Multi Agenten Systeme

VU SS 00, TU Wien

Teil 1 (Kapitel 1–4) basiert auf

> **Multi-Agent Systems** (Gerhard Weiss), MIT Press, June 1999.

Es werden allgemeine Techniken und Methoden dargestellt (BDI-, Layered-, Logic based Architekturen, Decision Making, Kommunikation/Interaktion, Kontrakt Netze, Coalition Formation).

Teil 2 (Kapitel 5–9) basiert auf

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

Hier wird ein spezifischer Ansatz vorgestellt, der formale Methoden aus dem logischen Programmieren benutzt, aber nicht auf PROLOG aufsetzt (Code Call Mechanismus, Aktionen, Agenten Zyklus, Status Menge, Semantiken, Erweiterungen um Beliefs, Implementierbarkeit).

Übersicht

1. Einführung, Terminologie
2. 4 Grundlegende Architekturen
3. Distributed Decision Making
4. Contract Nets, Coalition Formation
5. *IMPACT* Architecture
6. Legacy Data and Code Calls
7. Actions and Agent Programs
8. Regular Agents
9. Meta Agent Programs

6. The Code Call Mechanism

Overview

Timetable:

- Chapter 2 needs 1 lecture.

6 Legacy Data

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

CHAIN: supplier agents on top of an existing commercial relational DBMS system.

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

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

6.1 Software Code Abstractions

Definition 6.1 (Software Code $\mathcal{S} = (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$)
We may characterize the code on top of which an agent is built as a triple $\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ *where:*

1. $\mathcal{T}_{\mathcal{S}}$ *is the set of all data types managed by* $\mathcal{S}$*,*
2. $\mathcal{F}_{\mathcal{S}}$ *is a set of predefined functions which makes access to the data objects managed by the agent available to external processes, and*
3. $\mathcal{C}_{\mathcal{S}}$ *is a set of type composition operations. A type composition operator is a partial* n*-ary function* c *which takes as input types* $\tau_1, \ldots, \tau_n$ *and yields as a result a type* $c(\tau_1, \ldots, \tau_n)$*. As* c *is a partial function,* c *may only be defined for certain arguments* $\tau_1, \ldots, \tau_n$*, i.e.,* c *is not necessarily applicable on arbitrary types.*

Intuitively:

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

Given a software package $\mathcal{S}$, we use the notation $\mathcal{T}^{\star}_{\mathcal{S}}$ to denote the *closure* of $\mathcal{T}_{\mathcal{S}}$ under the operations in $\mathcal{C}_{\mathcal{S}}$. In order to formally define this notion, we introduce the following definition.

Definition 6.2 ($\mathcal{C}_{\mathcal{S}}(\mathcal{T})$ and $\mathcal{T}^{\star}_{\mathcal{S}}$)

a) *Given a set $\mathcal{T}$ of types, we define*

$$\mathcal{C}_{\mathcal{S}}(\mathcal{T}) =_{def} \mathcal{T} \cup \{\tau : \text{ there exists an } n\text{-ary composition operator } c \in \mathcal{C}_{\mathcal{S}} \text{ and types } \tau_1, \ldots, \tau_n \in \mathcal{T} \text{ such that } c(\tau_1, \ldots, \tau_n) = \tau\}.$$

b) *We define $\mathcal{T}^{\star}_{\mathcal{S}}$ as follows:*

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

CHAIN Revisited

$\mathcal{T}_{\mathcal{S}} =_{def} \{\texttt{Integer}, \texttt{Location}, \texttt{String}, \texttt{Date}, \texttt{OrderLog}, \texttt{Stock}\}$
`OrderLog` is a relation having the schema

$$(client/\texttt{string},\ amount/\texttt{Integer},\ part\_id/\texttt{String},\ method/\texttt{String},$$
$$src/\texttt{Location},\ dest/\texttt{Location},\ pickup\_st/\texttt{date},\ pickup\_et/\texttt{date}),$$

while `Stock` is a relation having the schema $(amount/\texttt{Integer},\ part\_id/\texttt{String})$.
`Location` is an enumerated type containing city names.

In addition, $\mathcal{F}_S$ might consist of the functions:

- ***monitorStock***$(Amount/$**`Integer`**$, Part\_id/$**`String`**$)$ of type `String`.
  This function returns either `amount_available` or `amount_not_available`.
- ***shipFreight***$(Amount/$**`Integer`**$, Part\_id/$**`String`**$, method/$**`String`**$,$
  $Src/$**`Location`**$, Dest/$**`Location`**$)$.
  This function, when executed, updates the order log and logs information about the order, together with information on (i) the earliest time the order will be ready for shipping, and (ii) the latest time by which the order must be picked up by the shipping vendor.

  Notice that this does *not* mean that the shipment will in fact be picked up by the `airplane` agent at that time.
- ***updateStock***$(Amount/$**`Integer`**$, Part\_id/$**`String`**$)$.
  This function, when executed, updates the inventory of the Supplier.

CFIT Revisited

$\mathcal{T}_{\mathcal{S}} =_{def} \{\texttt{Map}, \texttt{Path}, \texttt{Plan}, \texttt{SatelliteReport}\}$.
Special class of maps called *DTED Digital Terrain Elevation Data* that specify the elevations of different regions of the world.

Suppose the **autoPilot** agent's associated set of functions $\mathcal{F}_{\mathcal{S}}$ contains:

- ***createFlightPlan***$(Location/\texttt{Map}, Flight\_route/\texttt{Path}, Nogo/\texttt{Map})$ of type `Plan`.

Moreover, the $\mathcal{F}_{\mathcal{S}}$ of the **gps** might contain the following function:

- ***mergeGPSData***$(Data1/\texttt{SatelliteReport}, Data2/\texttt{SatelliteReport})$ of type `SatelliteReport`.

State of an Agent

Definition 6.3 (State of an Agent)

At any given point t in time, the state of an agent *will refer to a set* $O_S(t)$ *of objects from the types* $\mathcal{T}_S$*, managed by its internal software code.*

An agent may change its state by taking an action—either triggered internally, or by processing a message received from another agent.

We will assume that except for appending messages to an agent **a**'s mailbox, another agent **b** cannot directly change **a**'s state. However, it might do so indirectly by shipping the other agent a message issuing a change request.

6.2 Code Calls

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

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

Definition 6.4 (Code Call $\mathcal{S}:f(\mathtt{d}_1,\ldots,\mathtt{d}_n)$)
Suppose $\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ is some software code and $f \in \mathcal{F}_{\mathcal{S}}$ is a predefined function with n arguments, and $\mathtt{d}_1,\ldots,\mathtt{d}_n$ are objects or variables such that each $\mathtt{d}_i$ respects the type requirements of the i'th argument of f. Then,

$$\mathcal{S}:f(\mathtt{d}_1,\ldots,\mathtt{d}_n)$$

is a code call. *A code call is* ground *if all the $\mathtt{d}_i$'s are objects. We often switch between the software package $\mathcal{S}$ and the agent providing it. Therefore instead of writing $\mathcal{S}:f(\mathtt{d}_1,\ldots,\mathtt{d}_n)$ where $\mathcal{S}$ is provided by agent $\mathfrak{a}$, we also write $\mathfrak{a}:f(\mathtt{d}_1,\ldots,\mathtt{d}_n)$.*

$\mathcal{S}:f(\mathtt{d}_1,\ldots,\mathtt{d}_n)$ may be read as: *execute function f as defined in package $\mathcal{S}$ on the arguments* $\mathtt{d}_1,\ldots,\mathtt{d}_n$.

Comment 1 (Assumption on the Output Signature) *We will assume that the* ***output signature*** *of any code call is a* ***set****. There is no loss of generality in making this assumption—if a function does not return a set, but rather returns an atomic value, then that value can be coerced into a set anyway—by treating the value as shorthand for the singleton set containing just the value.*

1. supplier : *monitorStock*(3, part_008).
   Observe that the result of this call is either the singleton set
   { amount_available }, or the set { amount_not_available }.

2. create a pickup schedule for shipping 3 pieces of part_008 from location X to paris by truck. Notice that until a value is specified for X, this code call cannot be executed.

3. GPS : *mergeGPSData*(S1, S2) is a code call which merges two pieces, S1 and S2, of satellite data, but the values of the two pieces are not stated.

Variables

$\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ of software code. Given any type $\tau \in \mathcal{T}_{\mathcal{S}}$ (wrt. software code $\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$) we will assume that there is a set *root*(τ) of "root" variable symbols ranging over τ. Such "root" variables will be used in the construction of code calls.

Suppose τ is a complex record type having fields $\mathtt{f}_1, \ldots, \mathtt{f}_\mathtt{n}$.

- For every variable of type τ, we require that $\mathtt{X.f_i}$ be a variable of type τ_i where τ_i is the type of field $\mathtt{f_i}$.
- If $\mathtt{f_i}$ itself has a sub-field g of type γ, then $\mathtt{X.f_i.g}$ is a variable of type γ, and so on.
  These are called *path variables* .
- For any path variable `Y` of the form `X.path`, where `X` is a root variable, we refer to `X` as the root of `Y`, denoted by *root*$(\mathtt{Y})$.

Example 6.1 (CFIT Revisited)

Let `X` *be a (root) variable of type* `SatelliteReport` *denoting the current location of an airplane. Then* `X.2dloc`, `X.2dloc.x`, `X.2dloc.y`, `X.height`, *and* `X.dist` *are* *path variables*. *For each of the path variables* `Y`, *root*(`Y`) = `X`. *Here,* `X.2dloc.x`, `X.2dloc.y`, *and* `X.height` *are of type* `Integer`, `X.2dloc`*'s type is a record of two* `Integer` *s, and* `X.dist` *is of type* `NonNegative`.

Definition 6.5 (Variable Assignment)
An assignment of objects to variables *is a set of equations of the form* $\mathtt{V}_1 := \mathtt{o}_1, \ldots, \mathtt{V}_k := \mathtt{o}_k$ *where the* $\mathtt{V}_i$*'s are variables (root or path) and the* $\mathtt{o}_i$*'s are objects—such an assignment is* legal, *if the types of objects and corresponding variables match.*

Example 6.2 (CFIT Revisited)
A legal assignment may be

$$(\mathtt{X.height} := 50, \mathtt{X.sat\_id} := \mathtt{iridium\_17}, \mathtt{X.dist} := 25, \mathtt{X.2dloc.x} := 3, \mathtt{X.2dloc.y} := -4).$$

If the record is ordered as shown here, then we may abbreviate this assignment as (50, `iridium_17`*, 25,* $\langle 3, -4 \rangle$*). Note however that*

$$(\mathtt{X.height} := 50, \mathtt{X.sat\_id} := \mathtt{iridium\_17}, \mathtt{X.dist} := -25, \mathtt{X.2dloc.x} := 3, \mathtt{X.2dloc.y} := -4)$$

would be illegal, because -25 is not a valid object for `X.dist`*'s type* `NonNegative`.

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

Definition 6.6 (Code Call Atom)
If cc is a code call, and X *is either a variable symbol, or an object of the output type of cc, then*

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

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

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

What effects does this have on the **state** of an agent?
It is an infinite set of ground code call atoms!

1. **in(**`amount_available`, supplier : ***monitorStock*(**`3`, `part_008`**))**.
   This code call succeeds just in case the Supplier has 3 units of `part_008` on stock.

2. **not_in(**`spender(low)`, profiling : ***classifyUser*(**`U`**))**. This code call succeeds just in case user `U`, whose identity must be instantiated prior to evaluation, is *not* classified as a low spender by the profiling agent.

Definition 6.7 (Code Call Condition)
A code call condition *is defined as follows:*

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

We refer to any code call condition of form 1.-3. as an atomic *code call condition.*

1. $\chi^{(1)}$: $\mathbf{in}(\texttt{amount\_available}, \mathit{supplier} : \mathit{monitorStock}(3, \texttt{part\_008}))$.

2. $\chi^{(2)}$: $\mathbf{in}(\texttt{X}, \mathit{supplier} : \mathit{monitorStock}(3, \texttt{part\_008}))$ & $\texttt{X} = \mathit{amount\_available}$.

3.

$$
\begin{aligned}
\chi^{(3)} : \quad & \mathbf{in}(\texttt{amount\_available}, \mathit{supplier} : \mathit{monitorStock}(\texttt{U}, \texttt{part\_008})) && \& \\
& \mathbf{not\_in}(\texttt{amount\_available}, \mathit{supplier} : \mathit{monitorStock}(\texttt{U}+1, \texttt{part\_008})) && \& \\
& \mathbf{in}(\texttt{amount\_available}, \mathit{supplier} : \mathit{monitorStock}(\texttt{V}, \texttt{part\_009})) && \& \\
& \mathbf{not\_in}(\texttt{amount\_available}, \mathit{supplier} : \mathit{monitorStock}(\texttt{V}+1, \texttt{part\_009})) && \& \ \texttt{U} < \texttt{V}.
\end{aligned}
$$

4. **in(**`spender(medium)`, profiling : ***classifyUser***(`U`)**)** &
   in(`spender(high)`, profiling : ***classifyUser***(`V`)**)** & `U` = `V`.

5. **in(**`spender(medium)`, profiling : ***classifyUser***(`U`)**)** &
   not_in(`spender(high)`, profiling : ***classifyUser***(`U`)**)**.

Safety

Definition 6.8 (Safe Code Call (Condition))
A code call $S:f(\mathtt{d}_1,\ldots,\mathtt{d}_n)$ is safe *if and only if each* $\mathtt{d}_\mathtt{i}$ *is ground. A code call condition* $\chi_1 \& \ldots \& \chi_n$*, n* ≥ 1*, is* safe *if and only if there exists a permutation* π *of* $\chi_1,\ldots,\chi_n$ *such that for every* $i = 1,\ldots,n$ *the following holds:*

1. *If* $\chi_{\pi(i)}$ *is a comparison* $\mathtt{s}_1\, op\, \mathtt{s}_2$*, then*
   - *1.1 at least one of* $\mathtt{s}_1, \mathtt{s}_2$ *is a constant or a variable* $\mathtt{X}$ *such that* $root(\mathtt{X})$ *belongs to* $RV_{\pi(i)} =_{def} \{root(\mathtt{Y}) \mid \exists j < i \text{ s.t. } \mathtt{Y} \text{ occurs in } \chi_{\pi(j)}\}$*;*
   - *1.2 if* $\mathtt{s}_\mathtt{i}$ *is neither a constant nor a variable* $\mathtt{X}$ *such that* $root(\mathtt{X}) \in RV_{\pi(i)}$*, then* $\mathtt{s}_\mathtt{i}$ *is a root variable.*
2. *If* $\chi_{\pi(i)}$ *is a code call atom of the form* **in**$(\mathtt{X}_{\pi(\mathtt{i})}, \mathtt{cc}_{\pi(\mathtt{i})})$ *or* **not_in**$(\mathtt{X}_{\pi(\mathtt{i})}, \mathtt{cc}_{\pi(\mathtt{i})})$*, then the root of each variable* $\mathtt{Y}$ *occurring in* $\mathtt{cc}_{\pi(i)}$ *belongs to* $RV_\pi(i)$*, and either* $\mathtt{X}_{\pi(\mathtt{i})}$ *is a root variable, or* $root(\mathtt{X}_{\pi(\mathtt{i})})$ *is from* $RV_{\pi(i)}$*.*

Reconsider the three sample code call conditions $\chi^{(1)}$, $\chi^{(2)}$, and $\chi^{(3)}$.

- $\chi^{(1)}$ and $\chi^{(2)}$ are safe.
- $\chi^{(3)}$ is unsafe, since there is no permutation of the atomic code call conditions which allows safety requirement 2 to be met for either `U` or `V`.

Checking safety of code call conditions can be done at compile time of a program.

If χ is found to be safe, then we can reorder the constituents $\chi_1, \ldots, \chi_n$ by a permutation π such that $\chi_{\pi(1)}, \ldots, \chi_{\pi(n)}$ can be evaluated without problems.

We need an additional definition:

Definition 6.9 (Safety Modulo Variables)
Suppose χ is a code call condition, and let $\mathbf{X}$ be any set of root variables. Then, χ is said to be safe modulo $\mathbf{X}$ *if and only if for an (arbitrary) assignment θ of objects to the variables in $\mathbf{X}$, it is the case that $\chi\theta$ is safe.*

Checking safety of a code call χ modulo variables $\mathbf{X}$ can be reduced to a call to a routine that checks for safety. This may be done as follows:

1. Find a constant (denoted `c`) that does not occur in χ.
   Let $\theta =_{def} \{\mathbf{X} = \mathtt{c}\}$, i.e., every variable in $\mathbf{X}$ is set to `c`.
2. Check if $\chi\theta$ is safe.

Safety modulo variables **X** means: When these variables **X** are instantiated, the ccc can be evaluated.

Algorithm 6.1 (safe_ccc)

safe_ccc(χ: code call condition;
$\mathbf{X}$: set of root variables)

($\star$ input is a code call condition $\chi = \chi_1 \& \cdots \& \chi_n$; $\star$)
($\star$ output is a proper reordering $\star$)
($\star$ $\chi' = \chi_{\pi(1)} \& \cdots \& \chi_{\pi(n)}$ if χ is safe modulo $\mathbf{X}$; $\star$)
($\star$ otherwise, the output is `unsafe` ; $\star$)

$L := \chi_1, \ldots, \chi_n$;
$\chi :=$ **true**;
while *L is not empty* ***do***
$\{$ *select all $\chi_{i_1}, \ldots, \chi_{i_m}$ from L st. χ_{i_j} is safe modulo $\mathbf{X}$;*
  if $m = 0$ ***then return*** `unsafe` $(exit)$;
  else
  $\{$ $\chi := \chi \& \chi_{i_1} \& \cdots \& \chi_{i_m}$;
    remove $\chi_{i_1}, \ldots, \chi_{i_m}$ from L;
    $\mathbf{X} = \mathbf{X} \cup \{root(\mathtt{Y}) \mid \mathtt{Y}$ *occurs in some* $\chi_{i_1}, \ldots \chi_{i_m}\}$;
  $\}$
$\}$
return χ';

end.

Theorem 6.1 (Safety Computation)
Suppose $\chi =_{def} \chi_1 \,\&\ldots\&\, \chi_n$ *is a code call condition. Then,* χ *is safe modulo a set of root variables* $\mathbf{X}$*, if and only if* **safe_ccc**$(\chi, \mathbf{X})$ *returns a reordering* χ' *of* χ*. Moreover, for any assignment* θ *to the variables in* $\mathbf{X}$*,* $\chi'\theta$ *is a safe code call condition which can be evaluated left-to-right.*

- A straightforward implementation of **safe_ccc** runs in quadratic time, as the number of iterations is bounded by the number n of constituents χ_i of χ, and the body of the while loop can be executed in linear time.

- By using appropriate data structures, the algorithm can be implemented to run in overall linear time.
  Briefly, the method is to use cross reference lists of variable occurrences.

- safety of a code call condition χ can be checked by calling **safe_ccc**$(\chi, \emptyset)$. Thus, checking the safety of χ, combined with a reordering of its constituents for left-to-right execution can be done very efficiently.

Definition 6.10 (Code Call Solution)
Suppose χ *is a code call condition involving the variables* $\mathbf{X} =_{def} \{X_1, \dots, X_n\}$, *and suppose* $\mathcal{S} =_{def} (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S}, \mathcal{C}_\mathcal{S})$ *is some software code. A* solution of χ *w.r.t.* $\mathcal{T}_\mathcal{S}$ *in a state* $\mathcal{O}_\mathcal{S}$ *is a legal assignment of objects* $o_1, \dots, o_n$ *to the variables* $X_1, \dots, X_n$, *written as a compound equation* $\mathbf{X} := \mathbf{o}$, *such that the application of the assignment makes* χ *true in state* $\mathcal{O}_\mathcal{S}$.

We denote by

- $\mathsf{Sol}(\chi)_{\mathcal{T}_\mathcal{S},\mathcal{O}_\mathcal{S}}$ *(omitting subscripts* $\mathcal{O}_\mathcal{S}$ and $\mathcal{T}_\mathcal{S}$ *when clear from the context), the set of all solutions of the code call condition* χ *in state* $\mathcal{O}_\mathcal{S}$, *and by*
- $\mathcal{O}\_\mathsf{Sol}(\chi)_{\mathcal{T}_\mathcal{S},\mathcal{O}_\mathcal{S}}$ *(where subscripts are occasionally omitted) the set of all objects appearing in* $\mathsf{Sol}(\chi)_{\mathcal{T}_\mathcal{S},\mathcal{O}_\mathcal{S}}$

Comment 2 (Existence of ins, del and upd) *We assume that the set $\mathcal{F}_{\mathcal{S}}$ associated with a software code package $\mathcal{S}$ contains three functions described below:*

- *A function $\textbf{ins}_{\mathcal{S}}$, which takes as input a set of objects O manipulated by $\mathcal{S}$, and a state $\mathcal{O}_{\mathcal{S}}$, and returns a new state $\mathcal{O}'_{\mathcal{S}} = \textbf{ins}_{\mathcal{S}}(O, \mathcal{O}_{\mathcal{S}})$ which accomplishes the insertion of the objects in O into $\mathcal{O}_{\mathcal{S}}$, i.e., $\textbf{ins}_{\mathcal{S}}$ is an insertion routine.*
- *A function $\textbf{del}_{\mathcal{S}}$, which takes as input a set of objects O manipulated by $\mathcal{S}$ and a state $\mathcal{O}_{\mathcal{S}}$, and returns a new state $\mathcal{O}'_{\mathcal{S}} =_{def} \textbf{del}_{\mathcal{S}}(O, \mathcal{O}_{\mathcal{S}})$ which describes the deletion of the objects in O from $\mathcal{O}_{\mathcal{S}}$, i.e., $\textbf{del}_{\mathcal{S}}$ is a deletion routine.*
- *A function $\textbf{upd}_{\mathcal{S}}$ which takes as input a data object o manipulated by $\mathcal{S}$, a field f of object o, and a value v drawn from the domain of the type of field f of object o—this function changes the value of the f field of object o to v. (This function can usually be described in terms of the preceding two functions.)*

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

"*Insert, using a* FinanceRecord *insertion routine, all objects* o *such that* $\chi[\texttt{X}]$ *is true w.r.t. the current agent state when* $\texttt{X} := \texttt{o}$."

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

As a single agent program may manage multiple data types $\tau_1, \ldots, \tau_n$, each with its own insertion routine $\mathbf{ins}_{\tau_1}, \ldots, \mathbf{ins}_{\tau_n}$, respectively, it is often more convenient to associate with any agent a an insertion routine, $\mathbf{ins}_a$, that exhibits the following behavior:

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

We assume from now on that an insertion function $\mathbf{ins}_a$ and a deletion function $\mathbf{del}_a$ may be associated with any agent a in this way.

6.3 Message Box

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

The msgbox operates on objects of the form

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

1. `i/o` signifies an incoming or outgoing message respectively.
2. "src" specifies the originator
3. "dest" specifies the destination.
4. "message" is a table consisting of triples of the form ("varName","varType","value") where "varName" is the name of the variable, "varType" is the type of the variable and the "value" is the value of the variable in string format.
5. "time" denotes the time at which the message was sent.

We will assume that the agent has the following functions that are integral in managing this message box.

- ***sendMessage***(<*source_agent*>, <*dest_gent*>, <*message*>): This causes (o, "src", "dest", "message", "time") to be placed in `Msgbox`. The parameter `o` signifies an outgoing message. When a call of ***sendMessage***("src", "dest", "message") is executed, the state of `Msgbox` changes by the insertion of the above quintuple denoting the sending of a message from the source agent **`src`** to a given Destination agent **`dest`** involving the message body "message".

- ***getMessage***(<*src*>): This causes a collection of

  (i, "src", "agent", "msg", "time")

  to be read from `Msgbox`. The *i* signifies an incoming message. Note that all messages from the given source to the agent **`agent`** whose message box is being examined, are returned by this operation. "time" denotes the time at which the message was received.

- ***timedGetMessage***$(<op>,<valid>)$: This causes the collection of all quintuples tup of the form $tup =_{def} (\mathtt{i}, <src>, <agent>, <message>, \mathtt{time})$ to be read from `Msgbox`, such that the comparison `tup.time` *op valid* is true, where *op* is required to be any of the standard comparison operators $<$, $>$, $\leq$, $\geq$, or $=$.
- ***getVar***$(<mssgId>,<varName>)$: This functions searches through all the triples in the "message" to find the requested variable. First, it converts the variable from the string format given by the "value" into its corresponding data type which is given by "varType". If the requested variable is not in the message determined by the "MssgId", then an error string is returned.

Example 6.3 (STORE Revisited)
Suppose the **profiling** *agent is asked to classify a user* U *with ssn* S*. To do this, the* **profiling** *agent may need to obtain credit information for* U *from the* **credit** *agent. The following actions may ensue:*

1. *The* **profiling** *agent sends the* **credit** *agent a message requesting* S*'s credit information.*
2. *The* **credit** *agent reads this message and sends the* **profiling** *agent a reply.*
3. *The* **profiling** *agent reads this reply and uses it to generate an answer.*

1. The **profiling** agent is asked to ***classifyUser***(S). It generates a message $\mathtt{M}_1$ of a particular format, e.g., a string "ask_provideCreditInfo_S_low," which encodes the request for S's credit information, and calls ***sendMessage***(**profiling**, **credit**, $\mathtt{M}_1$).

2. The **credit** agent either periodically calls ***getMessage***(**profiling**) until $\mathtt{M}_1$ arrives, or calls it triggered by the event that M_1 has arrived. By parsing $\mathtt{M}_1$, it determines that it needs to execute ***provideCreditInfo***(S, `low`) and send the result back to **profiling**. Depending on the result of the call, **credit** assembles a message M_2 encoding the `FinanceRecord` which was returned, or an error message. Here, we are assuming that the underlying OS level message protocol does not drop or reorder messages (if it did, we would have to include $\mathtt{M}_1$ and $\mathtt{M}_1$'s *Time* in $\mathtt{M}_2$'s message). Next, the **credit** agent calls ***sendMessage***(**credit**, **profiling**, $\mathtt{M}_2$).

3. The **profiling** agent either periodically calls ***getMessage***(**credit**) until `M`$_2$ arrives, or it is triggered by the arrival of M_2 and reads the message. By parsing `M`$_2$, it can determine what errors (if any) occurred or what the resulting `finance_record` was. Finally, the **profiling** agent can use the contents of M_2 to construct the UserProfile to be returned.

6.4 Integrity Constraints

Each agent has an associated *agent state* $\mathcal{O}$, which is a set of objects (of the types that the software code underlying the agent manages).

- Not all sets of such objects are *legal*.

Definition 6.11 (Integrity Constraints $\mathcal{IC}$)
An integrity constraint *IC is an expression of the form*

$$\psi \Rightarrow \chi$$

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

1. IC_1 : $\mathbf{in}(\texttt{amount\_available}, \texttt{supplier} : \textit{monitorStock}(\texttt{U}, \texttt{part\_001}))\ \&$
   $\mathbf{in}(\texttt{amount\_available}, \texttt{supplier} : \textit{monitorStock}(\texttt{V}, \texttt{part\_002}))$
   $$\Rightarrow$$
   $\mathbf{in}(\texttt{amount\_available}, \texttt{supplier} : \textit{monitorStock}(\texttt{U} + \texttt{V}, \texttt{part\_008})).$

2. IC_3 : $\texttt{S} = 123\_45\_6789 \Rightarrow \mathbf{not\_in}(\texttt{spender}(\texttt{low}), \texttt{profiling} : \textit{classifyUser}(\texttt{S})).$

3. IC_5 : $\texttt{R.sat\_id} = \texttt{sat\_1} \Rightarrow \texttt{R.2dloc.x} \geq 0.$

Definition 6.12 (Integrity Constraint Satisfaction)
A state $\mathcal{O}_{\mathcal{S}}$ satisfies an integrity constraint IC of the form $\psi \Rightarrow \chi$, denoted $\mathcal{O}_{\mathcal{S}} \models IC$, if for every legal assignment of objects from $\mathcal{O}_{\mathcal{S}}$ to the variables in IC, either ψ is false or χ is true.

Let $\mathcal{IC}$ be a (finite) collection of integrity constraints IC, and let $\mathcal{O}_{\mathcal{S}}$ be an agent state. We say that $\mathcal{O}_{\mathcal{S}}$ satisfies $\mathcal{IC}$, denoted $\mathcal{O}_{\mathcal{S}} \models \mathcal{IC}$, if and only if $\mathcal{O}_{\mathcal{S}}$ satisfies every constraint $IC \in \mathcal{IC}$.

6.5 Service Descriptions and Code Calls

Definition 6.13 (Service Rule)
Suppose sn *is the name of a service offered by an agent. Let* $i_1, \ldots, i_k$, $mi_1, \ldots, mi_m$, *and* $o_1, \ldots, o_n$ *be the inputs, mandatory inputs, and outputs of the service* sn, *respectively. A* service rule defining sn *is an expression of the form:*

$$sn(i_1, \ldots, i_k, mi_1, \ldots, mi_m, o_1, \ldots, o_n) \quad \leftarrow \quad \chi$$

where χ *is a code call condition that is safe modulo* $mi_1, \ldots, mi_m$. *In this case,* χ *is said to be the* body *of the above rule.*

Definition 6.14 (Service Definition Program sdp)
Using the same notation as above, a service definition program *(sdp for short) associated with service* sn *is a finite set of service rules defining* sn.

- Consider a service *sn* defined through a service definition program containing r rules.
- Let the body of the i'th rule be $\chi^{(i)}$.
- Suppose an agent specifies the mandatory inputs, i.e., an agent requesting this service specifies a substitution θ that assigns objects to each of the variables $mi_1, \ldots, mi_m$. In addition, the agent may specify a substitution δ for the discretionary inputs.
- Then the service definition program treats the agent's request for service *sn* as described in algorithm **implement_service**.

Algorithm 6.2 (implement_service)

implement_service(P:sdp; μ:subs; δ:subst)

(⋆ P is a service definition program ⋆)
(⋆ μ a subst. specif. values of all mandatory inputs ⋆)
(⋆ δ a subst. specif. values of selected discret. inp. var's ⋆)
(⋆ Ans is the result of evaluating P w.r.t. inputs μ and δ ⋆)

$Ans := \emptyset$; $Q := P$;
while $Q \neq \emptyset$ ***do***
{ *select rule* $r_i \in Q$;
  $Q := Q \setminus \{r_i\}$;
  $SOL := \mathsf{Sol}((\chi)\mu\delta)$;
  (⋆ returns many substit.'s, one for each var. of sn ⋆)
  (⋆ that is not assigned an object by either of μ, δ ⋆)
  restrict SOL to output variables;
  $Ans := Ans \cup SOL$;
}
return *Ans*;

end.

Example 6.4 (STORE Revisited)

In HERMES, each sdp *for the* STORE *example can be thought of as a predicate within the mediator for one of* STORE*'s agents. A sample* sdp *is:*

goodSpender(⟨MI⟩*Category:* UserCat⟨\MI⟩
⟨O⟩*SSN:* ListOfStrings, *Class:* UserProfile⟨\O⟩)
←
in(SSN, profiling : ***listUsers(***Category***)*)** &
in(Class, profiling : ***classifyUser(***SSN***)*)** &
not_in(spender(low), general : ***makeSet(***Class***)*)**.

A HERMES invocation of this sdp *is shown in Figure 6.1. The query*

goodSpender (corporateUsers, Ssn, Class)

asks for the ssn *and* class *of all corporate users who are not low spenders. (Note that as the second parameter of the* ***not_in*** *must be a set, we use the function* general : ***makeSet(***Class***)*** *to turn Class into a singleton set.*

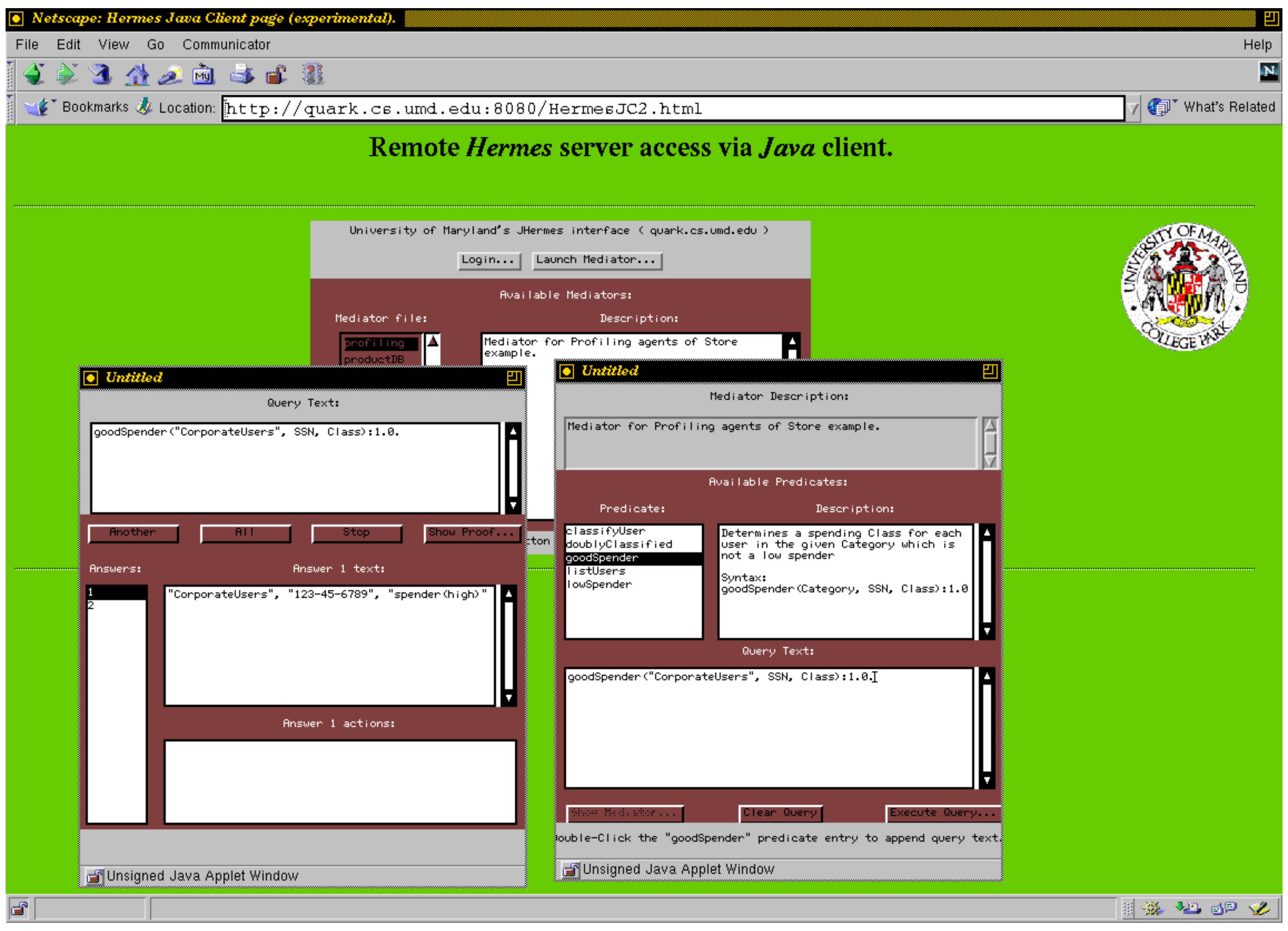

Figure 6.1: Sample query on the **profiling** agent's mediator (first result)

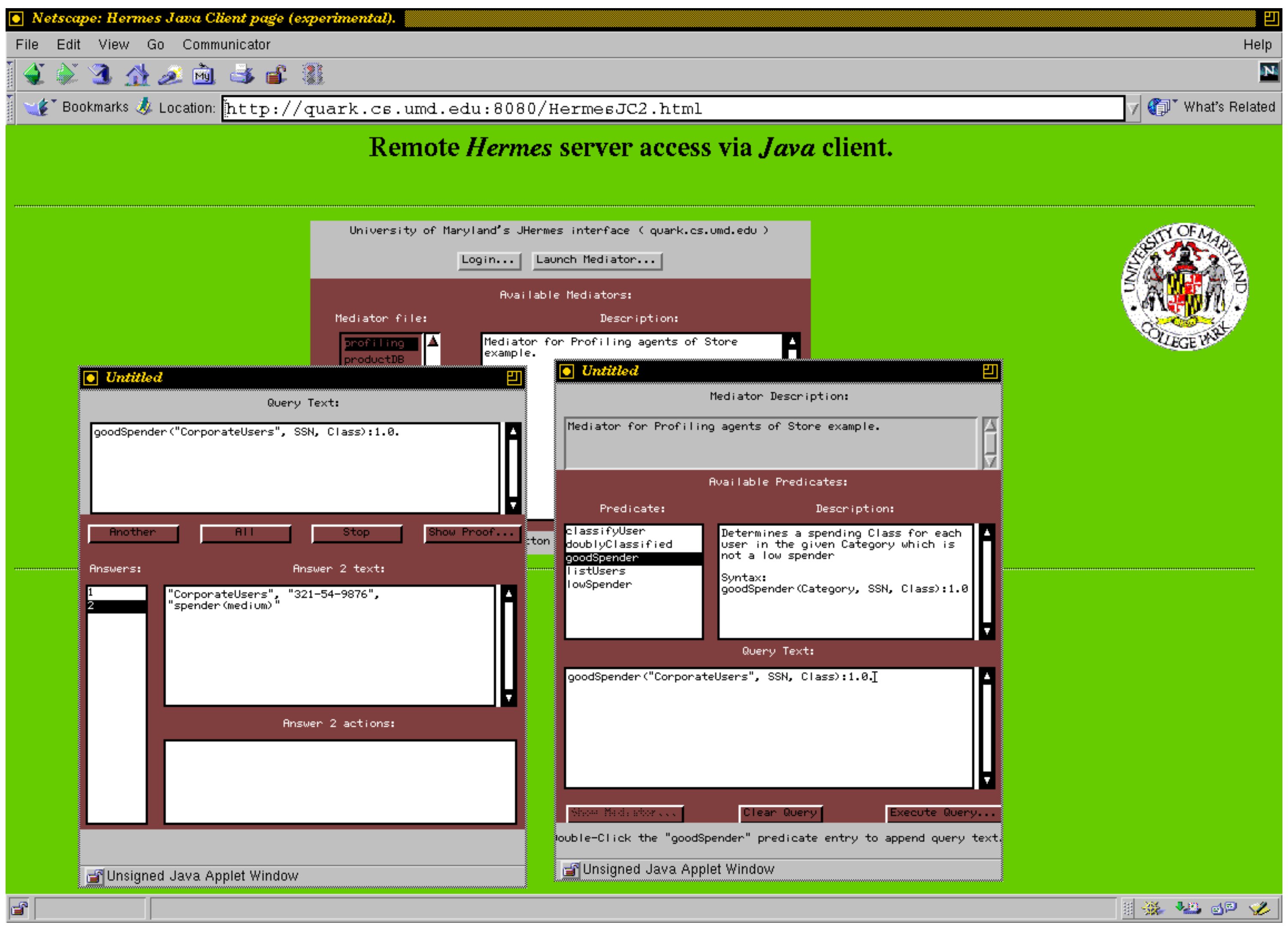

Figure 6.2: Queries on goodSpender and **profiling** Agent’s Mediator

Example 6.5 (CHAIN Revisited)

A sample query on the mediator for the **supplier** *agent of the CHAIN example is shown in Figure 6.3 on the next page. A sample sdp is:*

$$\begin{array}{l} \textit{sendViaTruck}(\langle\texttt{MI}\rangle \textit{Amount:}\,\texttt{Integer}, \textit{Part\_id:}\,\texttt{String}\langle\backslash\texttt{MI}\rangle \\ \quad \langle\texttt{MI}\rangle \textit{Src:}\,\texttt{String}, \textit{Dest:}\,\texttt{String}\langle\backslash\texttt{MI}\rangle \\ \quad \langle\texttt{O}\rangle \textit{Success:}\,\texttt{Boolean}\langle\backslash\texttt{O}\rangle) \\ \quad \leftarrow \\ \quad\quad \mathbf{in}(\texttt{amount\_available}, \texttt{supplier}:\textit{monitorStock}(\texttt{Amount}, \texttt{Part\_id})) \ \& \\ \quad\quad \mathbf{in}(\texttt{Success}, \texttt{supplier}:\textit{shipFreight}(\texttt{Amount}, \texttt{Part\_id}, \texttt{truck}, \texttt{Src}, \texttt{Dest})). \end{array}$$

If 5 units of part_008 are available, then sendViaTruck (3, part_008, rome, paris, `Success`*) will be satisfied and* `Success` *will be* **true**, *if the shipping was possible. But the query sendViaTruck (7, part_008, rome, paris,* `Success`*) will not be satisfied, as the first* **in(,)** *above was not satisfied and hence the second* **in(,)** *above was never called.*

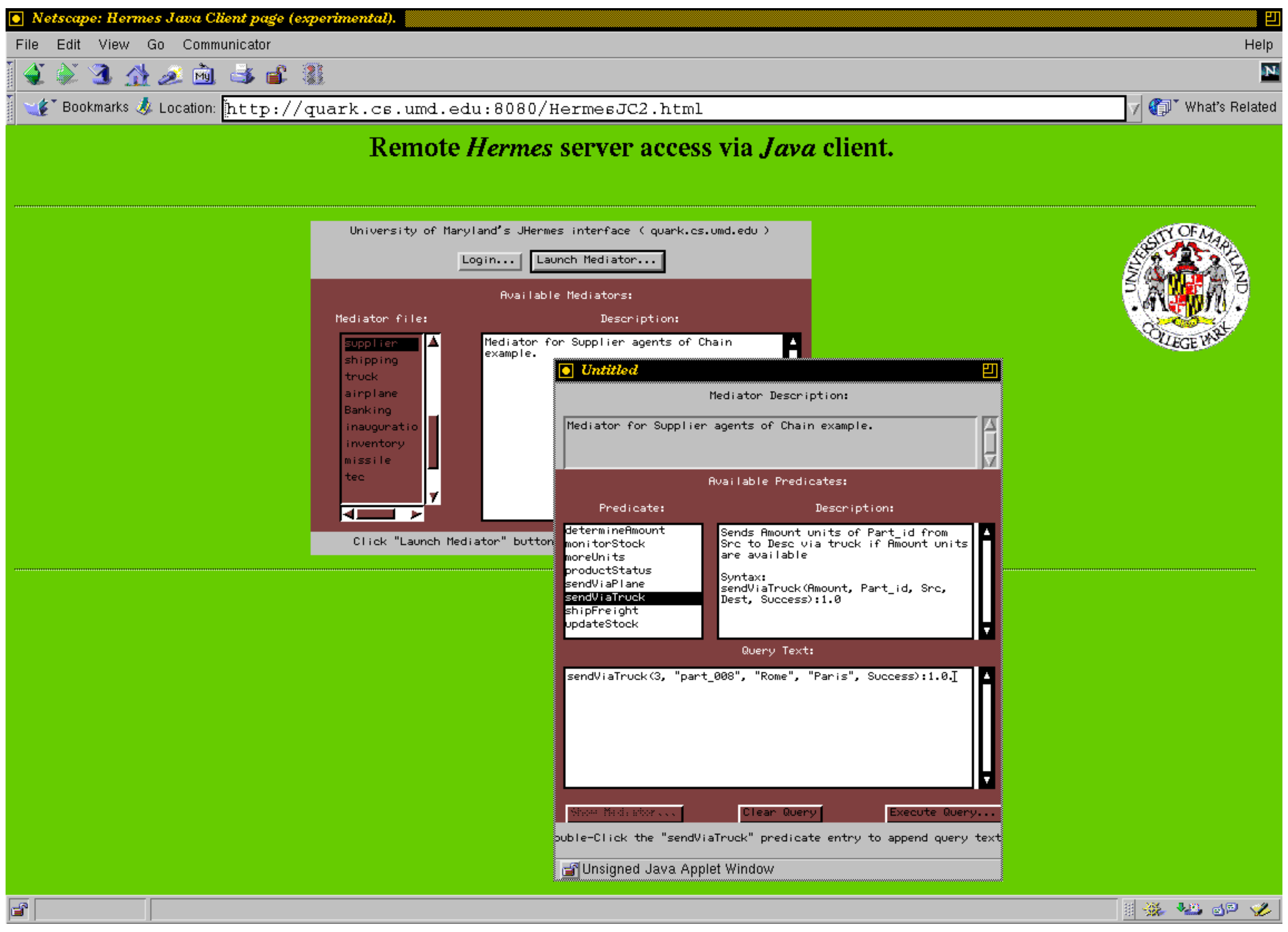

Figure 6.3: Sample query on the `supplier` agent's Mediator

6.6 Summary

This chapter was about a mechanism ($\leadsto$ **code call atoms**) to abstract from given legacy code and declaratively describe its effects.

1. In order to **agentize** legacy code, we must make the most important datatypes and functions of it available to *IMPACT*.
2. We call these functions f **code calls**: $\mathcal{S}:f(\mathtt{d}_1,\ldots,\mathtt{d}_n)$.
3. We assume that f always returns a set.
4. To encapsulate these functions in a logical language, we use ***code call atoms***: $\mathbf{in}(\mathtt{X},\mathcal{S}:f(\mathtt{d}_1,\ldots,\mathtt{d}_n))$.
5. Code call atoms can be conjunctively merged together (with comparison statements) and lead to **Code Call Conditions**.
6. To ensure that Code Call Conditions can be evaluated, we introduced the notion of **Safety**.

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 (1999, September). Temporal agent programs. Technical Report CS-TR-4055, Dept. of CS, University of Maryland, College Park, MD 20752. currently under submission for a Journal.

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 45*(1).

Eiter, T., V. Subrahmanian, and G. Pick (1999). Heterogeneous Active Agents, I: Semantics. *Artificial Intelligence 108*(1-2), 179–255.

Eiter, T., V. Subrahmanian, and T. 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.