Running agent-based simulations. Unternehmensmodell / Computersimulation.

Meyer, David; Karatzoglou, Alexandros; Buchta, Christian; Leisch, Friedrich; Hornik, Kurt

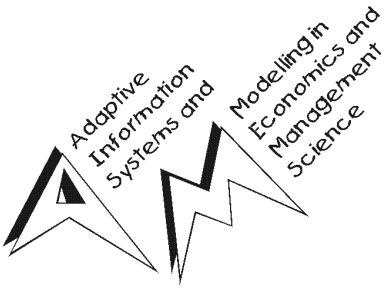
DOI:
10.57938/27b43eb8-2fef-4154-af2c-de882b93c462

Published: 01/01/2001

Document Version:
Publisher's PDF, also known as Version of record

Document License:
Unspecified

Working Paper Series

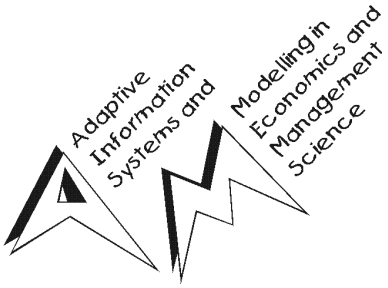


Running Agent-Based Simulations

David Meyer
Alexandros Karatzoglou
Christian Buchta
Friedrich Leisch
Kurt Hornik

Working Paper No. 80
June 2001

Working Paper Series



June 2001

SFB
'Adaptive Information Systems and Modelling in Economics and Management Science'

Vienna University of Economics
and Business Administration
Augasse 2–6, 1090 Wien, Austria

in cooperation with
University of Vienna
Vienna University of Technology

http://www.wu-wien.ac.at/am

This piece of research was supported by the Austrian Science Foundation (FWF) under grant SFB#010 ('Adaptive Information Systems and Modelling in Economics and Management Science').

Abstract

When running agent-based simulations using ready-made components, one usually faces heterogeneity problems both for the agents' implementation and for the underlying platform. To circumvent these kind of hindrances, we introduce a wrapper technique for mapping the functionality of agents living in an interpreter-based environment to a standardized CORBA interface, thus facilitating the task for any control mechanism (like a simulation manager) which just will need to handle one set of commands for all agents involved. This mapping is made by an XML-based definition file. We also have built a generic simulation manager which makes use of agents with homogeneous interfaces, and which can be used to run simple simulations. In a sample session, we illustrate how wrapper and simulation manager do interact. Finally, we describe the database interface representing the global Artificial Economy environment in which agents operate. In the Appendix, we give a brief overview of the current installation of the SFB reference computer platform.

1 Overview: The Simulation Concept

Simulations in the "Artficial Firms and Markets Lab" (AFML) can be implemented using an object-oriented style of programming, allowing for distributed objects (over a cluster of workstations or even the Internet). The AFML provides reusable components for economic simulations. One can think of each economic entity as an agent, e.g., firms, (groups of) consumers, investors, markets, etc., interacting with each other. A typical simulation takes several agents, defines their relationships and observes the resulting interactions between them over time.

Running a simulation usually amounts to writing a control program in one's favorite programming language, named the *central simulation manager* below. All agents provide standardized interfaces such that they automatically have AFML bindings and can be used for simulations as modularized components. To achieve this goal, we use the CORBA mechanism, which is available for most standard programming languages, including C, C++, Java and Perl. Additionally, all software packages which can dynamically load shared libraries written in one of the above languages can be made CORBA-aware, but this could be a tricky task, though. We therefore need an easy-to-use mechanism enabling the integration of data analysis environments like MATLAB or R, which are the target platforms for AFML simulations, as they offer convenient ways of analyzing simulation results and are also (typically) used for implementing objects and methods.

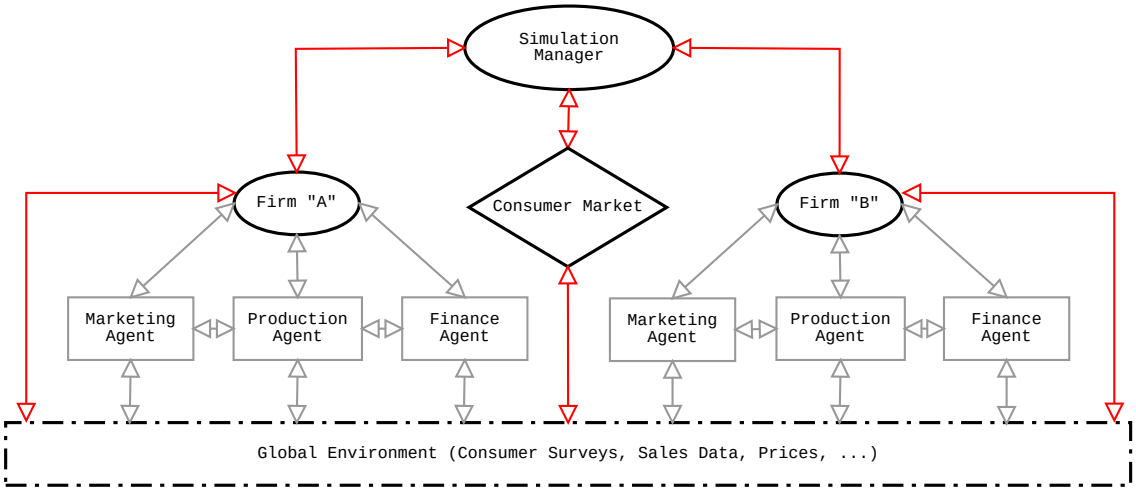


Figure 1: A simple simulation with two competing firms, each consisting of 3 agents for marketing, production and finance, respectively. Gray elements indicate features currently not implemented.

Consider a simple example involving two competing firms, named `"Firm A"` and `"Firm B"`, respectively, operating on a consumer market (see Figure 1). In a future project step, it is planned that each firm is modularized itself, having agents responsible for marketing, production and finance; in the

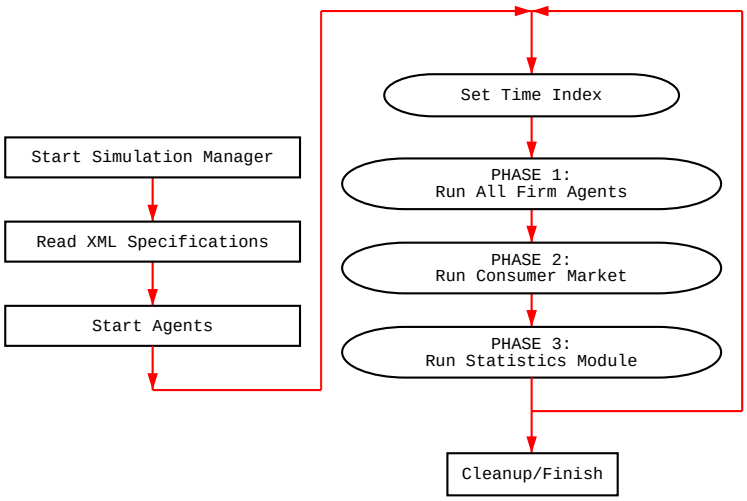


Figure 2: The simulation cycle

current project state, only simulations at firm-level are implemented. Market coordination and clearance is performed by the consumer market agent, which models a (disaggregated) consumer population.

Additionally there is a global environment representing the common knowledge of all agents involved. The global environment is stored in an SQL database. This frees us from problems that arise from simultaneous access by different agents like blocking etc. because modern databases are designed for exactly these purposes.

In this paper, we aim to describe some mechanisms allowing agents to interact with each other. There are mainly two aspects in this interaction: the communication between the agents and the simulation manager (via a CORBA based "wrapper" program) on the one hand, and the information interchange between the agents in a global environment (via a database interface) on the other hand. In this paper, we discuss the following topics:

- how to specify simulation settings in XML for a generic simulation manager, covering the most basic needs,
- how to "normalize" agent interfaces via a wrapping technique, thus allowing the simulation manager to treat all agents the same way, and
- the structure of the database representing the global environment.

In the appendix, we also briefly introduce the reference platform available for the simulation purposes.

2 The Simulation Manager

2.1 A typical simulation cycle

Figure 2 roughly sketches a typical simulation cycle. After an initialization block (essentially the start of all agents), the simulation enters the main loop: after updating the time index, phase by phase, all agents are run for one cycle. All agents of one phase need to complete their tasks, before the next phase is entered. Upon completion of the final cycle, a cleanup is performed.

In order to facilitate the development of specific simulation managers, we now introduce a basic implementation of a simulation manager behaving as just described, handling "wrapped" agents. Although simply designed, we consider it powerful enough as to be used as a ready-made tool. It is capable of running an arbitrary number of agents at different phases (e.g., a market clearing agent should only by started when all "normal" agents are done).

The simulation manager offers the following information to all agents (which actually not have to consider all variables):

TIME The current cycle; obviously essential to retrieve and store the correct information in the database.

NAID A unique agent ID, which could be useful for debugging output.

DBNAME The name of the database, which could change from simulation to simulation, if one wants to keep results from previous simulations.

DBTIMEOUT The maximum number of retries for database operations, after which the simulation stops with a timeout exception.

`TIME` is set at the begin of each cycle. The simulation components are specified in a definition file, which the Simulation Manager reads at startup. We use XML to store these settings (and also for agent interfaces which are presented in the next section).

2.2 XML - The Extended Markup Language

XML is a set of rules, guidelines, conventions etc. (see the XML (2000) recommendation) for designing text formats for data in general, in a way that files are easy to generate and read (by a computer), that they are unambiguous, and that they avoid common pitfalls, such as lack of extensibility, lack of support for internationalization/localization, and platform-dependency. XML is an open standard.

One main idea of XML is to extend data with meta-information, allowing applications to give it a special treatment. XML-tags are formed by a name, enclosed by "<" and ">". They are mostly paired, thus delimiting some information, but could also be empty. In addition, they may have attributes to indicate additional information. The following small example of an address book shows a possible application:

```
<book>
  <address>
    <name>David Meyer</name>

    <contact type="phone">+43/1/58801/10772</contact>
    <contact type="email">david.meyer@ci.tuwien.ac.at</contact>

    <work>
      <university/>
      <sfb initiative="1">
    </work>
  </address>

  <address>
    ...
  </address>
</book>
```

2.3 Using XML for simulation settings

The formal syntax of XML files is outlined in "data-tag-definition" (.dtd) files (see Appendix A); instead of describing the corresponding "`simulation.dtd`" directly, we give an illustrating example. The definition file for a sample simulation setting could look as follows:

```
<?xml version="1.0"?>
<!DOCTYPE simulation SYSTEM "simulation.dtd">

<simulation cycles="100",
            dbname="meyersimul1",
            notify="david.meyer@ci.tuwien.ac.at">
  <agent level="1" copies="2">SkilledAgent</agent>
  <agent level="1">RandomAgent</agent>
  <agent level="2">Consumer</agent>
  <agent level="3">Statistics</agent>
</simulation>
```

- The first two lines form the XML header which is similar for all XML files; the specific structure is defined in the ”wrapper.dtd”-file, indicated in the second line.
- `<Simulation>` takes some parameters which account for the whole simulation:
  - `cycles` indicates the number of simulation cycles,
  - `dbname` the name of the database (there are also tags `dbhost` and `dbuser` defined for network access),
  - `notify` an (optional) e-mail address where error messages are sent to if they occur, and
  - `timeout` gives the maximal number of retries for database operations before the simulation is aborted.

  `<Simulation>` may contain an arbitrary number of
- `<agent>`-tags, which indicate the name of the wrapper XML files (without extension), described in the next section.
  - The attribute `level` indicates the phase, in which the agent is run;
  - the attribute `copies` the number of copies to be executed.

3 Agent specification

To make agents simulation-aware, we need

1. a translator which accepts generic method calls from the simulation manager and passes the corresponding method call to the agent, and
2. an interface definition which describes the corresponding translation table.

3.1 Wrapping Agents

Because agents can be implemented in different programming languages (R, MATLAB, C++, ...) on different platfoms (Windows, Linux, ...) depending on the user's needs, the simulation manager has to operate in a technically heterogeneous environment. To address this kind of problem, we use CORBA, a standard communication tool, which e.g. features interface standardization and platform independence (see the technical section for more details). One basic goal of this work was to create a program that acts as an intermediate, translating the simulation manager's CORBA calls to the native agent programming environment method calls. This program “wraps” the agent and exports through CORBA a standard interface (in the following we will refer to this program as the wrapper). The translation of the agents interface is stored in an XML-defined interface definition format, which (mainly) makes one-to-one correspondences to the CORBA interface calls. The simulation concept is illustrated in Figure 3.

A typical simulation takes several agents, defines their relationships and observes the resulting interactions between the objects over time. The agents interact with the environment and subsequently with each other. The whole simulation is coordinated by a central agent which starts and synchronizes the simulation components (agents). The central coordinating agent (simulation manager) makes the CORBA calls to the wrapper, the wrapper (previously having parsed the XML interface definition of the agent) translates the call and executes the native agent command, as if it were typed at the command prompt.

3.2 How agents are controlled during simulations

Before we can use XML to define agent interfaces, we have to look at an agent's simulation “life” in order to derive the functionality to be handled by the wrapper. It can be summarized by the following steps, visualized in figure 4.

1. Start of the interpreter (MATLAB, R, ...)
2. Loading of the agent's source code (not needed e.g. for MATLAB, because the source code is loaded when functions are invoked)
3. Setting of some variables by the controlling client (like the name of the database)
4. Initializing step (variables, opening of database connection, ...)
5. *Action loop (executed several times):*

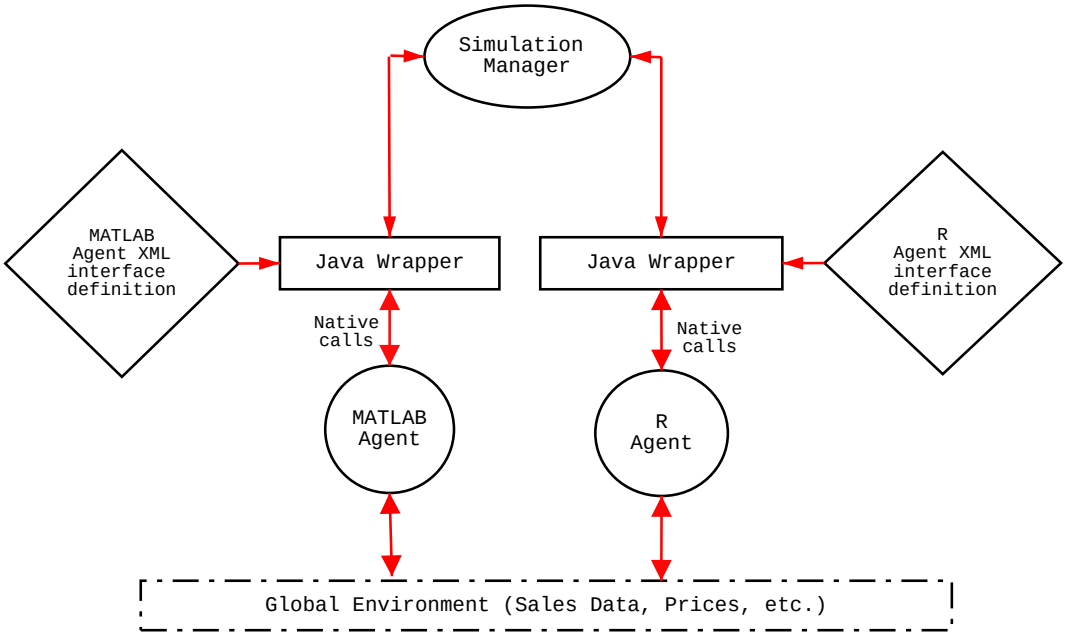


Figure 3: A simple simulation with two agents

(a) Setting of periodic information (like the time index) by the simulation manager

(b) Execution of task function

(c) Eventually retrieving of results by the simulation manager

6. Cleanup-Step (Saving of results, closing of database connections, . . .)
7. Quit from the interpreter

From this "life-cycle", we derive the specification for an appropriated interface.

3.3 Using XML for defining agent interfaces

The agent's interface is reduced to 6 main methods (`start`, `boot`,`init`, `action`, `finish` and `stop`) corresponding to the main steps just mentioned, and 2 helping methods (`setattr` and `getattr`) for information passing. In addition, we require a `printdone`-method, along with the definition of a `donestring`, both needed for communication control: each command string sent to the interpreter is immediately followed by the command defined by `printdone`, which should print a specified OK-message. If this string is detected by the wrapper, an OK-signal is sent to the simulation manager which subsequently can assume that the command has completely been executed, and that the agent is ready for more commands.

The interface defined in the example XML file below defines a simple R agent:

```
<?xml version="1.0"?>
<!DOCTYPE wrapper SYSTEM "wrapper.dtd">

<wrapper>
  <start>R -q --vanilla</start>
  <boot>source("Ragent.R")</boot>
  <init>init()</init>
  <action>action()</action>
  <finish>finish()</finish>
  <stop>q()</stop>

  <setattr>assign ("<name/>",<value/>)</setattr>
```

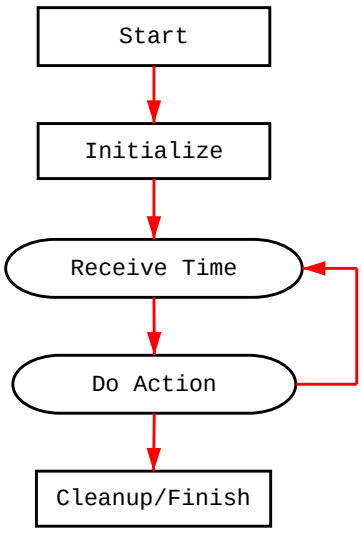


Figure 4: The simulation cycle (agent's view)

```
  <getattr><name/></getattr>

  <printdone>printdone()</printdone>
  <donestring>OK</donestring>
</wrapper>
```

An implementation in octave would look like this:

```
<?xml version="1.0"?>
<!DOCTYPE wrapper SYSTEM "wrapper.dtd">

<wrapper>
  <start>octave -q</start>
  <init>Oagent_ini</init>
  <action>Oagent</action>
  <finish></finish>
  <stop>quit</stop>

  <setattr><name/>=<value/></setattr>
  <getattr><name/></getattr>

  <printdone>printdone</printdone>
  <donestring>:-)</donestring>
</wrapper>
```

- The `<start>` tag defines the start-command (in quiet mode), executed as a shell command.
- The `<boot>` tag (mandatory) typically encloses an command for sourcing files into the interpreter.
- `<init>`, `<finish>` and `<action>` specify user-defined functions (for initialization, cleanup and the main action method, respectively); `init` and `finish` are mandatory.
- The `<setattr>` tag contains a method call which sets the various attributes of the agent (e.g., `TIME` or `DBNAME`). It takes 2 parameters: `<attrname/>` and `<value/>`, which are empty tags and play the role of placeholders. They are replaced by real values provided by the Simulation Manager before the `setattr` command is executed. It should be called as many times as necessary to set all the parameters of the agent. Note that the implementation could ouf course simply be done by an ordinary variable assignment like `<name/>=<value/>` as in the `octave` example, but the use of a seperate function allows the mapping from the generic variable names to the specific implementation variable names, thus preserving the agent's name space.

- `<getattr>` defines the complementary function to `setattr`: the implementation should simply print the requested value; it is parsed and returned to the client.
- `printdone`, as mentioned above, should simply print a defined OK-message enabling flow control. This message must be specified in the `donemessage` tag. Note that for this method, one should implement an extra function and not, for example, simply use a `print` statement. [1] Also note, that one could simply use the command prompt as an OK-message, and omit the `printdone`-tag completely.

4 A Sample Session

On the SFB reference computer platform, the software can be found in the archive `/home/ARCHIVE/simulation.tar.gz`.
Using the command `tar zxpvf /home/ARCHIVE/simulation.tar.gz`, it expands into the directory `simulation`, which contains a `README` file and 3 subdirectories:

`demo/` A simple sample simulation, described below.

`tests/` A test suit to check the various wrapper functions.

`wrapper/` The wrapper java files, along with the .idl-definitions.

To run the demonstration, you will need an `X term` with writing permissions for the host `elrond.ci.tuwien.ac.at`, and a secure shell (SSH) connection to this host. The demonstration is started with the command `./sim` in the `demo`-subdirectory. Two graphics windows should appear, both displaying the same random time series. One displaying agent is written in Octave, the other in R; the time series is created by a third (Octave) agent. The communication is established via a simple `POSTgres`-database. What the `sim`-script actually does is sketched as follows:

- First, the agents are started by the following commands:

  ```
  java wrapper/AgentWrapper Osimcore &
  java wrapper/AgentWrapper Oagent &
  java wrapper/AgentWrapper Ragent &
  ```

  running the three agents via the wrapper (`Osimcore` is the time series creating agent, implemented in Octave; `Oagent` and `Ragent` are the displaying agents implemented in Octave and in R, respectively). The XML definitions of `Oagent` and `Ragent` have already been introduced in section 3.3, the one of `Osimcore` looks like this:

  ```
  <?xml version="1.0"?>
  <!DOCTYPE wrapper SYSTEM "wrapper.dtd">

  <wrapper>
    <start>octave -q</start>
    <boot></boot>
    <init>Osimcore_ini</init>
    <action>Osimcore</action>
    <finish></finish>
    <stop>quit</stop>

    <setattr> <name/>=<value/> </setattr>
    <getattr><name/></getattr>

    <printdone>printdone</printdone>
    <donestring>:-)</donestring>
  </wrapper>
  ```

  For each started wrapper, the java-program `AgentWrapper` reads the corresponding XML-interface definition and waits for CORBA-commands.

[1] If the commands sent to the wrapper are echoed by the interpreter, the command itself would be parsed as the OK-message, and the OK-message itself would remain in the buffer and be parsed immediately after the next command invocation, thus confusing the synchronization of the agents.

- Second, the simulation manager is simply started with:

  ```
  java wrapper/SimManager simulation
  ```

  The simulation manager program `SimManager` reads the simulation-specification `simulation.xml`, similar to the one presented in section 2.3:

  ```
  <?xml version="1.0"?>
  <!DOCTYPE simulation SYSTEM "simulation.dtd">

  <simulation cycles="5" dbname="demo">
    <agent level="1">Osimcore</agent>
    <agent level="2">Oagent</agent>
    <agent level="2">Ragent</agent>
  </simulation>
  ```

  and starts the simulation cycle as presented in section 2.1 by passing CORBA-calls to the agents.

The simulation ends after the last period, when the last `action` call is done: first, all agents are sent the `finish`-command, and waits until all agents are done. Then, all interpreters are stopped by sending the `quit` command to all agents. After that, the wrapper programs are stopped, and finally, the simulation manager is stopped itself.

5 The Database Structure

Agents communicate on the market by manipulating a common database (for private storage needs, they may use local databases hidden from the other agents.) The EER- (Extended-Entity-Relationship-) diagram in figure 5 describes the structure of this global environment at a given period, as outlined in Buchta & Mazanec (2001), in terms of logical units (the *entities*) and their connections or *relationships*. The *main entities* are `Firm` and `Consumer`. Derived ("*weak*") *entities* are the firm's `products`, and the corresponding `features` and perceived `attributes`, which are modeled separately: these entities cannot exist without their associated "strong" entities from which they must derive the primary keys to build sensible units. *Relationships* are represented by connected rhombuses: to indicate a one-to-many (1:n) or many-to-many (m:n) connectivity, they have additional symbols on the corresponding side. Information is carried by *fields*, which can be affixed both on entities and on relationships. Fields in boldface are *primary*, i.e. they uniquely determine concrete database entries (for week entities only with the key from the associated strong entity).

From this scheme, we derive (from top left to bottom right) the database tables summarized in table 1, using some simple rules (see Teorey et al. (1986)):

1. Entities build separate tables, if they are composed of more than key fields.
2. 1:n- (one-to-many-) relationships are created by storing the key of the "one"-sided table into the other ("many") table.
3. m:n- (many-to-many-) relationships build separate tables.

In addition, the time index `tid` has been added to all relations. For convenience, we use shortages instead of the long canonical names (see table 2).

Note that a segment is identified by the composite key `firmid,pid,tid` because consumers are only offered one product per firm at a given time. Fields in italic font are non-prime (i.e., they do not belong to the key). Table 3 indicates the fields' domains.

Care should be taken on insert/update-operations that a record is inserted if it does not already exist and just updated else. This is especially important for tables filled by several agents (e.g., the `production` table).

We finally remark that the database is in so-called "Boyce-Codd-normal-form": all fields depend directly from the key fields or are key fields themselves. This guarantees consistency if data columns are erased or modified.

EER−Diagram
Simulation Database

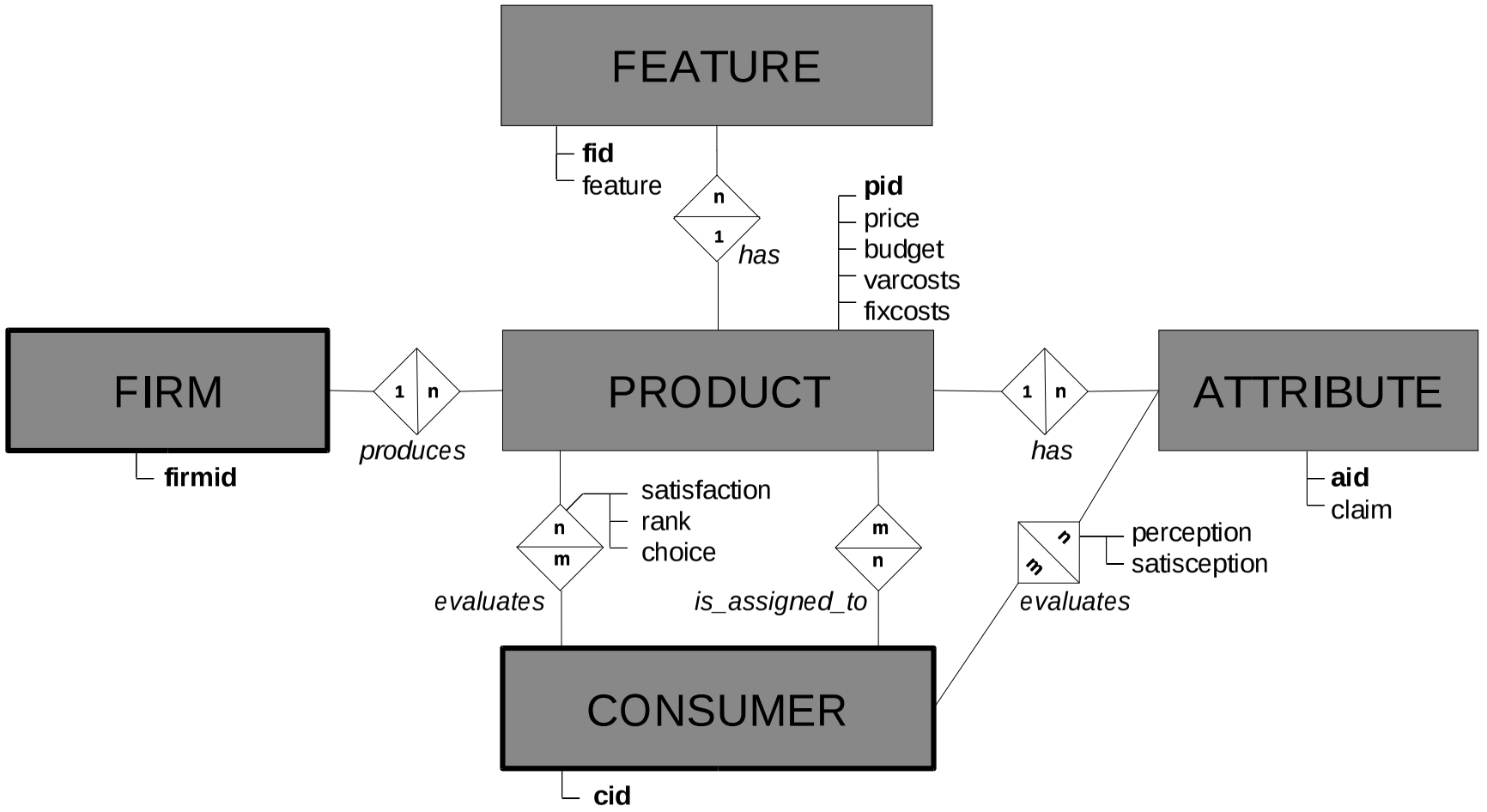


Figure 5: The EER-Diagram

	canonical relation name	field names
1.	product_features	`firmid`, `pid`, `fid`, `tid`, *`feature`*
2.	firm_produces_products	`firmid`, `pid`, `tid`, *`price`*, *`budget`* [,*`varcosts`*, *`fixcosts`*, *`prodbudget`*,...]
3.	products_have_attributes	`firmid`, `pid`, `aid`, `tid`, *`claim`*
4.	consumers_evaluate_products	`cid`, `firmid`, `pid`, `tid`, *`satisfaction`*, *`rank`*, *`choice`*
5.	products_are_assigned_to_consumers	`firmid`, `pid`, `cid`, `tid`
6.	consumers_evaluate_product_attributes	`firmid`, `pid`, `aid`, `cid`, `tid`, *`satisception`*, *`perception`*

Table 1: Derived Relations from the EER-Diagram

	canonical relation name	short form
1.	product_features	`features`
2.	products	`product`
3.	product_has_attributes	`claim`
4.	consumers_evaluate_products	`response`
5.	products_are_assigned_to_consumers	`segmentation`
6.	consumers_evaluate_product_attributes	`survey`

Table 2: Shortages for the relations

6 Technical Wrapper Details

6.1 CORBA

The use of different programming environments in agent based simulations (or any other projects) creates the problem of interprocess method calls. To address this kind of problems, we use CORBA. CORBA is the acronym for Common Object Request Broker Architecture, an vendor-independent architecture and infrastructure that computer applications use to work together on a single computer or over networks (see `http://corba.org/`). Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can inter-operate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network. This means that CORBA capable programs can call each others public available methods regardless of the programming language used for their creation.

Agents, by the means of CORBA, offer functionality to other programs, therefore they are also refered as (CORBA) *servers*. Accordingly, the simulation manager makes use of the functionality offered by the agent-servers, it therefore is an example of a (CORBA) *client*. Note that the notions of client and server are not mutually exclusive: if we consider e.g. firm agents, they could be clients and servers at the same time (controlled by the simulation manager and controlling firm departments).

field names	domain
`firmid, pid, fid, cid, sid, aid, tid,` *`rank`*	$\mathbb{N}_0$
`satisfaction, satisception`	{-2,-1,0,1,2}
`choice, perception, claim`	{0,1}
`feature`	(0,1)
`price, budget [, varcosts, fixcosts, prodbudget, ...]`	$\mathbb{R}_0^+$

Table 3: The fields' domains

6.2 Interface definition

The whole wrapper functionality is publicly available through CORBA calls and can be called by any CORBA-capable program. The interface definition (.idl)-file looks as follows:

```
interface Wrapper
{
    void   start    (in string XMLFILE, in string TAG);
    void   boot     (in string TAG);
    void   init     (in string TAG);
    void   action   (in string TAG);
    void   finish   (in string TAG);
    void   stop     (in string TAG);
    void   kill     (in string TAG);
    void   waitdone (in string TAG);
    void   setattr  (in string NAME, in string VALUE, in string TAG);
    string getattr  (in string NAME, in string TAG);
    void   settout  (in long   SECS, in string TAG);
};
```

The elements are decribed below:

start starts the interpreter where XMLFILE is the name of the file containing the method tags of an instance of server. *(synchronous operation)*

boot pipes a command containing the contents of a `boot` tag to the interpreter. *(asynchronous operation)*

init pipes a command containing the contents of an 'init' tag to the interpreter *(asynchronous operation)*.

action pipes a command containing the contents of an 'action' tag to the interpreter *(asynchronous operation)*.

finish pipes a command containing the contents of a 'finish' tag to the interpreter *(asynchronous operation)*.

stop pipes a command containing the contents of a 'stop' tag to the interpreter and waits for the interpreter to terminate. *(synchronous operation)*.

waitdone waits until the end of call symbol appears on the output stream of the interpreter *(synchronous operation)*.

setattr pipes a command containing the contents of a 'setattr' tag where the 'name' and 'value' tags are replaced by the arguments NAME and VALUE to the interpreter. Waits for the end of call symbol *(synchronous operation)*.

getattr pipes a command containing the contents of a 'setattr' tag where the 'name' tag is replaced by the argument NAME to the interpreter. Gets the next (non echo) line from the interpreter, waits for the end of call symbol, and returns the line *(synchronous operation)*.

kill terminates the server process (which will terminate its children too).

settout sets the time in seconds to elapse before a timeout condition is assumed after an invocation of waitdone *(synchronous operation)*.

TAG an arbitrary string appended to the debugging output of the server.

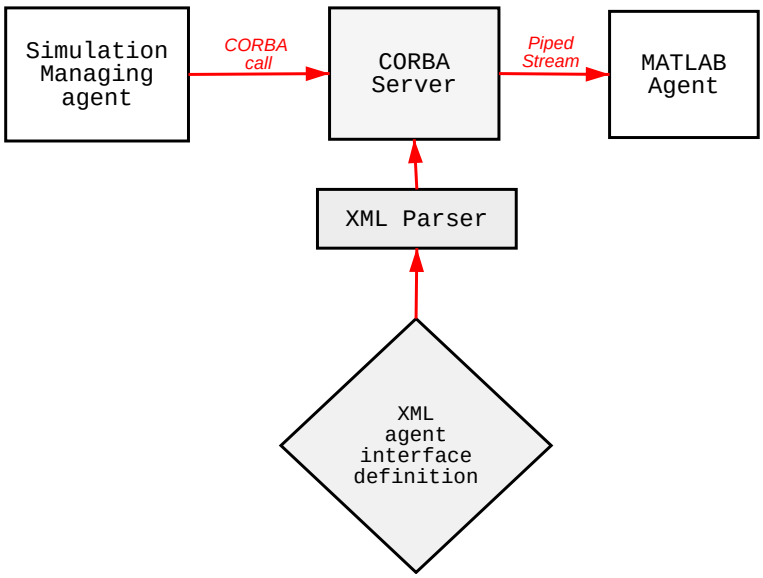


Figure 6: A schematic representation of the Wrapper structure

The `start()` method starts the agents computing environment (let's say R or MATLAB) and connects to it through a piped stream. Piped streams are a mechanism in the Java I/O library to set up a stream of data between two processes. Pipes are input/output pairs: data written on the output stream shows up on the input stream at the other end of the pipe. The wrapper can thus communicate with the agents computing environment and make all the necessary method calls (see Figure 6). Note that most of the functions are executed in asynchronous mode, i.e. return immediately. For determining the status of execution (i.e., whether the call is done), one must use the `waitdone()`-function after all asynchronous operations are started. `waitdone()` itself is synchronous.

The method call which the wrapper passes to the agent in reaction to a CORBA method call from a client on a specific function, say `action()`, is that defined in the `<action>` tag of the XML interface definition file.

6.3 Error Handling

During the simulation run, errors may occur, either caused by broken communication or by execution faults on the interpreter level. Errors raise (throw) a runtime exception that should be handled by the invoking client. Error message appear somewhere in the java virtual machine's error text.

The following error (warn) conditions can occur on the server ([] optional message parts):

```
ERROR Start : <String> UNSPECIFIED [DONESTRING | START | ACTION | STOP]
ERROR Start : <String> <Exception>
ERROR Call  : <String> <Command> [ILLEGAL STATE | <Exception>]
ERROR Wait  : <String> [ILLEGAL STATE | NOT DETECTED | <Exception>]
WARN  Stop  : <String> [ILLEGAL STATE | ABNORMAL TERMINATION | <Exception>]
```

where

`UNSPECIFIED` indicates that one of the mandatory xml tags was not found.

`Exception` indicates a runtime exception (I/O errors . . .)

`Command` indicates the command sent to the interpreter

`ILLEGAL STATE` indicates an illegal invocation sequence, either there is an attempt to invoke calls (init — action — finish — stop) or waits consecutively.

`ABNORMAL TERMINATION` indicates that the interpreter did not exit normally when invoking the stop method.

`String` indicates the Tag string from the client (e.g. some debugging identifier).

Care should be taken, however, if the MATLAB interpreter is used, because the server cannot detect any error except the one created when the interpreter is quit, which raises an `ERROR Wait : <String> NOT DETECTED`-exception. Therefore, the whole MATLAB program should be embedded in a `TRY...CATCH`-loop, and should quit if an error occurs.

7 Future Work

Although the existing framework is powerful enough to run first simulations, there are still open issues:

- A major drawback is that wrapped agents cannot automatically wrap other agents, which would be useful to implement firm agents which have to coordinate department agents (marketing, production, finance, ...) themselves. Currently, agents of this type have to be implemented in a CORBA-aware language like JAVA, which is not very comfortable.
- The same inconvenience applies to communication between agents (e.g., interdepartmental communication)
- A further issue is to port all tools to other platforms (at least Windows), which should be an easy task in theory (all components, like JAVA and CORBA, do exist on many platforms).

A The DTD for defining simulations

```
<!-- simulation DTD version="$Revision: 1.20 $" -->

<!ELEMENT simulation (agent+)>
<!ATTLIST simulation cycles     CDATA        #REQUIRED
                     dbname     CDATA        #REQUIRED
                     dbhost     CDATA        #IMPLIED
                     dbuser     CDATA        #IMPLIED
                     notify     CDATA        #IMPLIED
                     timeout    CDATA        "60">

<!ELEMENT agent       (#PCDATA)>
<!ATTLIST agent level  CDATA "1"
               copies CDATA "1">
```

B The DTD for defining Agent Interfaces

```
<!-- wrapper DTD version="$Revision: 1.20 $" -->

<!ELEMENT wrapper    (start, boot?, init?, action, finish?, stop,
                      setattr?, getattr?, printdone?, donestring)

<!ELEMENT start      (#PCDATA)>
<!ELEMENT boot       (#PCDATA)>
<!ELEMENT init       (#PCDATA)>
<!ELEMENT action     (#PCDATA)>
<!ELEMENT finish     (#PCDATA)>
<!ELEMENT stop       (#PCDATA)>

<!ELEMENT setattr    (#PCDATA|name|value)*>
<!ELEMENT getattr    (#PCDATA|name)*>
<!ELEMENT name        EMPTY>
<!ELEMENT value       EMPTY>

<!ELEMENT printdone  (#PCDATA)>
```

```
<!ELEMENT donestring (#PCDATA)>
```

C The Simulation Cluster

The Adaptive Modeling Cluster, which is the reference platform for all SFB-wide simulations, consists of 5 Linux computers, 4 of which are dual Pentium II machines running at 233MHz and one AMD Athlon 900MHz machine which is the cluster server (figure 7). The cluster computers are running on Debian Linux 2.2 with a MOSIX enhanced Linux kernel and communicate through a 100Mbps NEC Ethernet switch. MOSIX is a software package that enhances the Linux kernel with cluster computing capabilities. The enhanced kernel allows a cluster of x86/Pentium based computers to work cooperatively as if part of a single system. This means that the distribution and load balancing of the processes to the various CPU's in the cluster is done automatically, without any user intervention. The software currently installed on the cluster includes:

- MOSIX 0.97.10
- C/C++ gcc 2.95.2
- Java IBM SDK 1.3.0
- Perl 5
- R 1.3.0
- Matlab 6
- Octave 2.0.16
- MySQL 9.38
- PostgreSQL 6.5.3

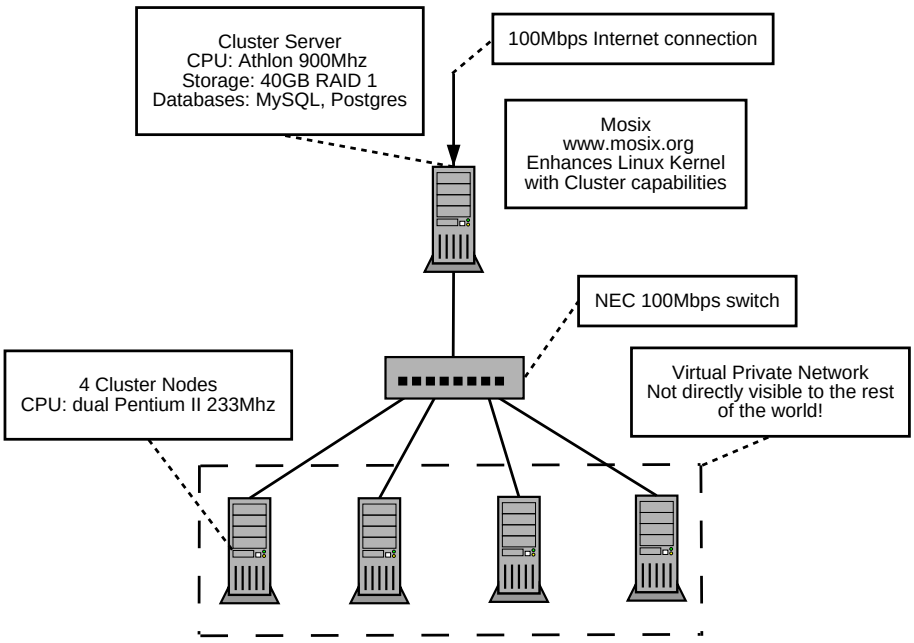


Figure 7: A schematic representation of the Cluster

On the SFB Knowledge Warehouse (`http://elrond.ci.tuwien.ac.at/`), the current installation status of the Cluster is listed, as well as technical user hints.

References

Buchta, C. & Mazanec, J. (2001). *SIMSEG/ACM - A Simulation Environment for Artificial Consumer Markets*. Tech. rep., SFB Working Paper Series Nr. 60.

Ihaka, R. & Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, **5**(3), 299–314.

Teorey, T. J., Yang, D., & Fry, J. (1986). A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, **18**(2), 197–222.

XML (2000). *Extensible Markup Language (XML), 1.0 (2nd Edition)*. World Wide Web Consortium, <http://www.w3.org>.