László Gulyás, Tamás Kozsik and John B. Corliss (1999)

The Multi-Agent Modelling Language and the Model Design Interface

Journal of Artificial Societies and Social Simulation vol. 2, no. 3, <http://jasss.soc.surrey.ac.uk/2/3/8.html>

To cite articles published in the *Journal of Artificial Societies and Social Simulation*, please reference the above information and include paragraph numbers if necessary

Received: 21-Oct-99 Published: 31-Oct-99

Abstract

While computer models provide many advantages over traditional experimental methods, they also raise several problems. The process of software development is a complicated task with high potential for errors, especially when it is carried out by scientists holding their expertise in other fields than computer science. On the other hand, the process of creating computer simulations of social systems which reflect the reality of such systems requires insights considerably beyond expertise in computer science. The Multi-Agent Modelling Language (MAML) is one of the efforts to ease these difficulties.

In its current version, MAML is a macro-language for Swarm (a freely distributed toolset under development at SFI), but it is also part of a larger Swarm-independent framework. Also, the design of MAML, while influenced by concepts from Swarm, is general enough to allow for later extension of the supported simulation kernels. This paper gives an overview of the mentioned larger framework, with special emphasis on MAML and its graphical CASE tool, the Model Design Interface.

Keywords: Social science simulation, Agent-based modelling, Integrated modelling environment

Introduction

1.1

In sciences which study complex systems, computer programs play an important role as scientific equipment. In the case of computer simulations the programs under use can be seen as experimental devices built in software. While computer models provide many advantages over traditional experimental methods, they also raise several problems. In particular, the process of software development is a complicated technical task with high potential for errors, especially when it is carried out by scientists holding their expertise in other fields than computer science (Minar et al). The converse is also true, that social systems, composed of human agents, are best modeled by researchers with extensive knowledge of these complex systems.

1.2

To ease the afore-mentioned difficulties and to strengthen the reliability of the developed simulation, and thus the reliability of the results gained, several simulation packages have been developed, or are under development. Our Telemodeling project falls into the second category: it is an ongoing research project carried out in the Complex Adaptive Systems Laboratory, CEU, Budapest. Its final goal is the creation of a framework that supports *all phases of computer simulation*, from the design of models, through parameter space search, to result-analysis. The interfaces between the modules of such a framework allow the scientists to produce and publish *results* that are *replicable and reusable* by others. Computer simulation cannot be considered a science before it meets these important requirements.

1.3

The kernel of the framework is a special purpose programming language, the *Multi-Agent Modelling Language*, or *MAML* for short. It serves for describing computer models with adequate tools. Elegance and simplicity is an advantage in modelling, but the scientists must remember that the really interesting models are usually the complex ones. The high-level language constructs of MAML make the *description of complex models* easier while ensuring the *simplicity and understandability of the computer program.*

1.4

A visual program building tool can further ease software engineering tasks. This idea gave birth to the *Model Design Interface* (or *MDI*, as we often refer to it). A model can be easily manufactured in this environment, which associates graphical representation with the elements of a MAML program.

1.5

MAML builds strongly on a freely distributed toolset, Swarm, which has a large, and constantly growing user community spanned all over the scientific disciplines, such as chemistry, economics, physics, anthropology, ecology, sociology and political science. This user community serves as a profound basis in the spread of Swarm-related model development tools.

1.6

This paper focuses on the Multi-Agent Modelling Language and the Model Design Interface. Firstly, in the rest of this section, our approach to the applied modelling paradigm is introduced. Section 2 gives an overview of the telemodeling project and designates the place of MAML inside the framework. MAML itself is investigated in Section 3. This is followed by the description of the Model Design Interface in Section 4. Finally, Section 5 outlines future and related efforts, and concludes the paper.

The modelling paradigm adopted

1.7

The modelling formalism that -- similarly to Swarm -- MAML adopts is agent-based modelling (Sumpter), within a discrete event simulation framework. That is, a *model* is made up of a collection of independent *agents* interacting via *messages and events* (Minar et al). An event is associated with a single *timestep* where it takes place. *Schedules* are used to generate events (or series of events, so called *plans*) in certain time-

steps: in fact these schedules define the control flow of the simulation program.

1.8

Every agent maintains a *state*, which can be changed through its set of *rules*. The activities performed by the agent during its life are described in terms of plans and schedules. The plans are action sequences that are placed on schedules. These schedules are often altered *dynamically*, following the change in the preferences and goals of the agents. An agent can have its own schedule, in which case it can carry out its plans or activities autonomously.

1.9

The model usually contains a huge and varying number of agents: agents can *be born* and *die* during the simulation. They also often *join or leave* certain *agent groups*. These groups define a collection of agents doing something together (e.g. communicating) or doing something similar (e.g. according to similar plans).

1.10

Although there are a lot of different approaches to agents (Wooldridge & Jennings 1995), even within the agent-based modelling community, we think that the concepts outlined above are a common base set for all of those; so whatever more specialized approach is being used, one can build upon our framework. The concepts introduced above will be further explained below.

The Telemodeling project

2.1

The Multi-Agent Modelling Language and the Model Design Interface are, in fact, part of a larger project that aims at supporting all phases of simulation within a single framework. The stages considered in the Telemodeling project (see Figure 1) include the design of a model, setting up search spaces of initial parameters, running simulations, storing data, and finally browsing and visualizing the results.

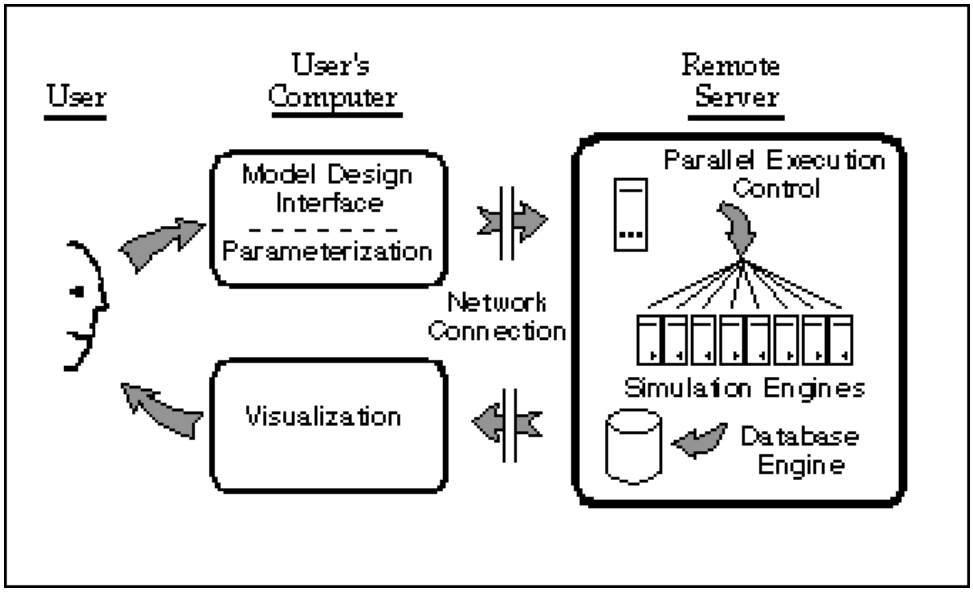

Figure 1: The Telemodeling Project

2.2

These stages are connected through a computer network: scientists can build their models on their workstations by model design tools downloaded through the Internet. The models then can be executed on high performance computers enabling simulations with large computational and storage demands. Later on, scientists can analyze the results of the simulations accessed from the server using data visualization tools downloaded from the network.

2.3

Although the Telemodeling project is still in its infancy, prototypes or case studies are available for all the important stages of simulation. Our MAML compiler, `xmc` is the first released product of this effort, forming the base for the model design toolset. In the following we summarize the main components of the telemodeling framework.

2.4

The Model Design Interface is based on the Multi-Agent Modelling Language, and offers a visual programming environment to design and implement models with. This component is in the focus of this paper, we will talk more about it in the following sections.

2.5

Parameterization is the process of associating series of values to the parameters of a model. In a scientific experiment the same simulation is run with different initial conditions, that is with different parameter values. Analyzing the relation between the values of the parameters and the outcome of the simulation is the main purpose of such an effort. Automating parameterization can be performed with the help of a program generating for example an HTML form, like the one on Figure 2. (Currently such HTML forms are created by hand.)

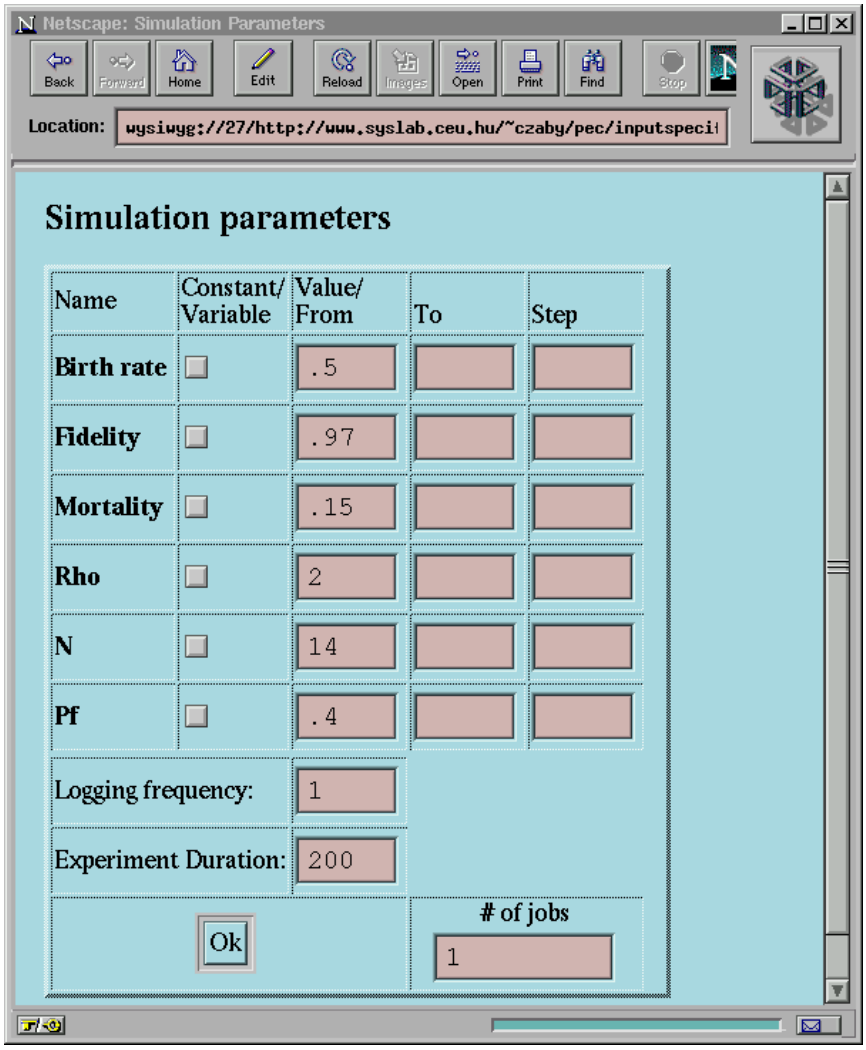

Figure 2: Parameterization

2.6

The Parallel Execution Control is responsible for co-ordinating the simulation instances according to the parameter value sets obtained from the previous step. The model and the parameter definitions are sent from the modeller's workstation to a server through a network. The server starts the simulation instances concurrently, if possible. As a proof of the idea we prepared an example where the different simulation instances were distributed on a local area network.

2.7

The Simulation Engines are the units running the simulation instances (the actual runs with a specified parameter set) and generating their results. Currently, we are using the Swarm package (and thus the Objective-C run-time) for this purpose, but from the framework's global point of view this component can be replaced with a different simulation environment.

2.8

The Database Engine is for collecting and storing the results of the simulation instances. In general, this database engine should be a professional, general-purpose database server. We have developed a case study, in which we defined a simplified description language to specify the format of the generated results, together with a simple software which is able to automatically merge the results of different simulation runs and store it in a database. This database can then be used during the result analysis. It is possible to visualize the data from different points of view without running the -- possibly very long -- simulations again and again.

2.9

Visualization is the final module in the framework. Querying the data collected during the simulation runs it allows the modeller to analyze the outcome of the simulation. He or she can utilize several visualization tools among the lot available on the market. We have also developed a couple of experimental visualization tools, the most interesting of which is the one based on an interactive virtual reality model (see Figure 3). This latter associates a VRML world with the simulation results: a general purpose VRML viewer can then be used to visualize the data.

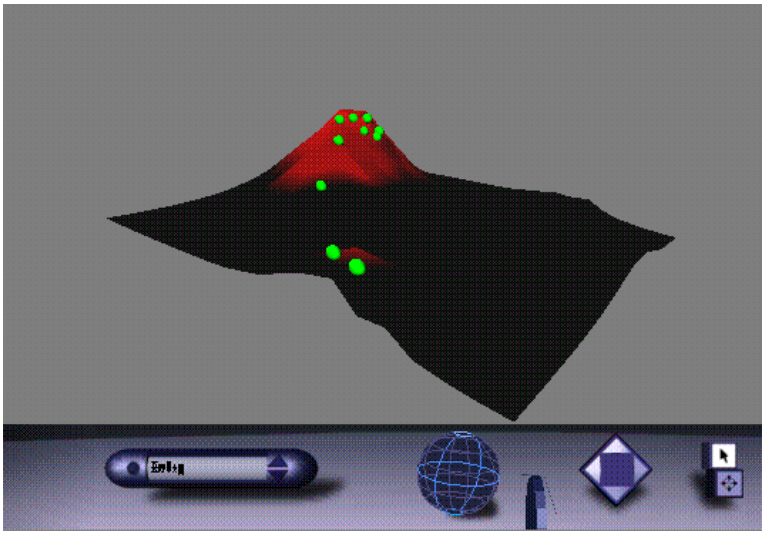

Figure 3: Visualization in VRML

The Multi-Agent Modelling Language

3.1

We have mentioned before that MAML and its compiler, `xmc`, is the first product of the telemodeling project. It does not mean that this is the final definition of the language, but simply, that a usable version with a fairly stable compiler has recently been completed. Throughout this paper we will concentrate on this (`v0.03`) release. [2]

3.2

At this early stage of the history of MAML the language relies strongly on the Swarm simulation package. Not only certain parts of a MAML program are written in Objective-C, the language, that Swarm is built upon, but also the `xmc` compiler produces Swarm code: this Swarm code should be further compiled by the appropriate C compiler, i.e. `gcc`. This architecture has the advantage that all the functionalities of the Swarm libraries (including random number and distribution generation, graphical and statistical tools, etc.) are available for MAML programmers.

3.3

MAML, similarly to Swarm, uses an object-oriented framework to define the model and its agents. This is justified by the fact that agents are in many ways similar to the objects in the object-oriented programming paradigm (Gulyás 1997). The state of an agent can be expressed as instance variables of an object, and its rules and communication channels can be coded as message handlers.[3] The object-oriented approach is also used by other simulation packages, e.g., by Ascape.

3.4

Swarm serves as a basis for MAML, and our language inherited a lot from the features of Swarm. Models, observers, agents, schedules, plans (or action groups in Swarm terminology), etc. are all formalized both in Swarm and MAML. However, there is a significant difference. While Swarm defines everything within the object-oriented framework, MAML takes one more step in abstraction. MAML is a *domain specific programming language* that, while exploiting the benefits of the object-oriented approach, explicitly implements the concepts of discrete-event simulation and agent-based modelling (see in section 1). Each of those concepts is mapped to a language construct of MAML, as it will be shown later in this section.

3.5

What are the advantages of this approach? A modeller -- especially a beginner in programming -- prefers thinking "in the problem domain", using the concepts of modelling, to thinking in terms of objects. On the one hand, model development becomes easier this way: the design and the implementation of a model get closer to each other. The implementation overhead of the object-oriented machinery can be avoided in many cases (see the example in paragraph 3.8), the code becomes shorter and more compact. This reduces the danger of making mistakes in the implementation, resulting in model behaviour dependent on these errors. On the other hand, software maintainability and understandability issues -- which are always important in software engineering -- really come to the front. A program, which describes the essence of the model without too much implementation details is much easier to maintain or just understood. This is an important aspect, because a computer simulation is a program that is used for making experiments: it is only a tool to discover and understand the simulated world. As a consequence, the program is being changed very frequently, and not only by its author, but also by other scientists, who try to understand, employ, modify, reuse or re-implement it. Finally we must not forget about the need for a common language spoken by agent-based modellers. This common language would then be used for publication of results, so making the description of models independent from implementation details is highly preferable.

3.6

In the following subsections an overview of MAML programming is presented.

The Model Construct

3.7

A *model* is the most important building block of MAML programs. It is implemented with a MAML construct, the one which is indicated with the `@model` keyword. The construct includes the name of the model and, between curly braces, its body.

```
@model School { ... }
```

The "components" of the model are defined in its body. For instance a model can contain agents. Agents are never described alone, but, similarly to the OOP terminology, *agent classes* can be defined. Agents are created as instances of the appropriate agent class. Again, a MAML construct, starting with the `@agent` keyword corresponds to agent classes. Analogously to models, the definition of an agent class includes a name and a body. This body contains the components of the agent class definition.

```
@agent Student { ... }
```

Plans and Schedules

3.8

There are other MAML constructs that build up similarly; e.g. schedules and plans are such. Plans are indicated with the `@planDef` keyword, and are composed of events (actions). Schedules (`@schedule`) bind events and plans to time-steps, that is to certain points on the discrete time scale of the simulation. The following example illustrates how plans and schedules are defined in MAML.

Example 1: The definition of a plan and a schedule in MAML.

```
@planDef StartLesson {
  @to teacher enterRoom;
  @forEach students stopTalking;
  @to monitor report;
}

@schedule cyclic(60) Lesson {
  0:  @plan StartLesson;
  44: @to bell ring;
  45: @plan StartBreak;
}
```

3.9

These definitions can be put inside the body of a model, for example, but also agent definitions can contain plans and schedules. Events specify the *receiver* (which can be (1) an agent, like in the first event of `StartLesson`, (2) a collection of agents, indicated with the `@forEach` keyword, or (3) the environment, like in `@to model`) and the *name* of the event (e.g. `enterRoom`). The events can be parameterized, that is, it is possible to pass some arguments with them. The keyword "`cyclic`" signs that the schedule has to be restarted in every 60 time-steps.

3.10

Our next example shows how to define the plan and schedule above in Swarm. Notice, how the low-level object-oriented machinery complicates things. You can also see, however, how much the logic of MAML schedules resembles to that of Swarm.

Example 2: The definition of the same plan and schedule in Swarm. (The first two lines should go into a header file, separately from the rest.)

```
id StartLesson;
id Lesson;

StartLesson = [ActionGroup create: [self getZone]];
[StartLesson createActionTo: teacher message: M(enterRoom)];
[StartLesson createActionForEach: students message: M(stopTalking)];
[StartLesson createActionTo: monitor message: M(report)];

Lesson = [Schedule createBegin: [self getZone]];
[Lesson setRepeatInterval: 60]; Lesson = [Lesson createEnd];
[Lesson at: 0 createAction: StartLesson];
[Lesson at: 44 createActionTo: bell message: M(ring)];
[Lesson at: 45 createAction: StartBreak];
```

Defining the Agents

3.11

Agents, similarly to objects in the OOP paradigm, encapsulate data and operations, or as they are called, state and rules. MAML implements these two components of agents as *variables* (`@var:`) and *subroutines* (`@sub:`). We also kept all the possibilities that an average object-oriented programming language has to offer. MAML support inheritance, just like visibility modifiers (`public`, `protected`, `private`) which are just like in e.g., C++, Objective-C or Java. Agent classes can also have variables and operations, separately from instance ones: the `static` modifier is to be used for this purpose.

```
@sub protected: (void) findNewPosition { ... }
@var static private: int numOfAgents;
```

The default values for the modifiers are chosen according to an important language-design concept of ours: "*if you don't understand it, don't care about it*". Thus, public visibility is assumed if it is not ordered explicitly otherwise. Two more remarks: first, subroutines are not only used for implementing rules, but also for event-handling and for inter-agent communication. Second, we should not forget about the pro-activeness of agents (Wooldridge & Jennings 1995): schedules and plans can be placed not only into models, but also into an `@agent` definition.

Example 3: In this example the agent class `Student` is defined, as a subclass of `Person`.

```
@agent Student : Person {

  @var protected: List marks; // list of marks received so far
  @sub: (void) receiveMark: (int) mark { [marks add: mark]; }
  @sub static: (void) goToSchool { ... }
  /* etc. */

  @schedule cyclic(1440) {
    480: @to self goToSchool;
    840: @to self goHome;
    850: @to mother giveLunch;
    960: @to self doHomework;
  }
}
```

Completing the Definition of the Model

3.12 In addition to agents, plans and schedules, one can also declare variables and subroutines within a `@model` structure: these describe the *environment* of the model. For instance, the parameters of the simulation are implemented as model variables. Also, before letting the schedules generate the events of the simulation, the model should be initialized. Thus the `@model` construct can contain an `@init:` section.

```
@model School {

  // components of the model
  @var: int numOfRooms; // this is a parameter
  ...

@init:
  // initialization code
  numOfRooms = 12;
  ...
}
```

Such an initialization usually contains the creation of the initial agent structure of the model. MAML supports this with the `@create` statement. It can be used to create and initialize agents and collections of agents in a fairly compact way. The following lines create a ten by ten matrix of `Bug` agents and initialize them by invoking their `setX:Y:` subroutine:

```
@create [10:i,10:j] Bug bugs {
  [bugs[i][j] setX: i Y: j];
}
```

Dynamic Behaviour

3.13 MAML provides facilities to add events (or plans) to a schedule or a plan dynamically. This extension of the schedules and plans can be carried out by the environment or by the agents themselves, according to the changes in their state (e.g., in their beliefs, goals, knowledge, etc.) The

```
@addToSchedule Lesson 40: @to Teacher AssignHomeWork
```

statement will add the `AssignHomeWork` action the schedule named `Lesson` to be generated at the 40^{th} time-step. Similarly, the

```
@addToPlan StartLesson @plan CheckHomework
```

statement will hierarchically add the `CheckHomework` plan to `StartLesson`.

Miscellaneous Features

3.14 In addition to the main model building elements described above, MAML provides a couple of "useful variables" for the modeller. For example, for each agent class a list is generated automatically which contains all the agents of that type. This variable is handy when, e.g., an event has to be delivered to all such agents. The list is named after the agent class in point, e.g., `groupOfStudents` belongs to the agent class `Students`.

Example 4: This plan definition is similar to the one in Example 1, but the second message goes for every `Students` agent instead of the programmer-defined group of agents, `students`. The definition of `groupOfStudents` is provided automatically by MAML.

```
@planDef StartLesson {
  @to teacher enterRooom;
  @forEach groupOfStudents stopTalking;
  @to monitor report;
}
```

An example model

3.15 After discussing the main features of MAML, let us demonstrate its capabilities by a relatively complex example that illustrates the use of the MAML constructs introduced so far. This example is a part of the MAML Tutorial which leads the reader through different levels of MAML programming. The final stage of this tutorial is the basic replication of Thomas Schelling's (1971) "Segregation" (*Micromotives*) model.

3.16 For the sake of understandability, we have split the source code into two files, one containing the behaviour of the *residents*, (i.e. agents, available here) and the other describing the properties of the environment (the model, available here). This was possible using the simple importing feature of MAML (`@import`).

3.17 These two files contain a full implementation of the model. (To run them, one has to compile *the model only*.) Even though this implementation runs perfectly, we would not encourage our readers to experiment with this exact version, as it does not contain any means either for setting its parameters, nor to analyse its output. This is due to the fact that these two files only form the *model aspect* of Schelling's work. The concept of the *observation aspect* will be introduced next, and we will also present the missing part of this example.

Observation of models

3.18 In the case of computer simulations the programs under use can be seen as experimental devices built in software. We can identify two different tasks in such a software application. On the one hand, the simulated world is mapped to a computer program. On the other hand, the scientist has to collect information about the model -- this is what the simulation is for. He or she must be able to observe what is going on in the modelled system. This can be done either while the program is running or later during an analysis phase, but even in this latter case, the results must be stored when the simulation is running. Thus the computer program must contain code which is not part of the model, but belongs to the observation.

3.19

Observation tools become a central issue not during the model construction process, but when the experiments with the model are made. As stated above, two main forms of observation are distinguished: posterior and instant. The first one means that information is "logged" during the simulation, e.g. data is saved into files. This technique is often used for discovering the parameter space. The other one usually assumes a graphical user interface that enables the scientist to observe or to interact with the model during the simulation. MAML intends to support both approaches. Moreover, since both techniques might be used during the life-cycle of a model, we stand for easily exchangeable observation components.

3.20

To increase the efficiency of experimenting the observation tools should convey as much information as possible. The software engineering skills and technologies that are needed to develop such observation tools differ from those needed during modelling. Hence, it often happens that the model and its observation are actually developed by different persons.

Disassociation of model and observation

3.21

Like in real world laboratories, it is very useful to make a strict distinction between the model and its observation, between the monitored world and the measuring instrument. The model alone is a "black box", with no ways to control, or just follow the behaviour it produces. If someone would like to know what is happening inside the black box, he must use an observational instrument, which can cross the boundaries of it. However, for software engineering reasons it is highly desirable that the model and the observation tool be as independent as possible.

3.22

The separation of models and their observers appears as soon as during the program design and must be present also at the source code level (Gulyás and Kozsik 1999). We considered this disassociation so important, that the -- not perfect -- solution used in the Swarm community, or a better, but not sufficiently simple OOP solution based on the Factory design pattern did not satisfy us. An elegant and methodologically clean resort was chosen: the introduction of *aspects* into our language (Kinczales et al 1997). One aspect is used to describe the model (how to define a `@model` was shown above). An observation of the model is written in another aspect, which is designated with the `@observe` keyword:

```
@observe School {...}
```

After the keyword the name of the observed model and -- between curly braces -- the body of the observation aspect should be specified. This body can define further (observational) agent classes, schedules, plans, variables and subroutines, and it can extend existing ones, as it will be outlined in the following.

3.23

Note, that several observation aspects can be constructed to the same model without making any changes to the original source code. This reduces the potential of introducing errors into a complete and already tested program. The choice among the different observations are made at compile-time: `xmc` "weaves" the model and the chosen observation aspect together.

Adding observation to a model and to its components

3.24

In the observation the model and its components are extended. This extension concerns their representation, functionality and activity. New variables, subroutines, schedules can be introduced into the model, and also into the agents. An agent class can be extended with the `@extendAgent` keyword. In the example below the `computeAverageOfMarks` function becomes part of the definition of the `Student` agent class. It can access the state of the agent, and it can be called by other agents or the environment, or triggered by an event.

Example 5: The observation of the `School` model introduces a histogram object and extends the `Student` agent class with a function. (For the details of the difference between this approach and the usual OOP inheritance, the interested reader may follow these links: Kinczales et al 1997; Gulyás and Kozsik 1999.)

```
@observe School {

  @var: Histogram h;

  @extendAgent Student {
    @sub: (double) computeAverageOfMarks { ... }
  }

}
```

3.25

It is also possible to extend schedules and plans, using the `@extendSchedule` and `@extendPlan` constructs. Following this logic, observers should have been named as "extendModel", but we chose the more emphatic `@observe` name. Figure 4 illustrates how models and observers are woven together into a single system.

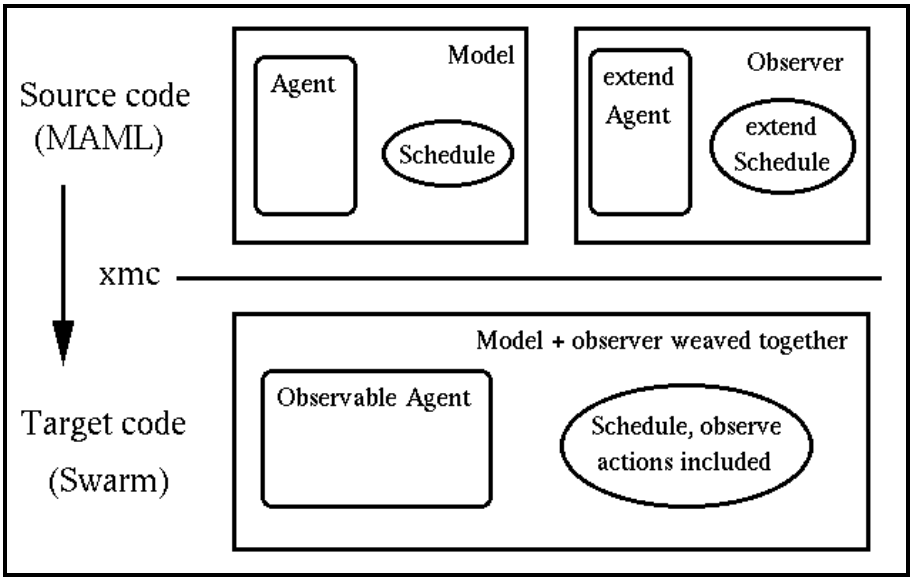

Figure 4: Weaving together a model and an observer

3.26

Initialization of the simulation raises an interesting problem. Creation of the model world (in `@model`) and setting the value of the model parameters (in the `@observer`) is hard to separate. MAML provides a facility though to weave the `@init:` section of the model and that of the observer together. We are currently working on a smoother solution that would give more freedom to model parameterisation.

The observation of the previous example model

3.27

After introducing and explaining the concept of *aspects*, with special emphasis on the observation of multi-agent simulations, let us complete the example presented above. The obstacle preventing the enjoyment of the power contained by this simple model was the missing *observation aspect*. This is what we provide here.

3.28

For the sake of readability, we have emphasized the distinction between the aspects by coding the observer into a separate file (available here). This file imports the source file of the *model aspect*, thus this is the only source to be compiled by the `xmc` compiler to gain the full, extended version of the example. Figure 5 shows a screen capture of the simulation.

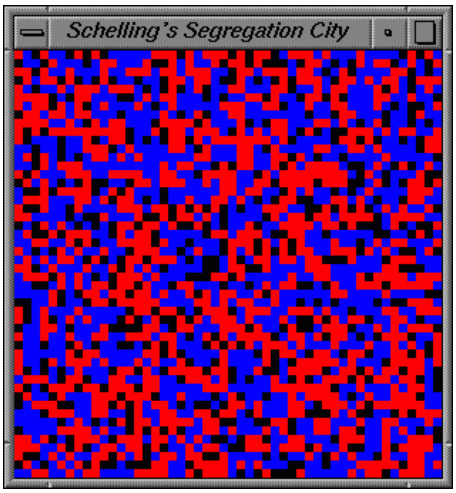

Figure 5: A screen capture of the simulation of Schelling's model

The Model Design Interface

4.1

A visual building tool can be a useful extension to any programming language. This is especially true with MAML, the language that is meant to be used mostly by non-professional programmers. During the design of MAML we always had the idea of MDI behind. For this reason, we accepted the trade-off between fast implementation of the `xmc` compiler and the somehow strange syntax of the language. The graphical user interface makes knowing of the syntax unnecessary, only understanding the structure of MAML programs is needed.

4.2

There are a lot more services a CASE tool can offer to make programming more convenient. To mention but a few, compilation and error localization, parsing and reverse engineering of MAML source code, providing more views for different parts of a model, or speeding up editing. So far only a very restricted set of these features has been implemented. In the following sections we will demonstrate our efforts in developing the Model Design Interface.

Functionality

4.3

To describe a model one has to define a set of components. From the point of view of the MDI not only agents are considered to be components, but also agent class definitions, subroutines, schedules, etc. An agent class definition can be further decomposed, as well. It can contain, for example, descriptions of instance state variables, subroutines and plans. Thus, when building a model with MDI one should create a *container-component hierarchy* of different language elements. At the top of the hierarchy is the "project", composed by a "model" and (zero or more) "observe" extensions. Simple items, such as variables or events are located at the bottom of the hierarchy.

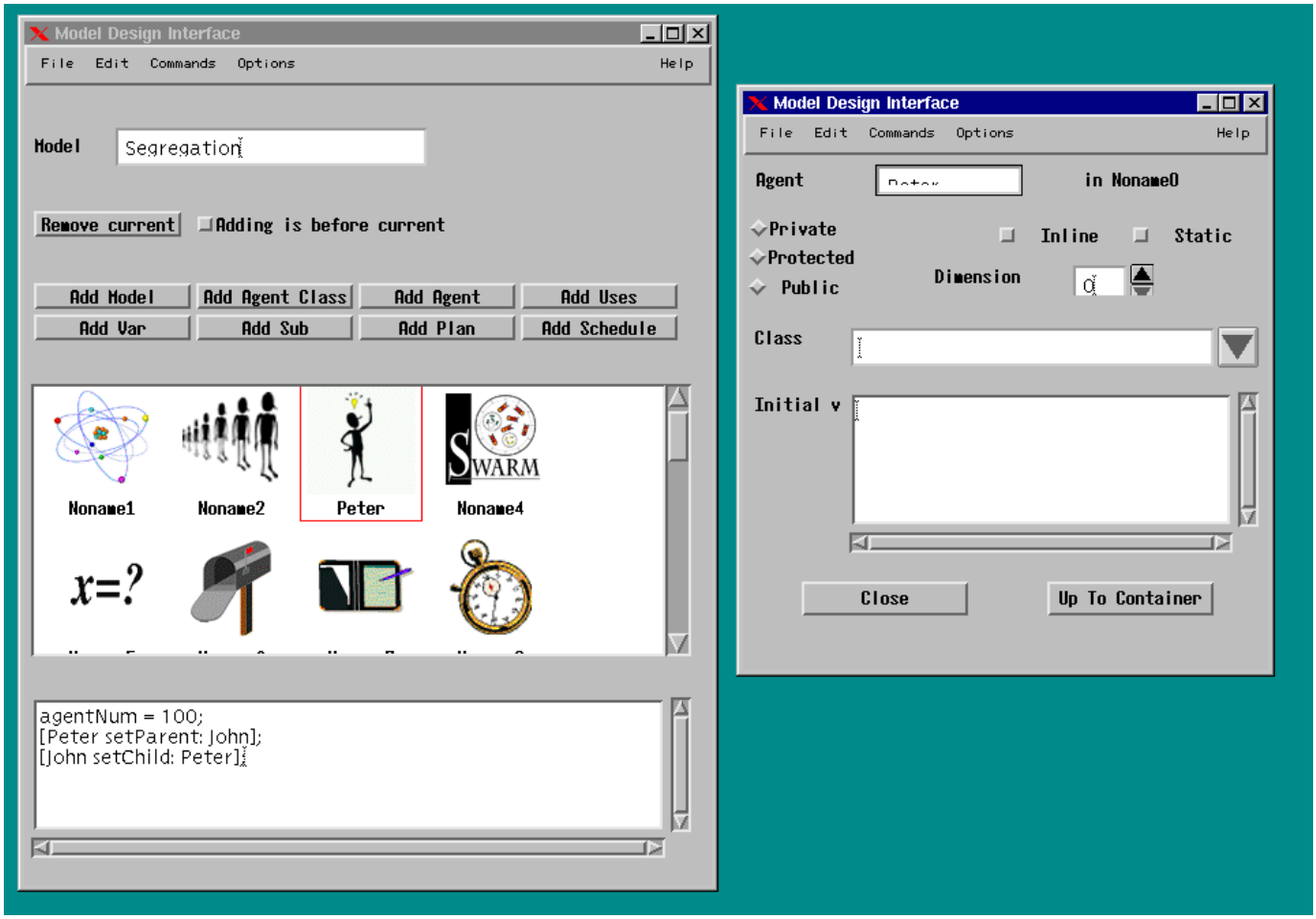

Figure 6: Screen Capture from the Model Design Interface

4.4

The graphical interface follows this structure. Components are represented by *icons*, that containers can hold. The containers correspond to *dialog boxes*, which can have -- apart from a collection of the icons of its components -- text fields, buttons, check-boxes, and so on (see Figure 6). For instance adding the definition of an agent class to a model can be performed by adding a new icon into the dialog box representing the model. This can be followed by filling in the dialog window of this new agent class. Rules for the agents are implemented as subroutines in MAML, more precisely as instance methods defined inside the agent classes. Describing such subroutines, as well as other instruction sequences -- like the "init section" of a model -- requires writing lines of code, basically in Objective-C. Currently this code should go into text fields, but in the future this will probably change: structograms (structure diagrams) or flow charts can be used, not to completely replace, but at least to ease the tedious process of typing.

4.5

As it must be obvious by now, the graphical representation of a model designed by MDI is similar in structure to the underlying MAML program. The container-component hierarchy matches the embedding of MAML language elements into one another. This is, however, subject to eventual change, as the graphical representations will give way to more intuitive concepts and notions of modelling, as our Telemodeling project evolves. The flexible architecture of the program (see paragraph 4.8) allows for the smooth introduction of these new features.

Current State and Short-term Plans

4.6

Only a prototype of the Model Design Interface is completed so far. It is possible to create and save simple models, but observers are not yet supported. Consistency checking is also missing, so it is easy to write a program in MDI that breaks even the most basic rules of the MAML syntax: this should certainly be fixed in the next version. Integration with the xmc compiler and adding the possibility to parse MAML code has also high priority. The implementation of advanced editing functions (copy and paste, drag and drop) will come next.

4.7

After these enhancements the MDI tool becomes usable in practice. This first official release can be achieved fairly easily, but the implementation of many other features are necessary before achieving the full functionality.

Architecture

4.8

In accordance with the logic behind the container-component hierarchy, the implementation of the MDI program meets the *Composite* design pattern. More importantly it also satisfies the *Model-Viewer-Controller* pattern, which enhances its flexibility and makes supplying extensions to it possible. The resulting structure is fairly general, new types of viewers can be easily added to the existing ones, and all of them can freely coexist. This property enables users customize the program to their needs. We have chosen Java to implement the MDI visual builder for two purposes. On the one hand, platform independence is an important issue for us. In the user community different Unix systems (Linux, Irix, etc.) and Win32 are equally used, and we want to support all these platforms. On the other hand, solutions suitable for application through the network are desirable in the Telemodeling framework.

Discussions

5.1

In parallel with the spread of agent-based scientific simulations in the social sciences, the development of new simulators and simulation toolkits is flourishing. On one hand, this leads to a convenient freedom of researchers in selecting the platform to use. On the other hand, however, the toolkits not only present opportunities but also, a set of specific, implicit constraints imposed on the modeller (as expressed in Gilbert 1997). Hence, the software environment to use should be carefully chosen. Moreover, published results must be readable by the wider community of scientists, so the emergence of at least *de facto standards* is vital. To fulfil this role a toolkit must be easily accessible (meaning e.g. free of charge licensing policy) for at least scientific activities and education. Also, it should be applicable to as wide an area of interest as possible, and it should provide an easy-to-use, intuitive (possible graphical) modelling environment.

5.2

>From the current state of the art one may conclude that the last two requirements are controversial. Some toolkits are relatively easy-to-use, but more or less focussed on a particular subset of models (e.g. SDML, Echo], or Ascape). On the other hand, packages like Swarm are general but hard to use. The general accessibility is also a problem: MOOSE is not freely available, SDML is based on a proprietary package, so is SIM_AGENT, while the future of ascape in the public domain is uncertain. Finally, efforts like the MASSIF project (based on the former MIMOSE system) are prevented from wider use due to their early state of development.

5.3

Based on the growing community of Swarm users the MAML language and the Model Design Interface has the possibility to meet all requirements outlined above, transforming Swarm into a widely and easily usable tool. This transformation will be enforced by the introduction of new high-level language elements and more and more intuitive visual notions (which is supported by the flexible architecture of MDI). Furthermore, another pathway to follow is to provide specialized notion-sets for particular scientific areas, thus unifying the benefits of the underlying general toolkit, and that of a focussed modelling environment. Moreover, since the original design of MAML allows for its decoupling from the Swarm package, our Telemodeling project has the potential to eventually bridge the gap between the major simulation systems.

Conclusions

5.4

In this article we have reported on our Telemodeling project aimed at the development of an integrated, easy-to-use modelling environment for agent-based social simulations. We have presented our vision of what a modelling environment of this kind should be like and we have also expressed our concerns why the currently available toolkits are failing to demonstrate the expected attributes. Besides these expectations this paper introduced the first major step resulting from the Telemodeling project, the first version of the Multi-Agent Modelling Language (MAML). We have also discussed the next step taken, the prototype of the Model Design Interface, a graphical model building tool developed on top of the MAML language.

Acknowledgement

Most of this work was carried out at the Complex Adaptive Systems Laboratory, Central European University.

Notes

[1] This work is an extended summary of the papers presented in Gulyás, Kozsik and Fazekas (1999) and Gulyás and Kozsik (1999).

[2] This definition of the language was announced at SwarmFest'99, The Third Annual Meeting of the Swarm Users Group held at UCLA, Los Angeles, March 1999.

[3] In contrast to objects, agents are also active, in the sense, that they can have their own thread of control. Certain object-oriented programming languages, for example (C++//) are appropriate to implement this.

References

KINCZALES, G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J-M., Irwin J.: *Aspect-Oriented Programming*. Xerox Palo Alto Research Center, http://www.parc.xerox.com/spl/groups/eca/pubs/complete.html#Kiczales-ECOOP97

GULYÁS L., Kozsik T.: *Aspect-Oriented Programming in Scientific Simulations*, Proceedings of The Sixth Fenno-Ugric Symposium on Software Technology, Estonia 1999, http://www.syslab.ceu.hu/maml/papers/aopss.ps

PARKER, M. T.: *Parallel Evolution, or Swarm and Swarm-a-like: the 'ascape' Agent Based Modelling Framework.* SwarmFest 1999, March 27-29 1999, UCLA http://www.brook.edu/es/dynamics/models/ascape/

CAROMEL, D., Belloncle F., Roudier Y.: *The C++// system*, in Parallel Programming Using C++, G. Wilson and P. Lu (eds.), MIT Press, 1996. http://www-sop.inria.fr/sloop/c++ll/c++ll.ps

JONES, T., Forrest, S.: *An introduction to SFI Echo* Technical Report 93-12-074, Santa Fe Institute, Santa Fe, NM. http://www.santafe.edu/~pth/echo/

GILBERT, N.: *Models, Processes, and Algorithms: Towards a Simulation Toolkit.* Paper given at the Dagstuhl Seminar on Social Science Microsimulation, May 5-9, 1997. http://www.uni-koblenz.de/~kgt/Dag9719/Gilbert.html

GULYÁS L.: *Using Object-Oriented Techniques to Develop Multi-Agent Simulations.* Proceedings of the 1st International Conference of PhD Students, University of Miskolc, Hungary, 1997, pp. 63-69.

GULYÁS L., Kozsik T., Fazekas S.: *The Multi-Agent Modelling Language*, http://www.syslab.ceu.hu/maml/

GULYÁS L., Kozsik T., Fazekas S.: *The Multi-Agent Modelling Language*, Proceedings of the 4th International Conference on Applied Informatics, Eger-Noszvaj, Hungary, 1999.

GULYÁS L., Kozsik T.: *The MAML Tutorial*. Central European University, Complex Adaptive Systems Laboratory, http://www.syslab.ceu.hu/maml/tutorial/

MENTGES, E.: *Concepts for an agent-based framework for interdisciplinary social science simulation*. Journal of Artificial Societies and Social Simulation vol. 2, no. 2. http://jasss.soc.surrey.ac.uk/2/2/4.html

GULYÁS L., Kozsik T.: *Model Design Interface - A CASE Tool for the Multi-Agent Modelling Language*, Proceedings of the 2nd International Conference of PhD Students, Miskolc, Hungary, 1999. http://www.syslab.ceu.hu/maml/papers/mdi.ps

GULYÁS L, Kozsik T, Czabala P, Corliss, J.B.: *Telemodeling ñ Overview of a System*, Teleteaching '98, Distance Learning, Training and Education, Proceedings of the XV. IFIP World Computer Congress '98, 31 Aug - 4 Sep 1998, Vienna/Austria and Budapest/Hungary, Austrian Computer Society (OCG), 1998. p. 59. http://www.syslab.ceu.hu/telemodeling/

MÖHRING, M., Troitzsch, K. G.: *MIMOSE (Micro- und Multilevel Modelling Software)*. http://www.uni-koblenz.de/~moeh/projekte/mimose.html

MINAR, N., Burkhart, R., Langton, C., Askenazi, M.: *The Swarm Simulation System: A Toolkit for Building Multi-agent Simulations*, http://www.santafe.edu/projects/ swarm/overview/overview.html

CUBERT, R. M., Fishwick, P. A.: *MOOSE: An Object-Oriented Multimodeling and Simulation Application Framework*. Submitted to Simulation, June 1997. http://www.cise.ufl.edu/~fishwick/moose.html

APPLE: *Object-Oriented Programming and the Objective-C Language*, http://developer.apple.com/techpubs/macosxserver/ObjectiveC/

GAMMA, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

SCHELLING, T.: 1971. *Dynamic Models of Segregation.*. Journal of Mathematical Sociology, 1971, 1:143-186.

MOSS, S., Gaylard, H., Wallis, S., Edmonds, B.: *SDML: A Multi-Agent Language for Organizational Modelling*. Computational and Mathematical Organization Theory (forthcoming). http://www.cpm.mmu.ac.uk:80/cpmrep16.html

SLOMAN, A., Poli, R.: *SIM_AGENT: A toolkit for exploring agent designs*. in Intelligent Agents Vol II (ATAL-95), Eds. Mike Wooldridge, Joerg Mueller, Milind Tambe, Springer-Verlag, pp. 392--407

SUMPTER, D. J. T.: *Introduction to Multi Agent Simulation.* http://www.ma.umist.ac.uk/dsumpter/beesim/Simulation/overview.htm

SANTA FE INSTITUTE: *The Swarm Simulation Package*, http://www.santafe.edu/projects/swarm/

SONNENSCHEIN, M., Köster, F., Lorek, H., Vogel, U.: *The WESP Project*. http://www.offis.uni-oldenburg.de/projekte/ecotools/project_ecotools5.htm

WOOLDRIDGE, M. and Jennings,N. R.: *Agent Theories, Architectures and Languages: A Survey*, Intelligent Agents ECAI-94 Worshop Proceedings, Lecture Notes in Artificial Intelligence 890, pp. 1-39, Springer-Verlag, Berlin, 1995.

Return to Contents of this issue