Verification of the DEVS Model Implementation using Aspect Embedded DEVS

Jong Hyuk Byun
KAIST, 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, Republic of Korea
jhbyun@smslab.kaist.ac.kr

Chang Beom Choi
KAIST, 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, Republic of Korea
cbchoi@smslab.kaist.ac.kr

Tag Gon Kim
KAIST, 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, Republic of Korea
tkim@ee.kaist.ac.kr

Keywords: Discrete Event Simulator Verification, Aspect Oriented Programming based Verification, DEVS Formalism

Abstract

Discrete event simulators are getting more complex due to a variety of requirements that include desired functions and properties. Therefore, verifying and testing the discrete event simulator is a complicated problem. Moreover, if the simulator is already implemented, inserting and deleting tracing/logging code into the source code of a simulator may cause the code tangling and the code scattering problems. This paper proposes the Aspect embedded DEVS Verification Framework. The Aspect embedded DEVS Verification Framework utilize Aspect Oriented Programming features to handle the code tangling and the code scattering problems. By applying aspect oriented feature, user can find and fix the inconsistency between requirement and implementation of simulator without suffering the code tangling and the code scattering problems. The first step of the verification process is verifying DEVS diagrams to show that the DEVS diagrams are satisfied with the requirement specifications. The second step is checking the event traces from the simulator compared with the possible execution sequences from DEVS diagrams.

1. INTRODUCTION

Recently, discrete event systems, such as computer/communication networks and manufacturing systems are getting more and more complex due to a variety of requirements which include desired functions/properties. Accordingly, the design of such systems has become much complicated, and hard to check satisfaction of the user requirements written in a natural language. In the development process, a user writes down requirement specifications in a natural language, and a software engineer draws the UML diagram based on the requirement specifications. Then, a modeling expert develops DEVS models from UML diagram, and implements DEVS models into a simulator. To check that the simulator satisfies the user requirements, the verification process must show that the DEVS diagrams satisfy the user requirements and the simulator satisfies the verified DEVS models. In order to check the consistency of implementation, the user should inject tracing/logging code into the implementation. However, this approach may cause several problems, such as code tangling and code scattering. Code tangling happens when the several concerns, which are particular area of interest in an application such as tracing/logging, authentication, caching, transaction, and so on, are in the module of a system at the same time. Due to the implementation of each concerns, too many code snippets will constitute the module of the system, therefore, the code of modules gets complicated. Code scattering occurs when the implementation of a crosscutting concern spreads across though several modules. The typical type of code scattering is the code duplication. Since crosscutting concerns are distributed to whole system, the code snippet will be scattered all over the application and it increases the code complexity.

Code tangling and code scattering problems have following negative effects in the simulator verification. First, the core modules of the application and tracing/logging module combine to constitute the application and the code of logic will be scattered all over the application. Consequently, tester should understand the logic of source code to insert the tracing/logging code. Second, inserting and removing tracing/logging code from implementation of the module can make new bugs. To handle these problems, the Aspect-Oriented Programming (AOP) technique has proposed [1].

This paper proposes Aspect embedded DEVS Verification (ADV) Framework and a simulator verificiation process using the ADV Framework. The ADV Framework utilizes the aspect oriented programming techniques to verify the feasible traces without modifying source code of the simulator by monitoring behavior of a simulator at runtime. By applying aspect-oriented feature, the user can find and fix the inconsistency between the requirement and the implementation of a simulator without suffering the code tangling and the code scattering problems. The first step of the verification process is verifying DEVS diagram using VeriTool[2], which verifies the DEVS diagram showing that the DEVS diagrams are satisfied with the requirement specification. The second step is checking the event traces from the simulator compared with the possible execution sequences from DEVS diagrams.

The rest of this paper is organized as follows. Section two present related work. In section three, we explain the

ADV Framework. Section four introduces the verification process of the framework and section five presents case study. Finally, we conclude this paper.

2. RELATED WORK

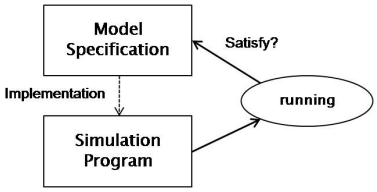


Figure 1. Problem definition

Figure 1 shows the problem definition of verification of the DEVS model implementation. Generally, a user implements the simulation program based on the model specifications. However, the implementation of the simulation program may be different from the model specifications. To verify the simulation program, the user should insert tracing/logging code into the source code of the simulator. Then the user executes the simulator and checks the satisfactions with the model specifications. After the verification of the simulator, the user should remove the tracing/logging code from the source code of the simulator. These inserting and removing code processes may lead to another problem.

To the authors' best knowledge, there are no framework and supporting tools to verify the discrete event simulator. However, some methods and Tools were reported in verification of general-purpose software. JASS (Java with ASSertion)[3] is a precompiler that supports boolean assertions for Java. JASS takes Java source code and inserts pre/post conditions for methods and invariants for classes in a special comment. The Java Run-time Timing constraint Monitor (JRTM) [4] aims to detect violation of timing properties in Java programs. JRTM uses Real-Time Logic (RTL)[5] as a requirement specification language. A Java program should be manually instrumented to put a probe in the place where a primitive event happens. Temporal Rover [6] monitors Java/C++ programs to check whether LTL requirement specification is violated. Probes are inserted into source code manually. These tools may suffer the code tangling and the code scattering problems while inserting monitor code and deleting monitor code. Moreover, writing requirement specifications of these Tool is hard to learn, and not sufficient to express discrete event system simulators.

3. BACKGROUND

The verification of simulator is to check the satisfactions of implementation to the requirement specification, which provided as DEVS diagrams. Therefore, the simulator execution follows the DEVS formalism, and to verify implementation of a simulator, one should collect simulation information during the execution of the simulator. The proposed framework utilizes AOP.

3.1. DEVS Formalism

DEVS (Discrete Event System Specification) is a set-theoretic formalism developed for specifying discrete event systems [7, 8, 9]. In the DEVS formalism, one can specify basic models and how these models are coupled in a hierarchical and modular fashion. A basic model, called an atomic model, specifies the dynamics of a model and is defined as Atomic Model and Coupled Model. In the ADV Framework, the modeling expert will draw Atomic Model and Coupled Model diagram to express requirement specification.

3.1.1. Atomic Model

An Atomic Model is defined as follow:

$$AM = < X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta >$$

where

X: a set of external input event types,
Y: an output set,
S: a sequential state set,
δ_{ext}: $Q \times X \rightarrow S$, an external transition function
where Q is the total state set of
$M = \{(s,e) | s \in S \, and \, 0 \leq e \leq ta(s)\}$,
δ_{int}: $S \rightarrow S$, an internal transition function,
λ: $S \rightarrow Y$, an output function,
ta: $S \rightarrow \mathcal{R}^{+}_{0,\infty}$, a time advance function, where the $\mathcal{R}^{+}_{0,\infty}$
is the non-negative real numbers with ∞ adjoined.

An atomic model AM is a model which is affected by external input events X and which in turn generates output events Y. The state set S represents the unique description of the model. The internal transition function δ_{int} and the external transition function δ_{ext} compute the next state of the model. If an external event arrives at the elapsed time e which is less than or equal to $ta(s)$ specified by the time advance function ta, a new state s' is computed by the external transition function δ_{ext}. Then, a new $ta(s')$ is computed, and the elapsed time e is set to zero. Otherwise, an internal event arrives at $ta(s)$, and then a new state s' is computed by the internal transition function δ_{int}. In the case of internal events, the output specified by the output function λ is produced based on the state s, which means the output function is processed before the internal transition function. Then, as before, a new $ta(s')$ is computed, and the elapsed time e is set to zero.

3.1.2. Coupled Model

Coupled Model is defined as follow:

$$CM = < D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, SELECT >$$

where

D: a set of component names,
For each i in D,
M_i: a component basic model
(an atomic or coupled model),
I_i: a set of influences of i,
and for each j in I_i,
$Z_{i,j}$: $Y_i \rightarrow X_j$, an i-to-j output translation,
$SELECT$: $2^M - \phi \rightarrow M$, a tie-breaking selector.

A coupled model CM consists of components $\{M_i\}$, which are atomic models and/or coupled models. The influences $\{I_i\}$ and i-to-j output translations $\{Z_{i,j}\}$ define the coupling specification as follows: An external input coupling (*EIC*) connects the input event of the coupled model to the input event of one of its components; An external output coupling (*EOC*) connects the output event of a component to the output event of the coupled model; An internal coupling (*IC*) connects the output event of a component to the input event of another component. The *SELECT* function is used to order the processing of simultaneous internal events for sequential simulation. Thus, all the events with the same time in a system can be ordered by this function.

3.1.3. Simulation Environment

The DEVSim++[10] is a DEVS simulation environment based on C++, which is integrated with the Microsoft Visual Studio.NET. It therefore provides the advantages of object-oriented framework, such as encapsulation, inheritance, and reuse. The DEVSim++ coordinates the event schedules of atomic models in a system and provides classes and APIs for simulation.

We get several advantages with the DEVS formalism and DEVSim++ to verify implementation of DEVS diagrams. First, we can specify the simulator models mathematically and easily verify the model. Second, we can model hierarchical and modularized systems, which enhance understandability and extensibility. Third, we can reuse simulation models.

3.2. Aspect Oriented Programming

The objective of the Aspect-Oriented technique is to modularize the crosscutting concerns, which cause the code tangling and code scattering in a system. AOP has several benefits, such as higher modularization, easier system evolution, late binding of design decisions, more code reuse and reduced costs of feature implementation. The AOP language implementation performs two logical steps: First process is called weaving and the processor for this process is called weaver. The weaver merges the individual implemented concerns into the target code using the weaving rules. Then, it converts the resulting information into executable code. In weaving rules, there are several anatomies to define the aspects. The Aspect-Oriented programming is performed using the AOP language, which is usually built as a language extension on the top of one of the traditional functional, imperative or OOP languages [11]. In this paper we use AspectC++[12], in order to interoperate with DEVSim++.

3.2.1. Join points

Aspects interact with a base application during run-time at well-defined interaction points. In addition, these points are called as join points. The set of join points can be described as pointcut. AspectC++ supports a number of different types of join points within a program, such as:

- Method call and execution
- Constructor call and execution
- Initialization of classes and objects
- Execution of an exception handler

At these points, AspectC++ may weave the additional code to implement crosscutting concerns.

3.2.2. Pointcuts

Pointcut expressions are composed from matched expressions used to find a set of join points, to filter or map specific join points from a pointcut, and from algebraic operators used to combine pointcuts.

Followings are pointcut expressions, which used in Aspect C++:

- Type Matching
- Namespace and Class Matching
- Function Matching
- Template Matching

Further information can be found in the AspectC++ site[13].

3.2.3. Advices

The advice is the functional unit of an aspect and it specifies the behavior, what to do at a particular join point. Therefore, each advice is consist of pointcut, which specifies the advice when and where should be applied and the body that executes code. There are three types of advices:

- Before advice
  Before advice specifies the behavior, and it should be executed before the join point is invoked.

- After advice
  After advice specifies the behavior, and it should be executed after the join point is invoked. After advice have three types: after returning advice, after throwing advice and after advice. After returning advice is that executed after the successful completion of a pointcut call. After throwing advice is executed after the join point throws the particular exception. Finally, after advice is executed after any call to the join point, regardless of whether it threw an exception or not.

- Around advice
  Around advice specifies the behavior, and it should be executed around the join point is invoked. This advice can intercept the flow and execute its own code and it can proceed to the original code.

4. ADV FRAMEWORK

In the ADV Framework, we assume that the simulator is already implemented, and the source code of a simulator should not be modified during the testing phase to avoid the code tangling and the code scattering problems.

4.1. ADVeriTool

The ADVeriTool(Aspect embedded DEVS Verification Tool) supports a tester to verify implementation of a simulator. The graphic user interface (GUI) of the ADVeriTool is shown in figure 2.

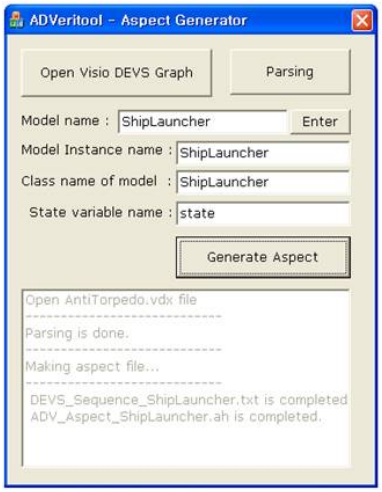


Figure 2. ADVeriTool

The ADVeriTool analyzes the Microsoft Visio XML drawing file format to generate feasible execution traces from verified DEVS graph. Since an Atomic model is consist of the three sets and four functions, ADVeriTool generates feasible execution traces, which consist of symbols of three sets and four functions. Figure 3 shows the type of feasible execution traces from the DEVS graph. Feasible execution traces with label TA represent time constraints that every state must satisfy and label EXT represents input constraint of a state. Similarly, label OUT and INT represent output constraint and constraint of internal transitions.

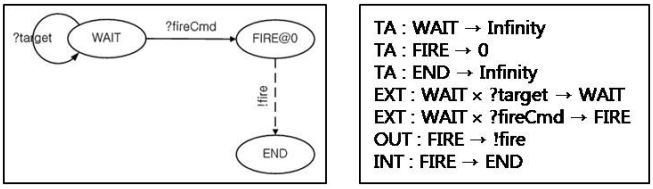


Figure 3. DEVS graph and feasible execution trace

The ADVeriTool extract the following feasible execution traces:

- TA : Time constraint of the state
- OUT : Output constraint of the state
- EXT : Constraint for the external transition
- INT : Constraint for the internal transition

After extracting feasible traces from the DEVS graph, the tester uses the ADVeriTool to generate the aspect file. Since the implementation may use different model name, and several identical model instances, the several information of the model may be different from DEVS graph. Therefore, the user should insert the model name, the model instance name, the class name of a model and the state variable name. In addition, implementation of a model does not restricted to have identical branch condition. As a result, the user might concretize the aspect code, the ADVeriTool generated.

The ADVeriTool generates the Trace Monitor aspect and the Verification Atomic Model. The Trace Monitor aspect extracts information of state changes, input, output, and time advance and condition variables. To collect this information, the Trace Monitor is weaved into the four DEVS functions by around advice. The Verification Model is an Atomic model that checks the time contraint of the state(i.e. TA) and checks the execution event traces(i.e. OUT, EXT, and INT). Since the Verification Model has feasible execution traces, the Verification Model matches every execution event trace to the feasible traces. When simulator terminates, the Verification Model generates two files, classified result and model trace history. The classified result contains the verified trace, not verified trace and violation, and the model trace history contains entire event trace of a simulator. These two files supports tester to find and fix the inconsistency.

4.2. Verification Process

Figure 4 introduces the discrete event simulator development process, and results of each phases.

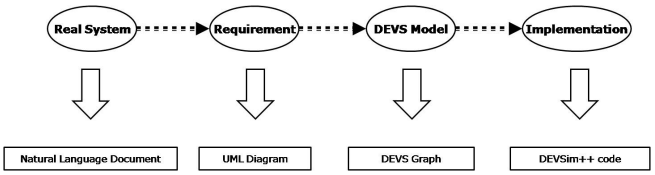


Figure 4. Simulator development process and its results

In the real system analysis phase, the domain engineer writes down the requirement specification in natural language documents. In the requirement analysis phase, the software engineer analyzes the natural language documents and completes corresponding UML diagrams. After the requirement analysis phase, modeling experts analyze the UML diagrams and come up with the DEVS diagrams, and implement the simulator based on DEVS diagrams. After implementing simulator, tester should verify that the simulator satisfies the requirement specifications. Figure 5 shows the verification process using the ADVeriTool.

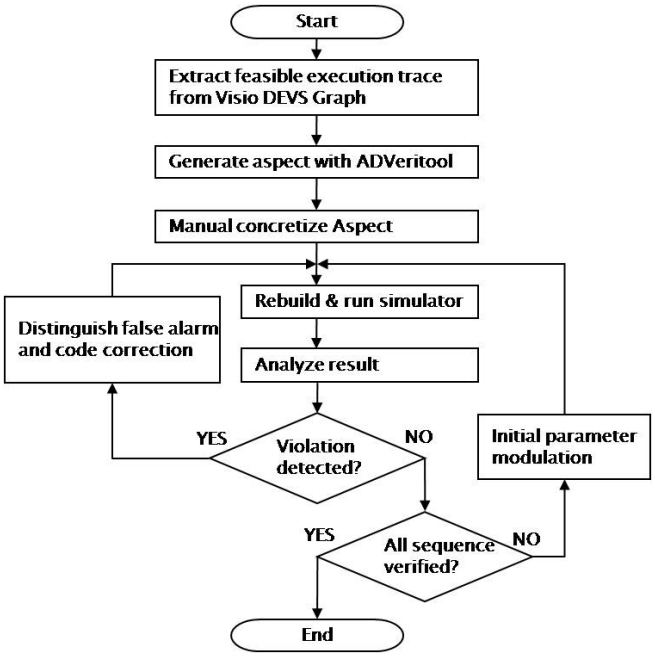
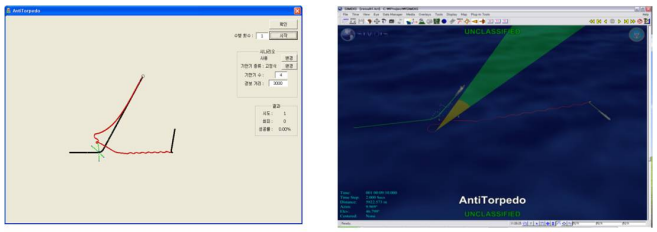


Figure 5. Proposed verification process

The ADVeriTool extracts feasible execution traces from the Visio DEVS graph, and it generates the Trace Monitor aspect and the Verification Model by adding additional information for the model implementation. After generating the aspect and the model, the tester should concretize the aspect, manually. After finishing concretization, tester rebuilds and run simulator. Then, the Verification Model generates classified results and the model trace history. Since the classified results contain the summary of the verification results, the tester should analyze the classified results. If the violation is detected, then the tester should determine the violation as a bug or a false alarm. If the violation is a false alarm, then the tester should match the verified or not verified trace, and if the violation is a bug, the tester refers the model trace history files to find and fix the bug. These processes are repeated until all sequences are verified and no violation is detected. If no feasible traces are verified, then tester should change the initial parameter of the simulator to cover all feasible traces. After every feasible trace is verified, the verification process terminates.

5. CASE STUDIES

This section describes verification results of the simulator of anti-torpedo combat system and the term projects of Discrete Event System Modeling and Simulation lecture[14] as case studies. The main objective of anti-torpedo combat simulator is to measure the effectiveness of the anti-torpedo combat system, and to find effects of some tactical or technical parameters of anti-torpedo combat system. The anti-torpedo combat simulator is composed of five components: Submarine model, Torpedo model, Ship model, Decoy model and Battle control model. GUI was also developed for editing scenarios, as a backend, and showing the simulation results, as shown in figure 6. After the termination of the simulator, it generates simulation results files in SIMDIS format and simulation results are displayed by SIMDIS. SIMIDIS[15] is a set of software tools that provide two and three-dimensional interactive graphical and video display of live and post processed simulation. The development period of this simulator is one year and source line of codes of this simulator are about 8000 lines, and this simulator is consist of 24 classes.



Back end Front end

Figure 6. Anti-torpedo combat system GUI

To verified the atomic model of the simulator, we have developed the DEVS graph for each atomic model, as shown in figure 7.

Table 1 shows the first trial verification results of the anti-torpedo combat system simulator using the ADV Framework.

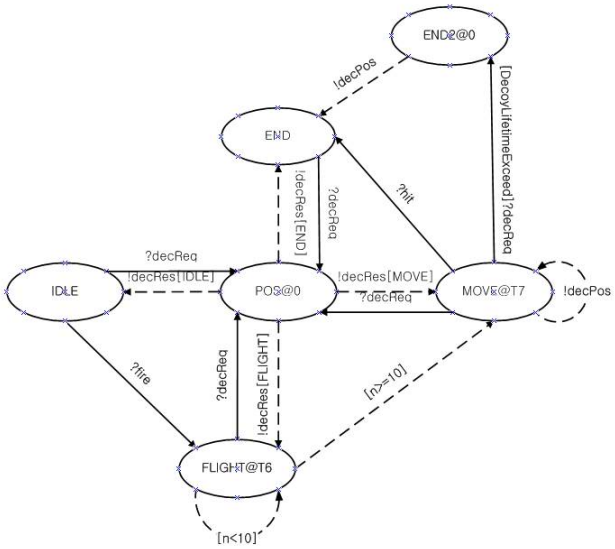


Figure 7. DEVS Graph of Atomic Model $Effector$

Note that the F/A denotes False Alarm. The coverage ratio is the ratio that the number of detected specified traces divided by total feasible traces.

Total nine violations are found and among them, two false alarms have been found. The rest of violations was faults, that the implementer did not follow the requirement specifications. After detecting the violations, implementer corrected the faults by referring the classified results and the model trace history. Then, we tested the same the initial parameter and found all violation has been removed. However, the initial parameter of the first trial did not cover all the feasible traces, we changed the initial parameters and found that the simulator covers all the feasible traces.

Table 1. Verification results of the anti-torpedo combat simulator

Atomic Model	States	Coverate ratio	Violations	F/A
PfMove	5	13/14(92.86%)	0	0
PfRadar	4	21/23 (91.3%)	0	0
PfFire	3	7/7 (100%)	0	0
ShipMove	3	9/10 (90%)	0	0
ShipRadar	4	11/11 (100%)	1	0
ShipLauncher	2	7/7 (100%)	0	0
TpdMove	5	16/18 (88.89%)	2	0
TpdRadar	5	15/15 (100%)	2	0
FloatingDecoy	6	21/24 (87.5%)	4	2
BattleControl	2	6/6 (100%)	0	0
Transducer	2	5/5 (100%)	0	0
Total(11)	41	86.43%	9	2

We used the ADV Fraework to verify the term projects of Discrete Event System Modeling and Simulation lecture in 2007. In this lecture, the students implement a simulator with their own domain field. Since the lecture is over, we cannot classify false alarms from violations. Table 2 shows the first trial verification results of term projects. However, the verification results of first trial shows that the coverage ratio is over 75%, and the ADV Framework detects violations easily.

Table 2. Verification results of term projects

Atomic Model	States	Coverate ratio	Violations
IF_SEL	2	7/8 (85.7%)	2
Process	4	9/9 (100%)	0
BN_Processor	12	13/22 (59%)	10
DtCat	5	12/16 (75%)	2
S	4	10/13 (76.9%)	0
Processor	4	9/9 (100%)	0
Receiver_Model	5	13/13 (100%)	0
Cache	10	128/32 (87.5%)	2

6. CONCLUSION

This paper describes the Aspect embedded DEVS Verification (ADV) Framework and its supporting tool ADVeriTool. The ADV Framework bridges the gap between verification of simulator design specifications and validation of simulator implementations. By taking advantages of aspect oriented programming technique, ADV Framework can find and fix the inconsistency easily. Moreover, ADV Framework does not affect to the simulator source code, and provides automatic aspect generation tools. Furthermore, this technique can be applied to all kinds of different simulators regardless of developers.

We have applied the ADV Framework successfully to the Anti-torpedo combat system. We are investigating other simulators. We are also extending the ADV Framework to verify the coupling relation of Coupled Model and researching automatic initial parameter generation.

REFERENCES

[1] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[2] D. H. Kim, "Method and implementation for consistency verification of devs model against user requirement," Master's thesis, Dept. of Electrical Eng., KAIST, 2005.

[3] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim, "Jass-java with assertions," in *Electronic Notes in Theoretical Computer Science*, 2001, pp. 103 – 117.

[4] A. K. Mok and G. Liu, "Early detection of timing constraint violation at runtime," in *Electronic Notes in Theoretical Computer Science*, Dec 1997, pp. 176 – 186.

[5] F. Jahanian and A. Goyal, "A formalism for monitoring real-time constraints at run-time," in *20th International Symposium on Fault-Tolerant Computing Systems (FTCS-20)*, Jun 1990, pp. 148 – 155.

[6] A. K. Mok and G. Liu, "The temporal rover and the atg rover," in *Proceedings of 7th International SPIN Workshop, LNCS 1885*, Dec 2000, pp. 323 – 329.

[7] B. P. Zeigler, *Multifacetted Modeling and Discrete Event Simulation*. Academic Press, 1984.

[8] B. P. Zeigler, *Object-Oriented Simulation with Hierarchical, Modular Models*. Academic Press, 1990.

[9] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*. Orlando, FL, USA: Academic Press, Inc., 2000.

[10] T. G. Kim, *DEVSimHLA User's Manual*, 2007. [Online]. Available: http://smslab.kaist.ac.kr

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *European Conference on Object-Oriented Programming*, Oct 2001, pp. 327 – 353.

[12] A. Gal, W. Schroder-Preikschat, and O. Spinczyk, "Aspectc++: Language proposal and prototype implementation," in *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Oct 2001, pp. 20 – 36.

[13] O. Spinczyk, *AspectC++ language referenceManual*, 2005. [Online]. Available: http://www.aspectc.org

[14] [Online]. Available: http://smslab.kaist.ac.kr/Course/EE612/

[15] [Online]. Available: https://simdis.nrl.navy.mil/