Software Development Method Using the Concurrency Control Approach Based on DEVS Simulation

Authors Yun Ho Kim, Yeong Rak Seong, and Ha Ryoung Oh

Department of Electrical Engineering, Kookmin University, Korea
yeong@kookmin.ac.kr

Abstract: Since the test results may differ according to the motion orders of a thread even in the same situation, it is very difficult to develop multithread software. To address this problem, a software development method using a concurrency control approach based on a discrete event simulation is proposed here. Concurrency control refers to the pursuit of a management method to maximize the concurrent execution when it is necessary to handle many tasks underway at the same time, while maintaining the capabilities of the system. However, it would not be cost effective if, when testing the functions of software, changes are made to the thread's concurrency by revising the software directly. This study shows the feasibility of effective concurrency control through the use of simulation. The proposed method, which is demonstrated with the development of navigation software, explains the process of software development.
[Authors Yun Ho Kim, Yeong Rak Seong, and Ha Ryoung Oh. **Software Development Method Using the Concurrency Control Approach Based on DEVS Simulation.** *Life Sci J* 2014;11(7):553-558]. (ISSN:1097-8135). http://www.lifesciencesite.com. 75

Keywords: modeling; simulation; concurrency control; DEVS formalism

1. Introduction

When various concurrent tasks must be performed in a single program, it is effective to realize each task with an individual thread. A thread is a flow of sequential control that is executed independently in a program. A multithread is a type of thread in which various threads are executed concurrently. The execution of a multithread program has the advantage of responding flexibly to events occurring in any order in an external program. However, it is very difficult to develop a multithread program when compared to general single thread programs. The main reason of such phenomenon is that as the various threads in a single program are being on the parallel executions competitively, the motion orders of the thread may differ even by the identical events occurring in identical orders and times.

Because the motion orders of a thread may affect the test results each time, this is a major factor that affects the results of a multithread program test. One of the ways of addressing this problem is to control the concurrency of the threads during execution of the program. In other words, the change of execution orders of a thread in an event sequence is decreased by controlling the concurrency of the threads accordingly. In multithread software, the performance will show no progress if the concurrency of the thread is restricted. Meanwhile, when too much concurrency is allowed, the performance results will lose consistency.

Given this background, one way to ensure the most effective concurrency control is to allow the maximized concurrency within the constraint of not losing the consistency of performance results. However, it is very difficult to control the concurrency of threads for effective operation according to the number of event sequences.

In general, modeling and simulation are used as a design tool to analyze the system behavior before making a complicated system. Modeling and simulation are also used to comprehend the characteristics of a system without realization of the actual system. Modeling and simulation are also used as a tool for optimized design by forecasting the results that might occur in actual situations. Therefore, when developing multithread software, the use of modeling and simulations makes verification of the number of sequences and effective concurrent control possible. Therefore, when developing a multi-threaded program, if the model verified via modelling and simulation is implemented in actual software, then the problem of testing being difficult can be overcome.

As an existing software development technology via modelling, there is MDA(Mukerji and Miller, 2003), in which a platform-independent model is created and then converted to fit the desired platform. Because MDA is based on UML 2.0(Object Management Group, 2005), although it is advantageous for verification of the model itself when it comes to finding grammatical errors in the model or contextual errors such as infinite loops, because for simulation of the system a lot of work is needed for the overall time management of the simulation, and because it is executed only within the case tool, it is rather limited for the purposes of developing a typical simulator. Moreover, to

implement a system that was modelled using UML, time management of the entire simulator has to be included in the modelling, but this is an extremely complex and difficult thing to do, necessitating a lot of room for error, so there are difficulties that the time management of the entire simulator has to be directly programmed during software implementation.

In this paper, for the modelling language the DEVS(Discrete Event System Specification) formalism (Bernard, 1984) and DEVSim++ (Kim, 1994) simulation engine were used in order to resolve the difficulty of the overall time management of the simulation. Also, to address the problem that a verified model has to be implemented again in actual software code, a method of converting simulation code to software code was suggested so that software code could be easily implemented.

This study is organized as follows. Chapter 1 states the purpose and the background of this research. Chapter 2 proposes and explains the concurrency control approach based on a DEVS simulation for software development. In addition, chapter 2 describes the realization processes of software and navigation software modeling and simulation through case studies. Lastly, a conclusion and directions for future research are given in chapter 3.

2. Concurrency Control Approach Based on DEVS Simulation

This paper proposes a method that enables control over the concurrency of threads effectively by using a DEVS modeling simulation. Furthermore, the process of navigation software development is cited as a case study for developing multithread software by applying the proposed method. Navigation software provides basic functions to guide destinations commanded by users, and multitasking functions showing other places. It also offers additional information such as the current location change, speed, and expected time to the final destination through GPS.

The method proposed in this study adheres to the development process of the prototyping model (Randy, 1991) under the Software Development Life Cycle, but modeling and simulation processes replace feedback loops after the prototype is designed and implemented. Software development using DEVS-based concurrency control approach can be largely divided into 4 stages: requirement analysis, discrete event modeling, simulation, and software implementation. This is shown in (Figure 1).

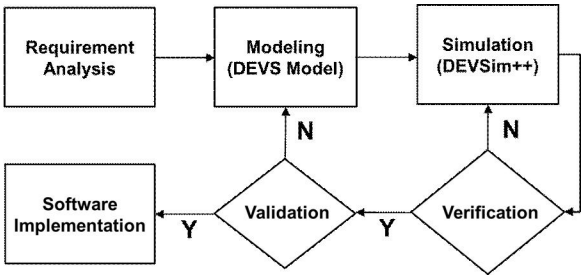


Figure 1. Software development process using concurrency control approach

2.1. Analysis of Requirements

The first stage involves the analysis of system characteristics and user requirements. In this stage, the events and functions required for the software are organized.

The functions and requirements to be implemented in the navigation system are as follows:

1. Processing of asynchronous user input
2. Display of scaled and translated maps
3. Refresh of current location based on periodically updated GPS data
4. Processing and end movement must be simultaneously achieved for each command
5. If various functions are performed simultaneously, the system must measure response lag or incorrect delivery of information

2.2. Modeling

The second stage involves the modeling of software characteristics and requirements from the previous stage into a simulation model. In this study, simulation is used to measure changes in results or system status while varying the priority of event sequences or threads. Thus, a simulation model must be developed ahead of the actual simulation.

The developed system was expressed as a discrete event system before modeling. A discrete event system is a dynamic system in which discrete state variables change according to randomly occurring events. All manmade systems can be considered as discrete event systems. The transition rule of discrete state variables can be expressed using set theory rooted in discrete mathematics (Donald, 1977), and the DEVS formalism is the mathematical framework for expressing discrete event systems based on state equation set theory (Bernard , 2001). The DEVS formalism has the following advantages (Bernard, 1990).

1. Provision of a system theoretical (input, output, state, and state transition) modeling framework
2. Model expression using functions and relations based on set theory

3. Provision of a mathematical foundation for hierarchical modeling following system modularization
4. Provision of abstract simulator algorithm to facilitate systematic, clear development

The DEVS model allows simulation to be easily performed using the DEVSim++ simulator, which is an implementation of the abstract simulator algorithm.

In this section, the DEVS formalism was applied to the modeling of navigation components developed based on system requirements. (Figure 2) is a block diagram of input/output relationships between models constituting the navigation system.

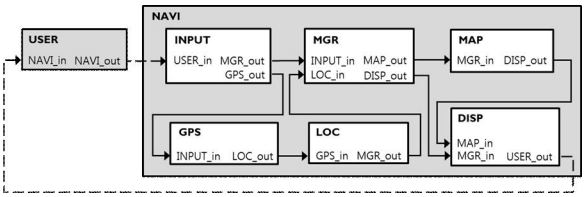


Figure 2. Components of navigation system and input/output relationships

Each model reflects the requirements specified in section 2.1.1. The function described in (1) is implemented in the INPUT model, (2) in the MAP model, (3) in the GPS and LOC models, and (4) in the MGR model. The USER model measures simulation results with consideration of the requirement in (5). The DISP model delivers information provided by the navigation to the user.

The navigation system consists of 7 atomic models and 1 coupled model, but only this paper only covers the DISP model due to space constraints. The DISP model distinguishes between input of the MGR model and the MAP model to enable simultaneous processing of responses to asynchronous user request such as map control and information request, in addition to synchronous location information. (Figure 3) shows the DISP model, modeled by using DEVS formalism.

$DISP = \{ X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \}$

$X = \{ MAP\_in, MGR\_in \}, Y = \{ USER\_out \}$

$S = \{ WAIT, PATH\_DISP, LOC\_DISP, MAP\_DISP, INFO\_DISP \}$

$\delta_{ext}(WAIT, -, MGR\_in) = INFO\_DISP$	$\lambda(PATH\_DISP) = USER\_out$
$\delta_{ext}(WAIT, -, MAP\_in) = MAP\_DISP$	$\lambda(LOC\_DISP) = USER\_out$
$\delta_{ext}(WAIT, -, MAP\_in) = if(cmd==SEL\_PATH)\ PATH\_DISP$	$\lambda(MAP\_DISP) = USER\_out$
$\delta_{ext}(WAIT, -, MAP\_in) = if(cmd==CUR\_LOC)\ LOC\_DISP$	$\lambda(INFO\_DISP) = USER\_out$
$\delta_{int}(PATH\_DISP) = WAIT$	$ta(WAIT) = \infty$
$\delta_{int}(LOC\_DISP) = WAIT$	$ta(PATH\_DISP) = 150$
$\delta_{int}(INFO\_DISP) = WAIT$	$ta(LOC\_DISP) = 120$
$\delta_{int}(INFO\_DISP) = WAIT$	$ta(MAP\_DISP) = 120$
	$ta(INFO\_DISP) = 10$

Figure 3. DISP model applying atomic DEVS model formalism

2.3. Simulation

The third stage involves the simulation of the DEVS model, which has been successfully modeled in the previous stage. In the previous section, the functions provided by the navigation software were each implemented into a model. This section uses the DEVSim++ simulator to simulate user response in accordance with model priority. In addition, changes in user response and operational status of the navigation were examined with varying priority. (Table 1) presents the navigation movement scenario according to user commands. And (Table 2) presents the simulation result of navigation model.

Table 1. Navigation movement scenario according to user commands

User command	Navigation Action
Turn on navigation	Receives GPS data and updates current location
Select destination	Displays path to destination
Select path	Displays distance and expected time to destination
Navigation request	Begin navigation
Scale (zoom in/out) or translate map	Displays scaled or translated map
View current location	Displays current location
Turn off navigation	Ends navigation to destination

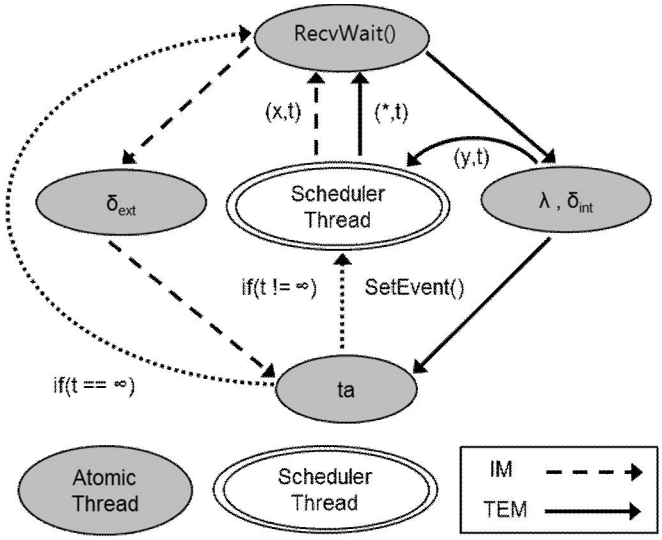
Table 2. Simulation result of navigation model

Case	Thread priority (INPUT–MGR–MAP–GPS–LOC–DISP)	Average waiting time
1	1 – 2 – 3 – 4 – 5 – 6	412 ms
2	2 – 3 – 4 – 5 – 6 – 1	396 ms
3	3 – 4 – 5 – 6 – 1 – 2	453 ms
…	…	…
10	1 – 2 – 4 – 5 – 6 – 3	374 ms (best)
…	…	…
703	6 – 5 – 2 – 1 – 3 – 4	687 ms (worst)
…	…	…
720	6 – 5 – 4 – 3 – 2 – 1	529 ms

2.4. Software Implementation

The final stage implements the actual software based on the model verified through simulation. For coding of the actual navigation software, the verified navigation model must be converted to a software code. Further, the software

code should be verified to ensure that the simulation model has been properly implemented. This paper proposes a method of using the simulation code to get a software code. Since the DEVS abstract simulator algorithm was implemented in C++ for simulation, the same requirements can be easily reflected in the actual software code, which is written in the same language. With the proposed method, we can expect the software to operate in the same manner as the simulation.



(x,t) : IM message at time t
(*,t) : TEM message at scheduled time t
(y,t) : output message at time t
x : external input event
y : output event
t : time

```
AtomicThread()
{
    while(true) {
      msg = RecvWiat(msgQueue);
      if(msg.type == IM)
        ExtTransFn();   // δext
      if(msg.type == TEM) {
        OutputFn();         // λ
        IntTransFn();       // δint
      }
        t = TimeAdvanceFn(); // ta
      if (t != INFINITY)
        SetEvent(t, msg);
    }
}
```

IM : input message
TEM : time expired message
δ_{ext} : external transition function
δ_{int} : internal transition function
λ : output function
ta : time advance function

Figure 4. Message processing between Atomic Thread and Scheduler Thread

The simulation code can be converted to software code as follows.

1. Each atomic model, with consideration of the simulation code, is implemented to match a single-thread.
2. The input/out relationships and hierarchical structure of atomic models are flattened into a single-layered structure, and managed by the scheduler thread.
3. The scheduler thread is in charge of delivering messages between atomic models.
4. Input/output events occurring at atomic models are classified according to message type sent from the scheduler thread to each thread.
   i) External transition function is performed if the message type is IM (input message)
   ii) Output function and internal transition function are performed if the message type is TEM (time expired message)
5. In the Atomic Thread, TimeAdvanceFn() is a function for the time during which the current state is maintained. The time for maintaining the current state is defined in TimeAdvanceFn(). This value represents the time to the next event, and the SetEvent() function is used for sending to the scheduler thread. If the set time is INFINITY, the waiting time becomes infinite without having to update the atomic thread.

The block diagram and pseudo code in (Figure 4) show the message processing between Atomic Thread and Scheduler Thread.

```
1: DISP_thread()                    28:    if(msg.type == TEM) {
2: {                                29:      /* OutputFn().. */
3:   while(true) {                  30:      Display(m_phase, msg);
4:     msg = RecvWiat(msgQ);        31:      /* IntTransFn().. */
5:     if(msg.type == IM) {         32:      m_phase = WAIT;
6:       /* ExtTransFn().. */       33:    }
7:       switch(msg.GetCmd()) {     34:    /* TimeAdvaceFn().. */
8:         case CMD_NAVI_ON:        35:    switch (m_phase) {
9:         case CMD_NAVI_OFF:       36:      case WAIT:
10:        case CMD_NAVI_START:     37:         t = INFINITY;
11:        case CMD_NAVI_STOP:      38:         break;
12:        case SEL_DEST:           39:      case PATH_DISP:
13:           m_phase = INFO_DISP;  40:         t = 100;
14:           break;                41:         break;
15:        case SEL_PATH:           42:      case LOC_DISP:
16:           m_phase = PATH_DISP;  43:      case MAP_DISP:
17:           break;                44:         t = 150;
18:        case CURRENT_LOC:        45:         break;
19:           m_phase = LOC_DISP;   46:      case INFO_DISP:
20:           break;                47:                t = 10;
21:        case SCALE_UP:           58:         break;
22:        case SCALE_DOWN:         59:    }
23:        case FOCUS_MOVE:         60:    if (t != INFINITY)
24:           m_phase = MAP_DISP;   61:      SetEvent(msg, t);
25:           break;                62:  }
26:      }                          63: }
27:    }                            64:
```

Figure 5. Code of the DISP_thread

(Figure 5) is the code of the DISP model implemented with the proposed method. ExTransFn() in (Figure 4) is equivalent to7~26 in (Figure 5), OutputFn() equal to 30 in (Figure 5), IntTransFn() in 32 in (Figure 5), and TimeAdvanceFn() equal to 35~59 in (Figure 5).

The other models constituting the navigation system were each implemented into a thread to complete the navigation software. The program is verified with the test cases used during the validation process. (Figure 6) gives the test results of the navigation software. From the test results, we can confirm that the proposed structure satisfies the specified requirements.

3. Conclusion

This paper proposes the concurrency control method on the basis of DEVS modeling and simulation, which can be used in developing multi-thread software. The proposed method controls concurrency of each thread to solve decreasing consistency of a result, which makes development of multi-thread software difficult, and uses DEVS simulation to measure the minimum range of concurrency control, which ensures consistency of performance results. In addition, this paper suggests the method to transform simulation codes to actual software codes for quick and easy preparation of software codes, considering the results of simulation. Furthermore, the process to develop multi-thread software using methods presented in navigation software development cases is explained.

For future study, the scope of the method proposed in this study should be expanded in order to apply it to not only C++ language platform but also other types of platforms and a study on an automatic software code generation tool is also required to develop quicker and more accurate multi-thread software.

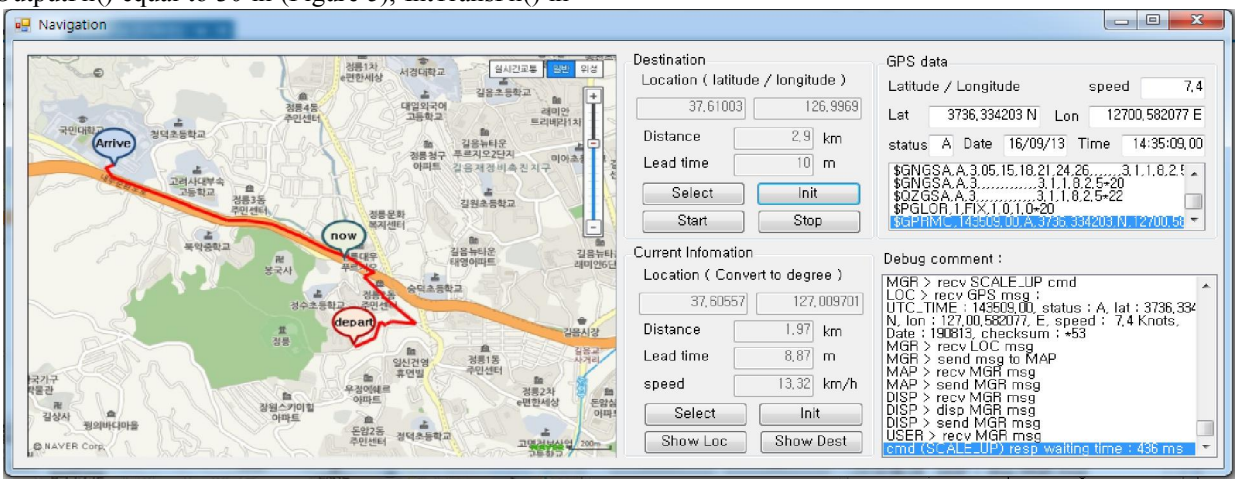


Figure 6. Test result of navigation software

Corresponding Author:
Prof. Yeong Rak Seong
Department of Electrical Engineering
Kookmin University, Korea
E-mail: yeong@kookmin.ac.kr

References

1. Mukerji J, Miller J. Model Driven Architecture Guide V1.0.1. 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01
2. OMG Architecture Board MDA Drafting Team. Model Driven Architecture - A Technical

Perspective. 2001. http://www.omg.org/chi-bin/doc?ormsc/01-07-01.pdf
3. Unified Modeling Language Home Page: http://www.uml.org.
4. Object Management Group. UML 2.0 Specification. 2005. http://www.omg.org/spec/UML/2.0/.
5. Bernard PZ. Theory of Modeling and Simulation. John Wiley. New York 1984.
6. Kim TG. DEVSim++ User's Manual: C++ Based Simulation with Hierarchical Modular DEVS Models. 1994.
7. Randy SW. Prototyping and the Systems Development Life Cycle. Journal of Information Systems Management 1991; 8(2): 47-53.
8. Donald FS, David FM. Discrete Mathematics in Computer Science. Prentice Hall 1977.
9. Bernard PZ. DEVS Representation of Dynamical System : Concepts, Algorithm, and Simulation. Journal of Parallel and Distributed Computing 2001; 9: 271-281.
10. Bernard PZ. Object-Oriented Simulation with Hierarchical, Modular Models. Academic Press 1990.

5/26/2014