A Flexible Simulation Architecture for Pandemic Influenza Simulation

Henrik Eriksson, PhD¹, Toomas Timpka, MD, PhD¹, Joakim Ekberg, PhD², Armin Spreco², Örjan Dahlström, PhD², Magnus Strömgren, PhD³, Einar Holm, PhD³

¹ Dept. of Comp. and Inform. Sci., Linköping University, Sweden;
² Dept. of Medical and Health Sci., Linköping University, Sweden;
³ Dept. of Social and Economic Geography, Umeå University, Umeå, Sweden

Abstract

Simulation is an important resource for studying the dynamics of pandemic influenza and predicting the potential impact of interventions. However, there are several challenges for the design of such simulator architectures. Specifically, it is difficult to develop simulators that combine flexibility with run-time performance. This tradeoff is problematic in the pandemic-response setting because it makes it challenging to extend and adapt simulators for ongoing situations where rapid results are indispensable.

Simulation architectures based on aspect-oriented programming can model specific concerns of the simulator and can allow developers to rapidly extend the simulator in new ways without sacrificing run-time performance. It is possible to use such aspects in conjunction with separate simulation models, which define community, disease, and intervention properties. The implication of this research for pandemic response is that aspects can add a novel layer of flexibility to simulation environments, which enables modelers to extend the simulator run-time component to new requirements that go beyond the original modeling framework.

Introduction

Computer-based simulation is an important resource for analyzing the dissemination of novel infectious disease agents before empirical data are available [1]. Such simulations can provide valuable information about the relative impact of different potential interventions measures and estimate the relative susceptibility in different types of communities [2]. A major challenge for simulation architectures to be used in these settings is to provide sufficient turn-around time for simulation runs. In addition to the run-time performance per se, the simulation architectures must provide sufficient flexibility to allow for changes to both the simulation model and to the simulation performance engine. As in many other types of software systems, there is a tradeoff between run-time performance and modeling flexibility. Hard-coded and hand-tuned simulator implementations can take advantage of available hardware, such as multi-core processors, and can achieve acceptable or even respectable performance. It is a significant challenge to combine performance and flexibility to attain a well-balanced simulation system that can support interactive modeling, scenario development, and simulation result analysis for research and decision-support tasks.

Previously, we have studied stimulation architectures that take advantage of ontology-based models for disease, community, and intervention aspects [3]. In this approach, we use an ontology to model and specify different simulation scenarios and a translator to produce an XML-based specification for the simulation engine. This approach supports flexible modeling and allows scenario developers to use ontology editors such as Protégé [4] to define different parts of the simulation model and to reuse model components by using ontologies as a library of scenario building blocks.

Although this method provides flexibility and a systematic way of developing models, it is still limited to the expressibility of the ontology language and the subsequent translation process to the specification for the simulator engine. There are still qualities that cannot be modeled easily using this approach. For example, it is not possible to extend the capabilities of the simulation engine and introduce new features, such as intervention types and new types of data collection from the simulation process.

The aim of this study is to introduce an architectural approach to provide an additional layer of flexibility to simulations of influenza pandemics while maintaining run-time performance. Aspect-oriented programming [5,6] offers new ways of developing simulation architectures that alleviates design compromises originating from the tradeoff between run-time performance and modeling flexibility. This architectural approach allows us to add new features to the simulator without explicitly changing the core functionality and potentially degrading performance by
introducing more overhead (e.g., overhead from extension programming interfaces). Another advantage of this approach is that it is possible to use aspects in different combinations to achieve alternative configurations of the simulator. This technique makes it possible to build different versions of the simulator for specific tasks, such as additional logging, output in new formats, and new types of interventions.

**Background**

Aspect-oriented programming (AOP) is a programming paradigm that builds on object-oriented programming to provide mechanisms for modeling crosscutting concerns at the source-code level throughout the program. Here, crosscutting concerns are widespread functionality that cannot easily be modeled in a specific part of the program (e.g., as an object). Examples of crosscutting concerns are logging, consistency checks, and error handling. There are several programming languages for AOP, for example AspectJ [5] and AspectC++ [6]. These languages extend the existing programming languages Java and C/C++, respectively, with language elements for AOP rather than providing completely new languages. In AspectC++, programmers write aspect definitions in special source-code header files, which the AspectC++ compiler `ac++` then transforms, or weaves, into C++ source code suitable for compilation to executables using standard C++ compilers, such as GNU `g++` and Microsoft VisualC++.

There are different modes for the weaving process. AspectJ has some additional features in that it supports weaving of compiled code (Java byte code), and allows for load-time weaving, which means that the run-time system weaves classes as they are loaded into the Java virtual machine. This type of flexibility is not available in AspectC++, because it is restricted to source-code weaving only. However, for run-time performance, there are clear advantages of source-code weaving. For example, source-code weaving (1) enables the compiler to perform optimizations on the complete (weaved) code, (2) allows aspects to introduce source-level constructs (potentially compiler directives and `#pragma` statements), and (3) facilitates manual inspection of weaved code if required.

One of the advantages of AOP is that it is possible to weave the program with different aspects to achieve radically different behavior from the program depending on the aspect configuration. For example, without any aspects a simulator could perform a basis simulation of an outbreak. However, with additional aspects weaved into the simulator code the program could collect additional information and statistics about the simulated outbreak, such as the gender and age distributions of infected persons. By combining different aspects, it is possible to achieve different program configurations and to develop special-purpose versions of simulators for specific tasks, such as experiments setups. In AspectC++, this type of configuration means that the program is weaved and compiled with a set of enabled aspects to form a new executable for each configuration.

**Methods**

The starting point for this work was an outbreak-simulator architecture developed in C++ [3]. As an initial step, we refactored the simulator source code to improve its structure and added unit tests (using CppUnit) to confirm proper functionality from the start. Next, we added testing aspects to ensure correct behavior from weaved code when running simulations. We then proceeded to evaluating the aspect-oriented architecture by implementing aspects for additional functionality, such as saving results directly to databases, checking consistency and supporting interventions.

**Results**

*Fundamental simulation architecture*

The fundamental simulation architecture is based on a simulator part that takes as input an XML-based specification of the simulation parameters and produces an XML file with the results [3]. In addition to the simulator, the architecture takes advantage of a scenario ontology that contains models of the community, disease, and basic interventions. Scenario developers use the ontology editor Protégé [4] to develop the scenario by instantiating concepts in the ontology. A Protégé extension transforms this ontology to the XML-based specification for the simulator core.
Figure 1 shows an overview of the architecture. The simulator consists of a simulator core algorithm with supporting modules for parsing specifications, applying interventions, logging, preparing result output, and so on. The architecture is based on object-oriented programming, which models essential modules and data components as C++ objects for efficient computations. From a user perspective, analysts can use the simulator architecture to prepare scenarios in the ontology editor, run them in the simulator, analyze the results, and, if required, revise the scenarios.

The main advantage to using ontologies for specification of scenarios and simulation parameters is that it allows for (1) modeling flexibility supported by the expressivity of the ontology and the high-level editing tool and (2) reuse of model components across scenarios where the ontology acts as a scenario-component library. Although such ontology-based specifications work well, there are sometimes scenario properties that cannot be modeled easily using ontologies. These situations require additional flexibility. For example, the scenario developer may want to add concepts to the ontology that currently have no mapping to the XML-based parameters that simulator uses as input. Furthermore, the simulator implementation may not support these concepts at all. There is a need for a modeling middle ground where developers can add functionality to the simulator without actually interfering with the simulator code and modifying it manually (which is sometimes difficult and time-consuming to do and potentially error prone).

**Aspect-oriented simulation architecture**

Based on the fundamental simulation architecture, we have developed a new model for aspect-oriented simulation architectures. By taking advantage of aspects it is possible to develop simulators that are flexible in the sense that the basic simulation system can remain small and relatively unchanged while developers can add new aspects independently to change the simulator behavior significantly. Furthermore, it is possible to use the aspects in different configurations to effectively create diverse versions of the simulator architecture. As simulation
requirements evolve and usage patterns change, developers can define additional aspects and add them to the architecture.

Figure 2 shows the simulator architecture populated with aspects. In this architectural approach, the aspects add to the existing simulator code by modifying the behavior of the simulator modules at specific points and by possibly extending the data model by introduction of new member field and functions to existing classes. It is possible to define aspects that address crosscutting concerns by extending modules such as the user interface, parameter parser, data model, and output with additional functionality. For example, the developer may want to extend the simulator with functionality for taking advantage of new data sources (e.g., new types of community-related data) by enhancing the parser and other data-input modules in combination with introducing new properties to the data model.

Figure 2. Simulator architecture with aspects. Aspects weaved into the original simulator use pointcut expressions to add advice to simulator modules to extend their functionality. In this configuration, the aspects extend the simulator with functions for new interventions, output formats, logging, consistency checks, and so on.

Potential simulator-architecture growth points

Growth points are areas where it is especially interesting to extend existing systems. Developers of APIs often preplan software extensions, sometimes tacitly, by identifying areas where other developers may want to add functionality to a system. For instance, an API may support the growth point of adding import filters for new data and media formats (i.e., by providing appropriate calls and hooks). In outbreak simulators, it possible to identify candidate growth points where it is especially interesting to introduce extensions beyond what is possible to express using simulation parameters and configuration files. Note that both modeling interests (e.g., refined disease and community models) and implementation requirements (e.g., internal consistency checking and logging) can motivate such simulator growth points. Let us discuss potential simulator-architecture growth points that are possible to approach using aspects.

1. Data sources. New types of data as well as old data in new formats can become available to the simulator. It is possible to use aspects to develop import filters and data converter that parse and translate the input data to the internal data model. For example, aspects can add advice to an input XML parser to enable it to recognize new tags for new data elements.

2. Data models. The underlying data model for the simulation (e.g., classes for persons, mixing groups and other domain-specific resources) may be insufficient for certain types of scenarios. For example, public-
health professionals may want to introduce socio-economic factors in the simulation, which requires new properties in the data model. Aspects can use introductions to add to existing class definitions for central simulator concepts, such as persons and mixing groups.

3. Intervention modeling. Although it is possible to model basic interventions in the scenario ontology, there are complex interventions and combinations of interventions that cannot readily be modeled in this way. The simulator architecture can benefit from having an additional, advanced, way of modeling interventions. If needed, aspects can implement interventions using program code, which maximizes the flexibility in intervention modeling.

4. Tracing, logging, and run-time statistics. Simulators must produce sufficient records of the simulation runs, especially for lengthy jobs running for days or weeks. Developers can use aspects to implement and extend various maintenance and recordkeeping functionalities of the simulator. In AspectC++, for example pointcut expressions are useful for pattern matching of function calls for tracing and logging.

5. Consistency checks and unit tests. To ensure correct simulator operation, developers often add data-consistency checks and unit tests. It is possible to use aspects for implementing and applying such tests as well as supplementing existing ones.

6. Result output. Analysts will rarely examine the raw simulation output directly. Normally, the simulation results will be stored and used as input to others tools, such as plotting and statistics packages. It is possible to use aspects to implement support for new output formats (e.g., suitable for efficient storage and processing) and to output additional data of interest from the simulation run, such as daily reports of the age of infected people.

7. Problem scalability. New scenario definitions can increase demand on the simulator, for instance in terms of community population and data granularity, which require increased memory and processing capabilities. Aspects can improve scalability by (1) changing data models (i.e., memory requirements) and (2) adding pragma statements, for example Open Multi-Processing (OpenMP) pragma directives for taking advantage of multicore processor architectures.

These growth points exemplify concerns that are fundamental to the simulator and, therefore, difficult to model as simulation parameters and in simulator configuration files. Aspects can contribute to simulator architectures by supporting such growth without developing full-grown API for each of them (which may add overhead and be difficult to implement because of their fundamental nature).

Architecture Evaluation

Let us consider two sample aspects as a basic evaluation of the architecture model. The aim of the first aspect is to change the output of the simulator. Figure 3 shows a sample aspect that adds an additional output format, HDF5, for the simulator results. HDF5 is a binary file format for storing large amounts of numerical data, such as measurements and other types of scientific data. This aspect adds a before advice to the simulation method to create and open the HDF5 file (by calling the H5Fcreate function in the HDF5 API) and an after advice to close the HDF5 file after finishing the simulation (by calling the H5Fclose function). Furthermore, the aspect adds an after advice to write the dataset to the HDF5 file (by calling the H5LTmake_dataset function) after the simulator has produced an array that summarizes the number of active infections for each day. The aspect defined in Figure 3 illustrates that it is possible to extend simulator functionality with a fairly compact aspect to achieve useful functionality. Moreover, it is possible to do so without requiring a formal extension API, which may unavailable or at least time-consuming to implement especially for specific situations.
Another application area for aspects in simulation architectures is implementation of interventions. Because of the variation of possible interventions, it is difficult to model many of them as simple statement or expression (e.g., an object with properties and logical expression). Potential interventions range from different types of social-distancing measures, such as avoiding physical contacts, canceling events, and closing locations, to immunization programs. Furthermore, the interventions can be applied at different times, according to different criteria, and in various combinations. Also, intervention compliance may vary. Therefore, simulation architectures require mechanisms for defining intervention models that are both simple and sufficiently expressive without actually making extensive modifications to the simulator source code.

The aim of the second aspect used for evaluation is to explore intervention modeling. Figure 4 illustrates how it is possible to implement interventions in the simulation architecture. This aspect adds functionality for closing and reopening schools in a community at different times in the simulation. The principal idea is to implement school closure by modifying the behavior of the simulator in such a way that school children do not get infected at schools. This task is accomplished by capturing the call to the method calculating the probability of people not getting infected and returning 1.0 if the mixing group is marked as closed. To control the closure and reopening of school mixing groups, the aspect adds a check before the call to the method responsible for stepping the simulation one day. This advice marks all school mixing groups as closed if the current simulation time is equal to the specified day of closure. Likewise, the advice marks all mixing groups as open once the current simulation time reaches the day of reopening. Furthermore, the aspect introduces a slice, which adds a closed property (data member), to the to the class representing mixing groups (Group). In addition, this slice enhances the class with member functions for closing, opening, and checking the closure status of the mixing group.
To illustrate the effect of using the school-closure aspect, let us examine the effect of applying the intervention in different configurations. Figure 5 shows the results from running the simulator with the aspect in different modes and rerunning the simulations ten times for each case. The baseline case is a simulation of an outbreak in a community without any intervention (Figure 5a). The intervention aspect was then configured to close all schools on day 10 of the outbreak and reopen them on day 20 (Figure 5b) and, finally, to close on day 20 and reopen on day 30 (Figure 5c). The outbreak curves show the distinct differences in the result profiles of the configurations. Note that this experiment is an evaluation of the intervention modeled by an aspect rather than a simulation aimed at exploring
public health policies (which would have required additional scenarios, background data, repeated simulation runs, and result analysis).

**Figure 5.** Results from outbreak simulation using the school-closure intervention aspect. Each simulation is repeated ten times and the resulting curves show the number of active infections over time. (a) No intervention. Schools are open as usual. (b) All schools close on day 10 of the outbreak and reopen on day 20. (c) All schools close on day 20 of the outbreak and reopen on day 30.

**Architectural model and development approach**

Ultimately, aim of this work is a generalized architectural model that directs the implementation of flexible simulators. Figure 6 shows the conceptual model for the aspect-oriented simulation architecture. The guiding architectural principle is that there should be a core simulation system and data model with basic capabilities, and that aspects should add functionality in different configurations. Normally, the core simulation system is implemented as a conventional object-oriented program whereas the adjacent aspects are implemented in a corresponding aspect-oriented language (e.g., AspectC++ in the case of C++ and AspectJ in the case of Java). Using this approach, developers can implement the simulation architecture by establishing the core functionality and then add aspects around the core incrementally during the development process. For preexisting simulator implementations, developers can use a refactoring approach where they identify the core simulator part and then gradually extract existing features from the simulator system and refactor them as aspects.

It is possible to prepare the core simulation system for aspects extensions. Although aspect-oriented programming does not require an API for making extensions, it is still useful to define areas for possible extensions by paying attention to the software design. For example, it is advantageous to employ a (class and function) naming convention that enables aspect pointcut expressions to pattern match on the names. Furthermore, it is useful to promote growth points by adding (perhaps dummy) function calls at specific locations of interest in the algorithm code to allow for aspects to add advice. For example, an algorithm for calculating probabilities of infections among individuals in a mixing group can call additional (and perhaps dummy) functions at different computational stages to permit aspects to add advice through pointcut expressions.
Discussion

Aspect orientation enables flexible simulator architectures that support scenario modeling beyond what is possible in simulation parameters and configuration files. The use of aspects to extend a core simulation system has the advantage of allowing developers to improve the architecture incrementally by adding aspects individually and independently. Furthermore, this type of architecture support extensions without predefined API mechanisms, which may add run-time performance overhead especially for lengthy simulation jobs. Thereby, aspect-oriented simulator architectures provide significant advantages to pandemic influenza modelers when compared to present static simulation frameworks [7,8].

Although aspect-oriented programming is a powerful technique, there are certain modeling drawbacks. In particular, aspect-oriented code is difficult to read and debug in the sense that is difficult to know exactly what code is executing at specific locations. This is because several aspects may change and add to the behavior of functions through advice code hidden from immediate view. Nevertheless, there are joinpoint visualization tools that aid developers in reading aspect and advice code, for example the AspectC++ Development Tools (ACDT) and AspectJ Development Tools (AJDT) for the Eclipse development environment.

Developers constantly make different kinds of modeling decisions. In this type of architecture, one particular consideration is the choice between objects and aspects; that is, which modeling paradigm to follow. In some situations, it is possible to implement the same functionality as objects in the simulator core or, alternatively, as aspects added to the architecture. Our preferred guideline for such modeling decisions is to (1) use object-oriented programming for basic and common functionality, such as the simulator core, and (2) reserve aspects for extended and independent functionality while avoiding aspect interdependencies where aspects rely in the functionality of other aspects. In other words, we refrain from using aspects in the simulation core to promote simplicity and to avoid potential programming errors that could affect the simulation outcome.

An alternative to aspect-oriented programming is to use traditional object-oriented programming and define conventional APIs for extensions. This approach has the advantage of not bringing in the complexity of aspect-oriented programming and the weaving process. This method also forces simulator developers to identify growth points to support and implement well-defined interfaces among the architectural components. However, the disadvantage of this API approach is that it is less flexible than aspect-oriented programming and that it is difficult to achieve the same run-time performance as aspects because of the overhead associated with programming interfaces.
In aspect-oriented programming, it is possible to use introductions to extend the basic data model, as discussed previously. We have seen that aspect developers can extend the definition of a person (i.e., the C++ class representing persons in the simulation) with additional fields to represent properties such as age and primary-care provider. Doing so in programming interfaces is possible, but at the cost of additional classes and computational overhead. For example, object-oriented APIs often take advantage of polymorphism and virtual member functions (methods) to allow extension by addition of new subclasses. Unfortunately, the use of virtual function calls in tight simulation loops can be quite costly (because of the dynamic run-time lookup of virtual member functions). Aspects can avoid such costly look-up operations by weaving advice with regular member functions.

**Conclusion**

Simulation of influenza pandemics is important for response planning because of the possibility to predict outbreak dynamics and the relative benefit of interventions before empirical data are available. In general, there is a trade-off between flexibility in configuring simulators and run-time simulator performance. In this work, we have explored a novel simulator-architecture approach based on aspect-oriented programming. The results illustrates that it is possible to develop simulator environments based on a core system combined with extensions implemented by aspects. The aspect-oriented approach makes it possible to avoid the potential overhead of using programming interfaces and generalized system designs with complex mechanisms for extensions. In summary, we believe that this type of flexible and extendable architecture is an important step towards rapid modeling and simulation for pandemic preparedness and response.

**Acknowledgements**

The Swedish Research Council supported this work under contracts 2008-5252 and 2009-6291.

**References**