5. Agent Programs

Overview

5.1 Syntax

5.2 Status Sets

5.3 Feasible Status Sets

5.4 Rational Status Sets

5.5 Reasonable Status Sets

5.1 Agent Programs: Syntax

Thus far, we have introduced the following important concepts:

Software Code Calls: this provides a single framework within which the interoperation of diverse pieces of software may be accomplished;

Software/Agent states: this describes exactly what data objects are being managed by a software package at a given point in time;

Integrity Constraints: this specifies exactly which software states are “valid” or “legal”;

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

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

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

Definition 5.1 (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 5.2 (Action Rule)
An action rule *(*rule, *for short) is a clause r of the form*

$$A \leftarrow L_1, \ldots, L_n \tag{5.2}$$

where A is an action status atom, and each of $L_1, \ldots, L_n$ is either an action status atom, or a code call atom, each of which may be preceded by a negation sign ($\neg$).

Definition 5.3 (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 (5.2), we denote by $H(r)$, the atom in the head of r, and by $B(r)$, the collection of literals in the body; by $B^{-}(r)$ we denote the negative literals in $B(r)$, and by $B^{+}(r)$ the positive literals in $B(r)$.
- Moreover, by $\neg.B^{-}(r)$ we denote the atoms of the negative literals in $B^{-}(r)$.
- Finally, the index as (resp., cc) for any of these sets denotes restriction to the literals involving action status atoms (resp., code call atoms).

Definition 5.4 (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 5.1 (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.*

$$\textbf{r1:}\ \mathbf{F}\ \mathit{update\_stockDB}(\texttt{Part\_id}, \texttt{Amount\_requested}, \texttt{Company}) \leftarrow$$
$$\mathbf{O}\ \mathit{process\_request}(\texttt{Msg.Id}, \texttt{Agent}),$$
$$\textbf{in}(\texttt{Amount\_requested}, \mathrm{msgbox}: \mathit{getVar}(\texttt{Msg.Id}, \texttt{"}\textsf{\textit{Amount\_requested}}\texttt{"})),$$
$$\textbf{in}(\texttt{Part\_id}, \mathrm{msgbox}: \mathit{getVar}(\texttt{Msg.Id}, \texttt{"}\textsf{\textit{Part\_id}}\texttt{"})),$$
$$\textbf{in}(\texttt{Company}, \mathrm{msgbox}: \mathit{getVar}(\texttt{Msg.Id}, \texttt{"}\textsf{\textit{Company}}\texttt{"})),$$
$$\textbf{in}(\texttt{amount\_not\_available}, \mathrm{supplier}: \mathit{monitorStock}(\texttt{Amount\_requested}, \texttt{Part\_id}))$$

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

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

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: $\mathbf{O}$ *order_part*(`Part_id`, `amount_to_order`) $\leftarrow$
$\mathbf{O}$ *process_request*(`Msg.Id`, `Agent`),
in(`Amount_requested`, msgbox : *getVar*(`Msg.Id`, "*Amount_requested*")),
in(`Part_id`, msgbox : *getVar*(`Msg.Id`, "*Part_id*")),
in(`supplies_too_low`, supplier : *too_low_threshold*(`Part_id`)),
in(`amount_not_available`, supplier : *monitorStock*(`supplies_too_low`, `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: $\mathbf{P}\ \mathit{order\_part}(\texttt{Part\_id}, \texttt{amount\_to\_order}) \leftarrow$
$\mathbf{O}\ \mathit{process\_request}(\texttt{Msg.Id}, \texttt{Agent}),$
$\mathbf{in}(\texttt{Amount\_requested}, \texttt{msgbox}:\mathit{getVar}(\texttt{Msg.Id}, \texttt{"}\mathsf{Amount\_requested}\texttt{"})),$
$\mathbf{in}(\texttt{Part\_id}, \texttt{msgbox}:\mathit{getVar}(\texttt{Msg.Id}, \texttt{"}\mathsf{Part\_id}\texttt{"})),$
$\mathbf{in}(\texttt{supplies\_low}, \texttt{supplier}:\mathit{low\_threshold}(\texttt{Part\_id})),$
$\mathbf{in}(\texttt{amount\_not\_available}, \texttt{supplier}:\mathit{monitorStock}(\texttt{supplies\_low}, \texttt{Part\_id})),$

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.*

$$\begin{array}{l}
\textbf{r5: } \mathbf{W}\ \mathit{order_part}(\texttt{Part_id}, \texttt{amount_to_order}) \leftarrow \\
\mathbf{O}\ \mathit{process_request}(\texttt{Msg.Id}, \texttt{Agent}), \\
\textbf{in}(\texttt{Amount_requested}, \texttt{msgbox}:\mathit{getVar}(\texttt{Msg.Id}, \texttt{"}\mathit{Amount_requested}\texttt{"})), \\
\textbf{in}(\texttt{Part_id}, \texttt{msgbox}:\mathit{getVar}(\texttt{Msg.Id}, \texttt{"}\mathit{Part_id}\texttt{"})), \\
\textbf{in}(\texttt{supplies_low}, \texttt{supplier}:\mathit{low_threshold}(\texttt{Part_id})), \\
\\
\textbf{in}(\texttt{amount_not_available}, \texttt{supplier}:\mathit{monitorStock}(\texttt{supplies_low}, \texttt{Part_id})), \\
\textbf{in}(\texttt{part_discontinued}, \texttt{supplier}:\mathit{productStatus}(\texttt{Part_id}))
\end{array}$$

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: $\mathbf{O}\ \mathit{request}(\text{"}\textit{plant}\text{"}, \text{"}\textit{find:supplier}\text{"}) \leftarrow$
$\mathbf{O}\ \mathit{process\_request}(\texttt{Msg.Id}, \texttt{Agent})$,
$\textbf{in}(\texttt{Amount\_requested}, \mathrm{msgbox}: \mathit{getVar}(\texttt{Msg.Id}, \text{"}\textit{Amount\_requested}\text{"}))$,
$\textbf{in}(\texttt{Part\_id}, \mathrm{msgbox}: \mathit{getVar}(\texttt{Msg.Id}, \text{"}\textit{Part\_id}\text{"}))$,
$\mathbf{Do}\ \mathit{order\_part}(\texttt{Part\_id}, \texttt{Amount\_requested})$

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: $\mathbf{O}$ *request*("*shipping*", "*prepare:schedule(shipping*") $\leftarrow$
$\mathbf{O}$ *process_request*(Msg.Id, Agent),
$\mathbf{O}$ *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: $\mathbf{O}\ \mathit{process\_request}(\mathtt{Msg.Id}, \mathtt{Agent}) \leftarrow$
$\mathbf{in}(\mathtt{Msg}, \mathrm{msgbox}: \mathit{getAllMsgs}()),$
$=(\mathtt{Agent}, \mathit{Msg.Source}),$

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: $\mathbf{O}\ \mathit{delete\_msg}(\mathtt{Msg.Id}) \leftarrow$
$\mathbf{Do}\ \mathit{process\_request}(\mathtt{Msg.Id}, \mathtt{Agent})$

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

Example 5.2 (CFIT Example Revisited)
The agent program used by the $\mathfrak{autoPilot}$ *agent may use the following rules in its agent program. The same methods used in the Example 5.1 on page 158 apply here too.*

$\mathbf{r1:}\ \mathbf{F}\ \mathit{maintain\_course}(\mathtt{No\_go}, \mathtt{Flight\_route}, \mathtt{Current\_location}) \leftarrow$
$\mathbf{O}\ \mathit{process\_request}(\mathtt{Msg.Id}, \mathfrak{terrain}),$
$\mathbf{in}(\mathtt{No\_go}, \mathfrak{msgbox}: \mathit{getVar}(\mathtt{Msg.Id}, \mathtt{"No\_go"})),$
$\mathbf{in}(\mathtt{Current\_location}, \mathfrak{autoPilot}: \mathit{location}()),$
$\mathbf{in}(\mathtt{Flight\_route}, \mathfrak{autoPilot}: \mathit{getFlightRoute}()),$
$\mathbf{Do}\ \mathit{adjust\_course}(\mathtt{No\_go}, \mathtt{Flight\_route}, \mathtt{Current\_location})$

After receiving the `No_go` *regions from the* $\mathfrak{terrain}$ *agent, if the* $\mathfrak{autoPilot}$ *agent decides to adjust_course, it is no longer allowed to perform the maintain_course action. Note that the* $\mathfrak{autoPilot}$ *agent maintains the local variables* `Flight_route` *and* `Current_location`.

r2: $\mathbf{F}$ $\mathit{adjust\_course}(\texttt{No\_go}, \texttt{Flight\_route}, \texttt{Current\_location}) \leftarrow$
$\mathbf{O}$ $\mathit{process\_request}(\texttt{Msg.Id}, \mathtt{terrain})$,
in$(\texttt{No\_go}, \mathtt{msgbox}: \mathit{getVar}(\texttt{Msg.Id}, \texttt{"No\_go"}))$,
in$(\texttt{Current\_location}, \mathtt{autoPilot}: \mathit{location}())$,
in$(\texttt{Flight\_route}, \mathtt{autoPilot}: \mathit{getFlightRoute}())$,
$\mathbf{Do}$ $\mathit{maintain\_course}(\texttt{No\_go}, \texttt{Flight\_route}, \texttt{Current\_location})$

If the agent has decided to maintain_course, it is no longer allowed to perform the adjust_course action.

r3: $\mathbf{O}\ \mathit{return\_control}() \leftarrow$
$\textbf{in}(\texttt{manual\_override}, \mathrm{autoPilot}{:}\mathit{pilotStatus}(\texttt{pilot\_message}))$

If the pilot wants to disengage the auto-pilot, manual control should be returned.

r4: $\mathbf{P}$ *compute_currentLocation*(Report) $\leftarrow$
$\mathbf{O}$ *process_request*(Msg.Id, terrain),
in(No_go, msgbox: *getVar*(Msg.Id, "No_go")),
in(Report, msgbox: *getVar*(Msg.Id, "*Report*")),
in(Current_location, autoPilot: *location*()),
in(Flight_route, autoPilot: *getFlightRoute*()),
$\mathbf{Do}$ *adjust_course*(No_go, Flight_route, Current_location)

If the agent decides to adjust_course, it may invoke compute_currentLocation in order to update its estimate for Current_location.

r5: $\mathbf{O}$ *create_flight_plan*(No_go, Flight_route, Current_location) $\leftarrow$
$\mathbf{O}$ *process_request*(Msg.Id, terrain),
in(No_go, msgbox: *getVar*(Msg.Id, "No_go")),
in(Current_location, autoPilot: *location*()),
in(Flight_route, autoPilot: *getFlightRoute*()),
Do *adjust_course*(No_go, Flight_route, Current_location)

If the agent decides to adjust_course, it must create a flight plan which avoids the No_go *regions received from the* terrain *agent. The* No_go *regions are specified by the* terrain *agent's determine: area(no-go) service.*

$$\begin{aligned}
&\textbf{r6:}\ \mathbf{O}\ \mathit{execute_flight_plan}(\texttt{Flight_route}) \leftarrow \\
&\mathbf{O}\ \mathit{process_request}(\texttt{Msg.Id}, \mathtt{terrain}), \\
&\textbf{in}(\texttt{No_go}, \mathtt{msgbox}: \mathit{getVar}(\texttt{Msg.Id}, \texttt{"No_go"})), \\
&\textbf{in}(\texttt{Current_location}, \mathtt{autoPilot}: \mathit{location}()), \\
&\textbf{in}(\texttt{Flight_route}, \mathtt{autoPilot}: \mathit{getFlightRoute}()), \\
&\mathbf{Do}\ \mathit{adjust_course}(\texttt{No_go}, \texttt{Flight_route}, \texttt{Current_location})
\end{aligned}$$

Once a flight plan (i.e., `Flight_route`*) has been generated, we can execute this plan by invoking the execute_flight_plan action.*

r7: $\mathbf{O}\ \mathit{process\_request}(\mathtt{Msg.Id}, \mathtt{Agent}) \leftarrow$
in$(\mathtt{Msg}, \mathrm{msgbox} : \mathit{getAllMsgs}()),$
$=(\mathtt{Agent}, \mathit{Msg.Source})$

This rule says that the agent is obliged to process all requests in its message box. This does not mean that it will respond positively to a request. Note that the autoPilot *agent may receive messages from different agents, for example the* terrain *agent and the* gps *agent.*

r8: $\mathbf{O}\ \mathit{delete\_msg}(\mathtt{Msg.Id}) \leftarrow$
$\mathbf{Do}\ \mathit{process\_request}(\mathtt{Msg.Id}, \mathtt{Agent})$

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

Example 5.3 (STORE Example Revisited)
The agent program used by the profiling *agent may use the following rules in its agent program. The same methods used in Example 5.1 apply here too.*

$$
\begin{array}{l}
\textbf{r1:}\ \mathbf{F}\ \mathit{update\_highProfile}(\mathtt{Ssn},\mathtt{Name},\mathtt{Profile}) \leftarrow \\
\mathbf{Do}\ \mathit{process\_request}(\mathtt{Msg.Id},\mathtt{Agent}), \\
\mathbf{in}(\mathtt{Ssn},\mathrm{msgbox}:\mathit{getVar}(\mathtt{Msg.Id},\texttt{"}\mathit{Ssn}\texttt{"})), \\
\mathbf{in}(\mathtt{Name},\mathrm{msgbox}:\mathit{getVar}(\mathtt{Msg.Id},\texttt{"}\mathit{Name}\texttt{"})), \\
\mathbf{in}(\mathtt{Profile},\mathrm{msgbox}:\mathit{getVar}(\mathtt{Msg.Id},\texttt{"}\mathit{Profile}\texttt{"})), \\
\mathbf{in}(\mathtt{spender}(\mathtt{low}),\mathrm{profiling}:\mathit{classifyUser}(\mathtt{Ssn}))
\end{array}
$$

If a user is classified as spender(low), then we may not use $\mathit{update\_highProfile}$ *to update their profile.*

r2: $\mathbf{F}\ \mathit{update\_highProfile}(\texttt{Ssn},\texttt{Name},\texttt{Profile}) \leftarrow$
$\mathbf{Do}\ \mathit{process\_request}(\texttt{Msg.Id},\texttt{Agent}),$
$\textbf{in}(\texttt{Ssn},\mathtt{msgbox}:\mathit{getVar}(\texttt{Msg.Id},\texttt{"}\mathit{Ssn}\texttt{"})),$
$\textbf{in}(\texttt{Name},\mathtt{msgbox}:\mathit{getVar}(\texttt{Msg.Id},\texttt{"}\mathit{Name}\texttt{"})),$
$\textbf{in}(\texttt{Profile},\mathtt{msgbox}:\mathit{getVar}(\texttt{Msg.Id},\texttt{"}\mathit{Profile}\texttt{"})),$
$\textbf{in}(\texttt{spender}(\texttt{medium}),\mathtt{profiling}:\mathit{classifyUser}(\texttt{Ssn}))$

If a user is classified as spender(medium), then we may not use update_highProfile to update their profile. If the only other user classification is spender(high), then rules R1 and R2 together specify that only users which are high spenders can have their profiles modified by the update_highProfile action.

r3: $\mathbf{P}$ *request*("*credit*", "*provide:information(credit)*") $\leftarrow$
$\mathbf{Do}$ *process_request*(Msg.Id, Agent),
in(Ssn, msgbox : *getVar*(Msg.Id, "*Ssn*")),
in(Name, msgbox : *getVar*(Msg.Id, "*Name*")),
$\mathbf{Do}$ *classify_user*(Ssn, Name)

If the agent needs to classify a user, it can easily request the credit *agent's provide: information(credit) service to help decide which classification to use.*

$\mathbf{r4:}\ \mathbf{P}\ \mathit{request}(\texttt{"}\mathit{credit}\texttt{"},\texttt{"}\mathit{provide{:}address}\texttt{"}) \leftarrow$
$\mathbf{Do}\ \mathit{process\_request}(\texttt{Msg.Id},\texttt{Agent}),$
$\mathbf{in}(\texttt{Ssn},\mathrm{msgbox}:\mathit{getVar}(\texttt{Msg.Id},\texttt{"}\mathit{Ssn}\texttt{"})),$
$\mathbf{in}(\texttt{Name},\mathrm{msgbox}:\mathit{getVar}(\texttt{Msg.Id},\texttt{"}\mathit{Name}\texttt{"})),$
$\mathbf{Do}\ \mathit{classify\_user}(\texttt{Ssn},\texttt{Name})$

If the agent needs to classify a user, it may also request the credit *agent's provide: address service to get further information about the customer.*

$\mathbf{r5:}$ $\mathbf{O}$ $\mathit{update\_highProfile}(\mathtt{Ssn}, \mathtt{Name}, \mathtt{RiskProfile}) \leftarrow$
$\mathbf{Do}$ $\mathit{process\_request}(\mathtt{Msg.Id}, \mathtt{Agent})$,
$=(\mathtt{Agent}, \mathtt{credit})$,
$\mathbf{in}(\mathtt{Ssn}, \mathrm{msgbox}: \mathit{getVar}(\mathtt{Msg.Id}, \texttt{"}\mathit{Ssn}\texttt{"}))$,
$\mathbf{in}(\mathtt{Name}, \mathrm{msgbox}: \mathit{getVar}(\mathtt{Msg.Id}, \texttt{"}\mathit{Name}\texttt{"}))$,
$\mathbf{in}(\mathtt{RiskProfile}, \mathrm{msgbox}: \mathit{getVar}(\mathtt{Msg.Id}, \texttt{"}\mathit{RiskProfile}\texttt{"}))$,
$\mathbf{in}(\mathtt{spender}(\mathtt{high}), \mathrm{profiling}: \mathit{classifyUser}(\mathtt{Ssn}))$

If a high spending customer overdrafts a check, update this user's information to reflect the fact that she has a risky profile. Note that the set of customers who overdraft a check is determined from a message from the credit *agent.*

r6: $\mathbf{O}\ \mathit{inform\_sale\_notifier}(\mathtt{Ssn},\mathtt{Name},\mathtt{Profile}) \leftarrow$
$\mathbf{Do}\ \mathit{process\_request}(\mathtt{Msg.Id},\mathtt{Agent}),$
$\mathbf{in}(\mathtt{Ssn}, \mathrm{msgbox}: \mathit{getVar}(\mathtt{Msg.Id}, \texttt{"}\mathit{Ssn}\texttt{"})),$
$\mathbf{in}(\mathtt{Name}, \mathrm{msgbox}: \mathit{getVar}(\mathtt{Msg.Id}, \texttt{"}\mathit{Name}\texttt{"})),$
$\mathbf{in}(\mathtt{Profile}, \mathrm{msgbox}: \mathit{getVar}(\mathtt{Msg.Id}, \texttt{"}\mathit{Profile}\texttt{"})),$
$\mathbf{Do}\ \mathit{update\_highProfile}(\mathtt{Ssn},\mathtt{Name},\mathtt{Profile}),$
$=(\mathtt{Profile}, \mathtt{riskProfile})$

If a high spending customer acquires a `riskProfile`, *notify the* saleNotification *agent. After this notification, the* saleNotification *agent may decide to defer the production of expensive brochures for this customer until his/her credit is in better standing.*

r7: $\mathbf{W}$ $\mathit{inform\_sale\_notifier}(\mathtt{Ssn}, \mathtt{Name}, \mathtt{Profile}) \leftarrow$
$\mathbf{Do}$ $\mathit{process\_request}(\mathtt{Msg.Id}, \mathtt{Agent})$,
$\mathbf{Do}$ $\mathit{process\_request}((\mathtt{Msg.Id1}, \mathtt{Agent1})$,
$=$(`Agent`, saleNotification),
in$(\mathtt{Ssn}, \mathrm{msgbox}: \mathit{getVar}(\mathtt{Msg.Id}, \texttt{"}\textit{Ssn}\texttt{"}))$,
in$(\mathtt{Name}, \mathrm{msgbox}: \mathit{getVar}(\mathtt{Msg.Id}, \texttt{"}\textit{Name}\texttt{"}))$,
in$(\mathtt{Profile}, \mathrm{msgbox}: \mathit{getVar}(\mathtt{Msg.Id}, \texttt{"}\textit{Profile}\texttt{"}))$,
in$(\mathtt{offseason}, \mathrm{msgbox}: \mathit{getVar}(\mathtt{Msg.Id1}, \texttt{"}\textit{offseason}\texttt{"}))$,
$\mathbf{Do}$ $\mathit{update\_highProfile}(\mathtt{Ssn}, \mathtt{Name}, \mathtt{Profile})$

If we are in the offseason (e.g., if it is summer and we are dealing with winter coats), then there is no need to call $\mathit{inform\_sale\_notifier}$ *whenever we invoke* $\mathit{update\_highProfile}$.

r8: $\mathbf{O}\ \mathit{process\_request}(\mathtt{Msg.Id}, \mathtt{Agent}) \leftarrow$
in$(\mathtt{Msg}, \mathrm{msgbox}: \mathit{getAllMsgs}()),$
$=(\mathtt{Agent}, \mathit{Msg.Source})$

This rule says that the agent is obliged to process all requests in its message box. This does not mean that it will respond positively to the request. Note that the profiling *agent can receive messages from different agents, such as the* credit *agent and the* saleNotification *agent.*

r9: $\mathbf{O}\ \mathit{delete\_msg}(\mathtt{Msg.Id}) \leftarrow$
$\mathbf{Do}\ \mathit{process\_request}(\mathtt{Msg.Id}, \mathtt{Agent})$

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 4.1 on page 115.

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 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 5.2), 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 5.1 (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. $\{$ $DoSet := ComputeSem(\mathit{Curr}, \mathcal{IC}, \mathcal{AC}, \mathcal{AB}, \textbf{conc}, \mathit{Newmsg})$;

($\star$ *find a set of actions to execute based on messages received* $\star$)

3. *Curr* $:=$ *result of executing the single action* **conc**$(DoSet)$; $\}$

end.

Example 5.4 (Clock-Driven Actions)
Let us consider a very simple agent in the STORE scenario, that sends a message at 10 pm every day to the sales manager of the store, summarizing the total value of merchandise sold on that day. This behavior may be modeled in terms of a clock *agent, and a* salesMonitor *agent (which will interact with other agents). At 10 pm every day, the* clock *agent sends a message to the* salesMonitor *agent saying it is 10 pm, together with the current date. This is an* update *to the state of* salesMonitor *which causes the* salesMonitor *to execute an action called* $sendsales(\mathtt{Today})$*. This causes the desired message to be sent to the Sales Manager.*

Example 5.5 (CHAIN Example Revisited)
Consider the agent program for the `supplier` *agent given in Example 5.1 on page 158.*

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 5.1 on page 158, 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 5.6 (CFIT Example Revisited)
Consider the agent program for the autoPilot *agent given in Example 5.2 on page 168.*

1. *We alternate between the maintain_course and adjust_course actions (while obeying rules r1 and r2) until control is returned to the pilot (rule r3).*

2. *Suppose our* autoPilot *agent discovers that by following its current course, it may fly into a mountain. Here, the adjust_course action will be invoked. By rule r4, the agent may now call compute_currentLocation to get a better idea of the plane's position.*

3. *The current estimate for the plane's position will now be used by the* terrain *agent to determine which areas are No_go (see rule r5). Furthermore, rule r5 indicates that a flight plan (*`Flight_route`*) which avoids these No_go areas should be generated.*

4. *Finally, our agent will invoke execute_flight_plan (by rule r6) to actually maneuver the plane out of harm's way by using* `Flight_route`.

Example 5.7 (STORE Example Revisited)

Consider the agent program for the `profiling` *agent given in Example 5.3 on page 176.*

1. *We begin by classifying the spending profile for a user. To do this, we may consult the* `credit` *agent by using its services such as provide: information(credit) (by rule r3) or provide: address (by rule r4).*

2. *If this classification indicates that our user has a low or medium spending profile, we do not worry about updating the user's profile (see rules r1 and r2).*

3. *Suppose our user is a high spender who overdrafts a check. When this occurs, the* credit *agent will send a message to our* profiling *agent. By rule r5, the* profiling *agent will now classify our user with a* `riskProfile`.

4. *Once a user obtains this profile, the* profiling *agent may send (by rule r6) a message to the* saleNotification *agent so the latter agent can take whatever actions it deems necessary. Note however that this message will not be sent if the* profiling *agent knows that the* saleNotification *agent is not receiving messages. By rule r7, this situation may occur during the off-season.*

Example 5.8 (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* $\mathbf{Do}\,wake$ *action to be executed, which in turn triggers other actions. These include:*

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

- *The* $\mathfrak{gps}$ *agent executes its* $getAllMsgs$ *and retrieves the message sent by the* $\mathfrak{autoPilot}$ *agent.*
- *The decision program of the* $\mathfrak{gps}$ *agent executes this request and also executes the action* $sendMessage(\mathtt{gps}, \mathtt{autoPilot}, <answer>)$ *where* $<answer>$ *is the answer to the request made by the* $\mathfrak{autoPilot}$ *agent.*
- *The* $\mathfrak{autoPilot}$ *agent executes the* $getAllMsgs$ *action and retrieves the message sent by the* $\mathfrak{gps}$ *agent.*
- *The decision program of the* $\mathfrak{autoPilot}$ *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*

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

  where $<request>$ *requests the* $\mathfrak{terrain}$ *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* $sendMessage(\mathtt{terrain}, \mathtt{autoPilot}, \mathtt{Ans})$ *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.*

5.2 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 $\mathbf{Do}\,\alpha$ *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 5.5 (Status Set)
A status set *is any set S of ground action status atoms over $\mathcal{S}$. For any operator $Op \in \{\mathbf{P}, \mathbf{Do}, \mathbf{F}, \mathbf{O}, \mathbf{W}\}$, we denote by $Op(S)$ the set* $Op(S) = \{\alpha \mid Op(\alpha) \in S\}$.

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

Definition 5.6 (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 $O_S \models \exists^* Pre(\alpha)$, 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 O_S.*

A status set S is called action consistent, *if $S, O_S \models \mathcal{AC}$ 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 5.7 (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 5.1

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 P.

Definition 5.8 (Operator $\mathbf{App}_{\mathcal{P},O_S}(S)$)
Suppose $\mathcal{P}$ is an agent program, and O_S is an agent state. Then, $\mathbf{App}_{\mathcal{P},O_S}(S)$ 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 O_S, and*
3. *every code call $\chi \in \neg.B^-_{cc}(r)$ does not succeed in 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 O_S.*

5.3 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 5.9 (Feasible Status Set)
Let $\mathcal{P}$ be an agent program and let $O_{\mathcal{S}}$ be an agent state. Then, a status set S is a feasible status set *for $\mathcal{P}$ on $O_{\mathcal{S}}$, if the following conditions hold:*

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

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

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

Example 5.9 (CHAIN example revisited)
Let us consider a simple agent program containing just the rule (r4) of Example 5.1, together with rule (r8) and (r9) that manage the message box.

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

Suppose the current state of the agent $\mathtt{supplier}$ *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* $\mathbf{P}\ \mathit{order\_part}(\mathtt{p50}, 2000)$ *but will not contain* $\mathbf{Do}\ \mathit{order\_part}(\mathtt{p50}, 2000)$. *However, other feasible status sets will contain both* $\mathbf{P}\ \mathit{order\_part}(\mathtt{p50}, 2000)$ *and* $\mathbf{Do}\ \mathit{order\_part}(\mathtt{p50}, 2000)$.

Example 5.10 (CHAIN example revisited)

On the other hand, suppose our agent program contains rules (r3), (r8) and (r9) of Example 5.1 on page 158, 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*

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

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 5.11
The following (artificial) example shows that some agent programs may have no feasible status sets at all.

$$\begin{aligned} \mathbf{P}\alpha &\leftarrow \\ \mathbf{F}\alpha &\leftarrow \end{aligned}$$

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 5.2 (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 5.12 (Expanded CHAIN Example)
Let us return to the example feasible status sets we saw in Example 5.9 on page 206. In this case, this agent program had multiple feasible status sets. Some feasible status sets will contain $\mathbf{P}\ \mathit{order\_part}(\mathrm{p50}, 2000)$ *but will not contain* $\mathbf{Do}\ \mathit{order\_part}(\mathrm{p50}, 2000)$*. However, other feasible status sets will contain both* $\mathbf{P}\ \mathit{order\_part}(\mathrm{p50}, 2000)$ *and* $\mathbf{Do}\ \mathit{order\_part}(\mathrm{p50}, 2000)$*. It is immediately apparent that we do not want* both *action status atoms* $\mathbf{P}\ \mathit{order\_part}(\mathrm{p50}, 2000)$ *and* $\mathbf{Do}\ \mathit{order\_part}(\mathrm{p50}, 2000)$ *to be present in feasible status sets: there is no good reason to in fact perform the action* $\mathit{order\_part}(\mathrm{p50}, 2000)$ *(the agent program in question does not mandate that* $\mathbf{Do}\ \mathit{order\_part}(\mathrm{p50}, 2000)$ *be true).*

5.4 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 5.10 (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 5.13 (Expanded CHAIN Example Continued)
Returning to Example 5.12 on page 210, it is immediately apparent that all feasible status sets that contain both $\mathbf{P}$ $\mathit{order\_part}(\mathtt{P},\mathtt{N})$ *and* $\mathbf{Do}$ $\mathit{order\_part}(\mathtt{P},\mathtt{N})$ *are not rational, while those that only contain* $\mathbf{P}$ $\mathit{order\_part}(\mathtt{P},\mathtt{N})$ *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 $O_\mathcal{S}$ an operator $\mathbf{T}_{\mathcal{P},O_\mathcal{S}}$ that maps a status set S to another status set.

Definition 5.11 ($\mathbf{T}_{\mathcal{P},O_\mathcal{S}}$ Operator)

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

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

Lemma 5.1
Let $\mathcal{P}$ be an agent program, let $O_{\mathcal{S}}$ 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 $\mathbf{T}_{\mathcal{P},O_{\mathcal{S}}}$, i.e., $\mathbf{T}_{\mathcal{P},O_{\mathcal{S}}}(S) \subseteq S$.

Theorem 5.1
Let $\mathcal{P}$ be a positive agent program, and let $O_{\mathcal{S}}$ be an agent state. Then, S is a rational status set of $\mathcal{P}$ on $O_{\mathcal{S}}$, if and only if $S = lfp(\mathbf{T}_{\mathcal{P},O_{\mathcal{S}}})$ and S is a feasible status set.

Corollary 1

Let $\mathcal{P}$ be a positive agent program. Then, on every agent state $O_{\mathcal{S}}$, 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 $O_{\mathcal{S}}$, then $S = S'$.

Example 5.14 (CHAIN example revisited)

Let us return to the agent program described in Example 5.12 on page 210. Let us augment this example with a new action, fax_order. Suppose we augment our agent program of Example 5.12 on page 210 with the two rules

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

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

It is now easy to see that there are two rational status sets—one of which contains the status atom $\mathbf{Do}\ \mathit{fax\_order}(\texttt{company1}, \texttt{Part\_id}, 2000)$ *and the other* $\mathbf{Do}\ \mathit{fax\_order}(\texttt{company2}, \texttt{Part\_id}, 2000)$*. Thus, the introduction of negated status atoms in rule bodies leads to this potential problem.*

As shown by Example 5.14 on page 218 Corollary 1 on page 217 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 5.3

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.

5.4.1 Reading of Rational Status Sets

We are now ready to return to the question "*Exactly how should we read the atoms* $Op(\alpha)$ *for* $Op \in \{\mathbf{P}, \mathbf{F}, \mathbf{W}, \mathbf{O}, \mathbf{Do}\}$ *appearing in a rational status set?*" In Subsection 5.1, we had promised a discussion of this issue. It appears that the reading:

$$\mathrm{Op}(\alpha) \equiv \text{“It is the case that } \alpha \text{ is } \mathrm{Op}^*\text{”}$$

where Op^* is the proper verb corresponding to operator Op (forbidden, permitted, etc), is *not* the one which is expressed inherently by rational status sets.

Rather, a status atom in a rational status set should be more appropriately interpreted as follows:

$\mathsf{Op}(\alpha) \equiv$ "It is <u>derivable</u> that α should be Op^*."

where "derivable"—without giving a precise definition here—means that $\mathsf{Op}(\alpha)$ is obtained from the rules of the agent program and the deontic axioms, under reasonable assumptions about the status of actions; the groundedness property of the rational status set ensures that the adopted assumptions are as conservative as possible.

Furthermore, a literal $\neg\mathsf{Op}(\alpha)$ in a program should be interpreted as:

$\neg\mathsf{Op}(\alpha) \equiv$ "It is not derivable that α should be Op^*."

This brings us back to our interpretation of $\mathrm{Op}(\alpha)$. The fallacy in the above argument is the implicit equivalence assumed to hold between the statements

- "β is either forbidden or permitted" and
- $\mathbf{P}\beta \vee \mathbf{F}\beta$.

The latter statement is read "It is either derivable that β is permitted, or it is derivable that β is forbidden" which is certainly very different from the former statement.

5.5 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 5.12 (Reasonable Status Set)
Let $\mathcal{P}$ be an agent program, let $O_{\mathcal{S}}$ be an agent state, and let S be a status set.

1. *If $\mathcal{P}$ is a positive agent program, then S is a* reasonable status set *for $\mathcal{P}$ on $O_{\mathcal{S}}$, if and only if S is a rational status set for $\mathcal{P}$ on $O_{\mathcal{S}}$.*

2. *The reduct of $\mathcal{P}$ w.r.t. S and $O_{\mathcal{S}}$, denoted by $red^S(\mathcal{P}, O_{\mathcal{S}})$, is the program which is obtained from the ground instances of the rules in $\mathcal{P}$ over $O_{\mathcal{S}}$ as follows.*

   (a) *First, remove every rule r such that $B^-_{as}(r) \cap S \neq \emptyset$;*

   (b) *Remove all atoms in $B^-_{as}(r)$ from the remaining rules.*

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

Example 5.15 (CHAIN example revisited)
Let us return to the case of the agent program presented in Example 5.14 on page 218. Here we have two rational status sets, one containaing $\mathbf{Do}$ $\mathit{fax\_order}(\texttt{company1}, \texttt{p50}, \texttt{500})$*, while the other contains* $\mathbf{Do}$ $\mathit{fax\_order}(\texttt{company2}, \texttt{p50}, \texttt{500})$*.*

According to the above definition, only the rational status set that contains the status atom $\mathbf{Do}$ $\mathit{fax\_order}(\texttt{company1}, \texttt{p50}, \texttt{500})$ *is reasonable. The reason is that the first rule listed explicitly in Example 5.14 on page 218 says that if we do not infer* $\mathbf{Do}$ $\mathit{fax\_order}(\texttt{company2}, \texttt{p50}, \texttt{500})$*, then we should infer* $\mathbf{Do}$ $\mathit{fax\_order}(\texttt{company1}, \texttt{p50}, \texttt{500})$*, thus implicitly providing higher priority to the rational status set containing* $\mathbf{Do}$ $\mathit{fax\_order}(\texttt{company1}, \texttt{p50}, \texttt{500})$*,.*

A more simplistic example is presented below.

Example 5.16
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 O_S is the program

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

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

$$\mathbf{Do}\,\alpha \leftarrow \neg\mathbf{F}\alpha \tag{5.3}$$

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 5.4

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 $O_{\mathcal{S}}$.