. B. and Y. Shoham (1995). Nonmonotonic Temporal reasoning. In D. Gabbay, C. Hogger, and J. Robinson (Eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press.

Baldoni, M., L. Giordano, A. Martelli, and V. Patti (1998b). A Modal Programming Language for Representing Complex Actions. manuscript.

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

Baldwin, J. F. (1987). Evidential Support Logic Programming. *Journal of Fuzzy Sets and Systems 24*, 1–26.

Baral, C. and M. Gelfond (1993, August). Representing Concurrent Actions in Extended Logic Programming. In R. Bajcsy (Ed.), *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, Chambery, France, pp. 866–871. Morgan Kaufman.

Baral, C. and M. Gelfond (1994). Logic Programming and Knowledge Representation. *The Journal of Logic Programming 19/20*, 73–148.

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

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

Bateman, J. A. (1990). Upper Modeling: organizing knowledge for natural language processing. In *5th. International Workshop on Natural Language Generation, 3-6 June 1990*, Pittsburgh, PA. Organized by Kathleen R. McKeown (Columbia University), Johanna D. Moore (University of Pittsburgh) and Sergei Nirenburg (Carnegie Mellon University).

Bayardo, R., et al. (1997). Infosleuth: Agent-based Semantic Integration of Information in Open and Dynamic Environments. In J. Peckham (Ed.), *Proceedings of ACM SIGMOD Conference on Management of Data*, Tucson, Arizona, pp. 195–206.

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

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

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

Benthem, J. v. (1991). *The logic of time*. Kluwer Academic Publishers.

Benthem, J. v. (1995). Temporal logic. In D. Gabbay, C. Hogger, and J. Robinson (Eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, pp. 241–350. Oxford University Press.

Benton, J. and V. S. Subrahmanian (1994). Using Hybrid Knowledge Bases for Missile Siting Problems. In I. C. Society (Ed.), *Proceedings of the Conference on Artificial Intelligence Applications*, pp. 141–148.

Bergadano, F., A. Puliafito, S. Riccobene, and G. Ruffo (1999, January). Java-based and secure learning agents for information retrieval in distributed systems. *INFORMATION SCIENCES 113*(1-2), 55–84.

Berkovits, S., J. Guttman, and V. Swarup (1998). Authentication for Mobile Agents. In G. Vigna (Ed.), *Mobile agents and security*, Volume 1419 of *Lecture Notes in Computer Science*, pp. 114–136. New York, NY: Springer-Verlag.

Berman, K. A., J. S. Schlipf, and J. V. Franco (1995, June). Computing the Well-Founded Semantics Faster. In A. Nerode, W. Marek, and M. Truszczynski (Eds.), *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Third International Conference*, Volume 928 of *Lecture Notes in Computer Science*, Berlin, Germany, pp. 113–126. Springer-Verlag.

Bertino, E., C. Bettini, E. Ferrari, and P. Samarati (1996). A Temporal Access Control Mechanism for Database Systems. *IEEE Transactions on Knowledge and Data Engineering 8*(1), 67–80.

Bertino, E., P. Samarati, and S. Jajodia (1993, November). Authorizations in relational database management systems. In *Proceedings of the 1st ACM Conference on Computer and Communication Security*, Fairfax, VA.

Bina, E. J., R. M. McCool, V. E. Jones, and M. Winslett (1994). Secure Access to Data over the Internet. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94)*, Austin, Texas, pp. 99–102. IEEE-CS Press.

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

Blakeley, J. (1996, June). Data Access for Masses through OLE DB. In H. V. Jagadish and I. S. Mumick (Eds.), *Proceedings of ACM SIGMOD Conference on Management of Data*, Montreal, Canada, pp. 161–172.

Blakeley, J. and M. Pizzo (1998, June). Microsoft Universal Data Access Platform. In L. M. Haas and A. Tiwary (Eds.), *Proceedings of ACM SIGMOD Conference on Management of Data*, Seattle, Washington, pp. 502–503.

Bonatti, P., S. Kraus, and V. S. Subrahmanian (1995, June). Foundations of Secure Deductive Databases. *IEEE Transactions on Knowledge and Data Engineering 7*(3), 406–422.

Bond, A. H. and L. Gasser (1988). An Analysis of Problems and Research in DAI. In A. H. Bond and L. Gasser (Eds.), *Readings in Distributed Artificial Intelligence*, pp. 3–35. San Mateo, California: Morgan Kaufmann.

Boole, G. (1854). *The Laws of Thought*. Macmillan, London.

Bowersox, D. J., D. J. Closs, and O. K. Helferich (1986). *Logistical Management: A Systems Integration of Physical Distribution, Manufacturing Support, and Materials Procurement*. New York: Macmillan.

Box, D. (1998, January). *Essential COM (The Addison Wesley Object Technology Series)*. Addison Wesley.

Boye, J. and J. Maluszynski (1995). Two Aspects of Directional Types. In *Proceedings of the 12th International Conference on Logic Programming*, Tokyo, Japan, pp. 747–761. MIT Press.

Brass, S. and J. Dix (1997). Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. *The Journal of Logic Programming 32*(3), 207–228. (Extended abstract appeared in: Characterizations of the Stable Semantics by Partial Evaluation *LPNMR, Proceedings of the Third International Conference, Kentucky*, pages 85–98, 1995. LNCS 928, Springer-Verlag).

Brass, S. and J. Dix (1998). Characterizations of the Disjunctive Well-founded Semantics: Confluent Calculi and Iterated GCWA. *Journal of Automated Reasoning 20*(1), 143–165. (Extended abstract appeared in: Characterizing D-WFS: Confluence and Iterated GCWA. *Logics in Artificial Intelligence, JELIA '96*, pages 268–283, 1996. Springer-Verlag, LNCS 1126.).

Brass, S. and J. Dix (1999). Semantics of (Disjunctive) Logic Programs Based on Partial Evaluation. *The Journal of Logic Programming 38*(3), 167–213. (Extended abstract appeared in: Disjunctive Semantics Based upon Partial and Bottom-Up Evaluation, *Proceedings of the 12th International Logic Programming Conference, Tokyo*, pages 199–213, 1995. MIT Press.).

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

Breibart, Y. and H. Korth (1997, May). Replication and Consistency: Being Lazy Helps Sometimes. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIG6ART Symposium on Principles of Database Systems*, Tuscon, Arizona, pp. 173–184.

Brewka, G. and J. Dix (1999). Knowledge Representation with Extended Logic Programs. In D. Gabbay and F. Guenthner (Eds.), *Handbook of Philosophical Logic, 2nd Edition, Volume 6, Methodologies*, Chapter 6. D. Reidel Publishing Company. Shortened version also appeared in Dix, Pereira, Przymusinski (Eds.), *Logic Programming and Knowledge Representation*, Springer-Verlag, LNAI 1471, pages 1–55, 1998.

Brewka, G., J. Dix, and K. Konolige (1997). *Nonmonotonic Reasoning – An Overview*. Number 73 in CSLI Lecture Notes. CSLI Publications, Stanford University.

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

Brooks, R. A. (1986). A robust layered control system for mobile robot. *IEEE Journal of Robotics and Automation 2*(1), 14–23.

Buneman, P., J. Ullman, L. Raschid, S. Abiteboul, A. Levy, D. Maier, X. Qian, R. Ramakrishnan, V. S. Subrahmanian, V. Tannen, and S. Zdonik (1996). Mediator Languages - A Proposal for a Standard. Technical report, DARPA's I3/POB working group.

Cadoli, M. and M. Schaerf (1995). Tractable Reasoning via Approximation. *Artificial Intelligence 74*(2), 249–310.

Campbell, A. E. and S. C. Shapiro (1998, January). Algorithms for Ontological Mediation. Technical Report 98-02, Department of Computer Science, SUNY Buffalo. ftp://ftp.cs.buffalo.edu/pub/tech-reports/98-02.ps.Z.

Campbell, R. and T. Qian (1998, December). Dynamic Agent-based Security Architecture for Mobile Computers. In *The Second International Conference on Parallel and Distributed Computing and Networks (PDCN'98)*, Australia.

Candan, K. S., B. Prabhakaran, and V. S. Subrahmanian (1996, November). CHIMP: A Framework for Supporting Multimedia Document Authoring and Presentation. In *Proceedings of ACM Multimedia Conference*, Boston, MA.

Carey, M. J., et al. (1995, March). Towards Heterogeneous Multimedia Information Systems: The Garlic Approach . In *Fifth International Workshop on Research Issues in Data Engineering - Distributed Object Management*, Taipei, Taiwan, pp. 203–214.

Castano, S., M. G. Fugini, G. Martella, and P. Samarati (1995). *Database Security*. Addison Wesley.

Cattell, R. G. G., et al. (Ed.) (1997). *The Object Database Standard: ODMG-93*. Morgan Kaufmann.

Chawathe, S., et al. (1994, October). The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proceedings of the 10th Meeting of the Information Processing Society of Japan*, Tokyo, Japan. Also available via anonymous FTP from host db.stanford.edu, file /pub/chawathe/1994/tsimmis-overview.ps.

Chellas, B. (1980). *Modal Logic*. Cambridge University Press.

Chen, Z.-Z. and S. Toda (1993). The Complexity of Selecting Maximal Solutions. In *Proceedings of the 8th IEEE Structure in Complexity Theory Conference*, San Diego, CA, pp. 313–325. IEEE Computer Society Press.

Chen, Z.-Z. and S. Toda (1995). The Complexity of Selecting Maximal Solutions. *Information and Computation 119*, 231–239.

Chess, D. M. (1996). Security in Agents Systems. `http://www.av.ibm.com/InsideTheLab/Bookshelf/ScientificPapers/`.

Chess, D. M. (1998). Security Issues in Mobile Code Systems. In G. Vigna (Ed.), *Mobile agents and security*, Volume 1419 of *Lecture Notes in Computer Science*, pp. 1–14. New York, NY: Springer-Verlag.

Chittaro, L. and A. Montanari (1998). Editorial: Temporal representation and reasoning. *Annals of Mathematics and Artificial Intelligence 22*, 1–4.

Cohen, P. and H. Levesque (1990a). Intention is Choice with Commitment. *Artificial Intelligence 42*, 263–310.

Cohen, P. R. and H. Levesque (1990b). Rational Interaction as the Basis for Communication. In P. R. Cohen, J. L. Morgan, and M. E. Pollack (Eds.), *Intentions in Communication*, pp. 221–256. Cambridge, MA: MIT Press.

Coradeschi, S. and L. Karlsson (1997). A Behavior-based Decision Mechanism for Agents Coordinating using Roles. In *International Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp. 100–105.

Cormen, T. H., C. E. Leiserson, and R. L. Rivest (1989). *Introduction to Algorithms*. McGraw-Hill.

Creamer, J., M. O. Stegman, and R. P. Signore (1995, February). *The ODBC Solution : Open Database Connectivity in Distributed Environments/Book and Disk*. McGraw-Hill.

Crosbie, M. and E. Spafford (1995, November). Applying genetic programming to intrusion detection. In *Proceedings of the AAAI 1995 Fall Symposium series*.

Dagan, I. (1998). Contextual Word Similarity. In R. Dale, H. Moisl, and H. Somers (Eds.), *A Handbook of Natural Language Processing*. New York: Marcel Dekker. to appear.

Damasio, C. V., W. Nejdl, and L. M. Pereira (1994). An Extended Logic Programming System for Revising Knowledge Bases. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR'94)*, Bonn, Germany, pp. 607–618. Morgan Kaufmann.

Dantsin, E., T. Eiter, G. Gottlob, and A. Voronkov (1997, June). Complexity and Expressive Power of Logic Programming. In *Proceedings of the Twelth IEEE International Conference on Computational Complexity (CCC '97)*, Ulm, Germany, pp. 82–101. IEEE Computer Society Press.

Date, C. J. (1995). *An Introduction to Database Systems* (sixth ed.). Addison Wesley.

Dayal, U. (1983). Processing Queries Over Generalization Hierarchies in a Multidatabase System. In M. Schkolnick and C. Thanos (Eds.), *Proceedings of the 9th International Conference on Very Large Data Bases*, Florence, Italy, pp. 342–353. VLDB Endowment / Morgan Kaufmann.

Dayal, U. and H.-Y. Hwang (1984). View Definition and Generalization for Database Integration in a Multidatabase System. *IEEE Transactions on Software Engineering 10*(6), 628–645.

Dean, T. and D. McDermott (1987). Temporal data base management. *Artificial Intelligence 32*(1), 1–55.

Dechter, R., I. Meiri, and J. Pearl (1991). Temporal Constraint Networks. *Artificial Intelligence 49*, 61–95.

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

Dekhtyar, A. and V. S. Subrahmanian (1997). Hybrid Probabilistic Logic Programs. In L. Naish (Ed.), *Proceedings of the 14th International Conference on Logic Programing*, Leuven, Belgium, pp. 391–405. MIT Press. Extended version accepted for publication in Journal of Logic Programming, `http://www.cs.umd.edu/TRs/authors/Alex\_Dekhtyar.html`.

Dignum, F. and R. Conte (1997). Intentional Agents and Goal Formation. In *International Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp. 219–231.

Dignum, F. and R. Kuiper (1997). Combining Deontic Logic and Temporal logic for Specification of Deadlines. In *Proceedings of Hawaii International Conference on System Sciences, HICSS-97*, Maui, Hawaii.

Dignum, F., H. Weigand, and E. Verharen (1996). Meeting the deadline: on the formal specification of temporal Deontic constraints. *Lecture Notes in Computer Science 1079*, 243–.

d'Inverno, M., D. Kinny, M. Luck, and M. Wooldridge (1997). A Formal Specification of dMARS. In *International Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp. 146–166.

Dix, J. (1995). Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In A. Fuhrmann and H. Rott (Eds.), *Logic, Action and Information. Proc. of the Konstanz Colloquium in Logic and Information (LogIn'92)*, pp. 241–329. DeGruyter.

Dix, J., U. Furbach, and I. Niemela (1999). Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations. In A. Voronkov and A. Robinson (Eds.), *Handbook of Automated Reasoning*. Elsevier Science Publishers. to appear.

Doan, A. (1996). Modeling Probabilistic Actions for Practical Decision-Theoretic Planning. In *Proceedings of the Third International Concerence on Artificial-Intelligence Planning Systems*, Edinburgh, Scotland, UK.

Dogac, A., et al. (1996a, February). A Multidatabase System Implementation on CORBA. In *Proceedings of the 6th International Workshop on Research Issues in Data Engineering - Interoperability of Nontraditional Database Systems(RIDE-NDS '96)*, New Orleans, Louisiana, pp. 2–11.

Dogac, A., et al. (1996b, June). METU Interoperable Database System. In H. V. Jagadish and I. S. Mumick (Eds.), *Proceedings of ACM SIGMOD Conference on Management of Data*, Montreal, Canada, pp. 552.

Donini, F. M., D. Nardi, and R. Rosati (1997). Ground Nonmonotonic Modal Logics. *Journal of Logic and Computation 7*(4), 523–548.

Dubois, D., J. Land, and H. Prade (1991, June). Towards Possibilistic Logic Programming. In *Proceedings of the Eigth International Conference on Logic Programming*, Paris, France, pp. 581–595. MIT Press.

Dubois, D., J. Lang, and H. Prade (1994). Automated Reasoning Using Possibilistic Logic: Semantics, Belief Revision, and Variable Certainty Weights. *IEEE Transactions on Knowledge and Data Engineering 6*(1), 64–71.

Dubois, D. and H. Prade (1988). Certainty and Uncertainty of Vague Knowledge and Generalized Dependencies in Fuzzy Databases. In *Proceedings of International Fuzzy Engineering Symposium*, Yokohama, Japan, pp. 239–249.

Dubois, D. and H. Prade (1989). Processing Fuzzy Temporal Knowledge. *IEEE Transactions on Systems, Man and Cybernetics 19*(4), 729–744.

Dubois, D. and H. Prade (1991). Epistemic Entrenchment and Possibilistic Logic. *Artificial Intelligence 50*(2), 223–239.

Dubois, D. and H. Prade (1995, August). Possibility Theory as a Basis for Qualitative Decision Theory. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95*, Montreal, Quebec, Canada, pp. 1924–1932. Morgan Kaufmann.

Durfee, E. H. (1988). *Coordination of Distributed Problem Solvers*. Boston: Kluwer Academic Publishers.

Eiter, T. and G. Gottlob (1992). Reasoning with Parsimonious and Moderately Grounded Expansions. *Fundamenta Informaticae 17*(1,2), 31–53.

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

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

Eiter, T., G. Gottlob, and H. Mannila (1997, September). Disjunctive Datalog. *ACM Transactions on Database Systems 22*(3), 364–417.

Eiter, T., V. Subrahmanian, and T. Rogers (1999, May). Heterogeneous Active Agents, III: Polynomially Implementable Agents. Technical Report INFSYS RR-1843-99-07, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria.

Eiter, T. and V. S. Subrahmanian (1999). Heterogeneous Active Agents, II: Algorithms and Complexity. *Artificial Intelligence 108*(1-2), 257–307.

Emden, M. v. (1986). Quantitative Deduction and its Fixpoint Theory. *Journal of Logic Programming 4*(1), 37–53.

Etzioni, O., N. Lesh, and R. Segal (1994). Building softbots for UNIX. In O. Etzioni (Ed.), *Software Agents—papers from the 1994 Spring Symposium*, pp. 9–16.

Etzioni, O. and D. S. Weld (1995, August). Intelligent Agents on the Internet: Fact, Fiction, and Forecast. *IEEE Expert 10*(4), 44–49.

Fagin, R. and J. Halpern (1989, August). Uncertainty, Belief and Probability. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence, IJCAI 89*, Detroit, MI, pp. 1161–1167. Morgan Kaufmann.

Fagin, R., J. Halpern, Y. Moses, and M. Vardi (1995). *Reasoning about Knowledge*. Cambridge, Massachusetts: MIT Press. 2nd printing.

Fagin, R., J. Y. Halpern, and N. Megiddo (1990, July/August). A logic for reasoning about probabilities. *Information and Computation 87*(1/2), 78–128.

Fagin, R. and M. Vardi (1986). Knowledge and Implicit Knowledge in a Distributed Environment. In *Proceedings of 1986 Conference on Theoretical Aspects of Reasoning about Knowledge*, pp. 187–206. Morgan Kaufmann.

Fankhauser, P., B. Finance, and W. Klas (1996). IRO-DB: Making Relational and Object-Oriented Database Systems Interoperable. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin (Eds.), *Proceedings of the 5th International Conference on Extending Database Technology*, Volume 1057 of *Lecture Notes in Computer Science*, Avignon, France, pp. 485–489. Springer-Verlag.

Farmer, W. M., J. D. Guttag, and V. Swarup (1996, September). Security for Mobile Agents: Authentification and State Appraisal. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo (Eds.), *Proceedings of the Fourth ESORICS*, Volume 1146 of *Lecture Notes in Computer Science*, pp. 118–130. Rome, Italy: Springer-Verlag.

Fenner, S., S. Homer, M. Ogihara, and A. Selman (1997). Oracles That Compute Values. *SIAM Journal on Computing 26*(4), 1043–1065.

Ferguson, I. A. (1992). Towards an Architecture for adaptive, rational, mobile agents. In E. Werner and Y. Demazeau (Eds.), *Decentralized Artificial Intelligence, Volume 3*, pp. 249–262. Germany: Elsevier Science Publishers.

Fiadeiro, J. and T. Maibaum (1991). Temporal Reasoning over Deontic Specifications. *Journal of Logic and Computation 1*(3), 357–395.

Finin, T., R. Fritzon, D. McKay, and R. McEntire (1994, July). KQML – A Language and protocol for Knowledge and Information Exchange. In *Proceedings of the 13th International Workshop on Distributed Artificial Intelligence*, Seatle, WA, pp. 126–136.

Finin, T., et al. (1993). Specification of the KQML Agent-Communication Language (Draft Version). The DARPA Knowledge Sharing Initiative External Interfaces Working Group.

Foltz, P. W. and S. T. Dumais (1992). Personalized information delivery: An analysis of filtering methods. In *Proceedings of ACM CHI Conference on Human Factors in Computing Systems – Posters and Short Talks*, Posters: Designing for Use, Monterey, CA.

Foner, L. N. (1993). What's an agent anyway? A Sociological Case Study. Technical Report Agents Memo 93-01, MIT, Media Laboratory.

Foner, L. N. (1996). A Security Architecture for Multi-Agent Matchmaking. In *Second International Conference on Multi-Agent Systems (ICMAS96)*, Japan.

Foner, L. N. (1997, February). Yenta: A Multi-Agent, Referral-Based Matchmaking System. In W. L. Johnson and B. Hayes-Roth (Eds.), *Proceedings of the 1st International Conference on Autonomous Agents*, New York, NY, pp. 301–307. ACM Press.

Frakes, W. B. and R. Baeza-Yates (Eds.) (1992). *Information Retrieval Data Structures and Algorithms*. Prentice-Hall.

Franklin, S. and A. Graesser (1997). Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In J. P. Muller, M. J. Wooldridge, and N. R. Jennings (Eds.), *Intelligent agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*. Springer-Verlag.

Fritzinger, S. and M. Mueller (1996). Java Security. `http://java.sun.com/docs/white/index.html`.

Garcia-Molina, H., et al. (1997). The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems 8*(2), 117–132.

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

Gasser, L. and T. Ishida (1991). A Dynamic Organizational Architecture For Adaptive Problem Solving. In *Proceedings of the 9th National Conference on Artificial Intelligence*, Anaheim, CA, pp. 185–190. AAAI Press/MIT Press.

Gelfond, M. and V. Lifschitz (1988). The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth International Conference and Symposium*, Cambridge, Massachusetts, pp. 1070–1080. MIT Press.

Gelfond, M. and V. Lifschitz (1991). Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing 9*, 365–385.

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

Genesereth, M. R. and R. E. Fikes (1992, June). Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford University. `http://www-ksl.stanford.edu/knowledge-sharing/papers/kif.ps`.

Genesereth, M. R. and S. P. Ketchpel (1994). Software Agents. *Communications of the ACM 37*(7), 49–53.

Genesereth, M. R. and N. J. Nilsson (1987). *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann.

Georgeff, M. and A. Lansky (1987). Reactive Reasoning and Planning. In *Proceedings of the Conference of the American Association of Artificial Intelligence*, Seattle, WA, pp. 677–682.

Giacomo, G. D., Y. Lesperance, and H. Levesque (1997). Reasoning about Concurrent Execution, Prioritized Interrupts, and Exogenous Actions in the Situation Calculus. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Nagoya, Japan.

Ginsberg, M. L. and D. E. Smith (1987). Reasoning About Action I: A Possible Worlds Approach. In F. Brown (Ed.), *Proceedings of the Workshop "The Frame Problem in Artificial Intelligence"*, pp. 233–258. Morgan Kaufmann. also in Artificial Intelligence, 35(165–195), 1988.

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

Gmytrasiewicz, P. and E. Durfee (1992). A Logic of Knowledge and Belief for Recursive Modeling. In *Proceedings of the 10th National Conference on Artificial Intelligence*, San Jose, CA, pp. 628–634. AAAI Press/MIT Press.

Gmytrasiewicz, P., E. Durfee, and D. Wehe. (1991). A Decision-Theoretic Approach to Coordinating Multiagent Interactions. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, pp. 62–68. Morgan Kaufmann.

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

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

Gottlob, G. (1992b, March). On the Power of Pure Beliefs or Embedding Default Logic into Standard Autoepistemic Logic. Technical Report CD-TR 92/34, Christian Doppler Laboratory for Expert Systems, TU Vienna. Extended Abstract in Proceedings IJCAI-93, Chambery, France, pp. 570–575.

Gottlob, G. (1995a). The Complexity of Propositional Default Reasoning Under the Stationary Fixed Point Semantics. *Information and Computation 121*(1), 81–92.

Gottlob, G. (1995b). Translating Default Logic into Standard Autoepistemic Logic. *Journal of the ACM 42*(4), 711–740.

Gottlob, G., N. Leone, and H. Veith (1995). Second-Order Logic and the Weak Exponential Hierarchies. In J. Wiedermann and P. Hajek (Eds.), *Proceedings of the 20th Conference on Mathematical Foundations of Computer Science (MFCS '95)*, Volume 969 of *Lecture Notes in Computer Science*, Prague, pp. 66–81. Full paper available as CD/TR 95/80, Christian Doppler Lab for Expert Systems, Information Systems Department, TU Wien.

Gray, J., P. Helland, P. O'Neil, and D. Shasha (1996). The dangers of replication and a solution. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Montreal, Quebec, pp. 173–182.

Gray, R., D. Kotz, G. Cybenko, and D. Rus (1998). D'Agents: Security in Multiple-language, Mobile-Agent System. In G. Vigna (Ed.), *Mobile agents and security*, Volume 1419 of *Lecture Notes in Computer Science*, pp. 154–187. New York, NY: Springer-Verlag.

Gruber, T. R. and G. R. Olsen (1994). An ontology for engineering mathematics. In J. Doyle, P. Torasso, and E. Sandewall (Eds.), *Fourth International Conference on Principles of Knowledge Representation and Reasoning*, Bonn, Germany. Morgan Kaufmann.

Guha, R. V. and D. B. Lenat (1994, July). Enabling Agents to Work Together. *Communications of the ACM, Special Issue on Intelligent Agents 37*(7), 126–142.

Guntzer, U., W. Kiessling, and H. Thone (1991, May). New Directions for Uncertainty Reasoning in Deductive Databases. In J. Clifford and R. King (Eds.), *Proceedings of ACM SIGMOD Conference on Management of Data*, Denver, Colorado, pp. 178–187.

Gupta, P. and E. T. Lin (1994, September). DataJoiner: A Practical Approach to Multi-Database Access. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, Austin, Texas. IEEE-CS Press.

Haddadi, A. (1995). Towards a Pragmatic Theory of Interactions. In *International Conference on Multi-Agent Systems*, pp. 133–139.

Haddawy, P. (1991). *Representing Plans under Uncertainty: A Logic of Time, Chance and Action*. Ph. D. thesis, University of Illinois. Technical Report UIUCDCS-R-91-1719.

Haddawy, P., A. Doan, and R. Goodwin (1996). Efficient Decision-Theoretic Planning: Techniques and Empirical Analysis. In *Proceedings of the Third International Concerence on Artificial-Intelligence Planning Systems*, Edinburgh, Scotland, UK.

Halpern, J. and Y. Moses (1992). A Guide to the Completeness and Complexity for Modal Logics of Knowledge and Belief. *Artificial Intelligence 54*, 319–380.

Halpern, J. and Y. Shoham (1991). A propositional modal interval logic. *Journal of the ACM 38*(4), 935–962.

Halpern, J. Y. and M. Tuttle (1992). Knowledge, Probability and Adversaries. Technical report, IBM. IBM Research Report.

Hansson, S. (1994). Review of Deontic Logic in Computer Science: Normative System Specification. *Bulletin of the Interest Group on Pure and Applied Logic (IGPL) 2*(2), 249–250.

Hayes-Roth, B. (1995). An Architecture for Adaptive intelligent systems. *Artificial Intelligence 72(1-2)*, 329–365.

He, Q., K. P. Sycara, and T. W. Finin (1998, May). Personal Security Agent: KQML-Based PKI. In K. P. Sycara and M. Wooldridge (Eds.), *Proceedings of the 2nd International Conference on Autonomous Agents (AGENTS-98)*, New York, pp. 377–384. ACM Press.

Heintze, N. and J. Tygar (1996, January). A model for secure protocols and their compositions. *IEEE Transactions on Software Engineering 22*(1), 16–30.

Hendler, J. and D. McDermott (1995). Planning: What it could be, An introduction to the special issue on Planning and Scheduling. *Artificial Intelligence 76*(1–2), 1–16.

Hendler, J., A. Tate, and M. Drummond (1990). Systems and techniques: AI planning. *AI Magazine 11*(2), 61–77.

Hindriks, K. V., F. S. de Boer, W. van der Hoek, and J. J. C. Meyer (1997). Formal Semantics of an Abstract Agent Programming Language. In *International Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp. 204–218.

Hohl, F. (1997). An Approach to Solve the Problem of Malicious Hosts in Mobile Agent Systems. `http://inf.informatik.uni-stuttgart.de:80/ipvr/vs/mitarbeiter/hohlfz.en%gl.html`.

Holler, E. (1981). *Distributed Systems-Architecture and Implementation: An Advanced Course*, Chapter Multiple Copy Update. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag.

Horstmann, C. S. and G. Cornell (1997). *Core Java 1.1 : Fundamentals (Sunsoft Press Java Series)*. Palo Alto, CA: Prentice-Hall.

Horty, J. F. (1996). Agency and obligation. *Synthese 108*, 269–307.

Hughes, M. (1998). Application and enterprise security with the JAVA$^{TM}$ 2 platform. `http://java.sun.com/events/jbe/98/features/security.html`.

Ishizaki, S. (1997). Multiagent Model of Dynamic Design: Visualization as an Emergent Behavior of Active Design Agents. In M. Huhns and M. Singh (Eds.), *Readings in Agents*, pp. 172–179. Morgan Kaufmann.

Jajodia, S. and R. Sandhu (1991, May). Toward a Multilevel Relational Data Model. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Denver, Colorado.

Jenner, B. and J. Toran (1997, January). The Complexity of Obtaining Solutions for Problems in NP and NL. In L. Hemaspaandra and A. Selman (Eds.), *Complexity Theory Retrospective II*, pp. 155–178. Springer-Verlag.

Johnson, D. S. (1990). A Catalog of Complexity Classes. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Volume A, Chapter 2, pp. 67–161. Elsevier Science Publishers.

Kaelbling, L. P., M. L. Littman, and A. R. Cassandra (1998). Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence 101*, 99–134.

Kanger, S. (1972). Law and Logic. *Theoria 38*(3), 105–132.

Kiessling, W., H. Thone, and U. Guntzer (1992, March). Database Support for Problematic Knowledge. In *Proceedings of the 3rd International Conference on Extending Database Technology, EDBT'92*, Volume 580 of *Lecture Notes in Computer Science*, Vienna, Austria, pp. 421–436.

Kifer, M. and V. S. Subrahmanian (1992). Theory of Generalized Annotated Logic Programming and its Applications. *Journal of Logic Programming 12*(4), 335–368.

Koblick, R. (1999, March). Concordia. *Communications of the ACM 42*(3), 96–97.

Koehler, J. and R. Treinen (1995). Constraint Deduction in an Interval-based Temporal Logic. In M. Fisher and R. Owens (Eds.), *Executable Modal and Temporal Logics*, Volume 897 of *Lecture Notes in Artificial Intelligence*, pp. 103–117. Springer-Verlag.

Koller, D. (1998). Structured Probabilistic Models: Bayesian Networks and Beyond. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence, AAAI'98*, Madison, Wisconsin. AAAI Press/MIT Press. `http://robotics.Stanford.EDU/~koller/Invited98`.

Koubarakis, M. (1994). Complexity Results for First Order Theories of Temporal Constraints. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR-94)*, Bonn, Germany, pp. 379–390.

Kowalski, R. (1995). Using metalogic to reconcile reactive with rational agents. In *Meta-Logics and Logic Programming*. MIT Press.

Kowalski, R. and F. Sadri (1998). Towards a unified agent architecture that combines rationality with reactivity. draft manuscript.

Kraus, S. and D. Lehmann (1988). Knowledge, Belief and Time. *Theoretical Computer Science 58*, 155–174.

Kraus, S., K. Sycara, and A. Evenchik (1998). Reaching agreements through argumentation: a logical model and implementation. *Artificial Intelligence 104*(1-2), 1–69.

Kraus, S., J. Wilkenfeld, and G. Zlotkin (1995). Multiagent Negotiation Under Time Constraints. *Artificial Intelligence 75*(2), 297–345.

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

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

Kushmerick, N., S. Hanks, and D. Weld (1995). An Algorithm for probabilistic planning. *Artificial Intelligence 76*(1-2), 239–286.

Labrou, Y. and T. Finin (1997a, February). A Proposal for a new KQML Specification. Technical Report TR CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD 21250. `http://www.cs.umbc.edu/~jklabrou/publications/tr9703.ps`.

Labrou, Y. and T. Finin (1997b). Semantics for an Agent Communication Language. In *International Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp. 199–203.

Ladkin, P. (1986). Primitives and units for time specification . In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Volume 1, Philadelphia, PA, pp. 354–359. Morgan Kaufmann.

Ladkin, P. (1987). The Completeness of a Natural System for Reasoning with Time Intervals. In J. McDermott (Ed.), *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, Milan, Italy, pp. 462–465. Morgan Kaufmann.

Lakshmanan, V. S., N. Leone, R. Ross, and V. S. Subrahmanian (1997, September). ProbView: A Flexible Probabilistic Database System. *ACM Transactions on Database Systems 22*(3), 419–469.

Lakshmanan, V. S. and F. Sadri (1994a, September). Modeling Uncertainty in Deductive Databases. In *Proceedings of the 5th International Conference on Database Expert Systems and Applications, (DEXA'94)*, Volume 856 of *Lecture Notes in Computer Science*, Athens, Greece, pp. 724–733.

Lakshmanan, V. S. and F. Sadri (1994b, November). Probabilistic Deductive Databases. In *Proceedings of the International Logic Programming Symposium, (ILPS'94)*, Ithaca, New York, pp. 254–268. MIT Press.

Lakshmanan, V. S. and N. Shiri (1999). A Parametric Approach with Deductive Databases with Uncertainty. accepted for publication in IEEE Transactions on Knowledge and Data Engineering.

Lamport, L. (1994, May). The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems 16*(3), 872–923.

Lande, D. B. and M. Osjima (1998). *Programming and Deploying Java Mobile Agents with Aglets*. Massachusetts: Adison Wesley.

Leban, B., D. McDonald, and D. Forster (1986). A Representation for Collections of Temporal Intervals. In *Proceedings of the 5th National Conference on Artificial Intelligence*, Volume 1, Philadelphia, PA, pp. 367–371. Morgan Kaufmann.

Lee, H., J. Tannock, and J. S. Williams (1993). Logic-based reasoning about actions and plans in artificial intelligence. *The Knowledge Engineering Review 11*(2), 91–105.

Lenat, D. (1995, November). CYC: A Large Scale Investment in Knowledge Infrastructure. *Communications of the ACM 38*(11), 32–38.

Li, M. and P. Vitanyi (1992). Average Case Complexity under the Universal Distribution Equals Worst Case Complexity. *Information Processing Letters 42*, 145–150.

Liu, L., L. Yan, and M. Ozsu (1997). Interoperability in Large-Scale Distributed Information Delivery Systems. In A. Dogac, L. Kalinichenko, M. Ozsu, and A. Sheth (Eds.), *Advances in Workflow Systems and Interoperability*. Springer-Verlag.

Lloyd, J. (1984, 1987). *Foundations of Logic Programming*. Berlin, Germany: Springer-Verlag.

Lobo, J., J. Minker, and A. Rajasekar (1992). *Foundations of Disjunctive Logic Programming*. Cambridge, MA: MIT Press.

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

Lu, J., G. Moerkotte, J. Schue, and V. S. Subrahmanian (1995, May). Efficient Maintenance of Materialized Mediated Views. In M. Carey and D. A. Schneider (Eds.), *Proceedings of ACM SIGMOD Conference on Management of Data*, San Jose, CA, pp. 340–351.

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

Luke, S., L. Spector, D. Rager, and J. Hendler (1997). Ontology-based Web Agents. In *Proceedings of First International Conference on Autonomous Agents*, Marina del Rey, CA, pp. 59–66.

Maes, P. (1989). The dynamics of action selection. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Detroit, MI, pp. 991–997.

Manna, Z. and A. Pnueli (1992). *Temporal Logic of Reactive and Concurrent Systems*. Addison Wesley.

Manola, F., et al. (1992, March). Distributed Object Management. *International Journal of Intelligent and Cooperative Informations Systems 1*(1), 5–42.

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

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

Marek, W. and M. Truszczyński (1991). Autoepistemic Logic. *Journal of the ACM 38*(3), 588–619.

Marek, W. and M. Truszczyński (1993). *Nonmonotonic Logics – Context-Dependent Reasoning*. Springer-Verlag.

Martelli, M., V. Mascardi, and F. Zini (1997). CaseLP: a Complex Application Specification Environment based on Logic Programming. In *Proceedings of ICLP'97 Post Conference Workshop on Logic Programming and Multi-Agents*, Leuven, Belgium, pp. 35–50.

Martelli, M., V. Mascardi, and F. Zini (1998). Towards Multi-Agent Software Prototyping. In *Proceedings of The Third International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM98)*, London, UK, pp. 331–354.

Mayfield, J., Y. Labrou, and T. Finin (1996). Evaluation of KQML as an Agent Communication Language. In M. Wooldridge, J. P. Müller, and M. Tambe (Eds.), *Proceedings on the IJCAI Workshop on Intelligent Agents II : Agent Theories, Architectures, and Languages*, Volume 1037 of *LNAI*, Berlin, Germany, pp. 347–360. Springer-Verlag.

McDermott, D. (1982, December). A temporal logic for reasoning about processes and plans. *Cognitive Science 6*, 101–155.

Meyer, J.-J. C. and R. Wieringa (Eds.) (1993). *Deontic Logic in Computer Science*. Chichester: John Wiley & Sons.

Microsoft (1999). The Component Object Model: Technical Overview. Adapted from an article appearing in Dr. Dobbs Journal, December 1994, `http://www.microsoft.com/com/comPapers.asp`.

Millen, J. and T. Lunt (1992, May). Security for Object-Oriented Database Systems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA.

Minton, S., M. D. Johnston, A. B. Philips, and P. Laird (1992). Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligece 58*, 161–205.

Moore, R. (1985a). A Formal theory of Knowledge and Action. In J. Hobbs and R. Moore (Eds.), *Formal Theories of the Commonsesnse World*. Norwood, N.J.: ABLEX publishing.

Moore, R. (1985b). Semantical Considerations on Nonmonotonic Logics. *Artificial Intelligence 25*, 75–94.

Moret, B. M. E. and H. D. Shapiro (1991). *Design and Efficiency*, Volume I of *Algorithms from* **P** *to* **NP**. Redwood City, CA: Benjamin/Cummings Publishing Company.

Morgenstern, L. (1988). *Foundations of a Logic of Knowledge, Action, and Communication*. Ph. D. thesis, New York University.

Morgenstern, L. (1990). A Formal Theory of Multiple Agent Nonmonotonic Reasoning. In *Proceedings of the 8th National Conference on Artificial Intelligence, AAAI'90*, Boston, Massachusetts, pp. 538–544. AAAI Press/MIT Press.

Moulin, B. and B. Chaib-Draa (1996). An Overview of Distributed Artificial Intelligence. In G. M. P. O'Hare and N. R. Jennings (Eds.), *Foundations of Distributed Artificial Intelligence*, pp. 3–55. John Wiley & Sons.

Neches, R., R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. R. Swarton (1991). Enabling Technology for Knowledge Sharing. *AI Magazine 12*(3), 57–63.

Necula, G. C. and P. Lee (1997). Research on Proof-Carrying Code on Mobile-Code Security. In *Proceedings of the Workshop on Foundations of Mobile Code Security*. `http://www.cs.cmu.edu/~necula/pcc.html`.

Ng, R. and V. S. Subrahmanian (1993a). A Semantical Framework for Supporting Subjective and Conditional Probabilities in Deductive Databases. *Journal of Automated Reasoning 10*(2), 191–235.

Ng, R. and V. S. Subrahmanian (1993b). Probabilistic Logic Programming. *Information and Computation 101*(2), 150–201.

Ng, R. and V. S. Subrahmanian (1995). Stable Semantics for Probabilistic Deductive Databases. *Information and Computation 110*(1), 42–83.

Niezette, M. and J. Stevenne (1992). An Efficient Symbolic Representation of Periodic Time. In *Proceedings First International Conference on Information and Knowledge Management*, Baltimore, Maryland.

Nilsson, N. (1986). Probabilistic Logic. *Artificial Intelligence 28*, 71–87.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Morgan Kaufmann.

Nirkhe, M., S. Kraus, D. Perlis, and M. Miller (1997). How to (plan to) meet a deadline between Now and Then. *Journal of Logic and Computation 7*(1), 109–156.

Nishigaya, T. (1997). Design of Multi-Agent Programming Libraries for Java. `http://www.fujitsu.co.jp/hypertext/free/kafka/paper`.

OMG (1998a, January). CORBA/IIOP2.2 Specification. Technical Report 98.02.1, OMG. `http://www.omg.org`.

OMG (1998b, December). CORBAServices: Common Services Specification. Technical Report 98-12-09, OMG. `http://www.omg.org/`.

Ozsu, M. T., U. Dayal, and P. Valduries (Eds.) (1994). *Distributed Object Management*. San Mareo, California: Morgan Kaufmann. Edited collection of papers presented at the International Workshop on Distributed Object Management, held in August 1992 at the University of Alberta, Canada.

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison Wesley.

Paterson, M. and M. Wegman (1978). Linear Unification. *Journal of Computer and System Sciences 16*, 158–167.

Patil, R., R. E. Fikes, P. F. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches (1997). The DARPA Knowledge Sharing Effort. In M. Huhns and M. Singh (Eds.), *Readings in Agents*, pp. 243–254. Morgan Kaufmann.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausable Inference*. Morgan Kaufmann.

Plaisted, D. (1993). Equational Reasoning and Term Rewriting Systems. In D. Gabbay, C. Hogger, and J. Robinson (Eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, Volume 1, pp. 274–364. Oxford: Clarendon Press.

Poole, D. (1997). The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence 94*(1-2), 7–56.

Rabinovich, A. (1998). Expressive Completeness of Temporal Logic of Action. *Lecture Notes in Computer Science 1450*, 229–242.

Rao, A. S. (1995). Decision Procedures for Propositional Linear-Time Belief-Desire-Intention Logics. In M. Wooldridge, J. Müller, and M. Tambe (Eds.), *Intelligent Agents II – Proceedings of the 1995 Workshop on Agent Theories, Architectures and Languages (ATAL-95)*, Volume 890 of *LNAI*, pp. 1–39. Berlin, Germany: Springer-Verlag.

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

Rao, A. S. and M. Georgeff (1995, June). Formal models and decision procedures for multi-agent systems. Technical Report 61, Australian Artificial Intelligence Institute, Melbourne.

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

Reiter, R. (1980). A Logic for Default Reasoning. *Artificial Intelligence 13*, 81–132.

Rogers Jr., H. (1967). *Theory of Recursive Functions and Effective Computability*. New York: McGraw-Hill.

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

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

Rosenschein, S. J. and L. P. Kaelbling (1995). A Situated View of Representation and Control. *Artificial Intelligence 73*, 149–173.

Ross, S. (1997). *A First Course in Probability*. Prentice-Hall.

Rouzaud, Y. and L. Nguyen-Phoung (1992). Integrating Modes and Subtypes into a Prolog Type Checker. In *Proceedings of the International Joint Conference/Symposium on Logic Programming*, pp. 85–97. MIT Press.

Rus, D., R. Gray, and D. Kotz (1997). Transportable Information Agents. In M. Huhns and M. Singh (Eds.), *Readings in Agents*, pp. 283–291. Morgan Kaufmann.

Russell, S. J. and P. Norvig (1995). *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice-Hall.

Salton, G. and M. McGill (1983). *Introduction to Modern Information Retrieval*. McGraw-Hill.

Sander, T. and C. Tschudin (1998). Protecting Mobile Agents Against Malicious Hosts. In G. Vigna (Ed.), *Mobile agents and security*, Volume 1419 of *Lecture Notes in Computer Science*, pp. 44–60. New York, NY: Springer-Verlag.

Sandholm, T. (1993, July). An Implementation of the Contract Net Protocol Based on Marginal Cost Calculations. In *Proceedings of the 11th National Conference on Artificial Intelligence*, Washington, DC, pp. 256–262. AAAI Press/MIT Press.

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

Schafer, J., T. J. Rogers, and J. Marin (1998, September). Networked Visualization of Heterogeneous US Army War Reserves Readiness Data. In S. Jajodia, T. Ozsu, and A. Dogac (Eds.), *Advances in Multimedia Information Systems, 4th International Workshop, MIS'98*, Volume 1508 of *Lecture Notes in Computer Science*, Istanbul, Turkey, pp. 136–147. Springer-Verlag.

Schoppers, M. and D. Shapiro (1997). Designing Embedded Agents to Optimize End-User Objectives. In *International Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp. 2–12.

Schroeder, M., I. de Almeida Mora, and L. M. Pereira (1997). A Deliberative and Reactive Diagnosis Agent based on Logic Programming. In M. W. J.P. Muller and N. Jennings (Eds.), *Intelligent Agents III: Lecture Notes in Artificial Intelligence Vol. 1193*, pp. 293–307. Springer-Verlag.

Schumacher, H. J. and S. Ghosh (1997, July). A fundamental framework for network security. *Journal of Network and Computer Applications 20*(3), 305–322.

Schwartz, R. and S. Kraus (1997). Bidding Mechanisms for Data Allocation in Multi-Agent Environments. In *International Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp. 56–70.

Schwarz, G. and M. Truszczynski (1996). Nonmonotonic Reasoning is Sometimes Simpler! *Journal of Logic and Computation 6*, 295–308.

Shafer, G. and J. Peal (Eds.) (1990). *Readings in uncertain reasoning*. Morgan Kaufmann.

Shehory, O. and S. Kraus (1998). Methods for Task Allocation via Agent Coalition Formation. *Artificial Intelligence 101*(1-2), 165–200.

Shehory, O., K. Sycara, and S. Jha (1997). Multi-Agent Coordination through Coalition Formation. In *International Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp. 135–146.

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

Shoham, Y. (1993). Agent Oriented Programming. *Artificial Intelligence 60*, 51–92.

Shoham, Y. (1999, March/April). What we talk about when we talk about software agents. *IEEE Intelligent Systems 14*, 28–31.

Siegal, J. (1996). *CORBA Fundementals and Programming*. New York: John Wiley & Sons.

Silberschatz, A., H. Korth, and S. Sudarshan (1997). *Database System Concepts*. McGraw-Hill.

Singh, M. P. (1997). A Customizable Coordination Service for Autonomous Agents. In *International Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp. 86–99.

Singh, M. P. (1998). Toward a model theory of actions: How Agents do it in branching time. *Computational Intelligence 14*(3), 287–305.

Smith, R. G. and R. Davis (1983). Negotiation as a Metaphor for Distributed Problem Solving. *Artificial Intelligence 20*, 63–109.

Soley, R. M. and C. M. Stone (Eds.) (1995). *Object Management Architecture Guide* (third ed.). John Wiley & Sons.

Sonenberg, E., G. Tidhar, E. Werner, D. Kinny, M. Ljungberg, and A. Rao (1992). Planned Team Activity. Technical Report 26, Australian Artificial Intelligence Institute, Australia.

Soueina, S. O., B. H. Far, T. Katsube, and Z. Koono (1998). MALL: A multi-agent learning language for competitive and uncertain environments. *IEICE TRANSACTIONS ON INFORMATION AND SYSTEMS 12*, 1339–1349.

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

Stallings, W. (1995). *Title Network and Internetwork Security: Principles and Practice*. Englewood Cliffs: Prentice-Hall.

Stoffel, K., M. Taylor, and J. Hendler (1997). Efficient Management of Very Large Ontologies. In *Proceedings of American Association for Artificial Intelligence Conference (AAAI-97)*, Providence, RI, pp. 442–447. AAAI Press/MIT Press.

Subrahmanian, V. S. (1987, September). On the Semantics of Quantitative Logic Programs. In *Proceedings of the 4th IEEE Symposium on Logic Programming*, pp. 173–182. Computer Society Press.

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

Sycara, J. and D. Zeng (1996a). Coordination of multiple intelligent software agents. *International Journal of Intelligent and Cooperative Information Systems 5*, 181–211.

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

Sycara, K. P. (1987). *Resolving Adversarial Conflicts: An Approach to Integrating Case-Based and Analytic Methods*. Ph. D. thesis, School of Information and Computer Science, Georgia Institute of Technology.

Tai, H. and K. Kosaka (1999, March). The Aglets Project. *Communications of the ACM 42*(3), 100–101.

Takeda, H., K. Iwata, M. Takaai, A. Sawada, and T. Nishida (1995). An ontology-based cooperative environment for real-world agents. In V. Lesser (Ed.), *Proceedings of the First International Conference on Multi–Agent Systems*, San Fransisco, CA. MIT Press.

Tambe, M., L. Johnson, and W.-M. Shen (1997). Adaptive Agent Tracking in Real-World Multiagent Domains. In M. Huhns and M. Singh (Eds.), *Readings in Agents*, pp. 504–508. Morgan Kaufmann.

Tari, Z. (1997, June). Using agents for secure access to data in the Internet. *IEEE Communications Magazine 35*(6), 136–140.

Tarski, A. (1981, January). *Logic, Semantics, Metamathematics*. Hackett Pub Co.

Thiébaux, S., J. Hertzberg, W. Shoaff, and M. Schneider (1995). A stochastic model of actions and plans for anytime planning under uncertainty. *International Journal for Intelligent Systems 10*(2).

Thirunavukkarasu, C., T. Finin, and J. Mayfield (1995, November). Secret Agents – A Security Architecture for the KQML Agent Communication Language. In *Intelligent Information Agents Workshop, held in conjunction with Fourth International Conference on Information and Knowledge Management CIKM'95*, Baltimore, MD.

Thomas, B., Y. Shoham, A. Schwartz, and S. Kraus (1991, August). Preliminary Thoughts on an Agent Description Language. *International Journal of Intelligent Systems 6*(5), 497–508.

Thomas, R. H. (1979, June). A Majority Consensus Approach to Concurrency Control for Multiple Copy Data Bases. *ACM Transactions on Database Systems 4*(2), 180–209.

Tomasic, A., L. Raschid, and P. Valduriez (1998). Scaling Access to Data Sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering 10*(5), 808–823.

Tork Roth, M., et al. (1996, June). The Garlic Project. In H. V. Jagadish and I. S. Mumick (Eds.), *Proceedings of ACM SIGMOD Conference on Management of Data*, Montreal, Canada, pp. 557–558.

Twok, C. T. and D. Weld (1996). Planning to Gather Information. In *Proceedings of the 13th National Conference on Artificial Intelligence*, Portland, Oregon, pp. 32–29.

Ullman, J. D. (1989). *Principles of Database and Knowledge Base Systems*. Computer Science Press.

Ushioda, A. (1996). Hierarchical clustering of words and application to NLP tasks. In *Proceedings of the 4th workshop on very large corpora*, Copenhagen.

Vardi, M. (1982). Complexity of Relational Query Languages. In *Proceedings 14th ACM Symposium on Theory of Computing*, San Francisco, CA, pp. 137–146. ACM Press.

Vere, S. and T. Bickmore (1990). Basic Agent. *Computational Intelligence 4*, 41–60.

Verharen, E., F. Dignum, and S. Bos (1997). Implementation of a Cooperative Agent Architecture Based on the Language-Action Perspective. In *International Workshop on Agent Theories, Architectures, and Languages*, Providence, RI, pp. 26–39.

Vigna, G. (1998a). Cryptographic Traces for Mobile Agents. In G. Vigna (Ed.), *Mobile agents and security*, Volume 1419 of *Lecture Notes in Computer Science*, pp. 137–153. New York, NY: Springer-Verlag.

Vigna, G. (Ed.) (1998b). *Mobile agents and security*. New York, NY: Springer-Verlag. Lecture Notes in Computer Science, Volume 1419.

Vila, L. (1994, March). A Survey on Temporal Reasoning in Artificial Intelligence. *AI Communications 7*(1), 4–28.

Vinoski, S. (1997, February). CORBA: Integrating Diverse Applications Within Distributed Heterogenous Environments. *IEEE Communications Magazine 35*(2), 46–55.

Vlasie, D. (1996). The Very Particular Structure of the Very Hard Instances. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence, AAAI'96*, Portland, Oregon, pp. 266–270. AAAI Press/MIT Press.

Weinstein, P. and W. P. Birmingham (1997, August). Service Classification in a Proto-Organic Society of Agents. In *Proceedings of the IJCAI-97 Workshop on Artificial Intelligence in Digital Libraries*, Nagoya, Japan.

White, J. (1997, April). Mobile Agents. In J. M. Bradshaw (Ed.), *Software Agents*. Cambridge, MA: MIT Press.

Wiederhold, G. (1993). Intelligent Integration of Information. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Washington, DC, pp. 434–437.

Wilder, F. (1993). *A Guide to the TCP/IP Protocol Suite*. Artech House.

Winslett, M., K. Smith, and X. Qian (1994, December). Formal Query Languages for Secure Relational Databases. *ACM Transactions on Database Systems 19*(4), 626–662.

Woelk, D., P. Cannata, M. Huhns, W. Shen, and C. Tomlinson (1993, January). Using Carnot for Enterprise Information Integration. In *Second International Conference on Parallel and Distributed Information Systems*, San Diego, CA, pp. 133–136.

Wooldridge, M. and N. Jennings (1997). Formalizing the Cooperative Problem Solving Proces. In M. Huhns and M. Singh (Eds.), *Readings in Agents*, pp. 430–440. Morgan Kaufmann.

Wooldridge, M. J. and N. R. Jennings (1995). Agent Theories, Architectures and Languages: A survey. In M. J. Wooldridge and N. R. Jennings (Eds.), *Intelligent Agents*, Volume 890 of *Lecture Notes in Artificial Intelligence*, pp. 1–39. Springer-Verlag.

Zadeh, L. A. (1965). Fuzzy Sets. *Information and Control 8*(3), 338–353.

Zaniolo, C., S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari (1997). *Advanced Database Systems*. Morgan Kaufmann.

Zapf, M., H. Mueller, and K. Geihs (1998). Security requirements for mobile agents in electronic markets. *Lecture Notes in Computer Science 1402*, 205–217.

Zeng, L. and H. Wang (1998, August). Towards a Multi-Agent Security System: A Conceptual Model for Internet Security. In *Proceedings of Fourth AIS (Association for Information Systems) Conference*, Baltimore, Maryland.

# Index

## Symbols