Hindawi Publishing Corporation
Modelling and Simulation in Engineering
Volume 2014, Article ID 736359, 16 pages
http://dx.doi.org/10.1155/2014/736359

Research Article

Practical Benefits of Aspect-Oriented Programming Paradigm in Discrete Event Simulation

Meriem Chibani,[1] Brahim Belattar,[2] and Abdelhabib Bourouis[1]

[1]*Research Laboratory on Computer Science's Complex Systems ReLa(CS)2, University of Oum El Bouaghi, P.O. Box 358, 04000 Oum El Bouaghi, Algeria*
[2]*Department of Computer Science, University of Batna, 5000 Batna, Algeria*

Correspondence should be addressed to Meriem Chibani; chibani_meriem@live.fr

Received 16 July 2014; Revised 20 October 2014; Accepted 20 October 2014; Published 28 December 2014

Academic Editor: Dimitrios E. Manolakos

Aspect-oriented modeling and simulation is a new approach which uses the separation of concerns principle to enhance the quality of models and simulation tools. It adopts the separation of concerns (SOC) principle. Thus, crosscutting concerns such as processes synchronization, steady state detection, and graphical animation could be separated from simulation functional modules. The capture of crosscutting concerns in a modular way is carried out to cope with complexity and to achieve the required engineering quality factors such as robustness, modularity, adaptability, and reusability. This paper provides a summary of aspect-oriented paradigm with its usage in simulation by illustrating the main crosscutting concerns that may infect simulation systems. A practical example is given with the use of the Japrosim discrete event simulation library.

1. Introduction

When designing software, there are usually certain requirements and considerations to be carried out in order to cope with complexity and to achieve the required engineering quality factors such as robustness, adaptability, and reusability. These requirements or considerations are called concerns. Obtaining the expected granularity of the identification and the separation of concerns is named the separation of concerns (SOC) principle. Aspect-oriented programming (AOP) has been proposed as a new technology that adopts the SOC principle with its six "C" properties: Concern-oriented, Canonicality, Composability, Computability, Closure, and Certifiability [1]. It grew out in the early 1990s with the aim to manage crosscutting concerns such as logging, security, and distribution which scattered in multiple software functional core modules. In addition to code scattering, when a module handles multiple concerns simultaneously, code tangling arises to decrease systems maintainability and increase their complexity.

The AOP does not introduce a completely new design process but just a new means to enhance design. As procedural programming brought functional abstraction and object-oriented programming gave birth to object abstraction, aspect-oriented programming introduces concern abstraction [2]. There are four main AOP programming approaches: Xerox PARC AOP, subject-oriented programming (SOP), adaptive programming (AP), and composition filters (CF) approach [3]. Xerox PARC AOP is the most popular method [4]. It is often called aspect-oriented programming and has AspectJ as an implementation language.

In our study, we have used Xerox PARC approach, namely, AOP, and our work focuses on AspectJ the most mature and the widely used in research area, since it is the first practical AOP implementation based on Java. It enhances Java with additional structures to manage crosscutting concerns as pointcuts, advices, static crosscutting structures, and aspects [4].

Simulation modeling software requires more and more features in order to cope with emerging problems. They

could include many crosscutting concerns. As an instance, there is synchronization, where many simulation tools implement concurrency trying to provide a more direct way of modeling real systems, scheduling policies, optimizations because performance is a desired property in simulation, and distribution (e.g., using web-based simulation) [5]. Thus, the applications of aspect-based modeling span a broad range of simulation software. Applications range from discrete event simulation (DES) frameworks as Tortuga [6], Simkit [7], SimJ [8], and the component-based instrumentation framework OSIF [9] that uses the AOP paradigm to separate its DES models from the experimental frame to enable software reuse and evolution to multiagent simulation as [10, 11] systems. Moreover, AOP has been used for developing large simulation software systems as in the disaster prevention simulation system [12] and the conduit and traffic simulators [13]. Furthermore, several approaches have been proposed to separate the performance concern from the functional core of software applications as presented in [14–16].

The aim of this paper is to illustrate key concepts of aspect-oriented approaches to identify the main crosscutting concerns that may be implemented in simulation and to discuss the aspect-oriented version of the Japrosim framework [17]. The use of the AO simulation is justified by the fact that the domain requirements for simulation software are complex and the object-oriented paradigm is unable to deal with such complexity.

The rest of the paper is organized into six sections. Section 2 presents the main goal behind the use of the aspect-oriented paradigm in simulation. Section 3 outlines most of the works in aspect-oriented simulation. Section 4 casts light on the AOP approaches. Section 5 presents the main crosscutting concerns that may infect simulation systems and the general staff for implementing them. In Section 6, a case study of applying the AOP paradigm to Japrosim is presented. Finally, a conclusion is given to open doors for future research.

2. Problem Statement and Contribution

Although the object-oriented simulation provides a rich and intuitive paradigm for building models of real-world systems, it suffers from the inherent weakness of object-oriented programming (OOP) methodology which is the inability to modularize all concerns. The limitation of the OOP approach is its inability to localize concerns that do not naturally fit into a single program module or even several closely related modules. Such concerns are called crosscutting concerns as they crosscut or span implementation modules.

OOP code suffers from two phenomena resulting from misalignment between requirements and code: tangling and scattering. Tangling occurs when multiple concerns are addressed in a single module making the module harder to understand and maintain. Scattering results occur when the implementation of a concern is spread over multiple modules leading to the risk of inconsistencies at each point of use. The AOP paradigm puts a greater focus on crosscutting concerns than OOP and other language paradigms. It provides language mechanisms that explicitly capture crosscutting concerns in a modular way and thus achieving the benefits that result from improved modularity. These benefits lay in making code easier to design, implement, maintain, reuse, and evolve. The AOP takes crosscutting concerns out of functional modules and place them in a separated location called aspect [20].

Besides that, simulation systems comprise concerns that not necessarily align with the functional components as synchronization, scheduling, optimizations, and distribution concerns. The way in which these factors affect software artifacts usually produces a messed design. The tangling code increases the dependencies between the functional components, thus making the source code difficult to develop, understand, and evolve. As a result, simulation system properties such as reusability and adaptability become fairly restricted. In this context, simulation domains need specific separation of concerns.

Figure 1 presents a motivation example for the simulation of a thermic control system which has a building with rooms requiring specific temperatures and a network consisting of heaters, pipes, and a boiler. The boiler is the heat source, and it generates a heat flow that is distributed through the circuit. Each heater has a valve, with a regulation mechanism, to transfer part of this heat flow to the rooms until they reach the selected temperatures. Each room has predefined dimensions, materials, and convection properties. The heaters store information about size, radiation area, and convection properties. The user can set up the desired temperature for a room, and when the current temperature in the room is lower than this value, a control action is activated asking a certain heat amount to the boiler.

To simulate this thermic control system, a mathematical model is required for describing the heat flow between the different components. The mathematical model adds new relationships between the components of the object model: each heater needs to know the temperature of its associated room, each room knows the temperatures of its neighbors, and the boiler is able to set heater valves during its computation. In addition, to specify how the simulation run, a natural modeling comprises the implementation of the components as independent threads are adopted. This situation implies taking synchronization and scheduling concerns into account, for example, to synchronize communication between heaters and rooms regarding to heat transference or valve manipulations in the heaters because race conditions might arise if the boiler try to use them. Synchronization code is scattered across several classes. The latter depends on the different relationships among variables defined by the mathematical model. Algorithm 1 shows how aspects related to mathematical models, concurrency, scheduling, and optimizations are scattered across the architecture. This is a simplified example and it is not difficult to deal with the relatively small amount of tangling existent in the code. But in real programs the complexity due to such tangling quickly expands to become a major obstacle regarding to maintenance and reusability [5].

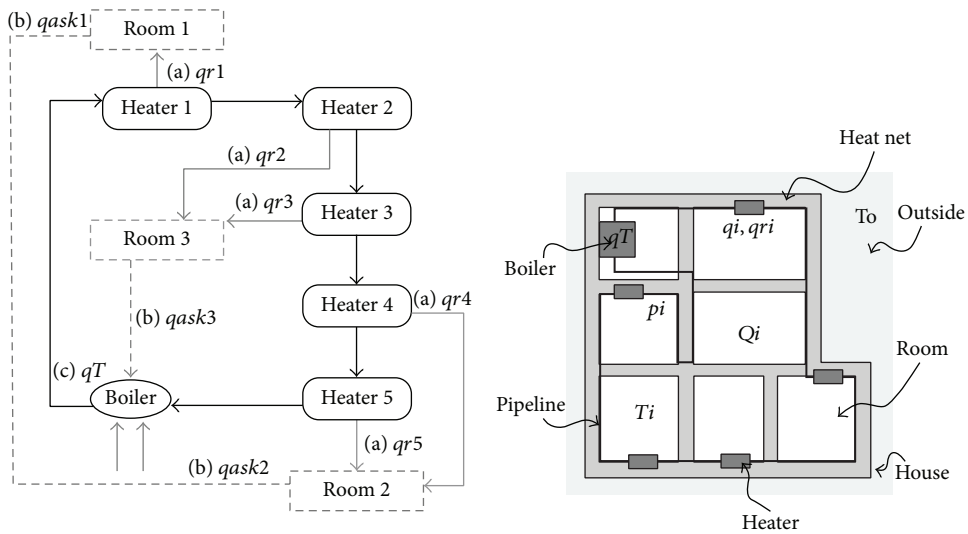
Discrete event simulators implement a number of concerns, such as event scheduling, event handling, and keeping

```
extends HeatPropagator {

// instance variables -
private double qT;
private double qrooms;
private Vector heaters;

private object qroomslook = new object[0];

// Instance methods --
public void requestHeat(double q,vector h) {
   synchronized (qroomsLock) {
      qrooms = qrooms + q;
      ...
      heaters.addAll(h);
   }
}
protected propagate() {
   qT - this.generateHeat();
   next.addHeat(qT);
}
protected double transferredHeat() {
   synchronized (qlock) {
   ...
   }
}
```

```
protect double generateHeat() {
   synchronized (qroomslook) {
      vector valves;
      ...
      for (Enumeration e - heaters.elements();
          e.hasMoreElements(); ) {
         Heater heater - (Heater) (e.nextElement());
         ...
         valves.add(h.getValve());
      }
      ...
      double q;
      q = MathPackage.RungeKutta(qi, qrooms, qT, ...);
      ...
      for (Enumeration e = heaters.elements ();
          e.hasMoreElements(); ) {
         Heater heater - (Heater) (e.nextElement());
         ...
         h.setValve(valves.elementAt(1));
      }
      return (q);
   }
}
} // End Boilar
```

Bold code corresponds to tangled code due to concurrency aspects.
Underlined code corresponds to tangled code due to mathematical aspects.

Algorithm 1: Resulting boiler class (with tangled code) [5].



Ti: current temperature in room *i*
To: external temperature
Qi: heat value in room *i*
qi: heat value in heater *i*
qri: heat transferred from the heater *i* to the room
qT: total heat produced in boiler
pi: threshold valve in heater *i*

Figure 1: Diagram of the thermic control system [5].

track of a simulation's state which crosscut over multiple modules in the system. This increases the complexity and reduces the maintainability. The separation of systems' crosscutting concerns improves their quality attributes as modularity, understandability, maintainability, reusability, and testability [21]. This paper exploits the aspect-oriented on simulation modeling domain through:

(i) giving an overview of aspect-oriented programming (AOP) approaches as a comparative study;

(ii) identifying the most crosscutting concerns of simulation systems and their general implementation process;

(iii) proposing the aspect-oriented (AO) full version of Japrosim library;

(iv) evaluating the AOP Japrosim packages using AOMetrics tool.

3. Literature Review

The use of the AOP in the discrete event simulation domain depends on several considerations related to the aspect-oriented paradigm itself as the level of its application (specification, design, or implementation phase), the coexist methodology (multiagent, components, or object oriented), and the used aspect-oriented language (e.g., AspecJ, AspectS, or MAML). Besides that, the aspect-oriented simulation systems could be classified from a simulation point of view, that is, the kind of the separated crosscutting concerns. The latter are specific for simulation modeling domain as steady state detection or they may be found in any application as exception handling. In the following, a summary of most of the existing works in the literature is provided.

Simkit is an open source tool based on the OO paradigm and event graph formalism for modeling. The authors in [7] proposed its AO versions in terms of AspectJ solutions for separating the simulation termination rules, restoring a simulation run, and resource pooling crosscutting concerns. However, they miss to separate all crosscutting concerns, especially those which could be found in any mature framework as steady state detection and graphical user interface. In [9], the component-based instrumentation framework OSIF uses the AOP paradigm to separate its DES models from the experimental frame to enable software reuse and evolution. Moreover, the AOP has been used in the development of the Open Simulation Architecture (OSA) [22] for the same purpose as in OSIF. It allows better reuse of components in both sides, reuse of a given model with various scenarios, or reuse of a given scenario with various models in order to save time, money, and human effort. A simple use case of network security study has been used to illustrate the benefits of the previous technique.

SimJ framework is an academic framework containing only the logging crosscutting concern. The authors in [8] confirmed their hypothesis in which AO GoF 20 design patterns decrease the code complexity, eliminate scattering code, and allow to design additional AO hot spots in frameworks and the loss of performance is minimal and fully acceptable by means of SimJ case study where the AO Adapter design pattern is used. The authors view SimJ as an application framework without giving any importance to its discrete event simulation domain. Next to this, in [6] the authors introduced their Java-based Tortuga simulation framework, which used AO to ensure the synchronization aspect only.

In [10], a multiagent simulation system is discussed, and it consists of two types of agents, a set for describing the simulation model and another for observational mechanisms. This collection of independent agents is interacting by discrete events where every agent has a schedule that generates its plan of activities. The system is executed on a framework that uses the OO paradigm to define its agent models and web technology to interact with the modeling and simulation environment. The kernel of this framework is the programming language MAML (Multiagent Modeling Language) which has the capacity of the dissociation between the model and the observational mechanisms thanks to the aspect-oriented paradigm from the design phases until the implementation phase, thereby increasing the maintainability of the system and decreasing its complexity. MAML has the xmc compiler which generates Objective-C code from the MAML source code after weaving the model object and the observation object. Despite the richness of MAML syntax to support AOP, it remains in its infancy when compared to AspectJ.

In [11], the AO paradigm has been used to develop a multiagent system dedicated to simulate physical phenomena. The MAFES system (Multiagent Finite Environment System) consists of an environment in the form of a node matrix and a set of agents operating on these nodes. Aspects are used to assign tasks to agents by adding appropriate functionality to perform their task. In addition, these aspects weave the appropriate resources and attributes to the environments nodes. MAFES contains three other types of aspects for control, visualization, and storing simulation results. The implementation of MAFES is based on AspectJ and makes the system generic to build versions for specific requirements (it is enough to weave appropriate aspects). The authors experiment their system with two phenomena. The first is heat exchange and a motion phenomenon. The second is heat exchange and crystallization.

In [12], a new aspect-oriented approach for disaster prevention simulation system (ABR) has been addressed. The proposed approach separates the core functionality of the simulation application from simulation crosscutting concerns thanks to horizontal decomposition (HD) method which relies on the AOP paradigm. The approach is implemented on AoSiF (Aspect-Oriented Simulation Framework) which is an extension of distributed simulation framework (DiSiF) [23]. It uses the resource paradigm, actor-based workflow modeling, web services, and Grid computing as implementation technology and Java Annotations for declarative programming in addition to AspectJ for the aspect-oriented implementation. To demonstrate the applicability of the approach, two crosscutting concerns, namely, distribution and tool integration, are implemented. Unfortunately, concerns are not specific to the simulation modeling domain.

In [13], the authors implemented a conduit simulator system which uses the AOP paradigm at the code level. The

simulator has crosscutting concerns as synchronization and order of execution, user interface (UI), and logging which are written in different aspect-specific languages (ASLs). It has a modular aspect-weaver mechanism that offers the generality of a general-purpose aspect language without losing the ability and advantages of defining aspects in aspect-specific languages.

In [24], an AO real-time traffic simulator has been developed. It uses AOP to encapsulate seven real-time crosscutting concerns using AspectJ: thread scheduling and dispatching, synchronization and resource sharing, asynchronous thread termination, memory management, physical memory access, asynchronous event handling, and asynchronous transfer of control. A comparison of the two systems, a real-time sentient traffic simulator and its aspect-oriented equivalent, is performed to indicate both benefits and drawbacks with AO approach. It is based on OO Chidamber and Kemerer (C\&K) metric suite. The aforementioned concerns are general, they could be found in any real-time systems. Moreover, despite the fact that C\&K metric suite is adapted to AO systems, further investigation is required into the metrics that are suitable for evaluating AO applications.

In [25], a simulation based design level performance analysis method has been proposed where the performance concern of the software application is separated from the functionality model since the design phase by the use of aspect-oriented programming. Unlike the design of the software system which is modeled using UML class diagrams and sequence diagrams, the performance model is an XML-based representation derived from the UML performance profile. After code generation from the design model, the AspectJ weaver is introduced in order to formulate the simulation code. The authors experiment their approach using a distributed map viewer system. In [14], they argue that their approach is generic and could be used for analysis of other quality attributes of software systems as reliability. In [26], another approach for analyzing performance is proposed. Unlike the precedent approaches, it defines performance as a collection of aspects which includes a long list of metrics such as response time, rate throughput, probability, and time between errors over aspect-oriented formal design analysis framework (FDAF). The authors focus on modeling the response time performance aspect and they have chosen real-time UML as the base notation which is translated into Architecture Description Language (ADL) Rapide. The authors use the simulation technique to evaluate response time for the DNS query processing subsystem.

In [15], the authors propose an aspect-oriented framework for agent-oriented software that separates the performance concern at the design level using the aSideML language, which is a UML extension for representing aspects at different levels of abstraction. This framework provides separation of performance concerns among the different agenthood properties (mobility, autonomy, adaptability, interaction, and learning) and scenario specific application concerns. The design model for scenario specific application concerns and agent-hood concerns has its own Java implementation code while the performance model has a separate AspectJ implementation. Later, these are all woven together using the AspectJ compiler. The framework architecture is composed of performance component, agent concerns, the agent platform component, workload and resources concerns, IInformatioGathering interface, and IperformanceReporting interface.

In [16], the authors propose a new approach that studies the performance effects of crosscutting aspects as security on the overall system performance. The approach proceeds by adding performance annotations to both the primary and aspect models using the UML performance profile and then instantiating the generic aspect model into a context-specific one by following a set of binding rules provided by the designer who transforms the parametric annotations of the generic aspect model into concrete ones. The latter is composed with the primary model according to a set of composition directives. The result is a composed annotated UML which can be transformed automatically into a performance model (Layered Queueing Networks (LQN) in this case) by using the transformation techniques. The LQN model is analyzed with existing solvers.

In [27], the authors discuss a new approach for separation of functional (qualitative) behavior and quantitative performance constraints since the specification phase. Thanks to AOP, the aspects of a specification are written in different languages: the process algebra LOTOS for an abstract specification of functional behavior and the probabilistic temporal logic for quantitative aspects (performance constraints). The aspect weaving composes the two aspect specifications and the result of this composition is an automata-style global model which can be generated from the composition of a labeled transition system (derived from LOTOS). Event schedulers are derived from temporal logic formulae. Finally, the global model can be used for performance analysis-based simulation.

This work is similar to these proposals in terms of harnessing the AOP paradigm for refactoring a discrete event simulation framework. In our case, it is Japrosim. From another side, this paper identifies the main simulation crosscutting concerns as steady state detection and graphical animation, which do not exist in the previous works and evaluate their impact on simulation systems design quality properties.

Moreover, in order to benefit from the advanced separation of concerns techniques such as subject-oriented programming, adaptive programming, composition filter, and AOP in the simulation modeling domain, a comparative study is presented, to clarify their principle constructs and the relation between them.

4. The Aspect-Oriented Approaches

Aspect-oriented programming paradigm aims to manage crosscutting concerns, that is, concerns that span across multiple modules leading to scattering and tangling code. It has two main features, quantification and obliviousness. Filman and Friedman in [28] say this "AOP can be understood as the desire to make quantified statements about the behavior of programs and to have these quantifications hold over

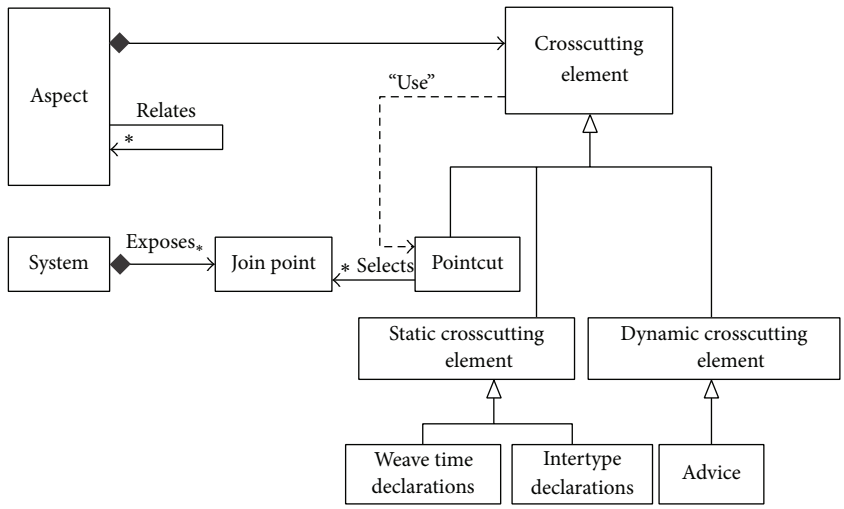


Figure 2: Generic model of an AOP system [4].

programs written by oblivious programmers." A closer look at four main AOP approaches is provided in the following.

4.1. Xerox PARC Aspect-Oriented Programming. Aspect-oriented programming grew out of Xerox Palo Alto Research Centre (PARC). It separates crosscutting concerns into new modularization units which are called aspects [29]. It has AspectJ as an implementation language which is an extension to Java, but there are other implementations languages like AspectC for C and Pythius for Python. To implement crosscutting concerns, an AOP system may include many constructs where the central one is the join point model which consists of two parts: join points that are points in the execution of an application and pointcuts which are a mechanism for selecting join points. The aspects have advices that are attached to one or more join points to inject the crosscutting concern. When an advice is attached to join points, it will be executed. Besides that, it has a modifier which may specify the execution time relative to the join points: before, after, around, after exception, or also after return value. Advice is a dynamic crosscutting element because it affects the execution of the system. Furthermore, AOP implementation may contain static crosscutting elements which alter static structure of the system as intertype and the weave-time declaration constructs. Figure 2 shows all these concepts and their relationships to each other in an AOP system. Each AOP system may implement a subset of the model.

4.2. Subject-Oriented Programming. Subject-oriented programming paradigm was introduced in 1993 at IBM Thomas J. Watson Research Centre. It is an extension of object-oriented programming that supports building systems with different subjective perspectives. Subject-oriented programming philosophy is based on two central ideas: the division of the system on several subjects and the composition rules. A subject is a complete or partial object model. It is a collection of related classes or a fragment of classes for particular purpose as shown in Figure 3 with shipping and transportation subjects. A subject could be complete or just parts of an application. These different subjects could define and operate upon shared objects in independent manner [18].

Composition rules are applied to compose different subjects together. The composition creates a new subject that merges the functionality of the existing subjects. There are three kinds of rules: correspondence rules, combination rules, and correspondence-and-combination rules. Correspondence rules define correspondences between the classes, methods, and members of different subjects. Combination rules specify how corresponding and noncorresponding classes, methods, and members of different subjects are composed together to form a composite subject. Correspondence-and-combination rules can do both simultaneously [30]. Each subject is compiled separately to produce a binary subject. The binary subject consists of a label providing information about it and a binary code produced by the compiler. The subject-oriented compositor uses information in the labels to tie the subjects together. It does not examine or modify the individual subjects' binary code.

The overall goal of subject-oriented programming is to facilitate the development and evolution of suites of cooperating applications. It supports decentralization in time as well as in space. It can also add extensions to an existing system in a noninvasive way. From its supports, "C++ Subjectifier" and the "Binary Subject Compositor" tools are built for C++ [31].

4.3. Adaptive Programming. Adaptive programming was introduced around 1991 by Demeter group at Northeastern University in Boston. The group has used the ideas of AOP several years before the name aspect-oriented programming was coined. After the collaboration with the Xerox PARC group had begun, the group redefined AP and the term AOP was introduced. AP is a kind of concern-shy programming. A program is concern-shy if it hides the details of a certain concerns it cuts across, and then it adapts automatically. The most

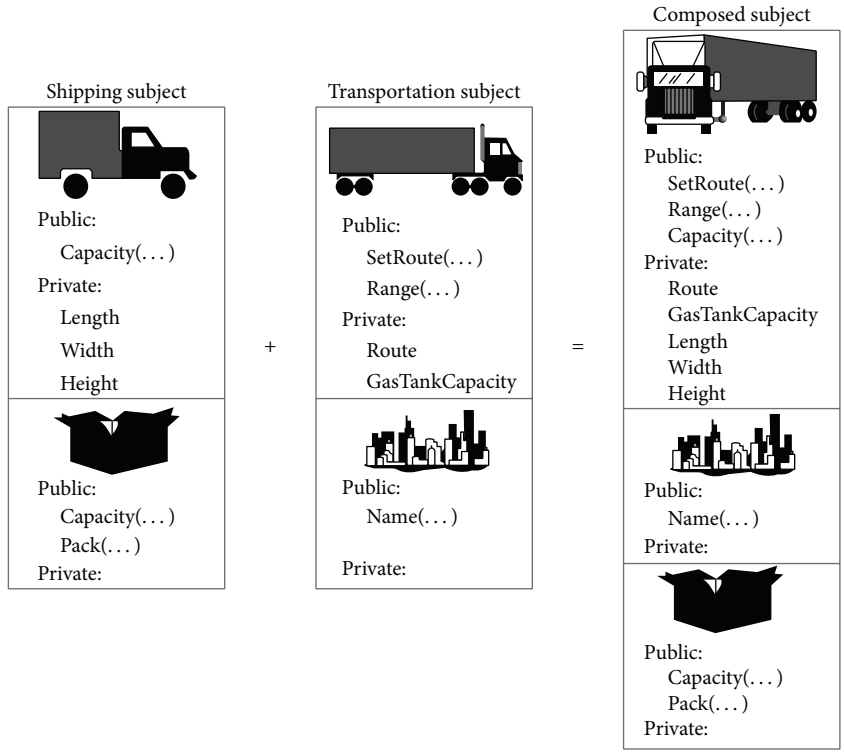


Figure 3: Application suite formed by composition of shipping and transportation subjects [18].

used form of AP is structure-shy programming. Structure-shy programming eliminates the concern structural details by making its behavior adaptive; that is, the behavior and the structure of the program are separated. The AP approach applies the Law of Demeter which argues that an object should only talk to its immediate friends by handling traversal strategy and visitor pattern concepts. There are various tools for developing adaptive software as DJ-tool, DemerterJ, and DAJ [32].

4.4. Composition Filters. Composition filter is an aspect-oriented programming approach in which the CF model distinguishes two kinds of constructs: (class-like) concerns and filters. Briefly, a concern is the unit for defining the primary behavior, while filters are used to extend or enhance concerns so that (crosscutting) propriety can be represented more effectively [19]. A message sent to an object is evaluated and manipulated by the filters of that object which are defined in an ordered set until it is discarded or dispatched to another object as shown in Figure 4. Filter behavior is simple: each filter can either accept or reject the received message, but the semantics of the operations depends on the filter type. For example, if an error filter accepts the received message, it is forwarded to the next filter, but if it was a dispatch filter, the message would be executed [2]. JAC (Java Aspect Component) framework and Sina language are implementations of this approach.

The AOP is the mature and the most used approach [4]. IBM members considered it as offspring of SOP by identifying and illustrating several useful nonfunctional concerns to be separated, such as concurrency properties, distribution properties, and persistence. AOP distinguishes the notion of "core classes" which encapsulates a system's functional requirements from "aspects" which encapsulate nonfunctional, crosscutting requirements. Aspects are written with respect to core classes and each aspect contains its part of the rule specifying how that aspect is to be woven into the base classes [33]. The AOP approach is used widely in simulation field in order to increase simulation system's reusability and developing large simulation software with less complexity as presented in [20]. In order to benefit from the advanced separation of concerns techniques in the simulation modeling domain, a comparative study is presented in Table 1, which clarifies their principle constructs and the relation between them.

5. Materials and Methods

5.1. Main Crosscutting Concerns of Simulation Systems. There are several nonfunctional concerns that may span simulation applications functional modules as follows.

5.1.1. Steady State Detection Crosscutting Concern. Steady state detection concern detects the system stabilization where the output data collected during the warming-up period of a simulation can be misleading and bias the estimated response measure. Thus, the removal of initialization bias is important for obtaining accurate performance estimators. There are five categories of methods for steady state detection:

Table 1: Comparison between AOP approaches.

	Xerox PARC AOP	SOP	AP	CF
Atomic unit	Aspect	Subject	Propagation pattern	Filter
Coexisting methodology	Object-oriented/components/mutiagents	Object-oriented	Object-oriented	Object-oriented
Composition mechanism	Weaving mechanism	Composition rules	Weaving mechanism	Superimposition mechanism
The mechanism used to determine crosscutting concerns injection places	Pointcuts specification	Labels specification	Traversal strategies specifications	Matching part of filters
The places of the crosscutting concerns injection	Join points	Subject parts	Traversal object model	The implementation part of the CF object after messages intercepting
Implementation languages	AspectJ	IBM VisualAge C++	DemeterJ	ComposeJ
Composition time	Compile time, load time, or run time	During the compilation	Compile time or run time	Compile time or run time

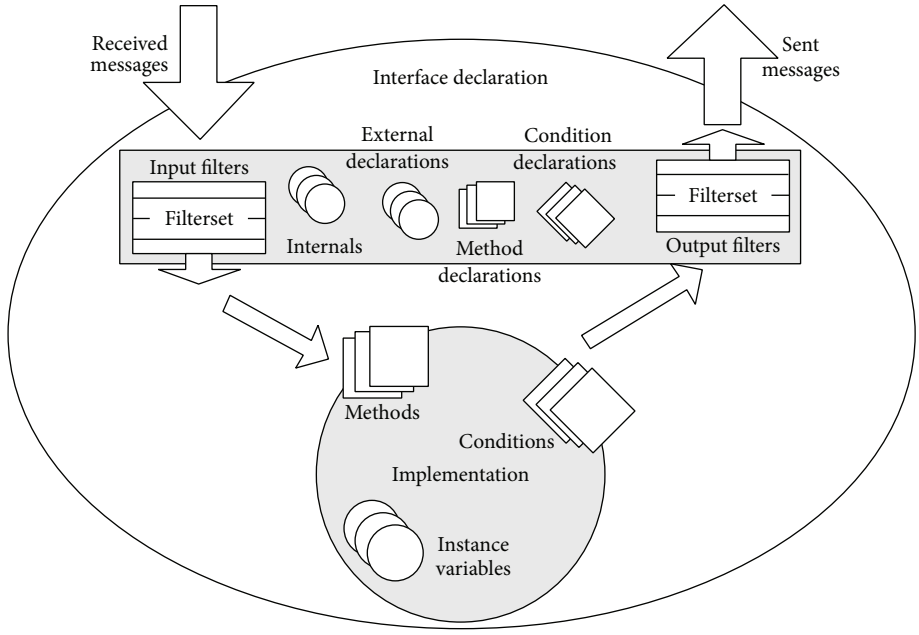


Figure 4: Composition filters object [19].

graphical, heuristic, statistical, initialization bias tests, and hybrid methods [34].

5.1.2. Graphical Animation. Graphical animation is a way of visualizing the system behavior during experimentation. This visualization is of use in validating a design model and interpreting the meaning and significance of simulation results. Further, it helps users to better understand the dynamics of the system under study and useful for teaching.

5.1.3. Graphical User Interface Crosscutting Concern. The GUI is used for presenting simulation statistics and increasing the interaction with users. Its code may crosscut various functional modules making GUIs hard to maintain.

5.1.4. Exception Handling. Exception handling is a nonfunctional requirement for any application to gracefully handle any erroneous condition like resource unavailability, invalid input, null input, and so on. Thereby, the solutions proposed for exceptions handling pollute simulation systems code and make it incoherent.

5.1.5. Calculation Accuracy. Unfortunately, during the operations of addition, multiplication, and division, overflow or underflow exceptions do not occur. This is the case of most programming languages, including Java where its arithmetic operators do not report overflow and underflow conditions. They simply override them. Hence, such a crosscutting concern will ensure the accuracy of calculation results, which is crucial in simulation systems.

5.1.6. Simulation Trace. In simulation systems, it is important to register the state of all resources, passive and active entities in separate time sequence. These registrations pollute the simulation functional code in crosscutting manner.

5.1.7. Synchronization. For process interaction modeling worldview, largely adopted by the discrete event simulation community, the model consists of a collection of interacting processes. Each process models the life cycle of a system's object, namely, a well ordered sequence of activities which are logically related. Further, mutual exclusion synchronization restricts concurrent activities in critical sections to protect them against data inconsistency due to simultaneous access for writing. For example, in Java multithread programming, an object could be accessed by many threads simultaneously. In consequence, data conflicts could occur if applications are not prepared to deal with concurrency. Thus, the synchronization should be implemented by using the synchronized modifier at the method level or the synchronized (object) construct at the instruction or block level [35]. Thus, the synchronization of simulation processes and the mutual exclusion synchronization are a crosscutting concern in simulation function code.

5.2. The Process of Implementing an Aspect-Oriented Simulation System. Generally, the process of developing an aspect-oriented simulator contains three phases, as shown in Figure 5, the identification of systems' concerns, their implementation, and the development of the final system by combining them in the following way.

(1) First, the crosscutting concerns as synchronization and steady state detection are separated from simulation engine concerns. This phase is compared with the passage of a beam of light through a prism to separate its different color components.

(2) Next to this, each crosscutting concern is implemented independently by using an aspect-oriented language as AspectJ which reduces the overall complexity of design and implementation. In addition to that, a procedural or OOP languages are used for the implementation of simulation functional concerns.

(3) Finally, an aspect weaver as AspectJ compiler is used to compose all crosscutting concerns and simulation functional concerns to produce the final system.

6. Experimental Work

6.1. Japrosim Library. Japrosim is a free and open source object oriented simulation library that adopts the widespread process interaction worldview. The library is implemented in Java programming language allowing profound access to its powerful features and is documented using the UML. Its classes are organized into packages that reflect important functional areas. Either experimented programmers in Java or simulation experts with elementary programming knowledge could effortlessly build discrete event simulation models using JAPROSIM. The library is extensible and customizable

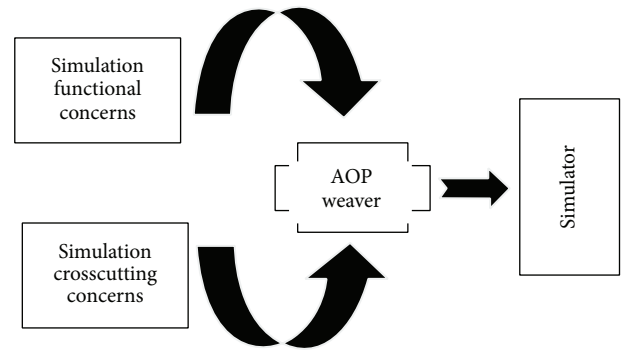


Figure 5: The process of implementing an aspect-oriented simulation system.

which allow it to serve as a basis for the development of domain specific simulation environments. Furthermore, it is used as an academic material for teaching discrete event modeling and simulation. The key feature of Japrosim is the implicit and automatic collection of all well-known and useful performance measures [17].

The library is divided into eight main packages:

(i) kernel: it is a set of classes dealing with active entities, scheduler, queues, and resources as shown in Figure 6;

(ii) random: it contains classes for uniform random stream generation;

(iii) random.distributions: it contains a rich set of classes for useful probability distributions;

(iv) statistics: it contains classes representing intelligent statistical variables;

(v) gui: it is a set of graphical user interface classes to use for project parameterization, trace, and simulation results presentation;

(vi) utilities: it is a set of useful classes for express model development;

(vii) statistics.steady: it is useful to detect the steady state;

(viii) animation: it is a set of classes used to provide a real time animation of simulation models.

6.2. The Proposed AOP Solutions. The main crosscutting concerns, discussed in the precedent section, are identified and separated in Japrosim using AJDT tool [36]. The aspect-oriented (AO) version of Japrosim is enhanced with the uoeb.Japrosim.crosscutting.concerns package which consists of eight aspects: SingletonConcern, Animation, ExceptionHandling, GraphicalUserInterface, SimulationTrace, SteadyStateDetection, Synchronization, and CalculationAccuracy. Thus, every aspect gives a solution to a separate crosscutting concern which pollutes the framework packages as shown in Figure 7.

6.2.1. SingletonConcern Aspect. The "Singleton" pattern prevents the use of two objects for the same class. The Scheduler

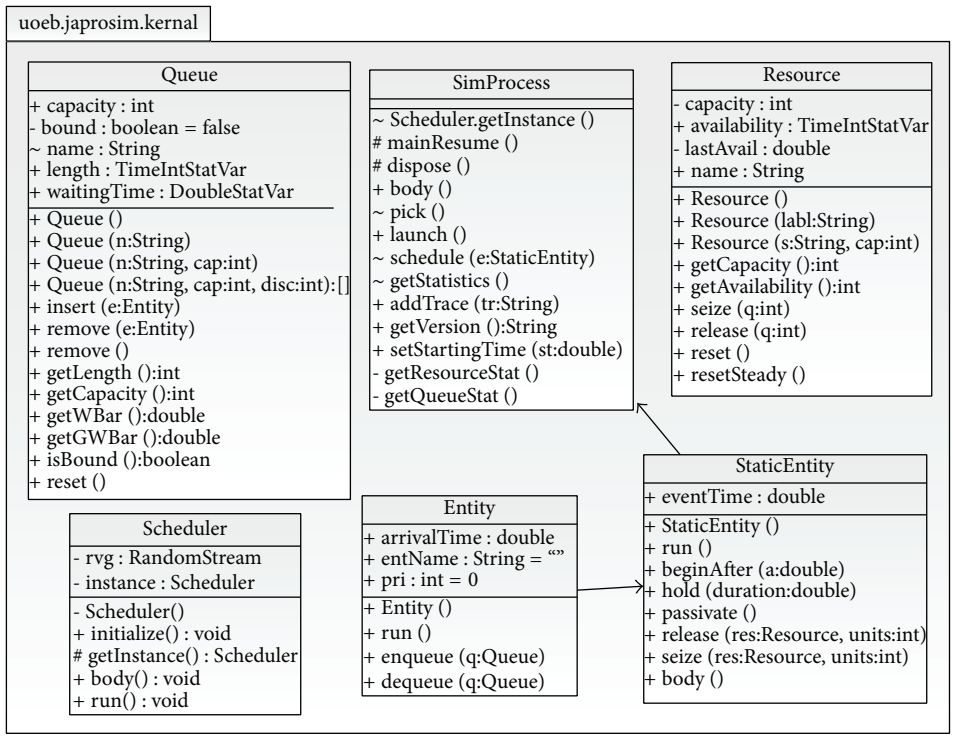


Figure 6: The kernel package.

Figure 7: The interaction between aspects and AO Japrosim packages.

class uses it to prevent multiple instances and ensure that the event list management is done exclusively by a unique thread. In the old object-oriented (OO) version, Japrosim ensures that only one Scheduler instance may exist by declaring its constructor as private, providing a public method, namely, getInstance(), to return the single existing instance and saving the single existing instance as a static member variable. The SingletonConcern aspect is now proposed as a solution which includes an around advice that applies at the moment of the constructor call that has the similar role of getInstance() method without the need to declare the singleton constructor as private. Furthermore, the Scheduler instance is saved in a static variable declared inside the aspect as a member introduction.

6.2.2. Animation Aspect. The graphical animation, inside OO Japrosim, is provided by means of the "observer" pattern. This pattern creates a relationship between a subject and an observer. An object is called a subject when its changes are interesting for other objects, namely, observers. Observer pattern elements are scattered inside the domain classes, which decreases their cohesion and maintainability; thus, AOP proposes the encapsulation of these observer elements in an aspect and the domain classes remain without infection. The SimAnim class is part of the animation package used to provide a real-time animation. It gets useful data by means of the "EventObserver" interface. Moreover, each of the "Queue," "Resource," and "StaticEntity" classes register listeners of "EventObserver" type to inform them in case of event occurrence. This mechanism is ensured by the observer pattern. The animation aspect is proposed as a solution. It separates these elements by providing the link between subject and observer using the "EventObserver" as inner interface in addition to member introduction and type-hierarchy modification which affected subjects and SimAnim class, respectively.

6.2.3. SteadyStateDetection. The SteadyStateDetection aspect is proposed to solve the problem of the Japrosim classic version which offers two methods that are Conway and Crossing the Means by means of factory and observer design patterns. The observer elements are tangled in the Entity, SteadyStateTechnique, Conway, and CrossingTheMean classes. The SteadyStateDetection aspect implements the AO version of

observer pattern to improve the modularity of Japrosim library.

6.2.4. Synchronization. For process synchronization, Japrosim implements the coroutine mechanism through SimProcess, Scheduler, StaticEntity, and Entity classes. A collection of threads run in quasiparallel mode under the Scheduler thread control. Each coroutine is an object with its own execution state, so that it may be suspended and resumed. At any instance of real-time, only one coroutine is active. The method processResume (Entity e) is called by the scheduler to reactivate a simulation process, and mainResume() is called by a simulation process to reactivate the scheduler. Each simulation process has its own lock object while the scheduler has the mainLock object. Locks are used in combination with wait() and notify() to synchronize the implementation threads. A thread which calls any of the previous methods will block on its own lock after notifying the appropriate one. At the end of its life cycle, a simulation process calls automatically the dispose() method to reactivate the scheduler without blocking itself. So the corresponding thread could be terminated [36]. The elements that ensure the coroutine mechanism are the processResume (Entity e), mainResume(), and dispose() methods in addition to the mainLock and lock objects. These elements are, respectively, separated in a single synchronization aspect that clears the design and increases understandability.

From another side, OO Japrosim methods which have a critical property are enhanced with the synchronized keyword as the getInstance() method in Scheduler class. This tends to pollute the Japrosim functional code. In order to overcome this problem, an around advice inside the synchronization aspect is developed to ensure methods synchronization by using a shared aspect lock.

6.2.5. GraphicalUserInterface. The Japrosim gui package includes a set of classes (TraceFrame, PresentationFrame, MainFrame, and GraphicFrame) used for project parameterization like the number of replications, experiment duration, trace, and simulation results presentation. Despite the benefits and flexibility provided by the GUI concern, its crosscutting nature leads to code pollution. It decreases the cohesion specifically inside Japrosim kernel classes. As an AOP solution, the GraphicalUserInterface aspect is proposed.

6.2.6. SimulationTrace. This aspect saves simulation trace, separately from Japrosim functional modules, through the generation of three files. One is in text format and the two others are conforming to XML format.

6.2.7. ExceptionHandling. The ExceptionHandling aspect contains five pointcuts and five advices that deal with all exceptions types inside Japrosim library.

6.2.8. CalculationAccuracy. This aspect handles the underflow and overflow of int, float, double, and long primitive types, over addition, subtraction, multiplication, and division arithmetic operators, as shown in Algorithm 2.

6.3. Result and Discussion. To measure the impact of applying the AOP paradigm to Japrosim design quality as flexibility, maintainability, and reusability, an automatic assessment of software metrics is achieved for AOP Japrosim version using the AOPMetrics tool [37] which is a common metrics tool for both java and AspectJ programs. It provides package dependencies metrics suite that is based on Robert Martin's suite proposed in [38].

(i) *Number of Types (NOT)*. It is a number of types within given package. This is an indicator of the extensibility of the package.

(ii) *Abstractness (A)*. It is the ratio of the number of abstract modules to the total number of modules in the package.

(iii) *Afferent Couplings (Ca)*. It is the number of modules outside the package that depend upon modules within the package. This is an indicator of the package's responsibility.

(iv) *Efferent Couplings (Ce)*. It is the number of modules inside the package that depend upon modules outside the package. This is an indicator of the package's independence.

(v) *Instability (I)*. I is the ratio of efferent coupling (Ce) to total coupling (Ce + Ca) such that I = Ce/(Ce + Ca). This metric is an indicator of the package's resilience to change. A value of zero indicates a completely stable package and a value of one indicates a completely instable package.

(vi) *Normalized Distance from Main Sequence (Dn)*. It is the normalized perpendicular distance of the package from the idealized line $A + I = 1$. This metric is an indicator of the package's balance between abstractness and stability.

Table 2 shows the results obtained from the measurement of these metrics for AOP Japrosim packages. The crosscutting package is characterized by the high (Ce) metric value because it implements the all crosscutting concerns of other packages and the null value of (Ca) which expresses the obliviousness of the AOP paradigm. The crosscutting package is completely unstable which confirms its crosscutting nature without any functional concern.

The difference between all package dependencies metrics values for both OO Japrosim and AO versions is registered as presented in Figure 8. A remarkable change roughly 62% in Japrosim packages is obtained except for the uoeb.Japrosim.gui, uoeb.Japrosim.random.distributions, and uoeb.Japrosim.utilities which are not affected because they have the minimal interaction with other packages in OO Japrosim version. Furthermore, the AO Japrosim version offered total decrease in (I) metric value after aspects implementation because the packages are flexible enough to be extended without requiring modification. The "Open/Closed" principle [38] is implemented in ideal way through

Table 2: The measurement results for AO Japrosim packages using AOPMetrics tool.

Package name	NOT	*A*	RMartin Ce	RMartin Ca	RMartin *I*	RMartin *D*	Ce	Ca	*I*	Dn
uoeb.japrosim.crosscutting.concerns	10	0.2	9	0	1	0.2	43	0	1	0.2
uoeb.japrosim.kemel	7	0.285714	5	42	0.106383	0.607903	7	42	0.142857	0.571429
uoeb.japrosim.statistics.steady	4	0.25	3	2	0.6	0.15	2	2	0.5	0.25
uoeb.japrosim.gui	9	0	3	2	0.6	0.4	2	2	0.5	0.5
uoeb.japrosim.random.distributions	19	0.052632	18	2	0.9	0.047368	2	2	0.5	0.447368
uoeb.japrosim.animation	8	0	2	3	0.4	0.6	1	3	0.25	0.75
uoeb.japrosim.statistics	2	0	2	11	0.153846	0.846154	1	11	0.083333	0.916667
uoeb.japrosim.utilities	3	0	3	0	1	0	5	0	1	0
uoeb.japrosim.random	6	0.166667	5	2	0.714286	0.119048	1	2	0.333333	0.5

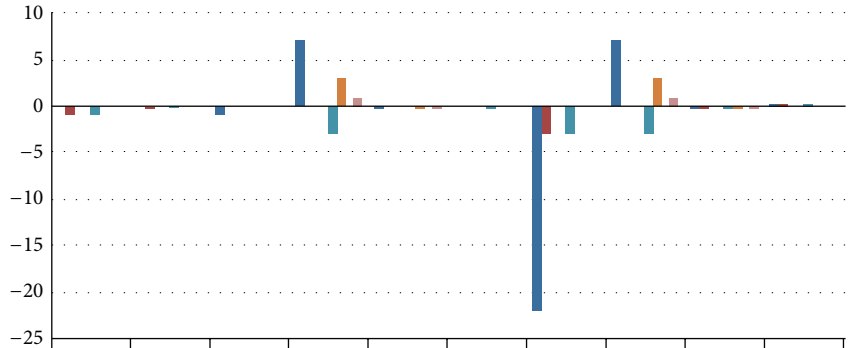
```
Object around (Object aa, Object bb):
    call (static public Object add(Object, Object)) && args(aa,bb)
{
    String ca = aa.getClass().getSimpleName();
    String cb = bb.getClass().getSimpleName();
    if(ca.equals(cb)&&ca.equals("Integer")) {
       int a=((Integer)aa).intvalue();
       int b=((Integer)bb).intValue();
       long c=(long)a+b;
       if(c <Integer.MIN_VALUE)
          throw new ArithmeticException("int underflow");
       else if(c> Integer.MAX_VALUE)
          throw new ArithmeticException("int overflow");
       return (new Integer((int)c));}
    if(ca.equals(cb)&&ca.equals("Float")) {
       float a=((Float)aa).floatValue();
       float b=((Float)bb).floatValue();
       float c=a+b;
       if(c==0 && (a!=-b))
          throw new ArithmeticException("float underflow");
       else if(c==Float.POSITIVE_INFINITY||c==Float.NEGATIVE_INFINITY)
          throw new ArithmeticException("float overflow");
       return (new Float(c));
    }
    if(ca.equals(cb)&&ca.equals("Double")) {
       Double a=((Double) aa).doubleValue();
       Double b=((Double)bb).doubleValue();
       double c=a+b;
       if(c==0 && (a!=-b))
          throw new ArithmeticException("double underflow");
       else if(c==Double.POSITIVE_INFINITY| |c==Double.NEGATIVE_INFINITY)
          throw new ArithmeticException("double overflow");
       return (new Double(c));}
    if (ca.equals(cb)&&ca.equals("Long")) {
       long a=((Long)aa).longValue();
       long b=((Long)bb).longValue();
       long c=a+b;
       if (a > 0 && b > 0 && (c < a||c < b))
          throw new ArithmeticException("long Overflow");
       else if (a < 0 && b < 0 && (c > a||c > b))
          throw new ArithmeticException("long underflow");
       return (new Long(c));
    }
    return null;
}
```

Algorithm 2: Calculation accuracy aspect snippet code for capturing addition operation overflow and underflow.

the specification of the uoeb.Japrosim.crosscutting.concerns package which has a total resilience and the high (Ce) metric value. Additionally, the overall increase in (Ce) gives an idea of the improvement in packages independency, thus increasing package reusability. A decrease in (NOT) and (A) for both uoeb.Japrosim.animation and uoeb.Japrosim .statistics.steady refers to the transfer of EventObserver and StatObserver interfaces to the animation and Steady-StateDetection aspects, respectively. Further, the increase in (Ca) metric refers to the presence of additional package, that is, uoeb.Japrosim.crosscutting.concerns. Finally, the increase in the (Dn) value refers to (I) value variation.

7. Conclusions

Aspect-oriented simulation is a promising research field that offers a solution for the main problem of the object-oriented simulation which is the separation of the simulation crosscutting concerns in modular way. Aspect-oriented



	NOT	*A*	RMartin *Ce*	RMartin *Ca*	RMartin *I*	RMartin *D*	*Ce*	*Ca*	*I*	*Dn*
uoeb.japrosim.kernal	0	0	−1	7	−0.03	0.03	−22	7	−0.31	0.31
uoeb.japrosim.statistics.steady	−1	−0.15	0	0	0	0.15	−2	0	−0.16	0.18
uoeb.japrosim.gui	0	0	0	0	0	0	0	0	0	0
uoeb.japrosim.random.distributions	0	0	0	0	0	0	0	0	0	0
uoeb.japrosim.animation	−1	−0.11	0	−3	0.15	−0.03	−3	−3	−0.15	0.26
uoeb.japrosim.statistics	0	0	0	3	−0.04	0.04	0	3	−0.02	0.02
uoeb.japrosim.utilities	0	0	0	0	0	0	0	0	0	0
uoeb.japrosim.random	0	0	0	1	−0.11	0.11	0	1	−0.16	0.16

Figure 8: The difference in measurement results for both AO and OO Japrosim packages metrics.

simulation systems have several features as the high modularity, reusability, maintainability, and visibility. The stated objective of this paper is to enhance the discrete event simulation with new programming paradigms as the aspect-oriented paradigm. This objective is realized in specific terms through the identification of the main simulation crosscutting concerns such as steady state detection, graphical animation, graphical user interface, exception handling, calculation accuracy, simulation trace, and synchronization which degrade simulation systems quality. At the experimental level, the object-oriented version of the Japrosim framework has been chosen as a practical example. The main simulation crosscutting concerns identified earlier are separated from the core concerns. The new aspect-oriented version of the framework has been successfully obtained and it is available at: http://sourceforge.net/projects/Japrosim/files/AOP-Japrosim. Additionally, to prove the impact of the AOP paradigm on Japrosim design quality as flexibility, maintainability, and reusability properties, an automatic assessment of software metrics for AO version of Japrosim is achieved using the AOPMetrics tool which is a common metrics tool for both Java and AspectJ programs.

Besides that, a brief overview of the advanced separation of concerns approaches, namely, SOP, AOP, CF, and AP with a comparative study is discussed, so the exploitation of each one in simulation domain could be considered as a future issue.

Recently, aspect-oriented ideas are spread through earlier stages of the software development process by means of the aspect-oriented software development (AOSD) techniques. An area of future research would be to assess the benefits of the integration of the AOSD techniques within simulation systems and using Japrosim library as a practical example.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

The authors are grateful to the anonymous reviewers for their significant roles in improvement of this research quality.

References

[1] M. Akşit, B. Tekinerdoğan, and L. Bergmans, "The six concerns for separation of concerns," in *Proceedings of the Workshop on Advanced Separation of Concerns (ECOOP '01)*, 2001, http://trese.cs.utwente.nl/workshops/ecoop01asoc/.

[2] L. Lucian and I. Despi, "Aspect oriented programming challenges," *Anale Seria Informatica*, vol. 2, no. 1, pp. 65–70, 2005.

[3] V. Vranié, "Towards multi-paradigm software development," *Journal of Computing and Information Technology*, vol. 10, no. 2, pp. 133–147, 2002.

[4] R. Laddad, *Aspectj in Action: Enterprise AOP with Spring Applications*, Greenwich, Conn, USA, Manning Publications, 2nd edition, 2009.

[5] J. A. P. Díaz, M. R. Campo, and M. E. Fayad, "A language for simulation: bringing separation of concerns to the front," in *Proceedings of the 14th Aspects and Dimensions of Concerns Workshop (ECOOP '00)*, Cannes, France, 2000.

[6] W. Weiland, R. Weatherly, K. Ring et al., "Simplified concurrency: a java simulation framework," in *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications Conference*, 2005.

[7] A. U. Aksu, F. Belet, and B. Zdemir, "Developing aspects for a discrete event simulation system," in *Proceedings of the 3rd*

Turkish Aspect-Oriented Software Development Workshop, pp. 84–93, Bilkent University, Ankara, Turkey, 2008.

[8] Z. Vaira and A. Caplinskas, "Application of pure aspect-oriented design patterns in the development of ao frameworks: a case study," *Information Sciences*, vol. 56, pp. 146–155, 2011.

[9] J. Ribault, O. Dalle, D. Conan, and S. Leriche, "OSIF: a framework to instrument, validate, and analyze simulations," in *Proceedings of the 3rd International Conference on Simulation Tools and Techniques (SIMUTools '10)*, pp. 56–60, Malaga, Spain, 2010.

[10] L. Gulyás and T. Kozsik, "The use of aspect-oriented programming in scientific simulations," in *Proceedings of the Fenno-Ugric Symposium on Software Technology (FUSST '99)*, J. Penjam, Ed., Technical Report CS 104/99, pp. 17–28, Tallinn, Estonia, August 1999.

[11] V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, "Multiagent simulation of physical phenomena by means of aspect programming," in *Computational Science—ICCS 2006*, S. Bieniasz, S. Ciszewski, and B. Śnieżyński, Eds., vol. 3993 of *Lecture Notes in Computer Science*, pp. 759–766, 2006.

[12] T. B. Ionescu, A. Piater, W. Scheuermann, and E. Laurien, "An aspect-oriented approach for the development of complex simulation software," *The Journal of Object Technology*, vol. 9, no. 1, pp. 161–181, 2010.

[13] J. Brichau, K. Mens, and K. D. Volder, "Building composable aspect-specific languages with logic metaprogramming," in *Generative Programming and Component Engineering: Proceedings of the ACM SIGPLAN/SIGSOFT Conference, GPCE 2002 Pittsburgh, PA, USA, October 6–8, 2002*, vol. 2487 of *Lecture Notes in Computer Science*, pp. 110–127, Springer, Berlin, Germany, 2002.

[14] D. Park, S. Kang, and J. Lee, "Design phase analysis of software qualities using aspect-oriented programming," in *Proceedings of the 7th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD '06)*, Y.-T. Song, C. Lu, and R. Lee, Eds., pp. 29–34, IEEE Computer Society, Las Vegas, Nev, USA, June 2006.

[15] T. Mehmood, N. Ashraf, K. Rasheed, and S. Tauseef-ur-Rehman, "Framework for modeling performance in multi agent systems (MAS) using aspect-oriented programming (AOP)," in *Proceedings of the 6th Australasian Workshop on Software and System Architectures (AWSA '05)*, 2005.

[16] H. Shen and D. C. Petriu, "Performance analysis of UML models using aspect-oriented modeling techniques," in *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS '05)*, pp. 156–170, Springer, Montego Bay, Jamaica, 2005.

[17] B. Belattar and A. Bourouis, "Ascertaining important features of the JAPROSIM simulation library," in *Proceedings of the EUROPMENT International Conference on Systems, Control, Signal Processing and Informatics*, pp. 515–522, Rhodes Island, Greece, 2013.

[18] W. Harrison and H. Ossher, "Subject-oriented programming (a critique of pure objects)," in *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pp. 411–428, Washington, DC, USA, 1993.

[19] L. Bergmans and M. Aksit, *Composing Multiple Concerns Using Composition Filters*, TRESE Group, Department of Computer Science, University of Twente, 2001.

[20] M. Chibani, B. Belattar, and A. Bourouis, "Toward an aspect-oriented simulation," *International Journal of New Computer Architectures and Their Applications*, vol. 3, no. 1, pp. 1–10, 2013.

[21] M. Chibani, B. Belattar, and A. Bourouis, "Aspect Oriented simulation: a case study with the JAPROSIM framework," in *Proceedings of the 27th Annual European Simulation and Modelling Conference (ESM '13)*, S. Onggo and A. Kavička, Eds., pp. 91–98, Lancaster University, Lancaster, UK, October 2013.

[22] J. Ribault and O. Dalle, "Enabling advanced simulation scenarios with new software engineering techniques," in *Proceedings of the 20th European Modeling and Simulation Symposium (EMSS '08)*, pp. 515–520, Briatico, Italy, September 2008.

[23] A. Piater, T. B. Ionescu, and W. Scheuermann, "A distributed simulation framework for mission critical systems in nuclear engineering and radiological protection," *International Journal of Computers, Communications & Control*, vol. 3, pp. 448–453, 2008.

[24] S. L. Tsang, *An evaluation of AOP for Java-based real-time systems development [M.S. thesis]*, University of Dublin, Dublin, Ireland, 2004.

[25] D. Park and S. Kang, "Design phase analysis of software performance using aspect-oriented programming," in *Proceedings of the 5th International Workshop on Aspect-Oriented Modeling*, Lisbon, Portugal, 2004.

[26] K. Cooper, L. Dai, and Y. Deng, "Modeling performance as an aspect: a UML based approach," in *Proceedings of the 4th AOSD Modeling with UML Workshop*, 2003.

[27] L. Blair, G. Blair, and A. Andersen, "Separating functional behaviour and performance constraints: aspect oriented specication," Tech. Rep., 1998.

[28] R. E. Filman and D. P. Friedman, "Aspect-oriented programming is quantification and obliviousness," in *Proceedings of the Workshop on Advanced separation of Concerns (OOPSLA '00)*, October 2000.

[29] G. Kiczales, J. Lamping, A. Mendhekar et al., "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, vol. 1241 of *Lecture Notes in Computer Science*, pp. 220–242, 1997.

[30] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal, "Subject-oriented composition rules," in *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, vol. 2, pp. 235–250, Austin, Tex, USA.

[31] H. Ossher, W. Harrison, F. Budinsky et al., "Subject-oriented programming: supporting decentralized development of objects," in *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, IBM, Santa Clara, Calif, USA, July 1994.

[32] K. Lieberherr and D. H. Lorenz, "Coupling aspect-oriented and adaptive programming," in *Aspect-Oriented Software Development*, R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, Eds., pp. 145–164, Addison-Wesley, Boston, Mass, USA, 2005.

[33] P. Tarr, H. Ossher, W. Harrison, and S. Sutton Jr., "N degrees of separation: multi-dimensional separation of concerns," in *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pp. 107–119, 1999.

[34] K. Hoad, S. Robinson, and R. Davies, "Automating warm-up length estimation," in *Proceedings of the Winter Simulation Conference*, S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, and J. W. Fowler, Eds., pp. 532–540, 2008.

[35] C. A. S. D. Cunha, *Reusable aspect-oriented implementations of concurrency patterns and mechanisms [M.S. thesis]*, University of Minho, Braga, Portugal, 2006.

[36] B. Abdelhabib and B. Brahim, "JAPROSIM: a Java framework for process interaction discrete event simulation," *Journal of Object Technology*, vol. 7, no. 1, pp. 103–119, 2008.

[37] M. Stochmialek, AOPmetrics, 2005, http://aopmetrics.tigris.org.

[38] R. Martin, "OO design quality metrics—an analysis of dependencies," in *Proceedings of the Pragmatic and Theoretical Directions in Object-Oriented Software Metrics Workshop*, OOPSLA, 1994.