IFI TECHNICAL REPORTS

Institute of Computer Science,
Clausthal University of Technology

IfI-05-07

Clausthal-Zellerfeld 2005

IMPACT: A multi-agent framework with declarative semantics*

Jürgen Dix [1] and Yingqian Zhang [2]

[1] Clausthal University of Technology, Department of Computer Science,
Chair for Computational Intelligence, Germany
dix@tu-clausthal.de
[2] University of Manchester, School of Computer Science, United Kingdom
zhangy@cs.man.ac.uk

Abstract

The *IMPACT* project (`http://www.cs.umd.edu/projects/impact`) aims at developing a powerful multi-agent system platform, which (1) is able to deal with heterogenous and distributed data, (2) can be realised on top of arbitrary legacy code, (3) is built on a clear foundational basis, and (4) scales up for realistic applications. We will describe its main features and several extensions of the language that have been investigated (and partially implemented).

1 Motivation

One of the main features of *IMPACT* is the idea of *agentisation*: *IMPACT* agents are usually built around given legacy code (see [24]). Another important feature is to provide a clear semantics for agents (based on the notion of an *agent program*) that can be easily extended (incorporating time, uncertainty, beliefs etc). The third feature is to identify classes of programs that can be efficiently implemented (polynomial modulo the underlying code).

In this chapter we are trying to illustrate these features through two examples. While Example 2.1 serves to illustrate the syntax and semantics of (temporal) agent programs, Example 2.2 shows the *agentisation* idea by turning a dedicated planning system into an agent collaborating with other agents in a wider environment. This example is also used to demonstrate some aspects of the third feature.

Before turning to the examples in Section 2, we need to make some general remarks. In order to turn legacy codeinto an agent α, we need to abstract from the given code and

*Several examples and screenshots included here are taken from the book and the following papers [6, 8, 15, 3]. The authors would like to thank their co-authors for their permission to use material from these papers.

describe its main features. Such an abstraction is given by the set of all datatypes and functions the software is managing. We call this a *body of software code* and denote it by $\mathcal{S}^{\mathfrak{a}} =_{def} (\mathcal{T}_{\mathcal{S}}^{\mathfrak{a}}, \mathcal{F}_{\mathcal{S}}^{\mathfrak{a}}, \mathcal{C}_{\mathcal{S}}^{\mathfrak{a}})$. $\mathcal{F}_{\mathcal{S}}^{\mathfrak{a}}$ is a set of predefined functions which makes access to the data objects ($\mathcal{T}_{\mathcal{S}}^{\mathfrak{a}}$) managed by the agent available to external processes. $\mathcal{C}_{\mathcal{S}}^{\mathfrak{a}}$ are composition operators to build new datatypes from the given ones.

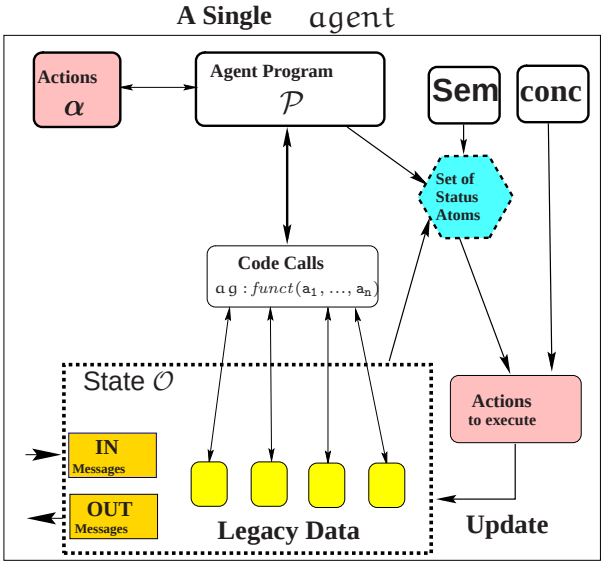

Figure 1: An Agent in *IMPACT*.

To get a bird's eye view of *IMPACT*, here are the most important features (see Figure 1):

- Each *IMPACT* agent has certain *actions* α available. Agents act in their environment according to their *agent program* $\mathcal{P}$ and a well defined *semantics* Sem determining which of the actions the agent should execute.
- Each agent continually undergoes the following cycle:
  - *(1)* Get messages sent by other agents. This changes the state $\mathcal{O}$ of the agent.
  - *(2)* Determine (based on its program $\mathcal{P}$, its semantics Sem and its state $\mathcal{O}$) for each action α its *status* (permitted, obliged, forbidden, ...). The agent ends up with a *set of status atoms*.
  - *(3)* Based on a notion of concurrency **conc**, determine the actions that can be executed and update the state accordingly.

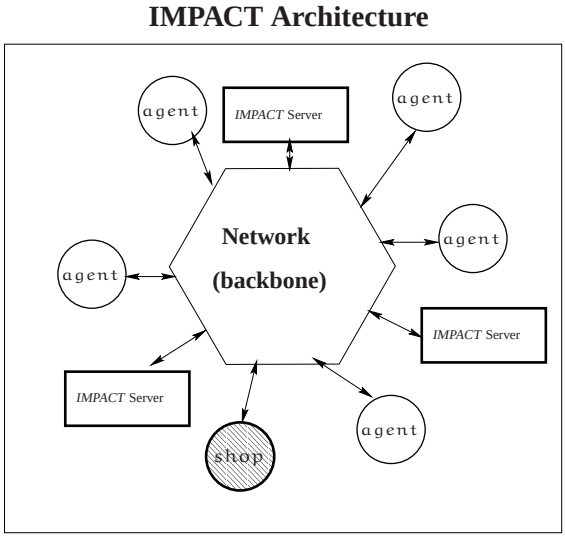

Figure 2: SHOP as a planning agent in *IMPACT*.

- *IMPACT* agents are built on top of arbitrary software code $\mathcal{S}^{\mathfrak{a}} =_{def} (\mathcal{T}_{\mathcal{S}}^{\mathfrak{a}}, \mathcal{F}_{\mathcal{S}}^{\mathfrak{a}}, \mathcal{C}_{\mathcal{S}}^{\mathfrak{a}})$ (*Legacy Data*).
- A methodology for transforming arbitrary software (legacy code) into an *agent* has been developed.

A complete description of all these notions is out of scope of this paper and we refer to [24] for a detailed presentation.

Before explaining an agent in more detail, we start with some remarks about the general architecture. In *IMPACT* agents communicate with other agents through the network. Not only can they send out (and receive) messages from other agents, they can also ask the server to find out about services that other agents offer. For example a planning agent (let us call it A-SHOP), confronted with a particular planning problem, can find out if there are agents out there with the data needed to solve the planning problem; or agents can provide A-SHOP with information about relevant legacy data.

In many applications a `statistics` agent is needed. This agent keeps track of distances between two given points and the authorised range or capacity of certain vehicles. This information can be stored in several databases. Another example is the `supplier` agent. It determines through its databases which vehicles are accessible at a given location.

Definition 1.1 (State of an Agent, $\mathcal{O}_{\mathcal{S}(t)}$) *At any given point t in time, the* state of an agent, *denoted $\mathcal{O}_{\mathcal{S}(t)}$, is the set of all data objects that are currently stored in the rela-*

tions the agent handles—the types of these objects must be in the base set of types in $\mathcal{T}_{\mathcal{S}}$.

The state of the statistics agent consists of all tuples stored in the databases it handles. The state of the supplier agent is the set of all tuples describing which vehicles are accessible at a given location.

We noted that agents can send and receive messages. There is therefore a special datastructure, the *message box*, part of each agent. This message box is just one of those types. Thus a state change already occurs when a message is received.

2 Language

Agents are specified in *IMPACT* through *agent programs*. The basic language of *IMPACT* does not allow to formalise *mental attitudes*, or *temporal* or *probabilistic reasoning*. However all these features have been investigated (see [8, 7, 6, 14, 16]) and the approach using both temporal as well as probabilistic reasoning is currently implemented.

In order to illustrate the language and semantics of *IMPACT* with an example, that is not too technical nor too trivial, we have chosen one involving temporal reasoning alone. This example serves to show the salient features of *IMPACT*.

Example 2.1 [Rescue Scenario I, temporal reasoning] Consider a simplistic rescue operation where a natural calamity (e.g., a flood) has stranded many people. Rescuing these people requires close coordination between helicopters and ground vehicles. For the sake of this example, we assume the existence of:

1. A helicopter agent that conducts aerial reconnaissance and supports aerial rescues;
2. A set gv1, gv2, gv3 of ground vehicles that move along the ground to appropriate locations—such vehicles may include ambulances as well as earth moving vehicles.
3. An immobile command centre agent comc that coordinates between the helicopter and the ground vehicles.

Here is a typical statement that should be expressible in an agent language.

> *"If the maximal time previously taken to ship some equipment* E *from location* A *to location* B *is* T_1*, and if equipment* E *is required to be at location* B *at time* T*, then ship* E *sometime between time* $T - T_1 - 10$ *and* $T - T_1$*."*

This is a very reasonable statement to make not only in our rescue example, but in any logistics application. The time T might depend on the production schedule of the company at location B (which may be determined at run-time from a database), and T_1

likewise might depend on the identities of locations A, B (which may be instantiated at run time and whose locations might therefore need to be inferred at run-time from a database).

The second example, similar in spirit, is used to illustrate the *agentisation procedure* .

Example 2.2 [Rescue Scenario II, agentising a planner] The planner SHOP [21] is a stand-alone system which did very well in planning competitions. It uses a particular framework to encode planning problems: *hierachical task networks*. While SHOP is a very efficient planner, it requires that all data is stored locally and given in a particular format (atomic facts in Lisp notation). Such planning systems usually support only one kind of reasoning: *symbolic* or *numeric*, but not both.

How can such a planning system be agentised in IMPACT as a planning agent A-SHOP?

The typical test domain for a planner where data is heterogenous and stored at different places is a simple transportation planning problem for a rescue mission (*NEO* [20]). Computing plans involves performing a rescue mission where a task force is grouped and transported between an *initial location* (the assembly point) and the *NEO* site (where the evacuees are located). After the troops arrived at the *NEO* site, evacuees are re-located to a *safe haven*.

The planning task involves:

1. selecting possible pre-defined routes, consisting of four or more segments each;
2. choosing a transportation mode for each segment;
3. determining conditions such as whether communication exists with *State Department personnel* and the type of evacuee registration process.

Here we have four different *IMPACT* information sources available:

- Transport Authority: Maintains information about the transportation assets available at different locations.
- Weather Authority: Maintains information about the weather conditions at the different locations.
- Airport Authority: Maintains information about availability and conditions of airports at different locations.
- Math Agent: `math` evaluates arithmetic expressions. Typical evaluations include to subtract a certain number of assets use for an operation and update time delays.

Agentising given legacy code cannot be done automatically: the agent designer has to determine the abstraction level. In particular she has to decide which of the data structures find their way into the state of the agent (to be built) and which are considered mere “implementation details”.

2.1 Specifications and Syntactical Aspects

In *IMPACT*, each agent $\mathfrak{a}$ is built on top of a body of software code (built in any programming language) that supports a well defined application programmer interface (either part of the code itself, or developed to augment the code).

Definition 2.3 (Software Code) *We may characterise the code on top of which an agent $\mathfrak{a}$ is built as a triple $\mathcal{S}^{\mathfrak{a}} =_{def} (\mathcal{T}_{\mathcal{S}}^{\mathfrak{a}}, \mathcal{F}_{\mathcal{S}}^{\mathfrak{a}}, \mathcal{C}_{\mathcal{S}}^{\mathfrak{a}})$ where:*

1. *$\mathcal{T}_{\mathcal{S}}^{\mathfrak{a}}$ is the set of all data types managed by $\mathcal{S}$,*
2. *$\mathcal{F}_{\mathcal{S}}^{\mathfrak{a}}$ is the set of predefined (API) functions over $\mathcal{T}_{\mathcal{S}}^{\mathfrak{a}}$ through which external processes may access $\mathfrak{a}$'s data, and*
3. *$\mathcal{C}_{\mathcal{S}}^{\mathfrak{a}}$ 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 output a type $c(\tau_1, \ldots, \tau_n)$.*

This characterisation of a piece of software code is widely used (cf. the *Object Data Management Group*'s *ODMG* standard [2] and the *CORBA* framework [22]).

Each agent also has a message box having a well defined set of associated code calls that can be invoked by external programs.

Example 2.4 [Rescue Scenario I] Consider the rescue mission described earlier. The heli agent may have the following data types and code calls.

- Data Types: $speed, bearing$ of type `int`, $location$ of type `point` (record containing x, y, z fields), $nextdest$ of type `string`, and $inventory$—a relation having schema (`Item`, `Qty`, `Unit`).
- Functions:
  - heli : $location()$: which returns the (x, y, z) coordinates of the current position of the helicopter.
  - heli : $inventory$(`Item`): returns a pair of the form $\langle$*Qty*, *Unit*$\rangle$. For example, heli : $inventory$(`blood`) may return $\langle$*25*, *litres*$\rangle$ specifying that the helicopter currently has 25 units of blood available.

An agent's state may change because it took an action, or because it received a message. We assume that except for appending messages to an agent $\mathfrak{a}$'s mailbox, another agent $\mathfrak{b}$ cannot directly change $\mathfrak{a}$'s state. However, it might do so indirectly by sending the other agent a message requesting a change.

Example 2.5 [Rescue Scenario I: State] For instance, at a given instant of time, the state of the heli agent may consist of $location = \langle 45, 50, 9000\rangle$, and $inventory$ containing the tuples: $\langle fuel, 125, gallons\rangle$, $\langle blood, 25, litres\rangle$, $\langle bandages, 50, -\rangle$, $\langle cotton, 20, lbs\rangle$.

Queries and/or conditions may be evaluated w.r.t. an agent state using the notion of a code call atom and a code call condition (CCC) defined below.

Definition 2.6 (Code Call (CC)/Code Call Atom) *If $\mathcal{S}$ is the name of a software package, f is a function defined in this package, and $(d_1, \ldots, d_n)$ is a tuple of arguments of the input type of f, then the term $\mathcal{S} : f(\mathtt{d_1}, \ldots, \mathtt{d_n})$ is called a* code call *(denoted by CC).*

If `cc` *is a code call, and* `X` *is either a variable symbol, or an object of the output type of* `cc`*, then* **in**$(\mathtt{X}, \mathtt{cc})$ *is called a* code call atom.

If `X` is a variable over type τ and τ is a record structure with field f, then `X`.f is a variable ranging over objects of the type of field f.

Definition 2.7 (Code Call Condition (CCC))

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

For example, **in**$(\mathtt{X}, \mathtt{heli} : \mathit{inventory}(\mathtt{fuel})) \,\&\, \mathtt{X.Qty} < 50$ is a code call condition that is satisfied whenever the helicopter has less than 50 gallons of fuel left.

The code call condition

$$\begin{array}{l}\textbf{in}(\mathtt{FinanceRec}, \mathtt{rel} : \mathit{select}(\textit{finRel}, \textit{date}, \texttt{"="}, \texttt{"Nov. 99"})) \,\& \\ \mathtt{FinanceRec.sales} \geq 10K \,\& \\ \textbf{in}(\mathtt{C}, \mathtt{excel} : \mathit{chart}(\textit{excFile}, \mathtt{FinanceRec}, \textit{day})) \,\& \\ \textbf{in}(\mathtt{Slide}, \mathtt{ppt} : \mathit{include}(\mathtt{C}, \texttt{"presnt.ppt"}))\end{array}$$

is a complex condition that accesses and merges data across a relational database, an Excel file, and a PowerPoint file. It first selects all financial records associated with "Nov. 99": this is done with the variable `FinanceRec` in the first line. It then filters out those records having sales more than $10K$ (second line). Using the remaining records, an Excel chart is created with day of sale on the x-axis and the resulting chart is included in the PowerPoint file "presentation.ppt" (fourth line).

In the above example, it is very important that the first code call be evaluable. If, for example, the constant $finRel$ were a variable, then

$$\mathtt{rel} : \mathit{select}(\mathtt{finRel}, \mathtt{date}, \texttt{"="}, \texttt{"Nov. 99"})$$

would not be evaluable, unless there were another condition instantiating this variable.

We have introduced syntactic conditions, similar to *safety* in classical databases, to ensure *evaluability* of CCC's. It is also quite easy to store CCC's as evaluation graphs

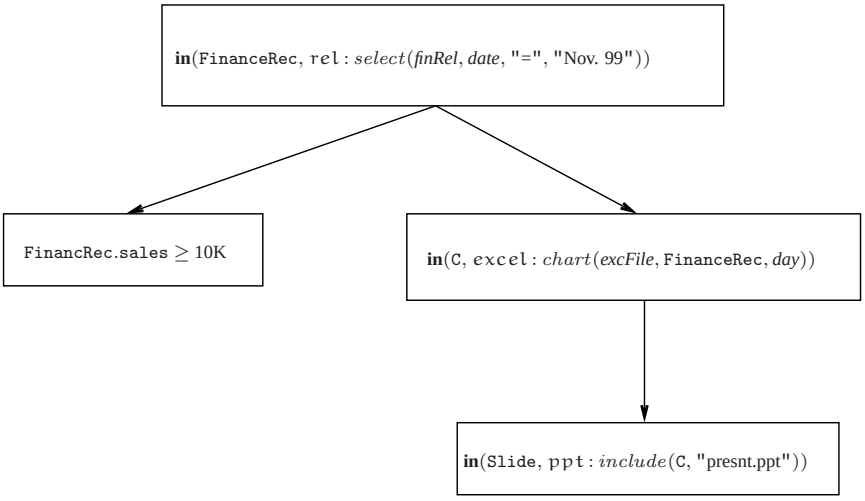

Figure 3: A code call evaluation graph

(see Figure 3), thereby making explicit the dependency relation between its constituents (see [15]).

Code call conditions provide a simple, yet powerful language syntax to access heterogeneous data structures and legacy software code. However, in general their use in agent programs is not constrained: it is perfectly possible that a CCC cannot be evaluated (and thus the status of actions cannot be determined). A reason for this could be uninstantiated variables (so that the underlying functions cannot be executed).

Actions in *IMPACT*

Each agent has an associated action-base describing various actions that the agent is capable of executing. An action (whose behaviour is that of a partial function from states to states) is implemented by a body of code in any suitable imperative (or declarative) programming language. The agent reasons about actions via a set of preconditions and effects defining the conditions an agent state must satisfy for the action to be considered executable, and the new state that results from such an execution. We assume that the preconditions and effects associated with an action correctly specify the behaviour of the code implementing the action. Note, that in addition to changing the state of the agent, an action may change the state of other agents' `msgboxes`.

Here is an example of a timed action $drive()$ of the `truck` agent which may be described via the following components:

Name: $drive(\texttt{From}, \texttt{To}, \texttt{Highway})$

Schema: (String,String,String)

Pre: **in**(From, truck : $location()$)

Dur: {T | **in**(X, math : $distance$(From, To)) & **in**(T, math : $compute(\frac{60X}{70})$)}

Tet:

1st arg	: rel : {20}
2nd arg	:{**in**(NewPosition, truck : $location$(X_{now})) }
3rd arg	:{**in**(OldPosition, truck : $location$($X_{now} - 20$)) }

The **Tet** part says that the truck agent updates its location every 20 minutes (assuming a time period is equal to 1 minute) during the expected time it takes it to drive the distance between From to To at 70km per hour.

2.2 Semantics and Verification

One of the main features of *IMPACT* is that it has a precise, formal semantics based on the notion of *agent programs*. These programs are, from an abstract point of view, logic programs (*if-then-else* rules). The semantics of such programs has been investigated extensively in the last three decades. Consequently there is a vast amount of techniques we can build on.

Our language is not purpose-specific: it is a general framework to design arbitrary agents collaborating together. While the original framework did not support *temporal* or *probabilistic* reasoning, these features are currently implemented.

While we have not yet developed the formal machinery for *verifying* agents, the path for doing so is certainly laid.

Each agent has (i) a set of **integrity constraints** $\mathcal{IC}$—only states that satisfy these constraints are considered to be *valid* or *legal* states, (ii) a notion of **concurrency** specifying how to combine a set of actions into a single action, (iii) a set of **action constraints** that define the circumstances under which certain actions may be concurrently executed, and (iv) an **agent program** that determines what actions the agent can take, what actions the agent cannot take, and what actions the agent must take. Agent programs are defined in terms of status atoms defined below.

Definition 2.8 (Status Atom/Status Set) *If* $\alpha(\vec{t})$ *is an action, and* Op $\in \{\mathbf{P}, \mathbf{F}, \mathbf{W}, \mathbf{Do}, \mathbf{O}\}$, *then* Op$\alpha(\vec{t})$ *is called a* status atom. *If* A *is a status atom, then* $A, \neg A$ *are called* status literals. *A* status set *is a finite set of ground status atoms.*

Intuitively, $\mathbf{P}\alpha$ means α is permitted, $\mathbf{F}\alpha$ means α is forbidden, $\mathbf{O}\alpha$ means α is obligatory, $\mathbf{Do}\,\alpha$ means α is to be done, and $\mathbf{W}\alpha$ means that the obligation to perform α is waived. Note that these operators are not independent from each other. For example, an action α cannot have the status $\mathbf{F}$ and $\mathbf{O}$ at the same time. And $\mathbf{O}\alpha$ should always imply $\mathbf{Do}\,\alpha$. These interrelations are taken into account by the semantics.

Definition 2.9 (Agent Program) *An agent program $\mathcal{P}$ is a finite set of rules of the form $A \leftarrow \chi \,\&\, L_1 \,\&\, \ldots \,\&\, L_n$, where χ is a code call condition, L_i are status literals and A is a status atom.*

Several alternative semantics for agent programs are presented in [19, 17].

For example, the heli agent in our Rescue Example may execute the action $fly(\texttt{"BigRag"}, \texttt{"StonyPoint"})$. This action lasts for a period of time during which the location of heli is changing continuously. More importantly, if we know the location of the plane *now* and we know the plane's velocity and climb angle, we can precisely compute its location in the future (assuming no change in these parameters). Thus, in order to specify a *timed action*, we must:

1. Specify an estimate of the total amount of time it takes for the action to be "completed".

2. Specify exactly how the state of the agent changes *while* the action is being executed.

It is worth noting that the duration of an action can be precisely specified in some cases, but not in others. For instance, saying that *the action* $drive(i95, south, 60)$ *should be executed for 2 hours* is a precise specification saying that the action "Drive south on Interstate I-95 at 60 mph" is to be executed for 2 hours. However, it is hard to specify durations of actions such as $drive(washington, baltimore)$. In this case, the above definition requires an *estimate* to be provided.

Definition 2.10 (Temporal Agent Rule/Program $\mathcal{TP}$) *A* temporal agent rule *is an expression of the form* $\mathsf{Op}\,\alpha : [\mathsf{tai}_1, \mathsf{tai}_2] \leftarrow \varrho_1 : \mathsf{ta}_1 \,\&\, \cdots \,\&\, \varrho_n : \mathsf{ta}_n$, *where* $\mathsf{Op} \in \{\mathbf{P}, \mathbf{Do}, \mathbf{F}, \mathbf{O}, \mathbf{W}\}$, *and* $\varrho_1 : \mathsf{ta}_1, \ldots, \varrho_n : \mathsf{ta}_n$ *are* tascs[1]. *A* temporal agent program *(tap) is a finite set of temporal agent rules.*

> **Intuitive Reading of Temporal Agent Rule**
> *"If for all $1 \leq i \leq n$, there exists a time point t_i such that ϱ_i is true at time t_i such that $t_i \in \mathsf{ta}_i$ then $\mathsf{Op}\,\alpha$ is true at some point $t \geq \mathsf{t}_{now}$ (i.e., now or in the future) such that $\mathsf{tai}_1 \leq t \leq \mathsf{tai}_2$".*

How can taps be used to express the statement in Example 2.1? We use two relational databases—one called `shipdata` containing at least the attributes `shiptime,orig,dest` (and perhaps other ones as well) which specifies data (such as shipping time) associated with past shipments. The other relational table is called `sched` which has at least the attributes `reqtime,place,item` specifying which items are required at what time by what places.

[1] A tasc (temporal action status conjunct) is, intuitively, a conjunction of temporal status actions. We refer to [6] for further detail.

$$\mathbf{Do}\ ship(\mathtt{P},\mathtt{A},\mathtt{B}):[\mathtt{T}-\mathtt{T}_1-10,\mathtt{T}-\mathtt{T}_1]\ \leftarrow$$
$$(\mathbf{in}(\mathtt{T}_1,\ \mathtt{db}:sql(\text{`}\mathtt{SELECT\ time\ FROM\ data\ WHERE\ orig}=\mathtt{A}\ \&\ \mathtt{dest}=\mathtt{B}))) \ \&$$
$$\mathbf{in}(\mathtt{T},\ \mathtt{db}:sql(\text{`}\mathtt{SELECT\ reqtime\ FROM\ place\ WHERE\ item}=\mathtt{P}'))) : [\mathtt{X}_{\mathsf{now}},\mathtt{X}_{\mathsf{now}}].$$

Here is another example. *"If a prediction package expects a stock to rise K% after T_K units of time and $K \geq 25$ then buy the stock at time* $(\mathtt{X}_{now}+T_K-2)$*."* We assume a prediction package that given a stock uses some stock expertise to predict the change in the value of the stock at future time points. This function returns a set of pairs of the form (T,C). Intuitively, this says that T time units from now, the stock price will change by C percent (positive or negative).

$$\mathbf{Do}\ buy(\mathtt{S}):[\mathtt{X}_{\mathsf{now}}+\mathtt{X.T}-2,\mathtt{X}_{\mathsf{now}}+\mathtt{X.T}-2]\ \leftarrow$$
$$\mathbf{(in}(\mathtt{X},\ \mathtt{pred}:dest(\mathtt{S}))\ \&\ \mathtt{X.C}\geq 25\mathbf{)}:[\mathtt{X}_{\mathsf{now}},\mathtt{X}_{\mathsf{now}}].$$

Finally, here is a tap using several rules and different status atoms.

1. $\mathbf{F}\,drive(\mathtt{was},\mathtt{bal},\mathtt{hw295}):[\mathtt{t}_{\mathsf{now}},\mathtt{t}_{\mathsf{now}}+2]\leftarrow$
   $\mathbf{in}(\mathtt{hw295},\ \mathtt{msgbox}:gatherWarning(\mathtt{comc})):[\mathtt{t}_{\mathsf{now}}-3,\mathtt{t}_{\mathsf{now}}]$
2. $\mathbf{Do}\,fill\_fuel():[\mathtt{t}_{\mathsf{now}},\mathtt{t}_{\mathsf{now}}]\leftarrow$
   $\mathbf{in}(\mathbf{true},\ \mathtt{truck}:tank\_empty()):[\mathtt{t}_{\mathsf{now}}-2,\mathtt{t}_{\mathsf{now}}]$
3. $\mathbf{O}\,order\_item(\mathtt{fa\_bag}):[\mathtt{t}_{\mathsf{now}},\mathtt{t}_{\mathsf{now}}+4]\leftarrow$
   $\mathbf{in}(1,\ \mathtt{truck}:inventory(\mathtt{fa\_bag}))[\mathtt{t}_{\mathsf{now}}-3,\mathtt{t}_{\mathsf{now}}]$
4. $\mathbf{P}\,drive(\mathtt{was},\mathtt{bal},\mathtt{hw95}):[\mathtt{t}_{\mathsf{now}},\mathtt{t}_{\mathsf{now}}]\leftarrow$
   $\mathbf{in}(\mathbf{false},\ \mathtt{truck}:tank\_empty()):[\mathtt{t}_{\mathsf{now}},\mathtt{t}_{\mathsf{now}}]\ \&$
   $\mathbf{F}\,drive(\mathtt{was},\mathtt{bal},\mathtt{hw295}):[\mathtt{t}_{\mathsf{now}}+1,\mathtt{t}_{\mathsf{now}}+2]$

Figure 4 shows two rules (with $\mathbf{Do}$'s in the head) of the monitoring agent in A-SHOP.

Our approach is to base the semantics of agent programs on *consistent* and *closed* status sets. Consistent means that there are no inconsistencies (such as $\mathbf{F}\alpha$ and $\mathbf{P}\alpha$ in the same set) and closed means that when $\mathbf{Do}\,\alpha$ is in the set, then so is $\mathbf{P}\alpha$.

However, we also have to take into account not only the rules of the program but also the integrity constraints $\mathcal{IC}$. This leads us to the notion of a feasible status set. The operator $\mathbf{App}_{\mathcal{P},\mathcal{O}_\mathcal{S}}(S)$ is similar to the immediate consequence operator in logic programming: it computes all the consequences obtainable from applying all agent rules once.

Definition 2.11 (Feasible Status Set) *Let $\mathcal{P}$ be an agent program, and let $\mathcal{O}_\mathcal{S}$ be an agent state. Then, a status set S is a* feasible status set *for $\mathcal{P}$ on $\mathcal{O}_\mathcal{S}$, if the following conditions hold:*

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

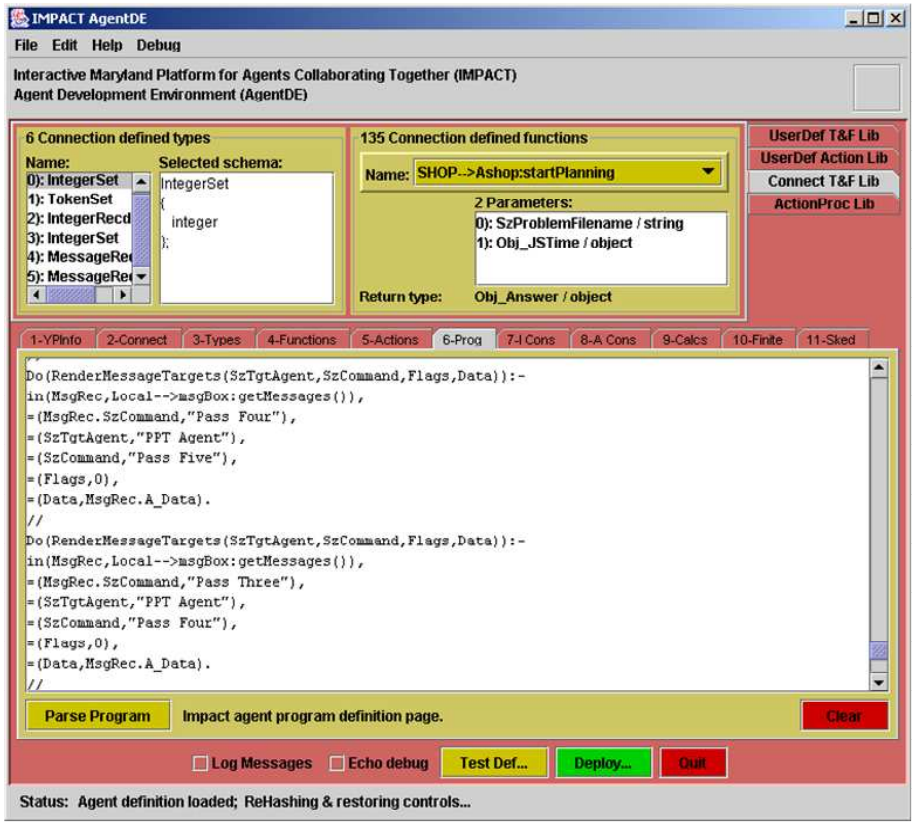

Figure 4: *AgentDE* Program

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

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

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

The last condition ensures that the successor state (when all doable actions are executed) still satisfies the integrity constraints $\mathcal{IC}$.

The semantics of agent programs is then defined by *rational status sets*.

Definition 2.12 (Groundedness; Rational Status Set) *A status set* S *is* grounded*, if there exists no status set* $S' \subsetneqq S$ *such that* 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.*

Thus given an agent program, our semantics computes all rational status sets of this program. In the case of positive agent programs (all examples in this chapter have this property) it can be shown that there always exists exactly one rational status set. Rational status sets are natural generalisations of stable models (or answer sets) in logic programming.

Figure 5 shows the successful compilation of an agent program (the monitoring agent in A-SHOP). In the first phase the rules are organised in several layers, then the program is unfolded (sometimes producing more rules but obtaining an optimised version), the data connection is checked, and the status set is generated.

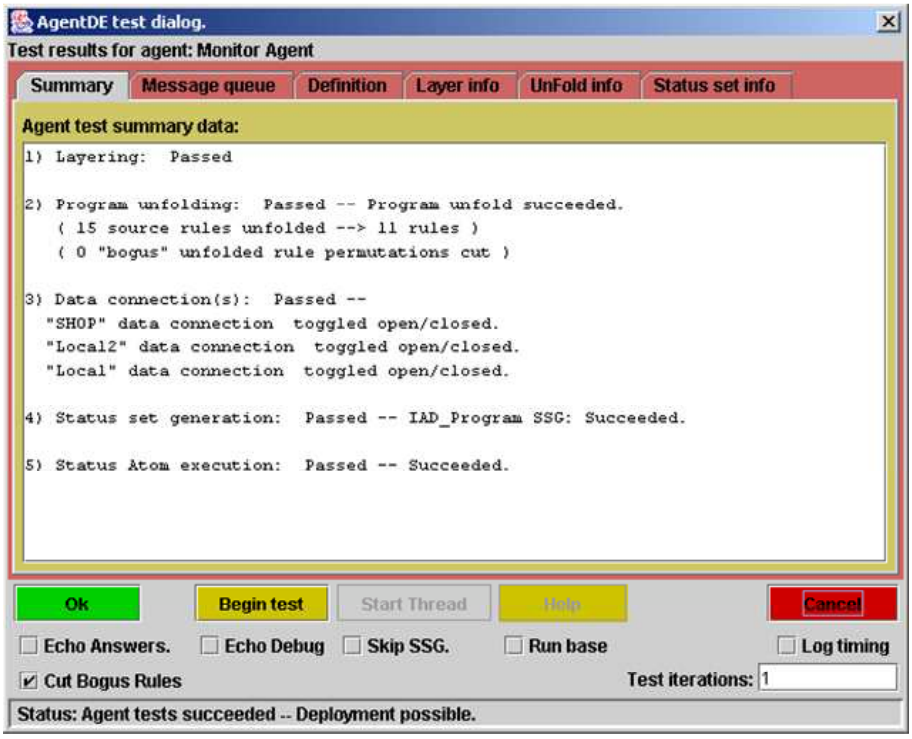

Figure 5: *AgentDE* Summary Table

2.3 Software Engineering Issues

We have finished the *IMPACT* implementation based on our main theory (extensions are underway for temporal programs, temporal probabilistic programs, etc.). Several nontrivial multiagent applications have been developed with *IMPACT*. The *IMPACT* implementation has a simple Java-based, web accessible interface which allows the

user to specify an agent's different component definitions (type, function, action, agent program, etc.) and communication between agents. It provides an easy way to maintain and test the different components within a multiagent system. We will introduce it further later in this chapter.

As we showed in the previous section, *IMPACT* is able to *agentise* any software program and plug it into the provided solution. *IMPACT* supports this both in its theory and in its implementation. *Code call condition* mechanism supports queries to arbitrary legacy code or specialised data structures. Moreover, the implementation of *IMPACT* supports execution of code call conditions over a wide variety of software packages.

We also consider the reliability issue in our method. The reliability of *IMPACT* is provided by replication and by minimising the dependency of individual agents in *IMPACT*. We refer to [24] for further detail.

2.4 Other features of the language

As already mentioned in the beginning, *IMPACT* is based on two important features.

Complexity: special emphasis is put on identifying classes of programs that can be efficiently implemented. The class of *regular* agents (based on a special class of agent programs) ensures that its complexity modulo the underlying legacy code is only polynomial [18].

Legacy code: existing legacy code can be turned into an *IMPACT* agent (*agentisation*). This is illustrated with A-SHOP, which is an agentised version of SHOP, a well-known planning system ([9, 12, 13, 11]).

Our framework supports the design of mobile agents because mobility can be considered as an action that any agent can execute. In addition, we show in [24] that Java applets can be viewed as IMPACT agents.

Our language is modular and can be easily extended by new constructs. Not only syntactic sugar, but also non trivial features such as temporal or probabilistic reasoning can be incorporated (through annotated logic programs). These extensions are not always trivial, but the overall system is designed so as to allow them. We consider this to be a salient feature of our framework.

Complexity Issues

We mentioned in Subsection 2.1 the condition of safeness to ensure evaluability of a code call. We also mentioned that an evaluable CC does not need to terminate. Consider the code call

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

which constitutes all numbers that are less than 2000 and that are squares of an integer greater than or equal to 25.

Clearly, over the integers there are only finitely many ground substitutions that cause this code call condition to be true. Furthermore, this code call condition is safe. However, its evaluation may never terminate. The reason for this is that safety requires that we first compute the set of all integers that are greater than 25, leading to an infinite computation.

Thus, in general, we must impose some restrictions on code call conditions to ensure that they are finitely evaluable. This is precisely what the condition of *strong safeness* ([18, 24]) does for the code-call conditions. Intuitively, by requiring that the code call condition is safe, we are ensuring that it is executable and by requiring that it is strongly safe, we are ensuring that it will only return finitely many answers.

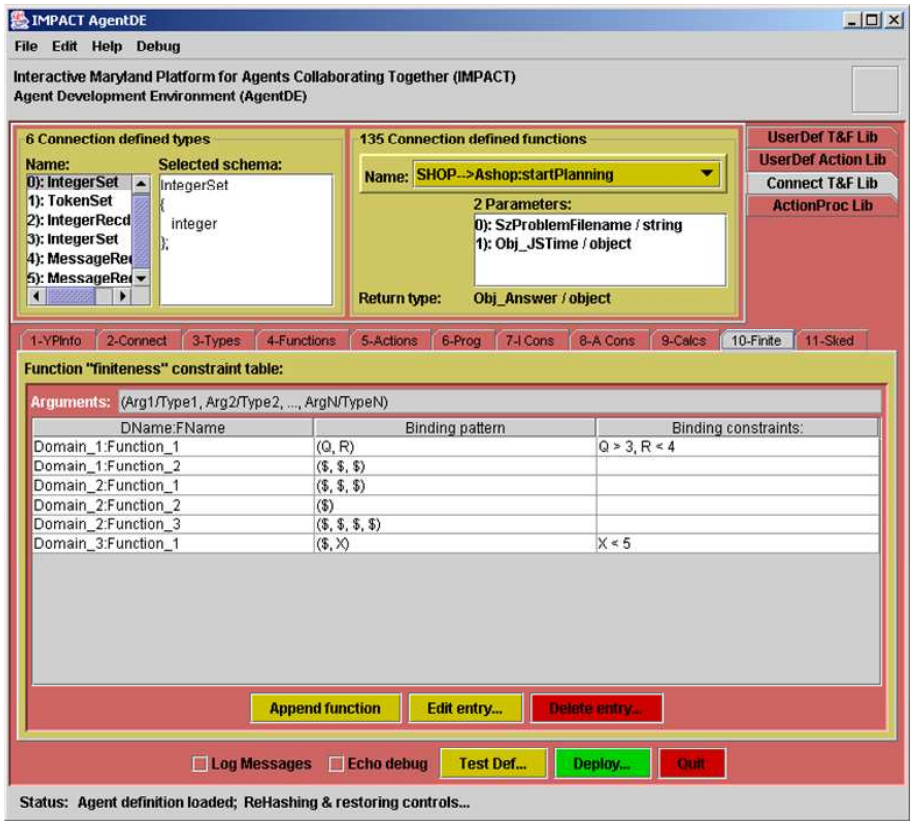

Figure 6: *AgentDE* Finiteness Table

Note that the problem of deciding whether an arbitrary code call execution terminates is undecidable (and so is the problem of deciding whether a code call condition χ holds in $\mathcal{O}$). Therefore we need some input of the agent designer (or of the person who is

responsible for the legacy code the agent is built upon). The information needed is stored in a *finiteness table* (see [18, 24] and Figure 6). This information is used in the *purely syntactic* notion of strong safeness. It is a *compile-time check*, an extension of the well-known (syntactic) safety condition in databases.

Agentisation

Our Example 2.2 serves to illustrate how to turn a planner into an planning agent within a multi-agent environment.

SHOP, as an *HTN* planner, is based on the concepts of *tasks*, *operators* and *methods*. Methods are used to decompose a nonprimitive task and form the heart of *HTN* planning.

A comparison between *IMPACT*'s actions and SHOP's methods shows that *IMPACT* actions correspond to fully instantiated methods. While SHOP's methods and operators are based on STRIPS, the first step is to modify the atoms in SHOP's preconditions and effects, so that SHOP's preconditions will be evaluated by *IMPACT*'s code call mechanism and the effects will change the state of the *IMPACT* agents. This is a fundamental change in the representation of SHOP. In particular, it requires replacing SHOP's methods and operators with *agentised* methods and operators. These are defined as follows.

Definition 2.13 (Rescue II, Agentised Operator) *An* agentised operator *is an expression of the form* $(\mathbf{AgentOp}\ h\ \chi_{add}\ \chi_{del})$*, where* h *(the* head*) is a primitive task and* χ_{add} *and* χ_{del} *are lists of code calls (called the* add- *and* delete-lists*). The set of variables in the tasks in* χ_{add} *and* χ_{del} *is a subset of the set of variables in* h*.*

Lemma 2.14 (Rescue II, Evaluating Agentised Operators) *Let* $(\mathbf{AgentOp}\ h\ \chi_{add}\ \chi_{del})$ *be an agentised operator. If the add and delete-lists* χ_{add} *and* χ_{del} *are strongly safe wrt. the variables in* h*, the problem of applying the agentised operator to* $\mathcal{O}$ *can be algorithmically solved.*

In SHOP, preconditions were logical atoms, and SHOP would infer these preconditions from its current state of the world using Horn-clause inference. In contrast, the preconditions in an agentised method are *IMPACT*'s code call conditions rather than logical atoms. Also A-SHOP (the agentised version of SHOP) does not use Horn-clause inference to establish these preconditions but instead simply invokes those code calls, which are calls to other agents (which may be Horn-clause theorem provers or may instead be something entirely different). This opens the way to use arbitrary reasoning mechanisms and data distributed over the net.

Theorem 2.15 (Rescue Scenario II, Sound- and Completeness) *Let* $\mathcal{O}$ *be a state and* $\mathcal{D}$ *be a collection of agentised methods and operators. If all the preconditions in the agentised methods and add- and delete-lists in the agentised operators are strongly safe wrt. the respective variables in the heads, then A-SHOP is sound and complete.*

Figure 7 shows a method for our application to logistics planning. The method indicates how to transport a cargo that has a certain weight between two locations. The method calls the statistics agent three times, in order to evaluate the *distance* between two geographic locations: (1) the *authorised range* of a certain aircraft type (the authorised range is lower than the real distance that the aircraft can fly), and (2) the *authorised capability* (in metric tones) of an aircraft. The method calls the *supplier* agent to evaluate the cargo planes that are available at a location.

Head:
$AirTransport(\texttt{LocFrom}, \texttt{LocTo}, \texttt{Cargo}, \texttt{CargoWeight})$

Preconditions:
in(CargoPL, supplier : $cargoPlane(\texttt{LocFrom})$)&
in(Dist, statistics : $distance(\texttt{LocFrom}, \texttt{locTo})$)&
in(DCargoPL, statistics : $authorRange(\texttt{CargoPL})$)&
Dist $\leq$ DCargoPL&
in(CCargoPL, statistics : $authorCapacity(\texttt{CargoPL})$)&
CargoWeight $\leq$ CCargoPL&

Subtasks:
$load(\texttt{Cargo}, \texttt{LocFrom})$
$fly(\texttt{Cargo}, \texttt{LocFrom}, \texttt{LocTo})$
$unload(\texttt{Cargo}, \texttt{LocTo})$

Figure 7: Agentised method for a logistics problem.

This top level task is decomposed into several subtasks, one for each segment in the route that the task force must cover (these segments are pre-determined as part of the problem description). Within each segment, A-SHOP must plan for the means of transportation (planes, helicopters, vehicles, etc.) to be used and select a route for that segment. The selection of the means of transportation depends on their availability for that segment, the weather conditions, and, in the case of airplanes, the availability and conditions of airports. The selection of the route depends on the transportation vehicle used and may lead to backtracking. For example, the choice of ground transportation assets needs to be revised if no roads are available, or they are blocked, or too risky to take.

Our test domain was a simplification of the actual conditions that occur in practice. Primarily because many more information sources are available in practice, and as such the resulting plans will be more complicated.

A-SHOP's knowledge base included six agentised operators and 22 agentised methods. We ran our experiments on 30 problems of increasing size and refer to [25, 13, 11] for detailed results.

3 Platform

3.1 Features of the platform

The *IMPACT* system consists of five major software components to support the development and deployment of *IMPACT* agents.

Agent Development Environment Agent developers can easily build and test agents within the *IMPACT* Agent Development Environment (*AgentDE* for short). As described earlier, the core parts of an *IMPACT* agent are:

1. a set of data type definitions and *API* function calls manipulated by the agent;
2. a set of actions that the agent may take;
3. a set of integrity constraints $\mathcal{IC}$ on the agent state and action constraints $\mathcal{AC}$;
4. an agent program $\mathcal{P}$ specifying the behaviour of the agent;
5. a notion of concurrency **conc**.

The *AgentDE* provides a network accessible, easy-to-use graphical user interface through which an agent developer can specify all the above parameters of an agent, compile and then test if they work properly.

AgentDE contains libraries of data types, *API* functions, actions and notions of concurrency. When the agent developer builds a new agent, each data type must be explicitly defined via the *AgentDE*. The agent manipulates its data types via *API* function calls, which can be defined within the *AgentDE*. Similarly, the developer needs to specify a set of actions that the agent can execute via *AgentDE*. Figure 8 shows how the developer can reuse actions in the library and assign them to the `monitoring` agent.

Figure 9 shows the interface of the *AgentDE* when the developer has finished specifying the data types, *API* functions and actions. The tab marked "Calcs" allows the user to specify the notion of concurrency[2] he wants to use. All these new items are added to the appropriate library so that during the development process, whenever the developer accesses the *AgentDE*, the definitions will be directly imported from the libraries for use.

After defining these parameters, the agent developer may start testing the agent. The *AgentDE* performs compile-time checks such as strong safety check, deontic satisfaction, and boundedness check. Pressing "Test Program" in Figure 9 triggers the test. When the test is started, unfolding is done first, then the data connections requested by the program are tested and established. After the test phase is completed, status sets are generated and executed. Figure 10 shows the status set computations.

[2] For example a very simple **conc** would be to just take the union of all add-lists and the union of all delete-lists. A more sophisticated **conc** would check whether all actions can be ordered in a way such that there are no conflicting actions, and then execute them one after the other. The latter is of course more complex than the first.

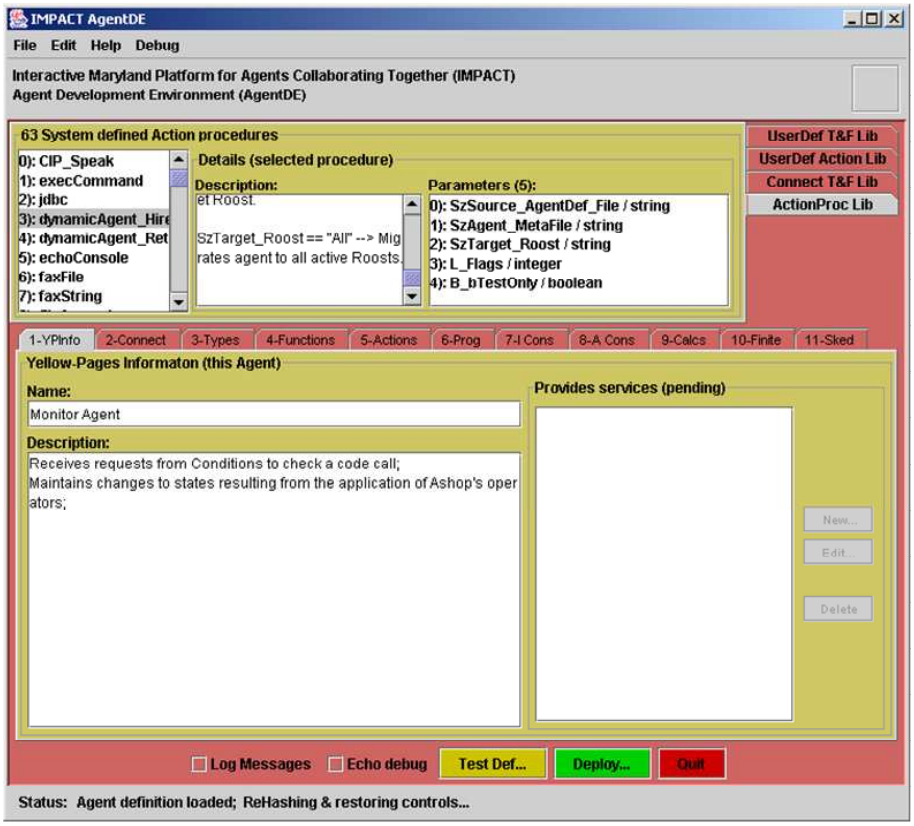

Figure 8: Actions for monitor

***IMPACT* Connection** The *IMPACT* connection library allows *IMPACT* agents to access third party platforms. The developer can define a connection alias and specific parameters for the target connection in the *AgentDE* Connection specification dialog. Figure 11 shows the *AgentDE* interface with the accepted Jilad connection definition, which taps a Hermes data mediator, through the remote Hermes interface accessed through the jilad.cs.umd.edu:8222 port. When a connection is established, *IMPACT* can execute code call over the data source and process the returned requests. Some currently implemented examples also include IBM Aglet, Oracle servers, ODBC (Open Database Connectivity), JDBC (Java Database Connectivity) and *CORBA* (Common Object Request Broker Architecture).

***IMPACT* Server** The *IMPACT* Server provides various services that are required by a group of agents as a whole. It supports the following services:

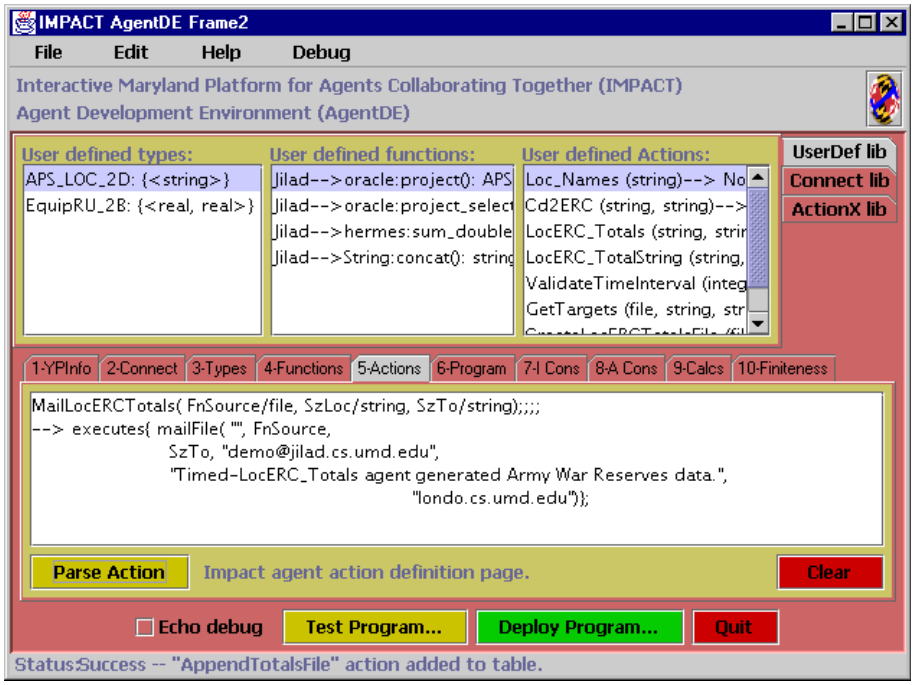

Figure 9: Actions in *AgentDE*

Registration Services: When the agent developer deploys an agent within the *IMPACT AgentDE*, it automatically provides her with the option of registering this agent with the registration server. The developer can register the services provided by the agent and also specify who can use those services.

Yellow Pages Services: the Yellow Pages Server can access the data structures created by the Registration Server. *IMPACT* agents can find the desired services by other agents via the Yellow Pages Server.

Type Services: Agent developers can specify the datatypes they use as well as the relationship between the newly created datatypes and other existing types within the *IMPACT* Type Server.

Thesaurus Server: This server receives requests when new agent services are being registered and when the *IMPACT* Yellow Pages Server is looking for agents providing a service.

Ontology Services: The *IMPACT* server is able to provide ontology services. An agent can reformulate its query in terms the other agent can understand.

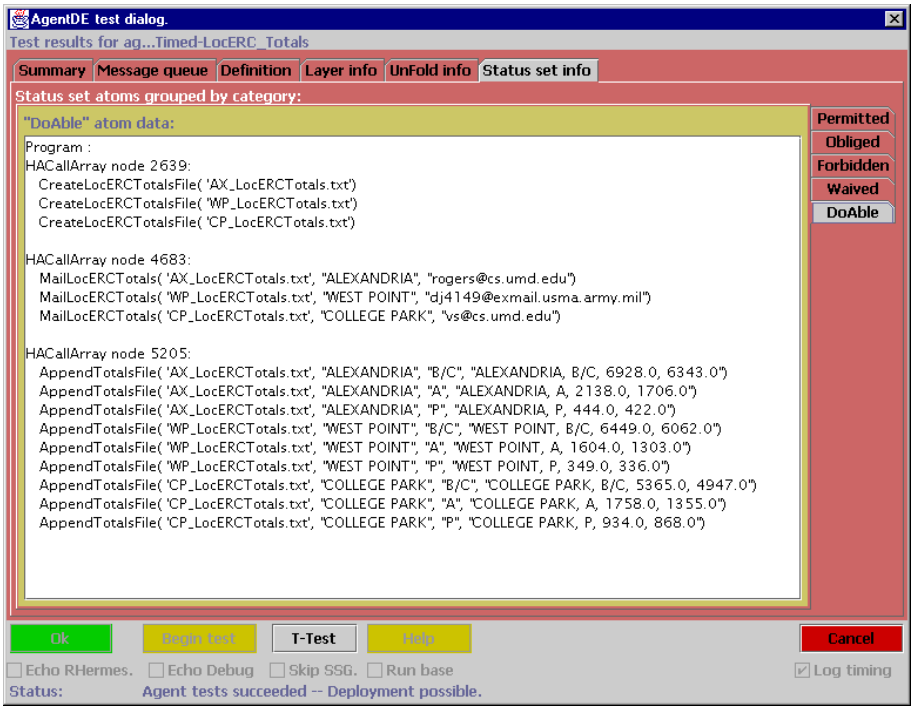

Figure 10: *AgentDE* Status Set Screen

Agent Roost An agent roost is a location where a set of deployed agents resides (Figure 12 shows the five agents in A-SHOP: the screen depicts the moment when the `codecallconditions` agent is active and sends a message to the `monitoring` agent). An agent roost serves as a duty officer since it manages all messages for this set of agents. Initially, all agents are inactive. When one of these agents receives a message, the agent roost includes it in this agent's message box and lets it run. If an agent sends out a message to another internal agent (i.e., an agent who is managed by the same roost), this message can be delivered by the roost in the same way. If the message is addressed to an external agent, the roost first contacts the *IMPACT* server to determine the location of the target agent. It then routes the message to the appropriate roost, which will pass it to the specified agent.

Agent Log The agent log allows an agent developer to maintain a record of agent communication and agent actions. The log supports log queries by content or time, and

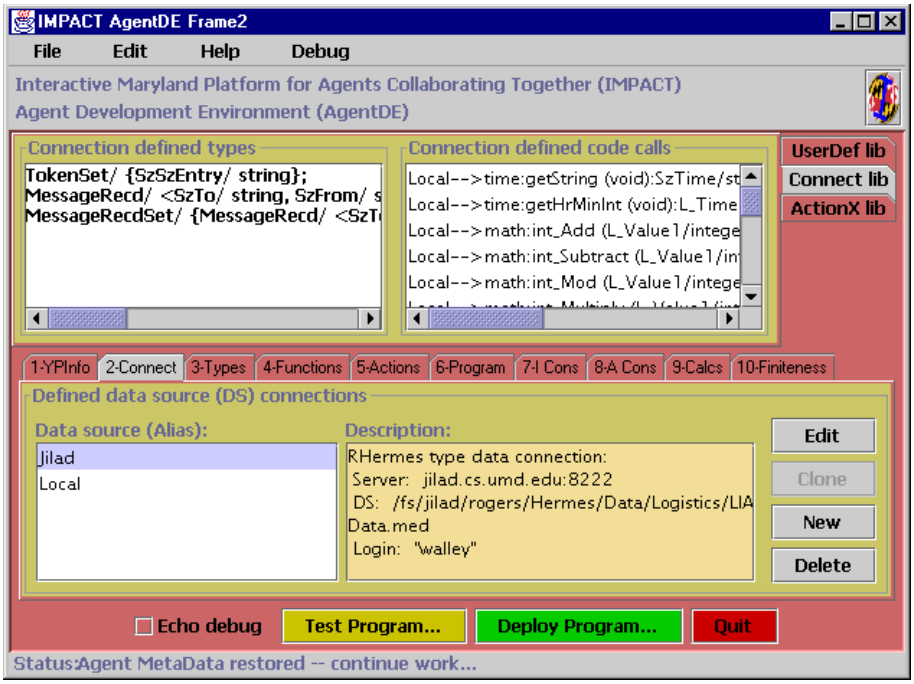

Figure 11: *AgentDE* Connect Library Screen

action browse, playback of video, text and image message objects. It can be used for many purposes such as record keeping, usage statistics, and it is essential for monitoring system performance and debugging.

3.2 Available tools and documentation

A tutorial about *IMPACT* can be found at `http://www.cs.umd.edu/projects/impact`. In particular, there is an *IMPACT* software library user documentation, which is available at `http://www.cs.umd.edu/projects/impact/Docs`, and includes: *(1)* implementation overview, *(2)* introduction of agent instantiation life cycle, *(3)* agent definition syntax, *(4)* sample agent development, and *(5)* selected user and developer code *API* JavaDocs.

3.3 Standards compliance, interoperability and portability

The implementation code consists of three main components: the *IMPACT AgentDE* (containing a series of compilers, written in Java, which render an agent instantiation from a given agent definition (text)); the *IMPACT* Yellow-pages server, written in Java

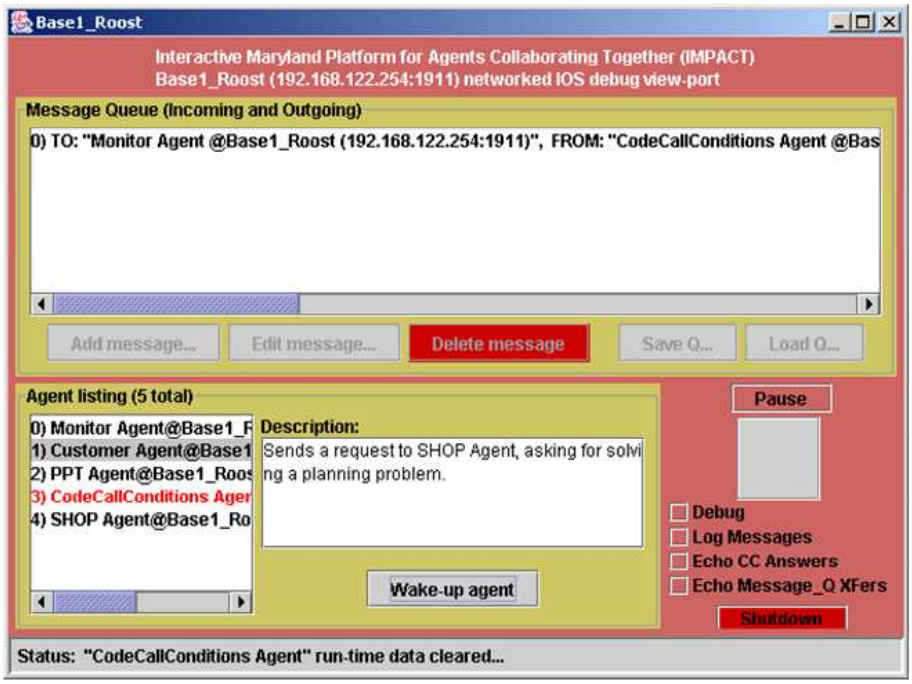

Figure 12: Agent Roost

and C, provides agent directory lookup services necessary for agent construction and run-time communication; the *IMPACT* Roost, written in Java, provides a run-time environment for *IMPACT* agents to work, sleep, or travel the network. Most of the implementation code is written currently compliant to the Java 1.2 specification. This provides maximal code portability across operating systems and platforms. It does, however, require loading Java 1.2 runtime library on the target platform. The existing implementation code libraries appear fairly generic. The code should prove readily adaptable to most micro-device environments through cross-compilation techniques. Some applications have been created to show the interoperability between *IMPACT* agents and IBM Aglets.

Future enhancements include an enhanced Roost network viewport for debugging global agent communities distributed across multiple roosts, and Java Jini enabled server front-ends to facilitate network configuration.

4 Applications supported by the language and the platform

The *IMPACT* project has built applications in the following areas:

1. US Army Logistics Integration Agency's "Virtual Operations Centre" involves the integration of a wide variety of distributed, heterogeneous databases, together with diverse alert, analysis and visualisation requirements.

2. US Army Research Laboratory's "Combat Information Processor" project where *IMPACT* is used to provide yellow pages matchmaking services, and is also providing alert mechanisms for multiple users with diverse battlefield monitoring requirements.

3. Aerospace applications where *IMPACT* technology has led to the development of a multi-agent solution to the "Controlled Flight into Terrain" problem which is the single largest cause of human fatalities in aircraft crashes (Washington Post, Feb. 7, 1998).

4. US Army STRICOM's JANUS project where *IMPACT* technology is used to analyse massive amounts of simulation data.

5. Coordinated route and flight planning applications over free terrain.

New applications in the banking and finance sector are under consideration. In addition, *IMPACT* has been used for student projects in academia, including University of Maryland, Technical University of Vienna, The University of Manchester, and Clausthal Institute of Technology.

5 Final Remarks

IMPACT has been started by VS Subrahmanian in 1997 and its core has been developed in a series of papers [1, 19, 17, 18] and also in a book [24].

References

[1] K. Arisha, F. Ozcan, R. Ross, V.S. Subrahmanian, T. Eiter, and S. Kraus. IMPACT: A Platform for Collaborating Agents. *IEEE Intelligent Systems*, 14:64–72, March/April 1999.

[2] Cattell, R. G. G., et al., editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1997.

[3] Jürgen Dix. A Computational Logic Approach to Heterogenous Agent Systems. In Th. Eiter, M. Truszczyński, and W. Faber, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Sixth International Conference*, LNCS 2173, pages 1–21, Berlin, September 2001. Springer.

[4] Jürgen Dix and Thomas Eiter. Theoretical foundations and practical applications of heterogenous agent systems. Technical report, Working Notes of the *14th annual European Summer School in Logic, Language and Information*, Trento 2002, 2002.

[5] Jürgen Dix, Thomas Eiter, Michael Fink, Axel Polleres, and Yingqian Zhang. Monitoring Agents using Declarative Planning. *Fundamenta Informaticae*, 57(2–4):345–370, 2003.

[6] Jürgen Dix, Sarit Kraus, and V.S. Subrahmanian. Temporal agent reasoning. *Artificial Intelligence*, 127(1):87–135, 2001.

[7] Jürgen Dix, Sarit Kraus, and V.S. Subrahmanian. Agents dealing with time and uncertainty. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 912–919. New York: ACM Press, July 2002.

[8] Jürgen Dix, Sarit Kraus, and V.S. Subrahmanian. Heterogenous temporal probabilistic agents. *ACM Transactions of Computational Intelligence*, 2005. to appear.

[9] Jürgen Dix, Ugur Kuter, and Dana Nau. Planning in answer set programming using ordered task decomposition. In R. Kruse, editor, *Proceedings of the 27th German Annual Conference on Artificial Intelligence (KI '03), Hamburg, Germany*, LNAI 2821, pages 490–504, Berlin, 2003. Springer.

[10] Jürgen Dix and João Alexandre Leite, editors. *Computational Logic in Multi-Agent Systems, 4th International Workshop, CLIMA IV, Fort Lauderdale, FL, USA, January 6-7, 2004, Revised Selected and Invited Papers*, volume 3259 of *Lecture Notes in Computer Science*. Springer, 2004.

[11] Jürgen Dix, Hector Munoz-Avila, and Dana Nau an Lingling Zhang. Theoretical and Empirical Aspects of a Planner in a Multi-Agent Environment. In Giovambattista Ianni and Sergio Flesca, editors, *Proceedings of Journees Europeens de la Logique en Intelligence artificielle (JELIA '02)*, LNCS 2424, pages 173–185. Springer, 2002.

[12] Jürgen Dix, Hector Munoz-Avila, Dana Nau, and Lingling Zhang. Planning in a multi-agent environment: Theory and practice. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 944–945. New York: ACM Press, July 2002.

[13] Jürgen Dix, Hector Munoz-Avila, Dana Nau, and Lingling Zhang. IMPACTing SHOP: Putting an AI planner into a Multi-Agent Environment. *Annals of Mathematics and AI*, 37(4):381–407, 2003.

[14] Jürgen Dix, Mirco Nanni, and V. S. Subrahmanian. Probabilistic agent reasoning. *ACM Transactions of Computational Logic*, 1(2):201–245, 2000.

[15] Jürgen Dix, Fatma Öczan, and V.S. Subrahmanian. Improving performance of heavily loaded agents. *Annals of Math and AI*, 41(2-4):339–395, 2004.

[16] Jürgen Dix, V.S. Subrahmanian, and George Pick. Meta Agent Programs. *Journal of Logic Programming*, 46(1-2):1–60, 2000.

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

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

[19] Thomas Eiter, V.S. Subrahmanian, and Georg Pick. Heterogeneous Active Agents, I: Semantics. *Artificial Intelligence*, 108(1-2):179–255, 1999.

[20] H. Muñoz-Avila, D.W. Aha, D.S. Nau, R. Weber, L. Breslow, and F. Yaman. SiN: Integrating case-based reasoning with task decomposition. In *IJCAI-2001*, August 2001.

[21] D.S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of IJCAI-99*, 1999.

[22] J. Siegal. *CORBA Fundementals and Programming*. John Wiley & Sons, New York, 1996.

[23] V. S. Subrahmanian, Sarit Kraus, and Yingqian Zhang. Distributed algorithms for dynamic survivability of multiagent systems. In Dix and Leite [10], pages 1–15.

[24] V.S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Özcan, and Robert Ross. *Heterogenous Active Agents*. MIT-Press, 2000.

[25] Lingling Zhang. Documentation for ASHOP 1.0. Technical Report CSC-TR 2102, University of Maryland, 2002. Master Thesis.

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Wojciech Jamroga
Contact: wjamroga@in.tu-clausthal.de
URL: http://www.in.tu-clausthal.de/~wjamroga/techreports/

The IfI Review Board