8. Implementing Agents

Overview

- We define (in Section 8.1), a class of agents called *weak regular agents* that serve as a stepping stone to later defining regular agents.
- Derive (in Section 8.2) various theoretical properties of weak regular agents that make the design of a computation procedure to compute regular agents polynomial.
- Extend (in Section 8.3) the definition of weak regular agents to define regular agents—the central contribution of this Section.

8.1 Weakly Regular Agents

WRAPs are characterized by three basic properties:

Strong Safety: In addition to the safety requirement on rules introduced in Section 3 (Definition 3.8), code call conditions are required to satisfy some additional conditions which ensure that they always return finite answers.

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

Deontic Stratifiability: This is a property in the spirit of stratification in logic programs (Apt, Blair, and Walker 1988), which prevents problems with negation in rule bodies. However, deontic stratification is more complex than ordinary stratification (due to deontic modalities).

8.1.1 Strong Safety

Safety is a *compile-time* check that ensures that all code calls generated at *run-time* have instantiated parameters. However, executability of a code call condition does not depend solely on safety. For example, consider the simple code call condition

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

- Though this code call condition is safe, it leads to an infinite set of possible answers, leading to non-termination.

Consider, for instance, the code call condition

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

Clearly, over the integers there are only finitely many ground substitutions that cause this code call condition to be true. Furthermore, this code call condition is safe.

However, its evaluation may never terminate. The reason for this is that safety requires that we first compute the set of all integers that are greater than 25, leading to an infinite computation.

This means that in general, we must impose some restrictions on code call conditions to ensure that they are finitely evaluable.

As is well known, determining whether a function is finite or not is undecidable (Rogers Jr. 1967), and hence, input from the agent developer is imperative.

Definition 8.1 (Binding Pattern)
Suppose we consider a code call $\mathcal{S}:f(\mathtt{a}_1,\ldots,\mathtt{a}_\mathtt{n})$ *where each* $\mathtt{a}_\mathtt{i}$ *is of type* τ_i*. A* binding pattern *for* $\mathcal{S}:f(\mathtt{a}_1,\ldots,\mathtt{a}_\mathtt{n})$ *is an* n*-tuple* $(bt_1,\ldots,bt_n)$ *where each* bt_i *(called a* binding term*) is either:*

1. *A value of type* τ_i*, or*

2. *The expression* $\flat$ *denoting that this argument is bound to an unknown value.*

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

$$\langle \mathcal{S}:f(\mathtt{a}_1,\mathtt{a}_2,\mathtt{a}_3),(\flat,5,\flat)\rangle$$

in the finiteness table. Then this row says that the answer returned by any code call of the form $\mathcal{S}:f(-,5,-)$ is finite.

Example 8.1 (Finiteness Table for AutoPilot Agent in CFIT Example)
An example of a finiteness table is given below.

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

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

The code call $\mathrm{autoPilot}:\mathit{calculateNFlightRoutes}(\texttt{CurrentLocation},\texttt{No\_go},\texttt{N})$ *returns up to* $\texttt{N}$ *flight routes when* $\texttt{N} \neq 0$*. If* $\texttt{N} = 0$*, then an infinite number of flight routes (which start at* $\texttt{CurrentLocation}$ *and avoid the given* $\texttt{No\_go}$ *areas) may be returned. Our finiteness table above indicates that when* $1 \leq \texttt{N} \leq 3$*,* $\mathrm{autoPilot}:\mathit{calculateNFlightRoutes}()$ *will only return a finite number of answers. Notice that this table is incomplete since it does not indicate that a finite number of answers will be returned when* $\texttt{N} > 3$*.*

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

Definition 8.2 (Ordering on Binding Patterns)
We say a binding pattern $(bt_1, \ldots, bt_n)$ *is* equally or less informative *than another binding pattern* $(bt'_1, \ldots, bt'_n)$ *if, by definition, for all* $1 \le i \le n$, $bt_i \le bt'_i$.

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

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

Definition 8.3 (Finiteness)
Suppose FINTAB is a finite finiteness table , and $(bt_1,\ldots,bt_n)$ *is a binding pattern associated with the code call* $\mathcal{S}:f(\cdots)$*. Then FINTAB is said to* entail the finiteness of $\mathcal{S}:f(\mathtt{bt_1},\ldots,\mathtt{bt_n})$ *if, by definition, there exists an entry of the form* $\langle \mathcal{S}:f(\ldots),(bt'_1,\ldots,bt'_n)\rangle$ *in FINTAB such that* $(bt_1,\ldots,bt_n)$ *is more informative than* $(bt'_1,\ldots,bt'_n)$*.*

Example 8.2 (Finiteness Table)
Let FINTAB be the finiteness table given in Example 8.1 on page 363. Then FINTAB entails the finiteness of $\mathrm{autoPilot}{:}\mathit{readGPSData}(\mathtt{5})$ *and* $\mathrm{autoPilot}{:}\mathit{calculateNFlightRoutes}(\langle\mathtt{221},\mathtt{379},\mathtt{433}\rangle,\emptyset,\mathtt{2})$ *but it does not entail the finiteness of* $\mathrm{autoPilot}{:}\mathit{calculateNFlightRoutes}(\langle\mathtt{221},\mathtt{379},\mathtt{433}\rangle,\emptyset,\mathtt{0})$ *(since this may have an infinite number of answers) or* $\mathrm{autoPilot}{:}\mathit{calculateNFlightRoutes}(\langle\mathtt{221},\mathtt{379},\mathtt{433}\rangle,\emptyset,\mathtt{5})$ *(since FINTAB is not complete).*

- We have now defined a condition to ensure finiteness of a code call of the form $\mathcal{S}:f(\ldots)$.
- Defining strong safety of a code call **condition** is more complex. For instance, even if we know that $\mathcal{S}:f(\mathtt{t_1},\ldots,\mathtt{t_n})$ is finite, the code call atom **not_in**$(\mathtt{X},\mathcal{S}:f(\mathtt{t_1},\ldots,\mathtt{t_n}))$ may have an infinite answer. Likewise for comparison conditions.

We make two simplifying assumptions, though both of them can be easily modified to handle other cases:

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

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

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

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

Definition 8.4 (Strong Safety)

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

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

   (a) *If* t_i *is a value, then* $bt_i = t_i$.

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

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

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

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

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

Algorithm **safe_ccc** defined in Section 3 may easily be modified to handle a strong safety check, by replacing the test "select all $\chi_{i_1}, \ldots, \chi_{i_m}$ from L such that χ_{i_j} is safe modulo $\vec{X}$" in step (4) of that algorithm by the test "select all $\chi_{i_1}, \ldots, \chi_{i_m}$ from L such that χ_{i_j} is <u>strongly</u> safe modulo $\vec{X}$."

Definition 8.5 (Strongly Safe Agent Program)
A rule r is strongly safe *if, by definition, it is safe, and $B_{cc}(r)$ is a strongly safe code call condition. An agent program is* strongly safe *if, by definition, all rules in it are strongly safe.*

8.1.2 Conflict-Freedom

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

Definition 8.6 (Conflicting Modalities)
Given two action modalities $Op, Op' \in \{\mathbf{P}, \mathbf{F}, \mathbf{O}, \mathbf{Do}, \mathbf{W}\}$ we say that Op conflicts with *Op' if, by definition, there is an entry "$\times$" in the following table at row Op and column Op':*

$Op \setminus Op'$	$\mathbf{P}$	$\mathbf{F}$	$\mathbf{O}$	$\mathbf{W}$	$\mathbf{Do}$
$\mathbf{P}$		$\times$			
$\mathbf{F}$	$\times$		$\times$		$\times$
$\mathbf{O}$		$\times$		$\times$	
$\mathbf{W}$			$\times$		
$\mathbf{Do}$		$\times$			

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

Definition 8.7 (Conflicting Action Status Literals)
Suppose L_i, L_j are two action status literals. L_i is said to conflict with *L_j if, by definition,*

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

$Op \setminus \neg Op'$	$\neg\mathbf{P}$	$\neg\mathbf{F}$	$\neg\mathbf{O}$	$\neg\mathbf{W}$	$\neg\mathbf{Do}$
$\mathbf{P}$	$\times$				
$\mathbf{F}$		$\times$			
$\mathbf{O}$	$\times$		$\times$		$\times$
$\mathbf{W}$				$\times$	
$\mathbf{Do}$	$\times$				$\times$

- the action status atoms $\mathbf{F}\alpha(a,b,X)$ and $\mathbf{P}\alpha(Z,b,c)$ conflict. However, $\mathbf{F}\alpha(a,b,X)$ and $\neg\mathbf{P}\alpha(Z,b,c)$ do not conflict.
- $\neg\mathbf{P}\alpha(Z,b,c)$ and $\mathbf{Do}\,\alpha(Z,b,c)$ conflict, while the literals $\mathbf{P}\alpha(Z,b,c)$ and $\neg\mathbf{Do}\,\alpha(Z,b,c)$ do not conflict.

The conflicts-with relation is *not* symmetric when applied to action status literals.

A definition expressing that an agent program does not conflict, not must apply just to the current state, but rather to all possible states the agent can be in.

Definition 8.8 (Conflicting Rules w.r.t. a State)
Consider two rules r_i, r_j (whose variables are standardized apart) having the form

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

We say that r_i and r_j conflict *w.r.t. an agent state O_S if, by definition, Op_i conflicts with Op_j, and there is a substitution θ such that:*

- $\alpha(\vec{t}\theta) = \beta(\vec{t'}\theta)$ *and*
- $(B_{cc}(r_i) \wedge B_{cc}(r_j))\theta\gamma$ *is true in O_S for some substitution γ that causes $(B_{cc}(r_i) \wedge Bcc(r_j))\theta$ to become ground and*

- *If* $Op_i \in \{\mathbf{P}, \mathbf{Do}, \mathbf{O}\}$ *(resp.,* $Op_j \in \{\mathbf{P}, \mathbf{Do}, \mathbf{O}\}$*) then* $\alpha(\vec{t}\theta)$ *(resp.,* $\beta(\vec{t'}\theta)$*) is executable in* $O_{\mathcal{S}}$*, and*
- $(B_{as}(r_i) \cup B_{as}(r_j))\theta$ *contains no pair of conflicting action status literals.*

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

Definition 8.9 (Conflict Free)
An agent program, $\mathcal{P}$, is said to be conflict free *if and only if it satisfies two conditions:*

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

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

Theorem 8.1 (Undecidability of Conflict Freedom Checking)
The problem of deciding whether an input agent program $\mathcal{P}$ satisfies the first condition of conflict-freedom is undecidable. Hence, the problem of deciding whether an input agent program $\mathcal{P}$ is conflict free is undecidable.

However, there are many possible ways to define *sufficient* conditions on agent programs that guarantee conflict freedom. If an agent developer encodes his agent program in a way that satisfies these sufficient conditions, then he is guaranteed that his agent is going to be conflict free.

Definition 8.10 (Conflict-Freedom Test)
A conflict-freedom test *is a function cft that takes as input any two rules r_1, r_2, and provides a boolean output such that: if cft$(r_1, r_2) = \mathbf{true}$, then the pair r_1, r_2 satisfies the first condition of conflict freedom.*

Definition 8.11 (Conflict-Free Agent Program w.r.t. cft)
An agent program $\mathcal{P}$ *is* conflict free w.r.t. cft *if and only if for all pairs of distinct rules* $r_i, r_j \in \mathcal{P}$, $\mathit{cft}(r_i, r_j) = \mathbf{true}$, *and all rules in* $\mathcal{P}$ *satisfy the second condition in the definition of conflict free programs.*

Intuitively, different choices of the function cft may be made, depending upon the complexity of such choices, and the accuracy of such choices (i.e. how often does a specific function cft return "**false**" on arguments (r_i, r_j) when in fact r_i, r_j do not conflict?).

In *IADE*, the agent developer can choose one of several conflict-freedom tests to be used for his application (and he can add new ones to his list). Some instances of this test are given below.

Example 8.3 (Head-CFT, cft_h)
Let r_i,r_j be two rules of the form

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

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

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

Example 8.4 (Body Code Call CFT, cft_{bcc})

Let us continue using the same notation as in Example 8.3 on the preceding page. Now let the body-code conflict-freedom test cft_{bcc} be as follows,

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

The expression "there exist a pair of contradictory code call atoms in $B_{cc}(r_1\theta), B_{cc}(r_2\theta)$" means that there exist code call atoms of form **in**$(\mathtt{X}, \mathtt{cc})$ *and* **not_in**$(\mathtt{X}, \mathtt{cc})$ *which occur in $B_{cc}(r_1\theta) \cup B_{cc}(r_2\theta)$, or comparison atoms of the form $\mathtt{s}_1 = \mathtt{s}_2$ and $\mathtt{s}_1 \neq \mathtt{s}_2$; $\mathtt{s}_1 < \mathtt{s}_2$ and $\mathtt{s}_1 \geq \mathtt{s}_2$ etc.*

Example 8.5 (Body-Modality-CFT, cft$_{bm}$)

The body-modality conflict-freedom test is similar to the previous one, except that action status atoms are considered instead. Now let cft$_{bm}$ *be as follows,*

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

Example 8.6 (Precondition-CFT, cft$_{pr}$)

Often, we might have action status atoms of the form $\mathbf{P}\alpha, \mathbf{Do}\,\alpha, \mathbf{O}\alpha$ *in a rule. For a rule* r_i *as shown in Example 8.3 on page 386, denote by* $r_i^\star$ *the new rule obtained by appending to* $B(i)$ *the precondition of any action status atom of the form* $\mathbf{P}\alpha, \mathbf{Do}\,\alpha, \mathbf{O}\alpha$ *(appropriately standardized apart) from the head or body of* r_i*. Thus, suppose* r *is the rule*

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

Suppose $pre(\alpha(X,Y)) = \mathbf{in}(\mathtt{Y}, \mathrm{d}_1 : f_1(\mathtt{X}))$ *and* $pre(\beta) = \mathbf{in}(\mathtt{3}, \mathrm{d}_2 : f_2())$*. Then* $r^\star$ *is the rule*

$$\begin{aligned}\mathbf{Do}\,\alpha(X,Y) \quad \leftarrow \quad & \mathbf{in}(\mathtt{X}, \mathrm{d}:f(\mathtt{Y})) \,\&\, \mathbf{in}(\mathtt{Y}, \mathrm{d}_1 : f_1(\mathtt{X})) \,\&\, \mathbf{in}(\mathtt{3}, \mathrm{d}_2 : f_2()) \,\&\\ & \mathbf{P}\beta \,\&\, \mathbf{F}\gamma(Y).\end{aligned}$$

We now define cft$_{pr}$ *as follows.*

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

Theorem 8.2
Suppose r is a rule, and $\alpha(\vec{X})$ is an action such that some atom Op$\alpha(\vec{t})$ appears in r's body where Op $\in \{\mathbf{P}, \mathbf{O}, \mathbf{Do}\}$. Then:

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

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

8.1.3 Deontic Stratification

Definition 8.12 (Layering Function)
Let $\mathcal{P}$ be an agent program. A layering function *ℓ is a function $\ell : \mathcal{P} \to \mathcal{N}$.*

A layering function assigns a nonnegative integer to each rule in the program, and in doing so, it groups rules into layers as defined below.

Definition 8.13 (Layers of an Agent Program)
If $\mathcal{P}$ is an agent program, and ℓ is a layering function over $\mathcal{P}$, then the i-th layer of $\mathcal{P}$ w.r.t. ℓ, denoted $\mathcal{P}_i^\ell$, is defined as:

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

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

Example 8.7 (Layering Functions)

Consider the agent program $\mathcal{P}$ given below.

r_1: $\mathbf{Do}$ $\mathit{execute\_flight\_plan}(\mathtt{Flight\_route}) \leftarrow$
$\mathsf{in}(\mathtt{automated}, \mathtt{autoPilot}:\mathit{pilotStatus}(\mathtt{pilot\_message}))$,
$\mathbf{Do}$ $\mathit{create\_flight\_plan}(\mathtt{No\_go}, \mathtt{Flight\_route}, \mathtt{Current\_location})$

If the plane is on autopilot and a flight plan has been created, then execute it.

r_2: $\mathbf{O}$ $\mathit{create\_flight\_plan}(\mathtt{No\_go}, \mathtt{Flight\_route}, \mathtt{Current\_location}) \leftarrow$
$\mathbf{O}$ $\mathit{adjust\_course}(\mathtt{No\_go}, \mathtt{Flight\_route}, \mathtt{Current\_location})$

If our agent is required to adjust the plane's course, then it is also required to create a flight plan.

r_3: $\mathbf{O}$ $\mathit{maintain\_course}(\mathit{no\_go}, \mathit{flight\_route}, \mathit{current\_location}) \leftarrow$
$\mathsf{in}(\mathtt{automated}, \mathtt{autoPilot}:\mathit{pilotStatus}(\mathtt{pilot\_message}))$,
$\neg$ $\mathbf{O}$ $\mathit{adjust\_course}(\mathit{no\_go}, \mathit{flight\_route}, \mathit{current\_location})$

If the plane is on autopilot and our agent is not obliged to adjust the plane's *course, then our agent must ensure that the plane maintains its current course.*

r_4: $\mathbf{O}$ *adjust_course(no_go, flight_route, current_location)* $\leftarrow$
$\mathbf{O}$ *adjustAltitude*(`Altitude`)

If our agent must adjust the plane's altitude, this it is obliged to also adjust the plane's flight route as well.

Note that for simplicity, these rules use constant valued parameters for *maintain_course* and *adjust_course*.

Let function ℓ_1 assign 0 to rule r_4, 1 to rules r_2, r_3, and 2 to rule r_1. Then ℓ_1 is a layering function which induces the program layers $\mathcal{P}_0^{\ell_1} = \{r_4\}$, $\mathcal{P}_1^{\ell_1} = \{r_2, r_3\}$, and $\mathcal{P}_2^{\ell_1} = \{r_1\}$. Likewise, the function ℓ_2 which assigns 0 to rule r_4 and 1 to the remaining rules is also a layering function. In fact, the function ℓ_3 which assigns 0 to all rules in $\mathcal{P}$ is also a layering function.

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

Definition 8.14 (Modality Ordering)
The partial ordering "$\leq$" on the set of deontic modalities $M = \{\mathbf{P}, \mathbf{O}, \mathbf{Do}, \mathbf{W}, \mathbf{F}\}$ is defined as follows (see Figure 8.1): $\mathbf{O} \leq \mathbf{Do}$, $\mathbf{O} \leq \mathbf{P}$, $\mathbf{Do} \leq \mathbf{P}$, and $Op \leq Op$, for each $Op \in M$. Furthermore, for ground action status atoms A and B, we define that $A \leq B$ if, by definition, $A = Op\alpha$, $B = Op'\alpha$, and $Op' \leq Op$ all hold.

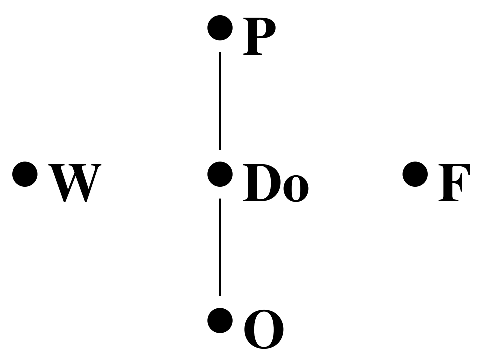


Figure 8.1: Modality ordering

Definition 8.15 (Deontically Stratifiable Agent Program)
An agent program $\mathcal{P}$ is deontically stratifiable *if, by definition, there exists a layering function ℓ such that:*

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

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

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

Example 8.8 (Deontic Stratifiability)
Consider the agent program and layer functions given in Example 8.7 on page 392. Then the first condition of deontic stratifiability requires $\ell(r_2) \leq \ell(r_1)$ and $\ell(r_4) \leq \ell(r_2)$. Also, the second condition of deontic stratifiability requires $\ell(r_4) < \ell(r_3)$. Thus, ℓ_1 and ℓ_2 (but not ℓ_3) are witnesses to the stratifiability of $\mathcal{P}$.

Note that some agent programs are not deontically stratifiable. For instance, let $\mathcal{P}'$ contain the following rule:

r_1': $\mathbf{Do}$ $compute\_currentLocation$(report) $\leftarrow$
$\neg$ $\mathbf{Do}$ $compute\_currentLocation$ (report)

Here, the author is trying to ensure that a plane's current location is always computed. The problem is that the second condition of deontic stratifiability requires $\ell(r_1') < \ell(r_1')$ which is not possible so $\mathcal{P}'$ is not deontically stratifiable. Note that if we replace r_1' with "$\mathbf{Do}$ $compute\_currentLocation$ (report) $\leftarrow$ ", then $\mathcal{P}'$ would be deontically stratifiable.

8.1.4 Definition of Weakly Regularity

Definition 8.16 (Strongly Safe Action)
An action $\alpha(\vec{X})$ *is said to be* strongly safe *w.r.t. FINTAB if its precondition is strongly safe modulo* $\vec{X}$*, and each code call from the add list and delete list is strongly safe modulo* $\vec{Y}$ *where* $\vec{Y}$ *includes all root variables in* $\vec{X}$ *as well as in the precondition of* α*.*

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

Definition 8.17 (Weak Regular Agent Program)
Let $\mathcal{P}$ be an agent program, FINTAB a finiteness table, and cft a conflict-freedom test. Then, $\mathcal{P}$ is called a weak regular agent program *(WRAP for short) w.r.t. FINTAB and cft, if, by definition, the following three conditions all hold:*

Strong Safety: *All rules in $\mathcal{P}$ and actions 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 8.9 (Sample WRAP)

Let $\mathcal{P}$ be the agent program given in Example 8.7 on page 392 and suppose that all actions in $\mathcal{P}$ are strongly safe w.r.t. a finiteness table FINTAB. Consider the conflict freedom test cft_h. Then $\mathcal{P}$ is a WRAP as it is conflict free under cft_h and as it is deontically stratified according to Example 8.8 on page 396. Now, suppose we add the following rule to $\mathcal{P}$:

$$r_5\textbf{:}\ \mathbf{W}\ \mathit{create\_flight\_plan}(\mathit{no\_go},\ \textsf{flight\_route},\ \textsf{current\_location}) \leftarrow$$
$$\quad \textsf{\textbf{not\_in}}(\texttt{automated}, \texttt{autoPilot}\text{:}\mathit{pilotStatus}(\texttt{pilot\_message}))$$

This rule indicates that our agent is not obligated to adjust the plane's course if the plane is not on autopilot. Note that as $\mathsf{cft}_h(r_2, r_5) = \mathbf{false}$, our new version of $\mathcal{P}$ is not conflict free and so $\mathcal{P}$ would no longer be a WRAP.

Definition 8.18 (Weakly Regular Agent)
An agent a *is* weakly regular *if, by definition, its associated agent program is weakly regular and the action constraints, integrity constraints, and the notion of concurrency in the background are all strongly safe.*

It remains to define strongly safeness for constraints and the concurrency notion.

Definition 8.19 (Strongly Safe Integrity and Action Constraints)
An integrity constraint of the form $\psi \Rightarrow \chi$ *is* strongly safe *if, by definition,* ψ *is strongly safe and* χ *is strongly safe modulo the root variables in* ψ*. An action constraint* $\{\alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)\} \hookleftarrow \chi$ *is* strongly safe *if and only if* χ *is strongly safe.*

Definition 8.20 (Strongly Safe Notion of Concurrency)
A notion of concurrency, **conc**, *is said to be* strongly safe *if, by definition, for every set* $\mathcal{A}$ *of actions, if all members of* $\mathcal{A}$ *are strongly safe, then so is* $\mathbf{conc}(\mathcal{A})$.

8.2 Properties of Weakly Regular Agents

- Every deontically stratifiable agent program (and hence every *WRAP*) has a so-called "canonical layering".
- Every *WRAP* has an associated fixpoint computation method—the fixpoint computed by this method is the only possible reasonable status set the *WRAP* may have.

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

$$\mathrm{can}^{\mathcal{P}}(r) \quad = \quad \min\{\ell_i(r) \,|\, \ell_i \in \mathrm{wtn}(\mathcal{P})\}.$$

Example 8.10 (Canonical Layering)
Consider the agent program and layer functions given in Example 8.7 on page 392. Recall that $\ell_1 \in \mathit{wtn}(\mathcal{P})$, $\ell_2 \in \mathit{wtn}(\mathcal{P})$, and $\ell_3 \notin wtn(\mathcal{P})$. Here, since ℓ_1 requires three layers ℓ_2 requires two layers, and ℓ_3 requires one layer, $\mathit{can}^{\mathcal{P}} = \ell_2$.

Proposition 8.1
Let $\mathcal{P}$ be an agent program which is deontically stratifiable. Then $\mathit{can}^{\mathcal{P}} \in \mathit{wtn}(\mathcal{P})$, i.e. $\mathit{can}^{\mathcal{P}}$ is a witness to the deontic stratifiability of $\mathcal{P}$.

Definition 8.21 ($\mathbf{T}^{i}_{\mathcal{P},O_{\mathcal{S}}}(S)$ and $\mathbf{T}^{\omega}_{\mathcal{P},O_{\mathcal{S}}}(S)$ Operators)

Suppose $\mathcal{P}$ is an agent program, $O_{\mathcal{S}}$ an agent state, and S is a status set. Then, the operators $\mathbf{T}^{i}_{\mathcal{P},O_{\mathcal{S}}}$, $i \geq 0$, and $\mathbf{T}^{\omega}_{\mathcal{P},O_{\mathcal{S}}}$ are defined as follows:

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

Example 8.11 ($\mathbf{T}^{\cdot}_{\mathcal{P},O_{\mathcal{S}}}$ Operator)

Let $\mathcal{P}$ contain rules r_1, r_2, r_4 from the agent program given in Example 8.7 on page 392, let $O_{\mathcal{S}}$ indicate that the plane is on autopilot, and let $S = \{\mathbf{O} adjustAltitude(5000)\}$. Then

$$
\begin{aligned}
\mathbf{T}^0_{\mathcal{P},O_{\mathcal{S}}}(S) &= \{\ \mathbf{O}\ adjustAltitude(5000)\ \},\\
\mathbf{T}^1_{\mathcal{P},O_{\mathcal{S}}}(S) &= \{\ \mathbf{O}\ adjust\_course(no\_go,\ flight\_route,\ current\_location),\\
&\qquad \mathbf{Do}\ adjustAltitude(5000),\ \mathbf{P}\ adjustAltitude(5000)\ \} \cup \mathbf{T}^0_{\mathcal{P},O_{\mathcal{S}}}(S),\\
\mathbf{T}^2_{\mathcal{P},O_{\mathcal{S}}}(S) &= \{\ \mathbf{O}\ create\_flight\_plan(no\_go,\ flight\_route,\ current\_location),\\
&\qquad \mathbf{Do}\ adjust\_course(no\_go,\ flight\_route,\ current\_location),\\
&\qquad \mathbf{P}\ adjust\_course(no\_go,\ flight\_route,\ current\_location)\ \} \cup \mathbf{T}^1_{\mathcal{P},O_{\mathcal{S}}}(S),\\
\mathbf{T}^3_{\mathcal{P},O_{\mathcal{S}}}(S) &= \{\ \mathbf{Do}\ create\_flight\_plan(no\_go,\ flight\_route,\ current\_location),\\
&\qquad \mathbf{P}\ create\_flight\_plan(no\_go,\ flight\_route,\ current\_location)\ \} \cup \mathbf{T}^2_{\mathcal{P},O_{\mathcal{S}}}(S),\\
\mathbf{T}^4_{\mathcal{P},O_{\mathcal{S}}}(S) &= \{\ \mathbf{Do}\ execute\_flight\_plan(flight\_route)\ \} \cup \mathbf{T}^3_{\mathcal{P},O_{\mathcal{S}}}(S),\\
\mathbf{T}^5_{\mathcal{P},O_{\mathcal{S}}}(S) &= \{\ \mathbf{P}\ execute\_flight\_plan(flight\_route)\ \} \cup \mathbf{T}^4_{\mathcal{P},O_{\mathcal{S}}}(S), \text{ and}\\
\mathbf{T}^6_{\mathcal{P},O_{\mathcal{S}}}(S) &= \mathbf{T}^5_{\mathcal{P},O_{\mathcal{S}}}(S), \text{ so } \mathbf{T}^{\omega}_{\mathcal{P},O_{\mathcal{S}}}(S) = \mathbf{T}^5_{\mathcal{P},O_{\mathcal{S}}}(S).
\end{aligned}
$$

Note that by removing rule r_3 from $\mathcal{P}$, we turned $\mathcal{P}$ into a positive WRAP. To see why this was necessary, suppose $\mathcal{P}$ included r_3. Then both $\mathbf{O}$ $maintain\_course(no\_go$*, flight_route, current_location) and* $\mathbf{O}$ $adjust\_course(no\_go$*, flight_route, current_location) would be members of* $\mathbf{T}^1_{\mathcal{P},O_\mathcal{S}}$ *(S). This is not good since the plane cannot maintain and adjust its course at the same time. Later in this chapter, we shall introduce a fixpoint operator for general WRAPs which effectively solves this problem.*

Recalling from Section 5 that $\mathit{lfp}(\mathbf{T}_{\mathcal{P},O_{\mathcal{S}}}) = \bigcup_{i=0}^{\infty} \mathbf{T}^{i}_{\mathcal{P},O_{\mathcal{S}}}$ is the only candidate for being a rational status set of a positive agent program, using the identity $\mathbf{T}^{i}_{\mathcal{P},O_{\mathcal{S}}}(\emptyset) = \mathbf{T}^{i}_{\mathcal{P},O_{\mathcal{S}}}$ we may restate Theorem 5.1 on page 216 as follows.

Proposition 8.2

Let $\mathcal{P}$ *be a* positive *agent program. Then, a status set* S *is a rational status set of* $\mathcal{P}$ *if and only if* $S = \mathbf{T}^{\omega}_{\mathcal{P},O_{\mathcal{S}}}(\emptyset)$ *and* S *is a feasible status set of* $\mathcal{P}$.

We now extend the fixpoint operator in the preceding subsection to arbitrary *WRAPs*. We will define below an operator $\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow\omega$ that evaluates (from bottom to top) the layers of a *WRAP* generated by a layering function ℓ.

The operator $\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow i$ evaluates the layer i by computing the fixpoint $\mathbf{T}^{\omega}_{\mathcal{P}_i,O_{\mathcal{S}}}$ for the program $\mathcal{P}_i$, starting from the result that has been computed at the previous layer $i-1$. The operator $\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow\omega$ accumulates the computation of all layers. Formally, the definition is as follows.

Definition 8.22 ($\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow i$ and $\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow\omega$ Operators)
Suppose $\mathcal{P}$ is a WRAP witnessed by layering function ℓ, and suppose the layers of $\mathcal{P}$ induced by ℓ are $\mathcal{P}_0,\ldots,\mathcal{P}_k$. The operators $\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow i(S)$ and $\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow\omega(S)$ are defined as follows.

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

We write $\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow i$ and $\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow\omega$ for $\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow i(\emptyset)$ and $\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow\omega(\emptyset)$, respectively.

Example 8.12 ($\Gamma^l_{\mathcal{P},O_\mathcal{S}} \uparrow \omega$ Operator)

Let $O_\mathcal{S}$ indicate that the plane is on autopilot and let $\mathcal{P}$ contain all rules for the agent program given in Example 8.7 on page 392. Additionally, let $\mathcal{P}$ contain the following rule:

r_0: **O** *adjustAltitude(5000)* $\leftarrow$

Then $\mathcal{P}$ is a WRAP which is witnessed by layering function ℓ where $\mathcal{P}^\ell_0 = \{r_0, r_4\}$ and $\mathcal{P}^\ell_1 = \{r_1, r_2, r_3\}$. Here,

$$\begin{aligned}\Gamma^l_{\mathcal{P},O_\mathcal{S}} \uparrow 0 = \{ & \mathbf{O}\ \mathit{adjustAltitude}(5000), \\ & \mathbf{O}\ \mathit{adjust\_course}(\mathit{no\_go}, \mathit{flight\_route}, \mathit{current\_location}), \\ & \mathbf{Do}\ \mathit{adjustAltitude}(5000), \mathbf{P}\ \mathit{adjustAltitude}(5000), \\ & \mathbf{Do}\ \mathit{adjust\_course}(\mathit{no\_go}, \mathit{flight\_route}, \mathit{current\_location}), \\ & \mathbf{P}\ \mathit{adjust\_course}(\mathit{no\_go}, \mathit{flight\_route}, \mathit{current\_location}) \},\end{aligned}$$

$$\begin{aligned}\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow 1 = \{\ & \mathbf{O}\ \mathit{create\_flight\_plan}(\mathit{no\_go}, \textit{flight\_route}, \textit{current\_location}),\\ & \mathbf{Do}\ \mathit{create\_flight\_plan}(\mathit{no\_go}, \textit{flight\_route}, \textit{current\_location}),\\ & \mathbf{P}\ \mathit{create\_flight\_plan}(\mathit{no\_go}, \textit{flight\_route}, \textit{current\_location}),\\ & \mathbf{Do}\ \mathit{execute\_flight\_plan}(\textit{flight\_route}),\\ & \mathbf{P}\ \mathit{execute\_flight\_plan}(\textit{flight\_route})\ \} \cup \Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow 0, \text{ and}\end{aligned}$$

$\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow 2 = \Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow 1$, *so* $\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow \omega = \Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}}\uparrow 1$.

Note that although r_3 was included in $\mathcal{P}$, it never had a chance to fire as it had to be assigned to the second layer. Thus, we have some insight into why this fixpoint operator solves the problem mentioned in Example 8.11 on page 406.

Theorem 8.3
Suppose $\mathcal{P}$ is a WRAP. Let $\ell \in wtn(\mathcal{P})$ be any witness to the regularity of $\mathcal{P}$. If $\mathcal{IC}$ and $\mathcal{AC}$ are both empty, then $\Gamma^{l}_{\mathcal{P},O_{\mathcal{S}}} \uparrow \omega$ is a reasonable status set of $\mathcal{P}$ w.r.t. $O_{\mathcal{S}}$.

8.3 Regular Agent Programs

- A regular agent program then is a program which is weakly regular and **bounded** (to be defined below).
- Boundedness means that by repeatedly unfolding the positive parts of the rules in the program, we will eventually get rid of all positive action status atoms.
- Thus, in this section, we will associate with any agent program $\mathcal{P}$ an operator $\mathrm{Unfold}_{\mathcal{P}}$ which is used for this purpose.

We want all positive action status atoms to disappear in rule bodies.

Definition 8.23 (Prerequisite-Free (pf) Constraint)
A prerequisite-free (pf) constraint is defined as follows:

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

Definition 8.24 (Prerequisite-Free Constraint Rule)
A prerequisite-free constraint rule is of the form

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

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

An agent program $\mathcal{P}$ may certainly contain rules r which have prerequisites $\mathbf{O}p\alpha$.

Each such prerequisite might be replaced by the body of a rule r' which derives $\mathbf{O}p\alpha$.

This way, the prerequisites can be eliminated from r, replacing them by rule bodies. This step may introduce new prerequisites from the body of some rule r', though; such prerequisites may be eliminated by repeating the process.

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

The operator $\mathrm{Coll}_{\mathcal{P}}$ introduced next is an intermediate operator for defining $\mathrm{Unfold}_{\mathcal{P}}$.

Definition 8.25 (Operator $\mathbf{Coll}_{\mathcal{P}}$)
Let $\mathcal{P}$ be an agent program and R be a set of pf-constraint rules which are standardized apart. Suppose $Op \in \{\mathbf{P}, \mathbf{O}, \mathbf{Do}, \mathbf{W}, \mathbf{F}\}$ and let α be any action name. Then the collect set, *$Coll_{\mathcal{P}}(R, Op, \alpha)$, is defined as the following set of pf-constraints:*

$$\begin{aligned} Coll_{\mathcal{P}}(R, Op, \alpha) = \Big\{\gamma \mid\ & Op\alpha(\vec{X}) \leftarrow \gamma_0 \in R, \\ & \text{there exists a rule } r \in \mathcal{P} \text{ such that } head(r) = Op'\alpha(\vec{t}), Op' \leq Op, \\ & B^{+}_{as}(r) = \{Op_1\alpha_1(\vec{t}_1), \ldots, Op_k\alpha_k(\vec{t}_k)\}, \\ & Op_i\alpha_i(\vec{X}_i) \leftarrow \gamma_i \in R, i = 1, \ldots, k, \\ & \text{and } \gamma = \gamma_0 \vee \Big[(\vec{X} = \vec{t})\&B_{cc}(r)\&\textstyle\bigwedge_{i=1}^{k}\Big((\vec{X}_i = \vec{t}_i)\&\gamma_i\Big)\&B^{-}_{as}(r)\Big]\Big\} \end{aligned}$$

Example 8.13 ($\mathrm{Coll}_{\mathcal{P}}$ Operator)
Let $\mathcal{P}$ be the agent program in Example 8.7 on page 392 and let

$$R = \{\, \mathbf{O} adjustAltitude(5000) \leftarrow \mathbf{in}(\mathtt{Alt}, \mathrm{autoPilot}: getAltitude()) \,\&\, (\mathtt{Alt} < 4000) \,\}.$$

Then

$$\begin{aligned} Coll_{\mathcal{P}}(R, \mathbf{O}, adjust\_course(\mathtt{X}, \mathtt{Y}, \mathtt{Z})) \quad = \{ \quad & (\mathtt{X} = no\_go) \,\&\, (\mathtt{Y} = flight\_route) \,\&\, \\ & (\mathtt{Z} = current\_location) \,\&\, (\mathtt{Altitude} = 5000) \,\&\, \\ & \mathbf{in}(\mathtt{Alt}, \mathrm{autoPilot}: getAltitude()) \,\&\, (\mathtt{Alt} < 4000) \}. \end{aligned}$$

Note that this expression can be simplified by removing the unused $(\mathtt{Altitude} = 5000)$ conjunct.

Definition 8.26 (Operator $\text{Unfold}_{\mathcal{P}}$)

$$\begin{aligned} Unfold_{\mathcal{P}}(Op,\alpha,R) &= Op\alpha(\vec{X}) \leftarrow \bigvee_{\gamma \in Coll_{\mathcal{P}}(R,Op,\alpha)} \gamma, \qquad \textit{and} \\ Unfold_{\mathcal{P}}(R) &= \bigcup_{Op,\alpha} Unfold_{\mathcal{P}}(Op,\alpha,R). \end{aligned}$$

When $Coll_{\mathcal{P}}(R,Op,\alpha)$ is empty in the above definition, the right hand side of the above implication is set to `false`*.*

The operator $\text{Unfold}_{\mathcal{P},Op,\alpha}$ may be iterated; its powers are (as usual) denoted by $\text{Unfold}^0_{\mathcal{P}}(\text{Op},\alpha,R) = R$, $\text{Unfold}^{i+1}_{\mathcal{P}}(\text{Op},\alpha,R) = \text{Unfold}_{\mathcal{P}}(\text{Op},\alpha,\text{Unfold}^i_{\mathcal{P}}(\text{Op},\alpha,R))$, $i \geq 0$, and similar with $\text{Unfold}_{\mathcal{P}}$.

Example 8.14 (Unfold$_{\mathcal{P}}$ Operator)

Let $\mathcal{P}$ be the agent program in Example 8.12 on page 411 and let $R = \emptyset$. Then

$$Unfold^0_{\mathcal{P}}(R) = \emptyset,$$

$$\begin{aligned} Unfold^1_{\mathcal{P}}(R) = \{\ & \mathbf{O}/\mathbf{Do}\,/\mathbf{P}\ \mathit{maintain\_course}(\mathit{no\_go},\ \mathit{flight\_route},\ \mathit{current\_location}) \leftarrow \\ & \quad \mathbf{in}(\mathtt{automated}, \mathtt{autoPilot}{:}\mathit{pilotStatus}(\mathtt{pilot\_message}))\ \& \\ & \quad \neg\ \mathbf{O}\ \mathit{adjust\_course}(\mathit{no\_go},\ \mathit{flight\_route},\ \mathit{current\_location}), \\ & \mathbf{O}/\mathbf{Do}\,/\mathbf{P}\mathit{adjustAltitude}(5000) \leftarrow \ \}, \end{aligned}$$

$$\begin{aligned} Unfold^2_{\mathcal{P}}(R) = \{\ & \mathbf{O}/\mathbf{Do}\,/\mathbf{P}\ \mathit{adjust\_course}(\mathit{no\_go},\ \mathit{flight\_route},\ \mathit{current\_location}) \leftarrow \\ & \quad (\mathtt{Altitude} = 5000)\ \} \cup Unfold^1_{\mathcal{P}}(R), \end{aligned}$$

$$\begin{aligned} Unfold^3_{\mathcal{P}}(R) = \{\ & \mathbf{O}/\mathbf{Do}\,/\mathbf{P}\ \mathit{create\_flight\_plan}(\mathtt{No\_go},\ \mathtt{Flight\_route},\ \mathtt{Current\_location}) \leftarrow \\ & \quad (\mathtt{No\_go} = \mathit{no\_go})\ \&\ (\mathtt{Flight\_route} = \mathit{flight\_route})\ \& \\ & \quad (\mathtt{Current\_location} = \mathit{current\_location})\ \& \\ & \quad (\mathtt{Altitude} = 5000)\ \} \cup Unfold^2_{\mathcal{P}}(R), \end{aligned}$$

$$
\begin{aligned}
\mathit{Unfold}^4_{\mathcal{P}}(R) = \{\ & \mathbf{Do}\,/\mathbf{P}\ \mathit{execute\_flight\_plan}(\mathtt{Flight\_route}) \leftarrow \\
& \mathbf{in}(\mathtt{automated}, \mathtt{autoPilot}{:}\mathit{pilotStatus}(\mathtt{pilot\_message}))\ \& \\
& (\mathtt{No\_go} = \mathit{no\_go})\,\&\,(\mathtt{Flight\_route} = \mathit{flight\_route})\,\& \\
& (\mathtt{Current\_location} = \mathit{current\_location})\,\& \\
& (\mathtt{Altitude} = 5000)\ \} \cup \mathit{Unfold}^3_{\mathcal{P}}(R),
\end{aligned}
$$

and $\mathit{Unfold}^5_{\mathcal{P}}(R) = \mathit{Unfold}^4_{\mathcal{P}}(R)$.

Note that in this example, $\mathbf{O}/\mathbf{Do}\,/\mathbf{P}\,\alpha(\vec{X}) \in \mathit{Unfold}^i_{\mathcal{P}}(R)$ *(or* $\mathbf{Do}\,/\mathbf{P}\,\alpha(\vec{X}) \in \mathit{Unfold}^i_{\mathcal{P}}(R)$*) indicates that* $\{\mathbf{O}\,\alpha(\vec{X}), \mathbf{Do}\,\alpha(\vec{X}), \mathbf{P}\,\alpha(\vec{X})\} \subseteq \mathit{Unfold}^i_{\mathcal{P}}(R)$ *(*$\{\mathbf{Do}\,\alpha(\vec{X}), \mathbf{P}\,\alpha(\vec{X})\} \subseteq \mathit{Unfold}^i_{\mathcal{P}}(R)$*).*

Definition 8.27 (pf-constraint Equivalence Test)
A pf-constraint equivalence check test $eqi_{a,\vec{X}',\vec{Y}'}$ *is a function that takes as input two pf-constraints* pfc_1, pfc_2*, such that if* $eqi_{a,\vec{X}',\vec{Y}'}(pfc_1, pfc_2) = \mathbf{true}$ *then* pfc_1, pfc_2 *are equivalent w.r.t.* $\mathcal{P}$*.*

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

Definition 8.28 (Equivalence of Two Sets of pf-constraint Rules) *Two sets R_1, R_2 of pf-constraint rules are equivalent w.r.t. a pf-constraint equivalence test eqi, denoted $R_1 \equiv_{\mathit{eqi}} R_2$, if, by definition, there is a bijection $\psi : R_1 \to R_2$ such that for all $r_1 \in R_1$, eqi$(r_1, \psi(r_1)) = \mathbf{true}$ and r_1, r_2 both have heads of the form Op$\alpha(\cdots)$, i.e. their heads involve the same action name and the same deontic modality.*

Definition 8.29 (b-bounded Agent Program)
An agent program $\mathcal{P}$ *is* bounded *w.r.t. an equivalence check test eqi if, by definition, there is an integer b such that* $\mathit{eqi}(\mathit{Unfold}^{b}_{\mathcal{P}}(R), \mathit{Unfold}^{b+1}_{\mathcal{P}}(R)) = \mathbf{true}$*, for any set of pf-constraints R. In this case, $\mathcal{P}$ is (eqi, b)-bounded.*

Observe that when $\mathcal{P}$ is a program not containing a truly recursive collection of rules, then $\mathcal{P}$ is (eqi, b)-bounded where eqi is an arbitrary pf-constraint equivalence test such that $\mathsf{eqi}(\mathsf{pfc}, \mathsf{pfc}) = \mathbf{true}$ for every pfc and b is the number of rules in $\mathcal{P}$.

Thus, only truly recursive rules—which seem to play a minor rule in many agent programs in practical applications—may prevent boundedness. If, moreover, $\mathcal{P}$ is deontically stratified and has a layering $\mathcal{P}_0, \ldots, \mathcal{P}_k$ then $\mathcal{P}$ is even $(\mathsf{eqi}, k+1)$-bounded.

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

$$\begin{aligned} R_0^\ell &= \emptyset, \\ R_{i+1}^\ell &= \mathsf{Unfold}_{\mathcal{P}_i}^b(R_i^\ell), \text{ for all } i \geq 0, \end{aligned}$$

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

Definition 8.30 (b-regular Agent Program)
*Suppose a layering ℓ and equivalence tests eqi$^{(i)}$ $(i \geq 0)$ have been fixed for an agent program $\mathcal{P}$. Then, $\mathcal{P}$ is said to be a b-*regular agent program *w.r.t. ℓ and the eqi$^{(i)}$, if, by definition, $\mathcal{P}$ is a WRAP, $\ell \in$ wtn$(\mathcal{P})$, and each layer $\mathcal{P}_i$ of $\mathcal{P}$ is $(\text{eqi}^{(i)}, b)$-bounded.*

Definition 8.31 (Regular Agent)
An agent is said to be regular *w.r.t. a layering ℓ and a selection of pf-constraint equivalence tests eqi$^{(i)}$, if it is weakly regular and its associated agent program is b-regular w.r.t. ℓ and the eqi$^{(i)}$, for some $b \geq 0$.*

8.4 Compile-Time Algorithms

Algorithm 8.1
Check_WRAP($\mathcal{P}$)

(⋆ input is an agent program $\mathcal{P}$, a conflict-freedom test cft, and a finiteness table FINTAB ⋆)
(⋆ output is a layering $\ell \in wtn(\mathcal{P})$, if $\mathcal{P}$ is regular and "no" otherwise ⋆)

1. **If** *some action α or rule r in $\mathcal{P}$ is not strongly safe* **then return** *"no" and* **halt**.

2. **If** *some rules $r : Op(\alpha(\vec{X}))$ and $r' : Op'(\alpha(\vec{Y}))$ in $\mathcal{P}$ exist such that $cft(r, r') = false$,* **then return** *"no" and* **halt**.

3. **If** *a rule $r : Op_i(\alpha(\vec{X})) \leftarrow \ldots, (\neg)Op_j(\alpha(\vec{Y})), \ldots$ is in $\mathcal{P}$ such that $Op_i(\alpha(\vec{X}))$ and $Op_j((\vec{Y}))$ conflict,* **then return** *"no" and* **halt**.

4. *Build the graph* $G = (V, E)$*, where* $V = \mathcal{P}$ *and an edge* $r_i \to r$ *is in* E *for each pair of rules* r_i *and* r *as in the two Stratifiability conditions.*

5. *Compute, using Tarjan's algorithm, the supergraph* $S(G) = (V^*, E^*)$ *of* G*.*

6. **If** *some rules* r_i, r *as in the second stratifiability condition exists such that* $r_i, r \in C$ *for some* $C \in V^*$*,* **then return** *"no" and* **halt else** *set* $i := 0$*.*

7. *For each* $C \in V^*$ *having out-degree* 0 *(i.e. no outgoing edge) in* $S(G)$*, and each rule* $r \in C$*, define* $\ell(r) := i$*.*

8. *Remove each of the above* C*'s from* $S(G)$*, and remove all incoming edges associated with such nodes in* $S(G)$ *and set* $i := i + 1$*;*

9. **If** $S(G)$ *is empty, i.e.,* $V^* = \emptyset$*,* **then return** ℓ *and* **halt else** *continue at 7.*

Theorem 8.4

For any agent program $\mathcal{P}$, **Check_WRAP**$(\mathcal{P})$ *returns w.r.t. a conflict-freedom test cft and a finiteness table FINTAB, a layering $\ell \in \mathit{wtn}(\mathcal{P})$ if $\mathcal{P}$ is a WRAP, and returns "no" if $\mathcal{P}$ is not regular.*

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

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

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

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

Algorithm 8.2
Reasonable-SS($\mathcal{P}, \ell, \mathcal{IC}, \mathcal{AC}, \mathcal{O}_\mathcal{S}$)

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

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

5. **while** $\mathcal{IC} \neq \emptyset$ **do**

select and remove some $ic \in \mathcal{IC}$*;*

if $\mathcal{O}'_{\mathcal{S}} \not\models ic$ **then return** *"no" and* **halt**.

6. **return** $\mathcal{S}$ *and* **halt**.

Even though Algorithm **Reasonable_SS** can be executed on weakly regular agent programs, rather than *RAPs*, there is no guarantee of termination in that case. The following theorem states the result that for a regular agent, its reasonable status set on an agent state is effectively computable.

Theorem 8.5
If a *is a regular agent, then algorithm* **Reasonable_SS** *computes a reasonable status set (if one exists) in finite time.*

Theorem 8.6

Suppose $\mathfrak{a}$ is a fixed regular agent. Assume that the following holds:

(1) *Every ground code call $\mathcal{S}:f(\mathtt{d}_1,\ldots,\mathtt{d}_\mathtt{n})$, has a polynomial set of solutions, which is computed in polynomial time; and*

(2) *no occurrence of a variable in $\mathfrak{a}$'s description loose.*

Furthermore, assume that assembling and executing $\mathbf{conc}(\mathbf{Do}(S), O_{\mathcal{S}})$ *is possible in polynomial time in the size of* $\mathbf{Do}(S)$ *and* $O_{\mathcal{S}}$*. Then, algorithm* **Reasonable_SS** *computes a reasonable status set (if one exists) on a given agent state* $O_{\mathcal{S}}$ *in polynomial time (in the size of* $O_{\mathcal{S}}$*).*

8.5 IADE

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

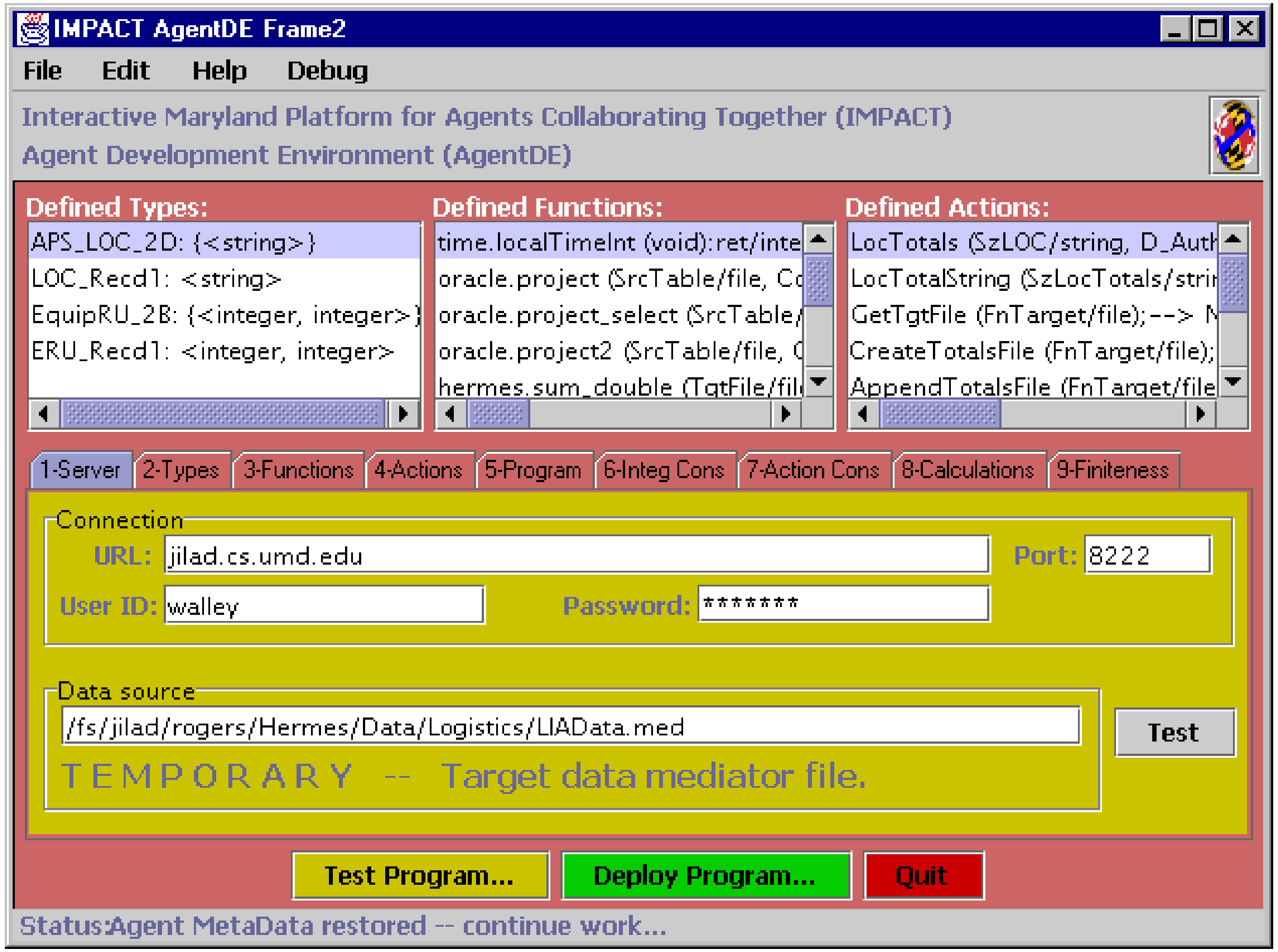


Figure 8.2: Main *IADE* Screen

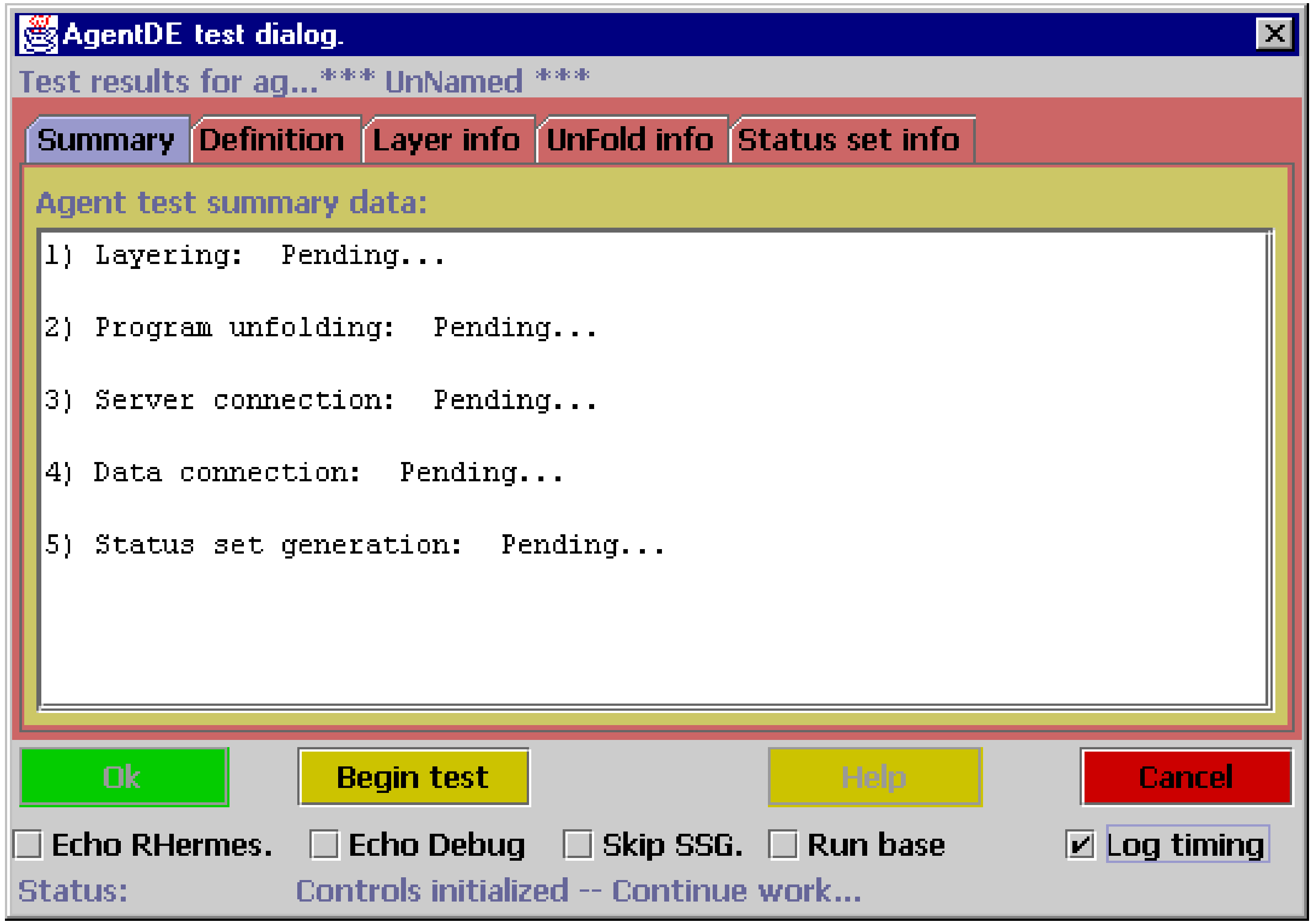


Figure 8.3: *IADE* Test Dialog Screen Prior to Program Testing

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

The *IADE* includes the safety, strong safety, conflict freedom algorithms, and the **Check_WRAP** algorithms (the last is slightly modified). The unfold algorithm currently works on positive agent programs—this is being extended to the full fledged case.

Figure 8.2 on page 439 shows a screendump of *IADE*'s top-level screen.

Figure 8.3 on page 440 specifies what happens when the agent developer presses the “Test Program” button in the Figure 8.2 on page 439 screen.

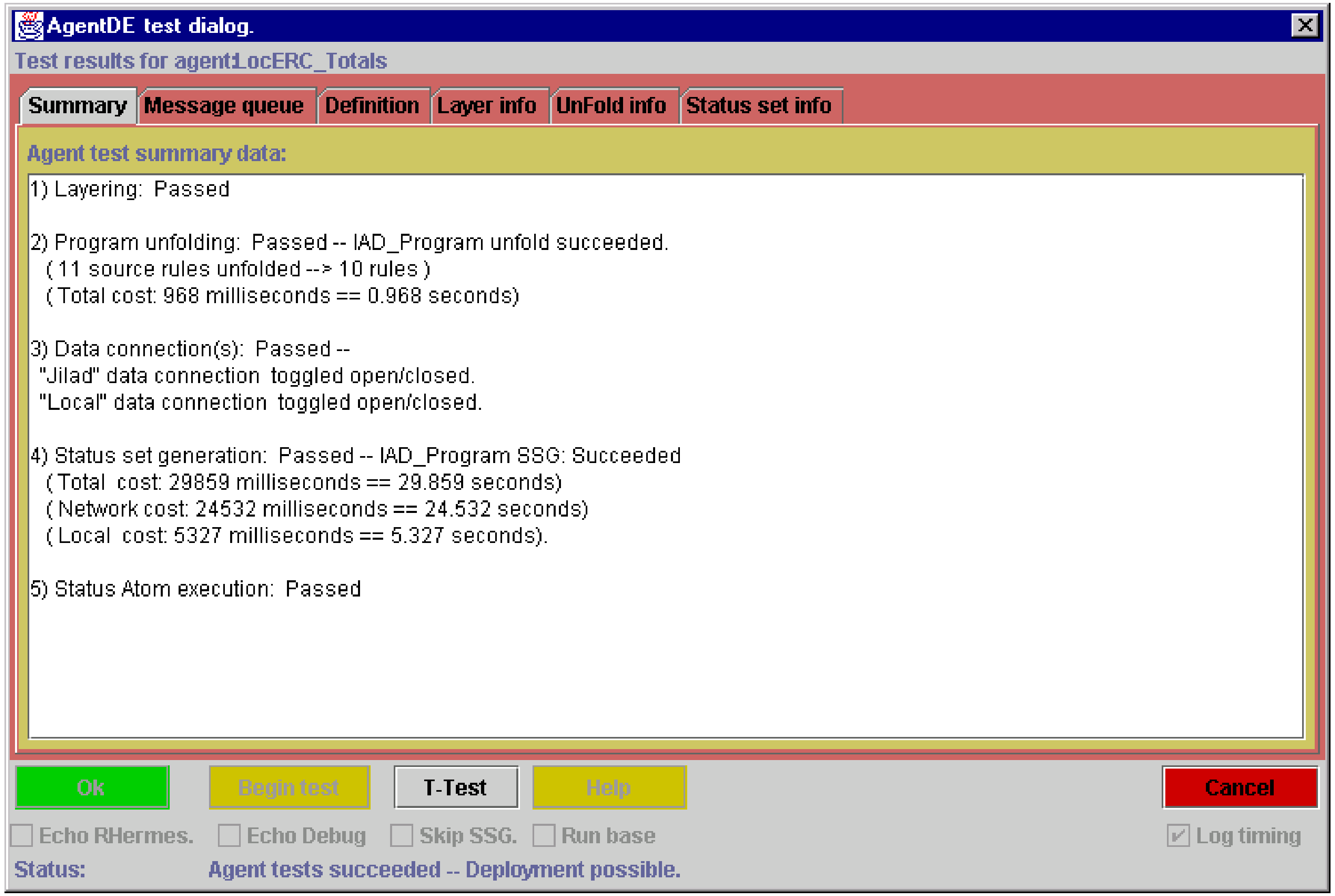


Figure 8.4: *IADE* Test Execution Screen

Once the status sets have been generated after the test execution phase is completed, the user can press the “Unfold Info” tab (to see the unfolded program) or the “Layer Info” tab (to see the layers of the agent program) or the “Status Set Info” tab (to see status information). Figure 8.5 on the following page shows the results of viewing the unfold information.

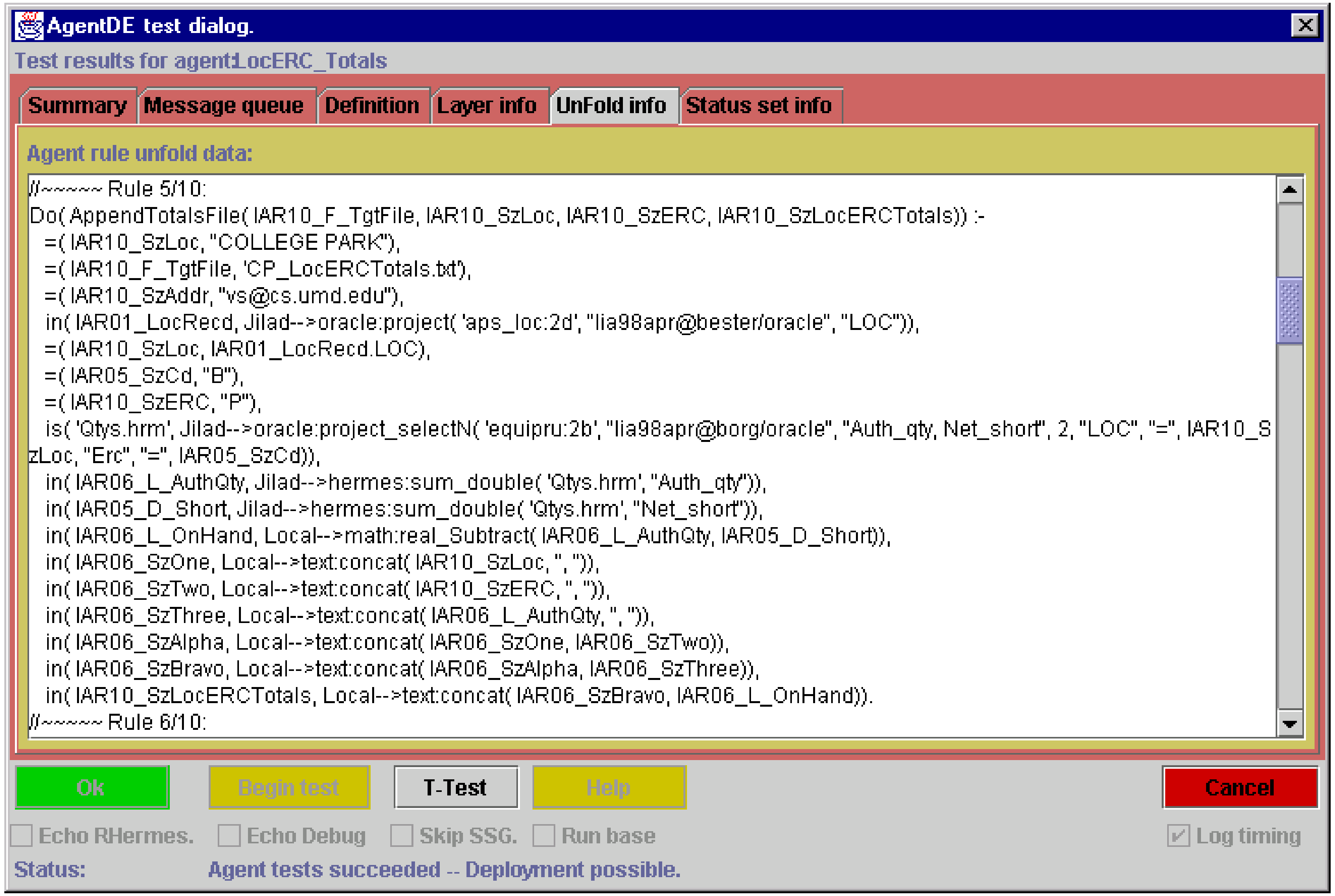


Figure 8.5: *IADE* Unfold Information Screen

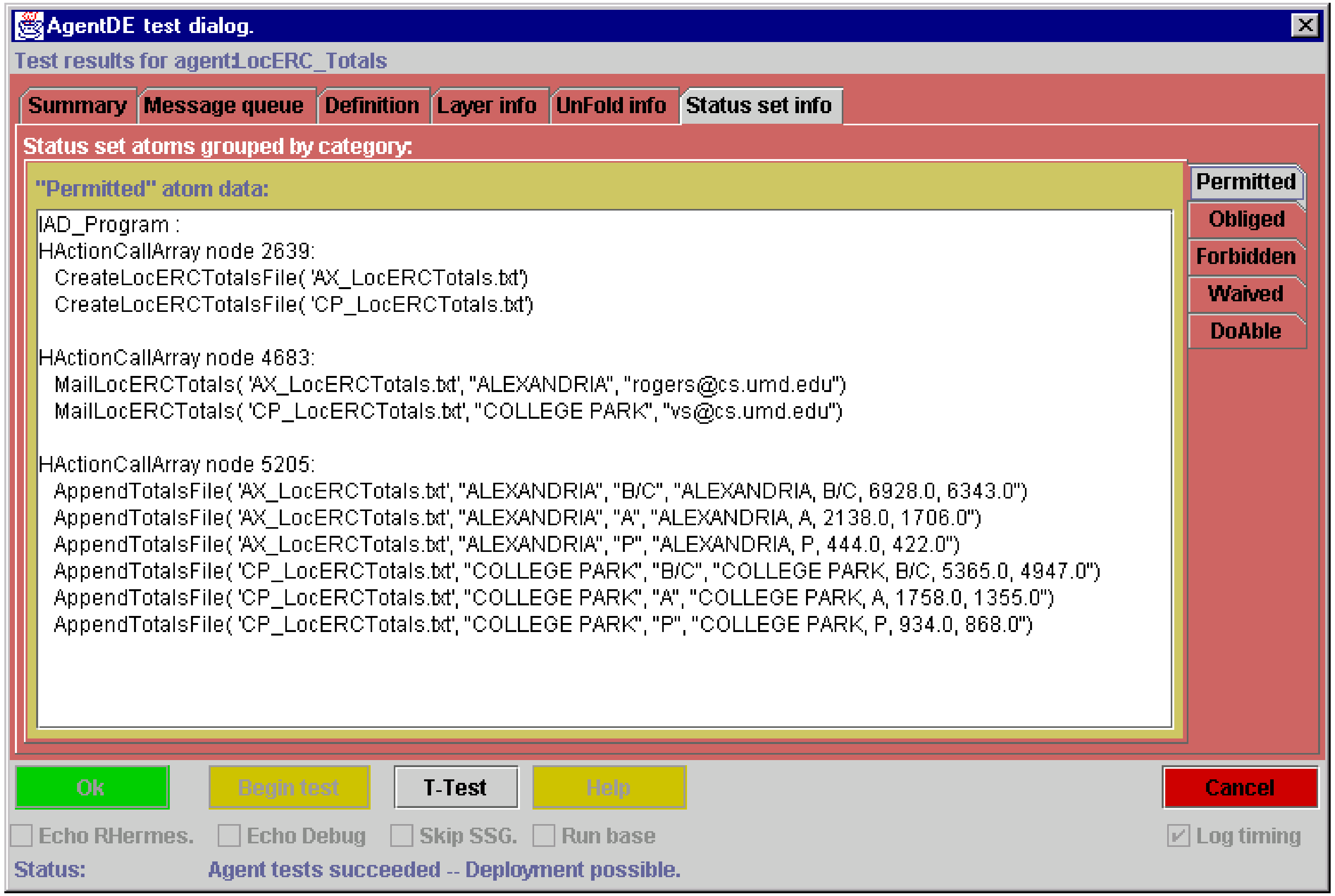


Figure 8.6: *IADE* Status Set Screen

When the user selects the “Status set Info” tab, he sees the screen shown in Figure 8.6 on the page before. Note that this screen has tabs on the right, corresponding to the various deontic modalities. By selecting a modality, the agent developer can see what action status atoms associated with that modality are true in the status set. Figure 8.6 on the preceding page shows what happens when the user wishes to see all action status atoms of the form $\mathbf{Do}(\ldots)$ in the status set.

Figure 8.7 on page 450 shows the interface used to specify the "finiteness" table. As mentioned earlier on in this chapter, in the *IMPACT* implementation, we actually represent code calls that are infinite in this table, using some extra syntax. Specifically, the first row of the table shown in Figure 8.7 on page 450 says that when $Q > 3$ and $R > 4$, all code calls of the form $\mathrm{domain}_1 : \mathit{function}_1(\mathtt{Q},\mathtt{R})$ are infinite.

Figure 8.8 on page 451 shows the interface used by the agent developer to specify what notion of concurrency he wishes to use, what conflict freedom implementation he wishes to use and what semantics he wishes to use. Each of the items in the figure have associated drop-down menus (not visible in the picture). The last item titled "Calculation Method" enables us (as developers of *IMPACT*) to test different computation algorithms. It will be removed from the final *IMPACT* release.

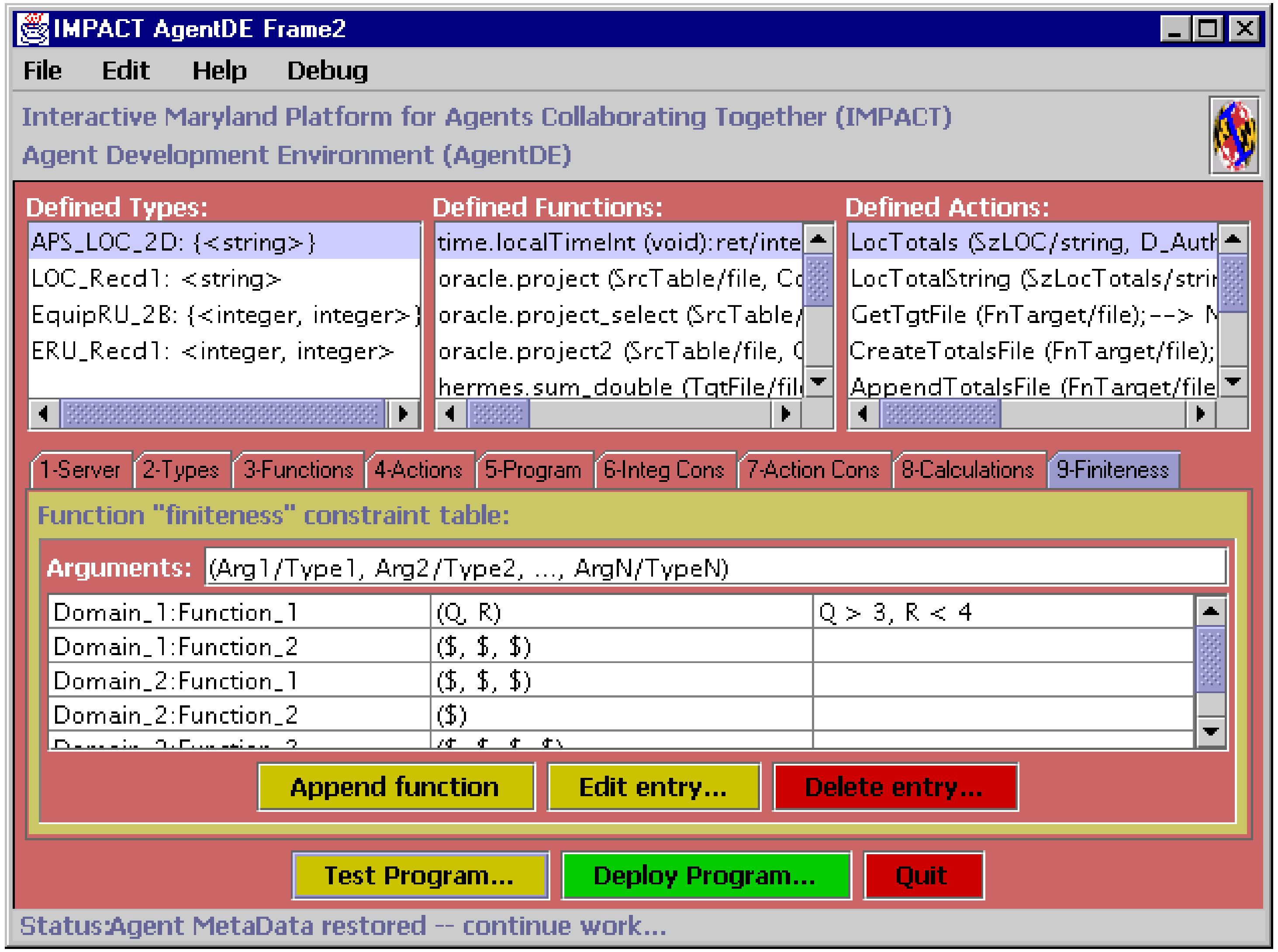


Figure 8.7: *IADE* (In-)Finiteness Table Screen

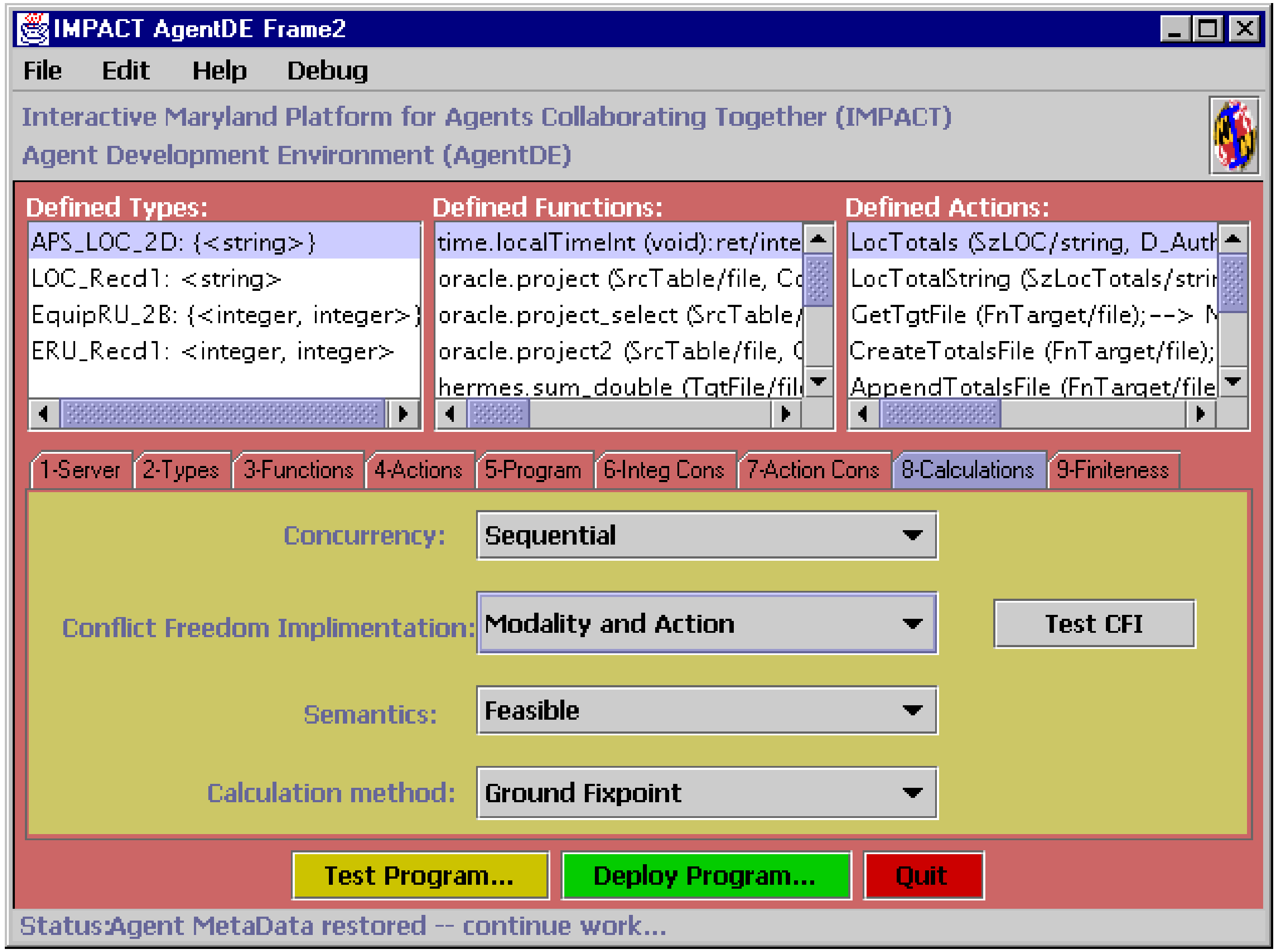


Figure 8.8: *IADE* Option Selection Screen

8.6 Experimental Results

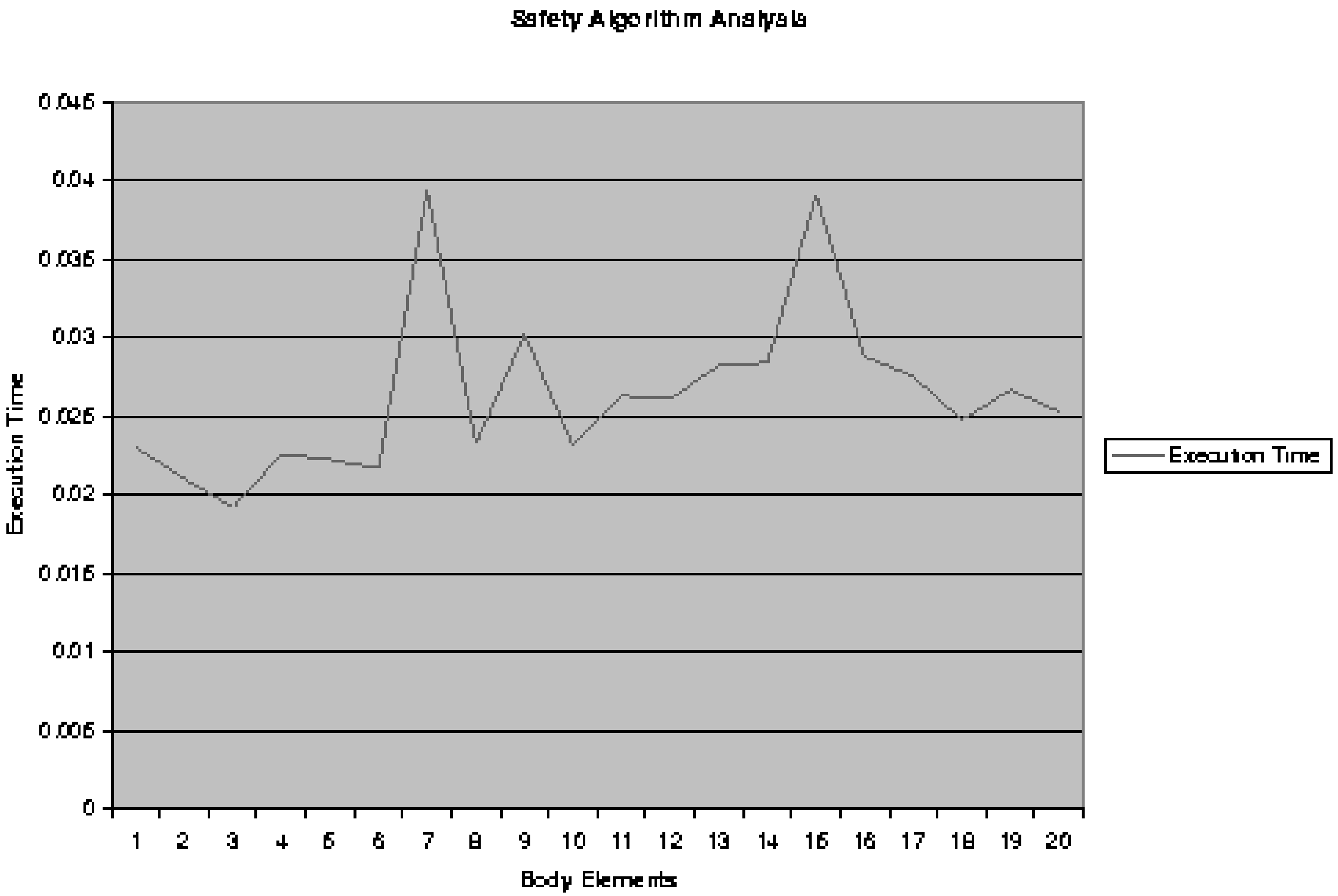


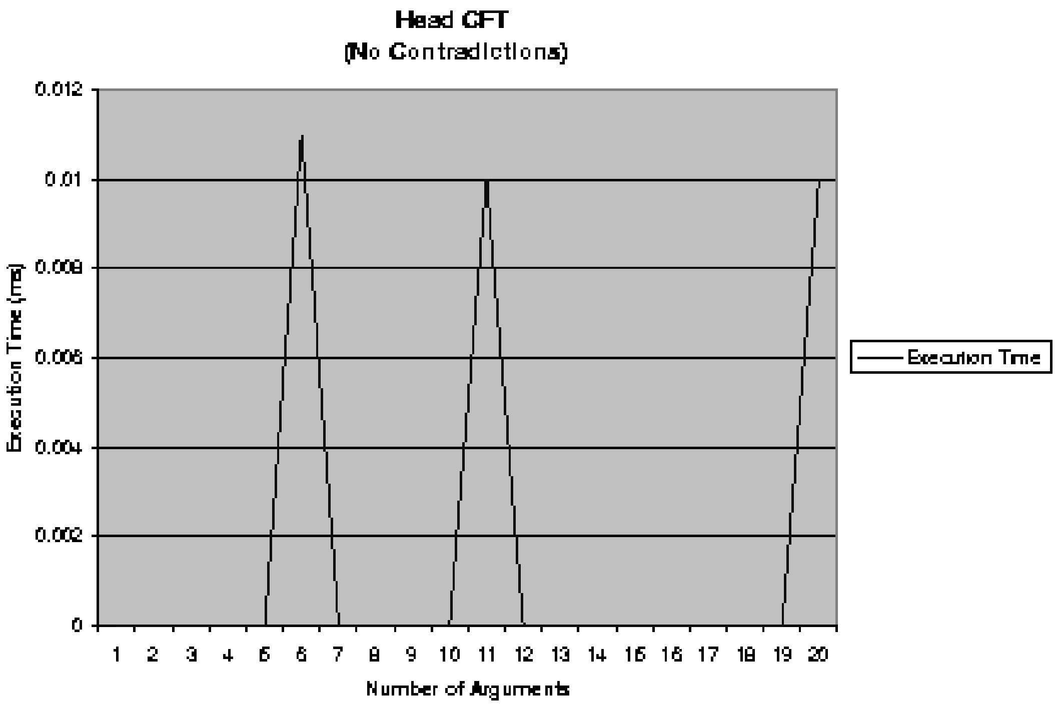
Figure 8.9: Safety Experiment Graphs

Figure 8.9 on the preceding page shows the performance of our implemented safety check algorithm. In this experiment, we varied the number of conjuncts in a code call condition from 1 to 20 in steps of 1. This is shown on the x-axis of Figure 8.9 on the page before.

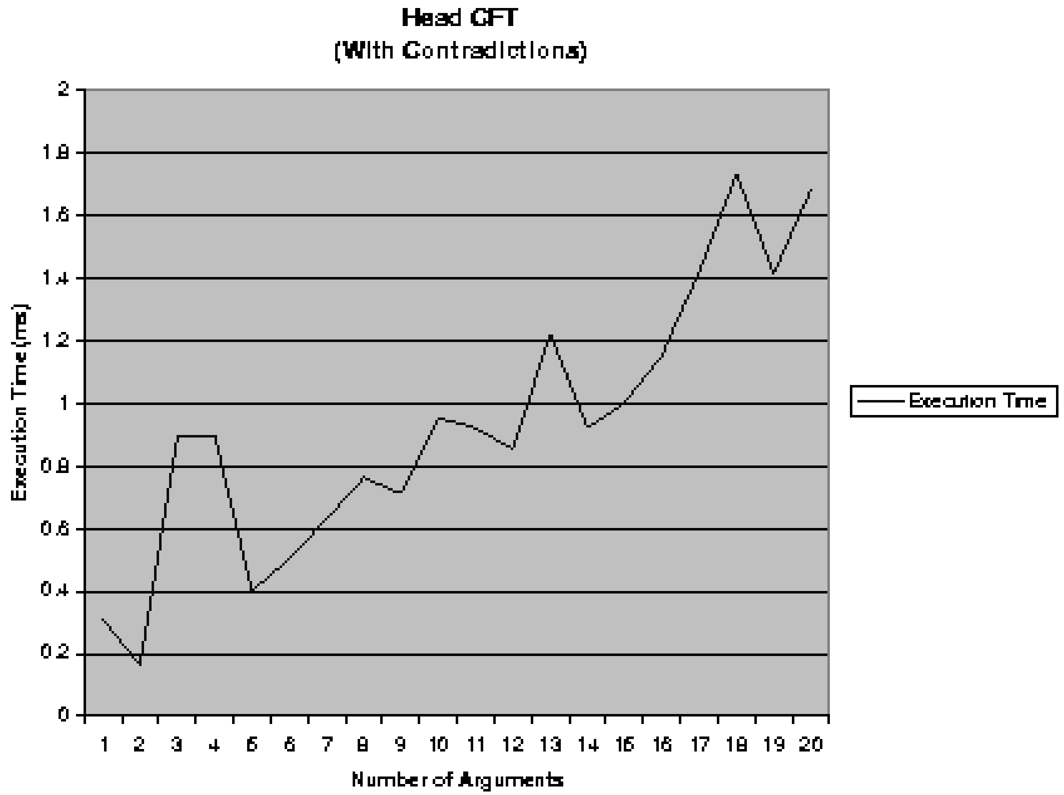
For each $1 \leq x \leq 20$, we executed the **safe_ccc** algorithm 1000 times, varying the number of arguments of each code call from 1 to 10 in steps of 1, and the number of root variables occurring in the code call conditions from 1 to twice the number of conjuncts (i.e., 1 to $2x$).

The actual conjuncts were generated randomly once the number of conjuncts, number of arguments, and number of root variables was fixed. For each fixed number $1 \le i \le 20$ of conjuncts, the execution time shown on the y-axis represents the average over 1000 runs with varying values for number of arguments and number of variables. Times are given in milliseconds.

The reader can easily see that algorithm **safe_ccc** is extremely fast, taking between 0.02 milliseconds and 0.04 milliseconds. Thus, checking safety for an agent program with a 1000 rules can probably be done in 20-40 milliseconds.

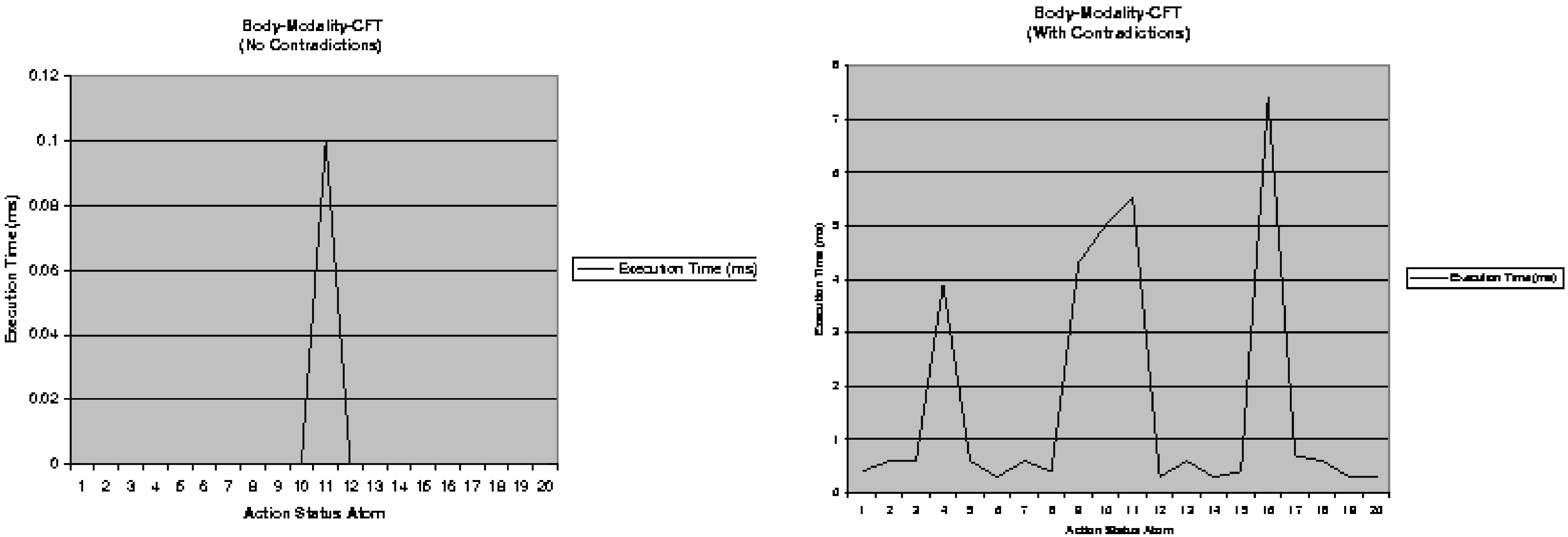


(a) HeadCFT returning "**true**"



(b) HeadCFT returning "**false**"

Figure 8.10: Performance of Conflict Freedom Tests



(c) BodyModalityCFT returning "**true**" (d) BodyModalityCFT returning "**false**"

Figure 8.11: Performance of Conflict Freedom Tests

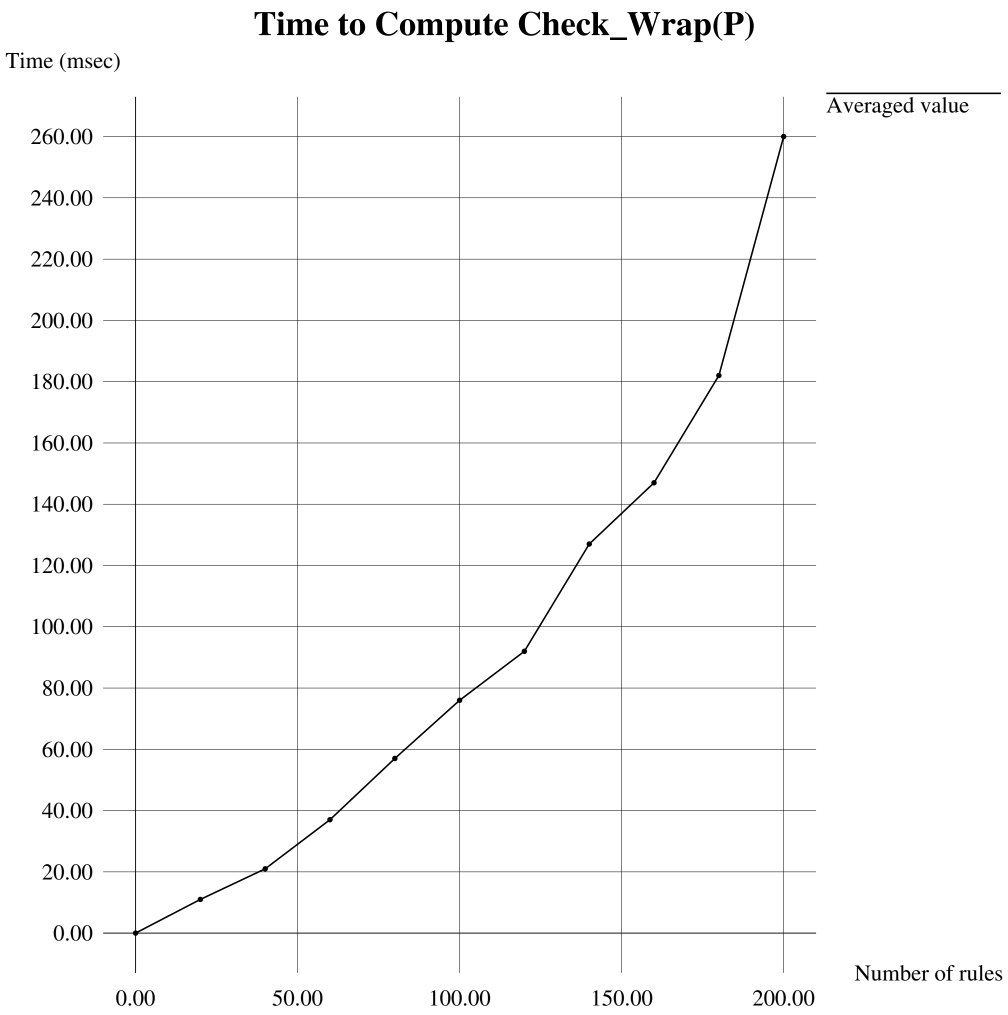
In *IADE*, we have implemented the Head-CFT and Body-Modality-CFT—several other CFTs are being implemented to form a library of CFTs that may be used by agent developers. Figures 8.10 on page 455, 8.11 on the page before shows the time taken to execute the Head-CFT and Body-Modality-CFTs.

Note that Head-CFT is clearly much faster than Body-Modality-CFT when returning "false"—however, this is so because Head-CFT returns "false" on many cases when Body-Modality-CFT does not do so. However, on returns of "**true**," both mechanisms are very fast, usually taking time on the order of $\frac{1}{100}$ to $\frac{1}{10}$ of a millisecond, with some exceptions.

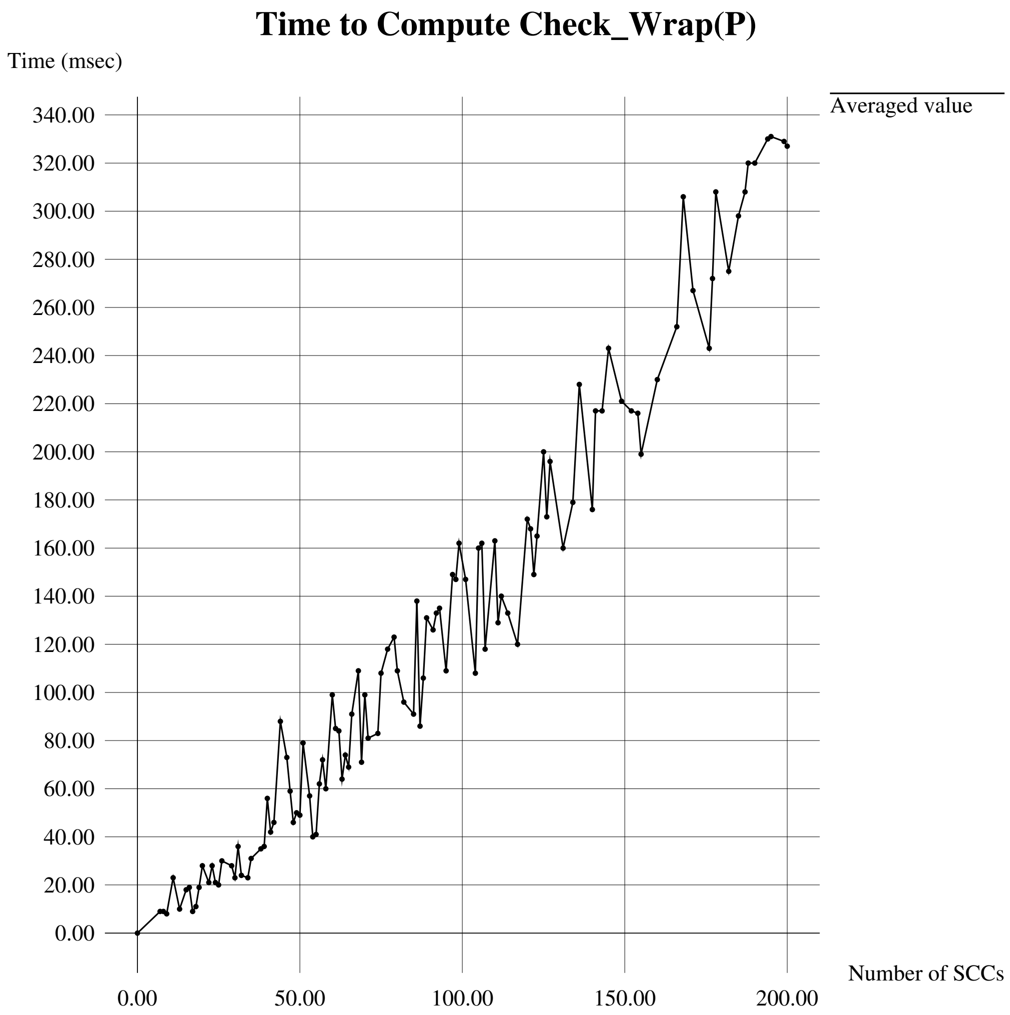
These very small times also explain the “zigzag” nature of the graphs—even small discrepancies (on the order of $\frac{1}{100}$ of a second) appear as large fluctuations in the graph. Even if an agent program contains a 1000 rules (which we expect to be an exceptional case), one would expect the Body-Modality-CFT to only take a matter of seconds to conduct the one-time, compile-time test—a factor that is well worth paying for in our opinion.

Our experiments generated graphs randomly (as described below) and the programs associated with those graphs can be reconstructed from the graphs.

In our experiments, we randomly varied the number of rules from 0 to 200 in steps of 20, and ensured the there were between V and $2V$ edges in the resulting graph, where V is the number of rules (vertices). The precise number was randomly generated. For each such selection, we performed twenty runs of the algorithm. The time taken to generate the graphs was included in these experimental timings. Figures 460 on the following page (a) and (b) show the results of our experiments.



(a) Varying Rules



(b) Varying SCC's

Figure 8.12: Performance of Deontic Stratification

Figure 460 on the page before(a) shows the time taken to execute all but the safety and conflict freedom tests of the **Check_WRAP** algorithm. The reader will note that the algorithm is very fast, taking only about 260 milliseconds on an agent program with 200 rules.

Figure 460 on the preceding page(b) shows the relationship between the number of SCCs in a graph, and the time taken to compute whether the agent program in question is deontically stratified. In this case, we note that as the number of SCCs increases to 200, the time taken goes to about 320 milliseconds. Again, the deontic stratifiability requirement seems to be very efficiently computable.

8.6.1 Performance of Unfolding Algorithm

We were unable to conduct detailed experiments on the time taken for unfolding and the time taken to compute status sets as there are no good benchmark agent programs to test against, and no easy way to vary the very large number of parameters associated with an agent.

In a sample application shown in Figures 8.5 on page 445 and 8.6 on page 446, we noticed that it took about 1 second to unfold a program containing 11 rules, and to evaluate the status set took about 30 seconds.

However, in this application, massive amounts of Army War reserves data resident in Oracle as well as in a multi-record, nested, unindexed flat file were accessed, and the time reported (30 seconds) includes times taken for Oracle and the flat file to do their work, plus network times. Network cost alone is about 25 seconds. We did not yet implement any optimizations, like caching etc.

References

Apt, K., H. Blair, and A. Walker (1988). Towards a Theory of Declarative Knowledge. In J. Minker (Ed.), *Foundations of Deductive Databases and Logic Programming*, pp. 89–148. Washington DC: Morgan Kaufmann.

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.

Chen, Z.-Z. and S. Toda (1995). The Complexity of Selecting Maximal Solutions. *Information and Computation 119*, 231–239.

Eiter, T., V. Subrahmanian, and T. Rogers (1999, May). Heterogeneous Active Agents, III: Polynomially Implementable Agents. Technical Report INFSYS RR-1843-99-07, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria. to appear in AI.

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

Rogers Jr., H. (1967). *Theory of Recursive Functions and Effective Computability*. New York: McGraw-Hill.

Vardi, M. (1982). Complexity of Relational Query Languages. In *Proceedings 14th ACM Symposium on Theory of Computing*, San Francisco, CA, pp. 137–146. ACM Press.

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.