# A Platform for Agent Behavior Design and Multi Agent Orchestration *

G. B. Laleci, Y. Kabak, A. Dogac, I. Cingil, S. Kirbas, A. Yildiz, S. Sinir, O. Ozdikis, O. Ozturk

Software Research and Development Center &
Dept. of Computer Eng.
Middle East Technical University (METU)
06531 Ankara Türkiye
+90 312 2105598
asuman@srdc.metu.edu.tr

**Abstract.** Agents show considerable promise as a new paradigm for software development. However for wider adoption and deployment of agent technology, powerful design and development tools are needed. Such tools should empower software developers to cater agent solutions more efficiently and at a lower cost for their customers with rapidly changing requirements and differing application specifications.
In this paper, an agent orchestration platform that allows the developers to design a complete agent-based scenario through graphical user interfaces is presented. The scenario produced by the platform is a rule based system in contrast to the existing systems where agents are coded through a programming language. In this way, the platform provides a higher level of abstraction to agent development making it easier to adapt to rapidly changing user requirements or differing software specifications. The system is highly transportable and interoperable.
The platform helps to design a multi-agent system either from scratch, or by adapting existing distributed systems to multi agent systems. It contains tools that handle the agent system design both at the macro level, that is, defining the interaction between agents and at the micro level which deals with internal design of agents.
Agent behaviour is modeled as a workflow of basic agent behaviour building blocks (such as receiving a message, invoking an application, making a decision or sending a message) by considering the data and control dependencies among them, and a graphical user interface is provided to construct agent behaviours. The platform allows agent templates to be constructed from previously defined behaviours. Finally through a Scenario Design Tool, a multi-agent system is designed by specifying associations among agents. The scenario is stored in a knowledge base by using the Agent Behaviour Representation Language (ABRL) which is developed for this purpose. Finally to be able to demonstrate the execution of the system on a concrete agent platform, we mapped the ABRL rules to JESS and executed the system on JADE.

* This work is supported by the European Commission's IST Programme, under the contract IST-2000-31050 Agent Academy.

## 1 Introduction

In the recent years agent technology has found many interesting applications in e-commerce, decision support systems and Internet applications. An increasing number of computer systems are being viewed in terms of autonomous agents [7]. They have proven particularly useful in business and production scenarios where they have facilitated the buying and selling of goods and services in electronic marketplaces, handling workflows, helping with personalization by managing user profiles or by tackling production planning. As the benefits of using agent societies in such applications become clear so does the need for developing high-level agent system building tools and frameworks.

Agent based approaches enable the development of increasingly powerful and complex distributed systems, since they provide a natural way to define high level of abstractions. The problems can be decomposed in terms of autonomous agents that can engage in flexible, high level interactions [7, **?**]. However the design and development of multi-agent-systems are not straight forward and these issues are among the main research areas in multi-agent systems [5]. There are considerable amount of work on designing a multi-agent system at a macro level, i.e., defining the interactions between the agents; however it is still cumbersome to design and develop multi-agent systems at a micro-level which deals with internal details of agents.

The aim of the work described in this paper is to abstract the application development from the detailed agent coding, and thus enabling the user to construct a multi-agent system by making use of her existing applications and by providing her an extensive and powerful design platform.

The orchestration platform we have developed allows users to define agent behaviour types, agent types (containing possibly more than one behaviour) and multi-agent scenarios through graphical user interfaces avoiding the coding effort. The scenario produced by the platform is a rule based system in contrast to the existing systems where agents are coded through a programming language. Thus the platform provides a higher level abstraction to agent development. The behaviours of the agents can easily be modified by changing the rules. The formalism developed within the scope of this work to represent the behaviours in a rule based system can be used in any agent platform to execute the multi-agent scenario, since there are rule based languages that can be used with most of the programming languages (For example CLIPS [2] language can be embedded within languages such as C, Java, FORTRAN and ADA). In this way, the system becomes highly transportable and interoperable, empowering the software developers to cater agent solutions more efficiently.

There are three main tools in the platform both for macro-level (societal) and for micro-level (agent internals) design. The *Behaviour Type Design Tool* helps to design agent behaviour templates which are then used by the *Agent Type Design Tool* for building agent types, which in return are exploited by the *Scenario Design Tool* to create application specific multi-agent scenarios.

The *Behaviour Type Design Tool* allows users to define the behaviours of an agent as a workflow template of basic agent operations such as sending a message,

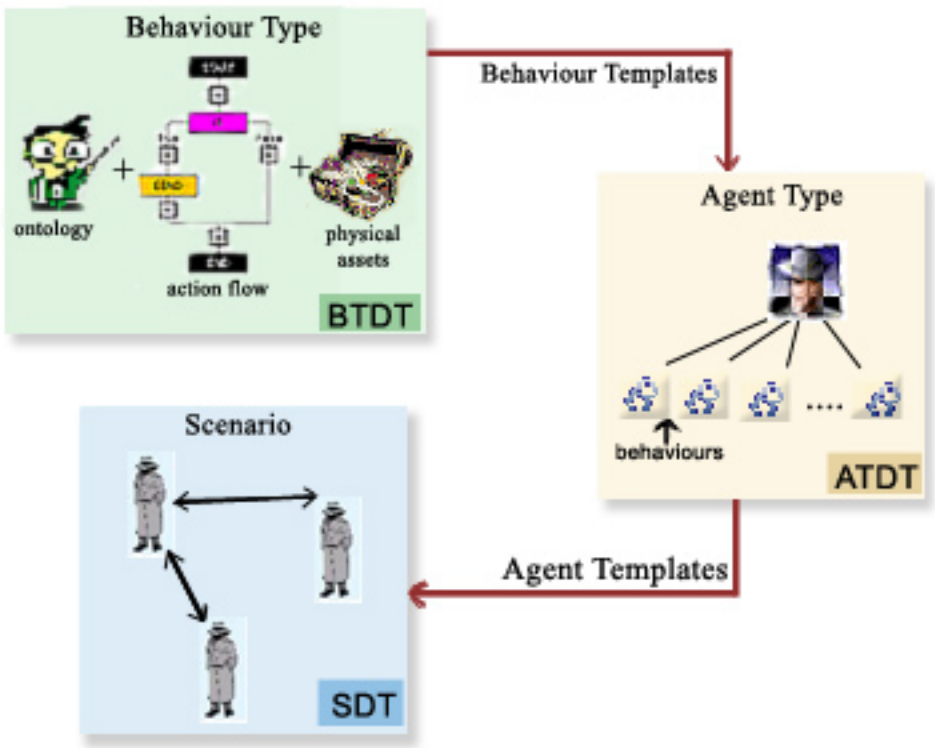


**Fig. 1.** The Components of the Agent Orchestration Platform

receiving a message, performing an action, or making a decision. There are also "if" and "while" blocks to control the flow of operations an agent performs. The existing applications can be inserted in the behaviours as activity blocks, so the platform provides an easy way for adapting existing distributed systems to multi-agent systems.

The platform enables the user to define the ontologies of the messages between the agents through an Ontology Design Tool.

Once the behaviour templates are designed through the *Behaviour Type Design Tool*, the *Agent Type Design Tool* helps the user to design an agent type by including the desired behaviour templates. Notice that these two graphical tools help with the micro level design. To assist the user with the design of agent societies, that is, for macro level design, *Scenario Design Tool* is used. This tool helps to select the necessary agent types (or new agents can also be built at this level with the *Agent Type Design Tool*) and to define the associations among agents. The scenario specific agent information is also given at this time with the proper interfaces provided, and then, the scenario is initialized. Once the scenario is initialized, it is converted into Agent Behaviour Representation Lan-

guage (ABRL) developed for this purpose and the multi-agent system becomes ready for operation.

There are four more tools in the platform called the *Consistency Checker Tool*, the *Ontology Design Tool*, the *Physical Asset Design Tool* and the *Monitoring Tool*. *Consistency Checker Tool* basically checks the consistency of the design in terms of the sent and received messages, i.e., if an agent A is sending a message to an agent B, then B should have the necessary mechanisms in place to receive this message. The *Ontology Design Tool* helps to design the ontologies and the *Physical Asset Design Tool* assists the users in designing external sources for the input variables of the behaviours. The *Monitoring Tool* provides a graphical interface for tracing the running agents.

The platform has some additional functionalities, such as monitoring the running agents, adding new behaviours to certain agent instances, killing some of their behaviours, killing some of the agents, or changing the parameters of the scenario.

To be able to demonstrate the execution of the system in a concrete agent platform, we mapped the ABRL rules to JESS [8] and executed the system on JADE [6]. The paper is organized as follows: Section 2 summarizes the related work. Section 3 is devoted to the description of the Agent Orchestration Platform developed. In Section 3.1, the *Behaviour Type Design Tool* is described and in Section 3.2 the *Agent Type Design Tool* is presented. These two tools address the micro level agent design. Section 3.3 explains the macro level multi-agent system design, that is the *Scenario Design Tool*. Section 3.4 presents the *Agent Behaviour Representation Language* and a brief example is provided in Section 3.5 to clarify the concepts. Section 3.6 describes the initialization of a multi-agent scenario in the orchestration platform. Finally Section 4 concludes the paper.

## 2 Related Work

Considering the increasing need for developing high-level agent system building tools, there has been a considerable amount of research on agent oriented software design, most of which are mainly based on Object Oriented analysis and design methods. Several methodologies are defined to specify the macro-level (agent society and organization structure) design of multi-agent systems [13]. These methodologies enable developers to go systematically from a statement of requirements to a design that is sufficiently detailed to be implemented directly.

There are two well known methodologies that provide a top-down and iterative approach towards modeling and developing agent-based systems, namely, Gaia [12], and MaSE [3]. They basically define the roles in a scenario, the responsibilities of these roles, and the interactions between them. These methodologies have been used in some applications, such as ZEUS [15] which uses Gaia and agent tool [3] which uses MaSE. However these methodologies and their applications mostly concentrate on the macro level agent design; their aim is not micro design. For example in ZEUS, the user has to define the functionality of the agents by writing Java codes with the given API.

In this paper we provide one more level of abstraction with the platform developed; a user is able to define the whole functionality of the behaviour via a GUI, and can make use of her existing applications through an API without the need to modify them.

ISLANDER [5] also provides a model and a tool for macro level agent design from a different perspective. The authors make an analogy between multi-agent-systems and human institutions, and describe a formal model for "agent based electronic institutions". They model the agent community in terms of, norms, roles, scenes, dialogic frameworks, and provide a graphical editor where the user can model his agent community in terms of these concepts. After the design, they provide a tool for the verification of the specification. However as also indicated in the paper [5], the work focuses on the macro-level (societal) aspects, instead of micro-level (internal) aspects of the agents.

## 3 Agent Orchestration Platform

As shown in Figure 1, there are three interacting tools, namely, the *Behaviour Type Design Tool* (BTDT), the *Agent Type Design Tool* (ATDT) and the *Scenario Design Tool* (SDT) in the agent orchestration platform developed. In this section, the details of these tools are presented.

### 3.1 Behaviour Type Design Tool (BTDT)

Behaviours of an agent specify its role in a scenario. We note that basic operations that an agent may perform include:

- An agent may receive input from the outside world. This could be data from sensors or messages from other agents.
- After evaluating the messages and/or some application data, an agent may take a decision, perhaps by invoking an inference engine.
- An agent may need to invoke existing applications. In doing this, the received messages and/or some application specific data may be used as input parameters to the invoked applications.
- An agent may decide to send messages to other agents.

An agent behaviour can be modeled as a workflow of these behaviour building blocks since there are data and control flow dependencies among them. In this way, it becomes possible to construct an agent behaviour as a workflow through a GUI by using these basic behaviour blocks. Note that behaviour blocks other than the ones specified above can also be defined.

A design tool needs the power to express the control statements like "if" or "while" to organize the flow among blocks an agent needs to execute. Furthermore an agent may be continuously executing its behaviours (called "Cyclic Behavior" in JADE terminology), or once ("One-Shot Behavior"), and such control statements should also be provided by a design tool.

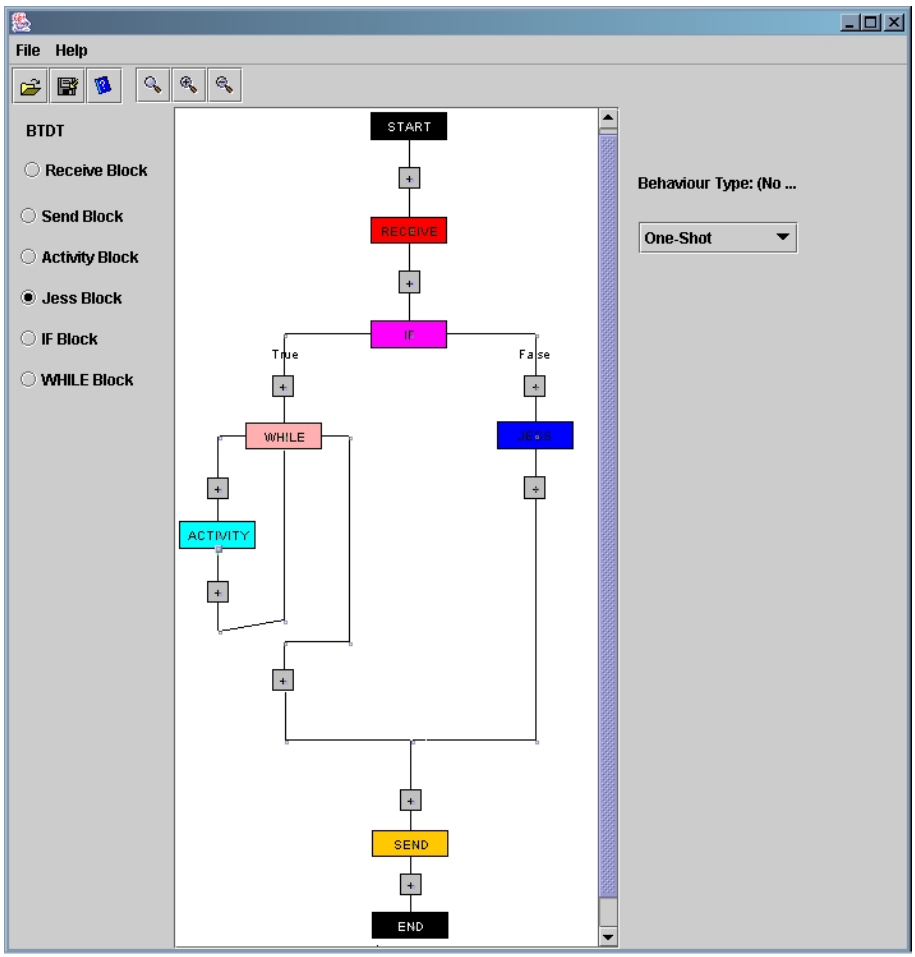


**Fig. 2.** Behaviour Type Design Tool

With these observations, we have developed the *Behaviour Type Design Tool* (BTDT) as shown in Figure 2 as a workflow design tool where there is a node for each possible generic action of an agent. The tool allows the users to select basic behaviour building blocks, drag them onto the canvas and draw the transitions among them. After a block is placed on the canvas, the user specifies the required semantics of the block. For example, for a "send block", the block name, the performative such as "inform" or "request", and the ontology of the message are specified.

The tool allows the following functionality to be defined through these nodes:

**Receive block:** Receive block models the task of receiving a message by an agent. Note that the users may wish to filter the messages received by an

agent by specifying some constraints. For example, a user may wish to define the ontology, the performative, or the sender of the message expected. The GUI tool provides a construct that enables the user to define all kinds of filters on messages. Since only the generic behaviour template is designed at this stage, the sender of the message is not specified here (This is specified through *Scenario Design Tool* while designing a specific scenario). While specifying the ontology of the message the user is provided with an Ontology Design Tool, and Protégé [10] is used for this purpose. Here the user defines the ontology of the message, then the ontology is saved as an Resource Description Framework (RDF) [11] file. The platform parses the RDF files, constructs and compiles the ontology classes to be used by the agents.

In order to provide data flow and sharing among the different building blocks of an agent, a "Global Variable Pool" is defined, which holds the variables that are produced and consumed as a result of the execution of the behaviour blocks. "Global Variable Pool" may contain the following types of variables:

- The variables extracted from a message received: The output of a receive block is a message in a specific ontology, and is stored in the "Global Variable Pool". In doing this, all the fields of the message are extracted and are stored in a collection of variables by conforming to the class/subclass hierarchy of the given ontology.
- The variables produced by an activity: The collection of variables that are produced as a result of the execution of activity blocks.
- The variables produced as a result of executing a rule engine: The collection of variables that are produced from an Inference Engine block.

**Activity block:** To be able to construct a multi-agent-system for an existing distributed system, the platform enables the user to invoke predefined applications, by involving them as activity blocks in the agent behaviour workflow. The tool gives the ability to the user to specify a predefined application, choose one of its methods and specify the input variables of that method. Variables from the "Global Variable Pool" can be assigned to the input variables of the methods, and/or the user can provide some external sources to be used to provide values to the input variables. These external sources are termed as "Physical Assets". The platform provides a tool also to define and manipulate the physical assets as shown in Figure 3. There are three kinds of physical assets:

- Simple variables: The user can assign a simple value to a variable. A simple variable is composed of the name of the variable, its type and its value.
- URLs (Uniform Resource Locators): The user can state that the value that is to be mapped to the variable should be extracted from an XML file. After getting the URL of the XML file, the tool presents the user the nodes of the XML file as a Document Object Model (DOM) hierarchy through the use of an XML parser (Xerces parser [14] is used in the implementation). When the user selects one of the nodes, an XPath expression is created by the tool to access the related node by simply constructing the path to the root of the XML file.

– Databases: The value of the variable can also be extracted from a database by providing the proper coordinates, that is, the database name, its URL, and the necessary login id and password. The tool helps the user to visualize the tables and their related rows. When the user selects the row she wants to extract, the query to obtain that information from the database is formulated.

  Note that to realize this functionality, traditional database techniques are used such as connecting to databases through their JDBC interfaces, querying the data dictionary to obtain the table names and their fields, querying the database for the content and displaying these results graphically. Although these technologies are mainstream and does not constitute the innovative aspects of our work; facilitating the job of the designer in this way is valuable.

With the *Behaviour Type Design Tool*, only the types of behaviours (i.e., templates) are designed, hence providing the names and types of the physical assets are sufficient. These physical assets are assigned to actual specific values through the *Scenario Design Tool* at the initialization phase. For example, with BTDT, we may design a behaviour that consists of a receive block and an activity block which will access a database. The specific type of agent that will send the expected message to this receive node and the specific database to be accessed are only known in the scenario design time and hence specified through *Scenario Design Tool*.

It is clear at this point that the tools of the platform share information. For example the physical assets designed through *Behaviour Type Design Tool* are later initialized through the *Scenario Design Tool*. We chose to store the shared information, such as the physical assets, as XML files to facilitate information sharing among the different tools of the platform.

Note that the output of an application may be used to set the global variables, which may be the input to other applications.

**Send block:** Defining send blocks is similar to defining receive blocks. The user needs to define the performative of the message, the ontology of the message, and the content language of the message. To fill the content of the message, the ontology of the message is presented to the user pictorially as a hierarchy, where the user assigns each node a value, by either choosing a variable from the global variable pool, or by defining new physical assets with the help of *Physical Asset Design Tool*. The receiver of the message and the contents of the physical assets are defined in the *Scenario Design Tool*, since these values are specific to a given scenario.

**Inference Engine:** An agent may want to execute an inference engine to decide on what to do at certain point in the flow of its behaviour, i.e., it may have some predefined rules, and according to the facts it gathers, it may execute these set of rules against the newly obtained facts. These rules may be predefined, or they can be dynamically obtained according to the changing aspects of its environment.

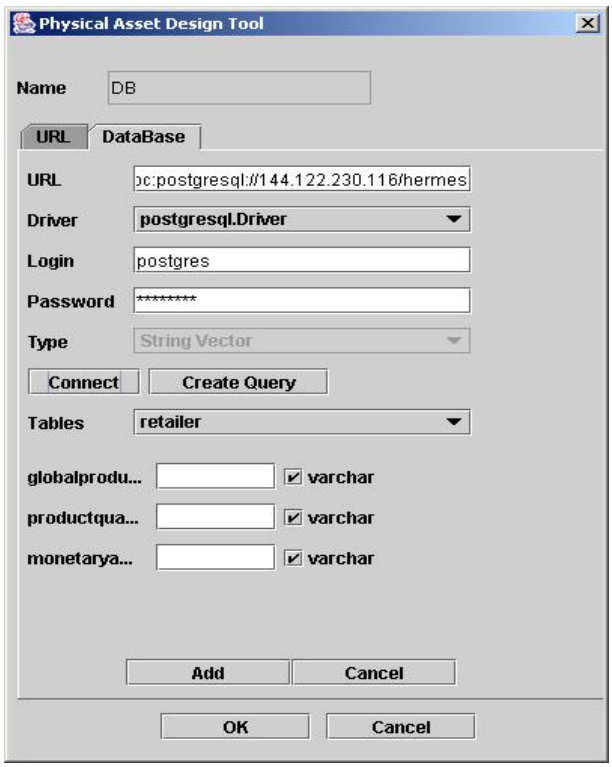


**Fig. 3.** Physical Asset Design Tool

After the user finishes the design of a new behaviour type, it is saved as an XML file which includes the parameters of the blocks, and the order of execution of these blocks. Again, XML is chosen as the intermediary format to facilitate information sharing among different tools of the platform. This XML definition is used in the *Scenario Design Tool* to visualize the flow of execution, where the scenario specific values are provided such as the missing values for physical assets, or the receiver and sender of the messages.

### 3.2 Agent Type Design Tool (ATDT)

After having defined the behaviours, the next step is to define the agent types. *Agent Type Design Tool* helps user to give a name and assign behaviour types to an agent type from the existing behaviour types that have been designed previously through the Behaviour Type Design Tool. These are saved again in an XML file to be used in the Scenario Design Tool. New agent types can be constructed either from scratch or by modifying the existing agent types.

### 3.3 Scenario Design Tool (SDT)

After having designed the necessary behaviour types and agent templates, the user is now ready to design her multi-agent scenario. Through a GUI, she first adds the agents‘ types necessary in her scenario to the panel. She selects the agents from the predefined agent types, or she can build a new agent type, with the help of Agent Type Design Tool. Then she edits each of the agents to make these agent types specialized to the scenario. From a graphical user interface she visualizes the agents and their behaviour types, she can edit each of these behaviours to instantiate specific behaviour instances to the agent, i.e., she assigns the senders and receivers of the messages selecting from the agents defined in that scenario. Finally, she configures the predefined physical assets to map them to specific values for this scenario. Once the initialization of the scenario is completed it is converted to the ABRL rules and stored in a knowledge base.

### 3.4 Agent Behaviour Representation Language (ABRL)

We represented the behaviours in the scenario through a rule based system specifically designed for this purpose, called Agent Behaviour Representation Language (ABRL), which is composed of rules, facts and functions. Representing the whole scenario in this way in a rule-based system makes it highly interoperable.

Note that in a rule based system, there is no way to specify an order of execution implicitly. A rule is like an “if-then” statement in a procedural language, but it is not used in a procedural way. While “if-then” statements are executed at a specific time and in a specific order, according to how they are written in the source code, rules are executed whenever their “if” parts (their left-hand-sides) are satisfied.

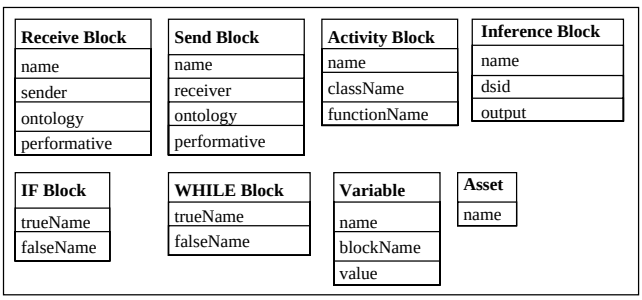


**Fig. 4.** The Fact Templates

On the other hand, there could be “execution dependencies” among the behaviour blocks in an agent behaviour. Assume that an agent is expecting a

message from another agent, and after having received this message, the agent is expected to invoke a predefined activity, say A, by using a field in this message as an input parameter. Clearly there is an execution dependency over here; the activity block can only be executed after the expected message is received. In other words, we need to introduce a mechanism to enforce an execution sequence among the rules.

Execution dependency issues has been addressed previously in the literature within the context of workflow systems and a formalism has been developed [9, 1] to specify inter-task dependencies as constraints on the occurrence and temporal order of events.

In the following we provide an intuitive explanation for the mechanism we have developed for enforcing an execution order in an agent behaviour. The formal treatment of the subject for workflow systems is given in [4].

We associate a "guard expression" with each behaviour block to manage the control flow in an agent behaviour through the rule-based system. For the example given above, the "start guard" of activity A is a condition expression stating that the previous receive block has to be executed before this block can start.

We represent the behaviour blocks in Agent Behaviour Representation Language (ABRL) as follows:

- Represent each block as a rule and a fact pair,
- Put the necessary guard expressions in the left-hand-side of the rules to enforce the required execution order,
- Represent global variables and physical assets as facts to satisfy the information passing constraints among blocks,
- Use some auxiliary functions to implement the required actions in a specific agent platform, whenever the rules are fired.

In order to model the behaviour blocks and their execution order through a rule-based system, we define fact templates for representing the semantics of the blocks and the rules and the guard expressions to describe the execution order as explained in the following.

The fact templates are given in the Figure 4. These facts include all the necessary attributes to define each block. For example the "ReceiveBlock" fact includes the name of the block, the sender of the message, its ontology and performative. The facts are asserted while the behaviours are initialized through ABRL.

To describe the set of actions that will be performed when a block is encountered, an ABRL rule is defined. The general template of a rule is given in Figure 5. On the left-hand-side of a rule, there is a fact and a guard expression. The fact informs the rule that "there is a block with the specified attributes pending to be executed in the flow of the behaviour", and the guard expression informs the rule about "the preconditions of that block to be fulfilled so that the block can be executed".

When a rule finds the related fact, and the asserted guard expression, the rule is fired. When a rule is fired, the necessary actions are executed. To represent

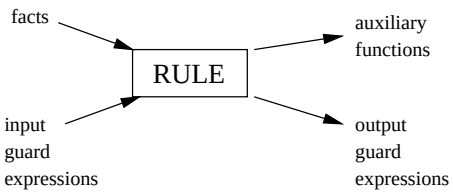


**Fig. 5.** The Rule Template

these set of actions, one auxiliary function per block is defined. There are four auxiliary functions one for each of the send, receive, inference engine, and action blocks. When a block is successfully handled, the guard expression of the block that follows is asserted. This in return fires the next rule in the sequence.

As explained in section 3.1 blocks have two kind of information sources: global variables and physical assets. Global variables are used to pass information among blocks and physical asset represent external information sources as shown in Figure 4. These are the inputs and outputs of the blocks. The "blockName" in Figure 4 of "Variable" specifies the block that outputs this global variable. Since Physical Assets are stored as XML files, and their values are extracted by parsing the XML files, there is no "value" slot in their template.

### 3.5 An Example

In this section, we provide an example to clarify the concepts introduced. Consider the example flow given in Figure 2. In this example, an agent behaviour template is defined as follows: first, the agent is expected to receive a message. Then it executes an "if" block, and depending on the "if condition", it either invokes an application within a "while" block or it activates an inference engine to take a decision. When one of these branches terminates successfully, it sends a message to another agent.

The behaviour design given in Figure 2 can be described through a "behaviour" tree as shown in Figure 6 to visualize the execution dependencies.

The nodes of this tree correspond to the blocks in the design tool and the top level "Behaviour" node implies that the child blocks will be executed sequentially (as default). By considering the execution dependencies in this tree (Figure 6) the guard expressions and their associated conditions can automatically be obtained. For example, which branch of the "if" block will be executed depends on the "guard expression" to be inserted by the "if" block, and this information, that is, the result of "if" condition is only available at run time. Table 1 gives an informal description of the guard expressions and conditions generated for this example. Note that if there are more than one block to be executed in parallel, more than one rule will fire at the same time.

For each block a rule is designed using these guard expressions. For example, for the "receive block" in Figure 2, the start guard is the start of the behaviour,

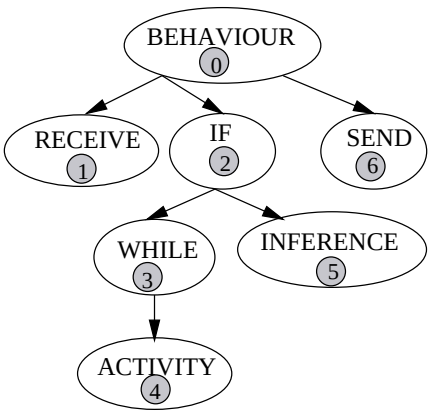


**Fig. 6.** The Process Tree

**Table 1.** Guards of the blocks of the example behaviour

| Task Number | Start Guard | Start Condition | Terminate Guard | Fail Guard | Terminate Cond. | Fail Cond. |
|---|---|---|---|---|---|---|
| 0 | TRUE | | 6 terminates | 1, 2, 6 fails | | |
| 1 | O starts | receive fact asserted | TRUE | TRUE | msg. received | msg. could not be received |
| 2 | 1 terminates | if fact asserted, cond. var. available | 3 or 5 terminates | 3 or 5 fails | | |
| 3 | 2 evaluates to TRUE | while facts asserted, iteration var. available | TRUE | 4 fails 4 fails | while cond. is false | |
| 4 | While Condition is TRUE | activity fact asserted, input var. available | TRUE | TRUE | activity terminates successfully | activity fails |
| 5 | 2 evaluates to TRUE | jess fact asserted, input var. available | TRUE | TRUE | inference eng. terminates successfully | inference eng. fails |
| 6 | 2 terminates | send fact asserted, msg. content ready | TRUE | TRUE | msg. sent | msg. could not be sent |

and its start condition is satisfied when the "receive fact" is asserted. The "receive fact" is prepared according to the templates given in Figure 4, and it includes all the necessary attributes for defining the semantics of this block. By using these information, the following rule is generated for the "receive block":

```
(Rule receiveBlockRule
      (receiveBlock (name ?n) (sender ?s) (ontology ?o)
      (performative ?p))
      (startGuard BehaviourStarted)
      ⇒
      (receiveBlockFunction ?s ?o ?p)
      (assert (startGuard (?n finished)))
)
```

On the left-hand-side of the rule, there are the "receive fact" and the "startGuard" of the "receive block". When the behaviour design is completed, all the facts of the blocks in the behaviour are asserted into the knowledge base, and the "BehaviourStarted" guard is asserted to initialize the first block in the flow

of the behaviour. So the "receiveBlockRule" finds its left-hand-side conditions satisfied and it fires. When it fires, the auxiliary function "receiveBlockFunction" is called with the necessary parameters, that is, the sender, the ontology, and the performative of the message expected. This function executes the specific actions which, in this case, is to receive the expected message from the sender. When the message is received, its content is asserted into the "Global Variable Pool" using the template given in Figure 4. Finally, to indicate that the "receive block" has successfully terminated, the "start guard" of the block that follows (i.e. the "if block") is asserted. Hence the "if block" fires and the execution continues in this way. When the last block (i.e. the "send block") in the flow of the behaviour successfully terminates, the guard that terminates the execution of the behaviour is asserted. Note that if the behaviour is "cyclic", the start guard that initiates the behaviour is asserted once again.

### 3.6 Initialization

When configuration of all agents in a scenario are completed, it is necessary to initialize them in a multi-agent scenario. At this point, a concrete agent platform is needed and we have chosen the JADEplatform to run the agents.

The platform has a default agent called as "Agent Factory" (AF) which initializes each agent in the scenario as a JADE agent with the following default behaviours: The first behaviour is called "Behaviour Initialization Behaviour", which listens to "Agent Factory" for new Behaviour Messages. AF agent sends new behaviour messages to the newly created agents to inform them about their behaviours. With this message, the agent receives the content of its behaviours, its ontologies, physical assets, and decision structures (used by the inference engine). The agent first initializes its ontologies by parsing the received RDF ontology file and it constructs JADE Ontology classes. It uses these ontology classes while receiving and sending messages. Then it parses the Physical Asset files and initializes its physical asset collection. The agent retracts the rules from a specified database (if this is specified at the design phase); converts the rules into JESS rules, and initializes its Decision Structures collections accordingly. After all of the parameters necessary for its behaviour to execute (i.e. ontologies, physical assets, decision structures) are in place, the agent behaviour is initialized. Then, a JESS engine is instantiated; the ABRL representation of the behaviours are asserted to the engine, and the behaviour starts to execute which completes the agent initialization.

The platform also provides a monitoring tool, where the user can monitor the existing agents, their behaviours, ontologies, physical assets, and decision structures in a scenario. With the help of this tool, the user can also visualize the messages exchanged by the agents. Having a view of the scenario from this tool, the user may want to add new behaviours to an agent, kill one of its behaviours. The AF agent sends the messages to the agent, which in turn handles the necessary actions. The user may also want to change the values of some physical assets; the platform provides a tool for this functionality. Again the AF agent sends the necessary update messages to the agent, this time, with the help

of its second default behaviour, "UpdateListeningBehaviour", so that it updates its physical asset collection accordingly. This behaviour also handles the Decision Structure update messages.

## 4 Conclusions

For wider deployment and exploitation of agent technology, it is important to provide software companies with powerful agent design tools to help them develop solutions for their customers effectively. There are agent development frameworks that aim to ease this process by providing APIs, but the developer still has to define the behaviour of the agents by writing code in a programming language.

In this paper, we present an orchestration platform that allows the developers to design a complete agent-based scenario through graphical user interfaces. To address the micro level design of an agent community, agent behaviours are modeled as workflow processes and designed accordingly. The multi-agent scenario produced by the platform is stored as a rule based system which makes it easy to accommodate the changing requirements of user scenarios. The formalism developed within the scope of this work to represent the behaviours in a rule based system can be used in any agent platform to execute the multi-agent scenario, since there are rule based languages that can be used with most of the programming languages.

Guard expressions are used to enforce an execution sequence in a rule-based system. Traditional database and file processing techniques are integrated into the system to facilitate the design for the developers. The platform also provides macro level design capabilities such as defining the ontologies of the agents, their roles in the system, and their interactions in a systematical way.

The aim of the tool is to design a multi-agent system either from scratch, or by adapting existing distributed systems to multi agent systems. Given a multi-agent scenario and existing applications, the orchestration tool helps to create the necessary agents, handles all the interactions between the agents, and outputs a multi-agent system that is ready to be executed on an agent platform.

Note that although any agent platform can execute the multi-agent scenario presented as a rule based system, to be able to demonstrate the execution of the system on a concrete agent platform, we mapped the ABRL rules to JESS and executed the system on JADE.

## References


1. P.A. Attie, M.P. Singh, A. Sheth, and M. Rusinkiewicz, "Specifying and enforcing intertask dependencies", in Proc. of 19th Intl. Conf. on Very Large Data Bases, September 1993.
2. C Language Integrated Production System (CLIPS), http://www.ghg.net/clips/CLIPS.html

3. S. A, DeLoach, M. Wood (2000), "Developing Multiagent Systems with agentTool", in Proc. of Intelligent Agents: Agent Theories, Architectues, and Languages - 7th International Workshop, ATAL-2000, Boston, MA, USA.
4. Dogac, A., Gokkoca, E., Arpinar, S., Koksal, P., Cingil, I., Arpinar, B., Tatbul, N., Karagoz, P., Halici, U., Altinel, M., "Design and Implementation of a Distributed Workflow Management System: METUFlow", In *Workflow Management Systems and Interoperability*, Dogac, A., Kalinichenko, L., Ozsu, T., Sheth, A., (Edtrs.), Springer-Verlag NATO ASI Series, 1998.
5. M. Esteva, D. de la Cruz, C. Sierra, "ISLANDER: an electronic institutions editor", in Proc. of AAMAS02, July, 2002, Bologna,Italy.
6. Java Agent Development Framework, http://sharon .cselt.it/projects/jade/
7. N.R. Jennings (2000), "On agent-based software engineering", 117 (2) 277-296. Artificial Intelligence
8. Jess, The Expert System Shell for the Java Platform, http://herzberg.ca.sandia.gov/jess
9. J. Klein, "Advanced rule driven transaction management", in Proc. of 36th IEEE Computer Society Intl. Conf. CompCon Spring 1991, San Francisco, CA, March 1991.
10. The Protege Project, http://protege.stanford.edu /index.html
11. Resource Description Framework (RDF), http:// www.w3.org/RDF/
12. M. Wooldridge, N. R. Jennings, and D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design", Journal of Autonomous Agents and Multi-Agent Systems 3 (3), 2000.
13. M. Wooldridge, N. R. Jennings, and D. Kinny, "A Methodology for Agent-Oriented Analysis and Design", Proc. 3rd Int Conference on Autonomous Agents (Agents-99), 1999.
14. Xerces Java Parse, http://xml.apache.org/xerces2-j /index.html
15. ZEUS, http://www.labs.bt.com/projects/agents/zeus