Advanced Science and Technology Letters
Vol.107 (Ubiquitous Science and Engineering 2015), pp.52-56
http://dx.doi.org/10.14257/astl.2015.107.13

Development of Concurrent Programs based on a Modeling and Simulation Formalism

Junghyun Im[1], Ha-ryung Oh[1], Yeong Rak Seong[2]

[1]Dept. of Secured-Smart Electric Vehicle, Kookmin University
77 Jeongneung-ro, Seongbuk-gu, Seoul 136-702, KOREA
ijh227@kookmin.ac.kr
[2]Dept. of Electric Engineering, Kookmin University
77 Jeongneung-ro, Seongbuk-gu, Seoul 136-702, KOREA
Corresponding Author: yeong@kookmin.ac.kr

Abstract. In this paper, we address how to develop a concurrent program using the discrete event system specification formalism. This paper mainly focuses on how to convert a discrete event model of a concurrent program to a multi-threaded program code written in a conventional programming language. In addition, this paper also suggests an efficient solution for the over-generation of threads.

Keywords: Modeling and Simulation, Concurrent Software, Multi-thread Software, Discrete Event System

1 Introduction

This paper suggests a framework for developing a software program with concurrency using the discrete event system specification (DEVS) formalism [1, 2]. In the proposed framework, a concurrent program is modeled and simulated by using the DEVS formalism and the DEVS abstract simulator algorithm [3]. Then the modeling result is translated to conventional programming language codes which can be compiled and executed in an actual environment through a simple conversion process. A naive version of the suggested framework was introduced in [4] and applied in the development of a navigation application. This paper describes the framework more specifically by focusing on the framework itself. Also, a solution is suggested to efficiently prevent the over-generation of threads.

2 Proposed Framework

In the DEVS formalism, a discrete event system is represented by a hierarchical or modular manner. Therefore, the modeling result is represented as a tree structure. A leaf node of the tree represents the behavior of a component; it is called the *atomic model*. Additionally, an internal node in the tree connects many components to form a large component. This is called the *coupled model*. This coupled model can be a component of a larger coupled model.

The DEVS abstract simulator is suggested for the simulation of DEVS models. It includes a *simulator* for atomic models, a *coordinator* for coupled models, and a *root coordinator* for controlling the entire simulation process. DEVSim++ [5, 6] is a DEVS abstract simulator environment based on the C++ language.

To simulate a DEVS model, a model tree should be converted to an abstract simulator tree. A simulator is placed at an atomic model node and a coordinator is placed at the coupled model node of the model tree. Each abstract simulator is connected to the corresponding atomic/coupled model. In addition, a root coordinator is placed over the top coordinator. A simulation starts with the root coordinator delivering the first message to the top coordinator via a link in the tree. Whenever an abstract simulator that receives a message, it requests the knowledge that is required to process the message to the associated DEVS models. According to the response of the DEVS models, the abstract simulator generates new messages and send them to other abstract simulators. By repeating these action, the simulation proceeds.

The DEVSim++ simulation code can be converted to an actual program code in various ways. This study implements the program code so that the software design result represented by the DEVSim++ simulation code can be maintained as much as possible. This implementation is performed with the knowledge that the behavior of the designed software appears only to atomic models in the DEVS formalism. The implementation phase consists of three parts. It 1) implements a pair of atomic model-simulators as an independent thread; 2) implements the connection information between atomic models stored in all coupled model-coordinator pairs into the port mapping table; and 3) implements the scheduling function included in the root coordinator and coupled model-coordinator pairs into one scheduler thread.

2.1 Atomic thread

A pair of an atomic model and simulator is implemented as a single thread, called an *atomic thread*. In this paper, to maintain the software design result included in DEVSim++ codes as much as possible, the atomic model code is scarcely changed, but the simulator part is entirely re-coded as the control code of the atomic thread.

Fig. 1 shows the simplified architecture of an atomic thread. Once the atomic thread receives a message from other threads, the message is delivered to the control code. The message is classified into two types: the messages transmitted from another atomic thread, and the messages transmitted from the scheduler thread. If the delivered message is transmitted from another atomic thread, the control code calls the external transition function and changes the status of the atomic model. It then calls the time advanced function and delivers the new schedule time to the scheduler thread. Meanwhile, if the delivered message is transmitted from the scheduler thread, the control code calls the output function to produce output messages, which will be transmitted to other atomic threads, and the internal transition function to change the status of the atomic model. It finally calls the time advanced function and delivers the new schedule time to the scheduler thread.

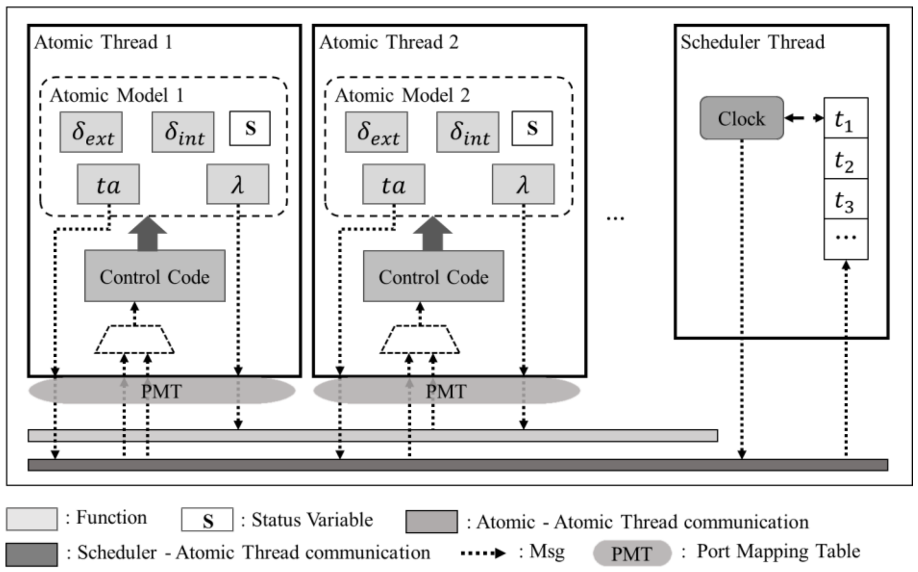

Fig. 1. Structure of the program implemented by the proposed framework

2.2 Port mapping table

The port mapping table is implemented by referencing the connection information between the models described in the coupled model-coordinator pairs. The port mapping table plays a role in connecting the input/output ports of the atomic threads. It therefore requires information on the connection between the input/output ports and information on the placement of input/output ports of atomic models. Because the modeling result using the DEVS formalism is a hierarchical structure, it requires a complex operation to extract specific information from the connection information separately stored in the many coupled models. To simplify these processes, the link information is gathered by flattening the modeling result. Furthermore, the information on the corresponding atomic thread of each input/output port can be obtained from the atomic DEVS models.

2.3 Scheduler thread

The scheduler thread generates a schedule message that activates the actions of the atomic threads. This scheduler thread has a scheduling function that is included in the root coordinator and coupled model-coordinator pairs of DEVSim++. For this function, the scheduler thread consists of a schedule time table and a timer. The scheduler thread transmits the schedule time whenever the status of the atomic threads changes and it updates the schedule time table. In addition, if the timer interval indicating the time progress matches the value of the schedule time table, it generates a schedule message and delivers the message to the corresponding atomic thread so that the atomic thread can be activated.

2.4 Thread population tuning

Depending on the modellers' expertise or perspective, a system can be modeled various ways even with the same modeling and simulation technique. Therefore, many atomic models may be generated in some cases. As explained earlier, if one atomic model is implemented in one atomic thread, too many threads may be initiated. This increases the context switching overhead and negatively influences the performance of the software. However, it is not appropriate to limit the number of atomic models in modeling.

This paper suggests a method for implementing the thread without over-generating them, while securing modeling freedom. Although there are many methods to adjust the number of threads, this paper suggests a method that implements many atomic model-simulator pairs in one thread, called a *combined thread*. Regardless of whether an atomic model is implemented by an atomic thread or a combined thread, it would be very favorable if both of the threads can be implemented with the nearly identical code. To this end, the control code of an atomic thread is re-written and modularized into a function called the *entrance function*. Thus, when a combined thread receives a message, the control code of the combined thread invokes an appropriate entrance function after it finds the destination of the message by using the port mapping table.

3 Conclusions

The proposed software development framework applies the DEVS formalism in the design and implementation of concurrent software. This paper describes how to translate a DEVS model of a concurrent program to an actual program code. Each atomic DEVS model is converted to an atomic thread, and the information distributed in coupled DEVS models are collected into a port mapping table. Moreover, we suggested a method to reduce the number of threads. The proposed method thus reduces the context switching overhead caused by thread over-generation, improves the software performance, and enables the development of software that is executable even in an environment with a limited number of threads.

References

1. Zeigler, B.P.: Theory of Modeling and Simulation. John Wiley, New York (1984)
2. Zeigler, B.P.: "DEVS formalism: A framework for hierarchical model development," IEEE Transactions on Software Engineering 2, pp. 228-241 (1988)
3. Zeigler, B.P.: Object-Oriented Simulation with Hierarchical, Modular Models, Academic Press (1990)

4. Kim, Y.H., Seong, Y.R., Oh, H.R.: “Software Development Method Using the Concurrency Control Approach Based on DEVS Simulation,” The 2014 FTRA International Conference on ACS, Vol.11, No.7, pp. 553-558 (2014)
5. Kim, T.G., Park, S. B.: “The DEVS Formalism: Hierarchical Modular Systems Specification in C++,” Proceeding of 1992 European Simulation Multiconference, pp. 152-156 (1992)
6. Kim, T.G.: DEVSim++ User's Manual: C++ Based Simulation with Hierarchical Modular DEVS Models (1994)