Developing Aspects for a Discrete Event Simulation System

M. Uğur Aksu, Faruk Belet, Bahadır Özdemir
Department Of Computer Engineering, Bilkent University
{aksu, fbelet,bozdemir} @ cs.bilkent.edu.tr

Abstract

Simkit is a simulation modeling tool. Like many simulation systems, Simkit implements a number of concerns, such as event scheduling, event handling, keeping track of a simulation's state and creating random number generators. It appears that these concerns crosscut over multiple modules in the system. This increases the complexity and reduces the maintainability and ease of use of this tool. In this paper, we identify the crosscutting concerns in Simkit and try to refactor it with Aspect Oriented Software Design.

1. Introduction

Simulation is the practice of using abstraction and idealization to produce less complex models of real life systems, whose present or proposed alternative behavior can be exercised in a feasible and less costly way. [1, 2]

Event Graphs are a way of graphically representing DES models, like Petri Nets, System Dynamics and UML. In a high level view, Event Graphs employ nodes and edges; to denote events and to scheme interactions among events, respectively. Very often, Event Graphs are preferred due to their expressive power.

Simkit is an open source Discrete Event Simulation (DES) tool that has been developed for educational purposes. It has been designed with object oriented approach and the Java language has been used for the implementation of it. Simkit aims to make implementation of Event Graph models as easy as possible. In this respect, Simkit simply extends the basic Event Graph paradigm by a component architecture that is based on loose coupling of simulation components. [3]

In this paper, we argue that some general and domain specific concerns of Simkit are crosscutting over multiple modules. This increases the complexity and reduces the maintainability and ease of use of the tool. This argument can be justified by the fact that the domain requirements for simulation systems are fairly complex and object oriented paradigm is not capable enough to deal with such complexity.

Concerns we face in Simkit can be architecturally divided into two groups: (1) Concerns that apply directly to the Simkit packages itself that implement library functions. (2) Concerns that appear in the components/classes defined by simulation modelers.

We propose to refactor Simkit using Aspect Oriented Software Development (AOSD) paradigm. AOSD is an emerging software development technology that seeks to modularize software systems by expressing scattered and tangling concerns separately from the core logic of the systems. It is usually practiced to enhance readability, modularity, maintainability and extensibility of software systems.

The remainder of this paper is organized as follows. In section 2, we provide a short background on simulations and DES modeling with Event Graphs. In section 3, we discuss the object oriented design of the Simkit and explain the Event Graphs in more detail with its corresponding implementation in Simkit. In section 4, we try to identify crosscutting concerns of Simkit and propose solutions i.e. aspects in the parlance of AOSD paradigm, to the crosscutting/tangling concerns observed. In section 4, we discuss our findings from this study. Finally, in section 5, we provide our conclusion and describe related future work.

2. Simulations & Discrete Event Simulation (DES) Modeling with Event Graphs

Simulation is "the process of describing a real system and using this model for experimentation, with the goal of understanding the system's behavior or to explore alternative strategies for its operation" [1] Using simulations we can improve our understanding of systems' behavior and the effects of interactions among components.

Simulation systems can be categorized in different ways and one of them is based on the properties of a simulation model's state variables. [2] Using this classification approach, models can be described either as static or dynamic. Dynamic models can be further categorized into discrete event or continuous models. In Discrete Event Simulations (DES), all system state changes are mapped to discrete events, assuming nothing relevant happens in between, while in continuous simulations states change steadily over time. [2]

There are three fundamental components for DES models. These are a set of events, a set of state variables and the simulation time.

In DES, events occur at the points in time at which state variables change values and simulation time is updated to the current event's time. Events are kept in an Event List which is simply a to-do list. It holds information regarding what event is being scheduled and the simulation time at which the event is to occur. [1]

Event Graphs, a way of graphically representing DES models, uses nodes and edges to depict DES models. Nodes denote events and edges are used to schedule other events with two optional parameters: a Boolean condition and/or a time delay. [1] Figure 1 shows the basic construct for event graphs.

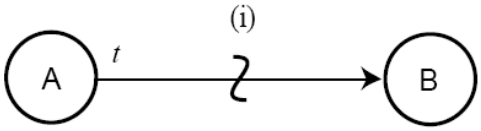

Figure 1. Fundamental Event Graph Construct [1]

Figure 1 is interpreted as follows: When event A occurs, event B is scheduled to occur at time *t* if the condition *i* is true.

More detail on DES and its corresponding implementation in Simkit will be elucidated in the next section.

3. Object Oriented Design of Simkit and its Correspondence to Event Graphs

Simkit is an Event Graph based DES modeling tool with an implementation of general purpose high level programming language, augmented by a package of suitable simulation related library procedures. Every element in an Event Graph model has a corresponding element in Simkit.

With regard to the object oriented design, three fundamental packages of Simkit are: (1) simkit: where the core simulation related logic is implemented; (2) simkit.random: a random number generation factory and (3) simkit.stats: a utility package to collect simulation related statistics.

A simplified object oriented design of Simkit that depicts some core components is shown in the Figure 2.

DES related core components of Simkit that implement event handling, state changes and simulating time are explained at the next subsections. More detail on DES with Simkit can be found at [4].

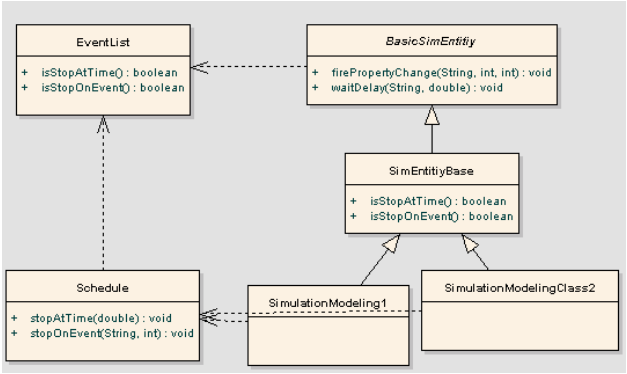

Figure 2. A Simplified Object Oriented Design of Simkit

3.1. Future Event List

As all DES frameworks require an implementation of a Future Event List (FEL), Simkit implements a FEL in a class called simkit.Schedule. FEL is represented by a variable java.util.SortedSet in the package simkit. Schedule class and it contains objects of type SimEvent that contains data on which event it represents and a scheduled time to occur information.

Instead of directly placing events on FEL, the programmer invokes the waitDelay() method on an instance of simkit.SimEntityBase so that the details of the FEL could be hidden from the modelers.

3.2. SimEntity and SimEntityBase

An abstract class and an interface are employed in Simkit to help encapsulate the activities with regard to scheduling and processing events.

Any class designed to interact with the FEL must implement the SimEntity interface that specifies a set of methods for interaction.

User defined "do" methods in a subclass of SimEntityBase are used to implement the behavior of events. The naming convention for "do" methods is as the event name followed by a "do" prefix. Behavior of scheduling edges are implemented by "waitDelay()" methods for which the simplest signature is as such: (string, double), where the first argument is the name of the event with out the "do" and the second argument is the delay associated with the scheduled event.

The Simkit code corresponding to Figure 1 is implemented in the Figure 3, to elaborate more on user defined "do" methods and calling of waitDelay() constructs.

```
public void doA() {
     <code to perform state transition for event A>
     if (1) {
         waitDelay("B", t);
     }
}
```

Figure 3. Simkit Code for Basic Event Graph Construct [4]

As stated before, the first argument in the waitDelay() methods is a string that stands for an event name. Using Java's reflection feature, Simkit determines the corresponding method by creating a SimEvent in SimEntityBase and adding it to the FEL. Then, when an event occurs, the Event List invokes a callback method on the object that scheduled it.

3.3. Event Handling Mechanism of Simkit

Event handling convention of Simkit is as follows: When an event occurs, first the simulation time is updated to the time of event that has occurred and all state changes associated with that event are performed. Next, all further events are scheduled and finally the event notice is removed from the Event List.

3.4. Sample Simkit Models

In this subsection, adopting the domain specific graphical notation of Event Graphs, we will explain two sample DES models that will be used in this paper. Figure 4 models a very simple simulation case: Arrival Process.

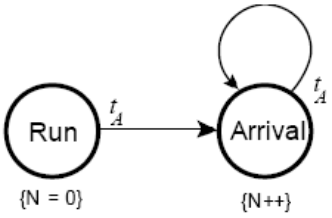

Figure 4. Event Graph for Arrival Process [1] [4]

The initialization of a Simkit simulation run is achieved by a dummy event called Run. At the start of the simulation, a Run event both initializes the state variables and schedules the initial events on the Event List. The initialization of state values are achieved by a method called reset(), while the scheduling of initial events is done by the doRun() method. In the model, there exists a single state variable, numberArrivals (N), which counts the cumulative number of arrivals since time 0.0 and it is incremented by one upon the occurrence of each Arrival event.

After explaining the basic constructs of the model, we can interpret Figure 4 as follows: A Run event is scheduled to occur at simulation time 0.0. With the occurrence of the Run event, first simulation time is set to 0.0 and the state variable N is set to 0. Then an Arrival event is scheduled to occur at time t_A. In the same way, for each Arrival event that is run, first the simulation time is updated to the time of the Arrival event, then state variable N is incremented by one and another Arrival event is scheduled to occur at time t_A.

The Simkit implementation of the Arrival Process is given in the Figure 5.

```
private RandomVariate interarrival;
protected int numberArrivals;
public void reset() {
     numberArrivals = 0;
}
public void doRun() {
     waitDelay("Arrival", interarrival.generate());
}
public void doArrival() {
     firePropertyChange("numberArrivals
          ",numberArrivals, ++numberArrivals);
     waitDelay("Arrival", interarrival.generate());
}
```

Figure 5. Simkit Code for Arrival Process [4]

Another example is the Multiple Server Queue case. In this model, customers arrive to a service facility according to an arrival process and are served by one of the k servers. Customers arriving to find all servers busy wait in a single queue and are served in order of their arrival time. [1]

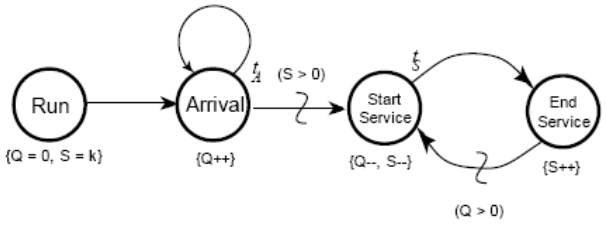

Figure 6. Event Graph for Multiple Server Queue [1]

State variables and parameters for the model in Figure 6 are: Q = # of customers in queue, S = # of available servers, t_A = interarrival times, t_S = service times, k = total number of servers.

3.5. Design Patterns Used in Simkit

In the Simkit library, several design patterns such as observer, factory, builder, and adapter are used. Observer and factory design patterns are the most important ones in the library. In general, we can not state any significant shortcoming in the design of Simkit with regard to design patterns. Yet, we have added another design pattern i.e. façade that facilitates the usage of Simkit for modelers.

3.5.1. Observer Pattern

Simkit uses two observer patterns to implement its component interoperability. [4]

First, the SimEventListener pattern is used to connect simulation components in a loosely coupled manner. SimEvents are always invoked by a callback to the scheduling object that invokes the corresponding "do" method. The SimEvent is then dispatched to every SimEventListener that has been explicitly registered in that object's SimEvents. [4]

Secondly, the PropertyChangeListener pattern comes into play whenever a state variable changes value. In that case, a PropertyChangeEvent is dispatched to registered PropertyChangeListener objects. The purpose of PropertyChangeEvents is to support generic observation of the simulation state trajectories, as well as any function thereof. [4]

3.5.2. Factory Method Pattern

Simkit uses a combination of RandomVariate interfaces and a factory that is called to produce instances of the desired implementation using only "generic" data which is strings, objects, and numbers. [4]

3.5.3. Façade Pattern

Running a Simkit model requires the implementation of the following tasks:

1. Instantiate desired objects.
2. Register SimEventListener objects.
3. Register PropertyChangeListener objects.
4. Set stopping time or stopping criteria.
5. Set the mode of the run.
   (verbose/quiet, single-step/continuous running)
6. Reset all SimEntityBase instances.
7. Start the simulation.

First of all, if the programmer wants to collect statistics for a property, a new statistics object should be created for each property. In addition, if the programmer desires to keep a log of a process, a new property dumper object needs to be created for each process. Moreover, SimEventListener and PropertyChangeListener objects need to be created. And finally, all these objects should be registered to the corresponding process objects. To sum up, all these operations creates confusion in the main method. Even though there is an interface in the Simkit library, i.e. BasicSimEntitiy that combines elements of a simulation, programmers are still to write simulation classes that implement this interface. [4]

In order to lessen the burden on modelers we have designed a Façade class, "Simulation", which offers a generic interface. For this aim, we have also designed an auxiliary class, "Process", which stores elements of processes. Simulation objects are constructed with an array of Process objects as an argument. The statistics and listener objects are created and registered by the Simulation class. After this point, the programmer should set the parameters for simulations if needed and just run.

The simplified class diagram of the façade pattern is shown in figure 7.

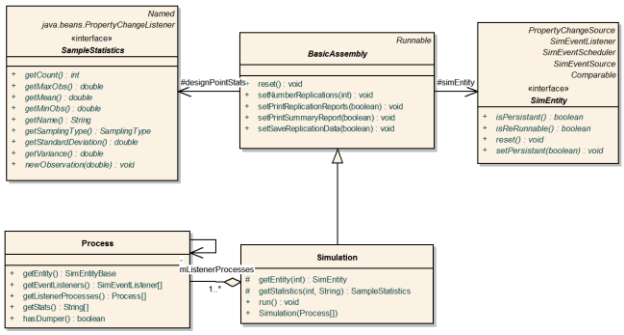

Figure 7. The Simplified Class Diagram of the Façade Pattern

4. Identification of Crosscutting Concerns & Proposed AOSD Solutions

In AOSD approach aspects are generally categorized into three categories: (3) Core Aspects: Form the core part of the system; (2) Development Aspects: Helps the development of the system; (1) Enforcement Aspects: Policies that are desired to hold in the design of the system. We will describe the crosscutting and/or tangling aspects detected in the Simkit by adopting this classification.

4.1. Production Concerns

4.1.1. Simulation Termination Rules

There are a number of ways for terminating a DES. Most commonly used four methods are as follows: (1) Specifying a fixed length model time for the simulation. (2) Defining a simulation termination event. (3) Defining a fixed length sample size. (4) Terminating the simulation in terms of the model's state.

Taking the best software development advices into use, this concern of terminating simulations can be implemented cohesively and separately from the other simulation related concerns.

For the AOSD solution, we have defined an aspect that combines the first three termination rules that has been implemented in the Schedule and EventList Classes in the Simkit. We used static introduction to move the related code to the target aspect. Moreover, in this aspect, we

have implemented the last termination rule as a new feature, which enables terminating the simulation in terms of the model's state.

An AspectJ code example for this solution is demonstrated in Figure 8.

```
privileged public aspect SimStopRulesAspect {

    public pointcut somePointInSimRunLoop(simkit.EventList eventList) :
          execution(public boolean simkit.EventList.isSingleStep()) && this(eventList);

    before(simkit.EventList eventList) : somePointInSimRunLoop(eventList) {
        eventList.checkStopState();
    }

    private int simkit.EventList.stopValue = 0;
    private simkit.SimEntityBase simkit.EventList.simObject = null;
    private String simkit.EventList.propertyName = "";
    private boolean simkit.EventList.isStopAtStateIfGreater = false;
    private boolean simkit.EventList.isStopAtState = false;

    public static void simkit.Schedule.stopAtTime(double atTime) {
        defaultEventList.stopAtTime(atTime);
    }

    public static void simkit.Schedule.stopOnEvent(int numberEvents,
            String eventName, Class... eventSignature) {
        defaultEventList.stopOnEvent(numberEvents, eventName, eventSignature);
    }

    public void simkit.EventList.stopAtTime(double time) {
        // Related code..
    }

    public void simkit.EventList.stopOnEvent(int numberEvents,
            String eventName, Class... signature) {
        // Related Code..
    }
    public static void simkit.Schedule.stopOnEvent(int numberEvents,
            String eventName, Class... eventSignature) {
        defaultEventList.stopOnEvent(numberEvents, eventName, eventSignature);
    }

    public void simkit.EventList.stopAtTime(double time) {
        // Related code..
    }

    public void simkit.EventList.stopOnEvent(int numberEvents,
            String eventName, Class... signature) {
        // Related Code..
    }

    public static void simkit.Schedule.stopAtState(int stopValue,
            boolean isStopAtStateIfGreater, simkit.SimEntityBase simObject,
            String propertyName) {
        defaultEventList.stopAtState(stopValue, isStopAtStateIfGreater,
                simObject, propertyName);
    }

    public void simkit.EventList.stopAtState(int stopValue,
            boolean isStopAtStateIfGreater, simkit.SimEntityBase simObject,
            String propertyName) {
        this.stopValue = stopValue;
        this.simObject = simObject;
        this.propertyName = propertyName;
        this.isStopAtStateIfGreater = isStopAtStateIfGreater;
        this.isStopAtState = true;
    }

    public void simkit.EventList.checkStopState() {
        // Related code..
    }
}
```

Figure 8. AspectJ Code for Simulation Termination Rules Aspect

The pointcut in the code snippet above locates a point in the simulation running loop and the before advice is used to inject the "stop the simulation at state" related code at that point.

4.1.2. Persistence: Restoring a Simulation Run

Persistence turns out to be a very generic production concern that has a fairly significant crosscutting and tangling nature in almost any software system that is considerably large.

Simulation runs are almost always costly in terms of time. Yet, simulations might terminate unexpectedly due to a number of reasons such as interruptions by misbehaving operating systems or misbehaving other applications. It is not only frustrating but also time consuming to rerun a simulation. Moreover, sometimes simulations can be terminated properly by users but later on, it might be desired to run the same simulation for larger sample sizes. In such cases, it comes in very handy if a simulation can resume from its last stable state.

To achieve such a persistence feature, a simulation program/package must remember some fundamental properties of the previous simulation run such as: (1) Scheduled but not handled events, in other terms, last stable state of the Event List. (2) Values of the simulation state variables. (3) Random number generation objects and the number of times they have been pooled before the simulation termination.

We have provided a persistence solution using AOSD only for the first two properties described above. Even this partial implementation proved to exhibit significant crosscutting view for the concern.

We have employed 3 aspects in the aspect oriented design. StoreEventListStateAspect stores the last stable Event List; StoreSimVariablesStatesAspect stores the last stable values of the state variables. These values are stored in files each time an event is handled. And the last Aspect PersistenceAspect reads the values stored in files for both state variables and scheduled events, then resumes the simulation run if a method call such as resumeSimulation(true) is provided by the user.

AspectJ code examples for the persistence aspect are demonstrated in Figure 9.

```
public privileged aspect StoreEventListStateAspect {

    private static Formatter fileOutput;

    public void simkit.EventList.storeEventListState() {
        StoreEventListStateAspect.storeEventListState(this);
    }

    public static void storeEventListState(simkit.EventList eventList) {
        // Related code: stores event list state
    }

    public pointcut storePoint(simkit.EventList eventList) :
          call(boolean simkit.EventList.isStopOnEvent()) && this(eventList)
          && (withincode(public void simkit.EventList.startSimulation()));

    after(simkit.EventList eventList) : storePoint(eventList) {
        eventList.storeEventListState();
    }
}
```

```
public aspect StoreSimVariablesStatesAspect {

    private static Map stateVariableTable = new TreeMap();
    private static Formatter fileOutput;

    pointcut propertyChangeMethods(String propertyName, int newPropertyValue):
        call(* simkit.BasicSimEntity.firePropertyChange(String, int))
        && ! within(simkit..*) && args(propertyName, newPropertyValue);

    before(String propertyName, int newPropertyValue) :
        propertyChangeMethods(propertyName, newPropertyValue) {
        stateVariableTable.put(propertyName, newPropertyValue);
    }

    public void simkit.EventList.storeStateVariablesStates() {
        StoreSimVariablesStatesAspect.storeStateVariablesStates();
    }

    public static void storeStateVariablesStates() {
        // Related code: stores state variables' states
    }

    public pointcut storePoint(simkit.EventList eventList) :
          call(boolean simkit.EventList.isStopOnEvent()) && this(eventList)
          && (withincode(public void simkit.EventList.startSimulation()));

    after(simkit.EventList eventList) : storePoint(eventList) {
        eventList.storeStateVariablesStates();
    }
}



privileged public aspect PersistenceAspect {

    public static boolean isResumeSimulation = false;

    public static void simkit.Schedule.resumeSimulation(boolean isResume) {
        PersistenceAspect.isResumeSimulation = isResume;
    }

    public pointcut resetProperty() :
        ! within(simkit..*) && set(protected int simkit.SimEntityBase+.*)
        && withincode(public void reset());

    after() : resetProperty() {
        // Related code: initiliaze using the last state variables' states
    }


    pointcut locateDoRun(simkit.SimEntityBase object):
        execution(* simkit.SimEntityBase+.doRun()) && ! within(simkit..*)
        && target(object);

    void around(simkit.SimEntityBase object): locateDoRun(object) {
        // Related code: set the event list to the last stable state recorded
    }

    public static String readEventListState() {
        // Related code: read event list from a file
    }

    public static String readSimVariablesStates() {
        // Related code: read state variables from a file
    }
}
```

Figure 9. AspectJ Code for Persistence Aspect

The pointcuts in the first two aspects locate points in the simulation running loop. At these points, codes related to writing to files are injected by after advices. The after advice in the third aspect, resets the state variables' values to the values read from the stored file and the around advice puts the events read from the file to the Event List instead of letting the doRun method start the simulation from the very beginning.

The crosscutting nature of this concern is demonstrated in Figure 10.

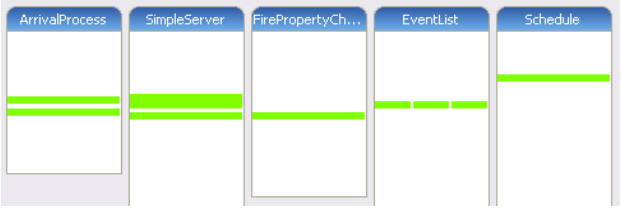

Figure 10. Crosscutting Nature of the Persistence Concern

4.1.3. Resource Pooling Concern

Resource pooling is a system-wide and generally a scattered concern, that is used to keep the resources created earlier around instead of disposing them, so that they can be reused later on instead of recreating new ones. A common example can be given for database applications where instead of creating and destroying database connection objects each time we need, previously created objects that are returned from the resource pools can be used. This feature is especially important in terms of time-efficiency. While resource pooling usually decrements the time needed of obtaining a resource, this benefit faces us with the challenge of extra memory and other resources consumed by resource pool to store objects created earlier. This is called the space/time tradeoff of resource pooling. Taking this tradeoff into account, a system designer must enable resource pooling only if the desire for improved speed outweighs the cost of extra memory.

In Simkit, we believe that keeping a resource pool for random number generation objects might come very handy to increase simulation compilation time.

In Simkit, random number generation is achieved by a group of factory classes in the simkit.random package. Each of these factory classes implement their own resource pools in a scattered way, to address the concern (faster simulation compilations) explained above. Factory classes initialize a random number generator object whose class name is taken via getInstance(string) method of related factory. By this approach, factory classes implement lazy initializations by searching packages or adding some suffix such as "variate" to the className. Behind getInstance(string) and its variations, each factory keeps a cache of previously created objects. At this point, crosscutting concern appears due to because a caching mechanism that is spread over different factories. To modularize and generalize this caching approach in system wide manner, we convert the scattered caches into a single caching aspect. A simplified class diagram of resource pooling concern is shown in Figure 11.

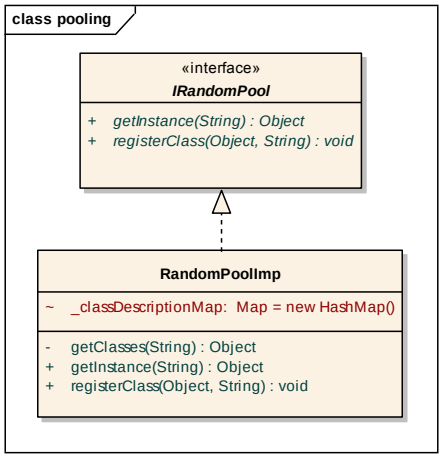

Figure 11. Resource Pooling Class Diagram

In this way, when a new random number generation factory is desired to be added to the simkit.random package, resource pooling mechanism for this newly added factory does not have to be implemented separately, which would be the case for a pure object oriented design that does not take advantage of AOSD paradigm. An aspect code example for this solution is given in Figure 12.

```
public aspect RandomPoolingAspect {

    /**
     * Holds the registered resources in the resource pooling
     */
    IRandomPool _randomPool = new RandomPoolImp();

    pointcut instanceCreation(String className):
        (call(* simkit.random..*Factory+.findFullyQualifiedNameFor(String))
        || call(* simkit.random..*Factory+.getClassFor(String)))
        && args(className);

    /**
     * around advise that returns objects from pool if it exist in the resource pool
     * using RandomPoolImp class.
     *
     */
    Object around(String className) : instanceCreation(className) {
        Object myClass = _randomPool.getInstance(className);
        //Related Code..
        //Return or register object in resource pool
    }
}
```

Figure 12. AspectJ code for Resource Pooling

4.1.4. Observer Pattern Concern

Observer/Listener pattern is widely used especially in multi-threaded programs. When a property of a class observed changes, then related functions of all listener classes are called. However, this cause tangling of fire property change codes into the observed class and scattering of observer pattern. In addition, this requirement may cause defects in the code. For example, the programmer may forget to add the code for calling fire property change functions affects the correctness of a program. To overcome this problem, a core aspect which automatically detects changes of state variables and handles calling corresponding fire property functions can be added.

In Simkit, when a change in state variables occurs, a corresponding fire property change function should be called. Providing an aspect that handles the call of fire property change functions will crosscut the process classes.

The programmer can change the values of state variables inside “do” methods, in Simkit. These changes should be detected by an aspect and corresponding fire property change functions should be called. Changing of state variables outside the “reset” and “do” methods is intended to be restricted by another aspect. Thus, finding property changes in “do” methods is sufficient because fire property change function is not invoked for “reset” functions. Corresponding fire property change functions are different in doRun and other “do” methods. As a result, we divided detecting property change into two parts. In the first part, we only detect property changes in “do” methods other than doRun.

Around advice is used for the pointcut which identifies property changes in “do” methods, because fire property change function should be invoked after setting the value to property; however, we need both old and new values of the property.

In the second part, we just need to identify calls to doRun methods. We use a pointcut which identifies calls to a specific library method in which “do” methods are invoked. Before advice is used to invoke the related fire property change functions at these pointcuts.

An AspectJ code example for this solution is given in Figure 13.

```
public aspect FirePropertyChangeAspect {

    public pointcut propertyChange(Object value) :
        ! within(simkit..*) && set(protected int simkit.SimEntityBase+.*)
        && withincode(public void do*()) && ! withincode(public void doRun())
        && args(value);

    void around(Object value) : propertyChange(value) {
        SimEntityBase target = (SimEntityBase) thisJoinPoint.getTarget();
        String field = thisJoinPoint.getSignature().getName();
        Object old = target.getProperty(field);

        proceed(value);
        // Related call: call corresponding firePropertyChange function

    }

    public pointcut callDoRun(SimEvent event) :
        call(public void simkit.SimEntityBase.processSimEvent(SimEvent))
        && args(event);

    before(SimEvent event) : callDoRun(event) {
        if (event.getFullMethodName().equals("doRun()")
                && event.getSource().equals(thisJoinPoint.getTarget())) {

            SimEntityBase target = (SimEntityBase) thisJoinPoint.getTarget();
            Field[] fields = target.getClass().getDeclaredFields();

            for (Field field : fields) {
                // Related call: call corresponding firePropertyChange function
            }
        }
    }
}
```

Figure 13. AspectJ Code for Observer Pattern Aspect

4.2. Development Concerns

4.2.1. Checking Main Method's Parameters

Checking main method's parameters is a classic idiom adopted widely by the AOSD community [5]. This feature also proves to be very useful in our case. Checking the parameters separately from the core business logic of the main method offers a more cohesive main method, thus preventing tangling of other concerns in the main method.

In our implementation we employ an around advice. If proper parameters are not fed by the user, first correct usage is printed out, and then program control is transferred to the main method with default parameters.

An AspectJ code example of this idiom is shown in Figure 14.

```
public pointcut captureMain(String[] args) :
      execution(void *.main(String[])) && args(args);


void around(String[] args) : captureMain(args) {
    if (args.length < 2 || args.length > 3) {
        args = new String[] { "someString", "anotherString" };
        System.err.println(CORRECTUSAGE);
    }
    proceed(args);
}
```

Figure 14. Idiom for Validating Parameters Passed to a Main Method

4.3. Enforcement Concerns

Asking simulation modelers to comply with a set of constraints is a very noticeable tip that tells us to employ enforcement aspects. Using enforcement aspects, we not only get a more robust design but also get rid of the tangling concerns we would have otherwise.

4.3.1. Imposing Constraints on Users for State Variables

Each state variable in Simkit models is implemented by an instance variable with protected access and should have public getter methods but setter methods, so that only events can change state variables by calls to firePropertyChange(propertyName, oldValue, newValue) methods. However, state variables are still susceptible to illegal modifications inside the classes in which they are declared or by setter methods defined by careless simulation modelers.

For the first Simkit usage constraint, where the users are asked not to define setter methods in order to prevent illegal state variable sets, we offer a solution in which first we define a pointcut that locates undesired set accesses and then by using a static declaration we notify the user by a compilation warning.

An AspectJ code example of this approach is demonstrated in Figure 15.

```
pointcut illegalSetAccesses():
    ! within(somePackage) && set(protected * someClass.someVariable)
    && !(withincode(public void aMethod()) ||
         withincode(public void anotherMethod(..))) ;

declare error : illegalSetAccesses() : "ERROR MESSAGE ";
```

Figure 15. AspectJ Code for Restricting Set Accesses to State Variables

Another constraint on simulation modelers with regard to state variables is as follows: For the first arguments of each property change method, there must be a matching state variable declaration.

For this concern, we offer a solution that makes use of Java Collection feature together with aspects. We use Java Reflection to keep a record of all the declared state variables. Then, we employ a pointcut to get the first argument of each property change methods and by a before advice we compare this argument with the previously kept record of state variables.

An AspectJ code example for this solution is demonstrated in Figure 16.

```
public aspect PolicyEnforcementForStateVariables2 {

    List<String> declaredStateVariables = new ArrayList<String>();

    pointcut locateASimulationClass():
        execution(simkit.SimEntityBase+.new(..)) && !within(simkit..*);

    before():locateASimulationClass(){
        // Related code:
        // collect context for state variables by Java Reflection
    }

    pointcut propertyChangeMethods(String propertyName):
        call(* simkit.BasicSimEntity.firePropertyChange(String, ..))
        && ! within(simkit..*) && args(propertyName, ..);

    before(String propertyName) :
        propertyChangeMethods(propertyName) {
        if (!declaredStateVariables.contains(propertyName)) {
            System.err.println("ERROR MESSAGE ");
            simkit.Schedule.stopSimulation();
            System.exit(1);
        }
    }
}
```

Figure 16. AspectJ for Checking State Variables Declarations

4.3.2. Imposing Constraints on User for Calling "do" Methods

Each event in a Simkit model is implemented as user-defined "do" methods. And events are scheduled by calls to waitDelay(string, delay) functions. Simkit uses Java's reflection to determine the corresponding method for scheduled events. Normally, the programmer does not have to deal with this method. The Simkit library implements a method to invoke the related "do" method indicated by the programmer using waitDelay(string, delay) function.

First of all, when there do not exist corresponding instance methods, for example, such as doRun() and

doArrival() for events such as Run and Arrival respectively, the simulation will run incorrectly. This problem might be caused by either misnaming of event names given as arguments in the waitDelay(string, delay) constructs or by misnaming in the event handling methods.

For this concern, we propose a solution which ensures that first parameters of each event scheduling methods such as waitDelay(eventName, delay) matches with the name of an event handling method without the 'do' prefix, such as doEventName(). We again employ the methodology of using Java Reflection together with Aspects for this implementation.

An AspectJ code example for this solution is demonstrated in Figure 17.

```
public aspect PolicyEnforcementForEventHandling perthis(locateASimulationClass()){

    List<String> declaredDoMethods = new ArrayList<String>();

    pointcut locateASimulationClass():
        execution(simkit.SimEntityBase+.new(..)) && !within(simkit..*);

    before():locateASimulationClass(){
        // Related code:
        // collect context for declared event handling methodsby Java Reflection
    }

    pointcut schedulingEventMethods(String scheduledEventName):
        call(public simkit.SimEvent simkit.SimEntityBase+.waitDelay(String, ..))
        && ! within(simkit..*) && args(scheduledEventName, ..);

    before(String schedulingEventName) :
        schedulingEventMethods(schedulingEventName) {
        if (!declaredDoMethods.contains(schedulingEventName)) {
            System.err.println("ERROR MESSAGE ");
            simkit.Schedule.stopSimulation();
            System.exit(1);
        }
    }
}
```

Figure 17. AspectJ Code for Enforcing Policies on "do" Methods

Secondly, there is no restriction which prevents calling the "do" methods by the programmer. Invoke "do" methods outside the Simkit library may result in changes in state variables and event list causing the simulation to run incorrectly. Therefore, an enforcement aspect which prevents calling "do" methods by the programmer is required for these conditions. To do this, first the points where a "do" method is called outside the Simkit library should be identified by a pointcut. Then, these types of calls are prevented by declaring error at this pointcut.

An AspectJ code example of this approach is demonstrated in Figure 18.

```
pointcut illegalCallToFirePropertyChange():
    ! within(simkit..*) && within(simkit.SimEntityBase+)
    && call(public void firePropertyChange(..));


declare error : illegalCallToFirePropertyChange()
    : "ERROR MESSAGE";
```

Figure 18. AspectJ Code for calling firePropertyChange

4.3.3. Imposing Constraints on User for Firing Property Change Methods

As described before, changes in values of state variables are monitored by a core aspect. This aspect invokes corresponding fire property change functions. Calling these functions by the programmer cause execution of these functions more than one time may result in erroneous behavior of the simulation or inaccurate statistics. Therefore, the library needs an enforcement aspect which ensures that the fire property change functions are not invoked by the programmer.

To prevent calling firePropertyChange methods, first the points where a firePropertyChange method is called should be identified. Finally, these types of calls are prevented by declaring error at this pointcut.

An AspectJ code example of this approach is demonstrated in Figure 19.

```
pointcut illegalCallToDoFunctions():
    ! within(simkit..*) && call(public void do*(..));

declare error : illegalCallToDoFunctions() :
    "ERROR MESSAGE";
```

Figure 19. AspectJ Code for Calling "do" Methods

5. Discussion

In this paper, we have tried to demonstrate mostly simulation systems specific concerns that are crosscutting/tangling and tried to develop aspects for an Event Graph based DES tool i.e. Simkit.

There are a number of observations that deserves noting in our study.

First of them is, some concerns such as simulation termination rules, restoring a simulation run and creating random number generators are crosscutting over multiple modules inherently for simulation systems. This is mostly due to the complexities associated with these concerns. For instance, in order to restore a simulation run, at least three essential properties of a simulation run need to be remembered. These properties are; the event list, the simulation's state and the random number generation objects. Since each of these features are usually implemented in separate components in a coherently and loosely coupled way, we are to interact with all of these modules to restore a simulation run.

Secondly, we have noticed that, using a high level programming language as a simulation tool that does not offer a graphical user interface, imposes many constraints on simulation tools users. Policies the modelers are asked to comply when creating event handling methods, scheduling events, declaring state variables and firing property change events are just a few to name to give examples. Such concerns can be overcome by enforcement aspects efficiently while reducing the tangling issues.

Moreover, our observation is that when working on legacy code for refactoring purposes, the feature of using Java Reflection feature and aspects in a combination might offer a convenient solution. The justification that lies behind this assertion is as such: (1) It may turn out to be very costly in terms of time to concoct other solutions. Usually, delving into somebody else's code to understand the design and provide a solution requires a lot of time. (2) Even when there can be found AOSD solutions, crosscutting modules need to be identified by code mining. Instead, using both Java Reflection and aspects, we can find solutions to some problems, working only on one module or class. We have used this idiom successfully in two enforcement aspects developed for the Simkit.

Finally, one striking question comes to mind. What if we would like to implement all these concerns with another AOSD framework other than AspectJ. In order to make a comparison, we choose JBoss as another framework and focus on how aspects are implemented in these frameworks.

We can begin with pointcuts which are the heart of any AOP implementation. JBoss supports pointcuts defined in XML and by using Java 5 annotations. On the other hand, AspectJ supports language based pointcuts and Java 5 based annotations. AspectJ and JBoss become quite a bit different when it comes to the pointcut syntax. However, expression power seems to be similar.

Another point to discuss is joinpoints. JBoss operates joinpoints as much more like an event framework while AspectJ brings a new approach which is a little hard to learn for beginners. For java programmers, JBoss AOP is easier to implement any joinpoint.

Finally, we compare their build and deployment sides. It seems that it is simpler to make deployment in AspectJ; just compile your code with AspectJ compiler and put aspectjrt.jar in your classpath and you are good to go. This is true regardless of whether you're developing a Java application or a Swing application. JBoss AOP deployment is a little more involved. The simplest means of deploying an aspect is to put the jboss-aop.xml in the META-INF folder of the Jar. However, if you are using an annotation, it needs extra configurations for the system. As a conclusion, AspectJ is handier for building and deployment of the system.

5. Conclusion

Simkit is a simulation modeling tool and like many simulation systems, it implements a number of concerns such as event scheduling, event handling, keeping a track of simulation's state and creating random number generators. It appears that these concerns crosscut over multiple modules in the system. This increases the complexity and reduce the maintainability and ease of use of this tool. In this paper, we have identified the crosscutting concerns in Simkit and tried to refactor it with Aspect Oriented Software Design.

For the results of our study, first of all, we have proved one more time that using only object oriented paradigm is not sufficient to manage all the concerns, i.e. the crosscutting concerns.

Secondly, we have shown that, concerns such as simulation termination rules, restoring a simulation run (persistence) and creating random number generators are inherently crosscutting for simulation systems.

Thirdly, we have shown that, using a high level programming language that does not offer a graphical user interface, imposes many constraints on simulation tools users. Such concerns can be overcome by enforcement aspects smoothly.

Finally, we demonstrated that when working on legacy code for refactoring purposes, using Java Reflection and aspects hand in hand might prove to be very convenient and less costly. Thus it can be adopted as an idiom.

We leave it as a future work to explore the usability and efficacy of such an idiom

6. Acknowledgements

The authors would like to thank to Asst. Prof. Dr. Bedir Tekinerdogan, who has organized the "Third Turkish Aspect-Oriented Software Development Workshop", for his endless motivation in teaching us Aspect Oriented Programming in a graduate course at Bilkent University and generously sharing his comments with us to support writing this paper.

7. References

[1] R.E. Shannon, Systems Simulation – The Art and Science, Prentice Hall, Englewood Cliffs, 1975.

[2] B. Page, W Kreutzer, The Java Simulation Handbook – Simulating Discrete Event Systems with UML and Java, Shaker Verlag, Aachen, 2005.

[3] A. Boss, "Basic Event Graph Modeling", Simulation News Europe, ARGESIM and ASIM, Germany/Austria, April 2001.

[4] A. Boss, "Discrete Event Programming with Simkit", Simulation News Europe, ARGESIM and ASIM, Germany/Austria, November 2001.

[5] R. Miles, AspectJ Cookbook, O'Reilly, Cambridge, December 2005.