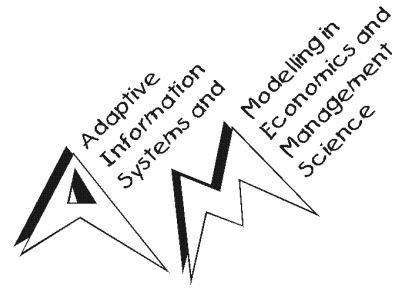


Working Paper Series



SIMENV: A Dynamic Simulation Environment for Heterogeneous Agents

David Meyer
Christian Buchta

Working Paper No. 100
August 2003

Working Paper Series



August 2003

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

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 a format based on the Extensible Markup Language (XML) that is simple-structured yet still enables the design of flexible models, with the possibility of varying both agent population and parameterization. Further, the tool allows the definition of communication channels to single or groups of agents, and handles the information exchange. Also, both (groups of) agents and communications channels can be added and removed at runtime by the agents, thus allowing dynamic settings with the agent population and/or communication structures varying during the simulation time. A second 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: for this, 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). Again, this mapping is made by an XML-based definition format.

1 Introduction

1.1 The Simulation Concept

Agent-based simulations are often implemented by using an object-oriented style of programming, allowing for detailed modeling of the artificial actors. In the following, we consider an agent-based economic simulation in which economic entities (firms, (groups of) consumers, investors, markets, and the like) can be thought of as interacting agents. A typical simulation combines several agents, defines their relationships, and observes their resulting interactions over time.

After the simulation design has been defined (Richter and März, 2000), running a simulation usually amounts to writing a control program in one's favorite programming language, named the *Simulation Manager* (see below), that coordinates a set of previously implemented, autonomous agents.

One might wish that the agents would have standardized interfaces so that they automatically have the same bindings allowing their use in simulations as modularized components. General mechanisms for providing standardized interfaces (like CORBA) do exist, but usually require advanced programming skills. Our objective, then, is to provide an easy-to-use mechanism suitable for use in data-analytical environments like MATLAB (The Mathworks, Inc., 2003), Octave (Eaton, 2003), or R (R Development Core Team, 2003), as they offer convenient ways to analyze simulation results and are also (typically) used for implementing objects and methods. We also deal with varying parameters in controlled experiments and provide a scheduling scheme to determine the order of invocation within a single experiment (design) and the number of runs (periods) per design.

Consider the simple introductory example involving two competing firms, named "Firm A" and "Firm B", respectively, operating in 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 (such as transaction control), 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.

As an extension to Meyer et al. (2001) and Meyer et al. (2003), our software also offers a simple yet flexible communication system which can be used for direct information exchange between the agents, without the need of using a database. Also, there might be a need for dynamic creation of agents (when, e.g., new department or daughter firm agents shall be created) and/or communication channels (e.g., when two firms decide to collaborate). Both can be done at runtime during a simulation run.

A prominent role in the SIMENV framework is played by XML, the Extensible Markup Language, which will be used for the specification of the simulation setup and the interface definitions. XML is a set of rules, guidelines and conventions (World Wide Web Consortium, 2000) for designing text formats for data so that files are easy to generate, computer readable, and unambiguous. Data are structured by

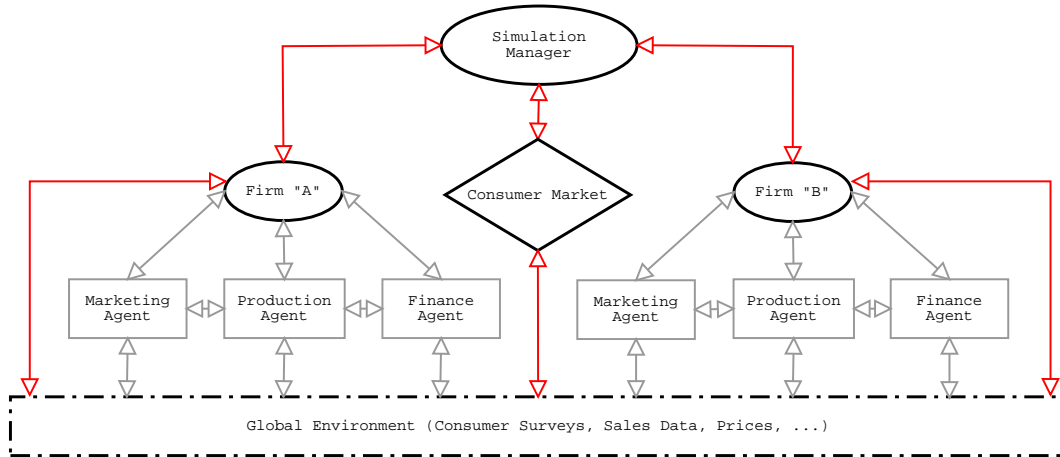


Figure 1: A Simple Simulation with two Competing Firms, each consisting of 3 Agents for Marketing, Production and Finance, respectively.

XML elements in a hierarchical and parameterizable way, allowing for easy computer-based information processing. XML files avoid common pitfalls such as lack of extensibility, lack of support for internationalization/localization, and platform-dependency. In addition, the syntax of XML files can formally be specified in “document-type-definition” (.dtd) files enabling formal validation (the .dtd files for the formats described in this paper are listed in the Appendix). In the context of interfacing systems, XML has been used by, e.g., [Wilson et al. \(2000\)](#) to describe port-based models (systems with interfaces and exchangeable implementations).

1.2 Review of Related Work

A large number of test beds for agent-based simulations exists, mostly complete toolkits with graphical user interfaces and ready-made components, but which are typically designed for particular tasks. [Tesfatsion \(2002\)](#) gives a good overview on these “Computational Laboratories”. Prominent examples are the work of [Valente and Anderson \(2002\)](#), applying the “Laboratory for Simulation Development” for evolution simulation modeling to the Nelson-Winter Model of Schumpeterian competition in an industry or economy, or the Aspen Microanalytical Simulation Model of the U.S. Economy developed by [Pryor et al. \(1996\)](#); further examples are the SME Spatial Modeling Environment from [Maxwell et al. \(2002\)](#), or z-Tree (Zurich Toolbox for Readymade Economic Experiments) developed by [Fischbacher \(2002\)](#) which successfully has been applied to public good experiments, structured bargaining experiments, and market experiments with auction pricing models. A crucial usability criterion is the possibility of integrating existing with new code, or code from heterogeneous platforms. The latter is imperative when a larger group of scientists—possibly from different institutions or even countries—work together in a joint project (see, e.g., the [Austrian Science Foundation project SFB 010—Adaptive Information Systems and Modeling in Economics and Management Science, 1999](#)) as it is unlikely that all people use the same platform and the same programming environment. As for the integration of legacy code, [Genesereth and Ketchpel \(1994\)](#), in their overview of agent-oriented programming, distinguish three possible mechanisms for this kind of “agentification”: a) building a “transducer” (writing a translating module; this requires only knowledge of the communications protocol), b) programming a “wrapper” (this usually requires the source code and a deeper understanding of technical details), and c) rewriting the code. In this framework, our work fits in the first category, we will nevertheless adopt the term “wrapper” for the corresponding module of our system.

One of the first attempts to integrate existing (semi-)autonomous systems into 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

exception handling (when, for example, one agent cannot find a solution to a particular problem, it requests help from other agents). The system has been applied to Electrical Networks and to CERN accelerator monitoring tasks. But as [Perriollat et al. \(1994\)](#) explains, 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: see, e.g., the macro-language extension MAML in [Gulyás et al. \(2002\)](#), but presupposes good programming skills. Modern developments are often JAVA-based. For example, the “Ascape” framework ([Parker, 2001](#)) which was inspired by the “SWARM” toolkit, “RePast” ([Collier, 1996](#)) and “SILK” ([Kilgore, 2000](#)) for simulation environment JAVA-classes, or the more user-friendly “TASK” environment ([Decker, 1996](#)), representing a JAVA-based test bed for abstract representations of task environments. All these frameworks 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 that 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 not support other platforms such as Linux.

All these approaches require some coding in a low-level programming language (e.g., Object C or JAVA) to integrate existing code, but scientists often use high-level computing environments such as MATLAB or R. Although the programming skills of scientists have certainly increased in the last decade, we assume that they would prefer to work at a conceptual level and not having to care about technical details of general-purpose, low-level languages such as JAVA or C++, whose mastery, in addition, necessitates a major time investment ([Gilbert and Banks, 2002](#)). Also, none of these systems seems 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 towards specifying simulations and the possibility of integrating a wide range of existing software.

The remainder of this work is structured as follows: First, we describe how the specification of the simulation settings is done in XML for a generic simulation manager, supporting multiple design specifications. Then, we explain the “normalization” of agent interfaces via a wrapping technique, thus allowing the simulation manager to treat all agents the same way. Section 4 deals with SIMENV’s communication mechanism, and is followed by a section on mechanisms for dynamic simulation setups. The last section treats the remaining control mechanisms (such as the meta-agent, handling of random number generation, and e-Mail notifications).

2 The Simulation Manager

2.1 A Typical Simulation Cycle

A complete simulation includes several designs with (typically) different parameter settings and/or a modified set of agents. Designs can be run repeatedly. 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 or skills, the simulation manager has to be capable of operating in a technically heterogeneous environment, and therefore is implemented in JAVA ([Gosling et al., 2000](#)), a platform independent programming language, that offers good support for network communication. Although simple in design, we consider it powerful enough to be

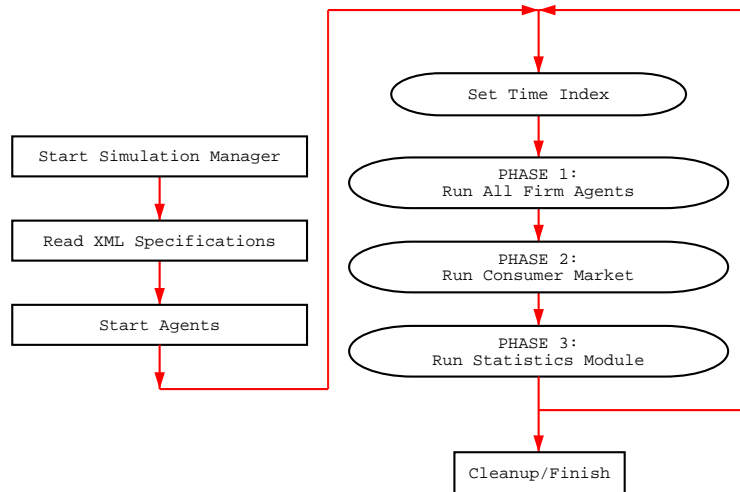


Figure 2: A Typical Simulation Cycle

used as a ready-made tool. It is capable of dealing with an arbitrary number of agents in different phases (e.g., a market clearing agent should only be started when all “normal” agents are done) by varying an arbitrary set of parameters through different designs. These parameters (such as market/product characteristics, initial prices, and 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 (propagated to specific agents). Usually, public information also includes technical information, like the current period (updated at the begin of each cycle), or the agent identifier (which the agent might include in its output information). The simulation components are specified in a definition file read by the simulation manager at startup. As mentioned above, we use XML to define these settings.

2.2 Using XML for Simulation Settings

The SIMENV framework is based on an object-oriented approach. Conceptually, we suppose the existence of agent classes with methods (functions) and attributes (parameters). A simulation setup consists of assigning one or more instances of these classes to run levels, along with a certain number of parameters. These assignments can vary from design to design. For example, the definition file for a sample simulation including firms and consumers with two designs might look as follows:

```

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

<simulation>
  <alldesigns repeat = "20" cycles = "30">
    <agent name = "consumer" level = "2" instances = "100">
      <p name = "reservprice">5</p>
      <p name = "budget">10</p>
      <r name = "firm">demand</r>
    </agent>
  </alldesigns>
  <design name = "A">
    <agent name = "firm" level = "1" instances = "2">
      <p name = "type">mass</p>
      <p name = "budget">100</p>
      <r name = "consumer">offer</r>
    </agent>
  </design>
</simulation>

```

```

<design name = "B">
  <agent name = "firm" level = "1" instances = "2">
    <p name = "type">niche</p>
    <p name = "budget">50</p>
    <r name = "consumer">offer</r>
  </agent>
</design>
</simulation>

```

The tags are described in the following:

- 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.
- The document starts with the <simulation> root tag. This tag has several parameters, which are described in Section 6 on control structures. <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, as has been done for the consumer agents. Each <design> tag may contain an arbitrary number of

- <agent> tags with the attributes name, (number of) instances, and level (the phase, in which the agent is scheduled to run). Parameters common to all agents can be put into an optional <allagents> section. Note that the <alldesigns> section mentioned above may also contain an <allagents> section, whose parameters would be copied into all agent-specific <allagents> sections, and subsequently into all <agent> sections. Hence, 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. Each agent "inherits" the parameters from the corresponding <agent> section in the <alldesigns> section, if any, as well as all parameters from a possibly existing <allagents> section (the latter inheriting for its part from the <allagents> section in the <alldesigns> section).
- <r> tags are used for defining communication channels and are discussed later in Section 4.

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.

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 plan) are usually created in an automated way, but the structure also enables more elaborated, fractional plans: 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 (see, e.g., [Dey and Mukerjee, 1999](#)), thus substantially reducing the simulation time needed.

So far, we described how parameters are specified in simulations. Now, we move to the agents' side to see how the methods are defined.

3 Agent Specification

One of the basic motivations for SIMENV was the need for integrating agents implemented in (possibly heterogeneous) high-level programming environments. To make such 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.

3.1 Wrapping Agents

First, we describe the program acting as an intermediary, translating the simulation manager’s JAVA calls to the native method calls in the agent’s programming environment. This program “wraps” the agent and exports through JAVA a standardized interface (we refer to this program as the *wrapper*). The translation of the agents’ interface is stored in an XML-based interface definition format, which (mainly) defines one-to-one correspondences to the JAVA interface calls.

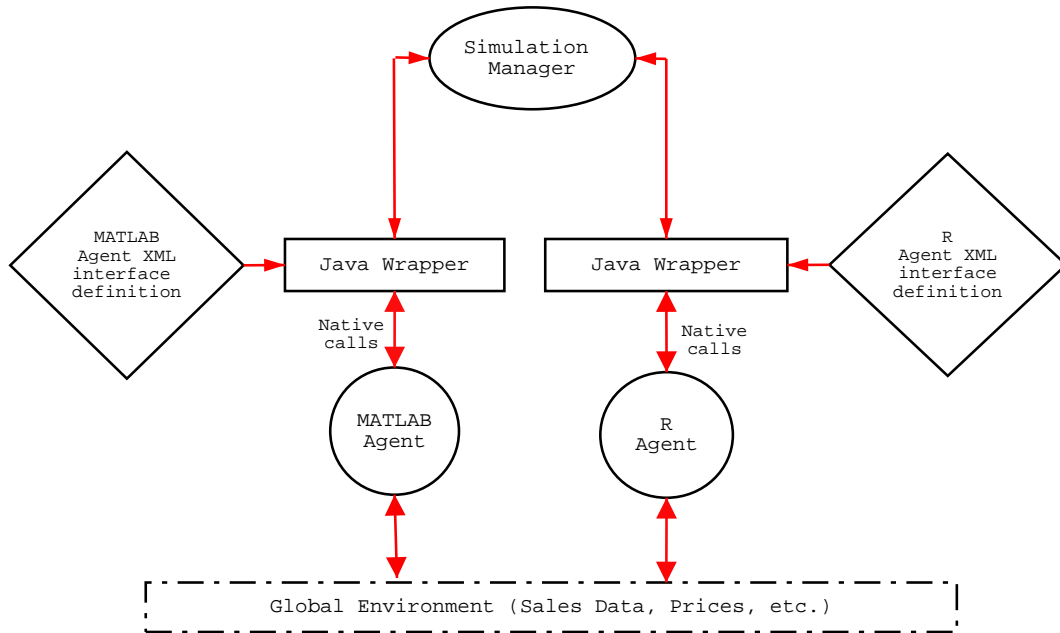


Figure 3: A Simple Simulation with Two Agents

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 that starts and synchronizes the simulation components (agents). The central coordinating agent (simulation manager) makes the JAVA calls to the wrapper, and the latter—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. Note that SIMENV currently targets interpreter-based environments only, redirecting their standard input and standard output devices. Compiled code (from, e.g., C or Fortran programs) can be integrated using a command shell as “interpreter”, which subsequently is used to call (execute) binary programs instead of making function calls.

Next, we explain how the user specifies these (interpreter-specific) function calls.

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” to derive the functionality to be handled by the wrapper. It can be summarized in the steps noted in Figure 4.

1. Start of the interpreter (MATLAB, R, ...).

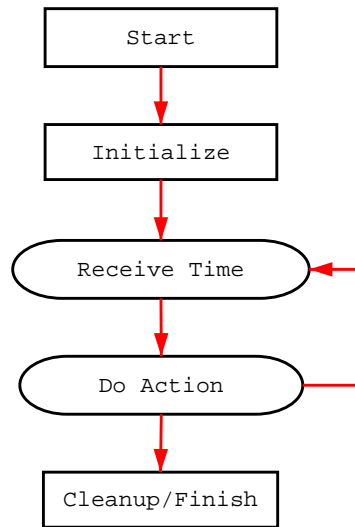


Figure 4: The Simulation Cycle (Agent's View)

2. Loading of the agent's source code (not needed 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, etc.
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.

3.3 Using XML for defining Agent Interfaces

The agent's interface is reduced to six 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 been executed completely and that the agent is ready for more commands.

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

```

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

<wrapper>
  <start>R --quiet --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>print(<name/>)</getattr>

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

```

For an implementation in Octave, the wrapper file would look like:

```

<?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>disp(<name/>)</getattr>

  <printdone>printdone</printdone>
  <donestring>:</donestring>
</wrapper>

```

The tags are described in the following:

- 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 that sets the various attributes of the agent (e.g., TIME or NAID). 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 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.¹ Also note

¹If 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.

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 of course are free to choose to use or not to use this information. This is another reason why we recommend implementing explicit functions for the parameter handling, allowing to keep the name space clean and to filter unused information. Currently, the public information set includes: `TIME` (the current time index), `SEED` (the current seed to be used for random number generation), `NAID` (the agents' internal unique identifier), and `ANAME` (the agents' full name). The latter has the format `NAME.INSTANCE`, where `INSTANCE` is an integer value and designates the copy number. In addition, `NRID` (the current replication), `NDID` (the unique internal design identifier), and `DNAME` (the optional design name) are passed to the special meta agent (see Section 6). The `<wrapper>` tag has an optional parameter: `separator`, which can be used to replace the dot separator (".") by another character, as the dot is not allowed in variable names in all programming environments.

On the other hand, the simulation manager may also retrieve some information from the agent (as specified in `<getattr>`): currently, only three variables—`CTRL`, `CTRL.TARGET`, and `CTRL.PARAMETERS`—are scanned. These “control” variables are used to pass commands to the simulation manager and are described in Section 5 on dynamic settings.

4 Communication Structures

Evidently, agents must be enabled to interact, for it is the outcome of this interplay which is of interest in agent-based simulations. In addition, there are simulation setups explicitly focused on the study of communication structures and cooperation (see, e.g., [Forrest and Jones, 1995](#); [Epstein and Axtell, 1996](#); [Axelrod, 1997](#)). One possibility is the use of a database, modeling the agents' global environment. But databases clearly have two main disadvantages: the simulation performance significantly decreases, while on the other hand the implementation complexity increases. The user (scientist) is forced to take care of database design to avoid redundancies, and to account for transaction control in the agents' program code to avoid concurrency problems. Therefore, we offer an alternative facility for information exchange, namely the specification of direct communication channels from one agent to another or to a group of agents. These channels can subsequently be used for the transmission of character-string messages. Note that this is not a too severe restriction as character strings can encode data objects of great complexity when one uses a structuring meta-data language like XML. For example, [Meyer et al. \(2004\)](#) have designed an XML-based format for statistical data allowing, e.g., for multidimensional arrays and recursive, tree-like list structures, which should suffice for most applications.

We distinguish three types of channels: “one-to-one”, “one-to-group”, and “one-to-all” (or “broadcast”). “one-to-one” relates one instance of a class to another instance of (possibly the same) a class. A “one-to-group” relation targets all instances of a class (again, possibly the own class). “broadcast” informations obviously are passed to all instances in the simulation.

The specification is quite simple and done using `<r>` tags in the `<agent>` sections of the XML design specification. All channels are unidirectional and specified at the “outgoing” side. Each channel is composed of a target name and the name of the exchange variable which will contain the message to be sent (the “outbox” in terms of a mailing software). If the agent's full name (“`NAME.INSTANCE`”) is specified, a “one-to-one” relationship is defined. If only “`NAME`” is used, all members of the class are targeted. If no name is given, a “broadcast” channel is defined. In our introductory XML example, the firm agents define a relationship using the variable “`offer`” targeting all consumer instances:

```
<agent name = "firm" level = "1" instances = "2">
  ...
  <r name = "consumer">offer</r>
</agent>
```

A channel to, say, consumer instance 23 would look like this:

```

<agent name = "firm" level = "1" instances = "2">
  ...
  <r name = "consumer.23">offer</r>
</agent>

```

and a “broadcast” channel (that is, to all consumers and to the second firm) could be defined as simply as:

```

<agent name = "firm" level = "1" instances = "2">
  ...
  <r>offer</r>
</agent>

```

Note that group or broadcast messages are not delivered to the sender itself.

Collection and delivery of mails is done in one step after the agents’ `init` phases (allowing dynamic initializations like random-generated start scenarios) and after all `action` calls of one level. It follows that messages can only be processed either in the next level of the current period or in the first level of the next period, depending on when the targeted agent is next called. The Simulation Manager collects mails by applying the `getattr` call of the sender agents on each registered communication variable. At the target side, delivery is done by setting a variable with the unique name “SENDER.INSTANCE.VARIABLE” to the message string using the `setattr` method of the target agent (INSTANCE and VARIABLE name relate to the sender agent). When the target agent does not exist, the message is ignored. As the full target name might be an overkill for simple settings involving only one instance per class, the `<simulation>` tag offers a `full.names` parameter which, when set to “false”, causes the use of the “VARIABLE” name only during delivery. Note, however, that name clashes between channels using the same variable name for different contents sent to the same target agent are not checked and this option, therefore, should be used with care.

5 Dynamic Settings

Some kind of simulations, in particular in the context of evolutionary research and network industries, necessitate a dynamic setup, that is, agents and/or communication channels are created and discarded during the simulation (see, e.g., Kirman, 1997; Tesfatsion, 1997; Zandt, 1998; Vriend, 1995). These dynamics are handled by the SIMENV framework using special control variables at the agent side: “CTRL”, “CTRL.TARGET”, and “CTRL.PARAMETERS”, which can be used by agents to alter the initial setting defined in the XML design file. Currently, four CTRL commands are handled: “start” and “stop” for the instantiation of new agents, and “commAdd” and “commRemove” for the construction and destruction of communication channels. CTRL variables are scanned and possible commands are executed right before message exchange takes place.

New agents:

The “stop” command simply causes the current agent to be removed from the simulation after all `action` calls of the current level are done, using the commands defined in the `<finish>` and `<quit>` sections.

The “start” command uses additional parameters specified in the CTRL.TARGET and CTRL.PARAMETERS variables: CTRL.TARGET is a semi-colon separated list of class names from which instances shall be created. If a class name is in the format: “classname.number”, “number” instances of this class are created. CTRL.PARAMETERS is a semi-colon separated list of comma separated parameter lists in the format “NAME=VALUE” initializing the new agents, each sublist complementing the corresponding class name in CTRL.TARGET. Using the following piece of code:

```

CTRL = "start"
CTRL.TARGET = "classA;classB.3"
CTRL.PARAMETERS = "var1=123,var2=test;var3=456"

```

one new instance of `classA` and three new instances of `classB` are created, the former initialized with variables `var1` and `var2`, the latter with variable `var3`. There are two special variable names: `LEVEL` and `SEED`, which have the same meaning and effects as the corresponding parameters in the `<agent>` sections of the XML design specification described in Section 2.2.

New communication channels:

Creation and destruction of communication channels is specified in a similar way, setting the `CTRL` variable to `"commAdd"` and `"commRemove"`, respectively. `CTRL.TARGET`, now, is a semi-colon separated list of target instances in the same format than the `<r>` tag in the `<agent>` sections described in Section 4, with the exception that broadcast channels are specified with a space character (" ") instead of an empty string. `CTRL.PARAMETERS` is a semi-colon separated list of corresponding exchange variables. The following three statements:

```
CTRL = "commAdd"
CTRL.TARGET = "classA.3; ;classB"
CTRL.PARAMETERS = "instMsg;broadcastMsg;classMsg"
```

would create three new channels: a one-to-one channel to instance 3 of `classA` using variable `instMsg`, a broadcast channel using variable `broadcastMsg`, and a one-to-group channel to all instances of `classB` using variable `classMsg`. The `"commRemove"` command is specified in the same way than `"commAdd"`.

6 Control Issues

In this final section, the technical control parameters of the `<simulation>` tag and the specification of the `meta` agent are described.

6.1 The “meta” Agent

The `meta` agent differs from the other agents in several ways:

- It is only created once at the beginning of the simulation, that is, “survives” the beginning of new replications and designs unlike the other agents which are restarted at these occasions.
- It has full information on the simulation schedule, that is, in addition to `TIME` also gets `NDID/DNAME` (design number/design name) and `NRID` (replication number).
- The `init`, `finish`, and `action` methods are replaced by several other methods, allowing the `meta` agent to perform tasks other agents cannot: `<preSim>` and `<postSim>` are called before/after a simulation is started/stopped, `<preDesign>` and `<postDesign>` before and after designs, `<preRepeat>` and `<postRepeat>` before and after replications, and `<preRun>` and `<postRun>` at the beginning and at the end of every period. Typical applications for these methods are database management (initialization, cleanup between designs) and logging.
- The `meta` agent is passive: it can receive messages (e.g., for logging purposes), but is not able to send messages or to start agents, as it is not expected to influence the simulation itself.

6.2 Technical Parameters of the `<simulation>` Tag

The `<simulation>` tag allows the specification of 7 optional parameters: `seed`, `mailserver`, `mailto`, `mailfrom`, `timeout`, `debug`, and `full.names`. The last parameter, `full.names`, has already been explained in Section 4, the effect of the others is detailed in the following:

Seeding:

The `<simulation>` and the `<agent>` tag allow the specification of optional seed parameters which can be used to control the initialization of the agents' random number generators, allowing exact replication of whole simulations. The seeds are created as follows. The master seed is taken from the `<seed>` parameter in the `<simulation>` tag (if omitted, it is drawn at random from a discrete uniform distribution) and used to produce seeds for each agent. These seeds can be overloaded at the agent level by using the "local" agent `<seed>` parameter. If multiple instances 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.

Status e-Mails:

The `<simulation>` tag further allows the specification of `mailserver`, `mailto`, and `mailfrom`. If all three parameters are set, the simulation manager notifies the indicated e-mail recipient of normal or abnormal termination of a simulation. This convenience feature has been added for simulations with long run-time.

Timeout:

The `timeout` parameter is used for detection of non-terminating agents (possibly due to programming errors or dead locks). If set to a positive value, a call to an agent function taking more than `timeout` seconds generates a runtime exception. Therefore, this parameter should be chosen with care.

Debugging level:

SIMENV offers three levels of verbosity for its log messages: 0, 1, and 2, which can be specified using the `delay` attribute. Level 0 is as quiet as possible, only the beginning of a new design, replication, and cycle is logged. Level 1, in addition, also logs the parsed tree of the XML files, communication activities, and all commands sent to the agents (with lines clipped at 256 characters). Level 2, finally, is more verbose for the XML parsing, and does not clip lines of the output.

7 Summary

In this work we introduced SIMENV, 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 a format based on the Extensible Markup Language (XML), that is simple-structured yet still enables the design of flexible models, with the possibility of varying both agent population and parameterization. Further, the tool allows the definition of communication channels to single or group of agents, and handles the information exchange. Also, both (groups of) agents and communications channels can be added and removed at runtime by the agents, thus allowing dynamic settings with a agent population and/or communication structures varying during the simulation time. A further 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: for this, we presented 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.

As experiments have shown, a number of open issues could be addressed regarding the SIMENV framework with respect to the design and wrapper modules:

Design: Currently, the scheduling scheme for agents, although the specification is very flexible, is fixed once the simulation is started. Simulation settings with complex, unpredictable agent behavior could necessitate more flexible, endogenous arrangements of agent calls. A solution to this might be to

implement event queues as used in [Collier \(1996\)](#) where call elements could be put in by agents at runtime, thus altering the original agent invocation order induced by the design specification. This, however, would significantly increase the complexity of possible simulation runs (with, e.g., the risk of endless loops due to mutual calls). Another weakness of the current scheduling scheme is that each agent runs in a separate programming environment which, in addition, is restarted at the beginning of every new design replication. When several agents on one machine use the same environment, this results in unnecessary increase of memory load, and currently inhibits the use of a greater number of agents in one simulation. In most cases, it should suffice to use only one instance of a programming environment and switch the agents when needed, by saving/restoring the current/last state of objects and variables. As for the specification of parameters, it could be helpful to define default values on the agent side, reducing the amount of parameters which has to be specified in dynamic settings. Also, the specification currently does not distinguish between fixed and varied (i.e., design) parameters. When parameters would be given a type attribute (or design parameters declared as such), the reporting tasks of statistics/logging agents could greatly be facilitated.

Wrapper: The current design was conceived for interpreter-based environments. Compiler languages have to be wrapped using a “shell agent”, and by writing separate programs for each of the calls the wrapper defines. At least, a native C/C++ interface could be useful. Furthermore, current simulations can only be distributed on several machines by using mechanisms like MOSIX. Because of the modular implementation of SIMENV (wrapper and simulation manager are separate classes) and the intrinsic network features of JAVA, it should be very simple to make the wrapper “Internet aware” by using, e.g., the JAVA RMI (Remote Method Invocation) protocol, or the more versatile (but far more complex) CORBA framework (see, e.g., [Siegel, 2000](#)). Simulation manager and wrapper programs could then be run on different machines and the method invocations routed over the LAN or the Internet. Finally, another important issue is hierarchical wrapping, that is, agents controlling other wrapped agents (like departments of a firm agent). This could be done now in principle when the interpreter has JAVA bindings, but the controlling agents would be entirely responsible of the simulation manager tasks for its sub-agents. The only practical solution currently seems to define the agent hierarchy at the meta-level, and to sort of “linearize” all agents in the simulations using different run phases for different hierarchical levels.

A Appendix

simulation.dtd

```
<!-- simulation DTD version="$Revision: 1.3$" -->

<!ELEMENT simulation (all?, meta?, alldesigns?, design+)>
<!ATTLIST simulation seed          CDATA #IMPLIED
                    mailserver     CDATA #IMPLIED
                    mailto         CDATA #IMPLIED
                    mailfrom       CDATA #IMPLIED
                    timeout         CDATA "0"
                    debug          CDATA "1"
                    full.names     (false | true) "true">

<!ELEMENT all (p*)>

<!ELEMENT meta (p*)>
<!ATTLIST meta name                CDATA #REQUIRED>

<!ELEMENT alldesigns (allagents?, agent*)>
<!ATTLIST alldesigns repeat        CDATA #IMPLIED
```

```

                cycles      CDATA #IMPLIED>

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

<!ELEMENT allagents (p*)>

<!ELEMENT agent (p*, r*)>
<!ATTLIST agent name      CDATA #REQUIRED
                level      CDATA #IMPLIED
                instances  CDATA #IMPLIED
                seed      CDATA #IMPLIED>

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

<!ELEMENT r (#PCDATA)>
<!ATTLIST r name          CDATA ">

```

wrapper.dtd

```

<!-- wrapper DTD version="$Revision: 1.2$" -->

<!ELEMENT wrapper      (start, boot?, init?, action, finish?, stop,
                setattr, getattr, printdone?, donestring)>
<!ATTLIST wrapper      separator CDATA ".">

<!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>

```

meta.dtd

```

<!-- meta DTD version="$Revision: 1.2$" -->

<!ELEMENT meta      (start, boot?, preSim?, preDesign?, preRepeat?,
                preRun?, postRun?, postRepeat?, postDesign?,
                postSim?, stop, setattr, getattr, printdone?,
                donestring)>

```

```

<!ATTLIST meta      separator CDATA ". ">

<!ELEMENT start      (#PCDATA)>
<!ELEMENT boot       (#PCDATA)>
<!ELEMENT preSim     (#PCDATA)>
<!ELEMENT preDesign  (#PCDATA)>
<!ELEMENT preRepeat  (#PCDATA)>
<!ELEMENT preRun     (#PCDATA)>
<!ELEMENT postRun    (#PCDATA)>
<!ELEMENT postRepeat (#PCDATA)>
<!ELEMENT postDesign (#PCDATA)>
<!ELEMENT postSim    (#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

- Austrian Science Foundation project SFB 010—Adaptive Information Systems and Modeling in Economics and Management Science (1999). *Project Report 1997–1999*. Vienna University of Economics and Business Administration. <http://www.wu-wien.ac.at/am/Download/report.pdf>.
- Axelrod, R. (1997). *The Complexity of Cooperation: Agent-based Models of Conflict and Cooperation*. The Princeton University Press, Princeton, N.J.
- Collier, N. (1996). RePast: An extensible framework for agent simulation. Software and Documentation available at <http://repast.sourceforge.net>.
- Decker, K. (1996). Task environment centered simulation. In Prietula, M., Carley, K., and Gasser, L., editors, *Simulating Organizations: Computational Models of Institutions and Groups*. AAAI Press/MIT Press.
- Dey, A. and Mukerjee, R. (1999). *Fractional Factorial Plans*. Wiley, Canada.
- Eaton, J. W. (2003). Octave software version 2.0.17, <http://www.octave.org/>.
- Epstein, J. and Axtell, R. (1996). *Growing Artificial Societies: Social Science from the Bottom Up*. MIT Press, Brookings, MA.
- Fischbacher, U. (2002). z-Tree: The Zurich toolbox for readymade economic experiments. Software and Documentation available at <http://www.iew.unizh.ch/ztree/>.
- Forrest, S. and Jones, T. (1995). Modeling complex adaptive systems with echo. *Complexity International*, 2.
- Genesereth, M. R. and Ketchpel, S. P. (1994). Software agents. *Communications of the ACM*, 37(7):48–53.
- Gilbert, N. and Banks, S. (2002). Platforms and methods for agent-based modeling. In *Proceedings of the National Academy of Sciences U.S.A.*, volume 99, pages 7197–7198.

- Gosling, J., Joy, B., Steele, G. L., and Bracha, G. (2000). *The Java Language Specification*. Addison-Wesley, second edition.
- Gulyás, L., Kozsik, T., and 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*, pages 246–252.
- Kirman, A. (1997). The economy as an interactive system. In Arthur, W. B., Durlauf, S. N., and Lane, D. A., editors, *The Economy as an Evolving Complex System II*, volume XXVII of *SFI Studies in the Sciences of Complexity*, pages 491–531, Reading, MA. Addison Wesley.
- Krahl, D. (2000). The Extend simulation environment. In *Proceedings of the 2000 Winter Simulation Conference*, pages 280–289.
- Maxwell, T., Villa, F., and Costanza, R. (2002). SME: Spatial modeling environment. Software and Documentation available at <http://www.uvm.edu/giee/SME3/>.
- Meyer, D., Buchta, C., Karatzoglou, A., Leisch, F., and Hornik, K. (2003). A simulation framework for heterogeneous agents. *Computational Economics*. Forthcoming.
- Meyer, D., Karatzoglou, A., Buchta, C., Leisch, F., and Hornik, K. (2001). Running agent-based simulations. Working Paper 80, SFB “Adaptive Information Systems and Modeling in Economics and Management Science”.
- Meyer, D., Leisch, F., Hothorn, T., and Hornik, K. (2004). StatDataML: An XML format for statistical data. *Computational Statistics*. Forthcoming.
- Minar, N., Burkhart, R., Langton, C., and 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., and Collis, J. C. (1999). ZEUS: a toolkit and approach for building distributed multi-agent systems. In Etzioni, O., Müller, J. P., and Bradshaw, J. M., editors, *Proceedings of the Third International Conference on Autonomous Agents (Agents’99)*, pages 360–361, Seattle, WA, USA. ACM Press.
- Parker, M. T. (2001). What is Ascape and why should you care? *Journal of Artificial Societies and Social Simulation*, 4(1). <http://www.soc.surrey.ac.uk/JASSS/4/1/5.html>.
- Perriollat, F., Skarek, P., and Varga, L. (1994). Cooperating expert systems in accelerator control—results and cern’s contributions to the esprit-ii archon project. Technical report, CERN.
- Pryor, R. J., Basu, N., and Quint, T. (1996). Development of Aspen: A microanalytical simulation model of the U.S. economy. Working Paper SAND96-0434, Sandia National Laboratories, Albuquerque, NM.
- Richter, H. and März, L. (2000). Towards a standard process: The use of UML for designing simulation models. In *Proceedings of the 2000 Winter Simulation Conference*, pages 394–398.
- R Development Core Team (2003). R software version 1.8.0, <http://www.R-project.org/>.
- Siegel, J. (2000). *CORBA 3 Fundamentals and Programming*. Wiley, second edition.
- Tesfatsion, L. (1997). A trade network game with endogenous partner selection. In Amman, H., Rustem, B., and Whinston, A. B., editors, *Computational Approaches to Economic Problems*, pages 249–269, Dordrecht, The Netherlands. Kluwer Academic Publishers.
- Tesfatsion, L. (2002). Agent-based computational economics: Growing economies from the bottom up. *Artificial Life*, 8(1):55–82.

- The Mathworks, Inc. (2003). MATLAB software: Release 13. Natick, MA: The Mathworks, Inc., <http://www.mathworks.com/>.
- Valente, M. and Anderson, E. (2002). A hands-on approach to evolutionary simulation: Nelson-winter models in the laboratory for simulation development. *The Electronic Journal of Evolutionary Modeling and Economic Dynamics*, 1003(1). <http://www.e-jemed.org/1003/index.php>.
- Vriend, N. J. (1995). Self-organizing markets: An example of a computational approach. *Computational Economics*, 8:205–231.
- Wilson, L. F., Burroughs, D., Sucharitaves, J., and Kumar, A. (2000). An agent-based framework for linking distributed simulations. In *Proceedings of the 2000 Winter Simulation Conference*, pages 1713–1721.
- Wittig, T., Jennings, N. R., and 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.
- World Wide Web Consortium (2000). *Extensible Markup Language (XML), 1.0 (2nd Edition)*. Recommendation 6-October-2000. Edited by Tim Bray (Textuality and Netscape), Jean Paoli (Microsoft), C. M. Sperberg-McQueen (University of Illinois at Chicago and Text Encoding Initiative), and Eve Maler (Sun Microsystems, Inc. - Second Edition). Reference: <http://www.w3.org/TR/2000/REC-xml-20001006>.
- Zandt, T. V. (1998). Organizations with an endogeneous number of information processing agents. In Majumdar, M., editor, *Organizations with Incomplete Information*, pages 239–305, Cambridge, UK. Cambridge University Press.