Reusing Models in Multi-Agent Simulation with Software Components

Paulo Salem da Silva
University of São Paulo
Department of Computer Science
São Paulo – Brazil
salem@ime.usp.br

Ana C. V. de Melo
University of São Paulo
Department of Computer Science
São Paulo – Brazil
acvm@ime.usp.br

ABSTRACT

Simulation models are abstract representations of systems one wants to study through computer simulation. In multi-agent based simulation, such models usually represent agents and their relations. An important issue concerning these models is how they can be effectively reused across different simulations. But while much attention has been given to other engineering issues, model reuse has remained mostly untreated. To help address this issue, in this paper we present both a method and a software architecture for multi-agent simulation designed with reuse in mind. We employ software components as fundamental reusable model assets and show how their composition can also be reused. Our technique depends on some domain specific assumptions, such as the fact that agents must be related by social networks, and we argue that these are actually helpful in the context of software components. A case study is also given in order to illustrate clearly how the same component can be reused in two distinct simulation problems using our approach.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence – multiagent systems; I.6.5 [**Simulation and Modeling**]: Model Development; D.2.13 [**Software Engineering**]: Reusable Software; D.2.11 [**Software Engineering**]: Software Architectures – domain-specific architectures

General Terms

Experimentation, Design, Human Factors

Keywords

multi-agent simulation, social simulation, software components, social networks

1. INTRODUCTION

A *simulation model* is an abstract representation of some system, which one wants to analyze through computer simulation. In multi-agent based simulation, these models usually capture the behavior of individual agents, as well as their relationships. Recently, several simulation frameworks have been built in order to foster and aid multi-agent simulation [3]. Their purpose is to provide common simulation facilities, allowing designers to focus on the models to be simulated and not on the simulator's infrastructure.

The development of good models demands time and very specific skills. Hence, their reuse becomes highly desirable. Current tools, however, do not provide adequate support to such a reuse, owing to both methodological and technical problems. Usually, models are created to address one particular question, without considering the future reuse of their parts in different circumstances. Accordingly, the tools provide facilities suitable for monolithic model construction, but few special facility for reuse of their parts. Thus, most actual reuse relies on the abstractions present in the programming language used to program the models (e.g., classes, procedures). This is a very limited way of reuse, since such languages are either not designed specifically for the concerns of simulation (e.g., Java [13]) or too simple to provide useful reuse abstractions (e.g., special purpose scripting languages). The few tools that do provide some useful form of model reuse, in turn, have other limitations that we review below.

To address these issues, this paper presents a new method for multi-agent simulation and a software architecture that gives support to it. Roughly speaking, we propose that simulation models should be assembled using reusable elements pertinent to particular classes of problems. To achieve this, such elements should be created as independent units intended for third-party composition. That is, they should be *software components*, as defined in [15]. The simulator, in turn, should: (i) allow the creation of simulation models using components; (ii) answer interesting questions about such models, employing the semantics associated with the components. This semantic is attained specially through the following domain-specific assumptions: (i) agents are arranged in social networks; and (ii) agents communicate through a special data structure called *stimulus*.

Our components represent both agents and properties about such agents. They are implemented as Java classes that follow particular conventions. To employ these components in concrete simulations, it is necessary to instantiate them and compose such instances. This is achieved using *scenario* descriptions, which are also reusable assets. The simulation execution is captured in separate *experiment* descriptions. And while to code a component it is necessary to be a Java programmer, both scenario and experiment descriptions are designed to be easy to use by non-programmers. Hence, our

Cite as: Reusing Models in Multi-Agent Simulation with Software Components, Paulo Salem da Silva and Paulo Salem da Silva,Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008), Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. 1137-1144.

method defines two kinds of users, namely, those that build components and those that in fact use them.

It is worth to emphasize that we have a working Java implementation of the system described here. Notice, however, that the focus of this paper is on simulation model reuse, not on the whole simulation system. Therefore, other engineering concerns (e.g., simulation algorithms and timing issues) shall not be considered, important as they may be. Our major contributions, here, are the following: (i) a conceptual account of component-based multi-agent simulators; (ii) the technical mechanisms needed for it; and (iii) a case study crafted to show the value of model reuse as we propose.

The paper provides a gradual explanation of our approach. We begin by placing our work in context with a review of other similar works and arguing that they are insufficient for the kind of model reuse we aim (Section 2). We do not expect the reader to be familiar with the concept of software components. Therefore, we then explain what a software component is and what a component-oriented approach must provide (Section 3). Next, we introduce our component-based method for multi-agent simulation model reuse (Section 4). There we present the assumptions, artifacts, user roles and process that constitute it. The method, however, requires an actual implementation to be useful. Thus, we present the main architectural elements of our implementation and show how they explicitly support our methodological concerns (Section 5). Then, we explore a case study that demonstrates how the proposed approach can, indeed, be useful (Section 6). Finally, we discuss our achievements, reflect about the current problems and set directions for further research (Section 7).

2. RELATED WORK

A good analysis of the current state of affairs in multi-agent simulation can be found in [3]. Below, we shall examine the approaches we regard as closer to ours and point out the problems that we address in this paper.

It is generally acknowledged that it is important to separate the simulator infrastructure from the models being simulated. Swarm [7], MASON [5] and Repast [8], which are popular multi-agent simulation platforms, try to achieve this by providing both a framework to program models and a simulation engine to run them. This helps reuse simulation infrastructure, but does not simplify the reuse of parts of simulation models in different simulations built by different people.

A new version of Repast, called Repast S [9], is being developed in order, partly, to address this issue. This new version is similar to our system in that external Java classes can be arranged together declaratively (i.e., without Java programming) to compose simulation models. However, we differ from them in a number of ways. First, the simulation models of Repast S are mostly restrictions on which components are in the model, while our models carry information regarding not only the components, but also their actual instantiation (i.e., we represent a complete state of affairs). Second, Repast S has a very inclusive definition of component, so that any Java class can be a component, while we enforce several requirements in order to attain more semantics. Moreover, it seems to us that although a technology of composition is available, the methodology associated with Repast S does not foster the creation of reusable *component families*, which is one of our core concerns. Third, Repast S aims at being a general platform, while we prefer to adopt a domain-specific approach, which we believe to lead to more elegant and manageable simulation models, albeit with more limited applications. Besides the points we shall discuss later, we think that the simplicity thus achieved is also important in order to make simulators more accessible to non-programmers, which is one of our objectives.

The idea of a component-based agent simulation environment is also used in the Quicksilver project [2]. In that work, any compiled Java class can be treated as a component. Some predefined classes of agents are provided and a special tool allows the user to instantiate classes, connect instances and run the simulation thus assembled. In this way, agents can be reused in several simulations. This approach, however, suffers from some problems. Like Repast S, it relies on a very inclusive definition of component, which implies that such components do not bring any advantage over normal Java classes. The reuse technology is in the composition tool that allows arbitrary instances to be easily connected by the user, but this is not really specific to simulation, nor does it help in enforcing any special semantics to the underlying classes. Hence, such a reuse mechanism is mostly a general Java technology, which allows one to build programs in a different manner. This contrasts with our approach, in which components are highly structured entities designed for simulation: they must implement predefined interfaces, be annotated in special ways and be deployed to a special location. Furthermore, Quicksilver assumes that the user has knowledge of Java programming. This makes it inaccessible to non-programmers, whereas our approach has the contrary goal of facilitating their access to simulation.

This goal is also shared by NetLogo [18], an environment designed to simplify the creation of simulation models. To this end, a graphical editor and a set of controls (e.g., buttons, sliders, plotters) are provided in order to build the simulation front-end, while a special procedural scripting language (assumed to be easier to learn than a general purpose language such as Java) can be employed to specify the actual simulation behavior. The system, however, does not offer any reuse mechanism beyond copy-and-paste of scripts.

With similar purposes, but in a more sophisticated realization, SeSAm [4] provides a rich application in which users may create agents, setup simulations and run them. Agent creation relies on a base library of agent properties, which must be used in order to define new agents. Though simple agents can be built easily with point-and-click interaction, more advanced ones require the use of a custom scripting language. The created agents are then instantiated in order to build one or more simulation situations, all of which are stored in a model file. SeSAm, however, does not provide advanced facilities to reuse such agents across different simulation models. The only way to do so is through importing a model into another, which amounts to a copy-and-paste technique. On the other hand, programmers have the possibility of extending SeSAm through plugins written in Java. Such plugins allow the definition of new elements that the final user may employ when building his agents (e.g., new functions to be used when specifying the behavior of agents). Therefore, SeSAm does provide an interesting reuse technology, but whose purpose differs from ours, which aims at the easy reuse of whole agents and other simulation elements.

Finally, in [10] we find the ELMS markup language, which bears some similarities to our scenario language in that both

describe features of the simulation environment. However, the objective of ELMS is to restrict the kinds of entities that exist, while ours is to explicitly define and compose individual entities. Moreover, ELMS is geared towards agent-oriented programming, while our underlying programming paradigm is the more common object-oriented one.

3. SOFTWARE COMPONENTS

Before proceeding to our particular application, let us first examine what software components are in general. The fundamental ideas concerning them were given in [6], in which it was envisioned that software should be built using reusable parts, much like electronics are built using reusable integrated circuits. To this end, the task of developing software would have to be divided into two branches. One that would take care of building components useful in many different situations, and another that would develop the final software using these reusable components. This way, developers would save time by not having to rewrite software parts.

These ideas have developed through the years, and today we have a Component-Based Software Engineering field. Following the contemporary treatment of the subject found in [15], we characterize a software component as follows:

- *It is an independent unit of deployment.* That is, it can be packaged and transmitted independently of anything else;
- *It is a unity of third-party composition.* Components are designed to be reused in unknown applications, built by different people;
- *It has no externally observable state.* This is just a technical detail to make sure that the same components will always perform the same functions;
- *It has contractually specified interfaces and explicit context dependencies only.* In other words, one can know what the component requires from and provides to an application;
- *It targets a particular component platform.* Components frequently assume the existence of a platform that provides useful services.

To be used, software components must provide *component instances* (i.e., objects that do have an observable state and are, thus, useful in particular applications) and such instances must be *composed*.

Naturally, technology alone does not guarantee the creation of reusable components. One can easily use a component-oriented technology in order to build software parts for a specific application which are not reusable at all. Consequently, truly component-oriented approaches require two cares. First of all, there must be a method that defines clearly what kind of entities the components must represent, how they are composed and how such compositions are to be employed. Second, a technology that supports such a method must also be provided, in order to enforce it as much as possible.

4. METHOD

In this section we present our method for multi-agent simulation model reuse. It depends on some domain assumptions (Section 4.1), defines artifacts to be manipulated (Section 4.2), roles for the involved people (Section 4.3) and a process to be followed (Section 4.4). The technological details for each of these are omitted here and are given in Section 5.

4.1 Domain Assumptions

Our method depends on assumptions concerning agent interaction. Clearly, the manner through which agents interact play a fundamental role in determining what kinds of simulations can be created. More subtly, however, the possible interaction mechanisms also determine the facilities that a simulator can provide in order to help experimenters. The more specific these mechanisms are, the more a simulator can do automatically. In our case, we have adopted two major domain restrictions. One concerning the agents' environment and the other relating to their communication.

4.1.1 Social Network Environment

A *social network* is a graph where vertices represent agents and edges account for relationships among those agents. Agents can be individuals, organizations, machines or anything else that can be part of a relation. Relationships, in turn, can represent both physical and logical concepts. For instance, a person may be physically related with its work place (e.g., a "works at" relation), but logically related to its colleagues (e.g., a "is friend of" relation). We have chosen such networks owing to their social relevance (e.g., see [17]) and mathematical tractability (e.g., through Graph Theory).

4.1.2 Stimuli Based Communication

Agents follow a behaviorist design. Each agent is seen as a black box which receives stimuli and reacts accordingly.[1] The main communication abstraction provided by the simulator, thus, is the *stimulus*. The purpose of this design is to simplify the communication that can take place, which allows the simulator to provide useful services (e.g., useful kinds of experiments).

4.2 Artifacts

Reuse is achieved by defining three kinds of artifacts: (i) the components themselves; (ii) the descriptions of how to instantiate and arrange the components, which we call *scenarios*; and (iii) the descriptions of the simulations to perform using these scenarios, which we call *experiments*. Let us analyze each one.

4.2.1 Software Components for Simulation Models

Our components represent very specific entities. We define two kinds of components, namely:

- *Agent components*, which account for the simulated agents;
- *Property components*, which define what can be measured and calculated concerning the agents.

An agent component defines a particular kind of agent. That is, it defines how such an agent behaves and what

[1]This design follows some of the key ideas found on [11].

parameters it has. In a simulation, one must *instantiate* agent components in order to have actual agents. All of the instances of a particular component will behave similarly, varying only according to the parameters specified during instantiation. Property components are analogous.

To be useful, component instances from different components must interact with each other. Hence, components should not be developed in isolation. Rather, whole *component families* for particular domains must be developed in order to allow interesting problems in such domains to be formulated.

4.2.2 Scenarios

In order to be used, components must be instantiated in particular scenarios. Scenarios define which instances must be created and how they are related. They are also reusable assets, because they can be frequently used in distinct situations (e.g., different people might want to study a scenario in different manners) and may not be cheap to produce (e.g., because the data needed to create them is hard to obtain). Therefore, scenarios must be kept in separate files, allowing their storage and sharing. Moreover, they should be easy to be created, so that the reuse of components becomes easy as well.

4.2.3 Experiments

It is not sufficient to have a description of an interesting scenario. It is also necessary to be able to do something useful with it. To this end, experiments provide standard strategies to explore them. Owing to the domain-specific assumptions associated with our components and scenarios, such strategies can be quite informative, as we shall see in Section 5. Typically, experiments should be performed in order to answer, automatically, particular questions about scenarios, instead of just "watching" the simulation unfolds.

The same scenario can be used to perform different experiments. And the same experiment, or a slightly modified version, can be performed in different scenarios. Thus, experiment definitions should also be specified in separate files. And much like scenarios, should be easy to create.

4.3 User Roles

We define two kinds of users, namely, *component designers* and *experimenters*. The former are responsible for building the components, while the latter are supposed to create scenarios and experiments using these components. The aim of this separation of roles is to concentrate the harder implementations efforts in the components' construction and make their actual use simple. We will see that the architecture of the system enforces this: while to create a component it is necessary to be a Java programmer, to build scenarios and run experiments it is only necessary to learn a simple, special purpose, markup language.

Naturally, the same person might be both a component designer and an experimenter. Still, the idea of defining roles is useful because it allows each user to reflect about his abilities, and, based on this, adopt the roles that are more suitable to him. Moreover, when acting in one of the roles, the user has clear and distinct responsibilities. Designers must develop reusable component families, while experimenters must use them to study problems in a cost-effective manner.

4.4 Process

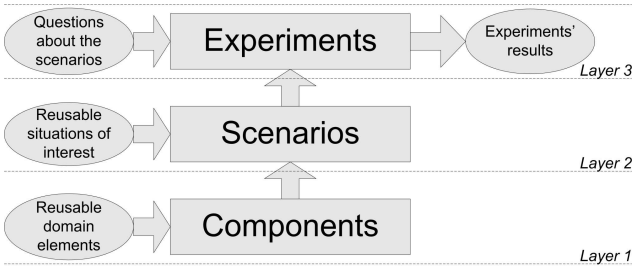

Figure 1: The dependencies between the several methodological concerns (in ellipses) and the artifacts that address them (in rectangles). Each layer has its particular worries and goals.

From the above discussion, the following process follows. First, components for a particular domain (i.e., a *component family*) must be created. Then, particular scenarios for this domain can be formulated. Finally, experimentation and analysis can be carried out using these scenarios. Hence, from the users' perspective, the method can be summarized by a layered structure as shown in Figure 1.

5. IMPLEMENTATION ARCHITECTURE

To be effective, the method described in Section 4 demands special facilities from a simulator. In this section we describe the fundamental architectural elements of our simulator, which is designed to provide such facilities. As we shall see, each of the elements described in the method have explicit architectural support and the domain assumptions made there are employed. Moreover, although the points discussed are mostly conceptual and language-independent, we shall refer to elements of our Java implementation whenever possible, for the sake of concreteness.

Components are loaded, managed and put to use by the *simulation container*. Together, components and simulation container form a complete *simulation system*. In order to use such a simulation system, the user must:

1. Deploy the components to a special directory, called *components' repository*, where the container can find them. This must be done only once for each new component that is added to the system;
2. Specify a particular scenario file to be employed;
3. Specify a particular experiment file to be employed.

After the user proceeds this way, the system executes the simulations specified in the experiment file. When it is finished, it prints the results to the user. The nature of such results depends on the kind of experiment being performed, as we will see below.

In what follows we explore our component conventions (Section 5.1) and the definition of scenarios and experiments (Section 5.2).

5.1 Components

Our components are Java classes which implement some special interfaces, are annotated in a special manner and

```
@AComponentInfo(
    id = "com.example.DiffusionAgent",
    version = 1,
    name = "Diffusion Agent",
    description = "A simple diffusion agent.",
    type = AComponentInfo.ComponentType.AGENT
 )
public class DiffusionAgent extends
                      StandardSocialNetworkAgent {

  // Agent implementation would come here.

  @AParameterInitializer(
    name = "Transmission Rate",
    description = "The probability that the agent will
                   transmit his resources to another."
  )
  public void setTransmissionRate(double tr) {
    this.transmissionRate = tr;
  }

}
```

Figure 2: An example of annotation that defines the meta-information of an agent component. First, a $AComponentInfo$ defines the general characteristics of the component. Then, $AParameterInitializer$ marks a method as a parameter initializer. Notice also that the agent is extending a $StandardSocialNetworkAgent$ class, which we provide as a convenience for component designers.

are packaged in a JAR (Java ARchive) file [14]. To create the several agents and properties specified in a scenario, the container instantiates such classes, using the parameters specified by the scenario.

The interfaces define standard ways in which the components communicate with the container and with each other. For example, every agent component must implement a predefined interface called $IAgent$.

Components also have associated meta-information that allow the container to know their names, versions and so on. We employ Java's annotations in order to attach such meta-information. Annotations are also used to mark methods that have special meaning to the simulator. For example, agents may have methods that set their *parameters* and it is important that such methods are known to the container. See Figure 2 for an example of both uses (this example will be further commented in Section 6).

There are two kinds of components, each defined by specific interfaces and annotations. Let us examine the general characteristics of each one.

5.1.1 Agent Components

Agent components are responsible for specifying kinds of agents that one wishes to simulate. In a simulation scenario, one specifies several instances of the available agent components. Each instance represents one particular agent with particular parameters. Their existence and interaction are supported by the following elements:

Environment Agents exist within a social network environment. Such an environment provides access to operations regarding agents' locations. For example, an agent may use its environment to discover neighboring agents.

Relations The relationships between agents are stored in the form of *relations*. These relations are analogous to mathematical ones: they are ordered pairs of elements. Each scenario may have several relations and each of them represents a particular kind of relationship. Agents may access and manipulate these relations. In particular, they can change them dynamically as the simulation unfolds.

Stimuli Agents can communicate with any other agents within the simulation, even with those to which they are not related. For this purpose, each agent implements an interface that can be accessed by others, the $IStimulusReceiver$. References to other agents, in the form of $IAgent$ interfaces, can be obtained in several ways (e.g., querying the environment as explained above). Once an agent holds a reference to another, it may send "him" a *stimulus*, which is the structure responsible for transmitting information to agents.

Agent behavior must be implemented in the *step*() method of the $IAgentControl$ interface. At every instant a *simulation cycle* is performed and the container calls this method for each agent in the simulation. Hence, time is discrete and equal for all agents, which act one after another during each cycle.

5.1.2 Property Components

Like agents, properties are a kind of component and are instantiated in scenarios. Their purpose is to provide assertions that experimenters might want to test, analyze or store concerning agents and their environment. Properties generate outputs in the form of numbers, truth values and text. How such values are calculated is entirely up to their developers.

Properties have targets. Either individual agents or their environment can be targets. Moreover, one property instance can be targeted at several different elements. The experimenter writing the scenario is responsible for instantiating properties and assigning targets to them.

The fact that properties are chosen by the experimenter facilitates abstraction during simulation analysis. It is possible to focus simulation resources only on what is important during a particular experiment, while still being able to change such a focus in later experiments (i.e., by instantiating different properties). Furthermore, new analysis can be performed as new properties are developed.

Since we assume that agents are organized in social networks, it is possible to provide several reusable properties. For example, in [17] several measures concerning such networks are defined (e.g., prestige, centrality).

5.2 Scenarios and Experiments

Scenarios and experiments descriptions are designed with the following goals:

- They must be easy to write even by non-programmers;
- It must be easy for other tools to read and write them (i.e., to allow interoperability of tools);
- They must make full use of domain assumptions, in order to keep them as elegant as possible.

To this end, they are specified as XML (Extensible Markup Language) [16] files following special markup languages. Both are abstractions for experimenters, not component designers.

5.2.1 Scenarios

A *scenario* defines an initial state for the simulation, built on top of the available components. It is formed by agents, properties and relations declarations. Concerning agents and properties, the following must be specified: (i) the component's identifier from which the agent is instantiated; (ii) the name and a unique identifier for the agent; (iii) primitive parameters to be used when instantiating the agent; and (iv) parameters in the form of lists to be used when instantiating the agent. Furthermore, properties must have targets (i.e., the objects used to calculate the value of the property), which can be either individual agents or the whole environment.

As for relations, the following is necessary: (i) a name, unique identifier and description for the relation; (ii) its several relational ties. Each tie indicates that some agent is related to another agent, using the agent's ids.

Figure 3 presents the general form of a scenario file.

Notice that the declaration of relations arise from the fact that our approach assumes that agents are arranged in social networks. If no such assumption existed, then it would not be possible to define such a simple markup.

5.2.2 Experiments

An *experiment* define which simulations should be performed. As the name suggests, experiments are created in order to discover new information or test hypothesis about a scenario. Each experiment contains the instantiation of one or more *simulation strategies*, which are simulation algorithms provided by the container. Different strategies answer different questions and, thus, are suitable for different experiments.

In the present version of our system, we have implemented the following two strategies:

Standard simulation The straightforward simulation of a scenario, without any special algorithm. Changes to the scenario are simulated according to the behavior of its agents. The final state is presented to the user. Broadly speaking, this strategy is analogous to the kind of simulation that the tools reviewed in Section 2 usually provide. In our system, however, this is just a special case;

Stimulus delivery optimization Provides a way to discover to which subgroup of agents a specified stimulus should be delivered in order to maximize the value of some property of interest (e.g., the spreading of information in a social network, which is a usual question concerning such networks). To this end, several simulations are performed, each one considering a particular subgroup of agents. When the strategy finishes, it presents the list of the best agents to which the specified stimulus should be delivered. Figure 4 presents the general form of an experiment employing this strategy. Notice that this strategy employs the assumption of stimuli-based agent communication.

While we realize that only two strategies provide a limited repertoire, the architecture is designed to allow the implementation of new strategies (e.g., by providing a suitable base class for later extension). In fact, we are currently developing other kinds of strategies (e.g., other kinds of optimizations, pattern recognition). And since domain-specific assumptions are available to the strategies, we think that a varied and useful repertoire can be created.

```
<scenario name="###" description="###">

    <agent component-id="###" id="###" name="###"/>
        <primitive-parameter name="###" value="###"/>
        <primitive-parameter name="###" value="###"/>
        (...)

        <list-parameter name="###">
          <parameter-value value="###"/>
          <parameter-value value="###"/>
          <parameter-value value="###"/>
        </list-parameter>
        (...)
    </agent>
    (...)

    <!-- Other similar <agent></agent>
         declarations come here -->

    <property component-id="###" id="###" name="###">
        <primitive-parameter name="###" value="###" />
        <primitive-parameter name="###" value="###" />
        (...)

        <agent-target id="###" />
        <agent-target id="###" />
        (...)

        <environment-target />
    </property>
    (...)

    <!-- Other similar <property></property>
         declarations come here -->

    <relation id="###" name="###" description="###">
        <tie id1="###" id2="###"/>
        <tie id1="###" id2="###"/>
        (...)
    </relation>
    (...)

    <!-- Other similar <relation></relation>
         declarations come here -->

</scenario>
```

Figure 3: The general form of a scenario file. The "###" marks indicate that the text should be chosen by the user. The "(...)" marks denote possible repetition of the previous element.

6. CASE STUDY: MODELLING DIFFUSION IN SOCIAL NETWORKS

Let us now study a concrete example to illustrate how the proposed approach can improve model reuse. Following the method given in Section 4, we shall first examine one kind of agent and define a component that captures its behavior (Section 6.1). We then show how it can be reused to create scenarios and experiments in two rather different simulation domains, namely, Epidemiology (Section 6.2) and Marketing

```
<experiment name="###" description="###" >
    <stimulus-delivery-optimization
           name="###" runs="###" iterations-per-run="###"
           property-id="###" targets="###">

        <agent-target id="###" />

        <stimulus type="###" content="###">
            <referenced-agents>
                <agent id="###"/>
                (...)
            </referenced-agents>
        </stimulus>

    </stimulus-delivery-optimization>
    (...)
</experiment>
```

Figure 4: The general form of a stimulus delivery optimization experiment. The "###" marks indicate that the text should be chosen by the user. The "(...)" marks denote possible repetition of the previous element.

(Section 6.3).

6.1 A Reusable Agent Component

In social networks, an important issue concerns how resources are diffused among agents. That is, given that a particular group of agents in the network holds a resource, one wishes to study how they transmit it to the other agents over time. This sort of study can be influenced by several factors, such as the network structure (e.g., diffusion is usually faster in a network in which every agent has many of neighbors) and the behavior of agents (e.g., agents may be specially susceptible to particular kinds of resources).

These resources can be many things. For example, they may represent opinions, information, diseases or money, just to name a few. It is in this variety that the possibility of model reuse is: since many problems can be cast in terms of resource diffusion over social networks, we may ask ourselves if it would also be possible to program a single agent component that could be used to simulate all of these problems.

And, indeed, we can program it. To achieve this, it suffices to look for the agent parameters necessary to describe diffusion. The following are frequently used:

- *Resource possession.* A boolean value that indicates whether the agent holds the resource.
- *Transmission rate.* A probability that indicates how likely an agent is to transmit a resource to his neighbors.
- *Susceptibility.* A probability that indicates how likely an agent is to accept a resource transmitted to him by a neighbor.
- *Immunity.* A boolean value that indicates whether an agent becomes immune to transmission after it has held the resource at least one time.

Then it is simply a matter of programming an agent component that implements exactly the behavior specified by these parameters. Figure 2 showed some annotations that provide the meta-information about the component. In order to use the component, an experimenter must only write a scenario where agent instances are specified as instantiations of this component.

For convenience, let us call the above parameters as *diffusion parameters* and the agents that employ them as *diffusion agents*.

6.2 Reuse in an Epidemiological Simulation

A central concern of Epidemiology is how diseases are disseminated among a population. One approach to studying this problem is modelling the population as a social network of agents (e.g., [1]) and employing diffusion parameters associated with particular diseases. Using our agent component, an experimenter could proceed, for instance, in the following way:

1. Choose a social group that can be modeled as a social network (e.g., in [1] such a network is given for a particular social group).
2. Write a scenario file that describes the instantiations of the diffusion agent component. Each instance must be assigned parameters. Typically, these parameters will reflect what is known about the disease and the social group being analyzed. For example, diseases which are highly contagious will have a high transmission rate. The style of the scenario file is always the same, similar to what was shown in Figure 3.
3. Write an experiment to simulate the scenario, using the standard simulation strategy of Section 5.2.2.
4. Run the simulation system specifying the input scenario and experiment.
5. Examine the final state of the social network to see how many agents were infected in the end.

In order to analyze several possible situations, the experimenter may create several scenarios, which vary only in some characteristic of interest (e.g., the agents initially infected). Moreover, these scenarios can be shared by many experimenters, each one analyzing different questions.

6.3 Reuse in a Marketing Simulation

In Marketing, product advertisement is of great importance. Typically, given a fixed budget, one wishes to make sure an advertisement reaches the largest possible group of consumers. One way to achieve this is to rely on the social network structure that connects consumers: they have friends and usually talk about products they know to each other. By advertising to the correct consumers, one can reach consumers that have not been exposed directly to the advertisement.

Transmission rate and *immunity* diffusion parameters can be used to model this kind of information spread over social networks, as shown in [12]. Using this sort of modelling, an experimenter can proceed in a manner similar to the epidemiological simulation we saw above, but with some particularities.

The social network structure for the target audience must be available. In general, this might be hard to obtain. However, in some particular domains it is feasible. For example, if the target audience is an online community (i.e., a website

in which users have personal pages and provide links to their friends), it is easy to extract a social network: it suffices to analyze the link structure between user pages.

To perform experiments, it is better to employ the stimulus delivery optimization strategy presented in Section 5.2.2. To do so, one first estimates how much it costs to advertise to an individual consumer. Then, one divides the available advertisement budged by the cost per consumer. This gives the size k of the subgroup required by the simulation strategy. Finally, we must specify a property to work as an objective function. We can use a simple standard property that counts how many agents have been reached by the advertisement.

After running the experiment, the user gets a list of the k best agents to which the advertisement should be delivered in order to maximize network diffusion.

7. CONCLUSION

In this paper we presented a method and a related implementation architecture for reusing model parts in multi-agent simulation. Our method is based on special software components, which represent agents and properties about agents. These components are instantiated in scenarios, which are also reusable assets. Experiment files define how to perform simulations and can be applied to multiple scenarios.

The case study presented a simple example of an agent component that is useful in distinct simulation problems. In that example, reuse was achieved entirely without Java programming, employing only simple XML declarations. Such simple solutions, however, depend on the possibility of creating component families flexible enough so that many interesting scenarios can be assembled. We showed that this is possible in this simple case, but more research needs to be done in order to assess this possibility in more general terms.

We have emphasized the domain specific nature of our simulation system. Of course it would be better to have a simulator capable of providing facilities for all kinds of domains. However, our point is that the simulator infrastructure should have as many assumptions as possible about what is being simulated in order to provide more useful mechanisms. In this paper, we argued that the method outlined in Section 4 is useful with some particular assumptions (e.g., social network environments). But the same components-scenarios-experiments method could be applied using different assumptions (e.g., grid-based environments), though, in this case, a different implementation would also be needed.

Another relevant point regards the creation of scenarios. Small scenarios can be composed by hand, but scenarios that capture a large network of agents (e.g., with thousands of agents) need special tool support to be generated. Our XML representation should facilitate interoperability with such tools.

Finally, in order to facilitate the automatic analysis of simulations, we have provided a special kind of component, called property. However, no method of making logical assertions (e.g., "such and such property must have a particular value") is currently available. Therefore, the next step in this direction is to provide an appropriate logical framework, which will allow the creation of more sophisticated simulation strategies.

8. REFERENCES

[1] P. S. Bearman, J. Moody, and K. Stovel. Chains of affection: The structure of adolescent romantic and sexual networks. *American Journal of Sociology*, 110(1):44–91, 2004.

[2] J. Burse. Quicksilver: A component-based environment for agent-based computer models and simulations. 2000.

[3] N. Gilbert and S. Bankers. Platforms and methods for agent-based modeling. *Proceedings of the National Academy of Sciences of the United States*, 99(Supplement 3), 2002.

[4] F. Klügl and F. Puppe. The multi-agent simulation environment SeSAm.

[5] S. Luke, C. Cioffi-Revilla, L. Panait, and K. Sullivan. MASON: A new multi-agent simulation toolkit. 2004. http://cs.gmu.edu/ eclab/projects/mason/.

[6] M. McIlroy. Mass produced software components. In P.Naur and B. Randel, editors, *NATO Conference on Software Engineering*. NATO Science Commitee, 1968.

[7] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The Swarm simulation system: A toolkit for building multi-agent simulations. 1996. Working Paper 96-06-042.

[8] M. North, N. Collier, and J. R. Vos. Experiences creating three implementations of the Repast agent modeling toolkit. *ACM Transactions on Modeling and Computer Simulation*, 16(1):1–25, 2006. http://repast.sourceforge.net/.

[9] M. North, T. Howe, N. Collier, and R. Vos. The repast simphony runtime system. 2005. http://repast.sourceforge.net/.

[10] F. Y. Okuyama, R. H. Bordini, and A. C. da Rocha Costa. ELMS: An environment description language for multi-agent simulation. In D. W. et al., editor, *Environments for Multi-Agent Systems*, volume 3374 of *Lecture Notes in Artificial Intelligence*, pages 91–108. Springer-Verlag, 2005.

[11] P. S. da Silva and A. C. V. de Melo. A simulation-oriented formalization for a psychological theory. In M. B. Dwyer and A. Lopes, editors, *FASE 2007 Proceedings*, volume 4422 of *Lecture Notes in Computer Science*, pages 42–56. Springer-Verlag, 2007.

[12] R. Stocker, D. G. Green, and D. Newth. Consensus and cohesion in simulated social networks. *Journal of Artificial Societies and Social Simulation*, 4(4), 2001. http://jasss.soc.surrey.ac.uk/4/4/5.html.

[13] Sun Microsystems. Java technology. 2007. http://java.sun.com/.

[14] Sun Microsystems. Lesson: Packaging programs in JAR files. 2007. http://java.sun.com/docs/books/tutorial/deployment/jar/.

[15] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 1999.

[16] The World Wide Web Consortium. Extensible markup language (XML). 2007. http://www.w3.org/XML/.

[17] S. Wasserman, K. Faust, D. Iacobucci, and M. Granovetter. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[18] U. Wilensky. Netlogo. 1999. http://ccl.northwestern.edu/netlogo/.