IMPACT: A Platform for Heterogenous Agents

Lecture Course given at Ushuaia, Argentina

October 2000

Jürgen Dix,
University of Koblenz and Technical University of Vienna

Based on the book

Heterogenous Active Agents
(Subrahmanian, Bonatti, Dix,
Eiter, Kraus, Özcan and Ross),
MIT Press, May 2000.

Timetable:

- 10 minutes to explain what is going on. Some sentences for each chapter.
- Chapter 1 can be entirely done in the remaining time.

3. Actions and Agent Programs

Overview

Timetable:

- Chapter 3 needs 1 lecture, but without detailed discussion of the semantics.

3 Actions and Agent Programs

A Single *agent*

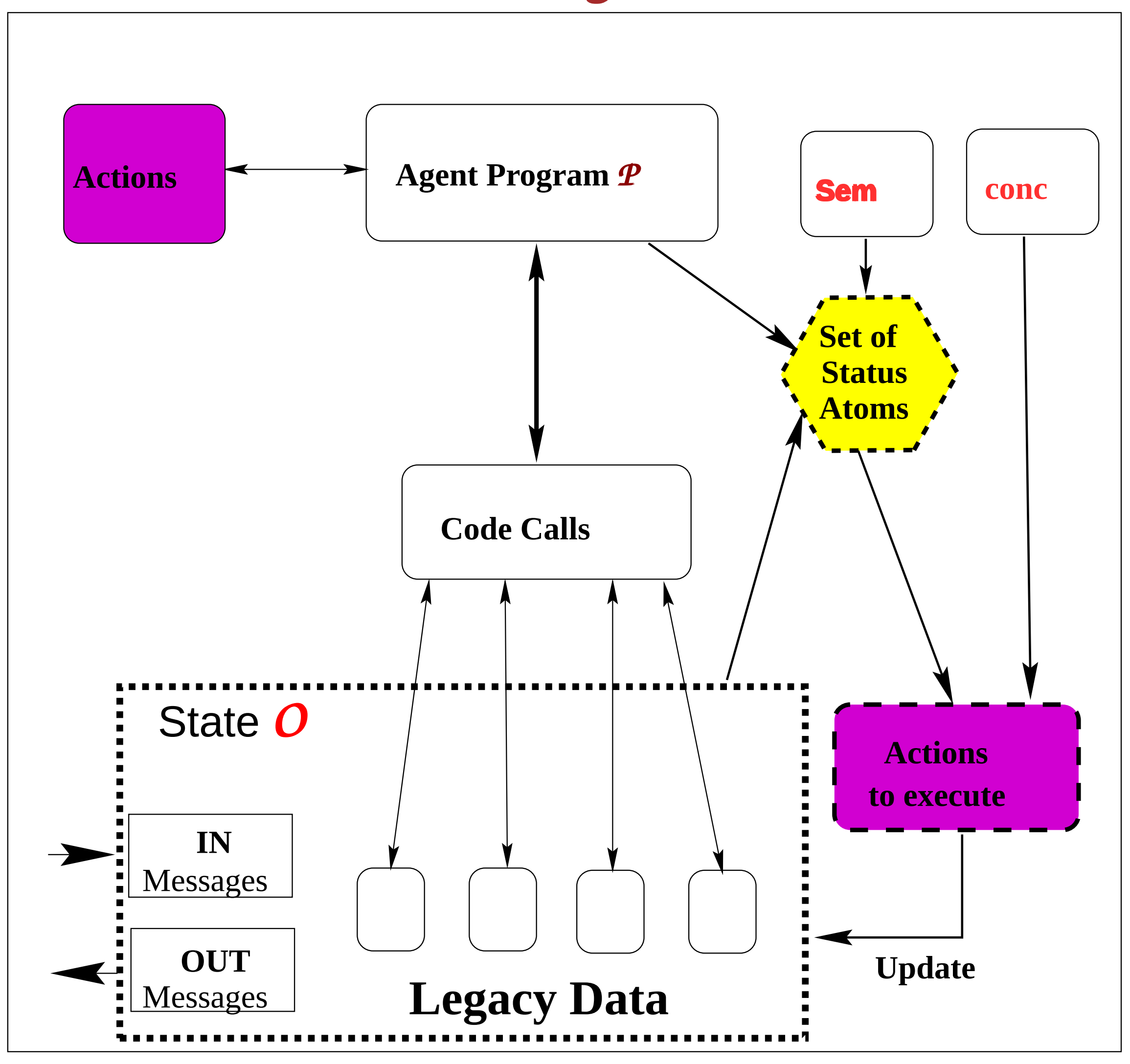

Figure 3.1: Agent Decision Architecture

Underlying Software Code: Basic set of data structures and legacy code 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***. Clearly, the state of the agent varies with time.

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

Actions: Each agent has an associated set of **actions**. An action is implemented by a body of code implemented in any suitable imperative (or declarative) programming language.

Action Constraints: In certain cases, the creator of the agent may wish to prevent the agent from concurrently executing certain actions even though it may be feasible for the agent to take them.

Agent Programs: Finally, an agent program is a set of rules, in a language to be defined, that an agent's creator might use to specify the principles according to which the agent behaves, and the policies governing what actions the agent takes, from among a possible plethora of possible actions.

In short, the **agent program** associated with an agent **encodes the "do's and dont's" of the agent**.

3.1 Action Base

Definition 3.1 (Action; Action Atom)
An action α *consists of six components:*

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

Schema: *A schema, usually written as* $(\tau_1,\ldots,\tau_n)$*, of types. Intuitively, this says that the variable* X_i *must be of type* 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*(α) *(Pre*(α) *must be* safe modulo the variables $\mathtt{X}_1, \ldots, \mathtt{X}_n$*);*

Add: *a set Add*(α) *of code-call conditions;*

Del: *a set Del*(α) *of code-call conditions.*

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

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

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 3.1 (CHAIN Revisited)
Suppose the supplier *agent of the CHAIN example has*

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

Schema: $(\mathtt{String}, \mathtt{Integer}, \mathtt{String})$

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

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

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

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

Example 3.2 (CFIT Revisited)

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

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.*

Example 3.3 (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.*

Definition 3.2 (Action Base)

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

3.2 Execution and Concurrency of Actions

What is the result of executing an action?

Definition 3.3 ((θ,γ)-Executability)
Let $\alpha(\vec{X})$ be an action, and let $\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ be an underlying software code accessible to the agent. A ground instance $\alpha(\vec{X})\theta$ of $\alpha(\vec{X})$ is said to be executable *in state $O_{\mathcal{S}}$, if, by definition, there exists a solution γ of $Pre(\alpha(\vec{X}))\theta$ w.r.t. $O_{\mathcal{S}}$. In this case, $\alpha(\vec{X})$ is said to be* (θ,γ)-executable *in state $O_{\mathcal{S}}$, and $(\alpha(\vec{X}),\theta,\gamma)$ is a feasible* execution triple *for $O_{\mathcal{S}}$.*

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

Intuitively, in $\alpha(\vec{X})$, the substitution θ causes all variables in $\vec{X}$ to be grounded. However, it is entirely possible that the precondition of α has occurrences of other free variables not occurring in $\vec{X}$. Appropriate ground values for these variables are given by solutions of $Pre(\alpha(\vec{X})\theta)$ with respect to the current state O_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.

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

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

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

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

Definition 3.5 (Concurrency Notion)
A notion of concurrency *is a function,* **conc***, that takes as input, an object state,* $\mathcal{O}_{\mathcal{S}}$*, and a set of execution triples AS, and returns as output, a single new execution triple such that:*

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

Definition 3.6 (Weakly Concurrent Execution)
Suppose AS is a set of feasible execution triples in the agent state $\mathcal{O}_{\mathcal{S}}$*. The* weakly concurrent execution of *AS in* $\mathcal{O}_{\mathcal{S}}$*, is defined to be the agent state*

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

where

$$O_{add} =_{def} \bigcup_{(\alpha(\vec{X}),\theta,\gamma)\in AS} \mathcal{O}\_Sol(Add(\alpha(\vec{X})\theta)\gamma),$$

$$O_{del} =_{def} \bigcup_{(\alpha(\vec{X}),\theta,\gamma)\in AS} \mathcal{O}\_Sol(Del(\alpha(\vec{X})\theta)\gamma).$$

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

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

of all feasible execution triples stemming from some grounded action in AS, and $\textbf{\textit{apply}}(A,\mathcal{O}_S)$ *denotes the resulting state.*

Definition 3.7 (Sequential-Concurrent Execution)
Suppose we have a set $AS =_{def} \{(\alpha_i(\vec{X}_i, \theta_i, \gamma_i)) \mid 1 \leq i \leq n\}$ *of feasible execution triples on an agent state* $\mathcal{O}_S$*. Then, AS is said to be S-*concurrently executable *in state* $\mathcal{O}_S$*, if, by definition, there exists a permutation* π *of AS and a sequence of states* $\mathcal{O}_S^0, \ldots, \mathcal{O}_S^n$ *such that:*

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

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

A set ACS of actions is S-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 S-concurrently executable on* $\mathcal{O}_S$.

Definition 3.8 (Full-Concurrent Execution)
Suppose we have a set $AS =_{def} \{(\alpha_i(\vec{X}_i, \theta_i, \gamma_i)) \mid 1 \leq i \leq n\}$ *of feasible execution triples and an agent state* O_S*. Then, AS is said to be* F-concurrently executable *in state* O_S *, if and only if the following holds:*

1. *For every permutation* π*, AS is* π*-executable.*
2. *For any two permutations* π_1, π_2 *of AS, the final states* $AS[\pi_1]$ *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* 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* O_S.

Example 3.4 (CHAIN Revisited)

Consider the following set of action executions:

$$\begin{aligned}&\mathit{update\_stockDB}(\texttt{widget5}, \texttt{250}, \texttt{companyA}),\\&\mathit{update\_stockDB}(\texttt{widget10}, \texttt{100}, \texttt{companyB}),\\&\mathit{update\_stockDB}(\texttt{widget5}, \texttt{500}, \texttt{companyB}).\end{aligned}$$

The uncommitted stock database contains $\langle\texttt{widget5}, \texttt{1000}\rangle$, $\langle\texttt{widget10}, \texttt{500}\rangle$ *and* $\langle\texttt{widget15}, \texttt{1500}\rangle$, *and the committed stock database contains* $\langle\texttt{widget5}, \texttt{2000}\rangle$, $\langle\texttt{widget10}, \texttt{900}\rangle$ *and* $\langle\texttt{widget15}, \texttt{1500}\rangle$. *Weak concurrent execution of these actions will attempt to execute an action, having delete list*

$$\begin{aligned}&\mathbf{in}(\texttt{X}, \texttt{supplier}:\mathit{select}('\mathit{uncommitted}', \texttt{id}, =, \texttt{widget5})),\\&\mathbf{in}(\texttt{Y}, \texttt{supplier}:\mathit{select}('\mathit{committed}', \texttt{id}, =, \texttt{widget5})),\\&\mathbf{in}(\texttt{X}, \texttt{supplier}:\mathit{select}('\mathit{uncommitted}', \texttt{id}, =, \texttt{widget10})),\\&\mathbf{in}(\texttt{Y}, \texttt{supplier}:\mathit{select}('\mathit{committed}', \texttt{id}, =, \texttt{widget10})).\end{aligned}$$

It is important to note that even though we should have two "copies" each of the first two code calls above, one copy suffices, because weak concurrent executions considers the union of the delete lists and the union of the add list. Likewise, this action has the add list

$$\mathbf{in}(\langle \mathtt{widget5}, 750\rangle, \mathtt{supplier}:\mathit{select}('uncommitted', \mathtt{id}, =, \mathtt{widget5})),$$

$$\mathbf{in}(\langle \mathtt{widget5}, 500\rangle, \mathtt{supplier}:\mathit{select}('uncommitted', \mathtt{id}, =, \mathtt{widget5})),$$

$$\mathbf{in}(\langle \mathtt{widget5}, 2250\rangle, \mathtt{supplier}:\mathit{select}('committed', \mathtt{id}, =, \mathtt{widget5})),$$

$$\mathbf{in}(\langle \mathtt{widget5}, 2500\rangle, \mathtt{supplier}:\mathit{select}('committed', \mathtt{id}, =, \mathtt{widget5})).$$

$$\mathbf{in}(\langle \mathtt{widget10}, 400\rangle, \mathtt{supplier}:\mathit{select}('uncommitted', \mathtt{id}, =, \mathtt{widget10})),$$

$$\mathbf{in}(\langle \mathtt{widget10}, 1000\rangle, \mathtt{supplier}:\mathit{select}('committed', \mathtt{id}, =, \mathtt{widget10})).$$

We see that the above executions lead to an intuitively inconsistent state in which the committed stock database claims that the number of committed items of widget 5 is both 2250 and 2500 !

Example 3.5 (CHAIN example revisited)
Let us return to the situation raised in Example 3.4 on page 141. The following set of action executions are F-concurrently executable:

$$\textit{update\_stockDB}(\texttt{widget5}, \texttt{250}, \texttt{companyA}),$$
$$\textit{update\_stockDB}(\texttt{widget10}, \texttt{100}, \texttt{companyB}),$$
$$\textit{update\_stockDB}(\texttt{widget15}, \texttt{500}, \texttt{companyB}).$$

Further assume that the uncommitted stock database contains the same tuples as in Example 3.4 on page 141. This set of action executions is F-concurrently executable on this state of the supplier *agent, because any permutation of these three actions will result in the same final agent state. That is, whatever the execution sequence is, the resulting uncommitted stock database will contain the following tuples:* $\langle\texttt{widget5}, \texttt{750}\rangle$, $\langle\texttt{widget10}, \texttt{400}\rangle$ *and* $\langle\texttt{widget15}, \texttt{1000}\rangle$.

Comment 4 *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***."*

3.3 Action Constraints

Definition 3.9 (Action Constraint)
An action constraint AC has the syntactic form:

$$\{\alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)\} \hookleftarrow \chi \tag{3.1}$$

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

Example 3.6 (CHAIN Example Revisited)

The following are some action constraints for the **supplier** *agent of CHAIN example:*

$$\begin{aligned}
&\{\ \mathit{update\_stockDB}(\mathtt{Part\_id1}, \mathtt{Amount1}, \mathtt{Company1}),\\
&\ \mathit{update\_stockDB}(\mathtt{Part\_id2}, \mathtt{Amount2}, \mathtt{Company2})\ \} \hookleftarrow\\
&\qquad\qquad \mathtt{Part\_id1} = \mathtt{Part\_id2}\ \&\\
&\qquad\qquad \mathbf{in}(\mathtt{X}, \mathtt{supplier} : \mathit{select}('\mathit{uncommitted}', \mathtt{id}, =, \mathtt{Part\_id1}))\ \&\\
&\qquad\qquad \mathtt{X.amount} < \mathtt{Amount1} + \mathtt{Amount2}\ \&\\
&\qquad\qquad \mathtt{Company1} \neq \mathtt{Company2}.
\end{aligned}$$

$$\{\textit{respond\_request}(\texttt{Part\_id1},\texttt{Amount1},\texttt{Company1}),\\ \textit{respond\_request}(\texttt{Part\_id2},\texttt{Amount2},\texttt{Company2})\ \} \hookleftarrow \texttt{Part\_id1} = \texttt{Part\_id2}\ \&\\ \texttt{Company1} \neq \texttt{Company2}.$$

The first constraint states that if the two update_stockDB actions update the same `Part_id` *and the total amount available is less than the sum of the requested amounts, then these actions cannot be concurrently executable. The second constraint states that if two companies request the same* `Part_id`*, then the* **supplier** *agent does not respond to them concurrently. That is, the* **supplier** *agent processes requests one at a time.*

Example 3.7 (CFIT Example Revisited)

The following is an action constraint for the **autoPilot** *agent:*

$$\{\mathit{compute\_currentLocation}(\mathtt{Report}),$$
$$\mathit{adjust\_course}(\mathtt{No\_go}, \mathtt{FlightRoute}, \mathtt{CurrentLocation})\} \hookleftarrow$$

This action constraint states that the actions compute_currentLocation and adjust_course may never be executed concurrently. This is because the adjust_course action requires the current location of the plane as input, and the compute_currentLocation action computes the required input.

The following example shows an action constraint for the `gps` *agent:*

$$\{\textit{collect\_data}(\texttt{Satellite}), \textit{merge\_data}(\texttt{Satellite1}, \texttt{Satellite2})\} \hookleftarrow$$
$$\texttt{Satellite} = \texttt{Satellite1}.$$
$$\{\textit{collect\_data}(\texttt{Satellite}), \textit{merge\_data}(\texttt{Satellite1}, \texttt{Satellite2})\} \hookleftarrow$$
$$\texttt{Satellite} = \texttt{Satellite2}.$$

These two action constraints state that the `gps` *agent cannot concurrently execute the action merge_data and collect_data, if the satellite it is collecting data from is one of the satellites whose data it is merging.*

Example 3.8 (STORE Example Revisited)
The following are some action constraints for the profiling *agent in the STORE example:*

$$\{\mathit{update\_highProfile}(\mathtt{Ssn1},\mathtt{Name1},\mathtt{profile}),\mathit{update\_lowProfile}(\mathtt{Ssn2},\mathtt{Name2},\mathtt{profile})\} \hookleftarrow$$
$$\mathbf{in}(\mathtt{spender}(\mathtt{high}),\mathtt{profiling}:\mathit{classifyUser}(\mathtt{Ssn1}))$$
$$\mathtt{Ssn1}=\mathtt{Ssn2}\ \ \mathtt{Name1}=\ \mathtt{Name2}$$

$$\{\mathit{update\_userProfile}(\mathtt{Ssn1},\mathtt{Name1},\mathtt{Profile}),\mathit{classify\_user}(\mathtt{Ssn2},\mathtt{Name2})\} \hookleftarrow$$
$$\mathtt{Ssn1}=\mathtt{Ssn2}\ \&\ \mathtt{Name1}=\mathtt{Name2}$$

The first action states that if the user is classified as a high spender, then the `profiling` *agent cannot execute* ***update_highProfile*** *and* ***update_lowProfile*** *concurrently. In contrast, the second action constraint states that the* `profiling` *agent cannot classify a user profile if it is currently updating the profile of that user.*

Definition 3.10 (Action Constraint Satisfaction)
A set S of ground actions satisfies an action constraint AC as in (3.1) on a state $\mathcal{O}_{\mathcal{S}}$, denoted $S, \mathcal{O}_{\mathcal{S}} \models AC$, if there is no legal assignment θ of objects in $\mathcal{O}_{\mathcal{S}}$ to the variables in $\mathcal{AC}$ such that $\chi\theta$ is true and $\{\alpha_1(\vec{X})\theta, \ldots, \alpha_k(\vec{X})\theta\} \subseteq S$ holds (i.e., no concurrent execution of actions excluded by AC is included in S). We say that S satisfies *a set $\mathcal{AC}$ of actions constraints on $\mathcal{O}_{\mathcal{S}}$, denoted $S, \mathcal{O}_{\mathcal{S}} \models \mathcal{AC}$, if $S, \mathcal{O}_{\mathcal{S}} \models AC$ for every $AC \in \mathcal{AC}$.*

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

Example 3.9 (STORE Example Revisited)

Suppose our state consists of the three uncommitted stock database tuples given in Example 3.4 on page 141 and let $\mathcal{AC}$ *be the first action constraint given in Example 3.6 on page 146. Then if* S_1 *consists of*

$$\textit{update\_stockDB}(\texttt{widget5}, 250, \texttt{companyA}),$$
$$\textit{update\_stockDB}(\texttt{widget10}, 100, \texttt{companyB}),$$
$$\textit{update\_stockDB}(\texttt{widget5}, 500, \texttt{companyB})$$

and S_2 *consists of*

$$\textit{update\_stockDB}(\texttt{widget5}, 750, \texttt{companyA}),$$
$$\textit{update\_stockDB}(\texttt{widget10}, 100, \texttt{companyB}),$$
$$\textit{update\_stockDB}(\texttt{widget5}, 500, \texttt{companyB})$$

S_1 *satisfies* $\mathcal{AC}$ *but* S_2 *does not because* $(\texttt{Part\_id1} = \texttt{Part\_id2} = \texttt{widget5})$, *only* $\texttt{X.amount} = 1000$ *units of this part are available, and* $(\texttt{Amount1} + \texttt{Amount2}) = (750 + 500) \geq 1000$.

3.4 Agent Programs: Syntax

Thus far, we have introduced the following important concepts:

Software Code Calls ($\mathcal{S}:f(\mathtt{a}_1,\ldots,\mathtt{a}_n)$): this provides a single framework within which the interoperation of diverse pieces of software may be accomplished;

Software/Agent states ($\mathcal{O}_{\mathcal{S}}$): 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 ($\mathcal{AB}$): 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 ($\mathcal{AC}$): this specifies whether a certain set of actions is incompatible.

Definition 3.11 (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* $\mathbf{P}(\alpha(\vec{t}))$, $\mathbf{F}(\alpha(\vec{t}))$, $\mathbf{O}(\alpha(\vec{t}))$, $\mathbf{W}(\alpha(\vec{t}))$, *and* $\mathbf{Do}\,(\alpha(\vec{t}))$ *are* action status atoms.

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

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

Definition 3.12 (Action Rule)
An action rule *(*rule, *for short) is a clause r of the form*

$$\mathsf{Op}\alpha(\vec{t}) \leftarrow L_1, \ldots, L_n \tag{3.2}$$

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

Definition 3.13 (Safety)

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 in its body.
- For any rule r of the form (3.2), 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 3.14 (Agent Program)
An agent program $\mathcal{P}$ *is a finite collection of rules. An agent program* $\mathcal{P}$ *is* positive *if all its rules are positive.*

Example 3.10 (CHAIN Example Revisited)
The supplier *agent may use the agent program shown below. In the following rules, the* supplier *agent makes use of the message box to get various variables it needs. In order to extract variables, the* supplier *agent invokes the code call getVar of the message box domain.*

r1: $\mathbf{F}$ *update_stockDB*(Part_id, Amount_requested, Company) $\leftarrow$
$\mathbf{O}$ *process_request*(Msg.Id, Agent),
in(Amount_requested, msgbox : *getVar*(Msg.Id, "*Amount_requested*")),
in(Part_id, msgbox : *getVar*(Msg.Id, "*Part_id*")),
in(Company, msgbox : *getVar*(Msg.Id, "*Company*")),
in(amount_not_available, supplier : *monitorStock*(Amount_requested, Part_id))

This rule ensures that we cannot invoke update_stockDB when Amount_requested *exceeds the amount available.*

r2: $\mathbf{F}\ \textit{update\_stockDB}(\texttt{Part\_id1}, \texttt{Amount\_requested1}, \texttt{Company1}) \leftarrow$
$\mathbf{O}\ \textit{process\_request}(\texttt{Msg.Id1}, \texttt{Agent1}),$
$\mathbf{O}\ \textit{process\_request}(\texttt{Msg.Id2}, \texttt{Agent2}),$
$\mathbf{in}(\texttt{Amount\_requested1}, \mathtt{msgbox} : \textit{getVar}(\texttt{Msg.Id1}, \text{"}\textit{Amount\_requested1}\text{"})),$
$\mathbf{in}(\texttt{Amount\_requested2}, \mathtt{msgbox} : \textit{getVar}(\texttt{Msg.Id2}, \text{"}\textit{Amount\_requested2}\text{"})),$
$\mathbf{in}(\texttt{Part\_id1}, \mathtt{msgbox} : \textit{getVar}(\texttt{Msg.Id1}, \text{"}\textit{Part\_id1}\text{"})),$
$\mathbf{in}(\texttt{Part\_id2}, \mathtt{msgbox} : \textit{getVar}(\texttt{Msg.Id2}, \text{"}\textit{Part\_id2}\text{"})),$
$\mathbf{in}(\texttt{Company1}, \mathtt{msgbox} : \textit{getVar}(\texttt{Msg.Id1}, \text{"}\textit{Company1}\text{"})),$
$\mathbf{in}(\texttt{Company2}, \mathtt{msgbox} : \textit{getVar}(\texttt{Msg.Id2}, \text{"}\textit{Company2}\text{"})),$
$=(\texttt{Part\_id1}, \texttt{Part\_id2}),$
$\mathbf{Do}\ \textit{update\_stockDB}(\texttt{Part\_id2}, \texttt{Amount\_requested2}, \texttt{Company2}),$
$=(\texttt{Amount\_requested}, \texttt{Amount\_requested1} + \texttt{Amount\_requested2}),$
$\mathbf{in}(\texttt{amount\_not\_available}, \mathtt{supplier} : \textit{monitorStock}(\texttt{Amount\_requested}, \texttt{Part\_id}))$
$\texttt{Company1} \neq \texttt{Company2}$

This rule ensures that we do not invoke update_stockDB for `Amount_requested1` *units of* `Part_id1` *when the* `Amount_requested1` *exceeds the amount that will be available after our agent finishes the update_stockDB action for* `Amount_requested2` *units of* `Part_id2`.

r3: $\mathbf{O}\ \textit{order\_part}(\texttt{Part\_id}, \texttt{amount\_to\_order}) \leftarrow$
$\mathbf{O}\ \textit{process\_request}(\texttt{Msg.Id}, \texttt{Agent}),$
$\mathbf{in}(\texttt{Amount\_requested}, \texttt{msgbox}: \textit{getVar}(\texttt{Msg.Id}, \text{"}\textit{Amount\_requested}\text{"})),$
$\mathbf{in}(\texttt{Part\_id}, \texttt{msgbox}: \textit{getVar}(\texttt{Msg.Id}, \text{"}\textit{Part\_id}\text{"})),$
$\mathbf{in}(\texttt{supplies\_too\_low}, \texttt{supplier}: \textit{too\_low\_threshold}(\texttt{Part\_id})),$
$\mathbf{in}(\texttt{amount\_not\_available}, \texttt{supplier}: \textit{monitorStock}(\texttt{supplies\_too\_low}, \texttt{Part\_id})),$

If our supply for `Part_id` *falls below the* `supplies_too_low` *threshold, then we are obliged to order* `amount_to_order` *more units for this part. Note that* `amount_to_order` *and* `supplies_too_low` *represent integer constants.*

r4: $\mathbf{P}\ \textit{order\_part}(\texttt{Part\_id}, \texttt{amount\_to\_order}) \leftarrow$
$\mathbf{O}\ \textit{process\_request}(\texttt{Msg.Id}, \texttt{Agent})$,
$\mathbf{in}(\texttt{Amount\_requested}, \texttt{msgbox} : \textit{getVar}(\texttt{Msg.Id}, \text{"}\textit{Amount\_requested}\text{"}))$,
$\mathbf{in}(\texttt{Part\_id}, \texttt{msgbox} : \textit{getVar}(\texttt{Msg.Id}, \text{"}\textit{Part\_id}\text{"}))$,
$\mathbf{in}(\texttt{supplies\_low}, \texttt{supplier} : \textit{low\_threshold}(\texttt{Part\_id}))$,
$\mathbf{in}(\texttt{amount\_not\_available}, \texttt{supplier} : \textit{monitorStock}(\texttt{supplies\_low}, \texttt{Part\_id}))$,

If our supply for `Part_id` *falls below the* `supplies_low` *threshold, then we may order* `amount_to_order` *more units for this part. When* `supplies_low` $>$ `supplies_too_low`*, we may use rule r4 to reduce the number of times we need to invoke rule R3. Note that* `amount_to_order` *and* `supplies_too_low` *represent integer constants.*

r5: **W** *order_part*(Part_id, amount_to_order) ←
O *process_request*(Msg.Id, Agent),
in(Amount_requested, msgbox : *getVar*(Msg.Id, "*Amount_requested*")),
in(Part_id, msgbox : *getVar*(Msg.Id, "*Part_id*")),
in(supplies_low, supplier : *low_threshold*(Part_id)),

in(amount_not_available, supplier : *monitorStock*(supplies_low, Part_id)),
in(part_discontinued, supplier : *productStatus*(Part_id))

If the part Part_id *has been discontinued, we are not obliged to order* amount_to_order *more units of the part when supplies fall below our* supplies_too_low *threshold (i.e., when rule R3 is fired).*

r6: **O** *request*(*"plant"*, *"find:supplier"*) ←
O *process_request*(`Msg.Id`, `Agent`),
in(`Amount_requested`, **msgbox** : ***getVar***(`Msg.Id`, *"Amount_requested"*)),
in(`Part_id`, **msgbox** : ***getVar***(`Msg.Id`, *"Part_id"*)),
Do *order_part*(`Part_id`, `Amount_requested`)

If we decide to order `Amount_requested` *units of part* `Part_id`*, request the* **plant** *agent's find: supplier service to determine if there is a supplier which can provide* `Amount_requested` *units of* `Part_id`*. Note that the* **supplier** *agent does not know how the* **plant** *agent decides upon which manufacturing plant to use.*

r7: **O** *request*("*shipping*","*prepare:schedule(shipping*") ←
O *process_request*(Msg.Id, Agent),
O *process_request*((Msg.Id1, Agent1),
=(Agent1, plant),
in(Amount_requested, msgbox : *getVar*(Msg.Id, "*Amount_requested*")),
in(Part_id, msgbox : *getVar*(Msg.Id, "*Part_id*")),
in(Part_supplier, msgbox : *getVar*(Msg.Id1, "*Part_supplier*")),
Do *order_part*(Part_id, Amount_requested),

If we decide to order Amount_requested *units of part* Part_id, *we must also use the* shipping *agent's prepare: schedule(shipping) service to determine how and when the requested* Amount_requested *units can be shipped to us from the* Part_supplier, *which is determined by the* plant *agent.* Part_supplier *is extracted from a message sent from the* plant *agent in response to the* supplier *agent's request to the find: supplier service.*

r8: **O** *process_request*(`Msg.Id`, `Agent`) $\leftarrow$
in(`Msg`, `msgbox` : *getAllMsgs()*),
=(`Agent`,*Msg.Source*),

This rule says that the agent is obliged to process all requests in its message box from other agents. This does not mean that it will respond positively to a request.

r9: **O** *delete_msg*(`Msg.Id`) $\leftarrow$
Do *process_request*(`Msg.Id`, `Agent`)

This rule says that the agent deletes all messages that it has processed from its message box.

Before proceeding to discuss the formal semantics of agent programs, we quickly revisit the architecture of an agent's decisionmaking component shown in Figure 3.1 on page 121.

1. Every agent manages a **set of data types** through a set of well-defined *methods*.
2. These data types and methods include a message box data structure, with associated manipulation algorithms described in Chapter 3.
3. At a given point t in time, the **state of an agent**, $\mathcal{O}$, reflects the set of data items the agent currently has access to—these data items must all be of one of the data types alluded to above.
4. At time t, the agent may receive a set of *new* messages—these new messages constitute a *change* to the state of the agent.

5. The aforementioned changes may *trigger* one or more rules in the agent's associated agent program to become true. Based on the selected semantics for agent programs (to be described in Subsection 3.5), 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.
6. The actions that are supposed to be performed according to the above mentioned 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.
7. The cycle continues perpetually.

Algorithm 3.1 (Agent-Decision-Cycle)

```
  Agent-Decision-Cycle(Curr: agent_state;
                       $\mathcal{IC}$: integrity constraint set;
                       $\mathcal{AC}$: action constraint set;
                       $\mathcal{AB}$ : action base;
                       conc: notion of concurrency;
                       Newmsg: set of messages )

    1. while true do
    2. { DoSet := ComputeSem(Curr, $\mathcal{IC}$, $\mathcal{AC}$, $\mathcal{AB}$, conc, Newmsg);
         ($\star$ find a set of actions to execute based on messages received $\star$)
    3.   Curr := result of executing the single action conc(DoSet); }
end.
```

Example 3.11 (CHAIN Example Revisited)

Consider the agent program for the `supplier` *agent given in Example 3.10 on page 160.*

1. *Each time we sell supplies, our agent consults rules r1 and r2 to ensure that the amount requested never exceeds the amount available, even if the requests are coming from multiple companies. If two concurrent requests for the same part are considered by the* `supplier` *of Example 3.10 on page 160, and if both these requests can be individually (but not jointly) satisfied, then our current example non-deterministically satisfies one. The agent program in question does not adopt any preference strategies.*
2. *If we do not replenish our supplies, rule r4 will fire when our supply of part* `Part_id` *falls below the supplies_low threshold. Our agent is now allowed to order more supplies. If more supplies are not ordered, rule r3 will eventually fire when our supply of part* `Part_id` *falls below the supplies_too_low threshold. The agent is now obliged to order more parts. This obligation can be waived, however, if part* `Part_id` *has been discontinued (see rule r5).*

3. *When we order parts, rule r6 will fire. Here, the* `supplier` *agent consults the* `plant` *agent to determine which supplier to use. Once an appropriate supplier has been found, the* `supplier` *agent asks the* `shipping` *agent to provide a shipping schedule (rule r7) so the ordered parts can be delivered.*

It is easy to see, from rules (r8) and (r9) that the same message requesting parts will not be considered twice. Rule (r9) ensures that once a message is processed, it is deleted from the message box.

Example 3.12 (CFIT Example: Multiagent Interaction)
The reader may be wondering exactly how the agents in a multiagent application interact with one another. In this example, we provide a discussion of how this happens in a microcosm of the CFIT example. Appendix A of this book contains the full working code for agents in the CFIT example.

Consider the **autoPilot** *agent in the CFIT example. 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 associated with **autoPilot** *causes the* **Do** *wake action to be executed, which in turn triggers other actions. These include:*

- *Executing an action* ***sendMessage***$(\mathtt{autoPilot}, \mathtt{gps}, <service\_request>)$ *where the service request* $<service\_request>$ *of the* **gps** *agent is requesting the current location of the plane.*

- *The* **gps** *agent executes its* ***getAllMsgs*** *and retrieves the message sent by the* **autoPilot** *agent.*
- *The decision program of the* **gps** *agent executes this request and also executes the action* ***sendMessage***$\texttt{gps}, \texttt{autoPilot}, <answer>)$ *where* $<answer>$ *is the answer to the request made by the* **autoPilot** *agent.*
- *The* **autoPilot** *agent executes the* ***getAllMsgs*** *action and retrieves the message sent by the* **gps** *agent.*
- *The decision program of the* **autoPilot** *agent checks to see if the location of the plane sent by the GPS is where the flight plan says the plane should be. If yes, it executes the action* ***sleep*** *and goes to sleep for another* Δ *units of time. If not, it executes the action*

  $$\textit{sendMessage}(\texttt{autoPilot}, \texttt{terrain}, <request>)$$

  where $<request>$ *requests the* **terrain** *agent to send the elevation of the plane at its current location (as determined by the GPS agents) as well as send the No_go areas.*

- *The* `terrain` *agent executes its* ***getAllMsgs*** *action and retrieves the message sent by the* `autoPilot` *agent.*
- *The decision program of the* `terrain` *agent executes this request and also executes the action* ***sendMessage***(`terrain`, `autoPilot`, `Ans`) *where* `Ans` *is the answer to the request made by the* `autoPilot` *Agent.*
- *The* `autoPilot` *agent executes the* ***getAllMsgs*** *action and retrieves the message sent by the* `terrain` *agent.*
- *It then executes its* ***replan*** *action with the new terrain (correct) location of the plan and the terrain "no go" areas.*

3.5 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 **Do**α *that are true with respect to* $\mathcal{P}$*, the current state,* $O_{\mathcal{S}}$*, the underlying set* $\mathcal{AC}$ *of action constraints, and the set* $\mathcal{IC}$ *of underlying integrity constraints on agent states?*

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

While feasible status sets do not constitute a semantics for agent programs, every semantics we define for Agent Programs will build upon this basic definition.

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

Definition 3.15 (Status Set)

A status set *is any set* S *of ground action status atoms over* $\mathcal{S}$. *For any operator* $\mathit{Op} \in \{\mathbf{P}, \mathbf{Do}, \mathbf{F}, \mathbf{O}, \mathbf{W}\}$, *we denote by* $\mathit{Op}(S)$ *the set* $\mathit{Op}(S) = \{\alpha \mid \mathit{Op}(\alpha) \in S\}$.

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

Definition 3.16 (Deontic and Action Consistency)
A status set S is called deontically consistent*, if, by definition, it satisfies the following rules for any ground action α:*

- *If $\mathbf{O}\alpha \in S$, then $\mathbf{W}\alpha \notin S$*
- *If $\mathbf{P}\alpha \in S$, then $\mathbf{F}\alpha \notin S$*
- *If $\mathbf{P}\alpha \in S$, then $\mathcal{O}_S \models \exists^* Pre(\alpha)$, where $\exists^* Pre(\alpha)$ denotes the existential closure of $Pre(\alpha)$, i.e., all free variables in $Pre(\alpha)$ are governed by an existential quantifier.*

  This condition means that the action α is in fact executable in the state $\mathcal{O}_S$.

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

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

- if $\mathbf{O}\alpha$ is in *S*, then we expect that $\mathbf{P}\alpha$ is also in *S*, and
- if $\mathbf{O}\alpha$ is in *S*, then we would like to have $\mathbf{Do}\,\alpha$ in *S*.

Definition 3.17 (Deontic and Action Closure)
The deontic closure of a status S, denoted $\mathbf{D\text{-}Cl}(S)$*, is the closure of S under the rule*

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

where α is any ground action. We say that S is deontically closed*, if* $S = \mathbf{D\text{-}Cl}(S)$ *holds.*

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

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

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

where α is any ground action. We say that a status S is action-closed, if $S = \mathbf{A\text{-}Cl}(S)$ *holds.*

Proposition 3.1
Suppose S is 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 3.18 (Operator App$_{\mathcal{P},\mathcal{O}_S}(S)$)

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

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

3.6 Feasible Status Sets

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

Definition 3.19 (Feasible Status Set)
Let $\mathcal{P}$ be an agent program and let $\mathcal{O}_\mathcal{S}$ be an agent state. Then, a status set S is a feasible status set *for $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, if the following conditions hold:*

(S1) *(closure under the program rules)* $\mathbf{App}_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S) \subseteq S$;

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

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

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

In general, there are action programs that have zero, one or several feasible status sets. This is illustrated through the following examples.

Example 3.13 (CHAIN example revisited)
Let us consider a simple agent program containing just the rule (r4) of Example 3.10, together with rule (r8) and (r9) that manage the message box.

r4: $\mathbf{P}\ \mathit{order\_part}(\texttt{Part\_id}, \texttt{amount\_to\_order}) \leftarrow$
$\mathbf{O}\ \mathit{process\_request}(\texttt{Msg.Id}, \texttt{Agent}),$
$\mathbf{in}(\texttt{Amount\_requested}, \mathbf{msgbox}:\mathit{getVar}(\texttt{Msg.Id}, \text{"}\mathit{Amount\_requested}\text{"})),$
$\mathbf{in}(\texttt{Part\_id}, \mathbf{msgbox}:\mathit{getVar}(\texttt{Msg.Id}, \text{"}\mathit{Part\_id}\text{"})),$
$\mathbf{in}(\texttt{supplies\_low}, \mathbf{supplier}:\mathit{low\_threshold}(\texttt{Part\_id})),$
$\mathbf{in}(\texttt{amount\_not\_available}, \mathbf{supplier}:\mathit{monitorStock}(\texttt{supplies\_low}, \texttt{Part\_id})).$

Suppose the current state of the agent **supplier** *is such that the number of items of a certain part say (p50) falls below the* `supplies_low` *threshold for that part. Suppose the company making the request is* `zzz_corp`*, and the* `Amount_requested` *is 50, and the* `amount_to_order` *is 2000. In this case, this agent program will have multiple feasible status sets. Some feasible status sets will contain* **P** ***order_part***(p50, 2000) *but will not contain* **Do** ***order_part***(p50, 2000). *However, other feasible status sets will contain both* **P** ***order_part***(p50, 2000) *and* **Do** ***order_part***(p50, 2000).

Example 3.14 (CHAIN example revisited)

On the other hand, suppose our agent program contains rules (r3), (r8) and (r9) of Example 3.10 on page 160, and suppose that for all parts, the amount of the part in stock is above the `too_low_threshold` *amount. Further, suppose our agent program contains the rule*

$$\begin{aligned}
&\mathbf{F}\ \mathit{order\_part}(\mathtt{Part\_id}, \mathtt{Amount\_requested}) \leftarrow \\
&\quad \mathbf{O}\ \mathit{process\_request}(\mathtt{Msg.Id}, \mathtt{Agent}), \\
&\quad \mathbf{in}(\mathtt{Amount\_requested}, \mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id}, \text{"}\mathit{Amount\_requested}\text{"})), \\
&\quad \mathbf{in}(\mathtt{Part\_id}, \mathtt{msgbox}:\mathit{getVar}(\mathtt{Msg.Id}, \text{"}\mathit{Part\_id}\text{"})), \\
&\quad \neg\mathbf{O}\ \mathit{order\_part}(\mathtt{Part\_id}, \mathtt{Amount\_requested}).
\end{aligned}$$

In this case, for all parts, we are forbidden from placing an order. Hence, this agent program has only one feasible status set, viz. that which contains status atoms of the form

$$\mathbf{F}\ \mathit{order\_part}(\mathtt{Part\_id}, \mathtt{Amount\_requested})$$

together with relevant message processing action status atoms .

Example 3.15

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

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

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.

Proposition 3.2 (Properties of Feasible Status Sets)

Let S be a feasible status set. Then,

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

We note that feasible status sets may include **Do**ing actions that are not strictly necessary.

Example 3.16 (Expanded CHAIN Example)
Let us return to the example feasible status sets we saw in Example 3.13 on page 186. In this case, this agent program had multiple feasible status sets. Some feasible status sets will contain **P** ***order_part***$(\mathtt{p50}, 2000)$ *but will not contain* **Do** ***order_part***$(\mathtt{p50}, 2000)$. *However, other feasible status sets will contain both* **P** ***order_part***$(\mathtt{p50}, 2000)$ *and* **Do** ***order_part***$(\mathtt{p50}, 2000)$. *It is immediately apparent that we do not want* both *action status atoms* **P** ***order_part***$(\mathtt{p50}, 2000)$ *and* **Do** ***order_part***$(\mathtt{p50}, 2000)$ *to be present in feasible status sets: there is no good reason to in fact perform the action* ***order_part***$(\mathtt{p50}, 2000)$ *(the agent program in question does not mandate that* **Do** ***order_part***$(\mathtt{p50}, 2000)$ *be true).*

3.7 Rational Status Sets

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

Definition 3.20 (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 3.17 (Expanded CHAIN Example Continued)
Returning to Example 3.16 on page 190, it is immediately apparent that all feasible status sets that contain both **P** ***order_part***(P,N) *and* **Do** ***order_part***(P,N) *are not rational, while those that only contain* **P** ***order_part***(P,N) *satisfy rationality.*

Observe that the definition of groundedness does not include condition $(S4)$ of a feasible status set. A moment of reflection will show that omitting this condition is indeed appropriate.

Recall that the integrity constraints must be maintained when the current agent state is changed into a new 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 $\mathcal{O}_{\mathcal{S}}$ an operator $\mathbf{T}_{\mathcal{P},\mathcal{O}_{\mathcal{S}}}$ that maps a status set S to another status set.

Definition 3.21 ($\mathbf{T}_{\mathcal{P},\mathcal{O}_{\mathcal{S}}}$ Operator)
Suppose $\mathcal{P}$ is an agent program and $\mathcal{O}_{\mathcal{S}}$ an agent state. Then, for any status set S,

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

Lemma 3.1

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

Theorem 3.1

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

Corollary 1

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

Example 3.18 (CHAIN example revisited)

Let us return to the agent program described in Example 3.16 on page 190. Let us augment this example with a new action, fax_order. Suppose we augment our agent program of Example 3.16 on page 190 with the two rules

$$
\begin{array}{l}
\mathbf{Do}\ \textit{fax\_order}(\texttt{company1}, \texttt{Part\_id}, \texttt{Amount\_requested}) \leftarrow \\
\quad \mathbf{O}\ \textit{process\_request}(\texttt{Msg.Id}, \texttt{Agent}), \\
\quad \mathbf{in}(\texttt{Amount\_requested}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, "\textit{Amount\_requested}")), \\
\quad \mathbf{in}(\texttt{Part\_id}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, "\textit{Part\_id}")), \\
\quad \mathbf{Do}\ \textit{order\_part}(\texttt{Part\_id}, \texttt{Amount\_requested}), \\
\quad \neg\ \mathbf{Do}\ \textit{fax\_order}(\texttt{company2}, \texttt{Part\_id}, \texttt{Amount\_requested}).
\end{array}
$$

$$
\begin{array}{l}
\mathbf{P}\ \textit{fax\_order}(\texttt{company2}, \texttt{Part\_id}, \texttt{Amount\_requested}) \leftarrow \\
\quad \mathbf{O}\ \textit{process\_request}(\texttt{Msg.Id}, \texttt{Agent}), \\
\quad \mathbf{in}(\texttt{Amount\_requested}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, "\textit{Amount\_requested}")), \\
\quad \mathbf{in}(\texttt{Part\_id}, \mathtt{msgbox}:\textit{getVar}(\texttt{Msg.Id}, "\textit{Part\_id}")), \\
\quad \mathbf{Do}\ \textit{order\_part}(\texttt{Part\_id}, \texttt{Amount\_requested}), \\
\quad =\!(\textit{Part\_id}, \textit{p50}).
\end{array}
$$

It is now easy to see that there are two rational status sets—one of which contains the status atom **Do** *fax_order*(`company1`, `Part_id`, `2000`) *and the other* **Do** *fax_order*(`company2`, `Part_id`, `2000`). *Thus, the introduction of negated status atoms in rule bodies leads to this potential problem.*

As shown by Example 3.18 on page 197 Corollary 1 on page 196 is no longer true in the presence of negated action status atoms .

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

Proposition 3.3
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.

3.8 Reasonable Status Sets

A more serious attack against rational status sets, is that for agent programs with negation, the semantics of rational status sets allows logical contraposition of the program rules. For example, consider the following program:

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

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

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

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

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

Definition 3.22 (Reasonable Status Set)
Let $\mathcal{P}$ be an agent program, let $\mathcal{O}_S$ be an agent state, and let S be a status set.

1. *If $\mathcal{P}$ is a positive agent program, then S is a* reasonable status set *for $\mathcal{P}$* on *$\mathcal{O}_S$, if and only if 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 which is obtained 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 $red^S(\mathcal{P}, \mathcal{O}_S)$ with respect to $\mathcal{O}_S$.*

Example 3.19 (CHAIN example revisited)
Let us return to the case of the agent program presented in Example 3.18 on page 197. Here we have two rational status sets, one containaing **Do** *fax_order*(company1, p50, 500), *while the other contains* **Do** *fax_order*(company2, p50, 500).

According to the above definition, only the rational status set that contains the status atom **Do** *fax_order*(company1, p50, 500) *is reasonable. The reason is that the first rule listed explicitly in Example 3.18 on page 197 says that if we do not infer* **Do** *fax_order*(company2, p50, 500), *then we should infer* **Do** *fax_order*(company1, p50, 500), *thus implicitly providing higher priority to the rational status set containing* **Do** *fax_order*(company1, p50, 500),.

A more simplistic example is presented below.

Example 3.20
For the program $\mathcal{P}$:

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

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

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

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

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

$$\mathbf{Do}\,\alpha \leftarrow \neg\mathbf{F}\alpha \tag{3.3}$$

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

Proposition 3.4
Let $\mathcal{P}$ be an agent program and $O_{\mathcal{S}}$ an agent state. Then, every reasonable status set of $\mathcal{P}$ on $O_{\mathcal{S}}$ is a rational status set of $\mathcal{P}$ on $O_{\mathcal{S}}$.

3.9 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: $\{\mathbf{P}, \mathbf{F}, \mathbf{O}, \mathbf{W}, \mathbf{Do}\}$.
2. The semantics is given by certain **status sets** of an agent program:
   (a) An agent program consists of rules $\mathbf{Op}\alpha \leftarrow \mathbf{Op}\beta_1, \ldots, \mathbf{Op}\beta_n, ccc_1, \ldots, ccc_n$.
   (b) A **feasible status set** is a set of status atoms $\{\mathsf{Op}_1\alpha_1, \ldots, \mathsf{Op}_n\alpha_n\}$ satisfying certain properties.
   (c) **Rational** status sets = Feasible + **Groundedness**
   (d) **Reasonable** status sets = **Rational** + **Contraposition not allowed**

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.

Arisha, K., F. Ozcan, R. Ross, V. S. 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.

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.

Dix, J., S. Kraus, and V. Subrahmanian (2001). Temporal agent reasoning. *Artificial Intelligence to appear*.

Dix, J., M. Nanni, and V. S. Subrahmanian (2000). Probabilistic agent reasoning. *Transactions of Computational Logic 1*(2).

Dix, J., V. S. Subrahmanian, and G. Pick (2000). Meta Agent Programs. *Journal of Logic Programming 46*(1-2), 1–60.

Eiter, T., V. Subrahmanian, and T. J. Rogers (2000). Heterogeneous Active Agents, III: Polynomially Implementable Agents. *Artificial Intelligence 117*(1), 107–167.

Eiter, T. and V. S. Subrahmanian (1999). Heterogeneous Active Agents, II: Algorithms and Complexity. *Artificial Intelligence 108*(1-2), 257–307.

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.

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