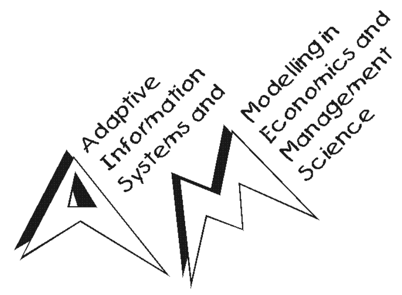
Report Series

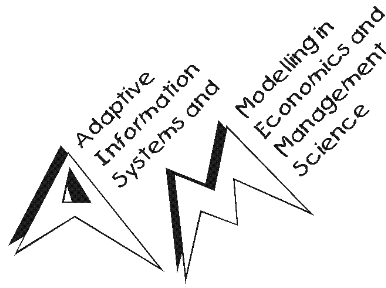


A Simulation Framework for Heterogeneous Agents

David Meyer
Christian Buchta
Alexandros Karatzoglou
Friedrich Leisch
Kurt Hornik

Report No. 74
October 2002

Report Series



October 2002

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

Papers published in this report series
are preliminary versions of journal articles
and not for quotations.

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

We introduce a generic simulation framework suitable for agent-based simulations featuring the support of heterogeneous agents, hierarchical scheduling and flexible specification of design parameters. One key aspect of this framework is the design specification: we use an XML-based format which is simple-structured yet still enables the design of flexible models. Another issue in agent-based simulations, especially when ready-made components are used, is the heterogeneity arising from both the agents' implementations and the underlying platforms. To tackle these kind of obstacles, we introduce a wrapper technique for mapping the functionality of agents living in an interpreter-based environment to a standardized JAVA interface, thus facilitating the task for any control mechanism (like a simulation manager) because it has to handle only one set of commands for all agents involved. Again, this mapping is made by an XML-based definition format. We demonstrate the technique by applying it to a simple sample simulation of two mass marketing firms operating in an artificial consumer environment.

1 Overview: The Simulation Concept

Simulations are often implemented by using an object-oriented style of programming, allowing for distributed objects (over a cluster of workstations or even the Internet). In the following, we consider the example of an agent-based economic simulation: 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 combines several agents, defines their relationships and observes the resulting interactions between them over time. After the simulation design has been defined (see, e.g., Richter & März, 2000), running a simulation usually amounts to writing a control program in one's favorite programming language, named the *simulation manager* (see below), coordinating a set of previously implemented, autonomous agents. One might wish that all agents should provide standardized interfaces such that they automatically have the same bindings and thus can be used for simulations as modularized components. General mechanisms for providing standardized interfaces (like CORBA) do exist, but usually require advanced programming skills to be used. We therefore offer an easy-to-use mechanism enabling the integration of data analysis environments like MATLAB, Octave or R, as they offer convenient ways of analyzing simulation results and are also (typically) used for implementing objects and methods. Another key issue is the variation of parameters in controlled experiments. Furthermore, we need a scheduling scheme determining the order of invocation within a single experiment (design) and the number of runs (periods) per design.

Consider a simple example involving two competing firms, named `"Firm A"` and `"Firm B"`, respectively, operating on a consumer market (see Figure 1). Each firm could be modularized itself, having agents responsible for marketing, production and finance. Market coordination and clearing may be performed by a consumer market agent, which models a (disaggregated) consumer population. In addition, a global environment, representing the common knowledge of all agents, is typically involved: this environment may be stored, e.g., in an SQL database, thus solving problems arising from simultaneous access by different agents (like blocking etc.), or managed by an information broker similar to the one described in Wilson et al. (2000)—but these mechanisms are highly specific to the simulation design.

After a brief review of related work, we therefore restrict the presentation of our software[1] to:

- The specification of the simulation settings in XML for a generic simulation manager, supporting multiple design specification, and
- The "normalization" of agent interfaces via a wrapping technique, thus allowing the simulation manager to treat all agents the same way.

We complement the technical presentation by an application to a simulation of two mass marketing firms operating in an artificial consumer environment.

[1] available at: `http://elrond.ci.tuwien.ac.at/software/simenv-1.0.tgz`

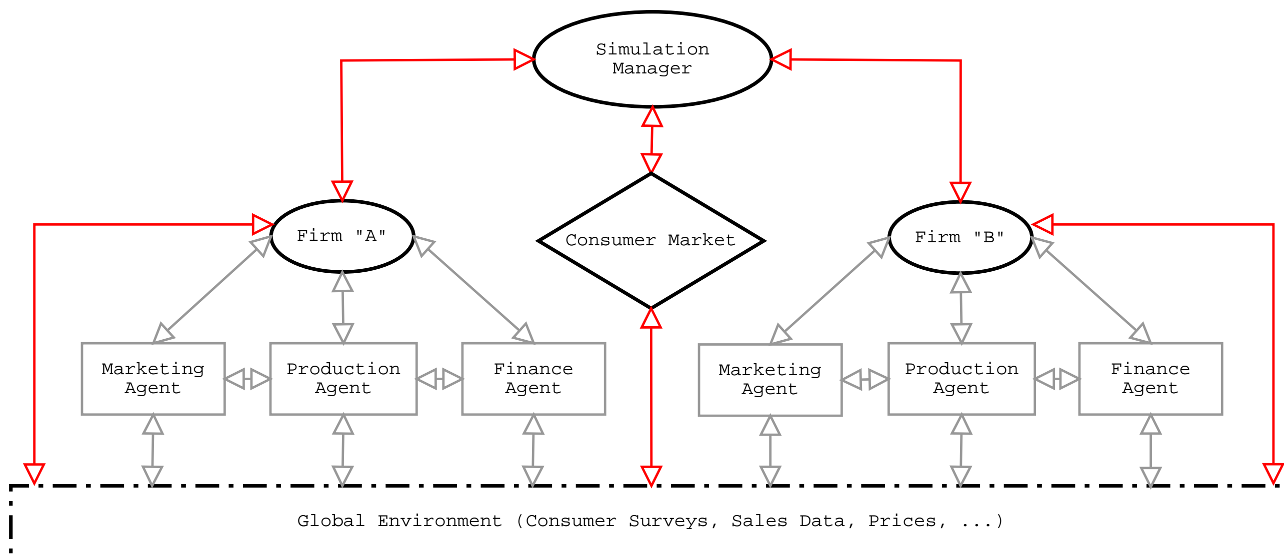


Figure 1: A simple simulation with two competing firms, each consisting of 3 agents for marketing, production and finance, respectively.

2 Discussion of Related Work

An incredible number of test beds for agent-based simulations exists, mostly complete toolkits with graphical user interfaces and ready-made components, but also very often designed for particular tasks. A crucial usability criterion, however, is the possibility of integrating existing code with new one. In their overview on agent-oriented programming, Genesereth & Ketchpel (1994) distinguish three possible mechanisms of this kind of "agentification": a) building a "transducer" (writing a translating module; this needs only the knowledge about the communications protocol), b) programming a "wrapper" (this usually necessitates the source code and deeper understanding of technical details), and c) rewriting the code. In this framework, our work fits in the first category.

One of the first attempts in integrating existing (semi-)autonomous systems in a larger coordinated framework seems to be the ARCHON project (Wittig et al., 1994) which offered a framework for intelligent cooperation—not necessarily of computer systems only. Each "intelligent system" is connected to an ARCHON layer containing a communications module, a planning and coordination module, an agent information module, and a monitoring module interfering with the software. The latter is responsible for the exception handling (i.e., must find a solution by requesting help from other agents). The system has been applied to Electrical Networks and to CERN accelerator monitoring tasks. But as Perriollat et al. (1994) explain, the integration of the various components has been done by embedding or modifying existing code to establish the general communication scheme.

A more generic approach is the "SWARM" toolkit (Minar et al., 1996), an Object C library for building hierarchical agent-based systems (a "swarm" designating a collection of agents). The system is still in use nowadays (see, e.g., the macro-language extension MAML by Gulyás et al., 2002), but presupposes good programming skills. Modern developments are rather often JAVA-based: see, e.g., "SILK" (Kilgore, 2000), offering a JAVA-class for simulation environments, or the more user-friendly "TASK" environment (Decker, 1996), representing a JAVA-based test bed for abstract representations of task environments. Both seem not explicitly to support the integration of legacy code. Another system, ZEUS (Nwana et al., 1999), is a complete toolkit for agent-based simulations with a graphical user interface which enables external connectivity by exporting a JAVA-API—but here also, the integration of existing software necessitates a programming step. A more recent approach for discrete event simulations, Extend (Krahl, 2000), offers various kinds of Windows connectivity features (IPC, Link & Embed, ODBC, ActiveX/OLE), thus facilitating the integration of Windows-based software, but does

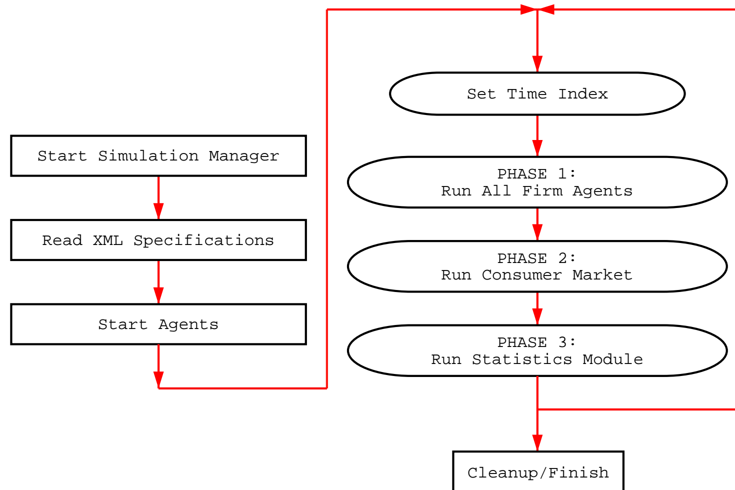


Figure 2: The simulation cycle

not support other platforms such as, e.g., Linux.

All these approaches have in common that the integration of existing code necessitates some coding in a low-level programming language (e.g., Object C or JAVA), although scientists often use high-level computing environments (like, e.g., MATLAB or R). Although the programming skills of scientists, as a matter of fact, certainly have increased in the last decade, we assume that scientists would prefer to work on a conceptual level and not have to care about technical details. In addition, none of these systems seem to support the specification of predefined design plans, which is a basic task in planning simulation runs. This is where our work fits in: we offer both a flexible approach to specify simulations and the possibility of integrating a wide range of existing software.

3 The Simulation Manager

3.1 A typical simulation cycle

A complete simulation design includes several simulations with (typically) different parameter settings for each single design. Designs can be run repeatedly. It may also happen that the set of agents is modified from design to design. Figure 2 sketches a typical simulation for a single design. After the simulation manager and the agents have been initialized, the simulation enters the main loop: after updating the time index, all agents—grouped by phase—are run for one cycle. All agents of one phase need to complete their tasks, before the next phase is entered. When the last phase is done, the next loop is entered. Upon completion of the final cycle, a cleanup is performed. This is repeated (usually with changing parameter sets) a specified number of times.

Our implementation of a generic simulation manager behaves just as described, handling "unified" agents. Because agents can be implemented in different programming languages (R, MATLAB, ...) on possibly different platforms (Windows, Linux, ...) depending on the user's needs, the simulation manager has to be capable of operating in a technically heterogeneous environment, and therefore was implemented in JAVA, a platform independent programming language, offering good support of network communication. Although of a simple design, we consider it powerful enough 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) by varying an arbitrary set of parameters through different designs. These parameters (e.g., market/product characteristics, initial prices, budgets) are offered by the simulation manager to the agents at the beginning of each new design block by using a simple broadcast mechanism. Information can either be public

(propagated to all agents) or private (propagetd to specific agents). Usually, public information will also include technical information, like the current period (updated at the begin of each cycle); or the agent identifier (which the agent might, e.g., include in its output information). The simulation components are specified in a definition file read by the simulation manager at startup. We use XML to define these settings.

3.2 Using XML for simulation settings

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, computer readable, and unambiguous. In addition, XML files avoid common pitfalls such as lack of extensibility, lack of support for internationalization/localization, and platform-dependency. In the context of interfacing systems, XML has been used, e.g., by Wilson et al. (2000) to describe port-based models (systems with interfaces and exchangeable implementations).

The formal syntax of XML files is outlined in “data-tag-definition” (.dtd) files enabling formal validation; but instead of describing the corresponding “`simulation.dtd`” directly, we give an illustrating example. The definition file for a sample simulation setting including firms and consumers with 2 designs could look as follows:

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

<simulation seed="4711"
            mailserver="mail.bar.com"
            notify="foo@bar.com"
            timout="10">
  <alldesigns repeat="20" cycles="30">
    <agent name="consumer" level="2" copies="100">
      <p name="reservprice">5</p>
      <p name="budget">10</p>
    </agent>
  </alldesigns>
  <design name="A">
    <agent name="firm" level="1" copies="2">
      <p name="type">mass</p>
      <p name="budget">100</p>
    </agent>
  </design>
  <design name="B">
    <agent name="firm" level="1" copies="2">
      <p name="type">niche</p>
      <p name="budget">50</p>
    </agent>
  </design>
</simulation>
```

- The first two lines form the XML header which is common to all XML files; the specific structure is defined in the ”`simulation.dtd`”-file, indicated in the second line.
- `<Simulation>` takes parameters which account for the whole simulation:
  - `seed` (optional) indicates the master seed for the whole simulation, allowing exact replication of whole simulations. If omitted, it is chosen randomly.
  - if `mailserver` and `notify` are specified, the simulation manager notifies the indicated e-mail recipient of normal or abnormal termination.
  - `timeout` is used for detection and removal of non-terminating agents (due, e.g., to programming errors or dead locks).

<Simulation> may contain an arbitrary number of

- <design> tags with the parameters:
  - name for identification in log files,
  - repeat for design replications, and
  - cycles for the number of periods.

  For convenience, parts common to all designs can be put into an (optional) <alldesigns> section. Each <design> tag may contain an arbitrary number of

- <agent> tags with the attributes name, copies, level (the phase, in which the agent is scheduled to run), and seed (which overloads the master seed if specified). Similar to <alldesigns>, parameters common to all agents can be put into an optional <allagents> section. Note that the <alldesigns> section may also contain an <allagents> section, whose parameters would be copied into all <alldesigns> sections. This allows the specification of parameters valid for all agents in all designs. For each <agent> tag, an arbitrary number of

- <p> tags specify the parameters for this particular agent, the name attribute this time indicating the parameter name.

If the same parameters exist both in general sections (<alldesigns> or <allagents>) and the <design> sections, the more specific parameters overrule the more general ones. This also accounts for agents with common names. The seeds for the agents' random number generators are created as follows: The master seed (either specified or drawn at random) is used to produce seeds for each agent. This seed can be overloaded at the agent level by using the <seed> parameter. If multiple copies are requested for an agent, the seed is used to create "sub-seeds" for all of its instances. Seeds specified in the <allagents> section are fixed for all designs.
By using this rather general framework, one is able to specify whole design plans in a flexible way. Simple design plans (like the full-factorial design) will usually be created in an automated way, but the structure also enables more elaborated designs: if one is not interested in the influence of all possible factor combinations, it is possible to reduce the number of parameter combinations by following certain design rules, thus substantially reducing the simulation time needed.

4 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 mappings.

4.1 Wrapping Agents

One basic goal of this work was to create a program that acts as an intermediary, translating the simulation manager's JAVA calls to the native agent programming environment method calls. This program "wraps" the agent and exports through JAVA 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) defines one-to-one correspondences to the JAVA interface calls. The whole 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 JAVA calls to the wrapper, the wrapper (initially having parsed the XML interface definition of the agent) translates the call and executes the interpreter command, as if it were typed at the command prompt.

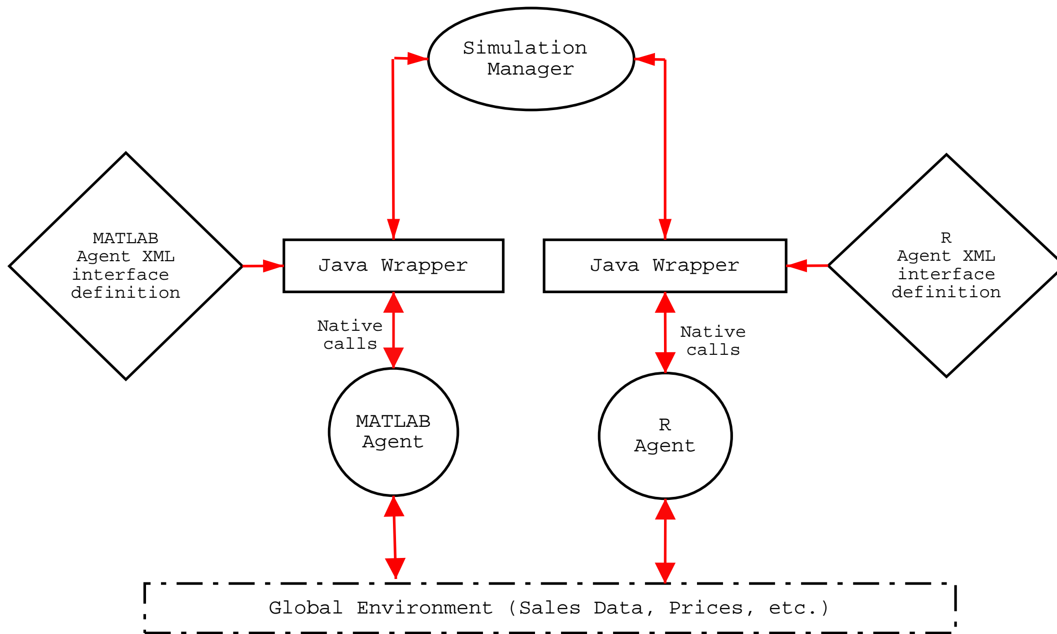


Figure 3: A simple simulation with two agents

Our modularized approach and the use of JAVA allows for easy extensions to distributed simulations: the calls from the simulation manager to the wrapper can also be routed through a LAN or the Internet using, e.g., the JAVA RMI (Remote Method Invocation) interface. If low-level languages have to be supported (like, e.g., C/C++), JAVA also offers native support of the CORBA communication protocol which is programming language independent.

4.2 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 simulation manager (like the data base name),
4. Initializing of variables, opening of a database connection, . . . ,
5. *Action loop (executed several times):*
   (a) Setting of periodic information (like the time index) by the simulation manager,
   (b) Execution of task function,
   (c) Possible retrieving of results by the simulation manager,
6. Cleaning up (saving of results, closing of database connections, . . .), and finally
7. Quitting from the interpreter.

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

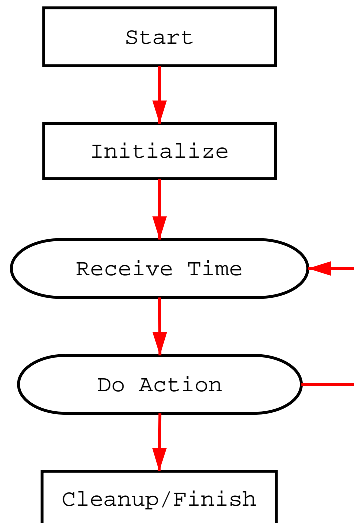


Figure 4: The simulation cycle (agent's view)

4.3 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 two 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>
  <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 (optional) typically encloses a command for loading 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 optional.
- 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 agent parameters. Note that the implementation could, of course, be simplified by an ordinary variable assignment like `<name/>=<value/>` as in the `octave` example, but the use of a separate function allows the mapping from the generic variable names to the specific variable names of the implementation, 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.[2] Also note that one could simply use the command prompt as an OK-message, and omit the `printdone`-tag completely.

In addition to the parameters specified in the `simulation.dtd`, the simulation manager offers some internal information to all agents (using the specification in `<setattr>`). Agents can of course choose to use or not to use this information (note that we recommend to implement an explicit function handling parameter setting, allowing to keep the name space clean and to filter unused information). Currently, the public information set includes: `TIME` (the current time index), `ANAME` (the agents' full name), `DNAME` (the optional design name), `NAID` (the agents' internal unique identifier), `NDID` (the unique internal design identifier), `NRID` (the current replication), and `SEED` (the current seed to be used for random number generation).

Furthermore, the simulation manager may also retrieve some information from the agent (as specified in `<getattr>`): currently, only one variable called `CTRL` is scanned. This "control" variable shall be used to pass commands to the simulation manager. In the current version, there is only one: "`stop`" causes the agent to be terminated by the simulation manager in a controlled way. We are planning to extend this yet simple mechanism in future versions: one could think, e.g., of introducing "`INBOX`", "`OUTBOX`" and "`RECIPIENT`" variables and an additional "`sendmail`" command for the `CTRL` variable to implement an inter-agent messaging system.

[2] 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.

5 Application

We demonstrate the use of the toolkit by applying it to a simple marketing simulation of two mass marketer firms living in an artificial consumer market environment. The setting is in principle depicted in Figure 1—without the explicit modeling of the firm's departments, though. The simulation goal is to study the influence of marketing budget and revision cycles of advertising policy on two firms representing two mass marketing strategies.

- The artificial consumer world the two firms are embedded in is described in detail by Buchta & Mazanec (2001): products consist of a predefined number of technical features on the one side and promotional attributes ("claims") communicated to the consumers on the other. Both consolidate in latent constructs inside the consumers (their internal representation of the product). Buying decisions are made upon the concordance of the internal scores on these latent dimensions with the consumers' preferences (i.e., ideal products). To keep things simple, we assume a pure advertising-driven market, i.e., product features are never discordant with the advertising messages (this is a realistic setup, e.g., for mass consumer articles like detergents). We use 12 perceived attributes and 4 latent dimensions, i.e. 3 attributes load on 1 latent variable. We model a consumer population with heterogenous preferences (the preference structure varies from consumer to consumer in a controlled way). Note that technically, consumers are handled through a single "consumer population" agent which performs the necessary computations for all consumers—but the model is still at a disaggregated level.
- The two firms are implemented separately by two autonomous agents: each firm produces one product, chooses the whole population as target for the advertisement, and selects an advertising profile determining the attributes included in the message ('claims'). The actual impact of the message on the sales figures depends also on the promotional budget used for the marketing activities in this period. Because both agents use a mass marketing strategy, the decision problem is reduced to finding an appropriate message (a subset of the perceptual attributes of the consumer market which a firm claims for its product) and a price. Each agent is guaranteed to claim at least one attribute. Both agents use the market share weighted average price. The firms' policies differ in the following:
  - The 'naive' mass marketer derives its claims by selecting attributes with above average perception (the most prominent if there are more than 5).
  - The 'differentiating' mass marketer derives its claims by selecting attributes that do not receive the attention of competing agents.

  Both agents revise their strategy depending on the factor 'thinking cycle' either every period or only in every sixth period. We also varied the influence of the per period marketing budget (100 or 200) on the firms' success.
- The actual simulation is an iterative process: at the beginning, the market is initialized with random perceptions of the firms' brands (normal distribution about the center of the perceptual space). Then, in each period, first the firms make their advertising decision, and then, the consumers' choices and new perceptions ("beliefs") are computed by the consumer population agent. These steps are repeated for 12 periods. The whole run is repeated for all 4 factor combinations. The whole design plan is repeated 20 times.

The `simulation.xml` for this simulation design looks as follows:

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

<simulation>
  <alldesigns repeat="20" cycles="12">
    <agent name="foo.mass.naive" level="1" copies="1"/>
    <agent name="foo.mass.diff" level="1" copies="1"/>
    <agent name="consumers" level="2" copies="1"/>
  </alldesigns>
```

```
  <design name="1">
    <allagents>
      <p name="budget">100</p>
      <p name="think">1</p>
    </allagents>
  </design>
  ...
  <design name="4">
    <allagents>
      <p name="budget">200</p>
      <p name="think">6</p>
    </allagents>
  </design>
</simulation>
```

In this simulation, the parameters are varied simultaneously for both firms involved (common parameters), but agent-specific (private) parameters could also have been specified in the `<agent>` sections. For the statistical analysis, we computed the cumulated profits (sum of profits over all periods) for each agent and each run. The results are visualized in figure 5: for each factor combination, we see two boxplots representing the 2 firms' distributions of cumulated profits. It seems obvious that the thinking cycle makes a difference: whereas the boxplots overlap for cycle $= 1$, they do not for cycle $= 6$. The dependence on the budgets seems less sure, though. In order to obtain statistical evidence, we then analyzed the influence of the 2 factors on the lead of firm 2 (in terms of cumulated profits) by the means of a simple Analysis of Variance (ANOVA), assuming a linear influence of these factors. Because we are interested in the relative performance of the agents only, we used the *differences* of cumulated profits for the two agents, yielding 80 values in total, 20 for each factor combination. The model includes the 2 main effects ('budget' and 'cycle') as well as the interaction effect of the joint factor ('budget:cycle'). The R^2 for the model is 0.2648, so the percentage of explained variance is not very high. The P-values for the model are indicated in Table 1: 'cycle' is highly significant with $P = 0.0125$, whereas both the 'budget' main effect and the interaction effect are not.

factor	P-value
cycle	**0.0125**
budget	0.5997
cycle:budget	0.1699

Table 1: ANOVA P-values

From these results we might conclude that in a setting with two competing mass marketing firms, the naive approach outperforms the differentiating one if the strategy is not revised every period.

6 Conclusion

In this work, we presented a framework for agent based simulations, allowing for flexible parameter specification on the one side and easy integration of legacy code from high-level programming environments on the other. Although the existing framework is powerful enough to run usual 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. The same inconvenience applies to communication between agents (e.g., inter-departmental communication). Future work will include the port of all tools to other platforms (at least Windows), which should be an easy task in theory (all components, like JAVA and XML, are supported), and the extension of the currently rather strict interface definition: we plan to experiment with the extension of the wrapper to handle

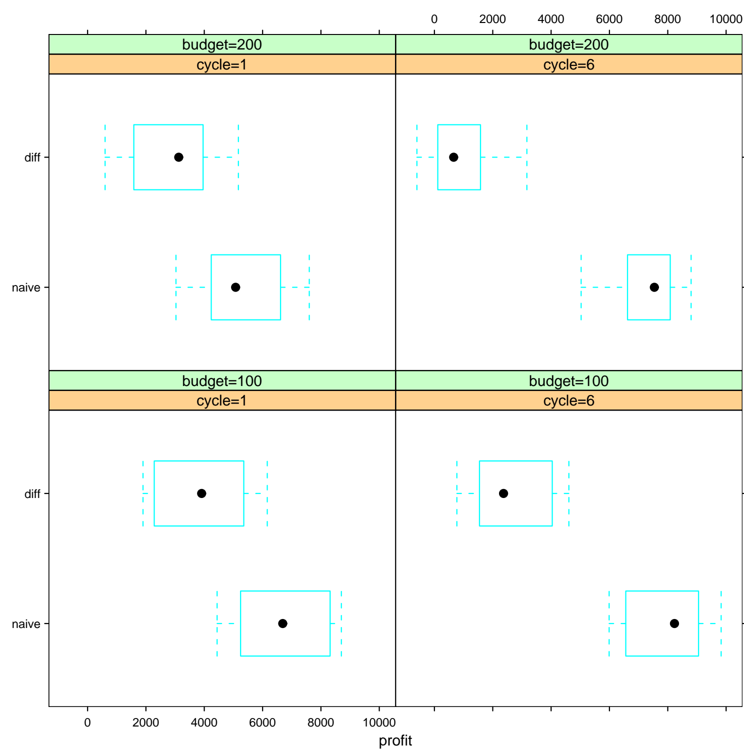


Figure 5: Boxplots of Cumulated Profits for the Two Firms.

dynamic interfaces, allowing the export of arbitrary defined functions. Further, the use of CORBA seems a promising tool to allow for simulations with a dynamic component set: agents distributed over a LAN or even the Internet could be added and removed at run-time to a centrally coordinated simulation.

Appendix A: `simulation.dtd`

```
<!ELEMENT simulation (alldesigns?, design+)>
<!ATTLIST simulation seed       CDATA #IMPLIED
                     mailserver CDATA #IMPLIED
                     notify     CDATA #IMPLIED
                     timeout    CDATA "0">

<!ELEMENT alldesigns (allagents?, agent*)>
<!ATTLIST alldesigns repeat     CDATA "1"
                     cycles     CDATA "1">
<!ELEMENT design (allagents?, agent*)>
<!ATTLIST design name        CDATA #IMPLIED
                 repeat      CDATA #IMPLIED
                 cycles      CDATA #IMPLIED>

<!ELEMENT allagents (p*)>
<!ATTLIST allagents level    CDATA #IMPLIED
                    copies   CDATA #IMPLIED>
<!ELEMENT agent (p*)>
<!ATTLIST agent name         CDATA #REQUIRED
                level        CDATA #IMPLIED
                copies       CDATA #IMPLIED
                seed         CDATA #IMPLIED>

<!ELEMENT p (#PCDATA)>
<!ATTLIST p name             CDATA #REQUIRED>
```

Appendix B: `wrapper.dtd`

```
<!-- wrapper DTD version="$Revision: 1.1 $" -->

<!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 printdone  (#PCDATA)>
```

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

<!ELEMENT name        EMPTY>
<!ELEMENT value       EMPTY>
```

References

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

Decker, K. (1996). Task environment centered simulation.

Genesereth, M. R. & Ketchpel, S. P. (1994). Software agents. *Communications of the ACM*, **37**(7), 48–53.

Gulyás, L., Kozsik, T., & Fazekas, S. (2002). The multi-agent modeling language. http://www.syslab.ceu.hu/maml/.

Kilgore, R. A. (2000). SILK, JAVA and object-oriented simulation. In *Proceedings of the 2000 Winter Simulation Conference*, pp. 246–252.

Krahl, D. (2000). The extend simulation environment. In *Proceedings of the 2000 Winter Simulation Conference*, pp. 280–289.

Minar, N., Burkhart, R., Langton, C., & Askenazi, M. (1996). The swarm simulation system. a toolkit for building multi-agent simulations. http://www.santafe.edu/projects/swarm/overview/overview.html.

Nwana, H. S., Ndumu, D. T., Lee, L. C., & Collis, J. C. (1999). ZEUS: a toolkit and approach for building distributed multi-agent systems. In Etzioni, O., Müller, J. P., & Bradshaw, J. M. (eds.), *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pp. 360–361, Seattle, WA, USA. ACM Press.

Perriollat, F., Skarek, P., & Varga, L. (1994). *Cooperating Expert Systems in Accelerator control—Results and CERN's contributions to the ESPRIT-II ARCHON Project*. Tech. rep., CERN.

Richter, H. & März, L. (2000). Towards a standard process: The use of UML for designing simulation models. In *Proceedings of the 2000 Winter Simulation Conference*, pp. 394–398.

Wilson, L. F., Burroughs, D., Sucharitaves, J., & Kumar, A. (2000). An agent-based framework for linking distributed simulations. In *Proceedings of the 2000 Winter Simulation Conference*, pp. 1713–1721.

Wittig, T., Jennings, N. R., & Mamdani, E. H. (1994). ARCHON — A framework for intelligent cooperation. *IEE-BCS Journal of Intelligent Systems Engineering — Special Issue on Real-time Intelligent Systems in ESPRIT*, **3**(3), 168–179.

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