The Multi-Agent Modeling Language

László Gulyás, Tamás Kozsik, Sándor Fazekas

Complex Adaptive Systems Laboratory,
Central European University, Budapest, Hungary
e-mail: {gulya,hto,fakir}@syslab.ceu.hu

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 Modeling Language (MAML) is one of the efforts to ease these difficulties.

1 Introduction

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 [10].

Our Telemodeling project [7] aims to ease the afore-mentioned difficulties and to strengthen the reliability of the developed simulation by the creation of a framework that supports all phases of multi-agent simulation, from the design of models, through parameter space search, to result-analysis. The kernel of the framework is a special purpose programming language, the Multi-Agent Modeling Language (MAML). It provides high-level language constructs to describe agent-based computer models, making the description of complex models easier while ensuring the simplicity and understandability of the computer program.

MAML builds strongly on a freely distributed toolset, Swarm [11], 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. In its current form MAML is a macro-language for Swarm which provides easy access to basically all functionality available in the original package.

MAML – similarly to Swarm – adopts the agent-based modeling paradigm [12], within a discrete event simulation framework. That is, a model is made up of a collection of
independent agents interacting via messages and events [10]. An event is associated with
a single time-step 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.

Every agent maintains its 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.

Although there are a lot of different approaches to agents [13], even within the agent-
based modeling 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 advantages of MAML can be exploited in two ways: using it as a traditional
programming language or as an underlying layer for a graphical model building tool (the
Model Design Interface, see [5]). The first approach is discussed in this paper. Section
2 provides an introduction to the language with a number of examples demonstrating
the strength of MAML. This is followed by Section 3, which outlines future and related
efforts, and concludes the paper.

2 The Multi-Agent Modeling Language

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 version applicable
in practice with a fairly stable compiler has recently been completed. Throughout this
paper we will focus on this (v0.03) release.\footnote{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.}

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 [1], 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 the whole functionality offered by the
Swarm libraries (including random number and distribution generation, graphical and
statistical tools, etc.) is available for MAML programmers.

MAML (similarly to Swarm and many other simulation packages), 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 [6]. 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.\footnote{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++ // [2] are appropriate to
implement this.}

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 termi-

nology), etc. are all formalized both in Swarm and MAML. However, there is a signif-

icant 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 modeling (see Section 1). Each of those concepts is mapped to a language construct of MAML, as it will be shown later in this section.

What are the advantages of this approach? A modeller – especially a beginner in programming – prefers thinking “in the problem domain”, using the concepts of modeling, 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 Examples 2.1 and 2.2), 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. In the following an overview of MAML programming is presented. For more details (reference manual, tutorial, etc.) refer to [8].

2.1 The Model Construct

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.

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.

@model School { ... @agent Student { ... } ... }

2.1.1 Plans and Schedules

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 2.1 The definition of a plan and a schedule in MAML.

```plaintext
@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;
}
```

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.

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

```plaintext
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];
```

2.1.2 Defining the agents

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 supports visibility modifiers (public, protected, private) which are just like in e.g., C++, Objective-C or Java, and (not multiple, only single) inheritance. Agent classes can also have variables and operations, separately from instance ones: the static modifier is to be used for this purpose.

```plaintext
@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 [13]: schedules and plans can be placed not only into models, but also into an @agent definition.

Example 2.3 In this example the agent class Student is defined, as a subclass of Person.

```java
@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) {
        600: @to self goToSchool;
        840: @to self goHome;
        850: @to mother giveLunch;
        960: @to self doHomework;
    }
}
```

2.1.3 Completing the Definition of the Model

In addition to agents, plans and schedules, one can also declare variables and subroutines within the @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. Hence, the @model construct contains an @init: section.

```java
@model School {
    @var: int numOfRooms; // this is a parameter
    /* further components of the model */
    @init:
        numOfRooms = 12;
        /* further initialization code */
}
```

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 line creates a ten by ten matrix of Bug agents and initialize them by invoking their setX:Y: subroutine:

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

2.1.4 Dynamic Behaviour

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 40th time-step.

Similarly, the “@addToPlan StartLesson @plan CheckHomework” statement will hierarchically add the CheckHomework plan to StartLesson.

2.1.5 Miscellaneous Features

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. We can, for example, replace students with it in Example 2.1, which would then mean, that we want the stopTalking message to be sent to every agent of type Student.

2.2 Observation of models

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 scientists have to collect information about the model – this is what the simulation is for. They must be able to observe what is going on in the modelled system. This can be done either while the program is running (using a - usually graphical – user interface) or later during an analysis phase, but even in this latter case, the results must be stored (e.g. saved to files) when the simulation is running. Thus the computer program must contain code which is not part of the model, but belongs to its observation.

MAML intends to support both observation approaches. Moreover, since the application of both techniques might be required during the life-cycle of a model, we stand for easily exchangeable observation components. For software engineering reasons it is highly desirable that the model and the observation tool be as independent as possible. This assertion is explained in details in [4] where the necessity of this disassociation is justified with several arguments and also an elegant and methodologically clean resort is presented: the introduction of aspects [9] into our language. One aspect is used to describe the model, using the @model construct. An observation of the model is written in another aspect, which is designated with the @observe keyword (see Example 2.4).

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.

2.2.1 Adding observation to a model and to its components

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 `averageOfMarks` 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 2.4** The observation of the School model introduces a histogram object and extends the Student agent class with an operation.

```java
@observe School {
    @var: Histogram h;
    @extendAgent Student { @sub: (double) averageOfMarks {...} }
}
```

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.

Initialization of the simulation raises an interesting problem. Creation of the model world (in the `@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 parametrisation.

## 3 Discussions

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 [3]). 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 fulfill 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) modeling environment.

From the current state of the art, however, one may conclude that these requirements are controversial. Some toolkits are relatively easy-to-use, but more or less focussed on a particular subset of models, while on the other hand, packages like Swarm are general but hard to use. (See [5] for references.) Therefore, there is a great potential for an integrated, easy-to-use modeling environment.

Based on the growing community of Swarm users the MAML language (and the Telemodeling project) 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 the completion of the Model Design Interface [5]. 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 modeling environment. Moreover, since the 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.
References

http://developer.apple.com/techpubs/macosxserver/ObjectiveC/

http://www-soop.inria.fr/sloop/c++ii/c++ii.ps

http://www.uni-koblenz.de/~kgt/Dag9719/Gilbert.html


http://www.syslab.cem.hu/telemodeling/

http://www.syslab.cem.hu/mami/

http://www.xes.xerox.com/spl/groups/eca/pubs/complete.html#Kiczales-ECOOP97

http://www.santafe.edu/projects/swarm/overview/overview.html

http://www.santafe.edu/projects/swarm/

http://www.ma.man.ac.uk/dsumpter/beesim/Simulation/overview.htm


Postal addresses

Gulyás L., Kozsk T., Fazekas S.
Complex Adaptive Systems Laboratory
Environmental Sciences and Policy
Central European University
1106 Budapest, Kepesút út 87.
Hungary