IMPACT Implementation overview.

1.0 Introduction: The current IMPACT implementation provides all components necessary to build and test one or more IMPACT agents "housed" in run-time environment wrappers we call the "Roosts". Briefly speaking, roosts come in two flavors; A local roost wraps a single collaborative community of agents in a non-networked "sand-box" environment (agents can only work with other agents within this one community). A networked roost, on-the-other-hand, forms the back-bone for a widely distributed global community of agents (agents on one roost can interact with agents on other roosts – spanning operating systems and hardware platforms).

The implementation code consists of three main components; The IMPACT agent development environment (AgentDE) contains a series of compilers, written in *Java*, which render an agent instantiation (an collection of IMPACT agent core class objects) from a given agent definition (text). The IMPACT Yellow-pages server (YPServer), written in *Java* and C, provides agent directory lookup services necessary for agent construction and run-time communications. The IMPACT Roost, written in *Java*, provides a run-time environment for IMPACT agents to work, sleep, or travel the networks. Most of the implementation code, contained in a 2 megabyte *Java* archive file (JAR) is written currently 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 (6+ MB on Windows NT).

Future revisions will include *Java JINI* support, the addition of caching and multi-query optimization techniques, and enhanced error handling and correction methods. Of note, pushing the communications layer into the *Jini* framework requires loading the *Jini* runtime library which further expands the minimum library footprint beyond the scope of some smaller platforms.

1.1 IMPACT agent architecture: An IMPACT agent definition includes a list of one or more action definitions, with inter-action constraints, which prescribe the agents behavior by mapping the action result data (the action status atom set) to procedures which ultimately change its' state (the data out component). At a deeper level, action determination runs according to the agent program which lists sets of rules (connection function code-calls and comparisons) evaluated over raw or sensor data, application or server request output.

The AgentDE, in short, allows the developer to compile agent definitions into *Java* object collections (each wrapped by a `Thread` class object) referred to here as agent instantiations; Every agent instantiation executes in it's own process thread and contains all the code (*Java* objects) necessary to perform its' defined behavior.

Agent instantiations typically "sleep" until awakened to process incoming message traffic. They may also, however, be designed to run continuously (with periodic sleep intervals). A pending enhancement will also allow the developer to specify a non-periodic run-time schedules.

1.1.1 IMPACT connections: As mentioned above, IMPACT agents commonly exploit local and networked assets through code-calls. To enhance development extensibility we developed the IMPACT connection module, a *Java* base-class, from which all connection "classes" extend. Similar to the notion of Dynamic Link Libraries common on some platforms, IMPACT connections allow agent programs to load/use (at run-time) only the code necessary to access the desired servers, frameworks, and method libraries. This keeps the execution foot-print small and the agent architecture highly flexible.

IMPACT connection classes typically define either a communications interface for external servers, or a code "wrapper" which provides function calls (an API) to some third-party program library or code. Our beta release connection library provides a few examples as we defined connection interfaces for generic *JDBC*, *Hermes* (a legacy data mediation system), the *IBM Aglets* framework, and some local programs which handle basic agent messaging, math, and time method code-calls.

Extending the connection library is relatively straight forward as the agent developer can create a new connection module by "extending" a new class from the connection base class (`IAD_Connect.java`). They would then encode a hand-full of required methods in the new connection module class definition, compile it, and ultimately copy the new created connection module somewhere in the java system path (`CLASSPATH`). On boot-up, IMPACT explores the archives listed by the system `CLASSPATH` refererence and notes all such connection extension classes for use in the AgentDE session.

1.1.2 IMPACT action procedures (`IAD_ActionExe` class extensions): Just as we provided an extendable library architecture for connections (typically data input), so too we defined an extendable library for output action procedures; These `IAD_ActionExe` classes provide the mechanism by which an agent may change its state (typically data updates and prompts to other processes and/or agents). Once an agent instantiation has calculated and approved its "To Do" list (the action status atom set), The agent can then act on the generated to-do data by mapping it to and executing the appropriate action procedures. The beta release `IAD_ActionExe` library includes methods such as `createFile`, `appendFile`, `mailFile`, `faxFile`, and `sendMessage`.

1.1.3 The agent instantiation life-cycle: We represent an agent here in Image 1.1 as a complex grouping of geometric objects which represent the underlying reality; An IMPACT agent instantiation is a complex grouping of *Java* class objects, a hierarchy, wrapped by the `IAD_MetaData` class into self-executing *Java* threads -- Every agent definition contains all the code (instantiated classes) necessary to execute independently while performing its designated behavior.

On the right side of Image 1.1 we see asset information (usually simple data) coming into the Agent composition through the connection modules. The agent then evaluates its current state by processing the underlying action rule body code-calls through which it generates "to-do" list entries (the action status atoms). The agent then checks its "to-do" list against constraint definitions (as applicable) and renders the "DoAble" action set. Finally the agent executes the "DoAble"actions, calling action procedures as specified in the corresponding action definitions – The answers flow out the left side of the agent composition diagram.

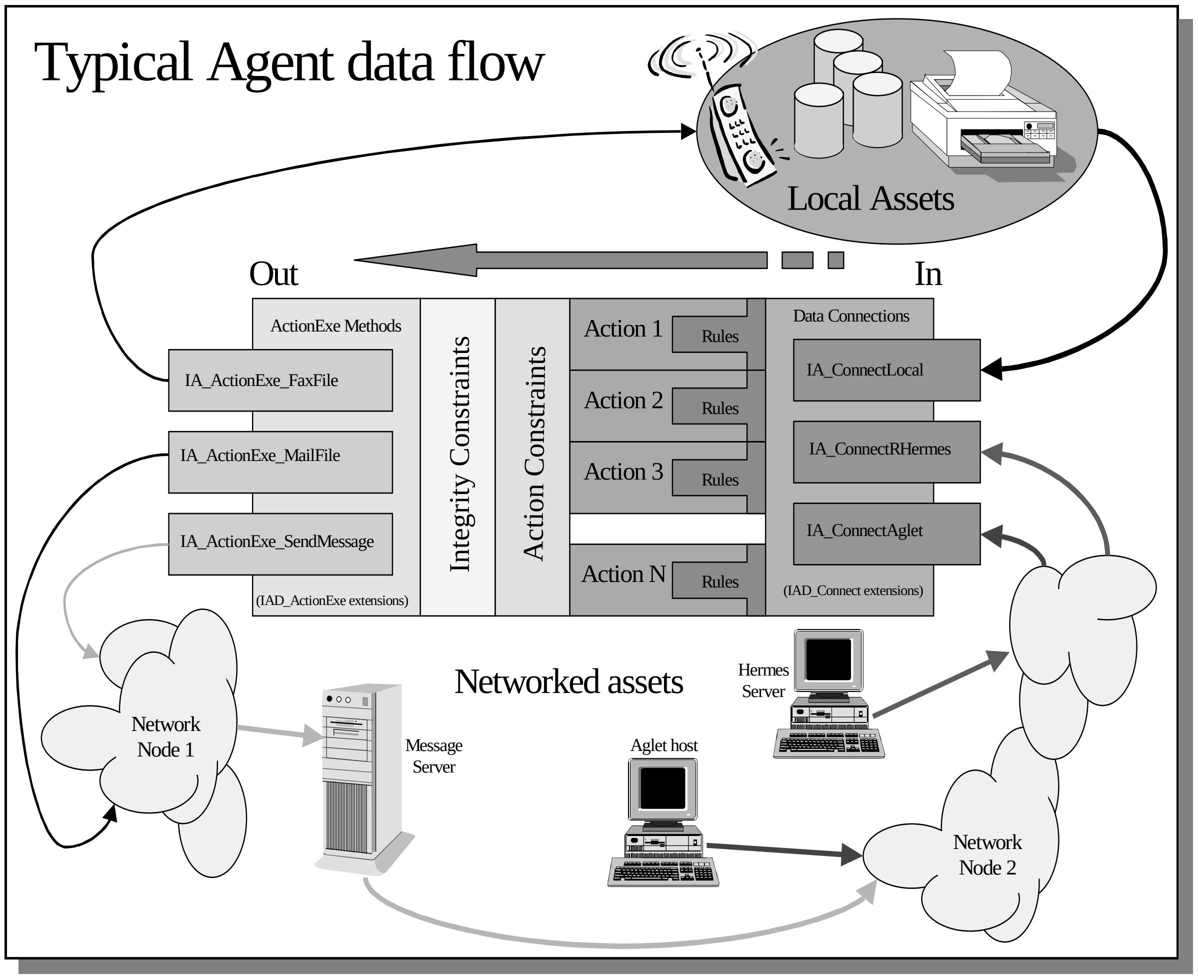


Image 1.1: Agent data flow.

1.2 IMPACT agent community “roost” implementation: An IMPACT agent roost (`IRR_RoostServer` class) comprises a collection of IMPACT agent objects which communicate through an internal message queue (`IA_MessageQueue` class) managed by a “duty officer” (`IR_DutyOfficer` class).

The roost duty officer functions primarily as a communications daemon for the local agents; the duty officer routes incoming messages to the queue, wakes agents for whom pending messages exist, and pulls out-going messages from the queue for remote routing. The duty officer, and thus the roost, remains active until commanded to shut-down.

Agents, by design, are independent. While the duty officer may prompt an agent to work (waking them), the officer does not control agents’ activities -- agents are free to sleep or run, come, and go, according to their design specification. As mentioned previously, each agent has everything it needs to perform its designated activity. If it requires access to some data, application, server, sensor, or actuator asset, then it includes the necessary connection specifications which provide proper asset usage. For example, image 1.2 shows agents one and two accessing local assets while agents three and four exploit remote resources via the network (Local or Internet).

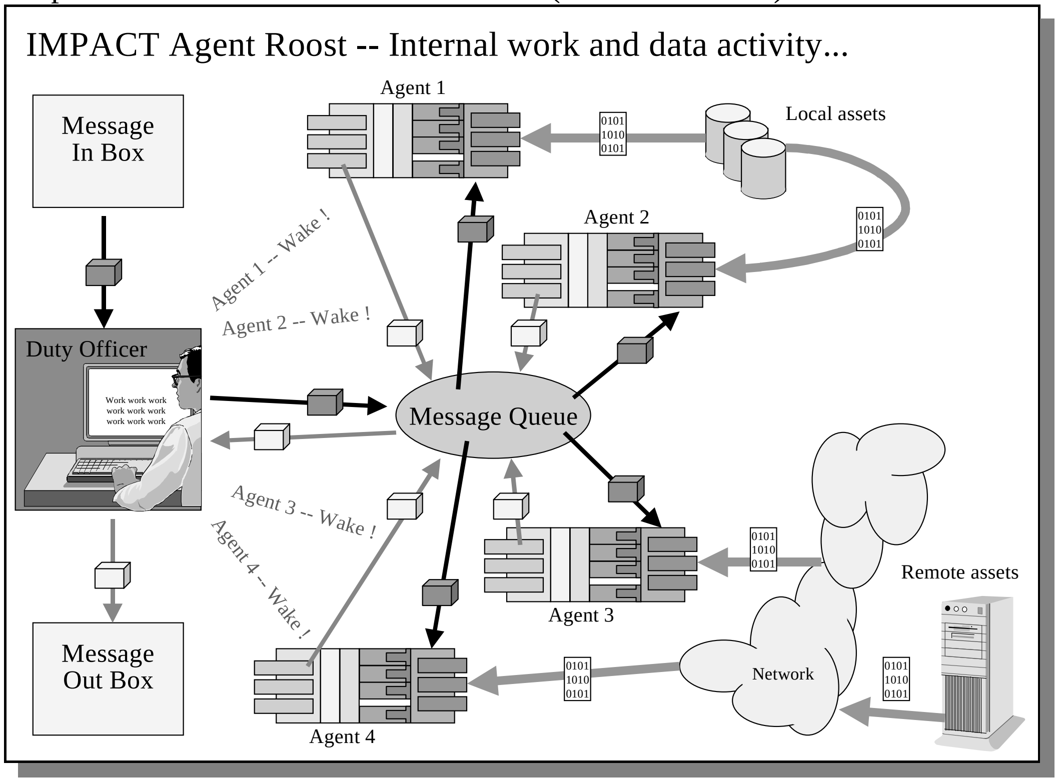


Image 1.2. Roost data flow.

Note that the communication “boxes” in Image 1.2 denote IMPACT agent message objects passed between agents through the message queue both in and out of the roost. The dark green “boxes” denote input messages while the light yellow boxes denote output messages. The asset data streams are tagged with rectangles containing zeros and ones to denote the binary formats native to the assets.

1.3 A global IMPACT agent community implementation, a network of roosts: The `IRR_RoostServer` and `IYR_YPServer` classes provide the communications layer necessary for Inter-Roost agent collaboration. This enables agents to collaborate directly with agents located on remote roosts, distributed across network connections (perhaps globally), spanning diverse operating systems; In short, agents running under *Unix* can work directly with agents deployed on other operating systems such as *Windows NT*, *MacOS*, and (hopefully soon) *PalmOS*.

As mentioned previously in section 1.2, the roost duty officer directs message traffic within their roost. Additionally, duty officers designate routing for newly generated message traffic (IMPACT agents, by design, do not know where the other agents are roosted) through calls to the IMPACT yellow-pages server. Thus in figure three we see the duty officer querying the Yellow-Pages server for LogTads agent location information. The resulting address denotes a remote roosting and so the applicable messages are placed in the "Out-box" and redistributed via standard *Java* Remote Method Invocation (RMI) procedures to the target roost.

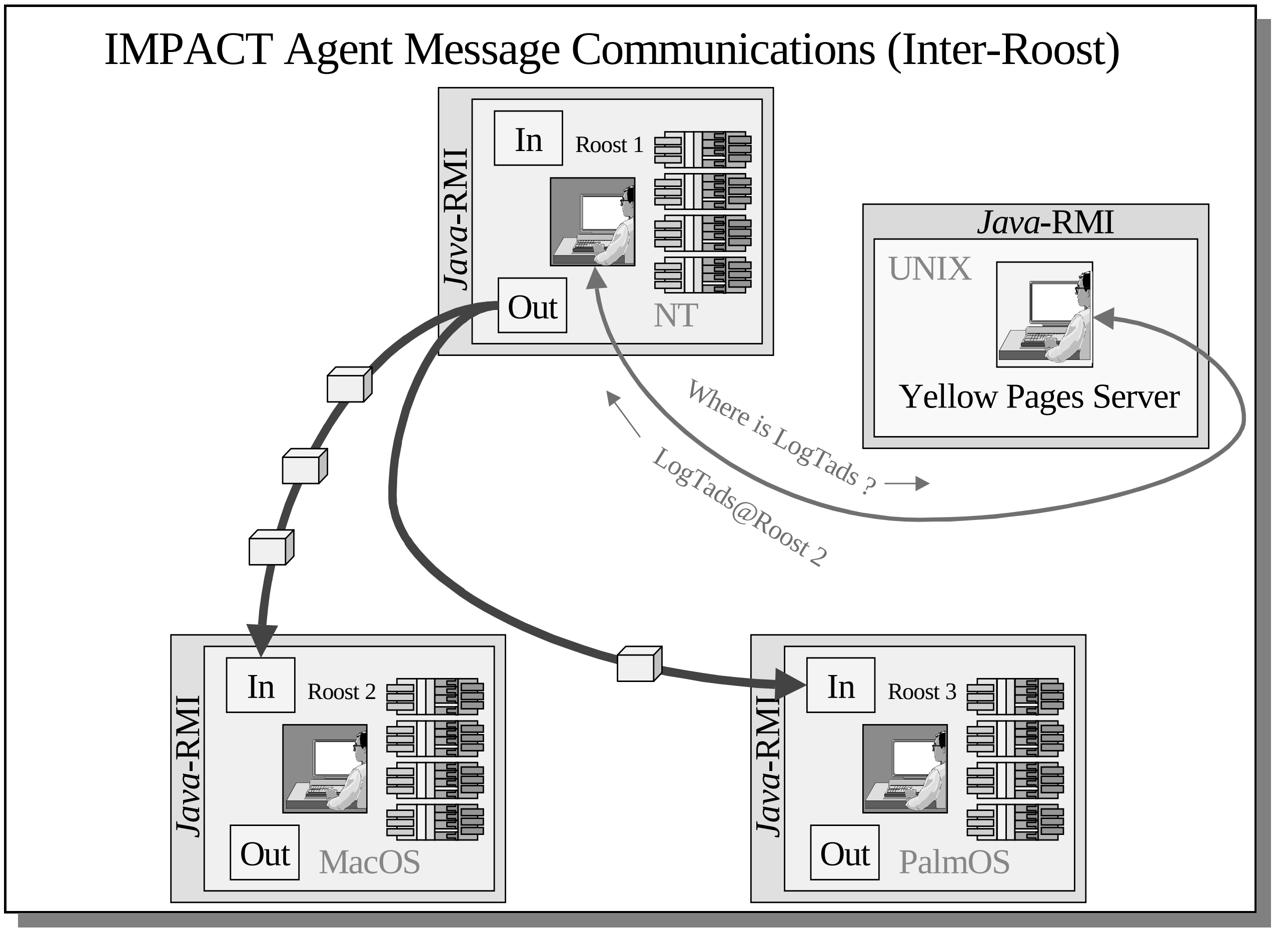
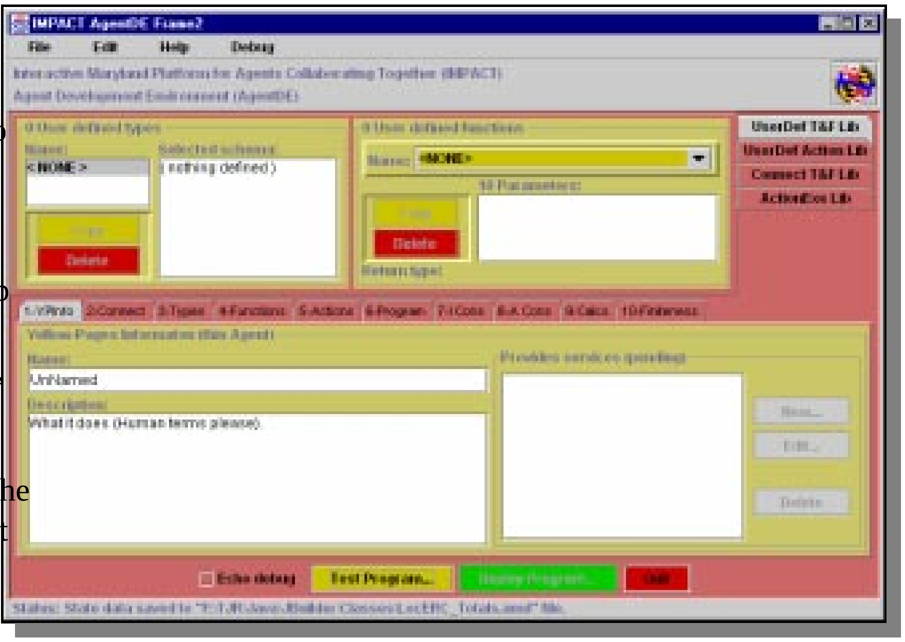


Image 1.3. Inter-Roost agent communications.

1.4 The IMPACT agent development environment (AgentDE): The (beta) AgentDE interface provides all the controls necessary to define and test a single IMPACT agent. It employs a series of compilers through which the developer parses the various agent component definitions (type, function, action, etc.) into the appropriate *Java* class objects. This interface groups sub-controls under two sets of tab-select panels. The upper tab set (containing UserDef T&D, UserDef Action, Connect T&D, and ActionExe Lib tabs) allows the developer to view the definitions already defined for the current agent. The lower tab set (containing YPInfo, Connect, Type, Function, Action, Program, I-Cons, A-Cons, Calcs, and Finiteness tabs) provide access to the corresponding definition input controls and parsers.


IMPACT AgentDE Frame2

Interactive Maryland Platform for Agents Collaborating Together (IMPACT)
Agent Development Environment (AgentDE)

Name: <NONE>
Selected schema: (nothing defined)
Name: <NONE>

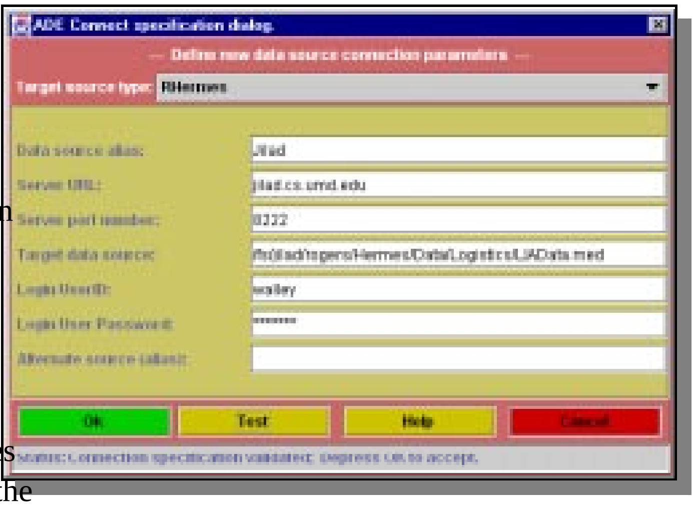
Name: UnNamed
Description: What it does (Human terms please)

Status: State data saved to "F:\JBuilder Classes\LocERC_Totals.amd" file.

Image 1.4 – The initial AgentDE frame..

1.4.1 Defining the agent, a quick walk-thru: An agent developer typically starts by defining the target agent yellow-pages information – select the lower "1-YPInfo" tab and then enter the agents name and a brief description.

Next the developer must select the appropriate asset connection modules (as necessary) – select the lower "2-Connect" tab to bring the connection control page forward, and then click on the "New" button located to the far-right. This launches the AgentDE Connection specification dialog (image 1.5) where a developer defines a connection alias (required) and specific parameters for the target asset connection. Please note that the "target source type" uses drop-down list selection control. Clicking the mouse button on the control causes the list to drop-down, allowing selection from the defined IMPACT connection library. Image 1.5 shows a connection named "Jilad", which taps a *Hermes* data mediator, the LIAData.med mediator file, through the remote *Hermes* interface accessed through the jilad.cs.umd.edu:8222 port. Image 1.6 shows the AgentDE interface with the accepted Jilad connection definition.


ADE Connect specification dialog

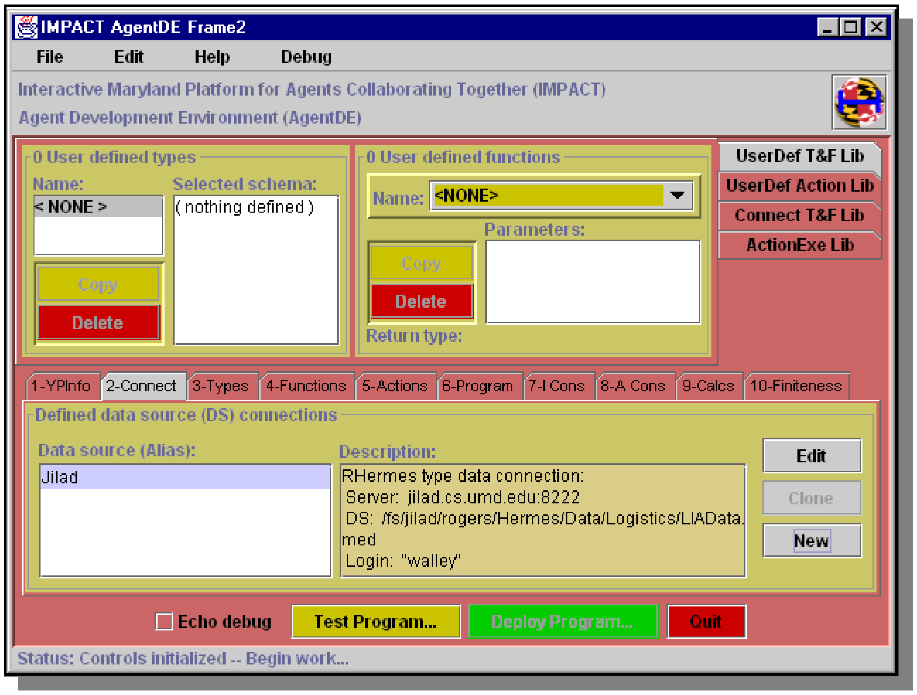
— Define new data source connection parameters —

Target source type: RHermes

Data source alias:	Jilad
Server URL:	jilad.cs.umd.edu
Server port number:	8222
Target data source:	/fs/jilad/rogers/Hermes/Data/Logistics/LIAData.med
Login UserID:	walley
Login User Password:	*******
Alternate source (alias):	

Status: Connection specification validated; press OK to accept.

Image 1.5: AgentDE Connect specification dialog.



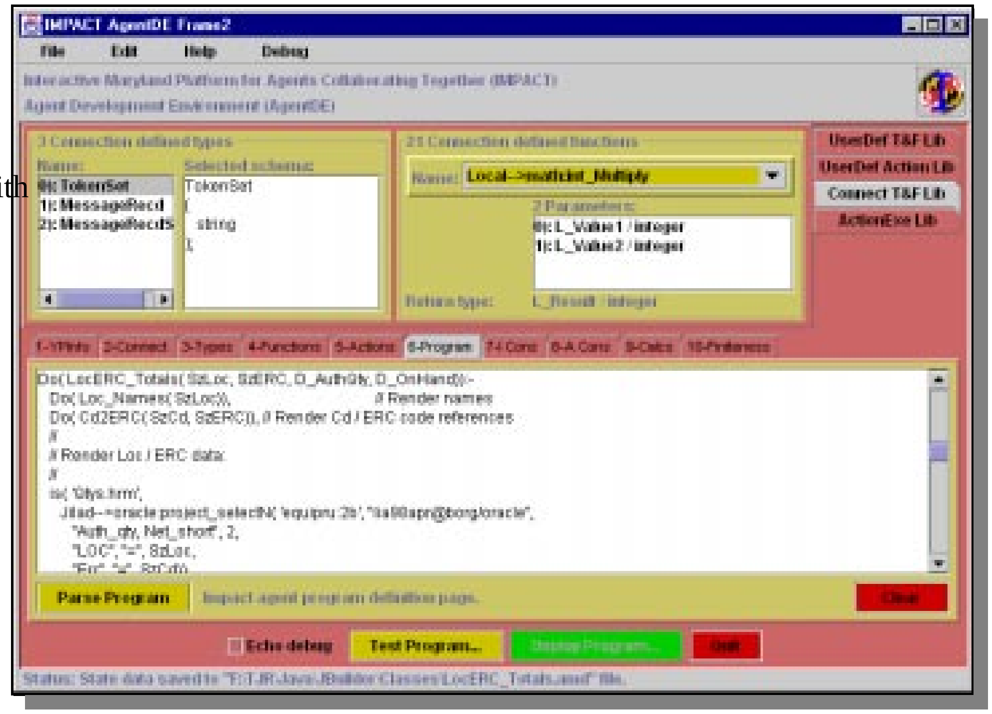
Jilad

```
RHermes type data connection:
 Server:  jilad.cs.umd.edu:8222
 DS:  /fs/jilad/rogers/Hermes/Data/Logistics/LIAData.
med
 Login:  "walley"
```

Status: Controls initialized -- Begin work...

Image 1.6: AgentDE showing the accepted connection definition..

Next the developer defines agent types, functions, actions, integrity constraints, and action constraints as necessary: The control sets for defining each of these agent components all work roughly the same. The developer enters the definition text into the text area and then clicks on the parse button. If the text parses correctly, then it becomes part of the agent definition. If the text fails to parse, then an error dialog will appear with the corresponding error text. Image 1.7 shows the program definition control with an accepted agent program definition.



0: TokenSet
1: MessageRecd
2: MessageRecdS

```
TokenSet
{
  string
};
```

0: L_Value1 / integer
1: L_Value2 / integer

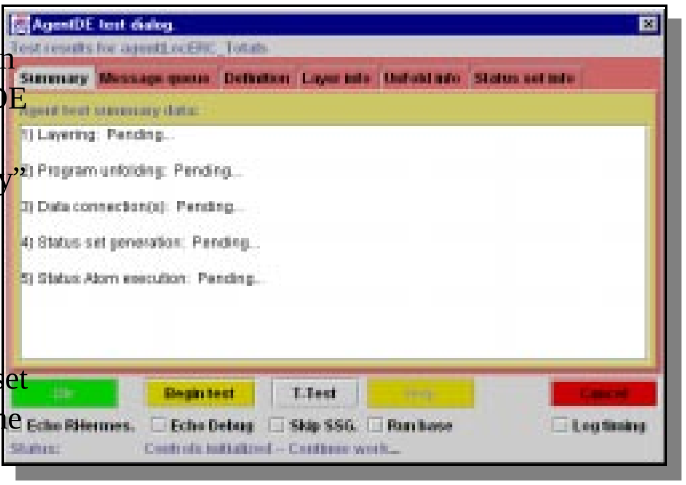
L_Result / integer

```
Do( LocERC_Totals( SzLoc, SzERC, D_AuthQty, D_OnHand)):-
  Do( Loc_Names( SzLoc)),                         // Render names
  Do( Cd2ERC( SzCd, SzERC)), // Render Cd / ERC code references
  //
  // Render Loc / ERC data:
  //
  in( 'Qtys.htm',
    Jilad-->oracle:project_selectN( 'equipru 2b', "lia98apr@borg/oracle",
      "Auth_qty, Net_short", 2,
      "LOC", "=", SzLoc,
      "Erc", "=", SzCd),
```

Status: State data saved to "F:\TJR\Java\JBuilder\Classes\LocERC_Totals.aed" file.

Image 1.7: AgentDE showing the program definition control page.

Following successful agent definition, the developer moves on to the testing phase. Clicking the cursor on the "Test Program" button causes the AgentDE to launch a test dialog as shown in image 1.8. Similar to the primary AgentDE interface, the test dialog is organized as selectable control groupings. The "Summary" group displays a running summary of the current test status. The "Message Queue" control set allows manipulation of the local message queue for testing inter-agent communications. The "Definition" control set displays the current agent definition text. The "Layer Info" set displays the rule layering information. The "Unfold Info" set displays the agent program unfolding, similar perhaps to in-line method substitutions, imposed to improve run-time efficiency. Selecting the "Status Set Info" tab displays another tab-control subgrouping, shown in image 1.10, which displays the rendered action status set (in effect, the "to-do" list).

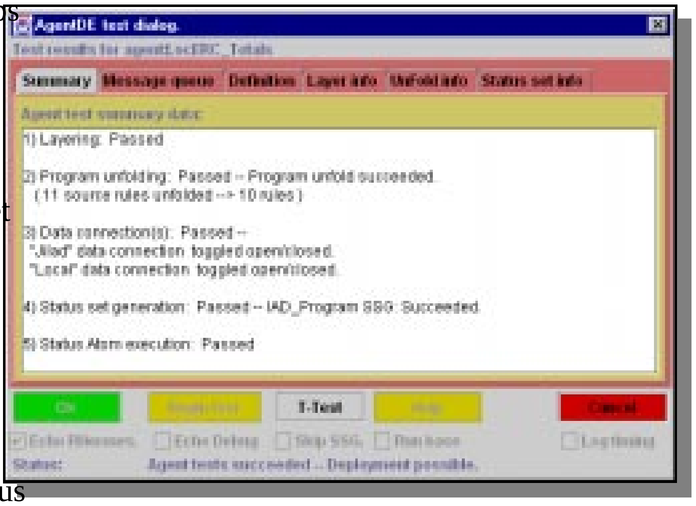


```
1) Layering:  Pending...

2) Program unfolding:  Pending...

3) Data connection(s):  Pending...

4) Status set generation:  Pending...

5) Status Atom execution:  Pending...
```

```
Status:          Controls initialized -- Continue work...
```

Image 1.8: The initial AgentDE test dialog.

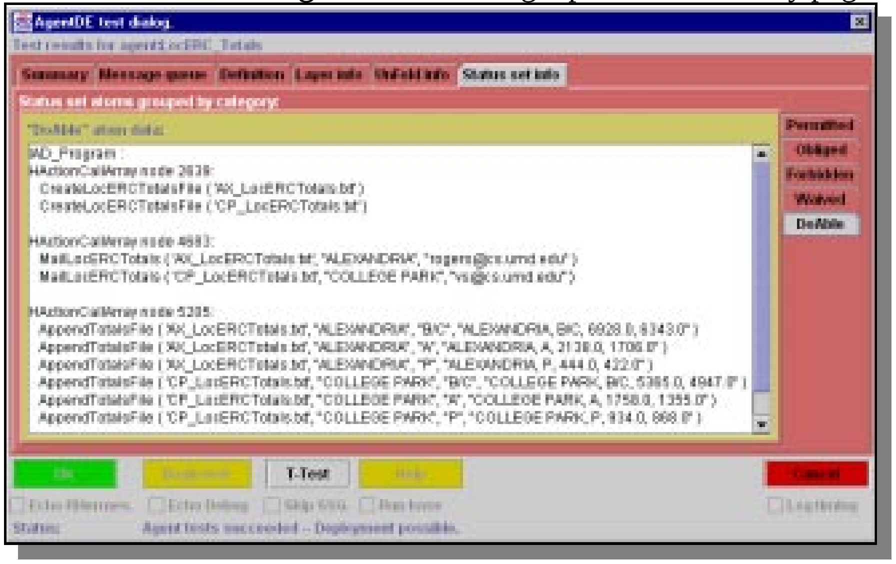
Selecting the "Begin Test" button causes the agent execution testing to begin. Image 1.9 shows the dialog summary page following a successful test execution. Image 1.10 shows the "DoAble" action status atom set following a successful agent execution. The "DoAble" set corresponds to actions the agent determined it should do at run-time.



```
1) Layering:  Passed

2) Program unfolding:  Passed -- Program unfold succeeded.
  (11 source rules unfolded --> 10 rules )

3) Data connection(s):  Passed --
 "Jilad" data connection  toggled open/closed.
 "Local" data connection  toggled open/closed.

4) Status set generation:  Passed -- IAD_Program SSG: Succeeded.

5) Status Atom execution:  Passed
```

```
Status:          Agent tests succeeded -- Deployment possible.
```

Image 1.9: Test dialog – post-test summary page.

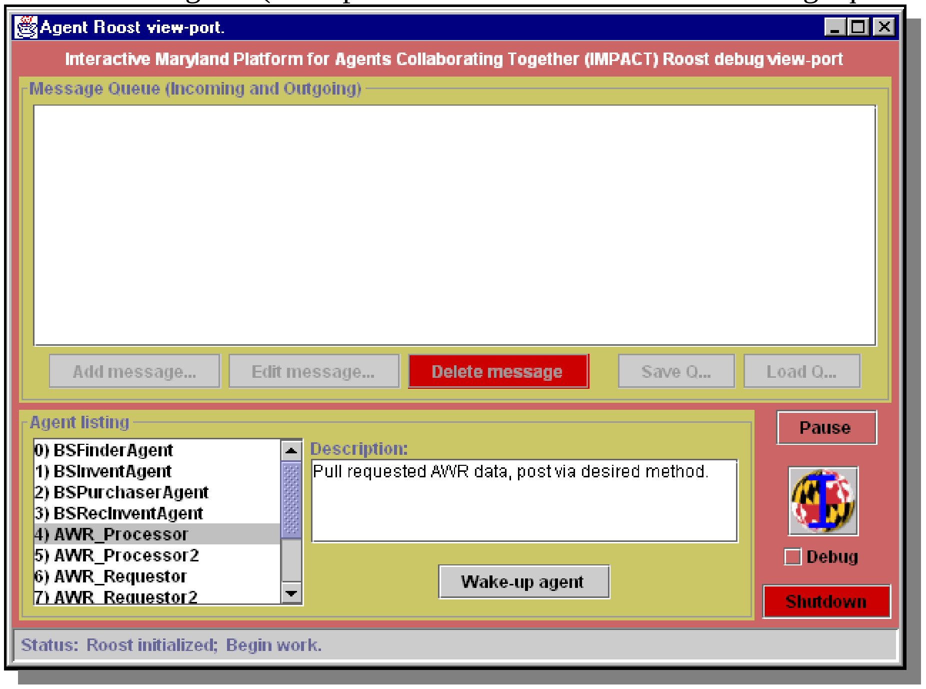


```
IAD_Program :
HActionCallArray node 2636:
  CreateLocERCTotalsFile ( 'AX_LocERCTotals.txt' )
  CreateLocERCTotalsFile ( 'CP_LocERCTotals.txt' )

HActionCallArray node 4683:
  MailLocERCTotals ( 'AX_LocERCTotals.txt', "ALEXANDRIA", "rogers@cs.umd.edu" )
  MailLocERCTotals ( 'CP_LocERCTotals.txt', "COLLEGE PARK", "vs@cs.umd.edu" )

HActionCallArray node 5205:
  AppendTotalsFile ( 'AX_LocERCTotals.txt', "ALEXANDRIA", "B/C", "ALEXANDRIA, B/C, 6928.0, 6343.0" )
  AppendTotalsFile ( 'AX_LocERCTotals.txt', "ALEXANDRIA", "A", "ALEXANDRIA, A, 2138.0, 1706.0" )
  AppendTotalsFile ( 'AX_LocERCTotals.txt', "ALEXANDRIA", "P", "ALEXANDRIA, P, 444.0, 422.0" )
  AppendTotalsFile ( 'CP_LocERCTotals.txt', "COLLEGE PARK", "B/C", "COLLEGE PARK, B/C, 5365.0, 4947.0" )
  AppendTotalsFile ( 'CP_LocERCTotals.txt', "COLLEGE PARK", "A", "COLLEGE PARK, A, 1758.0, 1355.0" )
  AppendTotalsFile ( 'CP_LocERCTotals.txt', "COLLEGE PARK", "P", "COLLEGE PARK, P, 934.0, 868.0" )
```

```
Status:          Agent tests succeeded -- Deployment possible.
```

Image 1.10: Post-Test "DoAble" status set.

1.5 IMPACT Agent roost interface: In a normal agent community, the roost runs seamlessly in the background without an interface. For testing and demo purposes, however, we developed a roost front-end, displayed in image 1.11, which allows us to view the on-going agent activities. This simple interface provides a list of available agents, the ability to wake sleeping agents, and to view message traffic generated between agents (the top window reflects the current message queue contents).


Agent Roost view-port.
Interactive Maryland Platform for Agents Collaborating Together (IMPACT) Roost debug view-port
Message Queue (Incoming and Outgoing)
Add message...
Edit message...
Delete message
Save Q...
Load Q...
Agent listing
Pause
Debug
Wake-up agent
Shutdown

0) BSFinderAgent
1) BSInventAgent
2) BSPurchaserAgent
3) BSRecInventAgent
4) AWR_Processor
5) AWR_Processor2
6) AWR_Requestor
7) AWR_Requestor2

Description:
Pull requested AWR data, post via desired method.

Status: Roost initialized; Begin work.

Image 1.11: The Agent Roost (debug) interface.

1.6 Future enhancements: Time allowing, we plan to include the following:

- Performance optimizations through caching and incremental update methods.
- An enhanced GUI ("drag&drop") AgentDE interface.
- An enhanced Roost network viewport for debugging global agent communities distributed across multiple roosts.
- *Java JINI* enabled server front-ends to ease network configuration issues.
- Improved "safety" checks applied during the compilation process (error detection and handling algorithms)

1.7 Targeting micro platforms, the future is near: Software installation requirements push the current implementation beyond the scope of most palm-top platforms. The (pending) arrival of embedded java devices, however, would quickly rectify this problem – *Jini* IMPACT agents could then reside directly, as mobile agents, on palm-top devices.

This aside, our existing implementation code libraries appear fairly generic – Our code should prove readily adaptable to most micro-device environments through cross-compilation techniques.