4. Actions in IMPACT

Overview

4.1 Action Base

4.2 Execution and Concurrency

4.3 Action Constraints

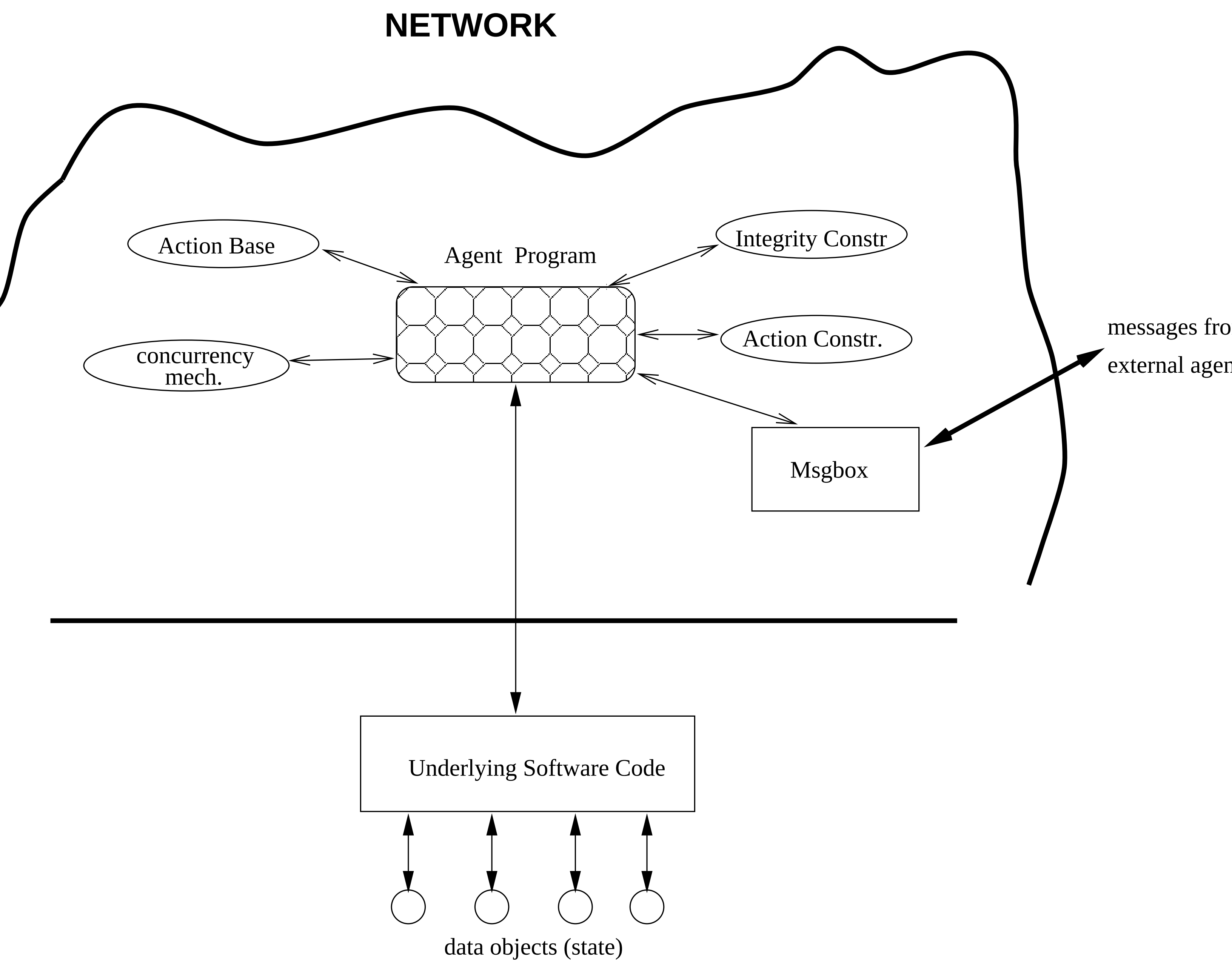

Figure 4.1: Agent Decision Architecture

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, 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.

4.1 Action Base

Definition 4.1 (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* $Pre(\alpha)$ *(*$Pre(\alpha)$ *must be* safe modulo the variables $X_1, \ldots, X_n$*);*

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

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

An action atom *is a formula* $\alpha(t_1, \ldots, t_n)$*, where* t_i *is a term, i.e., an object or a variable, of type* 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 4.1 (CHAIN Revisited)

Suppose the $\mathtt{supplier}$ *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:

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

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

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

Add:

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

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

This action updates the two ACCESS databases for uncommitted *and* committed *stock. The* $\mathtt{supplier}$ *agent should first make sure that the amount requested is available by consulting the* uncommitted *stock database. Then, the* $\mathtt{supplier}$ *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 4.2 (CFIT Revisited)

Suppose the $\mathtt{autoPilot}$ *agent in the CFIT example has the following action for computing the current location of the plane:*

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

Schema: $(\mathtt{SatelliteReport})$

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

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

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

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

Example 4.3 (STORE Example Revisited)

The profiling *agent might have the following action:*

Name: $\mathit{update\_highProfile}(\mathtt{Ssn}, \mathtt{Name}, \mathtt{Profile})$

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

Pre: $\mathbf{in}(\mathtt{spender}(\mathtt{high}), \mathrm{profiling}:\mathit{classifyUser}(\mathtt{Ssn}))$

Del:

$\mathbf{in}(\langle \mathtt{Ssn}, \mathtt{Name}, \mathtt{OldProfile} \rangle, \mathrm{profiling}:\mathit{all}('\mathit{highProfile}'))$

Add:

$\mathbf{in}(\langle \mathtt{Ssn}, \mathtt{Name}, \mathtt{Profile} \rangle, \mathrm{profiling}:\mathit{all}('\mathit{highProfile}'))$

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

Example 4.4 (Java Agents)

In today's world, the word "agent" is often considered (in certain non-AI communities) to be synonymous with Java applets. What is unique about an applet is that it is mobile. *A Java applet hosted on machine H can "move" across the network to a target machine T, and execute its operations there. The actions taken by a Java agent* $\mathfrak{agentID}$, *may be captured within our framework as follows.*

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

Pre:

$\textbf{in}(\mathtt{Host},\mathtt{java}{:}\mathit{location}(\mathtt{Agent\text{-}id}))$ &

$\textbf{in}(\mathtt{ok},\mathtt{security}{:}\mathit{authorize}(\mathtt{Agent\text{-}id},\mathtt{Op},\mathtt{Target},\mathtt{ArgumentList}))$.

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

Add/Del: *This consists of whatever insertions and deletions must be done to data in the* `Host`*'s workspace.*

Definition 4.2 (Action Base)

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

4.2 Execution and Concurrency of Actions

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

Definition 4.3 ((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 $O_{\mathcal{S}}$.*

Definition 4.4 (Action Execution)
Suppose $(\alpha(\vec{X}),\theta,\gamma)$ *is a feasible execution triple in state* O_S. *Then the* result *of executing* $\alpha(\vec{X})$ *w.r.t.* (θ,γ) *is given by the state*

$$\mathit{apply}((\alpha(\vec{X}),\theta,\gamma),O_S) = \mathbf{ins}(O_{add},\mathbf{del}(O_{del},O_S)),$$

where $O_{add} = O\_Sol(Add(\alpha(\vec{X})\theta)\gamma)$ *and* $O_{del} = O\_Sol(Del(\alpha(\vec{X})\theta)\gamma)$; *i.e., the state that results if first all objects in solutions of call conditions from* $Del(\alpha(\vec{X})\theta)\gamma$ *on* O_S *are removed, and then all objects in solutions of call conditions from* $Add(\alpha(\vec{X})\theta))\gamma$ *on* O_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 4.5 (Concurrency Notion)
A notion of concurrency *is a function*, **conc**, *that takes as input, an object state, $O_{\mathcal{S}}$, and a set of execution triples AS, and returns as output, a single new execution triple such that:*

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

Definition 4.6 (Weakly Concurrent Execution)
Suppose AS is a set of feasible execution triples in the agent state $O_{\mathcal{S}}$. The weakly concurrent execution of AS *in* $O_{\mathcal{S}}$*, is defined to be the agent state*

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

where

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

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

For any set A of actions, the execution of A on 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 apply(A, O_S) denotes the resulting state.

Definition 4.7 (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* O_S*. Then,* AS *is said to be* S-concurrently executable *in state* O_S*, if, by definition, there exists a permutation* π *of* AS *and a sequence of states* $O_S^0, \ldots, O_S^n$ *such that:*

- $O_S^0 = O_S$ *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* O_S^{i-1}*, and* $O_S^i = apply((\vec{X}_{\pi(i)}, \theta_{\pi(i)}, \gamma_{\pi(i)}), O_S^{i-1})$.

In this case, AS *is said to be* π-executable, *and* 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* O_S*, if the set* $\{(\alpha(\vec{X}), \theta, \gamma) \mid \alpha(\vec{t}) \in ACS, \alpha(\vec{X})\theta = \alpha(\vec{t})\theta$ *ground,* $(\theta, \gamma) \in \Theta\Gamma(\alpha(\vec{X}))\}$ *is* S*-concurrently executable on* O_S.

Definition 4.8 (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 4.5 (CHAIN Revisited)

Consider the following set of action executions:

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

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

$\mathbf{in}(\texttt{X}, \mathrm{supplier}: \mathit{select}('\mathit{uncommitted}', \texttt{id}, =, \texttt{widget5})),$

$\mathbf{in}(\texttt{Y}, \mathrm{supplier}: \mathit{select}('\mathit{committed}', \texttt{id}, =, \texttt{widget5})),$

$\mathbf{in}(\texttt{X}, \mathrm{supplier}: \mathit{select}('\mathit{uncommitted}', \texttt{id}, =, \texttt{widget10})),$

$\mathbf{in}(\texttt{Y}, \mathrm{supplier}: \mathit{select}('\mathit{committed}', \texttt{id}, =, \texttt{widget10})).$

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

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

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

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

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

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

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

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 4.6 (CHAIN example revisited)
Let us return to the situation raised in Example 4.5 on page 137. The following set of action executions are F-concurrently executable:

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

Further assume that the uncommitted stock database contains the same tuples as in Example 4.5 on page 137. 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:* $\langle\texttt{widget5}, 750\rangle$, $\langle\texttt{widget10}, 400\rangle$ *and* $\langle\texttt{widget15}, 1000\rangle$.

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 $\mathfrak{a}$, 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 of concurrency used by agent $\mathfrak{a}$."

4.3 Action Constraints

Definition 4.9 (Action Constraint)

An action constraint AC has the syntactic form:

$$\{\alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)\} \hookleftarrow \chi \tag{4.1}$$

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

Example 4.7 (CHAIN Example Revisited)
The following are some action constraints for the supplier *agent of CHAIN example:*

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

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

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 4.8 (CFIT Example Revisited)
The following is an action constraint for the $\mathrm{autoPilot}$ *agent:*

$$\begin{aligned}\{&\mathit{compute\_currentLocation}(\texttt{Report}),\\ &\mathit{adjust\_course}(\texttt{No\_go},\texttt{FlightRoute},\texttt{CurrentLocation})\} \hookleftarrow\end{aligned}$$

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:*

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

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

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

Example 4.9 (STORE Example Revisited)
The following are some action constraints for the profiling *agent in the STORE example:*

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

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

The first action states that if the user is classified as a high spender, then the $\mathtt{profiling}$ *agent cannot execute* $update\_highProfile$ *and* $update\_lowProfile$ *concurrently. In contrast, the second action constraint states that the* $\mathtt{profiling}$ *agent cannot classify a user profile if it is currently updating the profile of that user.*

Definition 4.10 (Action Constraint Satisfaction)
A set S *of ground actions satisfies an action constraint* AC *as in (4.1) on a state* $O_{\mathcal{S}}$*, denoted* $S, O_{\mathcal{S}} \models AC$*, if there is no legal assignment* θ *of objects in* $O_{\mathcal{S}}$ *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* $O_{\mathcal{S}}$*, denoted* $S, O_{\mathcal{S}} \models \mathcal{AC}$*, if* $S, O_{\mathcal{S}} \models AC$ *for every* $AC \in \mathcal{AC}$*.*

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

Example 4.10 (STORE Example Revisited)
Suppose our state consists of the three uncommitted stock database tuples given in Example 4.5 on page 137 and let $\mathcal{AC}$ be the first action constraint given in Example 4.7 on page 142. Then if S_1 consists of

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

and S_2 consists of

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

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 $(\texttt{Amount1} + \texttt{Amount2}) = (750 + 500) \geq 1000$.