

The Use of Aspect-Oriented Programming in Scientific Simulations

László Gulyás^{1,2} and Tamás Kozsik^{1,2}

¹ Complex Adaptive Systems Laboratory,
Central European University, Budapest, Hungary

² Department of General Computer Science
University Eötvös Loránd, Budapest, Hungary
gulya@syslab.ceu.hu, <http://www.syslab.ceu.hu/~gulya>
tamas.kozsik@elte.hu, <http://www.elte.hu/~kto>

Abstract. The Aspect-Oriented Programming paradigm is a newly introduced concept for program development. It can be seen as a technique that can co-exist with other methodologies, such as object-oriented, procedural, functional or event-driven programming. The purpose of AOP is to provide the programmer a toolset with which he or she can describe certain properties or parts of the software that cross-cut its functional structure. There are a few examples where AOP is applied in software development. Among others, aspects can be used to specify synchronization properties of distributed systems or handling optimization issues. In this paper another application domain is presented: our field of interest is the design and implementation of computer simulations.

In some sciences, especially in the study of the so-called complex systems, computer programs play an important role as scientific equipments. In the case of computer simulations, the programs under use can be seen as experimental devices built in software. More precisely, they are both playing the role of the system under study and the experimental tools used to observe them. We argue that considering the two afore-mentioned roles as different aspects of the same system makes both the implementation and later the understanding of the model considerably easier. The aspect-oriented approach can make the development of computer simulations even more straightforward if it is supported by the modeling language used, as in the case with our Multi-Agent Modeling Language (or MAML for short), which is also briefly discussed by the paper.

1 Introduction

Aspect-Oriented Programming: The Aspect-Oriented Programming [9, 15] paradigm is a newly introduced concept for program development. It is not supposed to replace the wide-spread Object-Oriented Programming methodology: it can be seen as a complementary technique that can co-exist with other approaches, such as object-oriented, procedural, functional or event-driven programming. The purpose of AOP is to provide the programmer a toolset with which he or she can describe certain properties or parts of the software that

cross-cut its functional structure. Aspects — like components — are building blocks of a program. They capture a part or property that affects several components, but should be handled at one place. An aspect, in spite of it doesn't fit into the layout specified by the problem decomposition, had better be designed and implemented separately. The resulting code is not linear any more; it is the task of the compiler (or aspect-weaver, as it is called in AOP terms) to put the code distributed in the aspects together, by inserting all pieces found in the aspects into the appropriate components.

Aspects are written either in the same language, as the components of the program, or in a language that was designed for a particular problem area. In the second case the use of a domain specific language makes filling the gap between design and implementation much easier. Programs written in AOP-style are usually much shorter and less complex than traditional ones that cannot exploit the benefits provided by aspects.

Examples of AOP exist in a plenty of applications. Among others, aspects can be used to specify synchronization properties of distributed or concurrent systems [10, 3] or handling optimization issues [8, 11]. In this paper another application domain is presented: our field of interest is the design and implementation of computer simulations.

Computer Simulations: In some sciences, especially in the study of the so-called 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 [12].

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 [6, 4]. In this project — among other activities — a special purpose programming language and a CASE tool are being developed, that will help scientists constructing their models. The programming language, called Multi-Agent Modeling Language [7], or MAML for short, builds strongly on an existing, freely distributed toolset, Swarm [14]. Swarm 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.

The modeling formalism that Swarm — thus also MAML — adopts is Agent-Based Modeling: a model is made up of a collection of independent agents interacting via discrete events. Within that framework, Swarm makes no assumptions

about the particular sort of model being implemented, but does build strongly on object-oriented concepts [12].

Contents: This paper presents how the aspect-oriented programming paradigm can be utilized in building computer simulations. First we give a short introduction to our Multi-Agent Modeling Language in section 2. A program design question is raised in section 3 which will be answered in section 4. This latter section is the kernel of our paper: 4.1 outlines the solution with aspect-oriented programming, while 4.2 focuses on how MAML supports AOP. A simple example (section 4.3) demonstrates the preceding ideas, then 4.4 sketches other ways of solution. Finally, the discussion in section 5 closes the paper.

2 The Multi-Agent Modeling Language

The Multi-Agent Modeling Language, or MAML for short, is a domain specific programming language that was designed to help scientists, who hold their expertise in other fields than computer science, develop computer simulations. It is based on the Swarm simulation package, which is a collection of libraries for the object-oriented programming language Objective-C. While it borrows a lot of notions from Swarm, it provides a simpler, easier way to create agent-based models by eliminating the need for a lot of implementation details. Its logically clean constructions can facilitate programming. This paper will reveal one of the ideas MAML is organized around, viz. how to separate a model from monitoring tools.

MAML is a set of structures, each corresponding to a special concept of agent-based modeling. The language is still under development, this means that this set of structures is still growing. The MAML compiler — which is called `xmc` — is trying to keep up with the evolving of the language; a fairly stable version was recently released. Throughout this paper we will concentrate on this (v0.03) release.¹ The reason why the design of the language is distributed over many different steps is that this way the language can constantly be under evaluation of the user community, and it can be formed to meet as much of the requirements of its future users as possible.

As currently MAML is implemented over Swarm, it can be seen as a sophisticated macro-language: the constructions of MAML are turned into coding schemes in Swarm.

2.1 Overview of MAML

Agents in the model are described like object classes in OOP programs. The definition of an *agent class* encapsulates the representation of states (as instance

¹ 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.

variables), the implementation of rules (as subroutines) and activities to perform (schedules). Agents can be created as instances of an agent class.

The control flow of the simulation is driven by discrete events. These events can be generated by *schedules*. Schedules can also be manipulated during runtime. *Plans* are sets of events to be generated. They correspond to compound activities to be performed, e.g. by agents. The events can be organized to be generated in sequence, in random order or in parallel.

The language provides other high-level tools for creating and initializing agents, objects and arrays, supports the “probing” of variables [14], and defines some useful shorthands for frequently used terms. These items are not addressed here: see [7] for more details.

2.2 Global view: Telemodeling

MAML is, in fact, part of a larger project aiming at supporting all phases of simulation in 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.

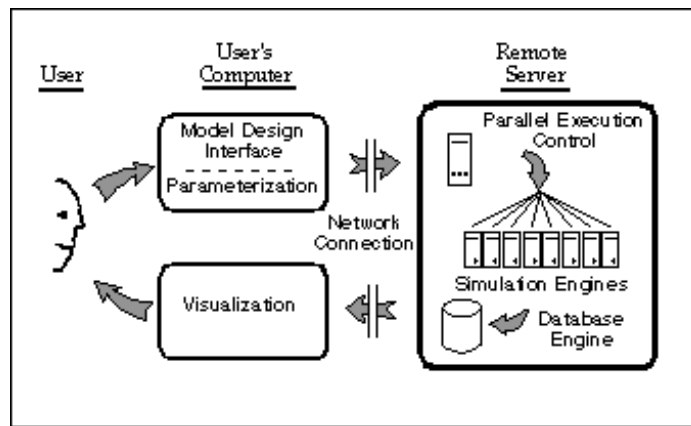


Fig. 1. The Telemodeling project

These stages are connected by a 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.

Although the Telemodeling project is still in its infancy prototypes or case studies are available for all the important stages of simulation [6]. MAML is

the first released product of this effort, forming the base for the model design toolset.

3 Disassociation of Model and Observation

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. Instead of making experiments of the real world, this computer program is the model, the subject of the investigation. On the other hand, the scientist has to collect information about the model – this is what the simulation is for. He or she has to be able to observe what is going on during the simulation, namely while the program is running. Thus the computer program must contain code which is not part of the model, but belongs to the observation.

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 should be a “black box”, with no input and output crossing its boundaries. It must be possible to create and execute it independently upon any observational mechanism. This separation appears as soon as during the program design and must be present also at the source code level. Let us line up some arguments to verify this thesis.

- Building a model means creating an abstraction of the real world. Realizing this abstraction in a computer language is always a hard job. It can be made easier, if — thanks to a good modeling toolset — the scientist doesn’t have to worry too much about implementation and data representation details, which is definitely demanded when one has to write a (possibly graphical) observational tool. This latter task, that requires mostly only programming, can be performed separately, at a later time. Furthermore, it is sometimes the case, that the model design and realization is carried out by the scientist, who has the necessary knowledge about the modelled domain, but the observational toolset is added to the system by a programmer who is better at exploiting the advantages of the used graphical libraries.
- 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, so maintenance issues — which are always important in software engineering — really come to the front. Even a little modification of the abstraction is very hard if the code fragments that describe the model mixed up with the implementation of observational tools should be identified first. Complexity of the program can be decreased and its understandability can be increased by eliminating such tangling code.
- There are cases when the scientist wants to change the means of observation, replace one or more monitoring tools with others. For example, after fine-tuning the model with a graphical or interactive observational technique the

scientist might want to do a parameter space search, so he or she switches to batch-mode observation. In such cases accidental alternation of the model is far undesirable. Explicit separation of the model code and the observation code is of great help in avoiding such a mishap.

- Computer based modeling is not a science until published results are not reproducible. Publishing only the abstraction, though it is the general case, is usually not enough for this purpose: the program that produced the result would also be needed. (The mapping of an abstraction of the real world to a computer program is not unambiguous. And different programs will produce different results, since even the seed of the used pseudo-random number generator has a gross impact on the outcome of the simulation.) Thus the program text is not only read by its author, but also by every other scientist, who wants to understand, employ, modify, reimplement or reuse the model. This is why the program complexity and maintainability issues should again be emphasized.

The developers of Swarm recognized this problem, and as a response they worked up a programming style — which became wide-spread in the community, thanks to the inertia and the “copy-and-paste” way of programming — that tries to separate the observation from the model. In spite of the efforts taken, certain parts of the code usually still contain the two things mixed. This originates from a weakness of the applied object-oriented programming technique, namely, that OOP is not very well suited to express properties that are orthogonal to the functional structure of a system. Using the aspect-oriented programming paradigm, as we do in our Multi-Agent Modeling Language, a better and more elegant solution can be given.

4 Solution

This section will present how to solve the problem outlined above. First we explain, why the resort proposed by the Swarm developers wasn't satisfactory for us. According to the concept of agent-based modeling, a model is defined as a collection of agents, interacting via discrete events. Since Swarm follows the object-oriented programming approach (it is a package of libraries for an object-oriented language, Objective-C), it is not surprising, that the building blocks of a Swarm program are the objects of some library- or user-defined classes. The model is an object of class `Swarm`, which can aggregate [13] agents and can possess event generators. The event generators (or as they are called, schedules) and the agents are again implemented as objects.

To add monitoring tools to the system, the programmer can define one or more `ObserverSwarm` classes, one for each monitoring toolset. (In general a project includes one with a graphical user interface, and another one for batch-mode execution.) An `ObserverSwarm` has the model as an aggregate, and defines the desired data analyzers, graphs, histograms, etc. for collecting, processing, storing and visualizing data.

The problem is the following. In order to make these observer tools capable for interacting with the model and its components, some additional pieces of code must be inserted into the `ModelSwarm` and the agent classes. These pieces of code are indifferent from the point of view of the model: they are part of the implementation of the observation. This problem is not trivially solvable in the OOP framework. The extra functionality mentioned above cannot be simply added to the components by the means of inheritance, because for this every agent creation in the model, and thus the whole system should be modified.² In the object-oriented programming paradigm both encapsulation and functional extension is performed at the level of objects. In the contrary, what we need here is the encapsulation of a system-wide functionality, and the functional extension of a whole system.

4.1 The Aspect-Oriented Approach

The aspect-oriented programming technology enables expressing properties or parts of a system that cross-cut its functional layout without increasing complexity or introducing tangling code. These cross-cutting issues can be formulated as aspects and they can be added to the system without affecting its implementation.

The observation of a model can be seen as such a cross-cutting issue. The code implementing it should be distributed in the whole system, additional attributes (variables and operations) should be added to several object class definitions. Instead of disturbing the process of designing and implementing the model, the observational tools are described separately in an aspect. This way the model remains fairly independent of its observations.

Notice, that apart from filling the values of the model parameters, the model doesn't depend on the observations; it will behave the same way with or without them. Hence an observer aspect is added to the model essentially with *superposition*. This is a quite simple application of the AOP paradigm.

Let us outline the process how computer simulations are prepared according to the technique discussed above. First the model is designed and implemented. This is followed by the development of different observation strategies. Each of these strategies is specified in an observational aspect. Different aspects can be created for a graphical user interface or for a monitoring toolset that only writes data to files. Before running the simulation an observational aspect can be associated with the model. An intelligent compiler, an aspect-weaver produces the code which contains the model and the chosen observer interwoven.

To be able to use the technique discussed above, the modeling framework used should support aspect-oriented programming. One way to ensure this is to introduce special language constructions into the modeling language for this purpose. In the following section we present the possibilities MAML has to offer.

² In section 4.4 we will show how to do this even so, using a trick, the Factory design pattern.

4.2 AOP support in the Multi-Agent Modeling Language

Currently the `xmc` compiler produces Swarm code from the MAML source files. The Swarm code is a program written in the Objective-C language, that uses the Swarm libraries. Hence the main task of `xmc` is to create a set of object class definitions from the model specification and an observation aspect. The observation is bound to the model during compilation and there can be only one observation aspect bound at a time. However, in our case this doesn't impose serious restrictions: the source code of the model, and in many cases only the model is to be made public, not an executable program.

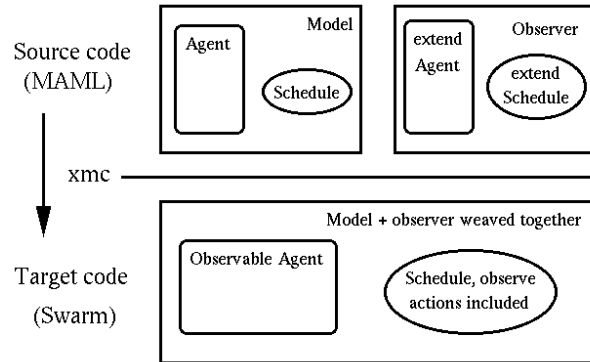


Fig. 2. Weaving in MAML

In MAML a model is described in a `@model <name> { ... }` section. It can contain agent class definitions (`@agent`), event generators (`@schedule`), parameters, components (`@var`), operations (`@sub`), etc. It also has an `@init` part which usually defines how the components must be created and initialized. Agents can also have attributes (variables and subroutines) that define their states and the accepted messages.

An observation aspect is declared in an `@observe` section. Additional components, operations, event generators can be put here. It is also possible to insert new attributes into an agent class definition (`@extendAgent`) or new events into an event generator (`@extendSchedule`). The relevant keywords are summarized in table 1.

The observer can also have an `@init` section, where the creation and initialization of the monitoring tools take place. Moreover, the values of the model parameters — which are read from files or acquired through a graphical user interface — are also set here, so the execution of the `@init` in the observer precedes that of the `@init` in the model. It is often the case that after initializing the model there is something more to do in the observer. In such cases the `@initModel` keyword can be used in the observer, which triggers the invo-

Table 1. Keywords for supporting AOP

@observe	@model	Model and observer
@extendAgent	@agent	Agent class
@extendSchedule	@schedule	Event generator
@extendPlan	@planDef	Ordered collection of events
@addToSchedule	@schedule	Dynamic extension of schedules
@addToPlan	@planDef	Dynamic extension of plans
@init, @initModel	@init	Initialization

cation of the model @init, but after its execution the control is returned to the observer.

This toolset proved to be sufficient for most of the problems so far. However, we have some ideas what constructions could make it even more powerful. We will come back to this issue later, in section 5.

4.3 Example

As an illustration of the possibilities discussed above, we show an extract of the MAML implementation of Heatbugs, a model that is frequently used for demonstration by “Swarmites”.

```

@model Heatbugs {
  @agent Heatbug {
    @var protected: int x, y;
    ...
    @sub: (void) Step { ... }
  }
  ...
@init:
  ...
}

@observe Heatbugs {
  @extendAgent Heatbug {
    @var static protected: id bugPixmap;
    @sub static: (void) initBugPixmap: (id) r { ... }
    ...
  }
  @var: id worldRaster;
  ...
@init:
  ...
  @initModel;
  @create ZoomRaster worldRaster {
    SET_WINDOW_GEOMETRY_RECORD_NAME (worldRaster);
  }
  ...
}

```

4.4 Other ways of solution

Before initiating the development of the Multi-Agent Modeling Language, which solved the disassociation problem of model and observation in a simple and straightforward way, we were seeking for a solution within the object-oriented programming paradigm. The standard OOP tool for code partitioning and functional extension, the inheritance provided an opportunity for that. As we mentioned before, the problem arisen is that inside the model code all the agent creation statements must be changed, like in the following example, where the second line was introduced to replace the first one:

```
// myAgent = [AnAgentClass create: aZone];
myAgent = [ObservedAgentClass create: aZone];
```

In this case `ObservedAgentClass` is a subtype of `AnAgentClass` containing all the extra properties related to observation.

Programming in accordance with the *Factory design pattern* [2] can eliminate this problem. Instead of using the appropriate class method to create an agent, the programmer can ask for one from a “factory object”. This object is responsible for choosing the exact subtype of the agent class desired:

```
myAgent = [factory giveMeNew_AnAgentClass: aZone];
```

The `giveMeNew_AnAgentClass` message will return a newly created object of class `AnAgentClass`, but the dynamic type of this object can be a subclass, in our example `ObservedAgentClass`. When adding or changing the observation, only the factory object must be replaced.

This solution, though quite natural to a computer scientist, is not so intuitive to someone who is not an expert in programming. Aspects are more easily explained to beginners, that’s why we prefer the AOP answer.

Alternative implementations of AOP: When we first committed to aspects we didn’t have our own weaver, the `xmc` compiler in mind.³ First the application of the `patch` command in Unix was brought up. This was destined to help the distinct publication of a model and its observations. The design phase would have not been supported by this technique. The scientist first develops his or her model in program A. Then he or she completes it with the observation, gaining program B. Determination of the differences of B and A comes next. Now the program of the model, A and this difference can be published separately, yet anyone willing to obtain B can manufacture it with the `patch` command. Several observations of the same model can be published without having to repeat its definition again and again.

Interestingly, Objective-C itself also has features to enable aspect-oriented programming. *Categories* provide an opportunity to extend an existing class definition. An aspect can be added to a system by declaring categories to its

³ In fact that time we didn’t even know about the AOP paradigm.

components. Unfortunately categories are bound to classes during linking, not in compile-time, hence they can only contain messages and no state variables. This restriction make them essentially inappropriate to meeting our requirements.

5 Discussion

Future plans: Surveying the `@extend...` structures of MAML the lack of the keyword `extendSub` must be conspicuous. Next versions of the language will enable the observer aspects to add some statements before and after the operations defined in the model, like in [16]. A keyword similar to `@initModel` can be introduced, that will be used to invoke the original subroutine from an extension.

Starting the initialization of the model in the observer `@init` is also about to change. We would like to give the opportunity to set up the values of the parameters in more than one steps. Interweaving code from the `@init` of the model and that of the observer would be necessary for this. The exact semantics — though we have developed some alternatives — is not chosen yet.

Finally an interesting possible application of aspects should be mentioned. In a very complex model it might be useful to describe certain “situations”, relationships of components separately. A situation might involve some of the agents, each agent playing a specific *role*. Hence an agent can be built up from its different roles, that were defined in different situation aspects. We are planning to improve MAML in this direction as well.

Conclusion: Dissassociation of a model and its observations is useful in the design and implementation of computer simulations. We investigated the technique that a widely used simulation package, Swarm adopts, and found it unsatisfactory. Although a solution with the Factory design pattern was found within the Object-Oriented Programming paradigm, we realized, that Aspect-Oriented Programming provides a simpler and more intuitive answer. Adding observation as an aspect to the model is a straightforward way to separate the development of models and observation toolsets.

The Multi-Agent Programming Language was designed to help scientists, who hold their expertise in other fields than computer science, develop computer models. According to the above reasons, MAML possesses some language elements that support AOP. The `xmc` compiler, which produces Objective-C code from the MAML source, plays the role of the aspect-weaver. In the future further AOP features will be introduced in MAML.

References

1. Apple: *Object-Oriented Programming and the Objective-C Language*.
<http://developer.apple.com/techpubs/macosxserver/ObjectiveC/>
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns*. Addison-Wesley, 1994.
3. Frohner Á., Kozsik T., Varga L.: *UML Design and Implementation of Synchronization Patterns*. Abstract will appear in: Proceedings of 3rd Joint Conference on Mathematics and Computer Science, Visegrád, Hungary, 1999.
4. Gulyás L.: *Using Object-Oriented Techniques to Develop Multi-Agent Simulations*. Proceedings of 1st International Conference of PhD Students, University of Miskolc, Hungary, 1997, pp. 63-69.
5. Gulyás L., Kozsik T.: *Model Design Interface – A CASE Tool for the Multi-Agent Modeling Language*. To appear in: Proceedings of the International Conference of PHD Students, University of Miskolc, Hungary, 1999.
6. Gulyás L., Kozsik T., Czabala P., Corliss, J. B.: *Telemodeling – Overview of a System*. Gordon Davies ed.: Teleteaching '98. Distance Learning, Training and Education. Short Papers, XV. IFIP World Computer Congress. Austria, 1998.
<http://www.syslab.ceu.hu/telemodeling/>
7. Gulyás L., Kozsik T., Fazekas S.: *The Multi-Agent Modeling Language*.
<http://www.syslab.ceu.hu/maml/>
8. Irwin, J., Loingtier, J.-M., Gilbert, J. R., Kiczales, G., Lamping, J., Mendhekar, A., Shpeisman, T.: *Aspect-Oriented Programming of Sparse Matrix Code*. Proceedings International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), Marina del Rey, CA. December 1997. Springer-Verlag LNCS 1343.
<http://www.parc.xerox.com/spl/groups/eca/pubs/papers/Irwin-ISCOPE97/>
9. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J.: *Aspect-Oriented Programming*. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.
10. Lopes, C. V., Kiczales, G.: *D: A Language Framework for Distributed Programming*. Technical report SPL97-010, P9710047 Xerox Palo Alto Research Center. February 1997.
<http://www.parc.xerox.com/spl/groups/eca/pubs/papers/PARC-AOP-D97/>
11. Mendhekar, A., Kiczales, G., Lamping, J.: *RG: A Case-Study for Aspect-Oriented Programming*. Technical report SPL97-009 P9710044 Xerox Palo Alto Research Center. February 1997.
<http://www.parc.xerox.com/spl/groups/eca/pubs/papers/PARC-AOP-RG97/>
12. 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>
13. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: *Object Oriented Modelling and Design*. Prentice Hall, 1991.
14. Santa Fe Institute: *The Swarm Simulation Package*.
<http://www.santafe.edu/projects/swarm/>
15. Xerox, Palo Alto Research Center: *Aspect-Oriented Programming*.
<http://www.parc.xerox.com/spl/projects/aop/>
16. Xerox, Palo Alto Research Center: *AspectJ? Home Page*.
<http://www.parc.xerox.com/spl/projects/aop/aspectj/>