

Combining Agents, ASP and Planning

NICTA, Jul-Aug 2003

- **July and August with the exception of third week in July.**
- **Time:** Thursday, Friday, 14-16, starting on 3rd July 2003.
- Lecture Course is in the **first 3 weeks on theoretical issues in general agent systems and answer set programming**, emphasis on mathematical-logical foundations. **Remaining two weeks devoted to a particular agent system** and some demonstrations.
- `www.cs.man.ac.uk/~jdix/LECTURING/NICTA03.html`.

First and second week (Chapters 1–3)

The first part of this lecture course is mainly based on

Multi-Agent Systems

(Gerhard Weiss)

MIT Press, June 1999.

We describe **general methods** and **techniques**.

Third week (Chapter 4)

The second part of this lecture course is mainly based on

1. **Knowledge representation, reasoning and declarative problem solving with Answer sets**
(Chitta Baral), MIT Press, February 2003.

2. **Planning in Answer Set Programming using Ordered Task Decomposition**
(Jürgen Dix, Ugur Kuter and Dana Nau)
Theory and Practice of Logic Programming, to appear 2004.
<www.cs.umd.edu/users/ukuter/ASP_Planning/>

We give an introduction to the newly emerged paradigm of **Answer Set Programming** and illustrate it with recent research on how to realise HTN-planning in this paradigm.

Fourth and fifth week (Chapters 5–9)

The third part of this lecture course is mainly based on

Heterogenous Agent Systems

(Subrahmanian/Bonatti/Dix/Eiter/Kraus/Özcan/Ross)

MIT Press, August 2000.

We describe the **IMPACT approach** and its **underlying foundations**. We also give two demos and present an approach of monitoring agents through planning (using an ASP engine).

Overview (**Agent Systems in general**)

1. Introduction

2. Distributed Decision Making (2 Lectures)

3. Contract Nets, Coalition Formation

Overview (**Answer Set Programming**)

4. ASP: Foundations and an Application to Planning (2 Lectures)

Overview (**IMPACT**)

- 5. *IMPACT* Architecture**
- 6. Actions and Agent Programs**
- 7. Implementing Agents: An Application**
- 8. Agent Systems and Planning**
- 9. Extensions of IMPACT**

Chapter 5. *IMPACT* Architecture and Code Calls

5.1 Scenarios

5.2 Agent/Server Architecture

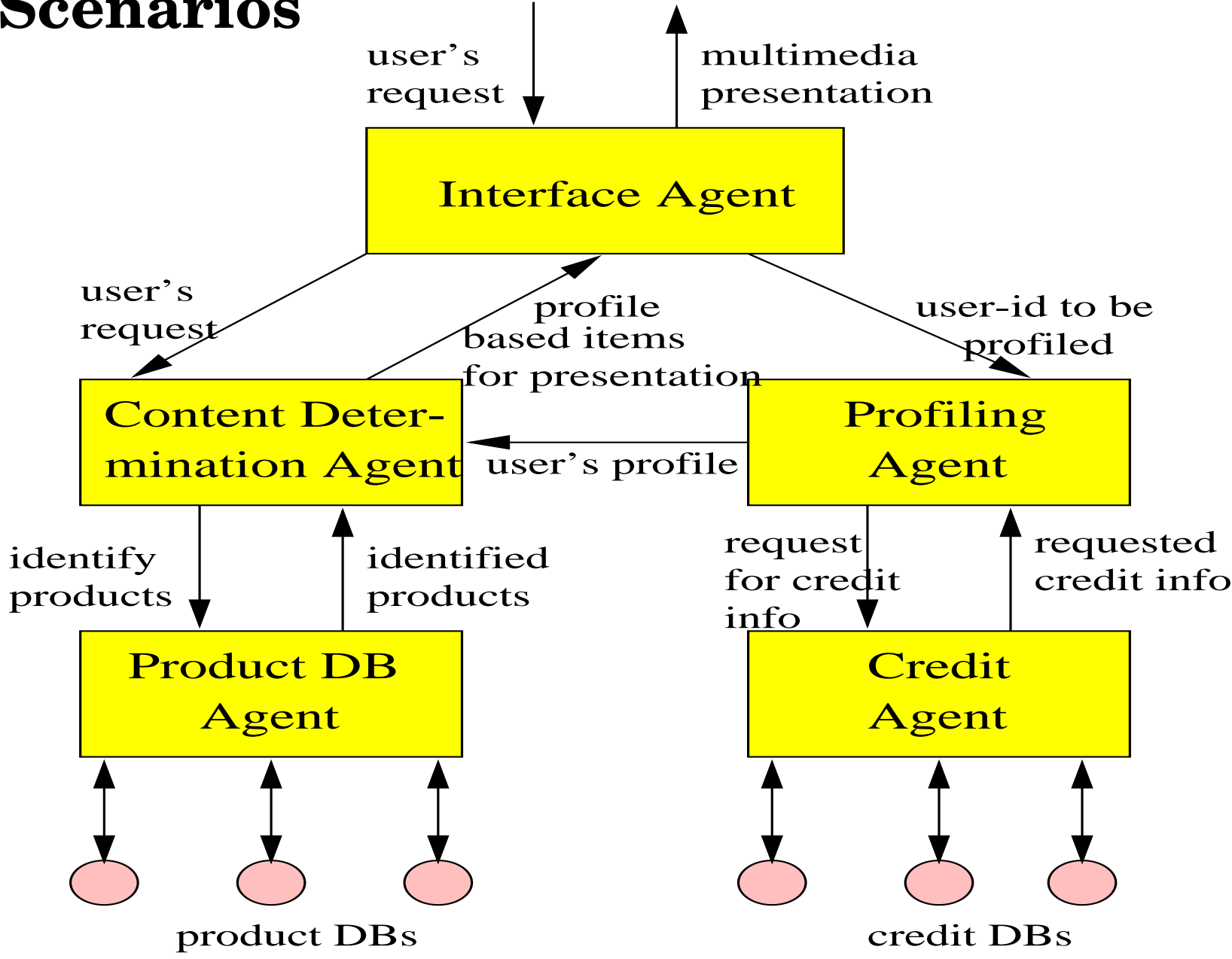
5.3 The Code Call Mechanism

5.4 Summary and References

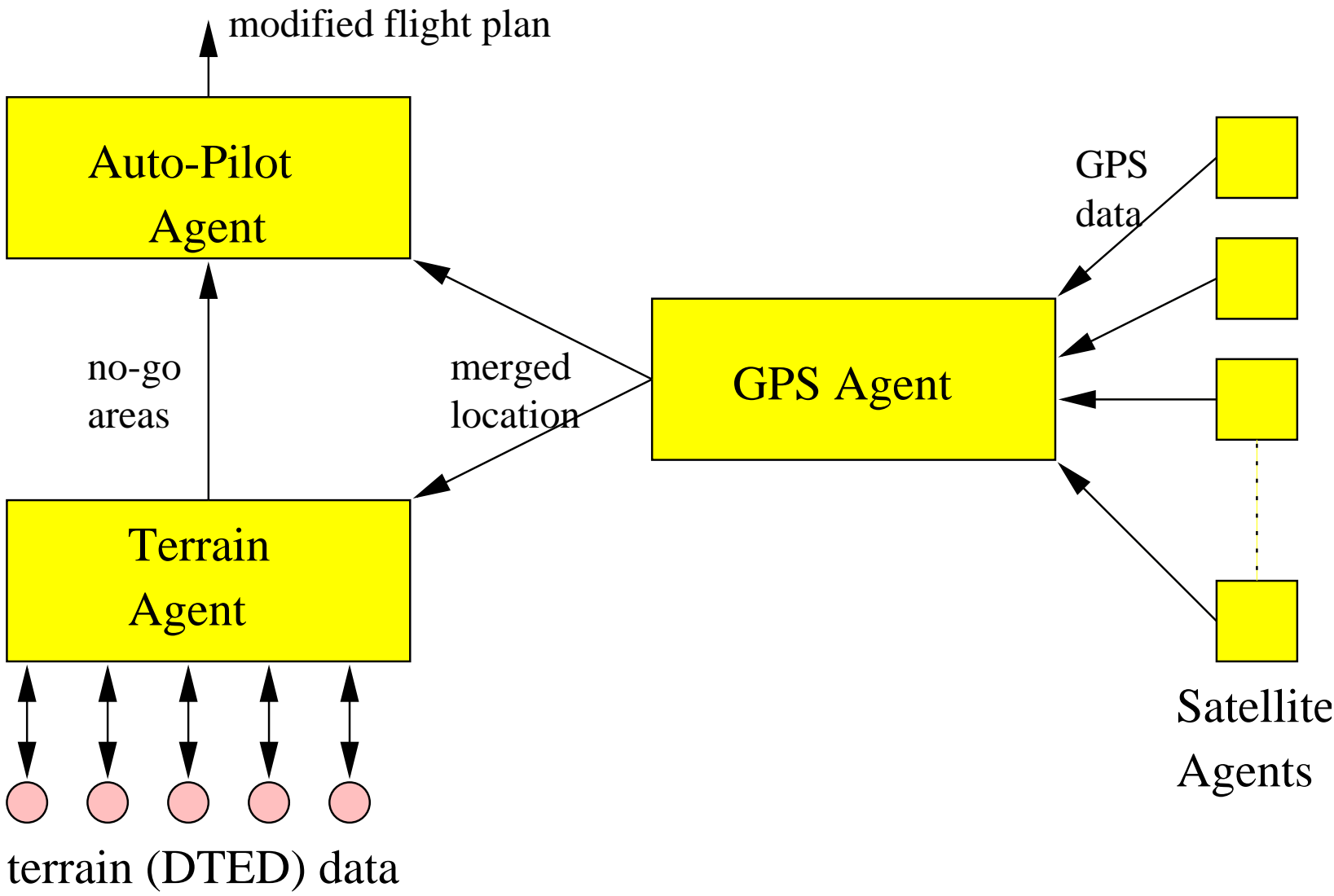
5 ***IMPACT* Architecture and Code Calls**

271-1

5.1 Scenarios



Controlled Flight into Terrain (CFIT) scenario



5.2 Agent/Server Architecture

Four main categories (Genesereth and Ketchpel 1994):

1. **transducer**: Each agent has an associated “transducer” which converts incoming messages and requests into a format intelligible to the agent.

Problem: n -agent system may need $O(n^2)$ transducers (not desirable).

2. **wrapper**: “*inject code into a program to allow it to communicate*”.

Principle: each agent has an associated body of code that is expressed in a common language (or one of few) languages used by other agents.

3. **code rewriting**: complete rewriting to implement an agent.

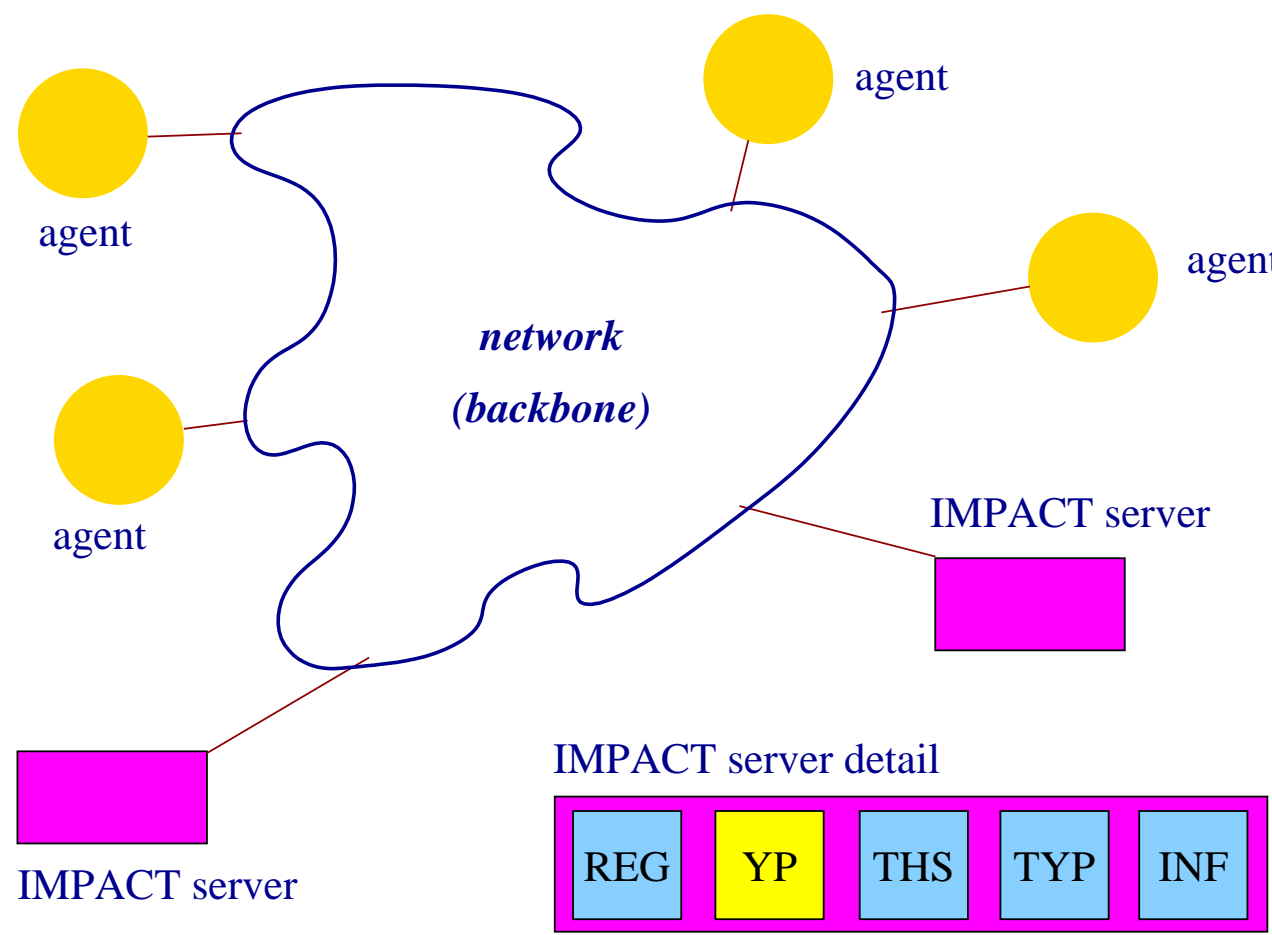
Problem: Very expensive alternative.

4. **mediation approach (Wiederhold 1993):** all agents 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).

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.

Overall *IMPACT* Architecture

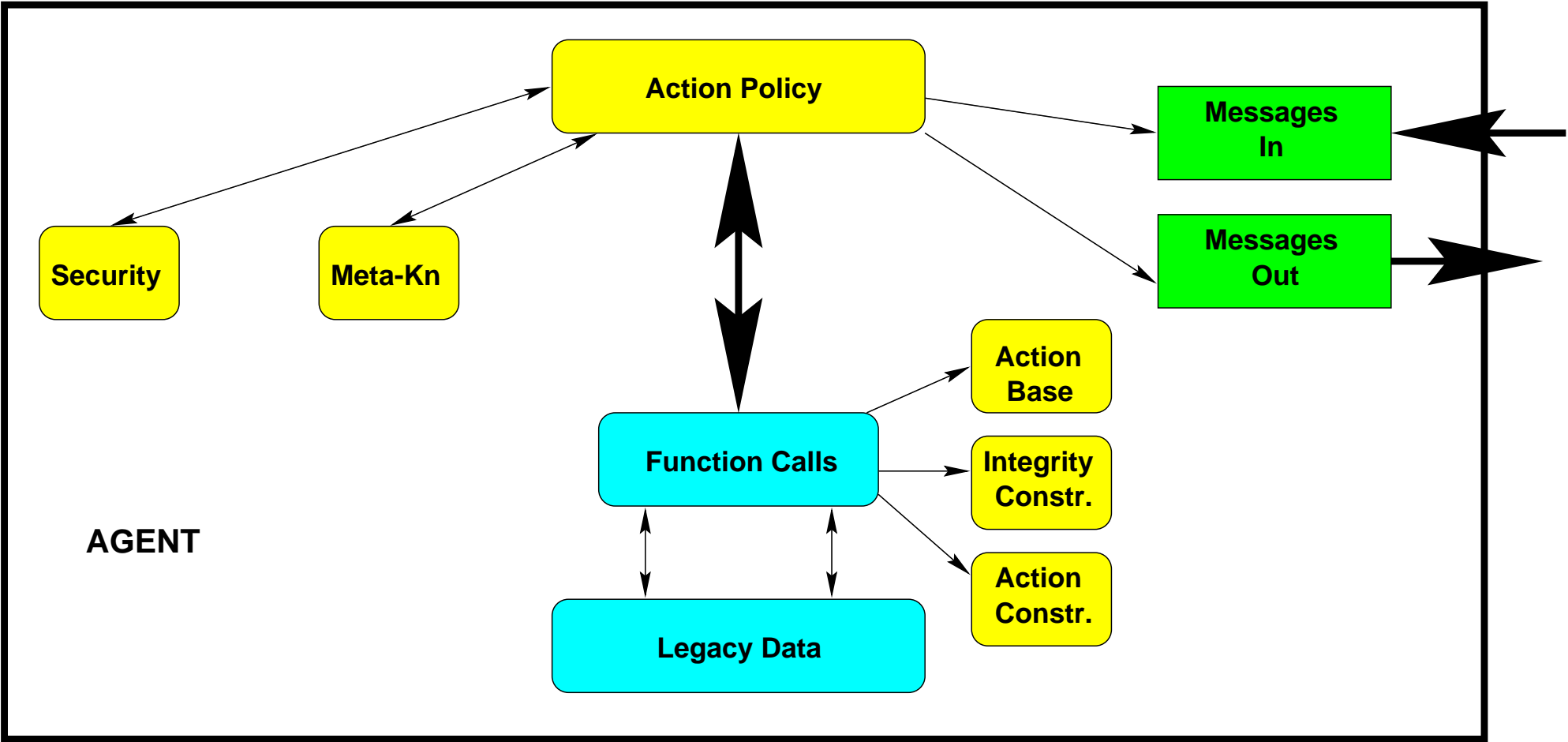


Server Architecture

An *IMPACT* Server is a collection of the following servers:

- **Registration (REG):** creator enters agent to the system, which services it provides, and who may use them.
- **Yellow Pages (YP):** Process requests from agents to find other agents providing some service (matchmaking).
- **Thesaurus (THS):** requested when agent services are entered, or YP server performs matchmaking.
- **Types (TYP):** Server maintains a set of class hierarchies used by different agents and the inclusion relationship(s).
- **Interfaces (INF):** GUI for human user

Basic *IMPACT* Agent Architecture



Agent/Service Registration

Register Services

Relevant Hierarchy: Service Location:

Agent Name: Content Determination Agent

Password: xxxxxxxx

Service Allowed: Inventory manager

Add ServiceRemove ServiceView AgentsView Services

Agent Type: html

Description: http://www.store.com/inventory

Agent Allowed: store employees

Select TypeAdd AgentRemove AgentsCancel

Warning: Applet Window

Agent Service Description Language (SDL)

Describe agent services in a simple HTML-style language.

Each service has

- a **name**:

<name> ::= <verb>’<nounterm>

<nounterm> ::= <noun> | <noun>’(<noun>)’
- **inputs**: (typed) values. Distinguish *mandatory* and *optional* inputs.
- **outputs**: (typed) values
- **attributes**: e.g. cost, response time (optional)

Example (STORE application):

```
⟨S⟩  classify:user
      ⟨MI⟩ssn:String⟨MI⟩
      ⟨I⟩name:String⟨I⟩
      ⟨0⟩class:UserProfile⟨0⟩
⟨\S⟩
```

Agent Service Descriptions

Definition 5.1 (Verbs, Nouns, $\text{nt}(\text{Nouns})$)

Let Verbs be a set of verbs in English and Nouns a set of nouns in English.

- 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.
- $\text{nt}(\text{Nouns})$ denotes the set of all syntactically valid noun terms generated by the set Nouns.

Definition 5.2 (Service Name)

If $v \in \text{Verbs}$ and $nt \in \text{nt}$, then $v:nt$ is called a *service name*.

Service List for the STORE example

AGENT	SERVICES
credit	<i>provide:information(credit)</i> <i>provide:address</i>
profiling	<i>provide:user-profile</i> <i>classify:user</i>
productDB	<i>provide:description(product)</i>
contentDetermin	<i>prepare:presentation(product)</i> <i>identify:items</i>
saleNotification	<i>identify:user-profile</i> <i>determine:items</i>

Service List for the CFIT Scenario

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

Synonyms and Thesaurus

What if agent **a** seeks another one offering a service q_s ?

We need to match q_s with other services in the yellow pages.

Example: An agent looks for an agent offering the service *generate:map(ground)*.

Answer:

CFIT **terrain** agent: *ground* and *terrain* are synonymous.

Let

- Σ be set of English words, such that Σ contains only verbs or only noun-terms.
- \sim be an equivalence relation on Σ .

Definition 5.3 (Σ -node)

A Σ -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. $x, y \in N \Rightarrow x \sim y$.

Σ -nodes are equivalence classes of Σ .

Example: An agent looks for an agent offering the service *generate:map(area)*.

Answer: CFIT **terrain** agent: *area* can be specialised to *terrain*.

Definition 5.4 (Σ -Hierarchy)

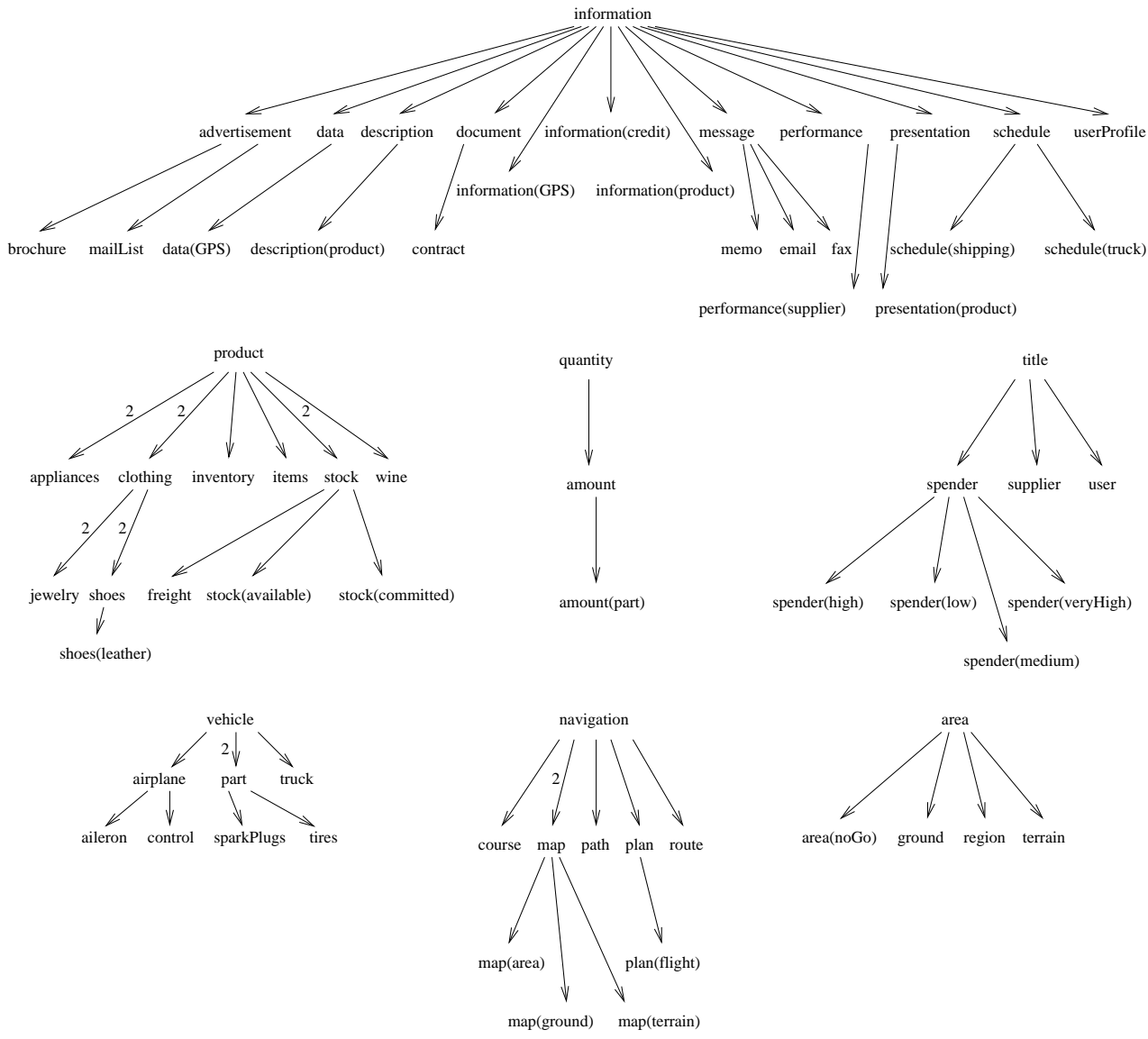
A Σ -Hierarchy is a weighted, directed acyclic graph

$\mathcal{SH} =_{def} (T, E, \wp)$ such that:

1. T is set of nonempty Σ -nodes;
2. If t_1 and t_2 are different Σ -nodes in T , then t_1 and t_2 are disjoint;
3. \wp is a mapping $\wp : E \rightarrow \mathbb{Z}^+$ indicating a positive distance between two neighbouring vertices.

Note: no requirement that \wp satisfies any metric axioms at this point.

Noun Hierarchy



Definition 5.5 (Distance between two terms)

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

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

$d_{\mathcal{SH}}$ is well defined and satisfies the triangle inequality.

Hierarchy Browsing Screen Dump

Verb Hierarchy

Hierarchy:book.vh

Location:/find

.../

choose (1)

classify (1)

determine (1)

identify (1)

locate (1)

Value:identify

Search Hierarchy

Thesaurus Lookup

Insert Node

Remove Node

Save Changes

BOOKMARKS

REGISTER SERVICES

Noun Hierarchy

Hierarchy:book.nh

Location:/product

.../

appliances (1)

clothing (1)

inventory (1)

items (1)

stock (1)

wine (1)

Value:items

Search Hierarchy

Thesaurus Lookup

Insert Node

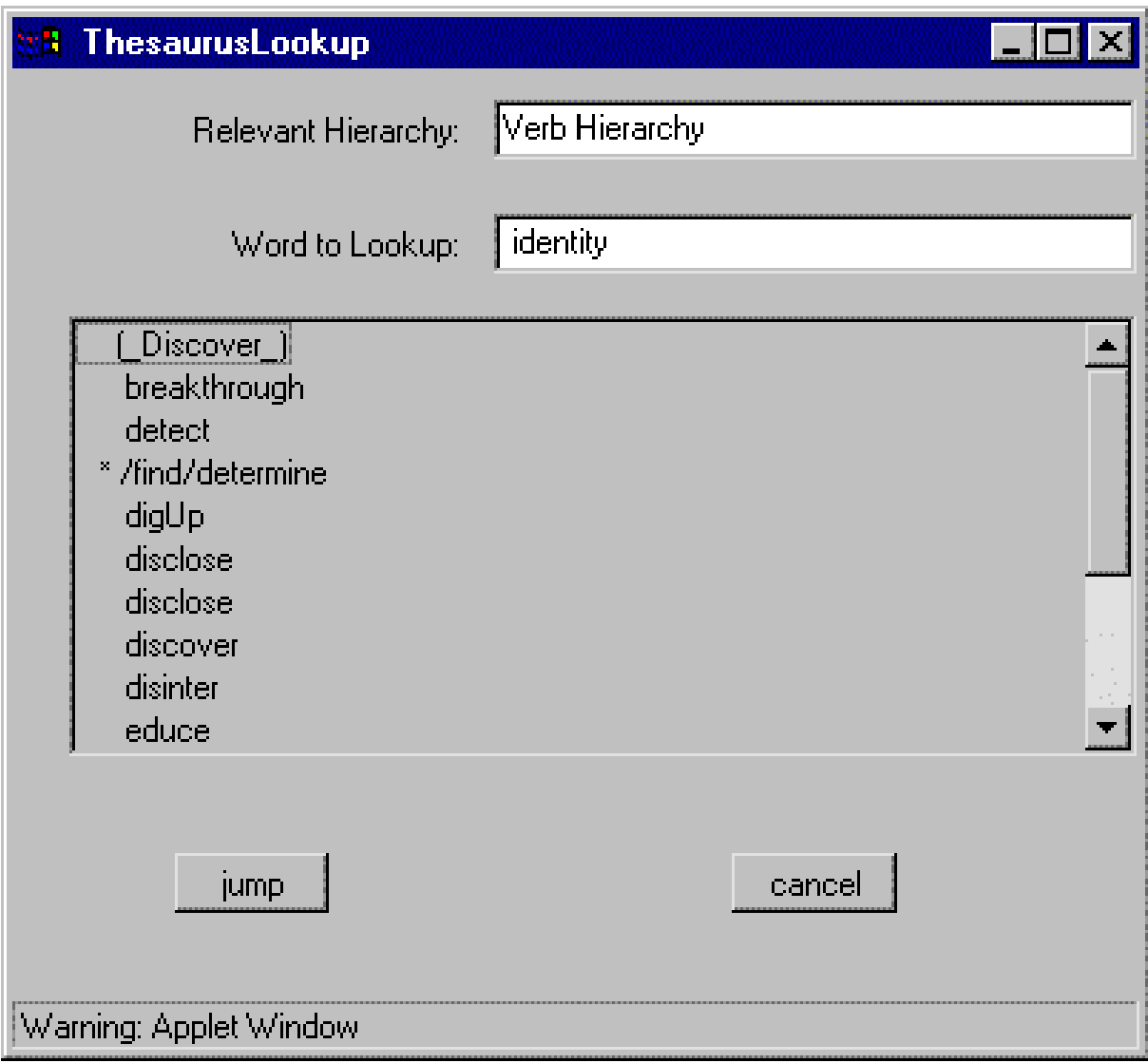
Remove Node

Save Changes

QUERIES

HELP

Thesaurus Screen Dump



5.3 The Code Call Mechanism

A definition of agents should not limit the choice of data structures and algorithms that an application designer must use.

CFIT: *terrain* agent on top of existing US military terrain reasoning software.

Accessing DB's: For instance, the Product Database agent *productDB* in the STORE example may access some file structures, as well as some databases.

Software Code Abstractions

Definition 5.6 ($\mathcal{S} = (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$)

Characterize the code on top of which an agent is built as

$\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$, 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 τ_1, \dots, τ_n and yields as a result a type

$c(\tau_1, \dots, \tau_n)$.

Intuitively:

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

Notation: $\mathcal{T}_{\mathcal{S}}^*$ is the *closure* of $\mathcal{T}_{\mathcal{S}}$ under the operations in $\mathcal{C}_{\mathcal{S}}$.

More formally:

Definition 5.7 ($\mathcal{C}_{\mathcal{S}}(\mathcal{T})$ and $\mathcal{T}_{\mathcal{S}}^*$)

a) Given a set \mathcal{T} of types, let

$$\mathcal{C}_{\mathcal{S}}(\mathcal{T}) =_{def} \mathcal{T} \cup \{\tau : \tau = c(\tau_1, \dots, \tau_n) \text{ for some } n\text{-ary} \\ c \in \mathcal{C}_{\mathcal{S}} \text{ and types } \tau_1, \dots, \tau_n \in \mathcal{T}\}.$$

b) $\mathcal{T}_{\mathcal{S}}^* =_{def} \bigcup_{i \in \mathbb{N}} \mathcal{T}_{\mathcal{S}}^i$, where

$$\begin{aligned} \mathcal{T}_{\mathcal{S}}^0 &=_{def} \mathcal{T}_{\mathcal{S}}, \\ \mathcal{T}_{\mathcal{S}}^{i+1} &=_{def} \mathcal{C}_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}^i). \end{aligned}$$

CFIT Revisited

$\mathcal{T}_S =_{def} \{\text{Map}, \text{Path}, \text{Plan}, \text{SatelliteReport}\}$.

Special class of maps called *DTED Digital Terrain Elevation Data* that specify the elevations of different regions of the world.

The **autoPilot** agent's set of functions \mathcal{F}_S might contain:

- *createFlightPlan*(*Location*/**Map**, *Flight_route*/**Path**, *Nogo*/**Map**) of type Plan.

The **gps** agent's set of functions \mathcal{F}_S might contain:

- *mergeGPSData*(*Data1*/**SatelliteReport**, *Data2*/**SatelliteReport**) of type SatelliteReport.

Agent State

Definition 5.8

At any given point t in time, the *state of an agent* will refer to a set $\mathcal{O}_s(t)$ of objects from the types \mathcal{T}_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.

Assumption: Except for appending messages to an agent **a**’s mailbox, another agent **b** cannot directly change **a**’s state. (However, it might do so indirectly by shipping the other agent a message issuing a change request.)

The Code Call Mechanism

Code Calls take data from heterogenous DB's so that such data can be considered as logical atoms (as terms in predicate logic).

An agent built on top of a piece, *S*, of software, may support several *API* functions, and it may or may not make all these functions available to other agents (through SDL).

Definition 5.9 (Code Call $\mathcal{S}:f(d_1, \dots, d_n)$)

Let $\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ be a software code. Let $f \in \mathcal{F}_{\mathcal{S}}$ be an n -ary function and d_1, \dots, d_n objects or variables such that each d_i matches the type requirements of the i 'th argument of f . Then,

$$\mathcal{S}:f(d_1, \dots, d_n)$$

is a *code call*. A code call is *ground*, if all d_i 's are objects.

$\mathcal{S}:f(d_1, \dots, d_n)$ may be read as: *execute function f as defined in package \mathcal{S} on the arguments d_1, \dots, d_n .*

Notation: We also write $\mathbf{a}:f(d_1, \dots, d_n)$ instead of $\mathcal{S}:f(d_1, \dots, d_n)$ where \mathcal{S} is provided by agent \mathbf{a} .

Comment 1 (Assumption on the Output Signature) *Without loss of generality, we assume that the **output signature** of any code call is a **set**.*

(If a function does not return a set, but rather returns an atomic value, then that value can be coerced into a set anyway—by treating the value as shorthand for the singleton set containing just the value.)

Examples:

1. **supplier** : *monitorStock*(3, part_008).
The result of this call is either { amount_available }, or the set { amount_not_available }.
2. **supplier** : *shipFreight*(3, part_008, truck, X, paris).
Create a pickup schedule for shipping 3 pieces of part_008 from location X to paris by truck. Until a value is specified for X, this code call cannot be executed.
3. **GPS** : *mergeGPSData*(S1, S2) merges two pieces, S1 and S2, of satellite data, but the values of the two pieces are not stated.

Variables

root variables: For any type $\tau \in \mathcal{T}_{\mathcal{S}}$ there is a set $root(\tau)$ of “root” variables ranging over τ .

Let τ be a complex record type having fields f_1, \dots, f_n .

- For every variable of type τ , $X.f_i$ is a variable of type τ_i where τ_i is the type of field f_i .
- If f_i itself has a sub-field g of type γ , then $X.f_i.g$ is a variable of type γ , and so on: *path variables* .
- For any path variable Y of the form $X.path$, where X is a root variable, X is the root of Y , denoted by $root(Y)$.

Example 5.1 (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`.

Variable Assignment

Definition 5.10

An *assignment of objects to variables* is a set of equations of the form $V_1 := o_1, \dots, V_k := o_k$ where the V_i 's are (root or path) variables and the o_i 's are objects.

An assignment is *legal*, if the types of objects and corresponding variables match.

Example 5.2 (CFIT Revisited)

A legal assignment may be

```
X.height := 50,  X.sat_id := iridium_17,
X.dist := 25,  X.2dloc.x := 3,  X.2dloc.y := -4.
```

We write this as $(50, \text{iridium_17}, 25, \langle 3, -4 \rangle)$.

Code Call Atoms

Code call atoms are *logical atoms* that are layered on top of code-calls.

If cc is a code call, and x is either a variable symbol, or an object of the output type of cc , then

Definition 5.11

- **in**(x, cc),
- **not_in**(x, cc),

are called **code call atoms**. A code call atom is *ground* if no variable symbols occur anywhere in it.

- A code call atom of the form **in**(X, cc) succeeds just in case when X can be set to a pointer to one of the objects in the set of objects returned by executing the code call.
- A code call atom of the form **not_in**(X, cc) succeeds just in case X is not in the result set returned by cc (when X is an object), or when X cannot be made to point to one of the objects returned by executing the code call.

What effects does this have on the **state** of an agent?

It is an infinite set of ground code call atoms!

Examples:

- **in**(spender(high), **profiling** : *classifyUser*(Johnny_Rich)). This code call succeeds just in case the Profiling agent classifies Johnny Rich as a big spender.
- **not_in**(spender(low), **profiling** : *classifyUser*(U)). This code call succeeds just in case user U, whose identity must be instantiated prior to evaluation, is *not* classified as a low spender by the **profiling** agent.

Definition 5.12 (Code Call Condition)

A *code call condition* is defined as follows:

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

Any code call condition 1.-3. is *atomic*.

Examples:

1. **in**(X, **profiling** : *classifyUser*(Johnny_Rich)) &
 in(Y, **profiling** : *classifyUser*(Joe_Foe)) &
 X = Y.
2. **in**(spender(medium), **profiling** : *classifyUser*(U)) &
 not_in(spender(high), **profiling** : *classifyUser*(U)).

Safety

A code call $\mathbf{S} : f(d_1, \dots, d_n)$ is *safe* iff each d_i is ground.

For evaluation, code call atoms must be ground.

Problem: Given a code call condition $\chi_1 \ \& \ \dots \ \& \ \chi_n$ with variables, how to assure that evaluation of the code calls and comparisons χ_i is possible?

Example (ctd):

$\chi^{(1)}$ is “safe” (evaluate from left to write).

$\chi^{(2)}$ is not safe (X, Y uninstantiated).

The code call

in(X, **a** : *f*(Y)) & **in**(Y, **a** : *f*(2))

can be evaluated *reordering atomic cc conditions*.

Definition 5.13 (Safe Code Call (Condition))

A code call condition $\chi_1 \& \dots \& \chi_n, n \geq 1$, is *safe* iff there is a permutation π of $1, \dots, n$ such that, for $i = 1, \dots, n$:

1. If $\chi_{\pi(i)}$ is a comparison $s_1 \text{ op } s_2$, then
 - 1.1 at least one of s_1, s_2 is a constant or a variable X such that $\text{root}(X)$ belongs to

$$RV_{\pi}(i) =_{\text{def}} \{\text{root}(Y) \mid \exists j < i \text{ s.t. } Y \text{ occurs in } \chi_{\pi(j)}\};$$
 - 1.2 if s_i is neither a constant nor a variable X such that $\text{root}(X) \in RV_{\pi}(i)$, then s_i is a root variable.
2. If $\chi_{\pi(i)} = \text{in}(X_{\pi(i)}, \text{cc}_{\pi(i)})$ or $\chi_{\pi(i)} = \text{not_in}(X_{\pi(i)}, \text{cc}_{\pi(i)})$, then either $X_{\pi(i)}$ is a root variable, or $\text{root}(X_{\pi(i)})$ is from $RV_{\pi}(i)$, and the root of each variable Y occurring in $\text{cc}_{\pi(i)}$ belongs to $RV_{\pi}(i)$.

If $\chi = \chi_1 \& \cdots \& \chi_n$ is found safe, we can reorder it by a permutation π such that $\chi_{\pi(1)} \& \cdots \& \chi_{\pi(n)}$ without problems.

Checking safety of code call conditions can be efficiently done (in linear time) at compile time of a program.

Straightforward generalisation:

Definition 5.14 (Safety Modulo Variables)

Let χ be a code call condition, and let X be any set of root variables. Then, χ is *safe modulo* X , if $\chi\theta$ is safe, for any legal assignment θ of objects to the variables in X .

Note: Checking safety of a code call χ modulo X easily reduces to a check for safety.

Definition 5.15 (Code Call Solution)

Let χ be a code call condition involving the variables

$\mathbf{X} =_{def} \{X_1, \dots, X_n\}$ and let $\mathcal{S} =_{def} (\mathbf{T}_S, \mathbf{F}_S, \mathbf{C}_S)$.

A *solution* of χ w.r.t. \mathbf{T}_S in a state \mathcal{O}_S is a legal assignment of objects $\mathbf{o} = o_1, \dots, o_n$ to the variables X_1, \dots, X_n , written $\mathbf{X} := \mathbf{o}$, such that the application of the assignment makes χ true in state \mathcal{O}_S .

- $\text{Sol}(\chi)_{\mathbf{T}_S, \mathcal{O}_S}$ is the set of all solutions of the code call condition χ in state \mathcal{O}_S , and by
- $\mathcal{O}_{\text{Sol}(\chi)_{\mathbf{T}_S, \mathcal{O}_S}}$ is the set of all objects appearing in $\text{Sol}(\chi)_{\mathbf{T}_S, \mathcal{O}_S}$

(subscripts are occasionally omitted)

Comment 2 (Existence of *ins*, *del* and *upd*) We assume that $\mathcal{F}_{\mathcal{S}}$ of code package \mathcal{S} includes three functions as follows:

- ***ins* $_{\mathcal{S}}$** , which takes as input a set of objects \mathcal{O} for \mathcal{S} and a state $\mathcal{O}_{\mathcal{S}}$, and returns a new state $\mathcal{O}'_{\mathcal{S}} = \mathbf{ins}_{\mathcal{S}}(\mathcal{O}, \mathcal{O}_{\mathcal{S}})$ accomplishing the insertion of the objects in \mathcal{O} into $\mathcal{O}_{\mathcal{S}}$.
- ***del* $_{\mathcal{S}}$** , which takes as input a set of objects \mathcal{O} for \mathcal{S} and a state $\mathcal{O}_{\mathcal{S}}$, and returns a new state $\mathcal{O}'_{\mathcal{S}} =_{\text{def}} \mathbf{del}_{\mathcal{S}}(\mathcal{O}, \mathcal{O}_{\mathcal{S}})$ which describes the deletion of the objects in \mathcal{O} from $\mathcal{O}_{\mathcal{S}}$.
- ***upd* $_{\mathcal{S}}$** , which takes as input a data object o manipulated by \mathcal{S} , a field f of o , and a value v from the domain of the type of $o.f$ —this function changes the value of the $o.f$ to v .
(Can be described in terms of the preceding two functions.)

Executing the function, $\mathbf{ins}_{\text{FinanceRecord}}(\chi(X))$ where $\chi(X)$ is a code call condition involving the (sole) free variable X means:

“Insert, using a FinanceRecord insertion routine, all objects o such that $\chi(X)$ is true w.r.t. the current agent state when $X := o$.”

In such a case, the code call condition χ is used to identify the objects to be inserted, and the $\mathbf{ins}_{\text{FinanceRecord}}$ function specifies the insertion routine to be used.

Agent **a** may manage multiple data types τ_i with peculiar insertion routines **ins** _{τ_i} , $i \in \{1, \dots, n\}$.

Associate with **a** an insertion routine **ins**_{**a**} as follows:

- given either a set **O** of objects (or a code call condition $\chi(X)$ of the above type), **ins**_{**a**}($\chi(X)$, **O**_{**S**}) is a generic *method* that selects which of the insertion routines **ins** _{τ_i} , associated with the different data structures, should be invoked to accomplish the desired insertion.

Assume that an insertion function **ins**_{**a**} and a deletion function **del**_{**a**} may be associated with any agent **a** in this way.

5.4 Agent Message Box

1. Each agent's associated software code includes a special type called `Msgbox` (short for message box).
2. The message box is a buffer that may be filled (when it sends a message) or flushed (when it reads the message) by the agent.
3. In addition, we assume the existence of an operating-systems level messaging protocol (e.g., *SOCKETS* or *TCP/IP* (Wilder 1993)) that can fill in (with incoming messages) or flush (when a message is physically sent off) this buffer.

The msgbox operates on objects of the form

`(i/o,"src","dest","message","time")` .

1. `i/o` signifies an incoming or outgoing message respectively.
2. `"src"` specifies the originator
3. `"dest"` specifies the destination.
4. `"message"` is a table containing triples
 `("command", "LFlag", "Data")`,
 where `"command"` is the name of a command, `"LFlag"` a flag,
 and `"Data"` the message content (of data type Any).
5. `"time"` is the time at which the message was sent.

Message box management – some functions

- *sendMessage*(*<dest_agent>*, *<message>*):

Places tuple (*o*, "src", "dest", "message", "time") into *Msgbox*.

Parameter *o* signifies an outgoing message, and "src" is the agent at hand.

When a call of *sendMessage*("dest", "message") is executed, the state of **src**'s *Msgbox* changes by the insertion of the above tuple, denoting the sending of a message from **src** to a given destination agent **dest** with the message body "**message**".

- *getMessages()*:

Read all tuples (*i*, "src", "a", "msg", "time") from `Msgbox` of agent **a** (*i* flags an incoming message and "time" the time at which the message was received).

Note: returns *all* messages from all agents to agent **a**.

- *timedGetMessages*(<*op*>, <*valid*>):

Read all tuples $\mathbf{t} =_{def} (i, \langle src \rangle, \langle agent \rangle, \langle message \rangle, \mathbf{time})$ from `Msgbox` for which $\mathbf{t.time} \text{ } op \text{ } valid$ is true, where *op* is any of the standard comparison operators <, >, ≤, ≥, or =.

Example 5.3 (STORE Revisited)

profiling agent should classify a user U with ssn S , and needs credit information for U from the **credit** agent:

1. **profiling** sends to **credit** a message M_1 of a special format, e.g., a string "ask_provideCreditInfo_S_low," encoding the request for S 's credit information:
sendMessage(**profiling**, **credit**, M_1).
2. **credit** reads M_1 , using *getMessage*(**profiling**) (periodically, or triggered by M_1 's arrival), assembles a message M_2 and replies: *sendMessage*(**credit**, **profiling**, M_2).
3. **profiling** reads M_2 , using *getMessage*(**profiling**) (e.g., triggered by M_2 's arrival), and uses M_2 to construct the desired UserProfile.

Integrity Constraints

Recall: Each agent has an associated *agent state*, \mathcal{O} , which is a set of objects (of proper types).

- Not every set of such objects may be *legal* for \mathcal{O} .
- \mathcal{O} must satisfy axioms in general.

Definition 5.16 (Integrity Constraints \mathcal{IC})

An *integrity constraint* IC is an expression of the form

$$\psi \Rightarrow \chi$$

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

Examples:

- $S = 123_45_6789$
 \Rightarrow
 not_in(spender(low), **profiling** : *classifyUser*(S)).
- **in**(spender(medium), **profiling** : *classifyUser*(S))
 \Rightarrow
 not_in(spender(high), **profiling** : *classifyUser*(S))
- $R.sat_id = sat_1 \Rightarrow R.2dloc.x \geq 0.$
- $R1.2dloc.x = R2.2dloc.x \ \& \ R1.2dloc.y = R2.2dloc.y$
 \Rightarrow
 $R1.height = R2.height$

Integrity Constraint Satisfaction

Definition 5.17

Let \mathcal{O}_S be an agent state and $IC = \psi \Rightarrow \chi$ an integrity constraint. Then \mathcal{O}_S satisfies IC , denoted $\mathcal{O}_S \models IC$, if for every legal assignment of objects from \mathcal{O}_S to the variables in IC , either ψ is false or χ is true.

Let \mathcal{I} be a (finite) collection of integrity constraints IC , and let \mathcal{O}_S be an agent state. Then \mathcal{O}_S satisfies \mathcal{I} , denoted $\mathcal{O}_S \models \mathcal{I}$, if $\mathcal{O}_S \models IC$ for every $IC \in \mathcal{I}$.

Note: Integrity constraints are universally quantified.

Can express easily, e.g., functional dependencies in databases.

5.5 Summary and References

In order to **agentize** legacy code, we must make the most important datatypes and functions of it available.

1. The *IMPACT* **code call** mechanism abstract from given legacy code, **\mathcal{S}** , and declaratively describes its effects.
2. It provides functions **f** named **code calls**: **$\mathcal{S} : f(d_1, \dots, d_n)$** .
3. To encapsulate these functions in a logical language, we use **code call atoms**: **$\text{in}(X, \mathcal{S} : f(d_1, \dots, d_n))$** .
4. Code call atoms can be conjoined, also with comparisons, and enable **code call conditions**.
5. **Safety** ensures evaluability of code call conditions.

References

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.

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.

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

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.

Genesereth, M. R. and S. P. Ketchpel (1994). Software Agents.
Communications of the ACM 37(7), 49–53.

Subrahmanian, V., P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Özcan, and R. Ross (2000). *Heterogenous Active Agents*. MIT-Press.

Weiss, G. (Ed.) (1999). *Multi-Agent Systems*. 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.

Chapter 6. Actions and Agent Programs

6.1 Action Base

6.2 Execution and Concurrency

6.3 Action Constraints

6.4 Agent Programs: Syntax

6.5 Status Sets

6.6 Feasible Status Sets

6.7 Rational Status Sets

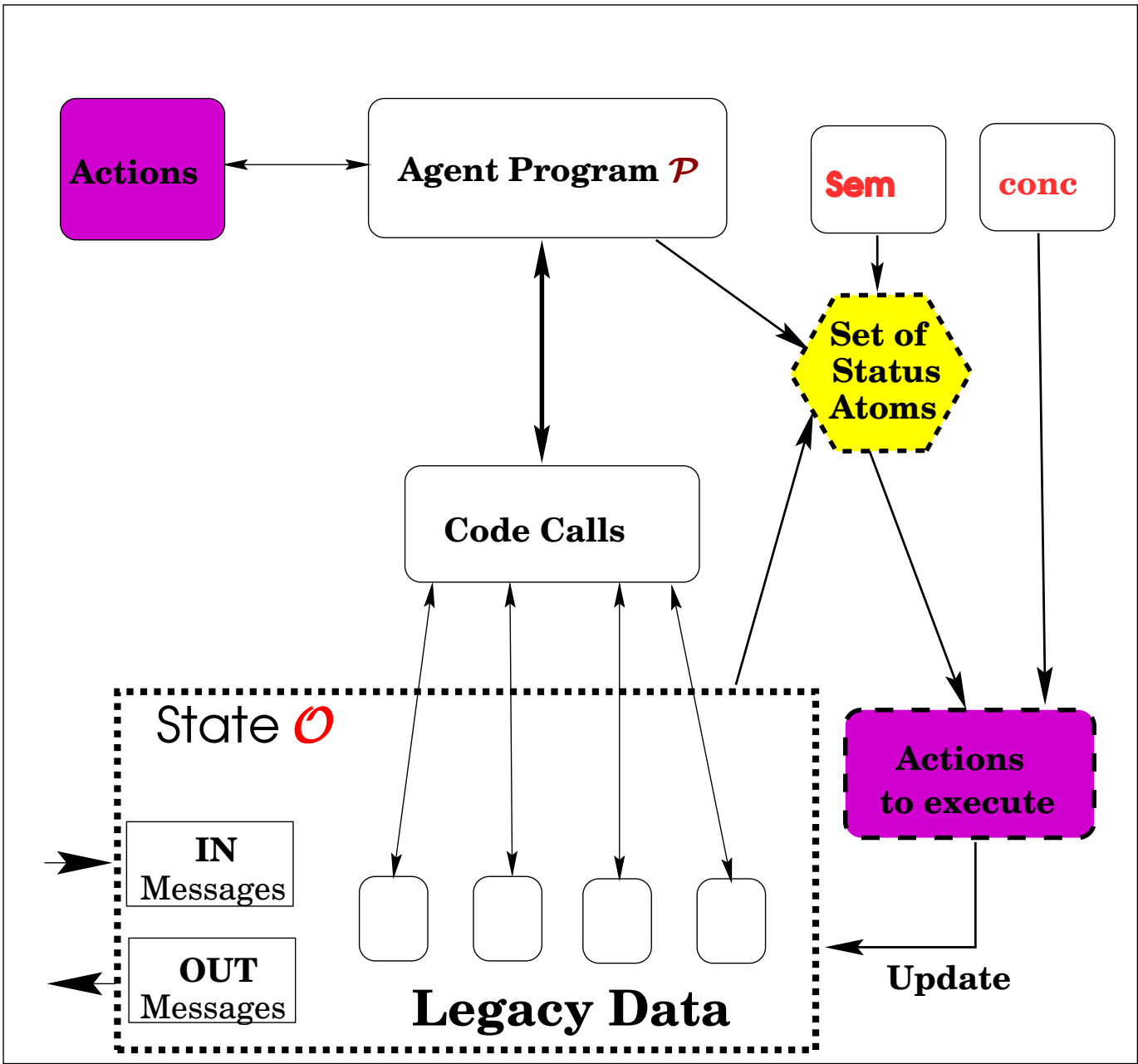
6.8 Reasonable Status Sets

Timetable:

- Chapter 6 needs 1 lecture, but without detailed discussion of the semantics.

6 Actions and Agent Programs

A Single agent



Already considered:

Underlying Software Code:

Basic set of data structures and legacy code, \mathcal{S} , on top of which the agent is built.

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** .

Integrity Constraints:

The agent has an associated finite set, \mathcal{I} .

These integrity constraints reflect the **expectations**, on the part of the designer of the agent, that the **state of the agent must satisfy**.

New

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.

Action Constraints:

Prevent the agent from concurrently executing certain actions.

Agent Programs:

An agent program is a set of rules, in a language defined below, 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 don’t’s” of the agent.**

Agent Decision Cycle

1. At time t , the agent may receive a set of new messages. They constitute a *change* to agent state.
2. The change may *trigger* some rules in the agent's associated Agent Program.

Based on the selected semantics for agent programs, the agent makes a decision on what actions to actually perform, in keeping with the rules governing its behavior encoded in its associated Agent Program.

This computation is made by executing a program called *ComputeSem* which computes the semantics of the agent.

3. The *actions* that are supposed to be performed according to the selected semantics *are then concurrently executed*, using the notion of concurrency, **conc**, selected by the agent's designer. The agent's state may (possibly) change as a consequence of the performance of such actions. In addition, the message box of other agents may also change.
4. **The cycle continues perpetually.**

Algorithm 6.1 (Agent-Decision-Cycle)

Agent-Decision-Cycle(Curr: agent_state;
 \mathcal{I} : integrity constraint set;
 \mathcal{A} : action constraint set;
 \mathcal{A} B : action base;
 conc: notion of concurrency;
 Newmsg: set of messages)

1. **while** true **do**
2. { $DoSet := \text{ComputeSem}(\text{Curr}, \mathcal{I}, \mathcal{A}, \mathcal{A}B, \text{conc}, \text{Newmsg})$;
 (★ find a set of actions to execute based on messages received ★)
3. $\text{Curr} :=$ result of executing the single action **conc**($DoSet$); }

end.

Example 6.1 (CFIT Example: Multiagent Interaction)

Every Δ units of time, the **autoPilot** agent receives a message from a **clock** agent. This message includes a “Wake” request telling the **autoPilot** agent to wake up.

The agent program of **autoPilot** causes the *wake* action to be executed, which in turn triggers other actions. These include:

- Executing an action *sendMessage*(gps, $\langle service_request \rangle$) where $\langle service_request \rangle$ of the **gps** agent requests the current plane location.
- The **gps** agent executes *getAllMsgs* and retrieves the message sent by **autoPilot**.

- The decision program of **gps** executes this request and also *sendMessage*(autoPilot, <answer>) where <answer> is the answer to the request of **autoPilot**.
- **autoPilot** executes *getAllMsgs* and retrieves the message sent by **gps**.
- The decision program of **autoPilot** checks if the plane location sent by **gps** with the one of the flight plan.
 - If yes, **autoPilot** executes the action *sleep* and goes to sleep for another Δ units of time.
 - If not, **autoPilot** executes *sendMessage*(terrain, <request>) where <request> requests the **terrain** agent to send the plane elevation at its current location (**gps**) as well as the No_go areas.

- **terrain** executes *getAllMsgs* and retrieves the message sent by **autoPilot**.
- The decision program of **terrain** executes this request and also *sendMessage*(**autoPilot**, Ans) where Ans is the answer to the request of **autoPilot**.
- **autoPilot** executes the *getAllMsgs* action and retrieves the message sent by **terrain**.
- **autoPilot** then executes *replan* with the new (correct) plane location and the terrain “no go” areas.

6.1 Action Base

Definition 6.1 (Action; Action Atom)

An *action* α consists of six components:

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

Schema: A schema, usually written as (τ_1, \dots, τ_n) , of types.

Intuitively, this says that the variable X_i must be of type τ_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 χ , called the *precondition* of the action, denoted by $Pre(\alpha)$ ($Pre(\alpha)$ must be *safe modulo the variables* X_1, \dots, 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 $\alpha(t_1, \dots, t_n)$, where t_i is a term, i.e., an object or a variable, of type τ_i , for all $i = 1, \dots, n$.

Definition 6.2 (Action Base)

An *action base*, \mathcal{AB} , is any finite collection of actions.

Item	Classical AI	<i>IMPACT</i> 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/delete lists	Via arbitrary code
Action Reasoning	Via add/delete lists	Via add list and delete list

Comment 3 *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.*

Example 6.2 (STORE Example Revisited)

The **profiling** agent might have the following action:

Name: *update_highProfile*(Ssn, Name, Profile)

Schema: (String, String, UserProfile)

Pre: **in**(spender(high), **profiling** : *classifyUser*(Ssn))

Del: **in**(⟨Ssn, Name, OldProfile⟩, **profiling** : *all*('highProfile'))

Add: **in**(⟨Ssn, Name, Profile⟩, **profiling** : *all*('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.

Example 6.3 (CFIT Revisited)

Suppose the **autoPilot** agent in the CFIT example has the following action for computing the current plane location:

Name: *compute_currentLocation*(Report)
Schema: (SatelliteReport)
Pre: **in**(Report, **msgbox** : *getVar*(Msg.Id, "Report"))
Del: **in**(OldLoc, **autoPilot** : *location***()**).
Add: **in**(NewLoc, **autoPilot** : *location***()**) &
 in(FlightRoute, **autoPilot** : *getFlightRoute***()**) &
 in(Velocity, **autoPilot** : *velocity***()**) &
in(NewLoc, **autoPilot** : *calculateLocation*(OldLoc, FlightRoute, 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.

6.2 Execution and Concurrency of Actions

What is the result of executing an action?

Definition 6.3 ((θ, γ)-Executability)

Let $\alpha(\vec{X})$ be an action and $\mathcal{S} = (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ a software code.

A ground instance $\alpha(\vec{X})\theta$ of $\alpha(\vec{X})$ is *executable* in state $\mathcal{O}_{\mathcal{S}}$, if, by definition, there exists a solution γ of $Pre(\alpha(\vec{X}))\theta$ w.r.t. $\mathcal{O}_{\mathcal{S}}$.

In this case, $\alpha(\vec{X})$ is *(θ, γ)-executable* in state $\mathcal{O}_{\mathcal{S}}$, and $(\alpha(\vec{X}), \theta, \gamma)$ is a *feasible execution triple (FET)* for $\mathcal{O}_{\mathcal{S}}$.

By $\Theta\Gamma(\alpha(\vec{X}), \mathcal{O}_{\mathcal{S}})$ we denote the set of all pairs (θ, γ) such that $(\alpha(\vec{X}), \theta, \gamma)$ is a FET in state $\mathcal{O}_{\mathcal{S}}$.

Comment 4 *Intuitively, in $\alpha(\vec{X})$, the substitution θ causes all variables in \vec{X} to be grounded.*

However, it is entirely possible that the precondition of α has occurrences of other free variables not occurring in \vec{X} .

Appropriate ground values for these variables are given by solutions of $\text{Pre}(\alpha(\vec{X})\theta)$ with respect to the current state \mathcal{O}_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.

Action Execution

Definition 6.4

Let $(\alpha(\vec{X}), \theta, \gamma)$ be a FET in state \mathcal{O}_S . Then the *result* of executing $\alpha(\vec{X})$ w.r.t. (θ, γ) is the state

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

where $\mathcal{O}_{add} = \mathcal{O}_{Sol}(Add(\alpha(\vec{X})\theta)\gamma)$ and
 $\mathcal{O}_{del} = \mathcal{O}_{Sol}(Del(\alpha(\vec{X})\theta)\gamma);$

i.e., the state resulting if first all objects in solutions of call conditions from $Del(\alpha(\vec{X})\theta)\gamma$ on \mathcal{O}_S are removed, and then all objects in solutions of call conditions from $Add(\alpha(\vec{X})\theta)\gamma$ on \mathcal{O}_S are inserted.

Concurrent Actions

Example 6.4 (Concurrency)

Consider the set of actions $ACS = \{\alpha_1, \alpha_2\}$ on an agent state \mathcal{O}_S , where

α_1 :	Pre: $\text{in}(\text{val}, \mathbf{a}:f\mathbf{0})$	α_2 :	Pre: $\text{in}(\text{val}, \mathbf{a}:f\mathbf{0})$
	Del: $\text{in}(\text{val}, \mathbf{a}:f\mathbf{0})$		Del: $\{\}$
	Add: $\{\}$		Add: $\{\}$

where $\text{in}(\text{val}, \mathbf{a}:f\mathbf{0})$ is true in \mathcal{O}_S .

Problem: Executing α_1 effects that α_2 is no longer executable.

There are many ways to resolve this. This leads to a *notion of concurrency*

Definition 6.5

A *notion of concurrency* is a function, **conc**, that takes, as input, a state \mathcal{O}_S and a set of execution triples AS , and returns, as output, a single execution triple such that:

1. if $AS = \{\alpha\}$ is a singleton action, then **conc**(\mathcal{O}_S, AS_i) = α .
2. if $AS_1 \subseteq AS_2$ and **conc**(\mathcal{O}_S, AS_i) = $(\alpha_i(\vec{X}_i), \theta_i, \gamma_i)$ for $i = 1, 2$, and α_2 is (θ_2, γ_2) -executable in state \mathcal{O}_S , then α_1 is (θ_2, γ_2) executable in state \mathcal{O}_S .

Some Notions of Concurrency

Suppose $AS = \{\alpha_1, \dots, \alpha_n\}$ is a set of actions.

- **Weakly Concurrent Execution (Naive):**
Takes add/del list of all actions α_i and executes them in parallel. (linear complexity in propositional case)
- **Sequential-Concurrent Execution:**
Take some executable sequence $\alpha_{\pi(1)}, \dots, \alpha_{\pi(n)}$.
(nondeterministic; NP-complete)
- **Full-Concurrent Execution:** Checks that every sequence $\alpha_{\pi(1)}, \dots, \alpha_{\pi(n)}$ is executable. (coNP-complete)

Other notions of concurrency might be used in *IMPACT* by the agent developer.

- Recall: **ins** and **del** are the generic insertion and deletion function

- **apply**(ACS, \mathcal{O}_S):

For any set ACS of actions, the execution of AS on \mathcal{O}_S is the execution of the set

$$\{(\alpha(\vec{X}), \theta, \gamma) \mid \alpha(\vec{t}) \in AS, \alpha(\vec{X})\theta = \alpha(\vec{t})\theta \text{ ground}, (\theta, \gamma) \in \Theta\Gamma(\alpha(\vec{X}))\}$$

of all FETs stemming from some grounded action in AS .

Then, **apply**(AS, \mathcal{O}_S) denotes the resulting state.

Definition 6.6 (Weakly Concurrent Execution)

Suppose AS is a set of FETs in the agent state \mathcal{O}_S . The *weakly concurrent execution of AS* in \mathcal{O}_S , is defined to be the agent state

$$\mathbf{apply}(AS, \mathcal{O}_S) =_{def} \mathbf{ins}(\mathcal{O}_{add}, \mathbf{del}(\mathcal{O}_{del}, \mathcal{O}_S)),$$

where

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

Definition 6.7 (Sequential-Concurrent Execution)

Let $AS =_{def} \{(\alpha_i(\vec{X}_i, \theta_i, \gamma_i)) \mid 1 \leq i \leq n\}$ be a set of FETs on state \mathcal{O}_S . Then, AS is S -concurrently executable in \mathcal{O}_S , if a permutation π of AS and a sequence of states $\mathcal{O}_S^0, \dots, \mathcal{O}_S^n$ exist where:

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

Such AS is π -executable, and \mathcal{O}_S^n is the *result of executing* $AS[\pi]$.

An action set ACS is *S-concurrently executable* on \mathcal{O}_S , if $\{(\alpha(\vec{X}), \theta, \gamma) \mid \alpha(\vec{t}) \in ACS, \alpha(\vec{X})\theta = \alpha(\vec{t})\theta \text{ ground}, (\theta, \gamma) \in \Theta\Gamma(\alpha(\vec{X}))\}$ is *S-concurrently executable* on \mathcal{O}_S .

Definition 6.8 (Full-Concurrent Execution)

Let $AS =_{def} \{(\alpha_i(\vec{X}_i, \theta_i, \gamma_i)) \mid 1 \leq i \leq n\}$ be a set of FETs and \mathcal{O}_S an agent state. Then, AS is *F-concurrently executable* in state \mathcal{O}_S , if and only if:

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

A set ACS of actions is *F-concurrently executable* on the agent state \mathcal{O}_S , if the set

$\{(\alpha(\vec{X}), \theta, \gamma) \mid \alpha(\vec{t}) \in ACS, \alpha(\vec{X})\theta = \alpha(\vec{t})\theta_{\text{ground}}, (\theta, \gamma) \in \Theta\Gamma(\alpha(\vec{X}))\}$,
is *F-concurrently executable* on \mathcal{O}_S .

Example 6.5 (Concurrency revisited)

Consider the set of actions $ACS = \{\alpha_1, \alpha_2\}$ on an agent state \mathcal{O}_S , where

α_1 :	Pre: $\text{in}(\text{val}, \mathbf{a}:f\mathbf{0})$	α_2 :	Pre: $\text{in}(\text{val}, \mathbf{a}:f\mathbf{0})$
	Del: $\text{in}(\text{val}, \mathbf{a}:f\mathbf{0})$		Del: $\{\}$
	Add: $\{\}$		Add: $\{\}$

where $\text{in}(\text{val}, \mathbf{a}:f\mathbf{0})$ is true in \mathcal{O}_S .

- weakly concurrent execution of ACS makes $\text{in}(\text{val}, \mathbf{a}:f\mathbf{0})$ false in the agent state.
- ACS is S -concurrently executable: $\pi = \alpha_2, \alpha_1$
- ACS is not F -concurrently executable: for $\pi = \text{identity}$, ACS is not S -concurrently executable.

Comment 5 *Throughout the rest of this course, we will assume that the developer of an agent has chosen some notion, **conc**, of concurrent action execution for his agent.*

*This may vary from one agent to another, but each agent uses a single notion of concurrency. Thus, when talking of an agent **a**, the phrase*

“AS is concurrently executable”

is to be considered to be synonymous with the phrase

*“AS is concurrently executable w.r.t. the notion **conc** used by agent **a**.”*

6.3 Action Constraints

In some cases, the concurrent execution of certain actions might not be desired.

Example 6.6 (STORE Example)

Reconsider the **profiling** agent.

1. If a user is classified as a high spender, then the **profiling** agent cannot execute *update_highProfile* and *update_lowProfile* concurrently.
2. The **profiling** agent cannot classify a user profile, if it is simultaneously updating the profile of that user.

Definition 6.9 (Action Constraint)

An action constraint AC is an expression of the form:

$$\{\alpha_1(\vec{X}_1), \dots, \alpha_k(\vec{X}_k)\} \leftarrow \chi \quad (6.1)$$

where $\alpha_1(\vec{X}_1), \dots, \alpha_k(\vec{X}_k)$ are action names, and χ is a code call condition.

Example 6.7 (STORE Example Revisited)

The following are action constraints for the above restrictions on the **profiling** agent:

1.
{ *update_highProfile*(Ssn1, Name1, profile),
 update_lowProfile(Ssn2, Name2, profile) } \leftarrow
 in(spender(high), **profiling** : *classifyUser*(Ssn1)) &
 Ssn1 = Ssn2 & Name1 = Name2
2.
 { *update_userProfile*(Ssn1, Name1, Profile),
 classify_user(Ssn2, Name2) } \leftarrow
 Ssn1 = Ssn2 & Name1 = Name2

Definition 6.10 (Action Constraint Satisfaction)

A set S of ground actions satisfies an action constraint

$AC : \{\alpha_1(\vec{X}_1), \dots, \alpha_k(\vec{X}_k)\} \leftarrow \chi$ on a state \mathcal{O}_S , denoted $S, \mathcal{O}_S \models AC$, if there is no legal assignment θ of objects in \mathcal{O}_S to the variables in \mathcal{AC} such that $\chi\theta$ is true and $\{\alpha_1(\vec{X})\theta, \dots, \alpha_k(\vec{X})\theta\} \subseteq S$ holds.

S *satisfies* a set \mathcal{AC} of actions constraints on \mathcal{O}_S , denoted $S, \mathcal{O}_S \models \mathcal{AC}$, if $S, \mathcal{O}_S \models AC$ for every $AC \in \mathcal{AC}$.

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

6.4 Agent Programs: Syntax

Thus far, we have introduced the following important concepts:

Software Code Calls ($\mathcal{S} : f(a_1, \dots, a_n)$): this provides a single framework within which the interoperation of diverse pieces of software may be accomplished;

Software/Agent states (\mathcal{O}_S): this describes exactly what data objects are being managed by a software package at a given point in time;

Integrity Constraints (\mathcal{IC}): this specifies exactly which software states are “valid” or “legal”;

Action Base (AB): this is a set of actions that an agent can physically execute (if the preconditions of the action are satisfied by the software state);

Concurrency Notion ($conc$): this is a function that merges together a set of actions an agent is attempting to execute into a single, coherent action;

Action Constraints (AC): this specifies whether a certain set of actions is incompatible.

Definition 6.11 (Action Status Atom)

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

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

The set $AS = \{P, F, O, W, Do\}$ is called the action status set .

Definition 6.12 (Action Rule)

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

$$\text{Op } \alpha(\vec{t}) \leftarrow L_1, \dots, L_n \quad (6.2)$$

where $\text{Op } \alpha(\vec{t})$ is an action status atom, and each of L_1, \dots, L_n is either an action status atom, or a code call atom, each of which may be preceded by a negation sign (\neg).

We require that each rule r be *safe* in the sense that:

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

-
- All variables in a rule r are implicitly universally quantified at the front of the rule.
 - A rule is *positive*, if no negation sign occurs in front of an action status atom.
 - For any rule r of the form (6.2), we denote by
 - $H(r)$, the atom in the head of r ,
 - $B(r)$, the collection of literals in the body;
 - $B^-(r)$ the negative literals in $B(r)$,
 - $B^+(r)$ the positive literals in $B(r)$,
 - $\neg.B^-(r)$ the atoms of the negative literals in $B^-(r)$.
 - Finally, the index *as* (resp., *cc*) for any of these sets denotes restriction to the literals involving action status atoms (resp., code call atoms).
-

Definition 6.13 (Agent Program)

An *agent program* \mathcal{P} is a finite collection of rules.

An agent program \mathcal{P} is *positive*, if all its rules are positive.

Example 6.8 (Simple Driving Example)

Consider an agent **driving** in an autonomous car.

Capability: Select the correct driving lane out of $\{l_lane, r_lane\}$ to go in, depending on the agent state.

- **Software code:** lane status database
Code call **status** : *free_lanes***()** returns the free lanes.
- **Actions:**
 - go_rightmost*: **Pre:** void
 Add = Del = \emptyset
 - drive*(*X*): **Pre:** **in**(*X*, **status** : *free_lanes***()**)
 Add: *go_in*(*X*)
 Del: *go_in*(*Y*)

Agent Program:

$$\begin{aligned}
 r1 : \quad & \text{O}(\textit{go_rightmost}) \leftarrow \\
 r2 : \quad & \text{O}(\textit{drive}(\textit{r_lane})) \leftarrow \text{Do}(\textit{go_rightmost}) \\
 r3 : \quad & \text{F}(\textit{drive}(X)) \leftarrow \text{not_in}(X, \text{status} : \textit{free_lanes}()) \\
 r4 : \quad & \text{Do}(\textit{drive}(\textit{l_lane})) \leftarrow \text{in}(\textit{l_lane}, \text{status} : \textit{free_lanes}()) \ \& \\
 & \text{F}(\textit{drive}(\textit{r_lane}))
 \end{aligned}$$

Note: $r1$ encodes the “Go Rightmost” norm

Agent program selects, by its semantics (introduced below), for each lane status the proper lane to go in.

6.5 Status Sets

If an agent uses an agent program \mathcal{P} , the question that the agent must answer, over and over again is:

What is the set of all action status atoms of the form $\text{Do } \alpha$ that are true with respect to \mathcal{P} , the current state, \mathcal{O}_s , the underlying set \mathcal{AC} of action constraints, and the set \mathcal{IC} of underlying integrity constraints on agent states?

This defines the set of actions that the agent must execute concurrently.

Definition 6.14 (Status Set)

A *status set* is any set S of ground action status atoms over \mathcal{S} .

For any operator $\text{Op} \in \{\text{P}, \text{Do}, \text{F}, \text{O}, \text{W}\}$, we denote by $\text{Op}(S)$ the set $\text{Op}(S) = \{\alpha \mid \text{Op}(\alpha) \in S\}$.

Informally, a status set S represents information about the status of ground actions. If some atom $\text{Op}(\alpha)$ occurs in S , then this means that the status Op is true for α .

Note: status sets are not a semantics for agent programs, but our semantics for Agent Programs build on them.

Intuitively, a “feasible” status set consists of assertions about the status of actions compatible with (but are not necessarily enforced) the rules of the agent program and the underlying action and integrity constraints.

Deontic and Action Consistency

Definition 6.15

A status set S is called *deontically consistent*, if it satisfies the following rules for every ground action α :

- If $O\alpha \in S$, then $W\alpha \notin S$
- If $P\alpha \in S$, then $F\alpha \notin S$
- If $P\alpha \in S$, then $O_S \models \exists^* Pre(\alpha)$
 $\exists^* Pre(\alpha)$... existential closure of $Pre(\alpha)$,
 This ensures that α is executable in O_S .

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

Deontic and Action Closure

Besides consistency, we also wish that the presence of certain atoms in S entails the presence of other atoms in S .

For example, if $\text{O}\textit{drive}(\text{r_lane})$ is in S , then we expect that $\text{P}\textit{drive}(\text{r_lane})$ is in S .

Definition 6.16 (Deontic Closure)

deontic closure of a status set S , denoted $\text{D-Cl}(S)$, is the closure of S under the rule

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

where α is any ground action.

S is *deontically closed*, if $S = \text{D-Cl}(S)$.

Definition 6.17 (Action Closure)

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

If $O\alpha \in S$, then $Do\alpha \in S$,

If $Do\alpha \in S$, then $P\alpha \in S$,

where α is any ground action.

Status S is *action-closed*, if $S = A-Cl(S)$.

Proposition 6.1

Let S be a status set. Then,

1. $\mathbf{A}\text{-Cl}(S) = S$ implies $\mathbf{D}\text{-Cl}(S) = S$
2. $\mathbf{D}\text{-Cl}(S) \subseteq \mathbf{A}\text{-Cl}(S)$, for all S .

A status set S which is consistent and closed is certainly a meaningful assignment of a status to each ground action.

Note that we may have ground actions α that do not occur anywhere within a status set—this means that no commitment about the status of α has been made.

The following definition specifies how we may “close” up a status set under the rules expressed by an agent program \mathcal{P} .

Definition 6.18 (Operator $\text{App}_{\mathcal{P}, \mathcal{O}_S}(S)$)

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

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

6.6 Feasible Status Sets

The above notions, combined together, give a formal notion of “feasible” status set.

Definition 6.19 (Feasible Status Set)

Let \mathcal{P} be an agent program and \mathcal{O}_S an agent state. A status set S is a *feasible status set* for \mathcal{P} on \mathcal{O}_S , if the following holds:

- (S1) $\text{App}_{\mathcal{P}, \mathcal{O}_S}(S) \subseteq S$;
- (S2) S is deontically and action consistent;
- (S3) S is action closed and deontically closed;
- (S4) $\mathcal{O}'_S \models \mathcal{I}$, where $\mathcal{O}'_S = \text{apply}(\text{Do}(S), \mathcal{O}_S)$ is the state which results after taking the actions in $\text{Do}(S)$ on \mathcal{O}_S (*state consistency*).

In general, action programs may have one, zero, or several feasible status sets.

Example 6.9 (Simple Driving Example Revisited)

Let \mathcal{O}_S be such that $\text{status} : \text{free_lanes}(\mathbf{O}) = \{l_lane, r_lane\}$.

Consider $S = \{ \text{Ogo_rightmost}, \text{Do go_rightmost}, \text{Pgo_rightmost}, \text{Odrive}(r_lane), \text{Do drive}(r_lane), \text{Pdrive}(r_lane) \}$.

- (S1):

$$\text{App}_{\mathcal{P}, \mathcal{O}_S}(S) = \{ \underbrace{\text{Ogo_rightmost}}_{\text{rule } r1}, \underbrace{\text{Odrive}(r_lane)}_{\text{rule } r2} \} \subseteq S$$

- (S2): Obviously, S is deontically and action consistent.
- (S3): $A\text{-}Cl(S) = S$, thus S is action and deontically closed.
- (S4): state consistency holds (no integrity constraints).

Example 6.10

The following (artificial) example shows that some agent programs may have no feasible status sets at all.

$$P\alpha \leftarrow$$

$$F\alpha \leftarrow$$

Clearly, if the current object state allows α to be executable, then no status set can satisfy both the closure under program rules requirement, and the deontic consistency requirement.

Some Properties of Feasible Status Sets

Proposition 6.2

Let S be a feasible status set. Then,

1. If $\text{Do}(\alpha) \in S$, then $\mathcal{O}_S \models \text{Pre}(\alpha)$;
2. If $\text{P}\alpha \notin S$, then $\text{Do}(\alpha) \notin S$;
3. If $\text{O}\alpha \in S$, then $\mathcal{O}_S \models \text{Pre}(\alpha)$;
4. If $\text{O}\alpha \in S$, then $\text{F}\alpha \notin S$.

6.7 Rational Status Sets

Feasible status sets may include status assignments that are not strictly enforced.

Example 6.11 (Simple Driving Example)

Let again \mathcal{O}_S be such that $\text{status} : \text{free_lanes}(\mathbf{0}) = \{l_lane, r_lane\}$.

Consider $S' = \{ \text{Ogo_rightmost}, \text{Do go_rightmost}, \text{Pgo_rightmost}, \\ \text{Odrive}(r_lane), \text{Do drive}(r_lane), \text{Pdrive}(r_lane), \\ \text{Pdrive}(l_lane) \}$.

- S' is a feasible status set.
- $\text{Pdrive}(l_lane)$ may be omitted (no “reason” to include it)

Principle: Each atom Op_α in S must be “founded” through (i) a rule in the agent program, or (ii) through action/deontic closure rules.

In particular, if execution of actions must be founded.

The notion of a rational status set is postulated to accommodate this kind of reasoning.

Definition 6.20 (Groundedness; Rational Status Set)

A status set S is *grounded*, if there exists no status set $S' \neq S$ such that $S' \subseteq S$ and S' satisfies conditions (S1)-(S3) of a feasible status set.

A status set S is a *rational status set*, if S is a feasible status set and S is grounded.

Example 6.12 (Driving Example Continued)

Let again \mathcal{O}_S be such that $\text{status} : \text{free_lanes}(\mathbf{O}) = \{l_lane, r_lane\}$.

The set $S = \{ \text{Ogo_rightmost}, \text{Do go_rightmost}, \text{Pgo_rightmost}, \\ \text{Odrive}(r_lane), \text{Do drive}(r_lane), \text{Pdrive}(r_lane) \}$.

is a rational status set:

- S is a feasible status set
- Rules $r1$ and r must fire in any feasible status set
- Thus, by (S1) and (S3) no smaller status set satisfying (S1)-(S3) exists.

Note: Groundedness does *not* include condition (S4) of a feasible status set.

A moment of reflection will show that omitting this condition is indeed appropriate.

Recall that the integrity constraints must be maintained when the current agent state is changed into a new one.

If we were to include the condition (S4) in groundedness, it may happen that the agent is forced to execute some actions which the program does not mention, just in order to maintain the integrity constraints.

We define, for every positive program \mathcal{P} and agent state \mathcal{O}_S , an operator $\mathbf{T}_{\mathcal{P}, \mathcal{O}_S}$ that maps a status set S to another status set.

Definition 6.21 ($\mathbf{T}_{\mathcal{P}, \mathcal{O}_S}$ Operator)

Suppose \mathcal{P} is an agent program and \mathcal{O}_S an agent state. Then, for any status set S ,

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

Lemma 6.1

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

Theorem 6.1

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

Corollary 3

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

Example 6.13 (Simple Driving Example revisited)

Clearly, the program \mathcal{P} of the **driving** agent is positive.

Let again \mathcal{O}_S be such that **status** : *free_lanes* $\mathbf{0} = \{l_lane, r_lane\}$.

$$\begin{aligned} \mathbf{T}_{\mathcal{P}, \mathcal{O}_S}(\emptyset) &= \mathbf{App}_{\mathcal{P}, \mathcal{O}_S}(\emptyset) \cup \mathbf{D-Cl}(\emptyset) \cup \mathbf{A-Cl}(\emptyset) \\ &= \{\mathbf{Ogo_rightmost}, \mathbf{Odrive}(r_lane)\} \quad (= S_1); \end{aligned}$$

$$\begin{aligned} \mathbf{T}_{\mathcal{P}, \mathcal{O}_S}(S_1) &= \mathbf{App}_{\mathcal{P}, \mathcal{O}_S}(S_1) \cup \mathbf{D-Cl}(S_1) \cup \mathbf{A-Cl}(S_1) \\ &= \{\mathbf{Ogo_rightmost}, \mathbf{Do go_rightmost}, \mathbf{Pgo_rightmost}, \\ &\quad \mathbf{Odrive}(r_lane), \mathbf{Do drive}(r_lane), \mathbf{Pdrive}(r_lane)\} \quad (= S_2); \end{aligned}$$

$$\begin{aligned} \mathbf{T}_{\mathcal{P}, \mathcal{O}_S}(S_2) &= \mathbf{App}_{\mathcal{P}, \mathcal{O}_S}(S_2) \cup \mathbf{D-Cl}(S_2) \cup \mathbf{A-Cl}(S_2) \\ &= S_2 \end{aligned}$$

S_2 coincides with the feasible status set S above. Thus, by Theorem 6.1, S is a rational status set on \mathcal{O}_S . Moreover, S is the unique rational status set.

Note: Corollary 3 is no longer true in the presence of negated action status atoms.

We observe the following property on the existence of a (not necessarily unique) rational status set.

Proposition 6.3

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

Example 6.14 (Simple Driving Example revisited)

Since $\mathcal{IC} = \emptyset$ and S is a feasible status set on the agent state, we can immediately infer that the agent program has a rational status set.

6.8 Reasonable Status Sets

Observation: For agent programs with negation, rational status sets allow logical contraposition of the program rules.

Example 6.15

Consider the following program:

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

It has two rational status sets: $S_1 = \{\text{Do}(\alpha), \text{P}(\alpha)\}$, and $S_2 = \{\text{Do}(\beta), \text{P}(\beta)\}$.

S_2 is obtained by applying the contrapositive of the rule:

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

However, S_2 seems less intuitive than S_1 as there is no explicit rule in the above program for deriving $\text{Do}(\beta)$.

We now introduce the concept of a *reasonable status set*.
Note: If contraposition is desired, then the rational status set approach rather than the reasonable status set approach should be used.

Definition 6.22 (Reasonable Status Set)

Let \mathcal{P} be an agent program, \mathcal{O}_S an agent state, and S a status set.

1. If \mathcal{P} is positive, then S is a *reasonable status set* for \mathcal{P} on \mathcal{O}_S , iff S is a rational status set for \mathcal{P} on \mathcal{O}_S .
2. The *reduct of \mathcal{P} w.r.t. S and \mathcal{O}_S* , denoted by $red^S(\mathcal{P}, \mathcal{O}_S)$, is the program resulting from the ground instances of the rules in \mathcal{P} over \mathcal{O}_S as follows.
 - (a) First, remove every rule r such that $B_{as}^-(r) \cap S \neq \emptyset$;
 - (b) Remove all atoms in $B_{as}^-(r)$ from the remaining rules.

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

Example 6.16

Consider the program \mathcal{P} :

$$\{ \text{Do } \beta \leftarrow \neg \text{Do } \alpha \}.$$

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

$$\text{red}^S(\mathcal{P}, \mathcal{O}_S) = \{ \text{Do } \beta \leftarrow \}.$$

Clearly, S is the unique rational status set of $\text{red}^S(\mathcal{P}, \mathcal{O}_S)$, and thus a reasonable status set of $\text{red}^S(\mathcal{P}, \mathcal{O}_S)$.

Hence, S is a reasonable status set of \mathcal{P} .

Proposition 6.4

Let \mathcal{P} be an agent program and \mathcal{O}_S an agent state. Then, every reasonable status set of \mathcal{P} on \mathcal{O}_S is a rational status set of \mathcal{P} on \mathcal{O}_S .

Knowledge representation

Reasonable status sets have some benefits with respect to knowledge representation. For example, the rule

$$\text{Do } \alpha \leftarrow \neg \text{F} \alpha \quad (6.3)$$

says that action α is executed *by default* (assuming its precondition succeeds), unless it is explicitly forbidden.

Moreover, reasonable (and rational) status sets are closely related to logic programming semantics:

- Reasonable status sets correspond to *answer sets* in LP.
- Rational status sets correspond to *minimal models* in LP.

6.9 Summary

This chapter was about the **decision making component** of an agent:

How to decide what actions to take given the current state of the world?

1. We introduced **actions α** .
 - (a) Much like the classical STRIPS-approach: instead of logical atoms, we consider code call atoms. Actions are implemented by code.
 - (b) How to concurrently execute actions? We assume given **conc.**
 - (c) Actions do have a status: $\{P, F, O, W, Do\}$.

2. The semantics is given by certain **status sets** of an agent program:
 - (a) An agent program consists of rules

$$\text{Op}\alpha \leftarrow \text{Op}\beta_1, \dots, \text{Op}\beta_n, \text{ccc}_1, \dots, \text{ccc}_n.$$
 - (b) A **feasible status set** is a set of status atoms
 $\{\text{Op}_1\alpha_1, \dots, \text{Op}_n\alpha_n\}$ satisfying certain properties.
 - (c) **Rational** status sets = Feasible + **Groundedness**
 - (d) **Reasonable** status sets = **Rational** + **Contraposition not allowed**

Chapter 7. Implementing Agents

7.1 Weakly Regular Agents

7.2 Regular Agents

7.3 *IADE*

7.4 The Gofish Post Office

7.5 Summary

Timetable:

- Chapter 7 needs 1 lecture.

7 Implementing Agents

7.1 Weakly Regular Agents

Issues:

1. An agent program may have no reasonable status set (RSS) because of
 - **Deontic conflicts:** e.g. $P\alpha \leftarrow$, $F\alpha \leftarrow$.
 - **Unstable negation:** e.g. $Do \leftarrow \neg Do \alpha$.
2. RSS Semantics is intractable:

We want a class of agent programs with **guaranteed** polynomially computable RSS.

⇒ Regular Agents

First step: Weakly Regular Agent Programs (*WRAPs*)

Three basic properties:

- 1. **Strong Safety:** Rules are safe, and furthermore code call conditions must some additional conditions which ensure that they *always return finite answers*.
- 2. **Conflict-Freedom:** no deontic conflicts
- 3. **Deontic Stratifiability:** graceful layering in the spirit of *stratification* in logic programs, to prevents problems with negation.

However, deontic stratification is more complex than ordinary stratification (due to deontic modalities).

7.1.1 Strong Safety

Safety is a *syntactic compile-time* check.

Safe code call conditions may lead to infinite results at run time.

Examples:

in(X, **math**: *geq*(25))

in(X, **math**: *geq*(25)) & **in**(Y, **math**: *square*(X)) & $Y \leq 2000$.

Note: Determining whether a function call has finite result is undecidable

⇒ requires additional input

Agent developer must specify a **finiteness table** FINTAB of entries

$$(\mathcal{S} : f(a_1, \dots, a_n), \langle \text{b-pattern} \rangle)$$

where $\mathcal{S} : f(a_1, \dots, a_n)$ is a code call of the underlying software code and $\langle \text{b-pattern} \rangle$ is binding pattern.

Definition 7.1 (Binding Pattern)

A *binding pattern* for a code call $\mathcal{S} : f(a_1, \dots, a_n)$ is a tuple (bt_1, \dots, bt_n) where each bt_i (called a *binding term*) is either:

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

Example 7.1 (Finiteness Table for AutoPilot Agent)

Code Call	Pattern	
autoPilot : <i>readGPSData</i> (SensorId)	(b)	
autoPilot : <i>calculateLocation</i> (Location, FlightRoute, Speed)	(b, b, b)	
autoPilot : <i>calculateNFlightRoutes</i> (CurrentLocation, No_go, N)	(b, b, 1)	
autoPilot : <i>calculateNFlightRoutes</i> (CurrentLocation, No_go, N)	(b, b, 2)	
autoPilot : <i>calculateNFlightRoutes</i> (CurrentLocation, No_go, N)	(b, b, 3)	

Note: **autoPilot** : *readGPSData*(·) and **autoPilot** : *calculateLocation*(· · ·) always return a finite number of answers.

Subsumption between binding patterns

Definition 7.2 (Ordering on Binding Patterns)

Binding pattern $B = (bt_1, \dots, bt_n)$ is *equally or less informative* than binding pattern $B' = (bt'_1, \dots, bt'_n)$, denoted $B \preceq B'$, if, by definition, for all $1 \leq i \leq n$, $bt_i \leq bt'_i$.

$B = (bt_1, \dots, bt_n)$ is *more informative* than $B' = (bt'_1, \dots, bt'_n)$, if $B' \preceq B$ but not $B \preceq B'$.

Trivial:

- (b, b, \dots, b) is unique least informative binding
- Each tuple (v_1, \dots, v_n) of objects is a maximal informative binding pattern.

Definition 7.3 (Finiteness)

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

Example 7.2 (CFIT Example)

FINTAB entails the finiteness of $\text{autoPilot}:readGPSData(5)$, but it does not entail the finiteness of $\text{autoPilot}:calculateNFlightRoutes(\langle 221, 379, 433 \rangle, \emptyset, 0)$ (since this may have an infinite number of answers),

**Achieved: finiteness of a code call
of the form $\mathcal{S} : f(\dots)$.**

More complex: strong safety of a code call **condition**.

Infinite answers due to (1) infinity of the *complementary* code call
or (2) infinitely decreasing value chains in comparisons.

Assumptions:

- 1. Function Complement.** Every function f has a complement \overline{f}
(to be considered in the finiteness table).
- 2. Downward Finiteness of types.** Type τ has the *downward finiteness property*, if it has an associated partial ordering \leq
such that for all objects x of type τ , the set $\{o' \mid o' \text{ is an object of type } \tau \text{ and } o' \leq o\}$ is finite.

Strongly Safe Actions and Programs

Definition 7.4 (Strongly Safe Action)

An action $\alpha(\vec{X})$ is *strongly safe* w.r.t. FINTAB, if $\mathbf{Pre}\alpha(\vec{X})$ is strongly safe modulo \vec{X} , and each code call from $\mathbf{Add}(\alpha(\vec{X}))$ and $\mathbf{Del}(\alpha(\vec{X}))$ is strongly safe modulo \vec{Y} , where \vec{Y} includes all root variables in \vec{X} and $\mathbf{Pre}\alpha(\vec{X})$.

Intuition: 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.

Definition 7.5 (Strongly Safe Agent Program)

A rule r is *strongly safe*, if it is safe, and $B_{cc}(r)$ is a strongly safe code call condition.

An agent program is *strongly safe*, if all rules in it are strongly safe.

Important basic notion:

Definition 7.6 (Strong Safety of Code Call Conditions)

A safe code call condition $\chi = \chi_1 \& \dots \& \chi_n$ is *strongly safe* w.r.t. root variables \vec{X} , if there is a permutation π witnessing the safety of χ modulo \vec{X} such that $\chi_{\pi(i)}$ is strongly safe modulo \vec{X} , for each $1 \leq i \leq n$.

Strong Safety of $\chi_{\pi(i)}$ modulo \vec{X} .

1. $\chi_{\pi(i)}$ is a code call atom.

Here, let the code call of $\chi_{\pi(i)}$ be $\mathcal{S}:f(\mathbf{t}_1, \dots, \mathbf{t}_n)$ and let the binding pattern

$\langle bt_1, \dots, bt_n \rangle$ be defined as follows:

- (a) If t_i is a value, then $bt_i =_{def} t_i$.
- (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, $bt_i =_{def} \mathbf{t}_i$.

Then, $\chi_{\pi(i)}$ is strongly safe if, by definition, FINTAB entails the finiteness of $\mathcal{S}:f(\mathbf{bt}_1, \dots, \mathbf{bt}_n)$.

2. $\chi_{\pi(i)}$ *is* $s \neq t$.

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* $s < t$ *or* $s \leq t$.

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* $s > t$ *or* $s \geq t$.

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

7.1.2 Conflict-Freedom

Conflict of literals $\text{Op}(\alpha(\vec{t}))$ and $(\neg)\text{Op}'(\alpha(\vec{t}'))$

Examples:

- $\mathbf{F}\alpha(a, b, X)$ and $\mathbf{P}\alpha(Z, b, c)$ conflict.
- $\neg\mathbf{P}\alpha(Z, b, c)$ and $\mathbf{Do}\alpha(Z, b, c)$ conflict,
- No conflict: $\mathbf{F}\alpha(a, b, X)$ and $\neg\mathbf{P}\alpha(Z, b, c)$, as well as $\mathbf{P}\alpha(Z, b, c)$ and $\neg\mathbf{Do}\alpha(Z, b, c)$.

Deontic conflicts might “kill” any feasible status set.
 \Rightarrow Enforce **deontic consistency** syntactically

Two types of deontic conflicts:

1. rule-head conflicts:

Rules r, r' conflict, if their heads conflict and their bodies are satisfiable without conflict.

Problem: *undecidable* over all agent states

\Rightarrow Use sound (but incomplete)

conflict-freedom test **cft**

(*IMPACT* supports several **cft**'s)

2. Intra-rule conflict:

Rule $r : \text{Op}_i(\alpha(\vec{t})) \leftarrow \dots, (\neg)\text{Op}_j(\alpha(\vec{t}')), \dots$ has a conflict, if $\text{Op}_i(\alpha(\vec{t}))$ and $(\neg)\text{Op}_j(\alpha(\vec{t}'))$ conflict.

Such conflicts are easily checked.

Definition 7.7 (Conflicts)

Literal $L_i = \text{Op}(\alpha(\vec{t}))$ *conflicts with* $L_j = (\neg)\text{Op}'(\alpha(\vec{t}'))$, if

- 1. $\alpha(\vec{t})$ and $\alpha(\vec{t}')$ are unifiable,
- 2. the modalities Op and $(\neg)\text{Op}'$ conflict, i.e., there is an entry “ \times ” in the following table:

$\text{Op} \setminus (\neg)\text{Op}'$	P	$\neg\text{P}$	F	$\neg\text{F}$	O	$\neg\text{O}$	W	$\neg\text{W}$	Do	$\neg\text{Do}$
P		\times	\times							
F	\times			\times	\times				\times	
O		\times	\times			\times	\times			\times
W					\times			\times		
Do		\times	\times							\times

Note: • conflicts-with is symmetric on Op and Op' .
• conflicts are *not* preserved under negation.

Conflicting Rules:

Informally, two rules r_i and r_j conflict in a given state, if

- they have a unifiable head
- conflicting head-modalities,
- deontically consistent bodies (under the unifying substitution),
and
- their bodies' code call components must have a solution.

Definition 7.8 (Conflicting Rules w.r.t. a State)

Rules

$$r_i : \text{Op}_i(\alpha(\vec{t})) \leftarrow B(r_i)$$

$$r_j : \text{Op}_j(\beta(\vec{t}')) \leftarrow B(r_j)$$

(standardised apart) **conflict** w.r.t. an agent state \mathcal{O}_S , if Op_i conflicts with Op_j , and there is a substitution θ such that:

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

Definition 7.9 (Conflict Free Agent Program)

An agent program, \mathcal{P} , is *conflict free*, if

1. There are no conflicting rules r_i, r_j in \mathcal{P} , for every possible agent state \mathcal{O}_S (no *rule-head conflicts*);
2. For any rule $\text{Op}_i(\alpha(\vec{t})) \leftarrow \dots, (\neg)\text{Op}_j(\vec{t}'), \dots$ in \mathcal{P} , $\text{Op}_i(\alpha(\vec{t}))$ and $(\neg)\text{Op}_j(\alpha(\vec{t}'))$ do not conflict (no *intra-rule conflicts*).

Problem: rule-head conflicts are undecidable.

Remedy: use incomplete rule-head conflicts checks.

Definition 7.10 (Conflict-Freedom Test)

A *conflict-freedom test* is a function

$$\mathbf{cft} : Rules \times Rules \rightarrow \{\mathbf{true}, \mathbf{false}\}$$

such that if $\mathbf{cft}(r_1, r_2) = \mathbf{true}$, then r_1, r_2 does not conflict w.r.t. any agent state \mathcal{O}_S .

Definition 7.11 (Conflict-Free Agent Program w.r.t. \mathbf{cft})

An agent program \mathcal{P} is *conflict free w.r.t. \mathbf{cft}* , if

- $\mathbf{cft}(r_i, r_j) = \mathbf{true}$, for every distinct $r_i, r_j \in \mathcal{P}$, and
- \mathcal{P} has no intra-rule conflicts.

Note: Different functions **cft** possible.

Tradeoff between accuracy and complexity:

The more accurate **cft**, i.e., less often returns “false” on arguments (r_i, r_j) when in fact r_i, r_j do not conflict, the higher is the complexity of **cft**.

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 Conflict-Freedom Tests

Example 7.3 (Head-CFT, \mathbf{cft}_h)

Let r_i, r_j be two rules of the form

$$\begin{aligned} r_i : \text{Op}_i(\alpha(\vec{t})) &\leftarrow B(i) \\ r_j : \text{Op}_j(\beta(\vec{t}')) &\leftarrow B(j). \end{aligned}$$

Now let the head conflict-freedom test \mathbf{cft}_h be as follows,

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

Example 7.4 (Body Code Call CFT, cft_{bcc})

Let the body-code conflict-freedom test cft_{bcc} be as follows:

$$\text{cft}_{bcc}(r_i, r_j) = \begin{cases} \text{true,} & \text{if either (1) } \text{Op}_i, \text{Op}_j \text{ do not conflict, or} \\ & \text{(2) } \alpha(\vec{t}) \text{ and } \beta(\vec{t'}) \text{ are not unifiable, or} \\ & \text{(3) } \text{Op}_i, \text{Op}_j \text{ conflict and (3.1) } \alpha(\vec{t}), \beta(\vec{t'}) \text{ are} \\ & \text{unifiable via mgu } \theta \text{ and (3.2) } B_{cc}(r_1\theta) \\ & \quad B_{cc}(r_2\theta) \text{ has a pair of contradictory cc atoms;} \\ \text{false} & \text{otherwise.} \end{cases}$$

Condition (3.2) means that code call atoms **in**(X, cc) and **not_in**(X, cc) occur in $B_{cc}(r_1\theta) \cup B_{cc}(r_2\theta)$, or comparison atoms $s_1 = s_2$ and $s_1 \neq s_2$; $s_1 < s_2$ and $s_1 \geq s_2$ etc.

Example 7.5 (Body-Modality-CFT, \mathbf{cft}_{bm})

Similar to the previous one, except that action status atoms are considered instead. \mathbf{cft}_{bm} be as follows,

$$\mathbf{cft}_{bm}(r_i, r_j) = \begin{cases} \text{true} & \text{if } \text{Op}_i, \text{Op}_j \text{ do not conflict or} \\ & \alpha(\vec{t}), \beta(\vec{t}') \text{ are not unifiable or} \\ & \text{Op}_i, \text{Op}_j \text{ conflict, and } \alpha(\vec{t}), \beta(\vec{t}') \text{ are unifiable} \\ & \text{via mgu } \theta \text{ and literals } (\neg)\text{Op}_i \alpha(\vec{t}'') \text{ in } B_{as}(r_i \theta) \\ & \text{for } i = 1, 2 \text{ exist such that } (\neg)\text{Op}_1 \text{ and } (\neg)\text{Op}_2 \\ & \text{conflict;} \\ \text{false} & \text{otherwise.} \end{cases}$$

Example 7.6 (Precondition-CFT, cft_{pr})

Often, we might have action status atoms of the form

$P\alpha$, $\text{Do}\alpha$, $O\alpha$ in a rule.

Simple Driving Scenario:

$$\begin{aligned} r1 : \quad & O(\text{go_rightmost}) \leftarrow \\ r2 : \quad & O(\text{drive}(\text{r_lane})) \leftarrow \text{Do}(\text{go_rightmost}) \\ r3 : \quad & F(\text{drive}(X)) \leftarrow \text{not_in}(X, \text{status} : \text{free_lanes}()) \\ r4 : \quad & \text{Do}(\text{drive}(\text{l_lane})) \leftarrow \text{in}(\text{l_lane}, \text{status} : \text{free_lanes}()) \ \& \\ & F(\text{drive}(\text{r_lane})) \end{aligned}$$

Note: no intra-rule conflicts.

$\text{cft}_{bcc}(r_3, r_4) = \text{cft}_{bm}(r_3, r_4) = \text{true}$ but $\text{cft}_{bm}(r_2, r_3) = \text{false}$.

Let r_i^* be r_i augmented with **Pre**(α) of any atom **P** α , **Do** α , **O** α (standardised apart) in r_i .

$$\begin{aligned}
 r1 : \quad & \text{O}(\textit{go_rightmost}) \leftarrow \\
 r2 : \quad & \text{O}(\textit{drive}(\text{r_lane})) \leftarrow \text{Do}(\textit{go_rightmost}) \ \& \\
 & \quad \text{in}(\text{r_lane}, \text{status} : \textit{free_lanes}()) \\
 r3 : \quad & \text{F}(\textit{drive}(X)) \leftarrow \text{not_in}(X, \text{status} : \textit{free_lanes}()) \\
 r4 : \quad & \text{Do}(\textit{drive}(\text{l_lane})) \leftarrow \text{in}(\text{l_lane}, \text{status} : \textit{free_lanes}()) \ \& \\
 & \quad \text{F}(\textit{drive}(\text{r_lane}))
 \end{aligned}$$

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

Then, $\text{cft}_{pr}(r_2, r_3) = \text{true}$, and \mathcal{P} is found head-conflict free.

7.1.3 Deontic Stratification

Extend the notion of stratified logic program (Apt *et al.*)

Definition 7.12 (Layering Function)

Let \mathcal{P} be an agent program. A *layering function* ℓ is a mapping

$$\ell : \mathcal{P} \rightarrow \mathcal{N} (= \{0, 1, 2, \dots\})..$$

The i -th layer of \mathcal{P} w.r.t. ℓ , denoted \mathcal{P}_i^ℓ , is

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

Intuition: Evaluate layer i before layer j , if $i < j$.

Note: Drop superscript ℓ if clear from context.

Example 7.7 (Simple Flight Program)

```
r1: Do execute_flight_plan(F_route) ←  
      in(automated, autoPilot: pilotStatus(pilot_message)),  
      Do create_flight_plan(No_go, F_route, C_loc)  
  
r2: O create_flight_plan(No_go, F_route, C_loc) ←  
      O adjust_course(No_go, F_route, C_loc)  
  
r3: O maintain_course(no_go, flight_route, current_location) ←  
      in(automated, autoPilot: pilotStatus(pilot_message)),  
      ¬ O adjust_course(no_go, flight_route, current_location)  
  
r4: O adjust_course(no_go, flight_route, current_location) ←  
      O adjustAltitude(Altitude)
```

Simplification: Rules use constant valued parameters for *maintain_course* and *adjust_course*.

Layering functions:

- ℓ_1 : $\ell_1(r_4) = 0, \ell_1(r_2) = 1, \ell_1(r_3) = 1, \ell_1(r_1) = 2$.

Program layers:

$$\mathcal{P}_0^{\ell_1} = \{r_4\}, \mathcal{P}_1^{\ell_1} = \{r_2, r_3\}, \mathcal{P}_2^{\ell_1} = \{r_1\}.$$

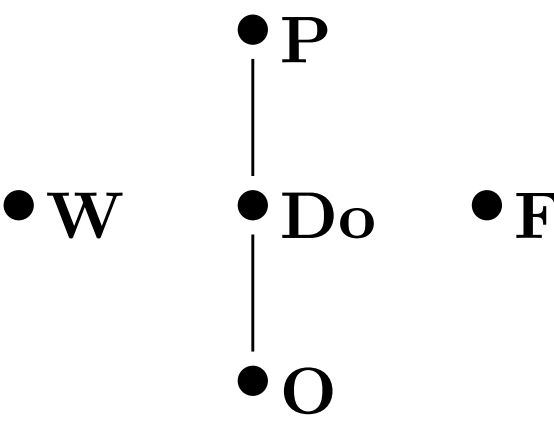
- ℓ_2 : $\ell_2(r_4) = 0, \ell_2(r_i) = 1, i \in \{1, 2, 3\}$.

- ℓ_3 : $\ell_3(r_i) = 0, i \in \{1, 2, 3, 4\}$.

deontic stratification: consequences in a sound setting

Definition 7.13 (Modality Ordering)

Partial ordering " \leq " on $M = \{P, O, Do, W, F\}$: $O \leq Do$,
 $O \leq P$, $Do \leq P$, and $Op \leq Op'$, for each $Op \in M$.



Ground action status atoms: $Op \alpha \leq Op' \alpha$ if $Op' \leq Op$.

Definition 7.14 (Deontically Stratifiable Agent Program)

An agent program \mathcal{P} is *deontically stratifiable*, if there exists a layering function ℓ such that:

1. For every rule $r_i : \text{Op}_i(\alpha(\vec{t})) \leftarrow \dots, \text{Op}_j(\beta(\vec{t}')), \dots$ in \mathcal{P}_i^ℓ , if $r : \text{Op}(\beta(\vec{t}'')) \leftarrow \dots$ is a rule in \mathcal{P} such that $\beta(\vec{t}')$ and $\beta(\vec{t}'')$ are unifiable and $\text{Op} \leq \text{Op}_j$, then $\ell(r) \leq \ell(r_i)$.
2. For every rule $r_i : \text{Op}_i(\alpha(\vec{t})) \leftarrow \dots, \neg \text{Op}_j(\beta(\vec{t}')), \dots$ in \mathcal{P}_i^ℓ , if $r : \text{Op}(\beta(\vec{t}'')) \leftarrow \dots$ is a rule in \mathcal{P} such that $\beta(\vec{t}')$ and $\beta(\vec{t}'')$ are unifiable and $\text{Op} \leq \text{Op}_j$, then $\ell(r) < \ell(r_i)$.

Such an ℓ is called a *witness* to the stratifiability of \mathcal{P} . $\text{wtn}(\mathcal{P}) \dots$
The set of all such witnesses.

Example 7.8 (Deontic Stratifiability)

Simple Flight Program:

Condition 1) of deontic stratifiability requires

- $\ell(r_2) \leq \ell(r_1)$
- $\ell(r_4) \leq \ell(r_2)$.

Condition 2) requires

- $\ell(r_4) < \ell(r_3)$.

$\Rightarrow \ell_1$ and ℓ_2 witness stratifiability of \mathcal{P} .

ℓ_3 is not a witness of stratifiability

Example 7.9 (Unstratifiable Program)

Let \mathcal{P}' contain the following rule:

$$r'_1: \text{Do } \textit{compute_currentLocation}(\text{report}) \leftarrow \\ \neg \text{Do } \textit{compute_currentLocation}(\text{report})$$

Condition 2) requires

- $\ell(r'_1) < \ell(r'_1)$

This is impossible \mathcal{P}' is not deontically stratifiable.

7.1.4 Definition of Weak Regularity

Definition 7.15 (Weak Regular Agent Program (*WRAP*))

Let \mathcal{P} be an agent program, FINTAB a finiteness table, and cft a conflict-freedom test. Then, \mathcal{P} is a weak regular agent program (*WRAP*) w.r.t. FINTAB and cft , if the following holds:

Strong Safety:

All rules in \mathcal{P} and actions α 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.

Example 7.10 (Sample *WRAP*)

Simple Flight Program: Suppose that actions are strongly safe w.r.t. some FINTAB.

- \mathcal{P} is conflict-free under cft_h ;
- \mathcal{P} is deontically stratified (see above)

\Rightarrow program \mathcal{P} is a *WRAP*.

Add the following rule:

r_5 : W $\text{create_flight_plan}(no_go, \text{flight_route}, \text{current_location}) \leftarrow$
 $\text{not_in}(\text{automated}, \text{autoPilot} : \text{pilotStatus}(\text{pilot_message}))$

$\text{cft}_h(r_2, r_5) = \text{false}$, and so \mathcal{P} is no longer a *WRAP*.

Finally, we arrive at weak regular agents:

Definition 7.16 (Weakly Regular Agent)

An agent **a** is *weakly regular*, if

- its associated agent program **P** is weakly regular

and

- the action constraints **AC**,
- the integrity constraints **IC**, and
- the notion of concurrency **conc**

in the background are all strongly safe.

Strong safety for constraints and concurrency notion

Definition 7.17 (Strongly Safe Integrity/Action Constraints)

An integrity constraint $\psi \Rightarrow \chi$ is *strongly safe*, if ψ is strongly safe and χ is strongly safe modulo the root variables in ψ .

An action constraint $\{\alpha_1(\vec{X}_1), \dots, \alpha_k(\vec{X}_k)\} \leftarrow \chi$ is *strongly safe* if and only if χ is strongly safe.

Definition 7.18 (Strongly Safe Notion of Concurrency)

A notion of concurrency **conc** is *strongly safe*, if for every set \mathcal{A} of actions, if all members of \mathcal{A} are strongly safe, then so is **conc**(\mathcal{A}).

Properties of Weakly Regular Agents

- Every *WRAP* (in fact, deontically stratifiable agent program) \mathcal{P} has a *canonical layering*, given by

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

- Every *WRAP* has **either one or no** reasonable status set.
- Any *WRAP*, if integrity and action constraints are discarded, has a reasonable status set.
- Every *WRAP* has an associated fixpoint computation method, which computes the unique reasonable status set:
 $\emptyset = S_0 \subseteq S_1 \subseteq \dots$.

7.2 Regular Agents

Problem:

- agent programs (including *WRAPs*) admit *recursion*

$\text{Do } \alpha \leftarrow \text{Do } \beta, \quad \text{Do } \beta \leftarrow \text{Do } \gamma.$

- unbounded recursion

$\text{Do } (\textit{send}(N1)) \leftarrow \text{Do } (\textit{send}(N)) \ \& \ \textit{in}(\text{N1}, \textit{math} : \textit{int_Add}(\text{N}, 1)).$

Leads to infinite status set!

For agent programs, *bounded* recursion is plausible

Unfolding:

$$\frac{\text{Do } \alpha \leftarrow \mathbf{P} \beta \& Body_1 \quad \mathbf{P} \beta \leftarrow Body_2}{\text{Do } \alpha \leftarrow Body_1 \& Body_2}$$

- Operator $\text{Unfold}_{\mathcal{P}}$: unfold all positive rules bodies in \mathcal{P} .
- Eliminate all **positive rule bodies** by repeated unfolding.

Technical realisation

- Associate with $\text{Op}_\alpha(X)$ a *prerequisite-free constraint (PFC)*
 pfc
PFCs: $\{\&, \vee\}$ -closure of code call conditions χ and negative status literals $\neg \text{Op}_\alpha$
- Define semantic equivalence of $\text{pfc}_1, \text{pfc}_2$ over all agent states
 \mathcal{O}_S
Note: Equivalence of $\text{pfc}_1, \text{pfc}_2$ is undecidable in general
- Use a sound (but incomplete) *PFC-equivalence test*
 $\text{eqi}(\text{pfc}_1, \text{pfc}_2)$

Definition 7.19 (Regular Agent Program)

A *regular agent program (RAP)* is a program which is weakly regular and **bounded**.

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, i.e.,

$$\text{eqi}(\text{Unfold}_{\mathcal{P}}^b, \text{Unfold}_{\mathcal{P}}^{b+1}) = \text{true}$$

for some b .

b -regular RAP: $\text{eqi}(\text{Unfold}_{\mathcal{P}}^b, \text{Unfold}_{\mathcal{P}}^{b+1}) = \text{true}$.

Refinement:

- Unfolding along levels of \mathcal{P} under deontic stratification ℓ ;
- Use PFC-equivalence test $\text{eqi}^{(i)}$ at layer i .

Definition 7.20 (Regular Agent)

An agent is *regular* w.r.t. a layering ℓ and a suite of PFC equivalence tests $\text{eqi}^{(i)}$, if it is weakly regular and its associated agent program is b -regular w.r.t. ℓ and the $\text{eqi}^{(i)}$, for some $b \geq 0$.

Note: Fix $\text{eqi}^{(i)}$, b **at compile time**: accept/reject \mathcal{P} after $\leq b$ unfolding steps.

Theorem 7.1 (Regular Agent Algorithm)

There is an algorithm, **Reasonable_SS**($\mathcal{P}, \ell, \mathcal{IC}, \mathcal{AC}, \mathcal{O}_S$), which given a RAP \mathcal{P} , a layering $\ell \in \text{wt}(\mathcal{P})$, strongly safe action constraints \mathcal{AC} and integrity constraints \mathcal{IC} , and an agent state \mathcal{O}_S , computes in finite time the reasonable status set S of \mathcal{P} on \mathcal{O}_S , if one exists, and “no” otherwise.

Under further conditions, **Reasonable_SS** is polynomial:

- Every ground code call $\mathcal{S} : f(d_1, \dots, d_n)$, has a set of solutions computed in polynomial time;
- no occurrence of a variable in \mathbf{a} ’s description is “loose”.
- assembling and executing **conc**(Do (S), \mathcal{O}_S) is possible in polynomial time.

7.3 IADE

Implementation of the regular agent program paradigm:

IMPACT Agent Development Environment (*IADE*)

Two major parts:

1. Agent Building: Specification part

Easy to use, network accessible GUI for agent program development:

- software code: data types, functions
- actions
- rules
- ...

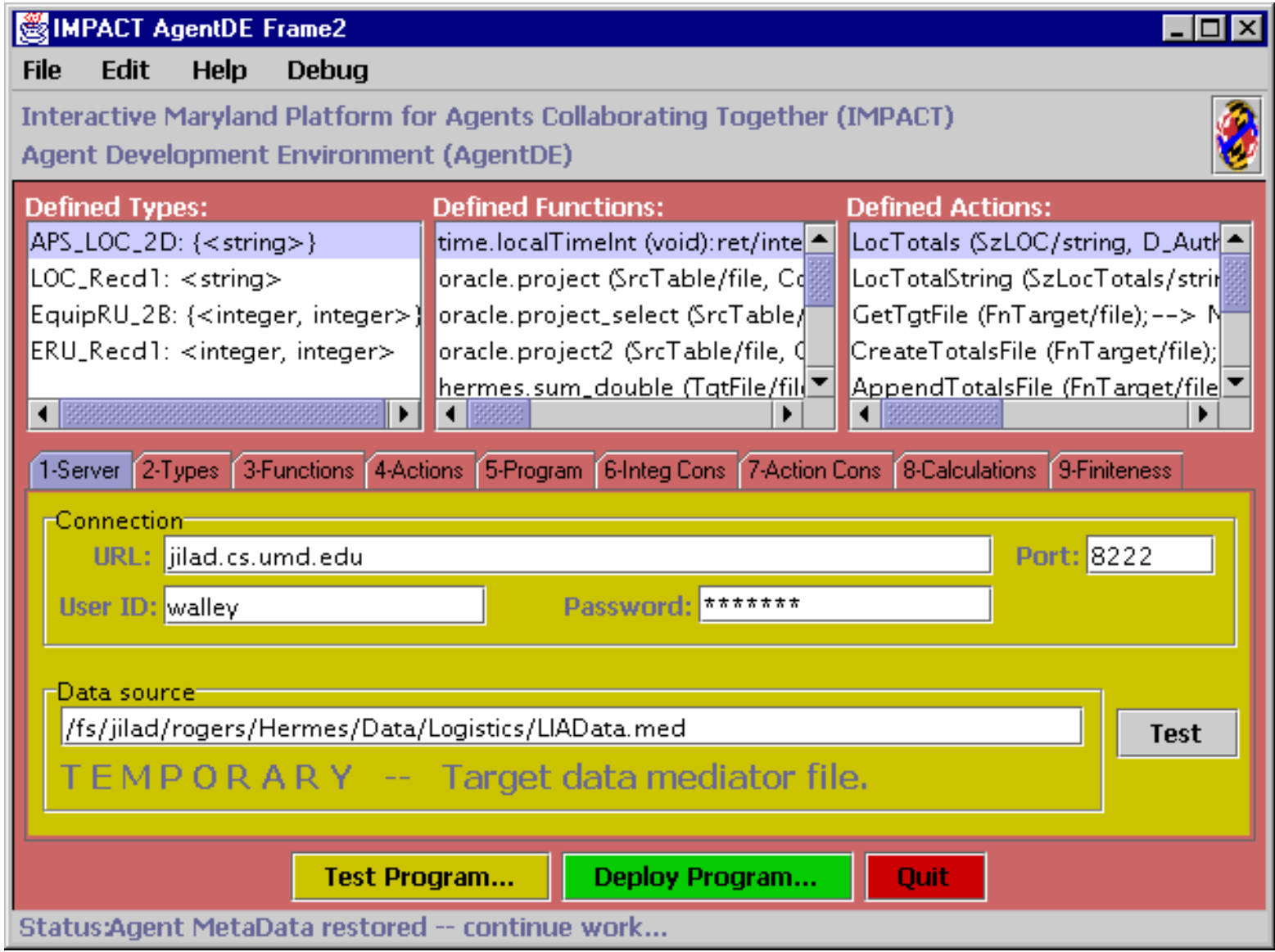
2. **Agent Testing:** Execution Part

Support for compilation and testing of *RAPs*

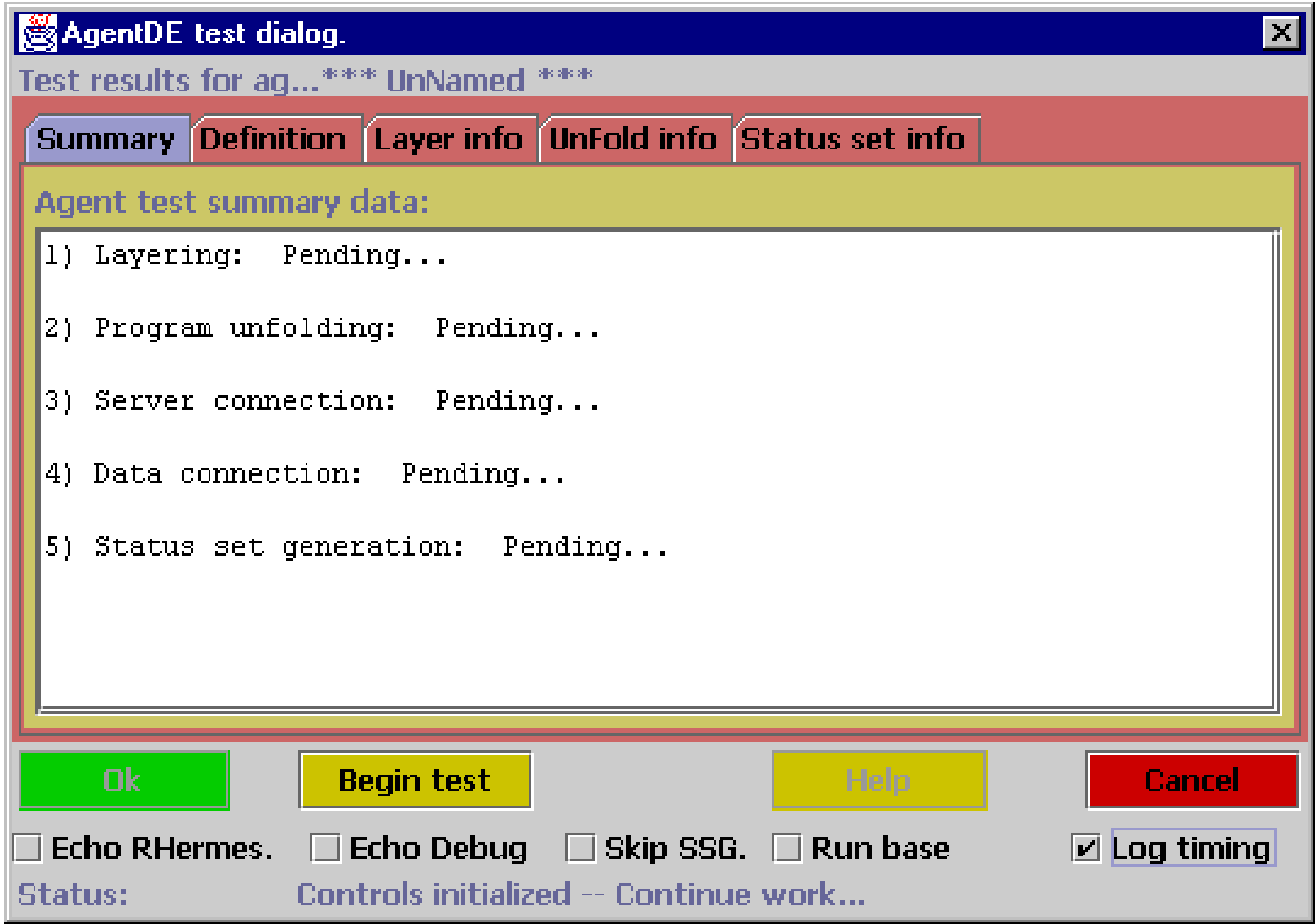
- conflict freedom test
- finiteness table
- program unfolding
- status set computation

Allows to view the reasonable status set of an agent program on the current agent state.

IADE Main Screen



IADE Test Dialog Screen (Prior to Program Testing)

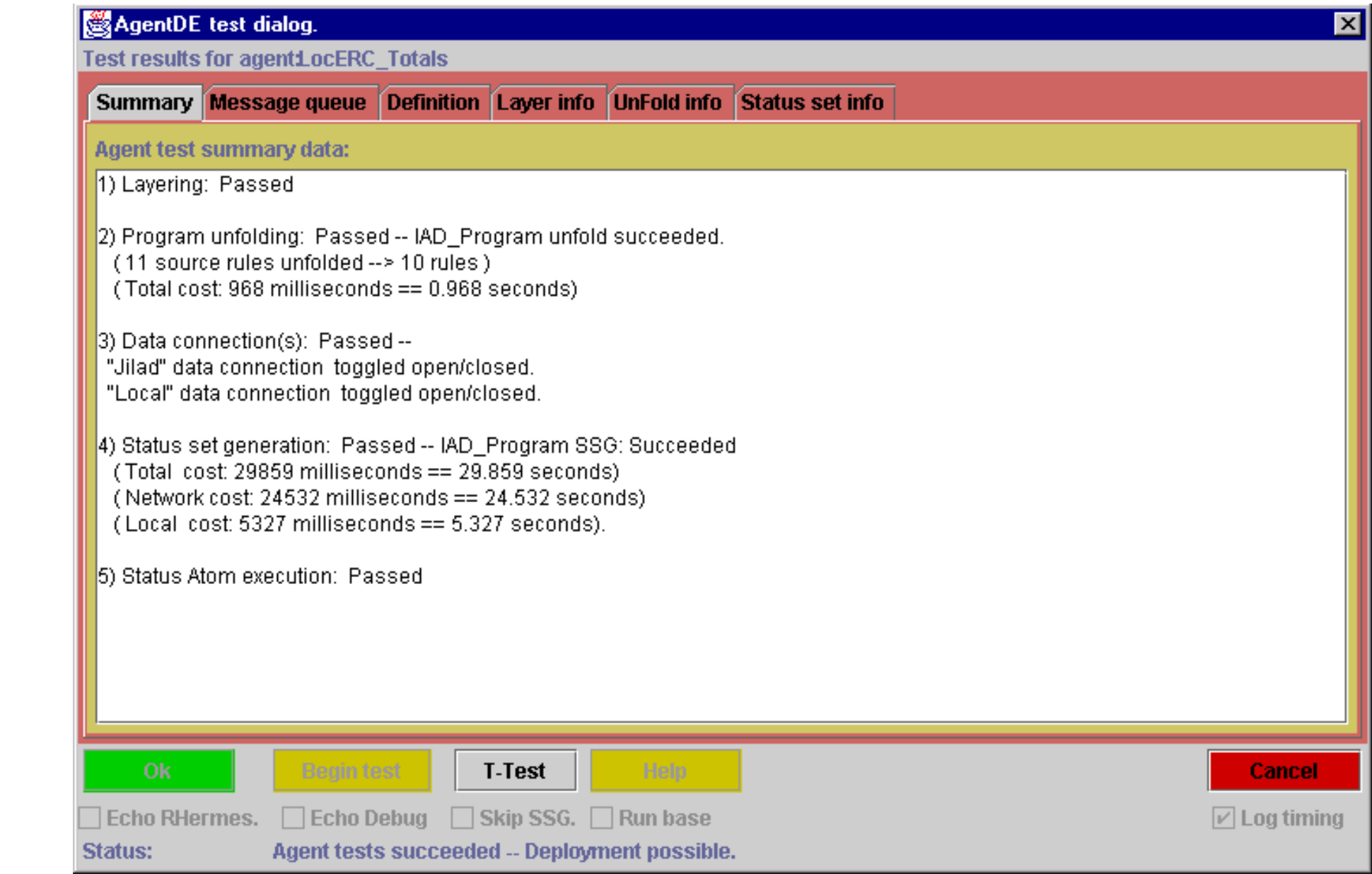


*IAD*E includes algorithms for checking

- safety
- strong safety
- conflict freedomness
- whether an agent program is a *WRAP*.

*IAD*E implements unfolding of *WRAP*s (currently, supported for positive agent programs only).

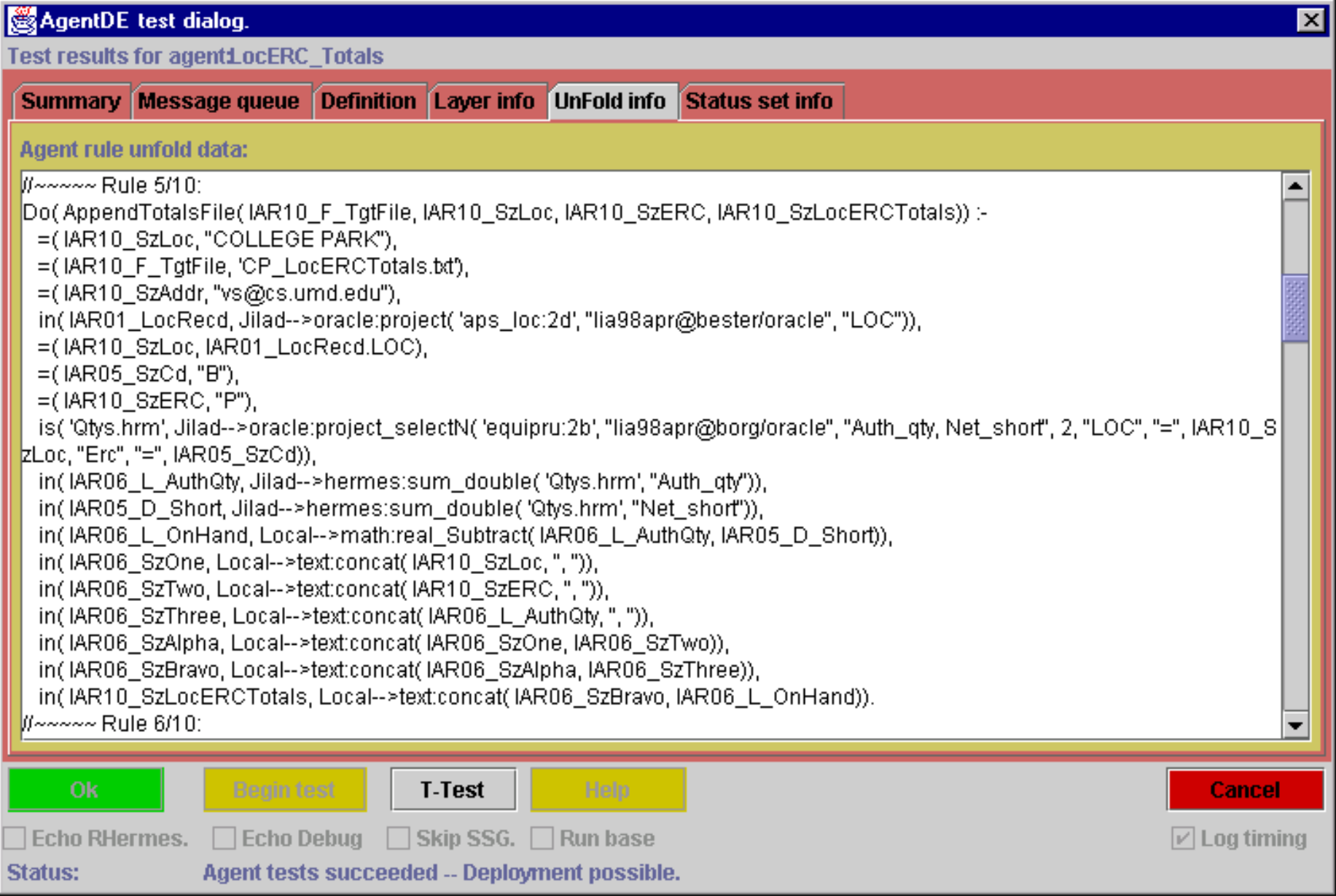
IADE Test Execution Screen

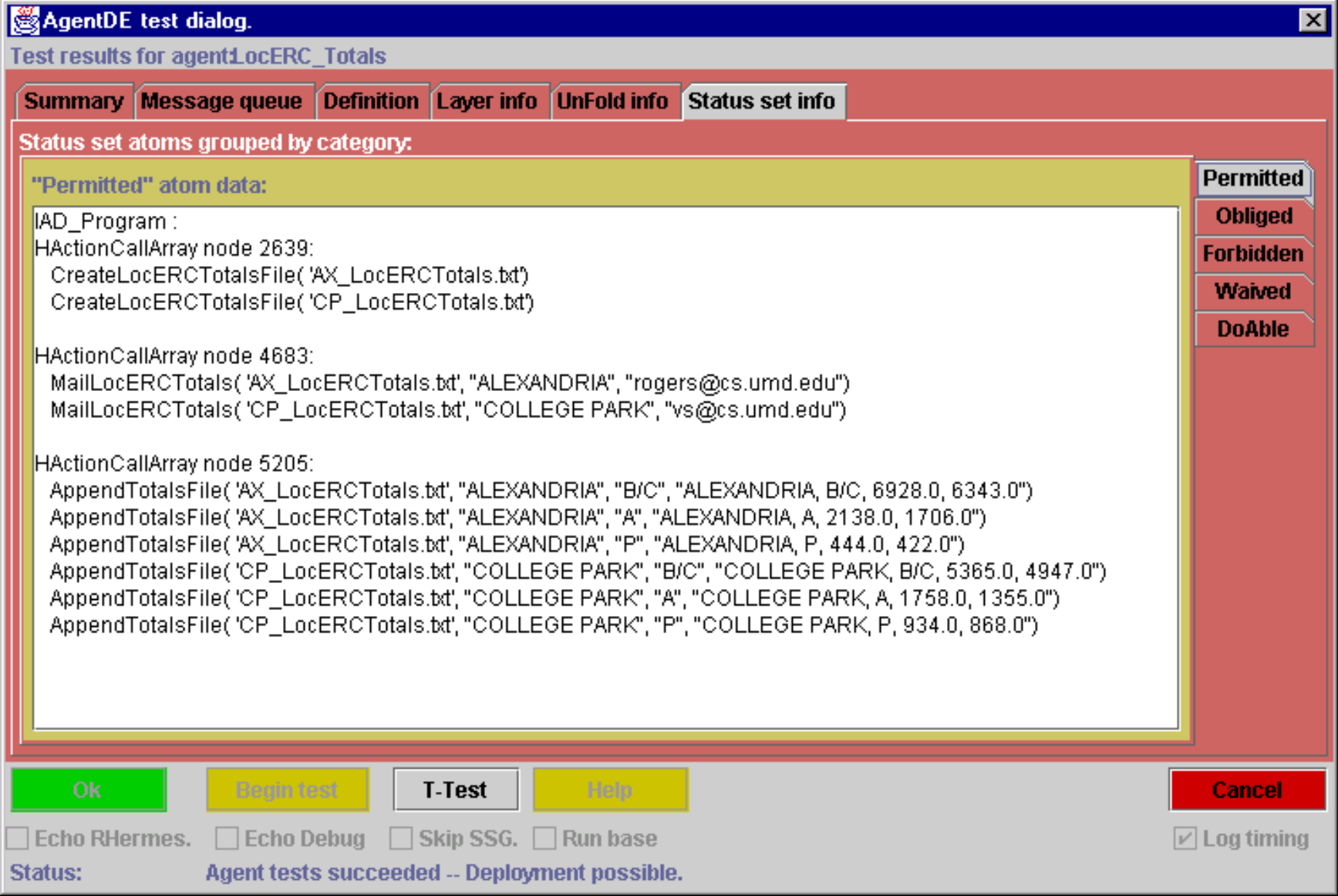


After successful test execution phase, the reasonable status sets is generated.

Options to continue:

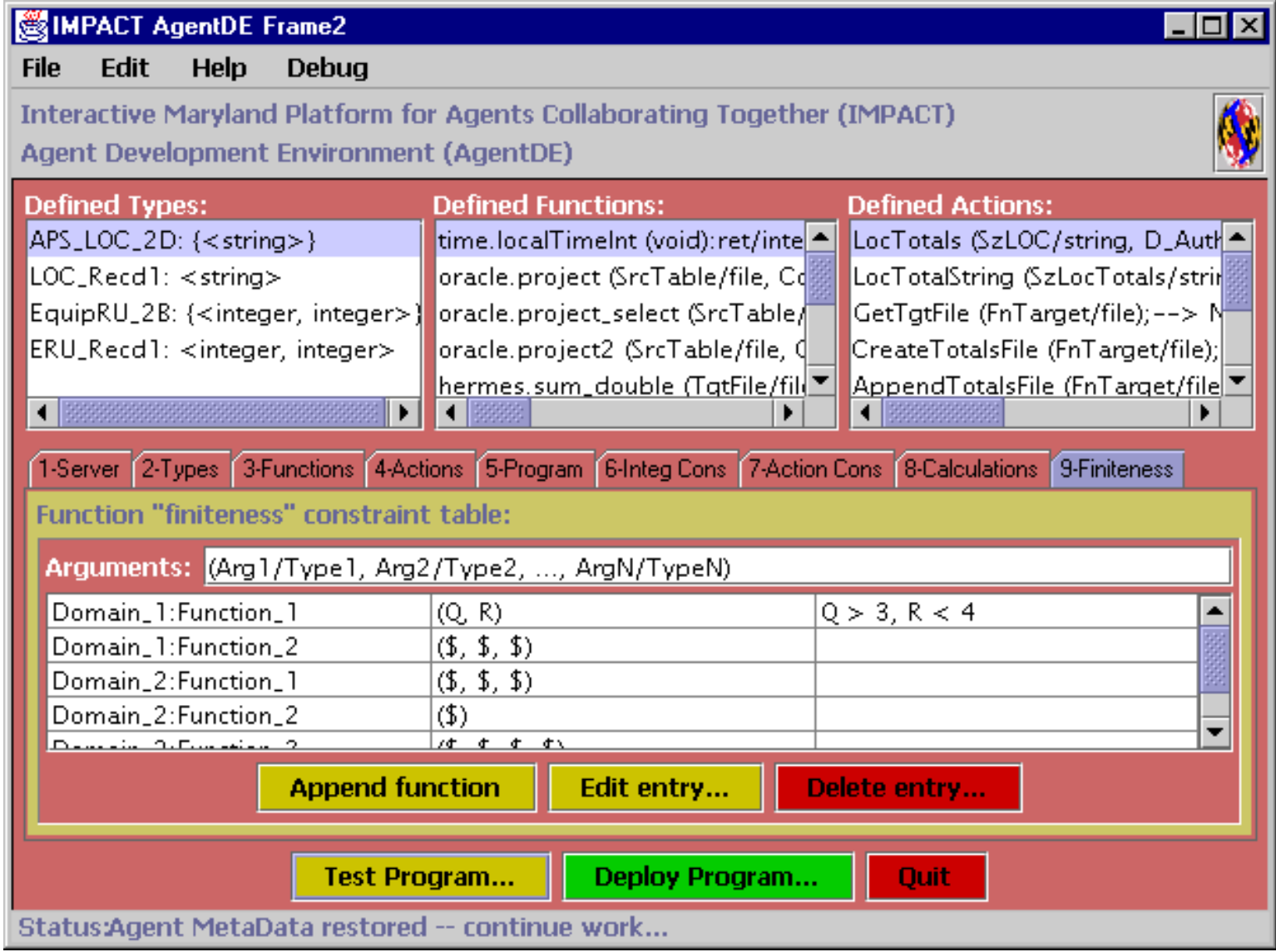
- **“Unfold Info”:**
Shows the unfolded program.
- **“Layer Info”:**
Show the layers of the agent program.
- **“Status Set Info”:**
Shows the status set (split in deontic modality parts).



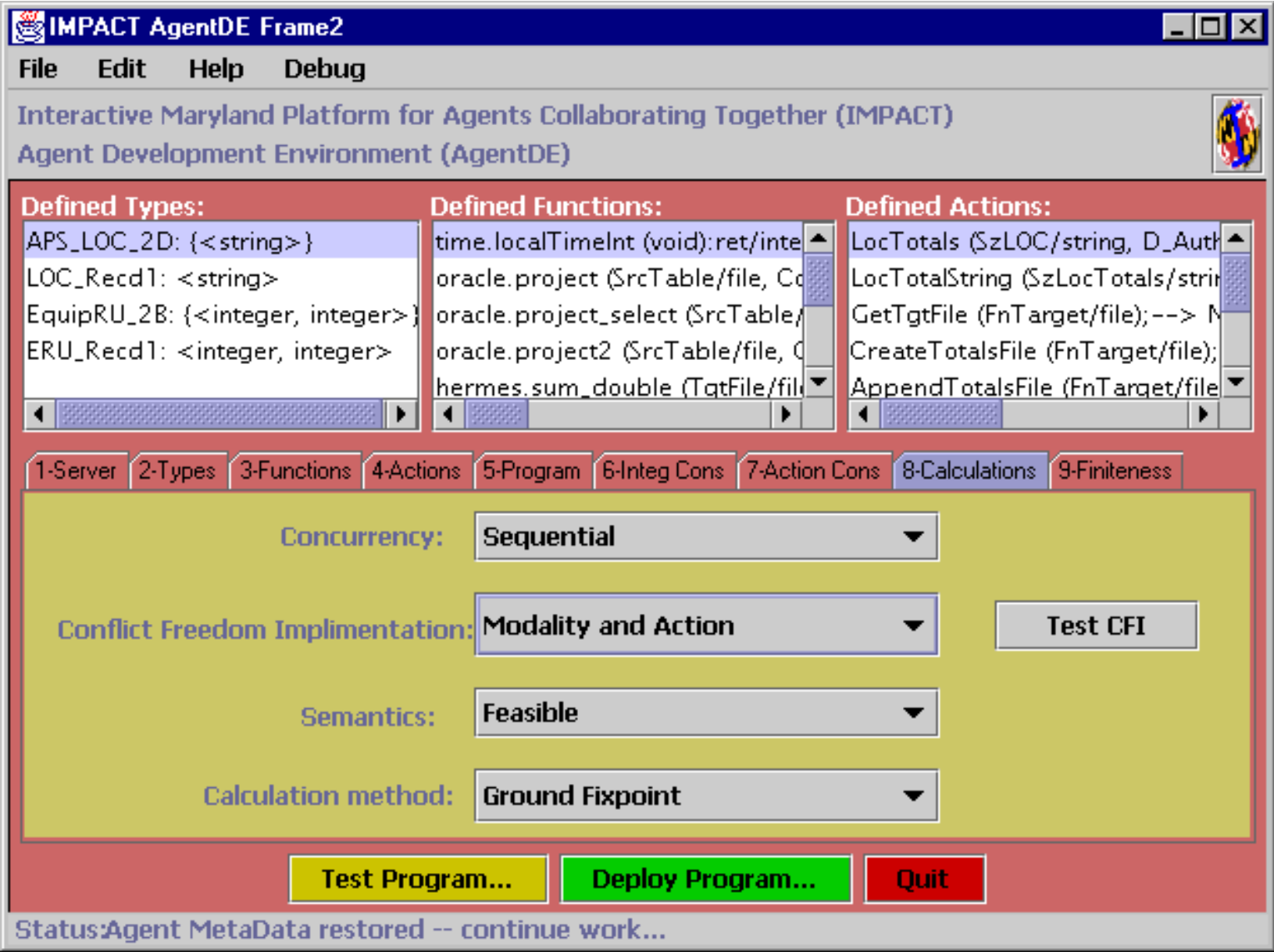


IADE Infiniteness Table Screen

Specify code calls with *infinite* results.



IADE Option Selection Screen



7.4 The Gofish Post Office

Mythical Gofish country

Task:

- Improve Gofish's *Lightning Mail* (which guarantees delivery within 48 hours of dropoff)

Desideratum:

- Flexibility (respect extensions to other products)

Lightning Mail product allows citizens to send certain kinds of packages to other citizens in Gofish.

Packages are dropped, shipped, and delivered.

Package Lifecycle Events

DropOff: Describes the package being dropped off to a Gofish’s Lightning mail collection point.

DistCenter: Refers to the arrival of the package at the distribution center closest to the destination.

Truck: Refers to the loading of the package into a truck at the destination center. The truck will deliver the package to the destination.

Delivery: Refers to the delivery of the package to the destination.

Requirements

- **Multi-stage Notification**

Provide a comprehensive information service about expect mail delivery.

- **Dropoff:** Sender may request to inform recipient about mail drop & expected delivery.
- **DistCenter:** Email recipient about package arrival at local distribution center, revise delivery prediction.
- **Delivery:** When loaded into the truck, phone the recipient to tell when the package will be delivered.
No phone calls between 10pm and 7am !!

- **Precise Delivery Prediction:**

Inform recipients about expected delivery time *as precise as possible*.

(Currently, only large unreliable time window (8am–8pm of a day), but 48h guarantee.)

Predict delivery using a *temporal probability distribution* $P_{O,D}$ for delivery from origination zipcode O to destination zipcode D :

$$P_{O,D} : [1..48] \rightarrow [0, 1]$$

Maintain statistics for determining $P_{O,D}(h)$.

- **Zipcode Monitoring:**

Track which zipcodes are not being well served by the current distribution center allocated them.

Gofish Databases

Lightning Mail has several databases, stored in relations.

package(Pid,TYpe, Wt,...):

Information about each package.

- **Pid:** package id (string);
- **PType:** this is one of envelope, tube, box (string);
- **Wt:** this is the weight of the package in pounds (real);
- **Vol:** this is the volume of the package in cubic inches (int);
- **LSender:** last name of sender (string);
- **FSender:** first name of sender (string);
- **LRecip:** last name of recipient (string);

- **FRecip:** first name of recipient (string);
- **RecipTel:** Recipient’s phone number (string);
- **RecipEMail:** Recipient’s email address (string);
- **OrigZip:** Zip code of the origin (int);
- **DestZip:** Zip code of the destination (int);
- **DestStreet:** destination street name (string);
- **DestNum:** destination street number (int);
- **Priority:** 1..5 (higher number is higher priority)
- **Cost:** the mailing cost (real).
- **DropTime:** time of package dropoff by the sender (0, 1, 2, . . .)
- **DelivTime:** time of package delivery (-1, if unknown)

events(Package,Event,Time):

Events occurring for each package.

- **Package:** Package ID
- **Event:** Type of event (DropOff,DistCenter,Truck,Delivery)
- **Time:** Time (integer)

Example: (123,Dropoff,2) means that package '123' was dropped at time 2.

zip(zip1,zip2,time,number,tot):

Statistics on past delivery, capturing $P_{O,D}$.

- **Zip1:** origination zipcode
- **Zip2:** destination zipcode
- **Time:** hour of delivery (1..48)
- **Number:** number of packages delivered within given hours
- **Tot:** total number of delivered packages

Example: (20742,20715,23,30,200) means 15% probability (30/200) that a package mailed from zip code 20742 to 20715 will arrive in 23 hours exactly.

centertohouse(**Zip,Time,Number,Tot**):

Statistics on past delivery from distribution center.

- **Zip:** destination zipcode
- **Time:** hour of delivery (1..48)
- **Number:** number of packages delivered within given hours
- **Tot:** total number of delivered packages

Example: (**20715,8,11,50**) means that 22% (11 of 50) packages intended for zip code 20715 are delivered in exactly 8 hours from their arrival at the Distribution center.

trucktohouse(Street,Time,Number,Tot,Avg):

Statistics on truck delivery.

- **Street:** destination address
- **Time:** hour of delivery (1..48)
- **Number:** number of packages delivered within given hours
- **Tot:** total number of delivered packages

Example: (**campus-drive, 2,5,20**) means that there is a 25% (5/20) probability of package delivery in exactly 2 hours.

managers(**Zip,Manager,Email**): manager email address

- **Zip**: zip code
- **Manager**: manager of zip code area
- **Email**: email address of the manager

Gofish Multi-Agent System

Develop a multi-agent system for simulation (TU students).

Gofish agents:

- **Package Agent (PA):** Manages the **package** database.
- **Notification Agent (NA):** Inform customers about estimated arrival time on Dropoff (optional), DistCenter (email), and Truck events (phone call, simulated).
NA must obtain the phone number by interaction with PA, and has direct query access to statistic databases.

- **Zip Monitor Agent (ZMA):** Send email messages to all managers responsible for a certain zip code if it is not served well.
ZMA accesses directly **centertohouse** and **manager** databases.

- **Event Manager Agent (EMA):** An event management agent maintains, updates, and uses the **events** database. It also maintains the statistics databases.

EMA receives messages with events from the Dispatcher Agent and sends them to the other GoFish Agents.

Auxiliary agents:

- **Dispatcher Agent (DA):** Processes incoming package drop off messages. Maintains small status DB through which it generates update events for the Event Manager Agent.
- **Package Dropper Agent (PDA):** Creates packages for drop off, using an address table, and sends “DropOff” message to the Dispatcher Agent for processing.

Interaction and Message Protocol

- Asynchronous interaction via message exchange
- Encode service commands in message content:
 - command name
 - Parameters: marshallng using *tokenstring (ts) format*

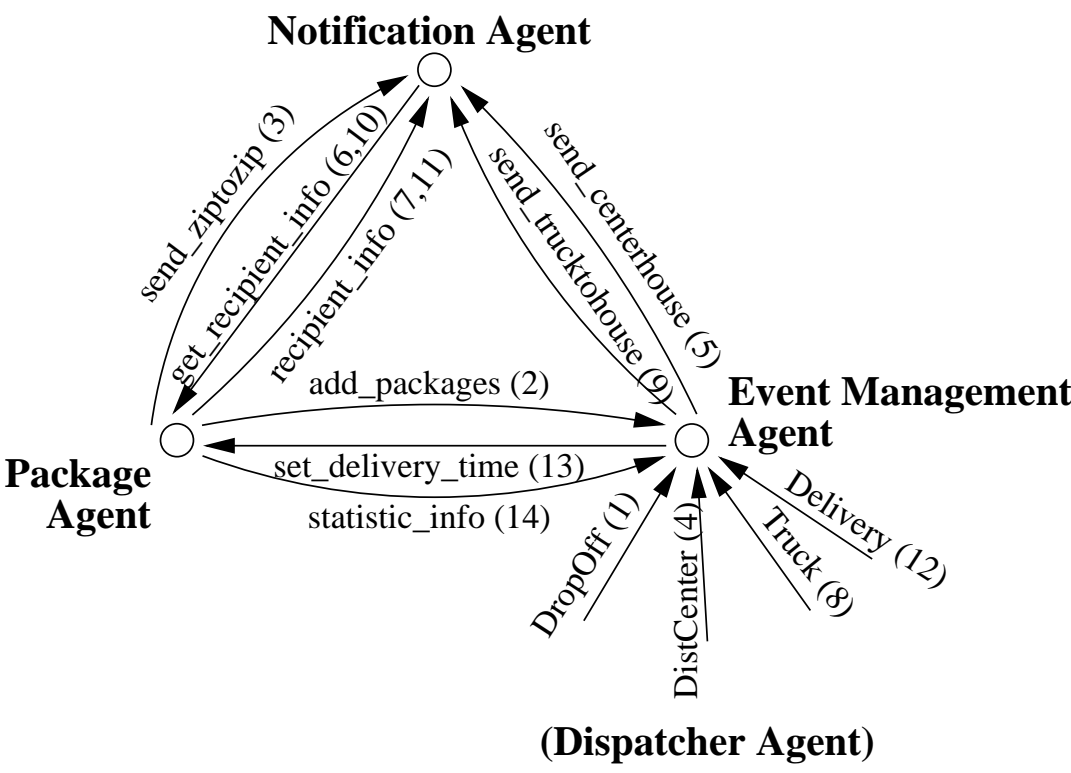
Example: command to get recipient information from Package Agent, after a DistCenter event.

- get_recipient_info
- ts(10,‘DistCenter’) = “10,DistCenter”

Gofish Message description:

#	Command	From To		Data (Parameter)
1	DropOff	any	EM	ts(Pid,Ptype,Wt,Vol,FSender,LSender,OrigZip,FRecip,LRecip,RecipTel,RecipEMail,DestNum,DestStreet,DestZip,Priority)
2	add_package	EM	PA	ts(Pid,PType,Wt,Vol,FSender,LSender,OrigZip,FRecip,LRecip,RecipTel,RecipEMail,DestNum,DestStreet,DestZip,Priority,DropTime)
3	send_ziptozip	PA	NA	ts(RecipEMail,OrigZip,DestZip,DropTime)
4	DistCenter	any	EM	string(Pid)
5	send_centertohouse	EM	NA	string(Pid)
6	get_recipient_info	NA	PA	ts(Pid,'DistCenter')
7	recipient_info	PA	NA	ts(Pid,'DistCenter',RecipTel,RecipEMail,DestZip,DestStreet,DropTime)
8	Truck	any	EM	string (Pid)
9	send_trucktohouse	EN	NA	string(Pid)
10	get_recipient_info	NA	PA	ts(Pid,'Truck')
11	recipient_info	PA	NA	ts(Pid,'Truck',RecipTel,RecipEMail, DestZip,DestStreet,DropTime)
12	Delivery	any	EM	string(Pid)
13	set_delivery _time	EM	PA	ts(Pid,DelivTime)
14	statistic_info	PA	EM	ts(Pid,OrigZip,DestZip,DestStreet)

Gofish message protocol



GoFish Agent Implementation

- Software code *O_S*:
 - MS Access databases, using JDBC, ODBC
 - internal (library) packages for message box, text, string manipulation etc
- Agent actions:
 - send messages
 - read files
 - send emails
 - open popup windows
 - data projections

Example:

```
#IMPACT Action Def#
sendStatisticInfo
(
    TgtAgent / string, Data / string
) ; ; ; ;
-->executes
    sendMessage(
        TgtAgent,
        "statistic_info",
        0,
        Data)
;
```

- Agent Programs:
Small agents have few rules, largest has ~ 30 rules.

Example: Rule from PA program

```
Do(sendStatisticInfo("GFEEventManager", Data)) :-
    Do(setDeliveryTime(PId)) &
    in(Query, Local-->
        text:concat("select * from Package where Pid='", PId, "'") &
    in(Package, GoFishDB-->JDBC:Sql(Query)) &
    =(OrigZip, Package.OrigZip) &
    =(DestZip, Package.DestZip) &
    =(DestStreet, Package.DestStreet) &
    in(Data, Local-->
        text:concat(PId, "^", OrigZip, "^", DestZip, "^", DestStreet)).
```

7.5 Summary and References

Regular Agents: An *efficiently implementable* class of agents.

What are suitable syntactic conditions on agent programs, to ensure polynomial implementability?

1. Weakly regular agents:

- (a) **Strong Safety**: To ensure that code calls return **finitely** many answers (\leadsto Finiteness Table).
- (b) **Conflict-Freedom**: The program should be conflict-free (\leadsto **cff**-tests).
- (c) **Deontic Stratifiability**: Problems with negation are ruled out.

2. Regular Agents: weakly regular + **Unfolding**.

Chapter 8. Planning in Agent Systems

8.1 Planners vs. Agent Systems

8.2 HTN Planning

8.3 Agentising SHOP

8.4 NEO Domain

8.5 Monitoring Agents

8.6 Summary and References

8 Planning in AgentSystems

483-1

8.1 Planning vs. Agent Systems

Traditional planning systems assume:

Homogeneity: all information is available in a common format;

Locality: information is stored locally;

Reasoning: Either symbolic or numerical reasoning are available, but not both.

IMPACT offers handling of heterogenous, distributed data, but lacks any planning component.

Idea: Let's realise a planner as an agent in IMPACT.

8.2 HTN Planning

HTN planning (Sacerdoti 1977; Tate 1977; Wilkins 1988; Currie and Tate 1991) is an AI planning methodology that creates plans by *task decomposition*.

An example is **SHOP** (Nau, Cao, Lotem, and Muñoz-Avila 1999; Munoz-Avila, Aha, Nau, Weber, Breslow, and Yaman 2001): A planning system, which decomposes tasks into smaller and smaller subtasks, until primitive tasks are found that can be performed directly.

SHOP needs to be given knowledge about the domain:

Methods: Describes how to decompose some complex task into a totally ordered sequence of subtasks, along with various restrictions that must be satisfied in order for the method to be applicable.

Operators: Describes what needs to be done to accomplish some primitive task.

Given the next task to accomplish, SHOP chooses an applicable method, instantiates it to decompose the task into subtasks, and then chooses and instantiates other methods to decompose the subtasks even further.

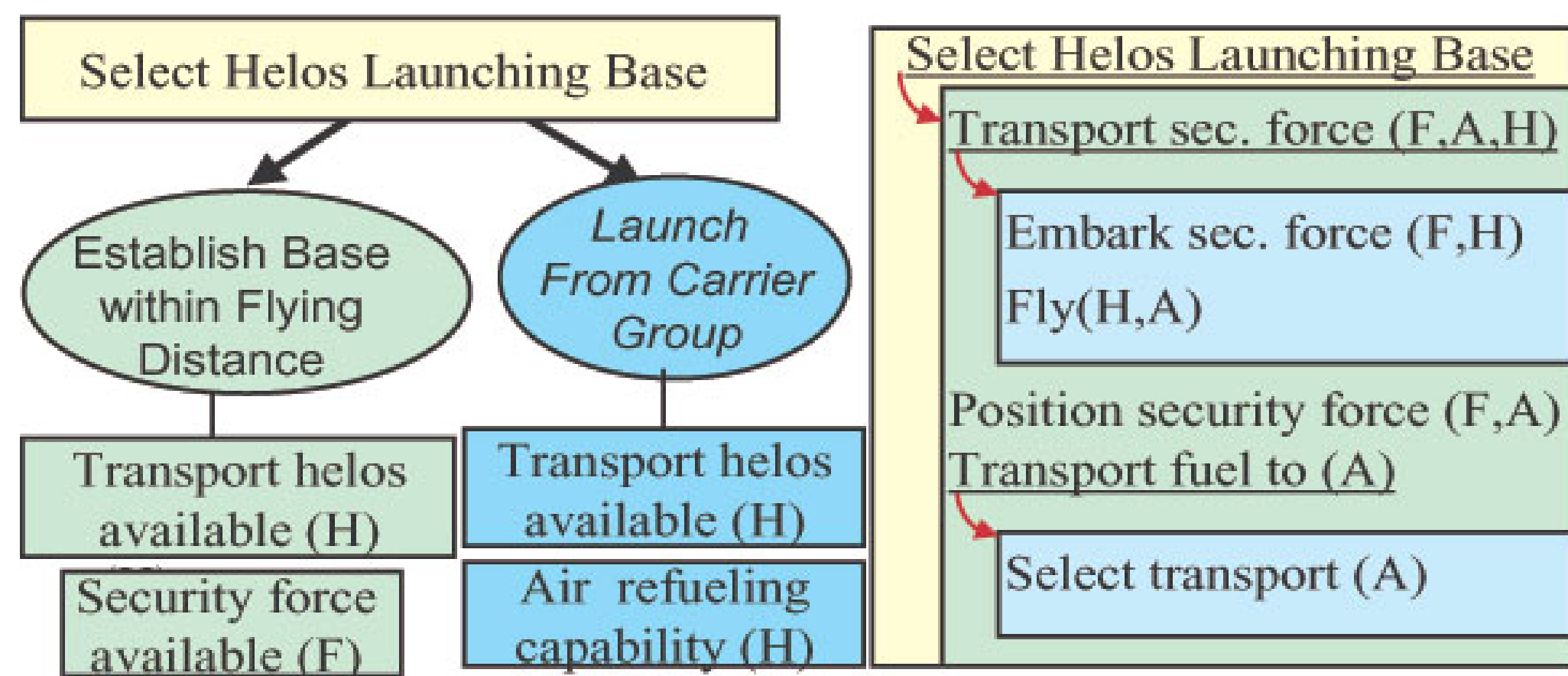


Figure 8.1: NEO transportation example.

8.3 ASHOP: Agentising SHOP

SHOP is a STRIPS planner. In order to agentise it, we need to adjust its methods and operators to something IMPACT like. See (Dix, Munoz-Avila, Nau, and Zhang 2003; Dix, Munoz-Avila, and an Lingling Zhang 2002).

Definition 8.1 (Agentised Method: $(\text{AgentMeth } h \chi t))$
An **agentised method** is an expression of the form $(\text{AgentMeth } h \chi t)$ where h (the method's *head*) is a compound task, χ (the method's *preconditions*) is a code call condition and t is a totally ordered list of subtasks, called the *task list*.

Definition 8.2 (Agentised Operator: $(\text{AgentOp } h \chi_{add} \chi_{del})$)

An **agentised operator** is an expression of the form

$(\text{AgentOp } h \chi_{add} \chi_{del})$, where h (the *head*) is a primitive task and χ_{add} and χ_{del} are lists of code calls (called the *add-* and *delete-lists*). The set of variables in the tasks in χ_{add} and χ_{del} is a subset of the set of variables in h .

Head:

AirTransport(LocFrom, LocTo, Cargo, CargoWeight)

Preconditions:

in(CargoPL, **supplier** : *cargoPlane*(locFrom))&
in(Dist, **statistics** : *distance*(locFrom, locTo))&
in(DCargoPL, **statistics** : *authorRange*(CargoPL))&
Dist ≤ DCargoPL&
in(CCargoPL, **statistics** : *authorCapacity*(CargoPL))&
CargoWeight ≤ CCargoPL&

Subtasks:

load(Cargo, locFrom)
fly(Cargo, locFrom, LocTo)
unload(Cargo, locTo)

procedure $A\text{-SHOP}(t, \mathcal{D})$

1. **if** $t = \textit{nil}$ **then return** \textit{nil}
2. $t :=$ the first task in t ; $R :=$ the remaining tasks
3. **if** t is primitive and a simple plan for t exists **then**
4. $q := \textit{simplePlan}(t)$
5. **return** $\textit{concatenate}(q, \text{A-SHOP}(R, \mathcal{D}))$
6. **else if** t is non-prim. \wedge there is a reduction of t **then**
7. **nondeterministically** choose a reduction:
 Choose (**AgentMeth** $h \chi t$), with μ the most general unifier of h
 and t and substitution θ s.t. $\chi\mu\theta$ is ground and holds
 in \textit{IMPACT} 's state \mathcal{O} .
8. **return** $\text{A-SHOP}(\textit{concatenate}(t\mu\theta, R), \mathcal{D})$
9. **else return** \textit{FAIL}
10. **end if**

end $A\text{-SHOP}$

```

procedure simplePlan( $t$ )
  11. nondeterministically choose agent. operator
      $op = (\text{AgentOp } h \chi_{add} \chi_{del})$  with  $\nu$  the most
     general unifier of  $h$  and  $t$  s.t.  $h$  is ground
  12. monitoring :  $apply(op \nu)$ 
  13. return  $op \nu$ 
end A-SHOP

```

- In step 12, A-SHOP does not issue code calls to the other agents directly, but instead communicates them to a **monitoring** agent.
- The **monitoring** agent keeps track of all operators that are supposed to be applied, without actually modifying the states of the other *IMPACT* agents.
- When A-SHOP queries for a code call $cc = \mathcal{S} : f(d_1, \dots, d_n)$ in χ to evaluate a method's precondition (Step 7), the **monitoring** agent examines if cc has been affected by the intended modifications of the operators and, if so, it evaluates cc .

- The *apply* function applies the operators and creates copies of the state of the world. Depending on the underlying software code, these changes might be easily revertible or not. In the latter case, the monitoring agent has to keep track of the old state of the world.

Lemma 8.1 (Evaluating Agentised Operators)

Let \mathcal{O} be a state, $(\text{AgentMeth } h \chi t)$ an agentised method and $(\text{AgentOp } h' \chi_{add} \chi_{del})$ an agentised operator. If the precondition χ is **strongly safe** wrt. the variables in h , the problem of deciding whether χ holds in \mathcal{O} can be **algorithmically solved**. If the add and delete-lists χ_{add} and χ_{del} are **strongly safe** wrt. the variables in h' , the problem of applying the agentised operator to \mathcal{O} can be **algorithmically solved**.

Theorem 8.1 (Soundness, Completeness)

Let \mathcal{O} be a state and \mathcal{D} be a collection of agentised methods and operators. If all the preconditions in the agentised methods and add and delete-lists in the agentised operators are **strongly safe** wrt. the respective variables in the heads, then **A-SHOP is correct and complete.**

Each cycle in the A-SHOP algorithm consist of three phases (see lines 3 and 7 of ASHOP procedure):

1. **Selection Phase**: Selecting a candidate agentised method or operator to reduce a task.
2. **Evaluation Phase**: Evaluating the applicability of the chosen agentised method or operator.
3. **Reduction Phase**: Performing the agentised method or operator.

To accomplish these phases we have implemented 3 *IMPACT* agents which perform pieces of these phases:

ashop: This is the agent that all *IMPACT* agents communicate with for generating a plan. It receives as input a problem and outputs a solution plan. The A-SHOP agent also performs the Selection Phase and the evaluation phase for the situation in which an operator is chosen.

The operator is then send to the Monitor Agent, to perform a *virtual execution* of it.

If the selection of a method is made, the A-SHOP agent sends a message to the Preconditions Agent with the code-call condition of the selected method.

preconditions: Receives a code-call condition and evaluates each code-call by sending it to the Monitoring Agent.

monitoring: The monitor agent has two functions: firstly, it receives a operator and performs a virtual execution of it. Secondly, it receives code-calls and evaluates them. We explain both of these operations in detail below as they are closely inter-related.

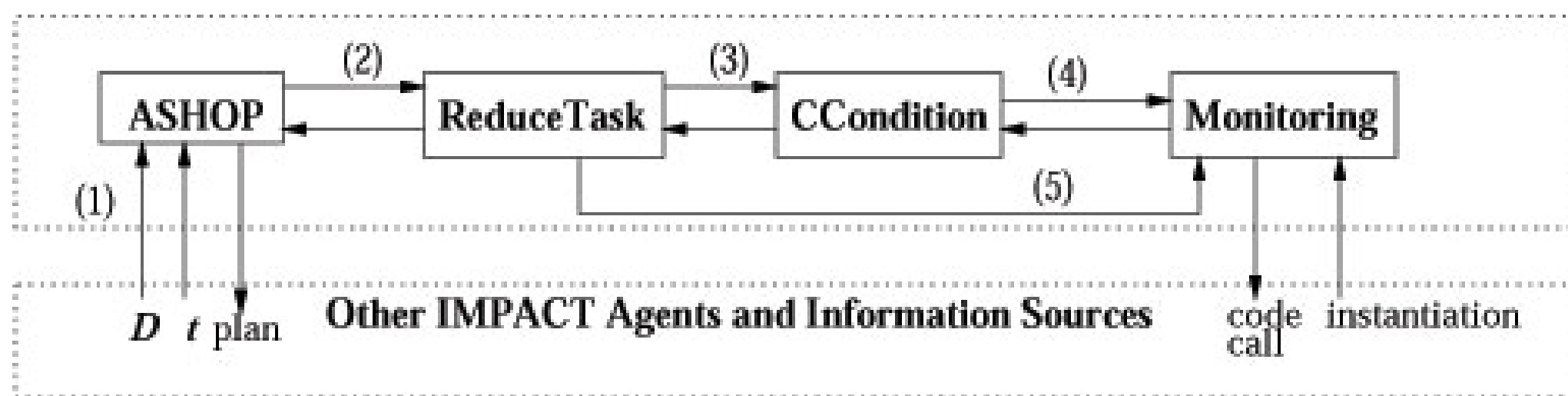


Figure 8.2: Interactions between the agents implementing the A-SHOP algorithm.

8.4 NEO Domain

Our test domain is a simple transportation planning for a NEO (noncombatant evacuation operation, (Munoz-Avila, Aha, Nau, Weber, Breslow, and Yaman 2001)).

Its plans involve performing a rescue mission where troops are grouped and transported between an initial location (the assembly point) and the NEO site (where the evacuees are located). After the troops arrived at the NEO site, evacuees are re-located to a safe haven.

Assembly point (AP) → Intermediate Staging Base (ISB) →
Neo Site (Neo) → Safe Haven (SH)

Planning involves

- selecting routes: **long**, **short**, ...
- means of transportation: **Helicopter**, **Plane**,
HMMVV.

A-SHOP’s knowledge base included six agentised operators and 22 agentised methods. There were four *IMPACT* information sources available:

- **Transport Authority:** Maintains information about the transportation assets available at different locations.
- **Weather Authority:** Maintains information about the weather conditions at the different locations.
- **Airport Authority:** Maintains information about availability and conditions of airports at different locations.
- **Math Agent:** **math** evaluates arithmetic expressions. typical evaluations include the subtract a certain number of assets use for an operation and update time delays.

8.5 Monitoring Agents

How can agent systems be monitored, i.e. the interplay of several agents be checked and the system be debugged?

1. The intended collaborative behavior of the agents is modelled as a **planning problem**. More precisely, knowledge about the agent actions (specifically, messaging) and their effects is formalized in an **action theory**, T , which can be reasoned about to automatically construct **plans** as sequences of actions to reach a given goal.
2. From T and the collaborative goal G , a set of intended plans, **I-Plans**, for reaching G is generated via a planner.
3. The observed agent behavior, i.e., the message actions from a **message log**, is then compared to the plans in *I-Plans*.
4. In case an incompatibility is detected, an error is flagged to the developer resp. user, pinpointing to the last action causing the failure so that further steps might be taken.

Steps 2-4 can be done by a special **monitoring agent**, which is added to the agent system providing support both in testing, and in the operational phase of the system. Among the benefits of this approach are the following:

- It allows to deal with **collaboration behavior regardless of the implementation language(s)** used for single agents.
- Depending on the planner used in step 2, **different kinds of plans** (optimal, conformant, ...), might be considered, reflecting **different agent attitudes and collaboration objectives**.

- Changes to the agent messaging by the system designer may be transparently incorporated to the action theory T , **without further need to adjust the monitoring process**.
- Furthermore, T adds to a **formal system specification**, which may be reasoned about and used in other contexts.
- As a by-product, the method may also be used for automatic **protocol generation**, i.e., determine the messages needed and their order, in a (simple) collaboration.

We consider multi-agent systems consisting of a finite set

$\mathbf{A} = \{\mathbf{a}_1, \dots, \mathbf{a}_n\}$ of collaborating agents \mathbf{a}_i .

Definition 8.3 (Message, \mathcal{M}_{log} file)

A message is a quadruple $m = \langle \mathbf{s}, \mathbf{r}, \mathbf{c}, d \rangle$, where $\mathbf{s}, \mathbf{r} \in \mathbf{A}$ are the identifiers of the *sending* and the *receiving* agents, respectively; $\mathbf{c} \in \mathbf{C}$ is from a finite set \mathbf{C} of *message commands*; d is a list of constants representing the *message data*. A *message-log file* is an ordered sequence $\mathcal{M}_{log} = t_1:m_1, t_2:m_2, \dots, t_k:m_k$ of messages m_i with timestamps t_i , where $t_i \leq t_{i+1}$, $i < k$.

We assume a fixed bound on the time within the next action should happen in the *MAS*, i.e., a timeout for each action (which may depend on previous actions), which allows to see from \mathcal{M}_{log} whether the *MAS* is stuck or still idle.

Example 8.1 (Simple *Gofish*)

For space reasons and to keep things simple and illustrative, we restrict the *Gofish MAS* to the package agent, **pa**, the event management agent, **em**, and the event dispatcher agent, **disp**; thus, $A = \{\mathbf{pa}, \mathbf{em}, \mathbf{disp}\}$.

The event dispatcher informs the event manager agent about the drop off of a package (identified by a unique identifier), its arrival at the distribution center, its loading on a truck, its successful delivery, or when a recipient shows up at the distribution center to pick up a package by herself:

$C_{\mathbf{disp}} = \{\text{dropOff}, \text{distCenter}, \text{truck}, \text{delivery}, \text{pickup}\}$.

The event manager agent instructs the package agent to add a package to the package database after drop off, as well as to update the delivery time after delivery or customer pickup:
 $C_{\text{em}} = \{\text{addPackage}, \text{setDelivTime}\}$. The package agent here only receives messages, thus $C_{pa} = \{\}$.

Running scenario: The message-log \mathcal{M}_{log} contains the messages $m_1 = \langle \text{dropOff}, p_1 \rangle$, $m_2 = \langle \text{addPackage}, p_1 \rangle$, $m_3 = \langle \text{distCenter}, p_1 \rangle$, $m_4 = \langle \text{truck}, p_1 \rangle$, and $m_5 = \langle \text{pickup}, p_1 \rangle$. The entries are 0: $\langle \text{disp}, \text{em}, \text{dropOff}, p_1 \rangle$, 5: $\langle \text{em}, \text{pa}, \text{addPackage}, p_1 \rangle$, 13: $\langle \text{disp}, \text{em}, \text{distCenter}, p_1 \rangle$, 19: $\langle \text{disp}, \text{em}, \text{truck}, p_1 \rangle$, and 20: $\langle \text{disp}, \text{em}, \text{pickup}, p_1 \rangle$.

A planning problem $\mathcal{P}^{\mathcal{K}}$ may be formalized as a tuple $\langle Act, Fl, T, G \rangle$, where Act defines the actions, Fl the fluents, T comprises BK and all axioms, and G is the goal, i.e. a set of ground fluent literals.

The semantics of \mathcal{K} is defined through **legal transitions** $t = \langle s, A, s' \rangle$ from states s to states s' by simultaneous execution of actions A , where a *state* s is any consistent set of ground fluent literals.

A **trajectory** Tr is any initial state s_0 or sequence t_1, \dots, t_n of legal transitions $t_i = \langle s_{i-1}, A_i, s_i \rangle, i \in \{1, \dots, n\}$, starting in an initial state s_0 . An **(optimistic) plan** for goal G is $P = \langle \rangle$, resp. the projection $P = \langle A_1, \dots, A_n \rangle$ of a trajectory Tr , such that G holds in s_0 resp. s_n .

Example 8.2 (Simple *Gofish* cont'd)

The following \mathcal{K} actions and fluents are defined (in $\text{DLV}^{\mathcal{K}}$ notation (Eiter, Faber, Leone, Pfeifer, and Polleres 2002)):

actions : dropOff(P) requires pkg(P).
 addPkg(P) requires pkg(P).
 distCenter(P) requires pkg(P).
 truck(P) requires pkg(P).
 delivery(P) requires pkg(P).
 pickup(P) requires pkg(P).
 setDelivTime(P) requires pkg(P).

fluents : pkgAt(P,Loc) requires pkg(P),loc(Loc).
 delivered(P) requires pkg(P).
 recipAtHome(P) requires pkg(P).
 added(P) requires pkg(P).
 delivTimeSet(P) requires pkg(P).

}

}

Act

FI

8.5 Monitoring Agents

513

- The first three external fluents describe the current location of a package, whether it has successfully been delivered, and whether its recipient is at home, respectively.
- The last two fluents are internal fluents about the state of agent **pa**; whether the package has already been added to the package database resp. whether the delivery time has been set properly.

Definition 8.4 (Structure of the **monitoring agent)**

The agent **monitor** loops through the following steps:

1. Read and parse the message log \mathcal{M}_{log} . If $\mathcal{M}_{log} = \emptyset$, the set of plans for \mathcal{P} may be cached for later reuse.
2. Check whether an action timeout has occurred.
3. If this is not the case, compute the current *intended plans* (according to the planning problem description and additional info from the designer) compatible with the actions as executed by the *MAS*.
4. If no compatible plans survive, or the system is no more idle, then inform the agent designer about this situation.
5. Sleep for some pre-specified time.

The desired collaborative *MAS* behavior is formalized as a planning problem \mathcal{P} (e.g., in language \mathcal{K}).

In general, not all \mathcal{P} -Plans may be admissible, as constraints may apply (derived from the intended collaborative behavior).

We thus distinguish a set $I\text{-Plans}(\mathcal{P}) \subseteq \mathcal{P}\text{-Plans}$ as *intended plans* (of the *MAS* designer).

In general, it is difficult to decide whether the faulty behavior is due to a coding or design error. However, the info given by **monitor** will aid the agent designer to detect the real cause.

Agent **monitor** continually checks and compares the actions taken so far for compatibility with all current plans. Once a situation has arisen in which no successful plan exists (detected by the planner employed), **monitor** writes a message into a separate file containing

- the first action that caused the *MAS* to go into a state where the goal is not reached,
- the sequence of actions taken up to this action, and
- all the possible plans *before* the action in 1) was executed (these are all plans compatible with the *MAS* behavior up to it).

Running scenario (coding error): Suppose on a preliminary run of our scenario, \mathcal{M}_{log} shows $m_1=\text{dropOff}(p_1)$. This is compatible with each plan P_i , $i \in \{1, 2, 3\}$. Next, $m_2=\text{distCenter}(p_1)$. This is incompatible with each plan; **monitor** detects this and gives a warning. Inspection of the actual code may show that the command for adding the package to the database is wrong. While this doesn't result in a livelock (the *MAS* is still idle), the database was not updated. Informed by **monitor**, this is detected at this stage already.

After correction of this coding error, the *MAS* may be started again and another error shows up:

Running scenario (design error): Instead of waiting at home (as in the *standard* plan P_2), Sue shows up at the distribution center and made a pickup attempt.

This *external* event may have been unforeseen by the designer (they could also arise from *MAS* actions).

We can expect this in many agent scenarios: we have no complete knowledge about the world, unexpected events may happen, and action effects may not fully determine the next state.

Only plan P_1 remains to reach the goal. However, there is *no guarantee of success*, if Sue is not back home in time for delivery.

Definition 8.5 (\mathcal{M}_{log} compatible plans)

Let the planning problem \mathcal{P} model the intended behavior of a *MAS*, which is given by a set $\text{I-Plans}(\mathcal{P}) \subseteq \mathcal{P}\text{-Plans}$. Then, for any message log $\mathcal{M}_{log} = t_1:m_1, \dots, t_k:m_k$, we denote by

$\text{C-Plans}(\mathcal{P}, \mathcal{M}_{log}, n), n \geq 0$, the set of plans from $\text{I-Plans}(\mathcal{P})$ which comply on the first n steps with the actions m_1, \dots, m_n .

Definition 8.6 ($\text{Culprit}(\mathcal{M}_{log}, \mathcal{P})$)

Let $t_n:m_n$ be the first entry of \mathcal{M}_{log} such that either (i) $\text{C-Plans}(\mathcal{P}, \mathcal{M}_{log}, n) = \emptyset$ or (ii) a timeout is detected. Then, $\text{Culprit}(\mathcal{M}_{log}, \mathcal{P})$ is the pair $\langle t_n:m_n, idle \rangle$ if (i) applies and $\langle t_n:m_n, timeout \rangle$ otherwise.

Initially, \mathcal{M}_{log} is empty and thus $\text{C-Plans}(\mathcal{P}) = \text{I-Plans}(\mathcal{P})$. As more and more actions are executed by the *MAS*, they are recorded in \mathcal{M}_{log} and the set $\text{C-Plans}(\mathcal{P})$ shrinks. **monitor** can thus compare at any point in time whether $\text{C-Plans}(\mathcal{P}, \mathcal{M}_{log}, n)$ is empty or not. Whenever this happens, $\text{Culprit}(\mathcal{M}_{log}, \mathcal{P})$ is computed and pinpoints to the problematic action.

Theorem 8.2 (Soundness)

Let the planning problem \mathcal{P} model the intended collaborative *MAS* behavior, given by $\text{I-Plans}(\mathcal{P}) \subseteq \mathcal{P}\text{-Plans}$. Let \mathcal{M}_{log} be a message log.

Then, the *MAS* is implemented incorrectly if $\text{Culprit}(\mathcal{M}_{log}, \mathcal{P})$ exists.

Semantically, the intended collaborative *MAS* behavior may manifest in a set of trajectories as described for \mathcal{K} planning problems, where trajectories correspond to possible runs of the *MAS* (sequences of states and executed actions).

We say that a set OP of such plans **covers** the intended collaborative *MAS* behavior, if each run of the *MAS* corresponds to some trajectory whose projection is in OP .

For example, this holds if OP is the set of all optimistic plans for $\mathcal{P}^{\mathcal{K}}$ and the intended collaborative MAS behavior is given by a secure plan, or, more liberally, by a conditional plan. We have:

Theorem 8.3 (Soundness of $\mathcal{P}^{\mathcal{K}}$ Cover)

Let $\mathcal{P}^{\mathcal{K}}$ be a \mathcal{K} planning problem, such that $\text{I-Plans}^o(\mathcal{P}^{\mathcal{K}})$ covers the intended collaborative MAS behavior. Let \mathcal{M}_{log} be a message log.

Then, MAS is implemented incorrectly if $\text{Culprit}(\mathcal{M}_{log}, \mathcal{P}^{\mathcal{K}})$ exists.

As for completeness, we need the assertion that plans can not grow arbitrarily long, i.e., have an upper bound on their length.

Theorem 8.4 (Completeness)

Let the planning problem \mathcal{P} model the intended collaborative *MAS* behavior, given by $\text{I-Plans}(\mathcal{P}) \subseteq \mathcal{P}\text{-Plans}$ where plans are bounded.

If the *MAS* is implemented incorrectly, then there is some

message log \mathcal{M}_{log} such that either (i) $\text{C-Plans}(\mathcal{P}, \mathcal{M}_{log}, 0) = \emptyset$,

or (ii) $\text{Culprit}(\mathcal{M}_{log}, \mathcal{P})$ exists .

In (i), we can conclude a design error , while in (ii) a

design or coding error may be present. There is no similar completeness result for \mathcal{P}^κ covers. However, the culprit vanishes if the cover contains plan P_1 , which is compatible with \mathcal{M}_{log} .

For more detailed information see (Dix, Eiter, Fink, Polleres, and Zhang 2004) and <http://www.cs.man.ac.uk/~zhangy/project/monitor/>.

8.6 Summary and References

In order to **combine** HTN planning in a multi agent system, we agentised SHOP's operators and methods

1. We stated **Soundness/Completeness** theorems for A-SHOP.
2. A-SHOP is a planning agent in *IMPACT* built on four agents. It allows to access **heterogenous** and **remote** data.
3. A-SHOP has been tested on the NEO domain, which is described in the literature.
4. It is implemented and freely available from the authors (J. Dix, D. Nau, H. Munoz-Avila, Lingling Zhang) under the GNU General Public License as published by the Free Software Foundation.

5. Planning is also useful for **debugging** agent systems. The intended collaborative behaviour needs to be formalised as a planning problem.
6. Our approach is based on just **checking the messages sent between the agents**.
7. While the agent system is running, the remaining possible plans are kept track and if a situation is reached where the overall goal can not be reached anymore, a warning is flagged.
8. The monitoring agent pinpoints to the last action as a **possible culprit**.

References

Currie, K. and A. Tate (1991). O-plan: the open planning architecture. *Artificial Intelligence* 52(1).

Dix, J., H. Munoz-Avila, and D. Nau. and Lingling Zhang (2002). Planning in a Multi-Agent Environment: Theory and Practice. In *Proceedings of Journees Europeens de la Logique en Intelligence artificielle (JELIA '02)*. Springer.

Dix, J., H. Munoz-Avila, D. Nau, and L. Zhang (2003). IMPACTing SHOP: Putting an AI planner into a Multi-Agent Environment. *Annals of Mathematics and AI* 37(4), 381–407.

- Eiter, T., W. Faber, N. Leone, G. Pfeifer, and A. Polleres (2002). A Logic Programming Approach to Knowledge-State Planning, II: The DLV^K System. *Artificial Intelligence* 144(1-2), 157–211.
- D.S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of IJCAI-99*, 1999.
- D.S. Nau, H. Muñoz-Avila, Y. Cao, A. Lotem, and S. Mitchell. Total-order planning with partially ordered subtasks. In *Proceedings of IJCAI-2001*, 2001.
- Sacerdoti, E. (1977). *A Structure for Plans and Behavior*. American Elsevier Publishing.

Subrahmanian, V., P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Özcan, and R. Ross (2000). *Heterogenous Active Agents*. MIT-Press.

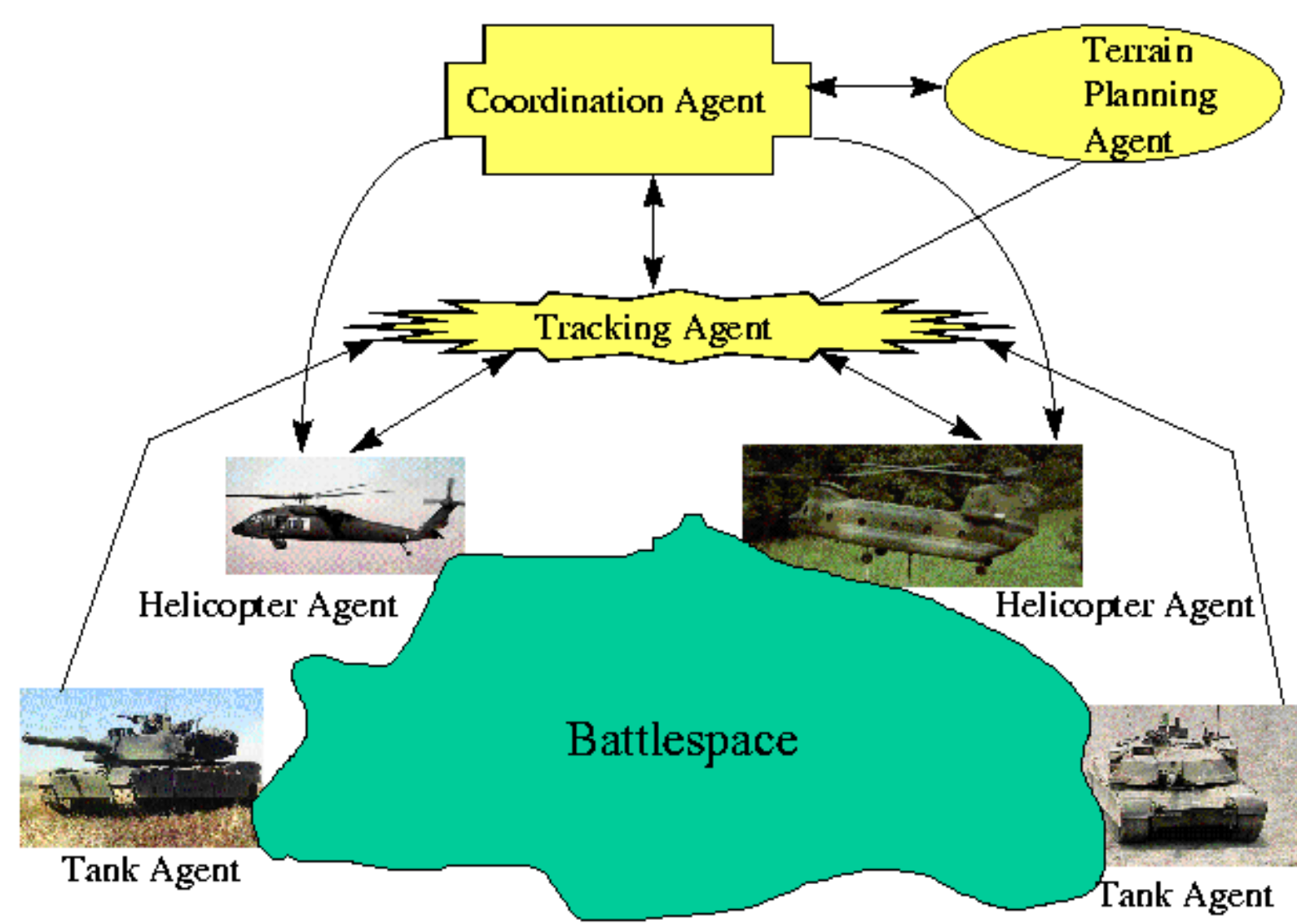
Tate, A. (1977). Generating Project Networks. In *Proc. IJCAI-77*, pp. 888–893.

Weiss, G. (Ed.) (1999). *Multi-Agent Systems*. MIT-Press.

Wilkins, D. (1988). *Practical planning - extending the classical AI planning paradigm*. Morgan Kaufmann.

Extensions of our Basic Approach

- **Beliefs:** Agents hold beliefs about other agents.
Dix/Subrahmanian/Pick: Meta Agent Programs, Journal of Logic Programming, 46(1–2)*1–60, 2000.
- **Uncertainty:** Available information might be uncertain.
Dix/Nanni/Subrahmanian: Probabilistic Agent Reasoning, Transactions of Computational Logic, 1(2)*201–245, 2000
- **Time:** Agents make commitments to the future.
Dix/Kraus/Subrahmanian: Temporal Agent Reasoning, Artificial Intelligence, 127(1)*87–135, 2001



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 terrain route planning agent **terrain:** 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.

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. All three of **beliefs**, **time** and **uncertainty** enter here.

A **coordination agent**: **coordination** 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.

- **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.

- **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*). 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.

9. Adding Beliefs

9.1 Belief Language and Data Structures

9.2 Meta Agent Programs and Status Sets

9.3 Reducing map's to Ordinary Agent Programs

Timetable:

- Chapter 9 needs 30 minutes.

9 Adding Beliefs

9.1 Belief Language and Data Structures

- 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.
- Most important are the beliefs about
what holds in another agents state

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

- In that case, agent **a** must also have **background information**: beliefs about agent **b**'s software package \mathcal{S}^b : the code call condition χ has to be contained in \mathcal{S}^b .

Example 9.1 (Belief Atoms In CFIT*)

$\mathcal{B}_{\text{heli1}}(\text{tank1}, \text{in}(\text{pos1}, \text{tank1} : \text{getPos}()))$

This belief atom says that the agent, **heli1** believes that agent **tank1**'s current state indicates that **tank1**'s current position is pos1.

$\mathcal{B}_{\text{heli1}}(\text{tank1}, \text{F}_{\text{attack}}(\text{pos1}, \text{pos2}))$

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.

$\mathcal{B}_{\text{heli3}}(\text{tank1}, \text{O}_{\text{drive}}(\text{pos1}, \text{pos2}, 35))$

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 mph.

The precise definition is very complicated!

A nested belief atom of the form

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

does not make sense (because $\mathbf{b} \neq \mathbf{c}$).

Thus every agent keeps track of only its *own* beliefs, not those of other agents!!

- We can use conjunctions with respect to different agents
 $\mathcal{B}_a(\mathbf{b}, \chi) \wedge \mathcal{B}_a(\mathbf{c}, \chi')$.
- We also use different nested levels of beliefs, like
 $\mathcal{B}_a(\mathbf{b}, \chi) \wedge \mathcal{B}_a(\mathbf{c}, \mathcal{B}_c(\mathbf{d}, \chi'))$.

Example 9.2 (Belief Formulae for CFIT*)

The following are belief formulae from $\mathcal{BL}_1^{\text{heli1}}$, $\mathcal{BL}_2^{\text{tank1}}$ and $\mathcal{BL}_3^{\text{coord}}$.

$\mathcal{B}_{\text{heli1}}(\text{tank1}, \text{in}(\text{pos1}, \text{tank1} : \text{getPosition}()))$.

This formula is in $\mathcal{BL}_1^{\text{heli1}}$. It says that agent **heli1** believes that agent **tank1**'s current state indicates that **tank1**'s current position is pos1.

$\mathcal{B}_{\text{tank1}}(\text{heli1}, \mathcal{B}_{\text{heli1}}(\text{tank1}, \text{in}(\text{pos1}, \text{tank1} : \text{getPosition}())))$.

This formula is in $\mathcal{BL}_2^{\text{tank1}}$. It says that agent **tank1** believes that agent **heli1** believes that agent **tank1**'s current position is pos1.

$\mathcal{B}_{\text{coord}}(\text{tank1}, \mathcal{B}_{\text{tank1}}(\text{heli1}, \mathcal{B}_{\text{heli1}}(\text{tank2}, \text{in}(\text{pos2}, \text{tank2} : \text{getPosition}(\text{0}))))).$

This formula is in $\mathcal{BL}_3^{\text{coord}}$. It says that agent **coord** believes that agent **tank1** believes that **heli1** believes that agent **tank2**'s current position is pos2.

However, the following formula does not belong to any of the above belief languages:

$$\mathcal{B}_{\text{tank1}}(\text{heli1}, \mathcal{B}_{\text{tank1}}(\text{tank1}, \text{in}_{\text{pos1}}(\text{tank1} : \text{getPosition}()))).$$

The reason for this is because in **heli1**'s state there can be no beliefs belonging to **tank1**.

Example 9.3 (Basic Belief Table for CFIT* Agents)

We define suitable basic belief tables for agent **tank1**.

Agent	Formula
heli1	in _{(pos1, heli1 : <i>getPosition</i>())}
heli2	B _{heli2} (tank1 , in _{(pos1, tank1 : <i>getPosition</i>())})
tank2	B _{tank2} (heli1 , B _{heli1} (tank1 , in _{(pos3, tank1 : <i>getPosition</i>())}))

Table 9.1: A Basic Belief Table for agent **tank1**.

What kind of operations should we support on belief tables?

We distinguish between two different types:

1. For a given agent **h**, other than **a**, we may want to select all entries in the table having **h** as first argument.
2. For a given belief formula ϕ , we may be interested in all those entries, whose second argument “implies” (w.r.t. some underlying definition of entailment) the given formula ϕ .

9.1.1 Belief Semantics Table

Agent **a** may associate different background theories with different agents: it may assume that agent **h** reasons according to semantics $\mathcal{BSem}_{\mathbf{h}}^{\mathbf{a}}$ and assumes that agent **h'** adopts a stronger semantics $\mathcal{BSem}_{\mathbf{h}'}^{\mathbf{a}}$. We will store the information in a separate relational data structure:

Example 9.4 (Belief Semantics Tables for CFIT* Agents)

We briefly describe what suitable Belief Semantics Table for **heli1** and **tank1** may look like. We define entailment relations $BSem_{heli2}^{tank1}$, and $BSem_{tank1}^{heli1}$. For simplicity we restrict these entailment relations to belief formulae of level at most 1, i.e., \mathcal{BL}_1^h .

1. $BSem_{tank1}^{heli1}$: The smallest entailment relation satisfying the schema

$$\mathcal{B}_{tank1}(tank1.1, \chi) \rightarrow \chi.$$

This says that **heli1** believes that all beliefs of **tank1** about **tank1.1** are actually true: **tank1** knows all about **tank1.1**.

2. $\mathcal{BSem}_{\text{heli2}}^{\text{tank1}}$: The smallest entailment relation satisfying the schema

$$\mathcal{B}_{\text{heli2}}(\text{tank2}, \chi) \wedge \mathcal{B}_{\text{heli2}}(\text{tank2.1}, \chi) \rightarrow \chi.$$

This says that **tank1** believes that if **heli2** believes that χ is true both for **tank2** and **tank2.1** then this is actually true.

The notion of a semantics used in the belief semantics table is very general: it can be an arbitrary relation on $\mathcal{BL}_i^{\mathbf{h}} \times \mathcal{BL}_i^{\mathbf{h}}$.

As an example, consider the following two simple axioms that can be built into a semantics:

- (1) $\mathcal{B}_{\mathbf{h}_2}(\mathbf{h}, \chi) \Rightarrow \mathcal{B}_{\mathbf{h}_2}(\mathbf{h}', \chi)$
- (2) $\mathcal{B}_{\mathbf{h}_2}(\mathbf{h}, \chi) \Rightarrow \chi$

The first axiom refers to different agents \mathbf{h}, \mathbf{h}' while the second combines different *levels* of belief atoms. In many applications, however, such axioms will not occur: $\mathbf{h} = \mathbf{h}'$ is fixed and the axioms operate on the same level i of belief formulae.

Suppose an agent \mathbf{a} believes that another agent \mathbf{h}_1 reasons according to the feasible semantics, \mathbf{h}_2 reasons according to the rational semantics etc. It would be nice if this could be encoded as follows in $\text{BSemT}^{\mathbf{a}}$

$$\langle \mathbf{h}_1, \text{Sem}_{feas} \rangle$$

$$\langle \mathbf{h}_2, \text{Sem}_{rat} \rangle$$

$$\langle \mathbf{h}_3, \text{Sem}_{reas} \rangle$$

This is indeed possible.

The idea is to use the semantics Sem of the action program $\mathcal{P}^{\mathbf{a}}(\mathbf{b})$ (that \mathbf{a} believes \mathbf{b} to have) for the evaluation of the belief formulae.

9.2 Meta Agent Programs and Status Sets

Definition 9.1 (Meta Agent Program (map) \mathcal{BP})

A *meta agent rule*, (mar for short), for agent \mathbf{a} is an expression r of the form

$$\text{Op } \alpha(\vec{t}) \leftarrow L_1, \dots, L_n \quad (9.4)$$

where $\text{Op } \alpha(\vec{t})$ is an action status atom, and each of L_1, \dots, L_n is either a code call literal, an action literal or a belief literal from $\mathcal{BLit}_\infty(\mathbf{a}, \mathbf{A})$.

A *meta agent program*, (map for short), for agent \mathbf{a} is a finite set \mathcal{BP} of meta agent rules for \mathbf{a} .

Example 9.5 (map's For CFIT*-Agents)

Let **heli1**'s meta agent program be as follows:

$$\begin{aligned} \text{P } \textit{attack}(\text{P1}, \text{P2}) \quad &\leftarrow \quad \mathcal{B}_{\text{heli1}}(\text{tank1}, \text{in}(\text{P2}, \text{tank1} : \textit{getPos}(\text{ }))) \quad , \\ &\text{P } \textit{fly}(\text{P1}, \text{P3}, \text{A}, \text{S}) \quad , \\ &\text{P } \textit{attack}(\text{P3}, \text{P2}) \quad . \end{aligned}$$

where $\textit{attack}(\text{P1}, \text{P2})$ is an action which means attack position P2 from position P1. **heli1**'s program says **heli1** can attack position P2 from P1 if **heli1** believes **tank1** is in position P2, **heli1** can fly from P1 to another position P3 at altitude A and speed S, and **heli1** can attack position P2 from P3.

Let **tank1**'s meta agent program be as follows:

$$\text{O } \textit{attack}(\text{P1}, \text{P2}) \leftarrow \text{O } \textit{driveRoute}([\text{P0}, \text{P1}, \text{P2}, \text{P3}], \text{S}) ,$$

$$\textcolor{blue}{B}_{\text{tank1}}(\text{tank2}, \text{in}(\text{P2}, \text{tank2} : \textit{getPos}(\text{ }))).$$

If **tank1** must drive through a point where it believes **tank2** is, it must attack **tank2**.

From now on we assume that the software package

$\mathcal{S}^{\mathbf{a}} = (\mathcal{T}_{\mathcal{S}^{\mathbf{a}}}, \mathcal{F}_{\mathcal{S}^{\mathbf{a}}})$ of each agent \mathbf{a} contains as distinguished data types

1. the belief table $\text{BT}^{\mathbf{a}}$, and
2. the belief semantics table $\text{BSemT}^{\mathbf{a}}$,

as well as the corresponding functions

$\text{BT}^{\mathbf{a}} : \text{B-proj-select}(r, \mathbf{h}, \phi)$ and $\text{BSemT}^{\mathbf{a}} : \text{select}(\text{agent}, =, \mathbf{h})$.

What is a status set?

Definition 9.2 (Belief Status Set BS)

A belief status set BS of agent \mathbf{a} , also written $BS(\mathbf{a})$, is a set consisting of two kinds of elements:

- ground action status atoms over $\mathcal{S}^{\mathbf{a}}$ and
- belief atoms from $BAt_{\infty}(\mathbf{a}, \mathbf{A})$ of level greater or equal to 1.

The reason that we do not allow belief atoms of level 0 is to avoid having code call conditions in our set. In agent programs without beliefs (which we want to extend) they are not allowed (see Definition 6.14 on page 375).

A status set must be determined in accordance with

1. the map \mathcal{BP} of agent \mathbf{a} ,
 2. the current state \mathcal{O} of \mathbf{a} ,
 3. the underlying set of action (\mathcal{AC}) and integrity constraints (\mathcal{IC}) of \mathbf{a} .
- In contrast to agent programs without beliefs we now have **cope with all agents about which \mathbf{a} holds certain beliefs**.
 - Even if the map \mathcal{BP} does not contain nested beliefs (which are allowed), we cannot restrict to belief atoms of level 1 ($\text{BT}^{\mathbf{a}}$ may contain nested beliefs and, by $\text{BSemT}^{\mathbf{a}}$, such nested beliefs may trigger other beliefs).

9.3 Reducing map's to Ordinary Agent Programs

Definition 9.3 (Extended Code Calls, \mathcal{S}^{ext})

Given an agent \mathbf{a} , we will from now on distinguish between *basic* and *extended* code calls (resp. conditions). The basic code calls refer to the package \mathcal{S} , while the latter refer to the extended software package which also contains

1. the following function of the belief table:
 - (a) $\mathbf{a} : \text{belief_table}()$, which returns the full belief table of agent \mathbf{a} , as a set of triples $\langle \mathbf{h}, \phi, \chi_{\mathcal{B}} \rangle$,

2. the following functions of the belief semantics table:
 - (b) $\mathbf{a} : \text{belief_sem_table}()$, which returns the full belief semantics table, as a set of pairs $\langle \mathbf{h}, BSem_{\mathbf{h}}^{\mathbf{a}} \rangle$,
 - (c) $\mathbf{a} : \text{bel_semantics}(\mathbf{h}, \phi, \psi)$, which returns true when $\phi \models_{BSem_{\mathbf{h}}^{\mathbf{a}}} \psi$ and false otherwise.

3. the following functions, which implement for every sequence σ the beliefs of agent \mathbf{a} about σ as described in $\Gamma^{\mathbf{a}}(\sigma)$:
- (d) $\mathbf{a} : \textit{software_package}(\sigma)$, which returns the set $\mathcal{S}^{\mathbf{a}}(\sigma)$,
 - (e) $\mathbf{a} : \textit{action_base}(\sigma)$, which returns the set $\mathcal{AB}^{\mathbf{a}}(\sigma)$,
 - (f) $\mathbf{a} : \textit{action_program}(\sigma)$, which returns the set $\mathcal{P}^{\mathbf{a}}(\sigma)$,
 - (g) $\mathbf{a} : \textit{integrity_constraints}(\sigma)$, which returns the set $\mathcal{IC}^{\mathbf{a}}(\sigma)$
 - (h) $\mathbf{a} : \textit{action_constraints}(\sigma)$, which returns the set $\mathcal{AC}^{\mathbf{a}}(\sigma)$,

4. the following functions which simulate the state of another agent **b** or a sequence σ ,
- (i) $\mathbf{a} : \text{bel_ccc_act}(\sigma)$, which returns all the code call conditions and action status atoms that **a** believes are true in σ 's state. We write these objects in the form "**in**(,)" (resp. "Op α " for action status atoms) in order to distinguish them from those that have to be checked in **a**'s state.
 - (j) $\mathbf{a} : \text{not_bel_ccc_act}(\sigma)$, which returns all the code call conditions and action status atoms that **a** does not believe to be true in σ 's state.

We also write \mathcal{S}^{ext} for this extended software package and distinguish it from the original \mathcal{S} from which we started.

We now

1. *transform meta agent programs into agent programs,*

(this is a source-to-source transformation: the belief atoms in a meta agent program are replaced by suitable code calls to the new datastructures),

2. *take advantage of extended code calls **Sext**.*

- Suppose the belief table does not contain any belief conditions (i.e., it coincides with its basic belief table).
- Then if χ is any code call condition of agent \mathbf{c} , the extended code call atom

$$\mathbf{in}(\langle \mathbf{c}, \chi, \mathbf{true} \rangle, \mathbf{a} : \mathit{belief_table}())$$

corresponds to the belief atom

$$\mathcal{B}_{\mathbf{a}}(\mathbf{c}, \chi).$$

- But beliefs can also be triggered by entries in the belief table and/or in the belief semantics table!

- What happens if the formula χ is not a code call, but again a belief formula, say $\mathcal{B}_{\mathbf{c}}(\mathbf{d}, \chi')$?

Here is where the inductive definition of $\mathcal{T}_{\text{rans}}$ comes in. We map

$$\mathcal{B}_{\mathbf{a}}(\mathbf{c}, \mathcal{B}_{\mathbf{c}}(\mathbf{d}, \chi'))$$

to

$$\text{in}(\chi', \mathbf{a} : \text{bel_ccc_act}([\mathbf{c}, \mathbf{d}])).$$

Our main theorem states that there is indeed a uniform transformation $\mathcal{T}rans$ from arbitrary meta agent programs (which can also contain nested beliefs) to agent programs such that the semantics are preserved:

$$\mathbf{Sem}(\mathcal{BP}) = \mathbf{Sem}(\mathcal{T}rans(\mathcal{BP})) \quad (9.5)$$

where \mathbf{Sem} is either the *feasible*, *rational* or *reasonable* belief status set semantics.

$$\begin{array}{ccccc}
 & \mathcal{BP} & \xrightarrow{\mathcal{T}rans} & \mathcal{P} & \\
 \text{Compatible with} & \uparrow \mathbf{Sem}^{\text{new}} & & \mathcal{I}^{\text{ext}} & \uparrow \mathbf{Sem}^{\text{old}} \\
 \text{Belief Semantics} & & & \text{Closure} & \\
 \text{Belief Table} & & & & \\
 & \mathcal{BS} & \xrightarrow{\mathcal{T}rans} & S &
 \end{array} \quad (9.6)$$

Chapter 10. Adding Uncertainty

10.1 Probabilistic Code Calls

10.2 Probabilistic Agent Programs

10.3 Kripke Style Semantics

Timetable:

- Chapter 10 needs 30 minutes.

10 Adding Uncertainty

10.1 Probabilistic Code Calls

Imagine a surveillance example, where `surv : identify(image1)` tries to identify all objects in a given image: it is well known that this is an uncertain task.

Some objects may be identified with 100% certainty, while in other cases, it may only be possible to say it is either a T-72 tank with 40–50% probability.

Definition 10.1 (Random Variable of Type τ)

A *random variable* of type τ is a finite set RV of objects of type τ , together with a probability distribution \wp that assigns real numbers in the unit interval $[0, 1]$ to members of RV such that $\sum_{o \in \text{RV}} \wp(o) \leq 1$.

Uncertainty can be captured as follows.

Definition 10.2 (Probabilistic Code Call $\mathbf{a} :_{\text{RV}} f(d_1, \dots, d_n)$)

Suppose $\mathbf{a} : \mathbf{f}(d_1, \dots, d_n)$ is a code call whose output type is τ . The *probabilistic code call* associated with $\mathbf{a} : \mathbf{f}(d_1, \dots, d_n)$, denoted $\mathbf{a} :_{\text{RV}} f(d_1, \dots, d_n)$, returns a set of random variables of type τ when executed.

Example 10.1

Consider the code call **surv**:_{RV} *identify*(image1). This code call may return the following two random variables.

$$\langle \{t72, t80\}, \{ \langle t72, 0.5 \rangle, \langle t80, 0.4 \rangle \} \rangle$$

and

$$\langle \{t60, t84\}, \{ \langle t60, 0.3 \rangle, \langle t84, 0.7 \rangle \} \rangle$$

This says that the image processing algorithm has identified two objects in image1:

- The first object is either a T-72 or a T-80 tank with 50% and 40% probability, respectively, while
- the second object is either a T-60 or a T-84 tank with 30% and 70% probability respectively.

Probabilistic cc's and ccc's look exactly like ordinary cc's and ccc's—however, as a probabilistic code call returns a set of *random variables*, **probabilistic code call atoms are true or false with some probability.**

Example 10.2

Consider the probabilistic code call condition

in(X , **surv**: RV *identify*(image1)) & **in**($a1$, **surv**: RV *turret*(X)).

This ccc attempts to find all vehicles in “image1” with a gun turret of type a1. Let us suppose that the first cc is as on the previous page, but gives back only the first random variable.

When this result (X) is passed to the second code call, it returns one random variable with two values—a1 with probability 30% and a2 with probability 65%.

What is the probability that the code call condition above is satisfied by a particular assignment to X ?

Let's suppose X is assigned T72. If all T72's have a2-type turrets, then the answer is "0".

Let's suppose X is assigned T80. If the vehicule and turret identification is independent, then the answer is " $0.4 \times 0.3 = 0.12$ ".

Example 10.3

Suppose we consider a code call cc returning the following two random variables.

$$\mathbf{RV}_1 = \langle \{a, b\}, \wp_1 \rangle$$

$$\mathbf{RV}_2 = \langle \{b, c\}, \wp_2 \rangle$$

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 cc ?

Answering this question is problematic.

Definition 10.3 (Probabilistic State of an Agent)

The probabilistic state of an agent \mathbf{a} at any given point t in time, denoted $\mathcal{O}^p(t)$, consists of the set of all instantiated data objects and random variables of types contained in $\mathcal{T}_{\mathbf{a}}$.

Definition 10.4 (Satisfying a Code Call Atom)

Suppose $\mathbf{a} :_{\text{RV}} f(d_1, \dots, d_n)$ is a ground probabilistic code call and o is an object of the output type of this code call w.r.t. probabilistic agent state \mathcal{O}^p . Suppose $[\ell, u]$ is a closed, nonempty subinterval of the unit interval $[0, 1]$.

- $o \models_{\mathcal{O}^p}^{[\ell, u]} \text{in}(X, \mathbf{a} :_{\text{RV}} f(d_1, \dots, d_n))$
if there is a (Y, \wp) in the answer returned by evaluating $\mathbf{a} :_{\text{RV}} f(d_1, \dots, d_n)$ w.r.t. \mathcal{O}^p such that $o \in Y$ and $\ell \leq \wp(o) \leq u$.
- $o \models_{\mathcal{O}^p}^{[\ell, u]} \text{not_in}(X, \mathbf{a} :_{\text{RV}} f(d_1, \dots, d_n))$
if for all random variables (Y, \wp) returned by evaluating $\mathbf{a} :_{\text{RV}} f(d_1, \dots, d_n)$ w.r.t. \mathcal{O}^p , either $o \notin Y$ or $\wp(o) \notin [\ell, u]$.

Probabilistic code call conditions are defined in exactly the same way as code call conditions. However, extending the above definition of “satisfaction” to probabilistic code call conditions is highly problematic because (as shown in Examples 10.2, 10.3)

the probability that a conjunction is true depends not only on the probabilities of the individual conjuncts, but also on the dependencies between the events denoted by these conjuncts.

We allow the user to specify certain strategies.

Definition 10.5 (Probabilistic Conjunction Strategy \otimes)

A *probabilistic conjunction strategy* is a mapping \otimes which maps a pair of probability intervals to a single probability interval satisfying the following axioms:

1. **Bottomline:** $[L_1, U_1] \otimes [L_2, U_2] \leq [\min(L_1, L_2), \min(U_1, U_2)]$
 where $[x, y] \leq [x', y']$ if $x \leq x'$ and $y \leq y'$.
2. **Ignorance:**
 $[L_1, U_1] \otimes [L_2, U_2] \subseteq [\max(0, L_1 + L_2 - 1), \min(U_1, U_2)]$.
3. **Identity:** When $(e_1 \wedge e_2)$ is consistent and $[L_2, U_2] = [1, 1]$,
 $[L_1, U_1] \otimes [L_2, U_2] = [L_1, U_1]$.

4. **Annihilator:** $[L_1, U_1] \otimes [0, 0] = [0, 0]$.

5. **Commutativity:** $[L_1, U_1] \otimes [L_2, U_2] = [L_2, U_2] \otimes [L_1, U_1]$.

6. **Associativity:**

$$([L_1, U_1] \otimes [L_2, U_2]) \otimes [L_3, U_3] = [L_1, U_1] \otimes ([L_2, U_2] \otimes [L_3, U_3]).$$

7. **Monotonicity:** $[L_1, U_1] \otimes [L_2, U_2] \leq [L_1, U_1] \otimes [L_3, U_3]$ if $[L_2, U_2] \leq [L_3, U_3]$.

The concept of a conjunction strategy is very general, and has as special cases, the following well known ways of combining probabilities.

1. When we do not know the dependencies between e_1, e_2 , we may use the conjunction strategy $\otimes_{\mathbf{ig}}$ defined as

$$([L_1, U_1] \otimes_{\mathbf{ig}} [L_2, U_2]) \equiv [\max(0, L_1 + L_2 - 1), \min(U_1, U_2)].$$
2. When e_1, e_2 have maximal overlap, use the positive correlation conjunctive strategy $\otimes_{\mathbf{pc}}$ defined as

$$([L_1, U_1] \otimes_{\mathbf{pc}} [L_2, U_2]) \equiv [\min(L_1, L_2), \min(U_1, U_2)].$$

3. When e_1, e_2 have minimal overlap, use the negative correlation conjunctive strategy $\otimes_{\mathbf{nc}}$ defined as

$$([L_1, U_1] \otimes_{\mathbf{nc}} [L_2, U_2]) \equiv [\max(0, L_1 + L_2 - 1), \max(0, U_1 + U_2 - 1)].$$
4. When the two events occur independently, use the independence conjunction strategy

$$([L_1, U_1] \otimes_{\mathbf{in}} [L_2, U_2]) = [L_1 \cdot L_2, U_1 \cdot U_2].$$

10.2 Probabilistic Agent Programs

We assume the existence of an *annotation language* L^{ann} —the constant symbols of L^{ann} are the reals in the interval $[0, 1]$.

Definition 10.6 (Annotation Item)

We define annotation items inductively:

- Every constant and every variable of L^{ann} is an annotation item.
- If f is an annotation function of arity n and ai_1, \dots, ai_n are annotation items, then the term $f(ai_1, \dots, ai_n)$ is an annotation item.

An annotation item is *ground* if no variables occur in it.

Definition 10.7 (Annotation $[ai_1, ai_2]$)

If ai_1, ai_2 are annotation items, then the term $[ai_1, ai_2]$ is an *annotation*. If ai_1, ai_2 are both ground, then $[ai_1, ai_2]$ is a *ground annotation*.

For instance, $[0, 0.4], [0.7, 0.9], [0.1, \frac{v}{2}], [\frac{v}{4}, \frac{v}{2}]$ are all annotations. The annotation $[0.1, \frac{v}{2}]$ denotes an interval only when a value in $[0, 1]$ is assigned to the variable v .

Definition 10.8 (Annotated Code Call Condition $\chi : \langle [ai_1, ai_2], \otimes \rangle$)

If χ is a probabilistic code call condition, \otimes is a conjunction strategy, and $[ai_1, ai_2]$ is an annotation, then $\chi : \langle [ai_1, ai_2], \otimes \rangle$ is an *annotated code call condition*. $\chi : \langle [ai_1, ai_2], \otimes \rangle$ is *ground* if there are no variables in either χ or in $[ai_1, ai_2]$.

For example, when X is ground,

$\mathbf{in}(X, \mathbf{surv} :_{\mathbf{RV}} \mathit{identify}(\mathit{image1})) \ \& \ \mathbf{in}(a1, \mathbf{surv} :_{\mathbf{RV}} \mathit{turret}(X)) : \langle [0.3, 0.5], \otimes_{\mathbf{ig}} \rangle$

is true *if and only if*

the probability that X is identified by the **surv** agent and that the turret is identified as being of type $a1$ lies between 30 and 50% assuming that nothing is known about the dependencies between turret identifications and identifications of objects by **surv**.

Definition 10.9 (Probabilistic Agent Programs \mathcal{PP})

Suppose Γ is an **annotated code call condition**, and A, L_1, \dots, L_n are status atoms. Then

$$A \leftarrow \Gamma \ \& \ L_1 \ \& \ \dots \ \& \ L_n \quad (10.7)$$

is a *probabilistic action rule*.

A *probabilistic agent program* (pap for short) is a finite set of probabilistic action rules.

It is important to note in the above definition that in a probabilistic action rule, status atoms are *not* annotated—**uncertainty is present only in the state**, and on the basis of this uncertainty, the agent must determine what it is obliged to do, forbidden from doing, etc.

$\text{Do } send_warn(X) \leftarrow \text{in}(F, \text{surv} : \text{file}(\text{imagedb})) \ \&$
 $\text{in}(X, \text{surv} :_{\mathbf{RV}} \text{identify}(F)) \ \&$
 $\text{in}(a1, \text{surv} :_{\mathbf{RV}} \text{turret}(X)) : \langle [0.7, 1.0], \otimes_{ig} \rangle$
 $\neg \mathbf{F} send_warn(X).$

$\mathbf{F} send_warn(X) \leftarrow \text{in}(F, \text{surv} : \text{file}(\text{imagedb})) \ \&$
 $\text{in}(X, \text{surv} :_{\mathbf{RV}} \text{identify}(F)) \ \&$
 $\text{in}(L, \text{geo} :_{\mathbf{RV}} \text{getplnode}(X.\text{location})) \ \&$
 $\text{in}(L, \text{geo} :_{\mathbf{RV}} \text{range}(100, 100, 20)).$

Definition 10.10 (Feasible Probabilistic Status Set)

Suppose \mathcal{PP} is an agent program and \mathcal{O}^p is a probabilistic agent state. A probabilistic status set \mathcal{PS} is *feasible* for \mathcal{PP} on \mathcal{O}^p if the following conditions hold:

($\mathcal{PS}1$): $\mathbf{App}_{\mathcal{PP}, \mathcal{O}^p}(\mathcal{PS}) \subseteq \mathcal{PS}$ (*closure under the program rules*);

($\mathcal{PS}2$): \mathcal{PS} is deontically and action consistent (*deontic/action consistency*);

($\mathcal{PS}3$): \mathcal{PS} is action closed and deontically closed (*deontic/action closure*);

($\mathcal{PS}4$): \mathcal{PS} is state consistent (*state consistency*).

Definition 10.11 (Deontic and Action Consistency)

A probabilistic status set \mathcal{PS} is *deontically consistent* with respect to a probabilistic agent state \mathcal{O}^p if, by definition, it satisfies the following rules for any ground action α :

- If $\mathbf{O}\alpha \in \mathcal{PS}$, then $\mathbf{W}\alpha \notin \mathcal{PS}$.
- If $\mathbf{P}\alpha \in \mathcal{PS}$, then $\mathbf{F}\alpha \notin \mathcal{PS}$.
- If $\mathbf{P}\alpha \in \mathcal{PS}$, then $\mathcal{O}^p \models^{[1,1]} \text{Pre}(\alpha)$.

A probabilistic status set \mathcal{PS} is *action consistent* w.r.t. \mathcal{O}^p if, by definition, for every action constraint of the form

$$\{\alpha_1(\vec{X}_1), \dots, \alpha_k(\vec{X}_k)\} \leftarrow \chi \quad (10.8)$$

either $\mathcal{O}^p \not\models^{[1,1]} \chi$ or $\{\alpha_1(\vec{X}_1), \dots, \alpha_k(\vec{X}_k)\} \not\subseteq \mathbf{Do}(\mathcal{PS})$.

Definition 10.12

Let \mathcal{PP} be a probabilistic agent program, \mathcal{PS} a probabilistic status set and \mathcal{OP} a probabilistic agent state. Assume further that each random variable contains exactly one object with probability 1. Then we can define the following mappings:

Red₁(\cdot), which maps every random variable of the form $\langle \{o_{\mathbf{RV}}\}, 1 \rangle$ to o : $\text{Red}_1(\langle \{o_{\mathbf{RV}}\}, 1 \rangle) = o$.

Red₂(\cdot), which maps annotated code call conditions to code call conditions by simply removing the annotations and the conjunction strategy: $\text{Red}_2(\chi : \langle [a_1, a_2], \otimes \rangle) = \chi$. We can extend $\text{Red}_2(\cdot)$ to a mapping from arbitrary conjunctions of annotated code calls to conjunctions of ccs.

Red₃(·), which maps every probabilistic agent program to a non-probabilistic agent program: it clearly suffices to define Red₃(·) on probabilistic agent rules. This is done as follows

$$\text{Red}_3(A \leftarrow \Gamma \ \& \ L_1 \ \& \ \dots \ \& \ L_n) = A \leftarrow \text{Red}_2(\Gamma) \ \& \ L_1 \ \& \ \dots \ \& \ L_n.$$

Theorem 10.1 (Semantics as an Instance of paps)

Suppose all random variables have the form

$$\langle \{\text{object}_{\text{RV}}\}, 1 \rangle.$$

Then: $(\chi : \langle [\text{ai}_1, \text{ai}_2], \otimes \rangle$ is a ground annotated ccc, \mathcal{O}^p a probabilistic agent state)

Satisfaction: the satisfaction relations coincide, i.e.

$$\mathcal{O}^p \models^{[\text{ai}_1, \text{ai}_2]} \chi : \langle [\text{ai}_1, \text{ai}_2], \otimes \rangle \text{ if and only if } \mathcal{O}^p \models \text{Red}_2(\chi : \langle [\text{ai}_1, \text{ai}_2], \otimes \rangle).$$

App-Operators: the App-Operators coincide, i.e.

$$\mathbf{App}_{\text{Red}_3(\mathcal{PP}), \mathcal{O}^p}(\mathcal{PS}) = \mathbf{App}_{\mathcal{PP}, \mathcal{O}^p}(\mathcal{PS}).$$

Feasibility: Feasible probabilistic status sets coincide with feasible status sets under our reductions, i.e. \mathcal{PS} is a feasible probabilistic status set w.r.t. \mathcal{PP} *if and only if* \mathcal{PS} is a feasible status set w.r.t. $\text{Red}_3(\mathcal{PP})$.

10.3 Kripke Style Semantics

Up to now, we assumed:

- An action can be executed only if its precondition is believed by the agent to be true in the agent state **with probability 1**.
- Every action that is permitted must also have a precondition that is believed to be true **with probability 1**.

Every probabilistic state implicitly determines a set of (ordinary) states that are “compatible” with it.

Definition 10.13 (Compatibility w.r.t. State: $\text{COS}(\mathcal{O}^p)$)

Let \mathcal{O}^p be a probabilistic agent state. An (ordinary) agent state \mathcal{O} is said to be *compatible* with \mathcal{O}^p if, by definition, for every ground code call $\mathbf{a}:f(\mathbf{d}_1, \dots, \mathbf{d}_n)$, it is the case that for every object $o \in \text{eval}(\mathbf{a}:f(\mathbf{d}_1, \dots, \mathbf{d}_n), \mathcal{O})$, there exists a random variable $(X, \wp) \in \text{eval}(\mathbf{a}:\text{RV } f(\mathbf{d}_1, \dots, \mathbf{d}_n), \mathcal{O}^p)$ such that $o \in X$ and $\wp(o) > 0$, and there is no other object $o' \in X$ such that $o' \in \text{eval}(\mathbf{a}:f(\mathbf{d}_1, \dots, \mathbf{d}_n), \mathcal{O})$.

We use the notation $\text{COS}(\mathcal{O}^p)$.

Example 10.4

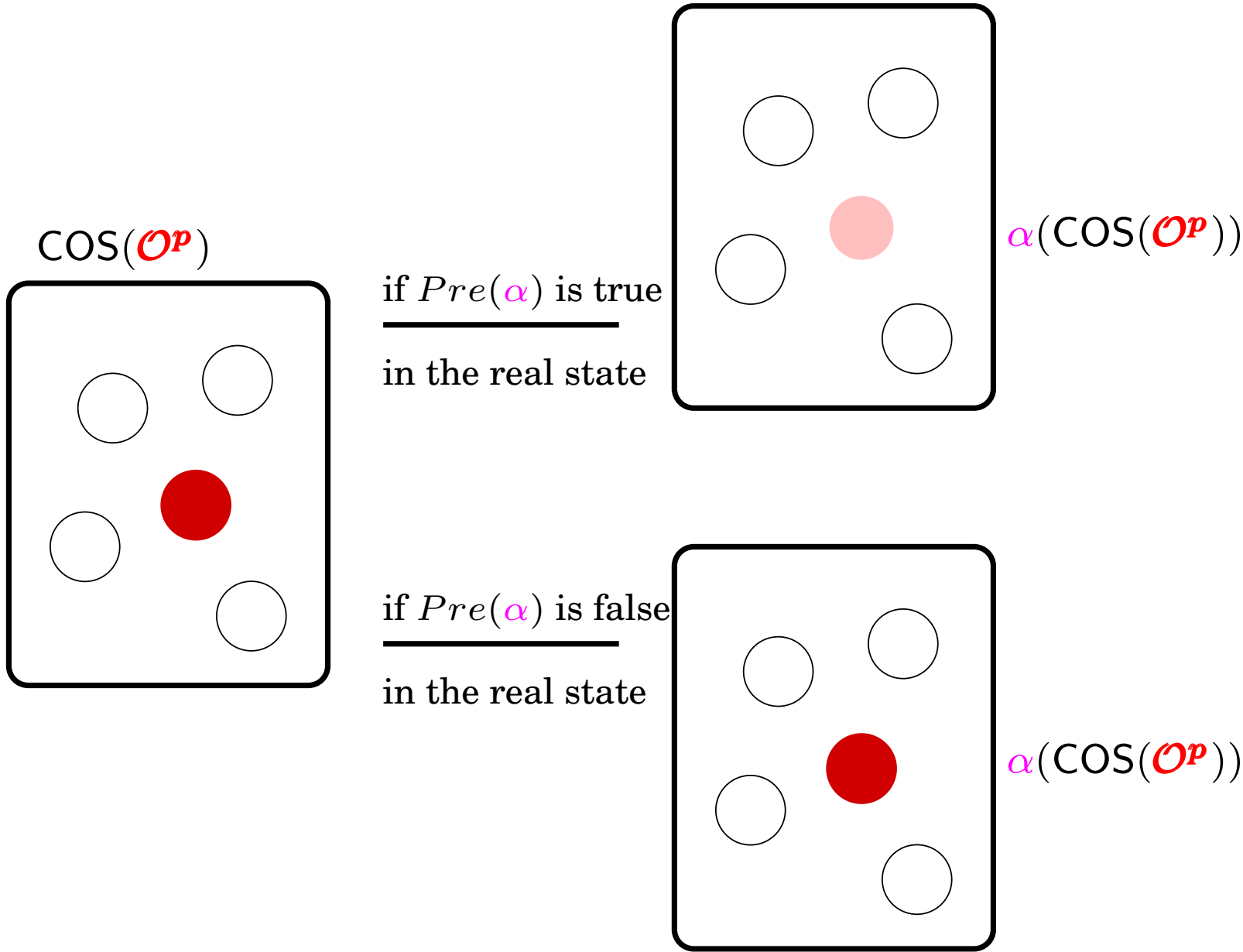
Consider a probabilistic agent state \mathcal{O}^p with only two code calls $\text{surv} : \text{identify}(\text{image1})$ and $\text{surv} : \text{location}(\text{image1})$, which respectively return the random variables

$$\langle \{t80, t72, t70\}, \{ \langle t80, 0.3 \rangle, \langle t72, 0.7 \rangle, \langle t70, 0.0 \rangle \} \rangle$$

and $\langle \{loc2\}, \{ \langle loc2, 0.8 \rangle \} \rangle$. The agent states compatible w.r.t. \mathcal{O}^p are described in the following table:

State	Vehicle	Location
1	none	none
2	t80	none
3	t72	none

State	Vehicle	Location
4	none	loc2
5	†80	loc2
6	†72	loc2



It would be nice if

- agents were able to reason about the effects of their actions even when they are not exactly sure what the world state is.

~→ **Probabilistic Kripke Structures**

- actions could be applied even when the precondition is only true wrt. a certain probability $p < 1$.

~→ **p-Feasible Status Sets**

11. Adding Time

11.1 Timed Actions

11.2 Temporal Agent Programs

11.3 Semantics

Timetable:

- Chapter 11 needs 30 minutes.

11 Adding Time

11.1 Timed Actions

- Most real-world actions have a **duration** :
heli: fly("BA", "US").
- It might be important to specify intermediate timepoints, **checkpoints** (Definition 11.1), and **to update the current state** incrementally at these prespecified points.

Thus, in order to specify a **timed action** , we must:

1. **Specify the total amount of time it takes for the action to be “completed”.**
2. **Specify exactly how the state of the agent changes *while* the action is being executed.**

Definition 11.1 (Checkpoint Expressions $\text{rel}: \{X \mid \chi\}, \text{abs}: \{X \mid \chi\}$)

- If $i \in \mathbb{N}$ is a positive integer, then $\text{rel}: \{i\}$ and $\text{abs}: \{i\}$ are checkpoint expressions.
- If χ is a code call condition involving a non-negative, integer-valued variable X , then $\text{rel}: \{X \mid \chi\}$ and $\text{abs}: \{X \mid \chi\}$ are *checkpoint expressions*.

Example 11.1 (Rescue: Checkpoints)

- $\text{rel}:\{100\}$. This says that a checkpoint occurs at the time of the start of the action, 100 units later, 200 units later, and so on.
- $\text{abs}:\{T \mid \text{in}(T, \text{clock} : \text{time}()) \ \& \ \text{in}(0, \text{math} : \text{remainder}(T, 100)) \ \& \ T > 5000\}$. This says that a checkpoint occurs at absolute times 5000, 5100, 5200, and so on.
- $\text{abs}:\{T \mid \text{in}(T, \text{clock} : \text{time}()) \ \& \ \text{in}(X, a : \text{getMessage}(\text{comc})) \ \& \ X.\text{Time} - T = 5\}$. This says that a checkpoint occurs at 5 time units after a message is received from the **comc** agent.

Definition 11.2 (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.

Example 11.2 (Rescue: Timed Effect Triples)

- The **truck** agent may use the following timed effect triple to update its fuel at absolute times 5000, 5100, 5200, and so on.

1st arg :

abs: $\{T \mid \text{in}(T, \text{clock} : \text{time}()) \ \& \ \text{in}(0, \text{math} : \text{remainder}(T, 100)) \ \& \ T > 5000\}$

2nd arg: $\{\text{in}(\text{NewFuelLevel}, \text{truck} : \text{fuelLevel}(X_{\text{now}})) \}$

3rd arg: $\{\text{in}(\text{OldFuelLevel}, \text{truck} : \text{fuelLevel}(X_{\text{now}} - 20)) \}$

Definition 11.3 (Timed Action)

A *timed action* α consists of:

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

Schema: A schema, usually written as (τ_1, \dots, τ_n) , of types.

Intuitively, this says that the variable X_i must be of type τ_i , for all $1 \leq i \leq n$.

Pre: A code-call condition χ , the *precondition* of the action, denoted by $Pre(\alpha)$

Dur:

An expression of the form $\{i\}$ or $\{x \mid \chi\}$. Depending on the current object state, this expression determines a *duration* $\text{duration}(\alpha) \in \mathbb{N}$ of α .

Tet:

A set $\text{Tet}(\alpha)$ of timed effect triples such that if both $\langle \{chk\}, Add, Del \rangle$ and $\langle \{chk\}', Add', Del' \rangle$ are in $\text{Tet}(\alpha)$, then $\{chk\}$ and $\{chk\}'$ have no common solution w.r.t. any object state. The set $\text{Tet}(\alpha)$ together with $\text{Dur}(\alpha)$ determines the set of checkpoints $\text{checkpoints}(\alpha)$ for action α (as defined below).

Intuitively, if α is an action that we start executing at t_{start}^α , then

- $\text{Dur}(\alpha)$ specifies how to compute the duration $\text{duration}(\alpha)$ of α , and
- $\text{Tet}(\alpha)$ specifies the checkpoints associated with action α .

It is important to note that $\text{Dur}(\alpha)$ and $\text{Tet}(\alpha)$ may not specify the duration and checkpoint times explicitly (even if the associated checkpoints are of the form $\text{abs}:\{X \mid \chi\}$, i.e. absolute times). There is a method to compute $\text{duration}(\alpha)$.

11.2 Temporal Agent Programs

Definition 11.4 (Temporal Annotation Item tai)

1. Every integer is a temporal annotation item.
2. The distinguished integer valued variable x_{now} is a temporal annotation item.
3. Every integer valued variable is a temporal annotation item.
4. If $\text{tai}_1, \dots, \text{tai}_n$ are temporal annotation items, and b_1, \dots, b_n are integers (positive or negative), then $(b_1 \text{tai}_1 + \dots + b_n \text{tai}_n)$ is a temporal annotation item.

- 1 , x_{now} , $x_{\text{now}} + 3$, $x_{\text{now}} + 2v + 4$ are all temporal annotation items if v is an integer valued variable.
- Temporal annotation items, when ground, evaluate to time points. They are used to specify a time interval.

Definition 11.5 (Temporal Annotation $[\mathbf{tai}_1, \mathbf{tai}_2]$)

If $\mathbf{tai}_1, \mathbf{tai}_2$ are annotation items, then $[\mathbf{tai}_1, \mathbf{tai}_2]$ is a temporal annotation.

- $[2, 5]$ is a temporal annotation item describing the set of time points between 2 and 5 (inclusive).
- $[2, 3X + 4Y]$ is a temporal annotation.
- When $X := 2, Y := 3$, this defines the set of time points between 2 and 18. $[X_{\text{now}}, X_{\text{now}} + 5]$ is a temporal annotation.

Definition 11.6 ((Temporal) Action State Condition)

Suppose χ is a (possibly empty) code call condition, L_1, \dots, L_n are action status literals, and ta is a temporal annotation. Then:

1. $(\chi \& L_1 \& \dots \& L_n)$ is called an *action state condition*.
2. $(\chi \& L_1 \& \dots \& L_n) : \text{ta}$ is called a *temporal action state conjunct* (*tasc*).
3. If χ is empty, then $(L_1 \& \dots \& L_n) : \text{ta}$ is called a *state-independent tasc*.

Intuitively, when $\varrho:\text{ta}$ is ground for some action state condition ϱ , we may read this as “ ϱ is true at some point in ta ”. The following is a simple tASC .

- $(\text{in}(\text{X}, \text{heli}:\text{inventory}(\text{fuel})) \ \& \ \text{X.Qty} < 50) : [\text{X}_{\text{now}} - 10, \text{X}_{\text{now}}]$.

Intuitively, this tASC is true if at some point in time t_i in the last 10 time units, the helicopter had less than 50 gallons of fuel left.

We are now ready to define the most important syntactic construct of this chapter, a *temporal agent rule*.

Definition 11.7 (Temporal Agent Rule/Program \mathcal{TP})

A *temporal agent rule* is an expression of the form

$$\text{Op } \alpha : [\text{tai}_1, \text{tai}_2] \leftarrow \varrho_1 : \text{ta}_1 \& \cdots \& \varrho_n : \text{ta}_n,$$

where $\text{Op} \in \{\mathbf{P}, \mathbf{Do}, \mathbf{F}, \mathbf{O}, \mathbf{W}\}$, and $\varrho_1 : \text{ta}_1, \dots, \varrho_n : \text{ta}_n$ are tascS.

A *temporal agent program* is a finite set of temporal agent rules.

Intuitive Reading of Temporal Agent Rule

“If for all $1 \leq i \leq n$, there exists a time point t_i such that ϱ_i is true at time t_i such that either

- 1. ϱ_i is state independent and $t_i \in \text{ta}_i$, or*
- 2. ϱ_i is not state independent and $t_i \leq \text{t}_{\text{now}}$ (i.e. t_i is now or is in the past) and $t_i \in \text{ta}_i$,*

then $\text{Op}\alpha$ is true at some point $t \geq \text{t}_{\text{now}}$ (i.e. now or in the future) such that $\text{tai}_1 \leq t \leq \text{tai}_2$ ”.

$$\text{Op}\alpha : [\text{tai}_1, \text{tai}_2] \leftarrow \varrho_1 : \text{ta}_1 \& \cdots \& \varrho_n : \text{ta}_n,$$

“If a prediction package expects a stock to rise $K\%$ after T_K units of time and $K \geq 25$ then buy the stock at time $(X_{\text{now}} + T_K - 2)$.”

We assume a prediction package that given a stock uses (some stock expertise) to predict the change in the value of the stock at future time points. This function returns a set of pairs of the form (T, C) . Intuitively, this says that T units from now, the stock price will change by C percent (positive or negative).

Do *buy* $S: [X_{\text{now}} + X.T - 2, X_{\text{now}} + X.T - 2] \leftarrow$
 $(\text{in}(X, \text{pred} : \text{dest}(S)) \ \& \ X.C \geq 25) : [X_{\text{now}}, X_{\text{now}}]$.

11.3 Semantics

Definition 11.8 (Temporal Status Set $TS_{t_{\text{now}}}$)

A temporal status set $TS_{t_{\text{now}}}$ at time t_{now} is a mapping from natural numbers to ordinary status sets satisfying $TS_{t_{\text{now}}}(i) = \emptyset$ for all $i > i_0$ for some $i_0 \in \mathbb{N}$.

As usual a feasible status set must satisfy

- Closure under rules.
- Deontic consistency wrt. **State History** (\leadsto Definition 11.9).
- Deontic closure.
- **Checkpoint consistency** (\leadsto Definition 11.10).

As an agent that reasons about time may need to reason about the current, as well as past states it was/is in, a notion of state history is needed.

Definition 11.9 (State History Function $\text{hist}_{t_{\text{now}}}$)

A *state history function* $\text{hist}_{t_{\text{now}}}$ at time t_{now} is a partial function from \mathbb{N} to agent states such that $\text{hist}_{t_{\text{now}}}(t_{\text{now}})$ is always defined and for all $i > t_{\text{now}}$, $\text{hist}_{t_{\text{now}}}(i)$ is undefined.

The definition of state history does not *require* that an agent store the entire past.

1. He may decide to store no past information at all. In this case, $\text{hist}_{t_{\text{now}}}(i)$ is defined *if and only if* $i = t_{\text{now}}$.
2. He may decide to store information only about the past i units of time. This means that he stores the agent's state at times $t_{\text{now}}, (t_{\text{now}} - 1), \dots, (t_{\text{now}} - i)$, i. e. $\text{hist}_{t_{\text{now}}}$ is defined for the following arguments: $\text{hist}_{t_{\text{now}}}(t_{\text{now}})$, $\text{hist}_{t_{\text{now}}}(t_{\text{now}} - 1), \dots, \text{hist}_{t_{\text{now}}}(t_{\text{now}} - i)$ are defined.

3. He may decide to store, in addition to the current state, the history every five time units. That is, $\text{hist}_{t_{\text{now}}}(t_{\text{now}})$ is defined and for each $0 \leq i \leq t_{\text{now}}$, if $i \bmod 5 = 0$, then $\text{hist}_{t_{\text{now}}}(i)$ is defined. Such an agent may be specified by an agent designer when he believes that maintaining some (but not all) past snapshots is adequate for his application's needs.

For a temporal status set to be feasible, at each checkpoint the state needs to be updated.
The expected future states of the agent need to satisfy the integrity constraints.

Definition 11.10 (Checkpoint Consistency)

$\mathcal{TS}_{t_{\text{now}}}$ is said to be *checkpoint consistent* at time t_{now} if, by definition, for all $i > t_{\text{now}}$, $\mathcal{EO}(i)$ (see Definition 11.11) satisfies the integrity constraints \mathcal{IC} .

Definition 11.11 (Expected States at time t : $\mathcal{EO}(t)$)

Suppose the current time is t_{now} , $\text{hist}_{t_{\text{now}}}$ is the agent's state history function and $TS_{t_{\text{now}}}$ is a temporal status set. The agent's expected states are defined as follows:

- $\mathcal{EO}(t_{\text{now}}) = \text{hist}_{t_{\text{now}}}(t_{\text{now}})$.
- For all time points $i > t_{\text{now}}$, $\mathcal{EO}(i)$ is the result of concurrently executing

$$\{\alpha \mid \mathbf{Do} \alpha \in TS_{\text{now}}(i - 1)\} \cup$$

$$\{\beta' \mid \mathbf{Do} \beta \in TS_{\text{now}}(j) \text{ for } j \leq i - 1 \text{ and } i - 1 \text{ is a checkpoint for } \beta,$$

and β' denotes the action (non-timed) which has

an empty precondition, and whose add and del lists are as specified by $\text{Tet}(\beta)\}$

in state $\mathcal{EO}(i - 1)$.

We note that that from a certain $i_0 \in \mathbb{N}$ onwards, we have $\mathcal{EO}(i) = \emptyset$ for all $i > i_0$ (this is because of the same property for the action history and the temporal status set).

References

Combining Agents, ASP and Planning, NICTA

References

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.

Arisha, K., F. Ozcan, R. Ross, V. Subrahmanian, T. Eiter, and S. Kraus (1999, March/April). IMPACT: A Platform for Collaborating Agents. *IEEE Intelligent Systems* 14, 64–72.

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.

Brass, S. and J. Dix (1994). A disjunctive semantics based on unfolding and bottom-up evaluation. In B. Wolfinger (Ed.), *Innovationen bei Rechen- und Kommunikationssystemen*, (IFIP ’94-Congress, Workshop FG2: Disjunctive Logic Programming and Disjunctive

626

Databases), Berlin, pp. 83–91. Springer.

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

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

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.

Currie, K. and A. Tate (1991). O-plan: the open planning architecture. *Artificial Intelligence* 52(1).

Dix, J., T. Eiter, M. Fink, A. Polleres, and Y. Zhang (2003). Monitoring Agents using Declarative Planning. In R. Kruse (Ed.), *Proceedings of the 27th German Annual Conference on Artificial Intelligence (KI*

'03), *Hamburg, Germany*, LNAI ???, Berlin. Springer.

Dix, J., T. Eiter, M. Fink, A. Polleres, and Y. Zhang (2004). Monitoring Agents using Declarative Planning. *Fundamenta Informaticae*, to appear.

Dix, J., S. Kraus, and V. Subrahmanian (2001). Temporal agent reasoning. *Artificial Intelligence* 127(1), 87–135.

Dix, J., S. Kraus, and V. Subrahmanian (2002, July). Agents dealing with time and uncertainty. In C. Castelfranchi and W. L. Johnson (Eds.), *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*. New York: ACM Press.

Dix, J., U. Kuter, and D. Nau (2003). Planning in answer set programming using ordered task decomposition. In R. Kruse (Ed.), *Proceedings of the 27th German Annual Conference on Artificial Intelligence (KI '03), Hamburg, Germany*, LNAI ???, Berlin. Springer.

-
- Dix, J. and M. Müller (1994a). Partial Evaluation and Relevance for Approximations of the Stable Semantics. In Z. Ras and M. Zemankova (Eds.), *Proceedings of the 8th Int. Symp. on Methodologies for Intelligent Systems, Charlotte, NC, 1994*, LNAI 869, Berlin, pp. 511–520. Springer.
- Dix, J. and M. Müller (1994b). The Stable Semantics and its Variants: A Comparison of Recent Approaches. In L. Dreschler-Fischer and B. Nebel (Eds.), *Proceedings of the 18th German Annual Conference on Artificial Intelligence (KI '94), Saarbrücken, Germany*, LNAI 861, Berlin, pp. 82–93. Springer.
- Dix, J., H. Munoz-Avila, and D. N. an Lingling Zhang (2002). Theoretical and Empirical Aspects of a Planner in a Multi-Agent Environment. In G. Ianni and S. Flesca (Eds.), *Proceedings of Journees Europeens de la Logique en Intelligence artificielle (JELIA '02)*, LNCS 2424, pp. 173–185. Springer.
-
- Dix, J., H. Munoz-Avila, D. Nau, and L. Zhang (2002, July). Planning in
- 629**

-
- a multi-agent environment: Theory and practice. In C. Castelfranchi and W. L. Johnson (Eds.), *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*. New York: ACM Press.
- Dix, J., H. Munoz-Avila, D. Nau, and L. Zhang (2003). IMPACTing SHOP: Putting an AI planner into a Multi-Agent Environment. *Annals of Mathematics and AI* 37(4), 381–407.
- Dix, J., M. Nanni, and V. S. Subrahmanian (2000). Probabilistic agent reasoning. *ACM Transactions of Computational Logic* 1(2), 201–245.
- Dix, J., V. Subrahmanian, and G. Pick (2000). Meta Agent Programs. *Journal of Logic Programming* 46(1-2), 1–60.
- Eiter, T., W. Faber, N. Leone, G. Pfeifer, and A. Polleres (2002). A Logic Programming Approach to Knowledge-State Planning, II: The DLV^{κ} System. *Artificial Intelligence* 144(1-2), 157–211.
- Eiter, T. and V. Subrahmanian (1999). Heterogeneous Active Agents, II: Algorithms and Complexity. *Artificial Intelligence* 108(1-2),
-
- 630

- Eiter, T., V. Subrahmanian, and G. Pick (1999). Heterogeneous Active Agents, I: Semantics. *Artificial Intelligence* 108(1-2), 179–255.
- Eiter, T., V. Subrahmanian, and T. Rogers (2000). Heterogeneous Active Agents, III: Polynomially Implementable Agents. *Artificial Intelligence* 117(1), 107–167.
- Franklin, S. and A. Graesser (1997). Is it an Agent, or Just a Program? In J. P. Müller, M. Wooldridge, and N. R. Jennings (Eds.), *Intelligent Agents III*, Berlin, Germany. Springer-Verlag. LNAI Volume 1193.
- Genesereth, M. R. and S. P. Ketchpel (1994). Software Agents. *Communications of the ACM* 37(7), 49–53.
- 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.
- Munoz-Avila, H., D. Aha, D. Nau, R. Weber, L. Breslow, and F. Yaman (2001). Sin: Integrating case-based reasoning with task

decomposition. In *Proceedings of IJCAI-01*.

Nau, D., Y. Cao, A. Lotem, and H. Muñoz-Avila (1999). Shop: Simple hierarchical ordered planner. In *Proceedings of IJCAI-99*.

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.

Sacerdoti, E. (1977). *A Structure for Plans and Behavior*. American Elsevier Publishing.

Sakama, C. and H. Seki (1994). Partial Deduction of Disjunctive Logic Programs: A Declarative Approach. In *Logic Program Synthesis and Transformation – Meta Programming in Logic*, LNCS 883, Berlin, pp. 170–182. Springer.

Son, T., C. Baral, and S. McIlraith. (2001, September). Planning with domain-dependent knowledge of different kinds – an answer set programming approach. In T. Eiter, M. Truszczyński, and W. Faber (Eds.), *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Sixth International Conference*, LNCS 2173, Berlin, pp. 226–239. Springer.

Subrahmanian, V., P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Özcan, and R. Ross (2000). *Heterogenous Active Agents*. MIT-Press.

Tate, A. (1977). Generating Project Networks. In *Proc. IJCAI-77*, pp.

888–893.

Weiss, G. (Ed.) (1999). *Multi-Agent Systems*. 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.

Wilkins, D. (1988). *Practical planning - extending the classical AI planning paradigm*. 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.