International Journal of Multimedia and Ubiquitous Engineering
Vol.10, No.10 (2015), pp.329-336
http://dx.doi.org/10.14257/ijmue.2015.10.10.32

# Construction of Concurrent Programs Based on a Modeling and Simulation Formalism

Jung Hyun Im[1], Ha-Ryoung Oh[1] and Yeong Rak Seong[2]

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

***Abstract***

*This paper proposes a framework for developing a concurrent program using the discrete event system specification (DEVS) formalism. Within the proposed framework, a concurrent program is modeled by the DEVS formalism, and the modeling result is validated through simulation in a DEVS abstract simulator environment, called DEVSim++. Then, the validated modeling results are translated to multi-threaded program codes written in a conventional programming language. For that, each DEVS model which specifies behavior of a component is converted to a single thread, called an atomic thread, and every connection information between the components are clustered together and converted to a data structure, called a port mapping table. This paper also proposes an efficient solution which combines several atomic threads into a new thread, called a combined thread.*

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

## 1. Introduction

Modeling and simulation (MAS) is a powerful technique for system design. Many studies have been conducted on the application of MAS techniques in the development of software systems [1-2]. Especially, if we develop a software system by using formal MAS techniques, we can use formal modeling techniques in software design, and can verify the design results through simulation. Consequently, we can reduce software development time. In most of existing software development methodologies based on formal MAS techniques, however, the formal MAS techniques are employed only for design, and the design results are implemented by using conventional procedural or object-oriented languages. Therefore, the formal design results cannot be properly expressed in the implemented code due to the limitation of the implementation languages.

On the other hand, to develop a concurrent program is known as a very exhausting job. Especially, it takes very long time to test a concurrent program. This is because the execution order of threads in a single program may vary occasionally even if the identical events occur at the identical time with the identical order [3].

This paper proposes a framework for developing a software program with concurrency using the discrete event system specification (DEVS) formalism [4-5]. In the proposed framework, a concurrent program is modeled by using the DEVS formalism and simulated by using the DEVS abstract simulator algorithm [6]. Due to a synchronization mechanism based on simulation time in DEVS, execution of components, which will be implemented by threads later, can be strictly controlled, and thus behavior of concurrent

ISSN: 1975-0080 IJMUE

programs can be preciously specified with the framework. 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. Although the generated code is also written in a conventional language, it exhibits the semantics of the DEVS formalism, and includes the synchronization mechanism stated earlier. Therefore, the code differs from the code developed by using conventional informal methods. A naive version of the suggested framework was introduced in [7] and applied in the development of a navigation application. This paper explains the framework more specifically by focusing on the framework itself. Furthermore, a solution to efficiently prevent the over-generation of threads is also proposed.

This paper is organized as follows. Section 2 presents a brief review of the DEVS formalism. Section 3 presents how modeling results are translated to multi-threaded program codes written in a conventional programming language. Finally, Section 4 concludes this paper.

## 2. DEVS Formalism

The DEVS formalism specifies a discrete event system in a hierarchical, modular manner. There are two classes of models within the DEVS formalism: *atomic* and *coupled models*. An atomic DEVS model specifies behavior of a component. An atomic DEVS model AM is specified as follows [8]:

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

- X : Input events set
- Y : Output events set
- S : Sequential state set
- $\delta_{ext}$ : $Q \times X \rightarrow S$ : External transition function

  $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$: total state of AM
- $\delta_{int}$ : $S \rightarrow S$ : Internal transition function
- $\lambda$ : $S \rightarrow Y$ : Output function
- $ta$ : $S \rightarrow R^+$ : Time advance function

The state of an atomic model is changed by both the external transition function and the internal transition function. When an input event arrives from other models, the state of the atomic model is changed by the external transition function. The next state is decided based on the current state, the time elapsed during the current state, and the input event. On the other hand, when no input event arrives until the schedule time, which is determined by the time advance function, the state of the model is changed by the internal transition function. At this time, the next state is decided based solely on the current state. Right before any internal transition, the atomic model can generate output events determined by applying the output function to the current state. The output events are transmitted to other models as input events.

A coupled model, the second class of the DEVS models, combines component models together to form a new model. Since a coupled model can be employed as a component of a larger coupled model, a complex model can be constructed in a hierarchical manner. A coupled model CM is specified as follows:

$$CM = < X, Y, \{M_i\}, EIC, EOC, IC, SELECT >$$

- X : Input events set
- Y : Output events set

 

- $\{M_i\}$ : Components set
- EIC $\subseteq$ CM.X $\times$ $M_i$.X : External input coupling relation
- EOC $\subseteq$ $M_i$.Y $\times$ CM.Y: External output coupling relation
- IC $\subseteq$ $M_i$.Y $\times$ $M_j$.X : Internal coupling relation
- SELECT = subsets of D $\rightarrow$ D : tie-breaking function

The three coupling relationships connect the coupled model and its component models; the external input coupling EIC specifies coupling from the input events of the coupled model to the input events of the component models, the external output coupling EOC specifies coupling from the output events of the component models to the output events of the coupled model; and the internal coupling IC specifies coupling from the output events of the component models to input events of the other component models.

Due to the hierarchical feature, the modeling result within the DEVS formalism can be represented as a tree structure, as shown in Figure 1(a). Each leaf node of the tree is an atomic DEVS model, and each internal node in the tree is a coupled DEVS model.

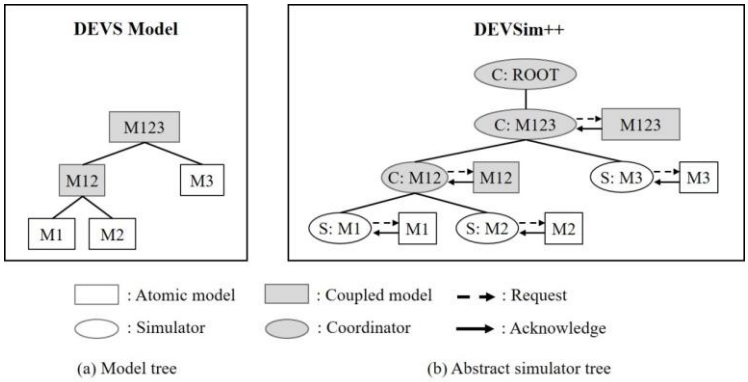


**Figure 1. Transformation from a Model Tree to an Abstract Simulator Tree**

The DEVS abstract simulator algorithm 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++ [9, 10] is a DEVS abstract simulator environment based on the C++ language.

To simulate a DEVS model with the abstract simulator algorithm, a model tree should be converted to an abstract simulator tree. Figure 1(b) shows the abstract simulator tree of the Figure 1(a) in DEVSim++. A simulator is placed at each atomic model node and a coordinator is placed at each 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. Simulation starts with the root coordinator delivering the first message to the top coordinator via the link connecting them in the tree. Whenever an abstract simulator receives a message, it requests to the associated DEVS model for the knowledge that is required to process the message. According to the response of the DEVS model, the abstract simulator generates new messages and sends them to other abstract simulators. By repeating theis action, the simulation proceeds.

## 3. Proposed Framework

For developing concurrent programs efficiently, this paper proposes a novel software development framework based on discrete event model and simulation formalism. Figure 2 illustrates the framework. It looks similar to the traditional waterfall framework [11] or prototyping framework [12]. However, it has two significant differences.

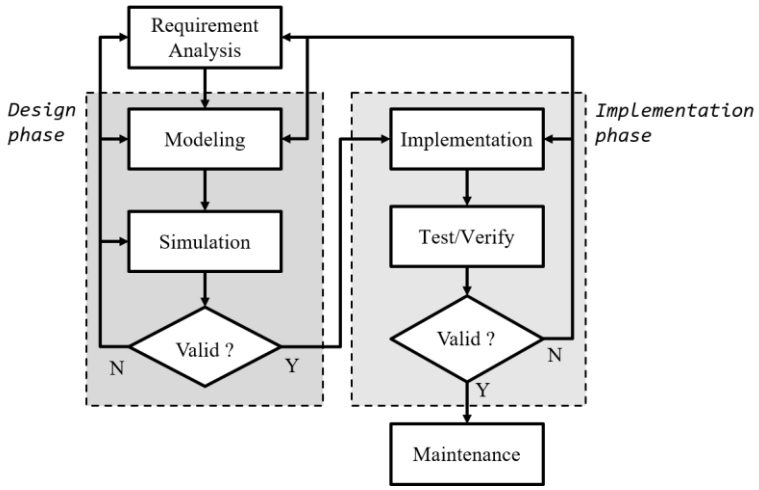


**Figure 2. The Flowchart of the Proposed Framework**

First, in the design phase, a concurrent program is modeled by using the DEVS formalism and the modeling result is verified and validated through simulation experiments in the DEVSim++ environment. Software validation is a quite difficult, time-consuming job. Validation of a multi-thread program is more difficult due to non-determinism in thread scheduling. Generally, to validate software design result, it should be implemented to an actual program. On the contrary, the design result specified by the DEVS formalism can be easily validated through simulation experiments in the DEVSim++ environment without implementing to an actual program. Moreover, non-determinism in thread scheduling can be easily eliminated, since execution of components is strictly controlled by a synchronization mechanism based on virtual time in the DEVS formalism.

Second, in the implementation phase, the verified/validated modeling results are transformed to an actual program code through a series of conversion steps. The rest of this paper is devoted to explain how a DEVSim++ code which models and simulates a concurrent program can be translated to an actual program code.

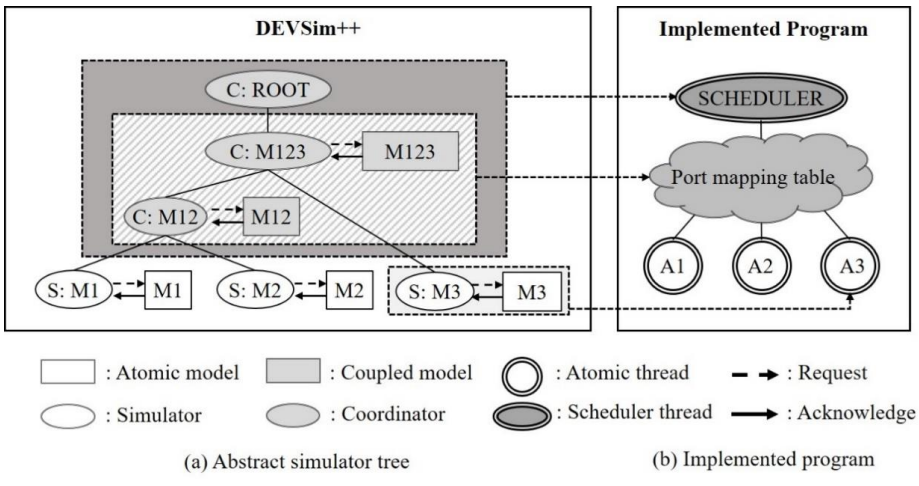


**Figure 3. Transformation from an Abstract Simulator Tree to a Multi-Threaded Program**

A DEVSim++ simulation code can be converted to an actual program code in various ways. From the fact that the behavior of the designed software is specified by only atomic models within the DEVS formalism, this paper implements the program code so that the

 

software design result represented by the DEVSim++ simulation code can be maintained as much as possible. The implementation phase consists of three parts as shown in Figure 3(b); 1) each pair of 'atomic model–simulator' is implemented as an independent thread (Section 3.1); 2) the connection information between atomic models stored in all 'coupled model–coordinator' pairs is converted to a data structure (Section 3.2); and 3) the scheduling function included in the root coordinator and 'coupled model–coordinator' pairs is implemented as a thread (Section 3.3).

### 3.1. Atomic Thread

Each pair of an atomic model and a simulator is implemented as a single thread, called an *atomic thread*. In this paper, to preserve 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 recoded as the control code of the atomic thread. More precisely, the four characteristic functions (*e.g*. the external/internal transition function, the time advanced function, and the output function) and the state variable set, specified in DEVSim++ atomic model codes, are unchanged. However, the specification of the input/output events sets are migrated into the port mapping table, which will be explained in Section 3.2, and is referenced during inter-thread communication.

Figure 4 shows the simplified architecture of an atomic thread. An atomic thread is divided into two parts: the control code part and the atomic model part. The control code is the main routine of the thread. It checks arrival of messages continuously, and invokes appropriate functions of the involved atomic model. The atomic model part consists of the four characteristic functions and the state variable set of an atomic model as stated earlier.

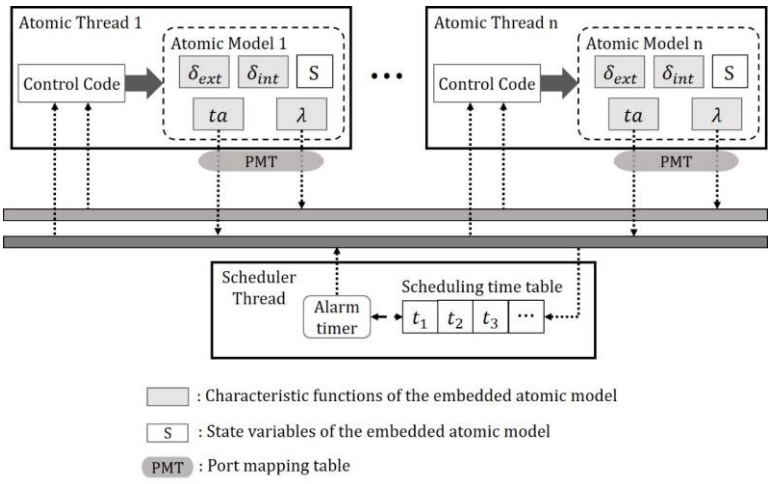


**Figure 4. Structure of the Transformed Multi-Threaded Program**

Whenever an atomic thread receives a message from other threads, the message is delivered to the control code. The message is classified into two types depending on the source of the message: the messages transmitted from other atomic threads, and the messages transmitted from the scheduler thread. If the delivered message is transmitted from other atomic threads, this means that the message is an output event generated by other atomic model. Therefore, the control code calls the external transition function to change the state of the atomic model. It then calls the time advanced function to determine the next schedule time, and send the time to the scheduler thread, which will be explained in Section 3.3.

Meanwhile, if the delivered message is transmitted from the scheduler thread, the message corresponds to the schedule message generated by the root coordinator in

DEVSim++. Now, the control code calls the output function to produce output events, which will be transmitted to other atomic threads, and calls the internal transition function to change the state of the atomic model. It finally calls the time advanced function and delivers the new schedule time to the scheduler thread.

### 3.2. Port Mapping Table

A *port mapping table* plays a role in connecting the input/output events of atomic models embedded in atomic threads. To attain this end, two types of information are required; the information of input/output events sets of atomic and coupled models, and the coupling information between the input/output events sets. Since the modeling result within the DEVS formalism is represented by a tree as shown in Figure 1(a), it requires a complex operation to extract specific information from the connection information separately stored in the many coupled models. To simplify these processes, this paper flattens the hierarchical structure of the modeling result first. By flattening shown in Figure 5, every hierarchical information in the modeling result is eliminated. There is only one coupled model in the flattened model tree. Since the coupled model contains all atomic models, it specifies every connection between the input/output events sets of the atomic models. Finally, a port mapping table can be easily constructed from the specification of the single coupled model.

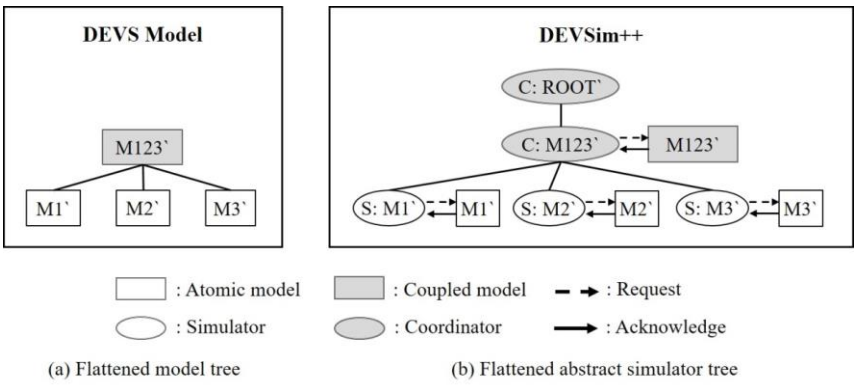


**Figure 5. Flattening of Figure 3.1**

### 3.3. Scheduler Thread

The *scheduler thread* generates a schedule message to activate execution of the atomic threads. It implements the scheduling function that is included in the root coordinator and 'coupled model–coordinator' pairs in DEVSim++. To attain this end, the scheduler thread consists of a scheduling time table and an alarm timer. Whenever an atomic model embedded in an atomic thread changes its state, it reports its next scheduled execution time to the scheduler thread by transmitting a message. Then, the scheduler thread updates the scheduling time table properly, and set the alarm timer to the minimum value of the scheduling time table. Eventually, when the alarm timer expires, the scheduler thread generates a schedule message and delivers the message to the corresponding atomic thread to activate the atomic thread.

### 3.4. Thread Population Tuning

Depending on the modelers' expertise or perspective, a system can be modeled in various ways even with the same MAS technique. Therefore, a huge number of atomic models may be generated in some cases. As explained earlier, if one atomic model is implemented in one atomic thread, too many threads should be generated. This increases the context switching overhead and negatively influences the performance of the software.

 

Besides, the proposed framework may not be applied if the execution environment has limitations in the number of threads (*e.g.*, uC/OS [13]). However, it is not appropriate to limit the total number of atomic models in modeling, since it restricts modelers' degree of freedom.

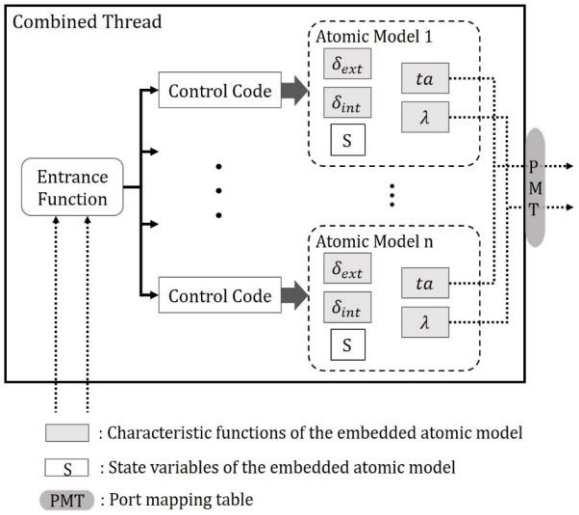


**Figure 6. Structure of a Combined Thread**

This paper suggests a method for implementing threads without over-generating them, while securing the modeling freedom. Although there may be many ways to adjust the number of threads, this paper suggests a method that implements several '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 attain this end, the control code of an atomic thread is rewritten and modularized into a function called the *entrance function* in figure 6. 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.

## 4. Conclusions

This paper proposes a framework to develop concurrent programs based on a discrete event system modeling and simulation methodology. The proposed framework employs the DEVS formalism and DEVSim++ for the design, simulation, and implementation of concurrent programs. This paper mainly focuses on 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 coupling information distributed in coupled DEVS models are collected into a port mapping table. In addition, for scheduling of operation of atomic threads, the scheduling function of DEVSim++ is implemented by a scheduler thread.

Lastly, we suggested a method to reduce the number of threads through combining several atomic threads into a combined thread. The proposed method thus reduces the context switching overhead caused by thread over-generation, improves the software performance, and enables us to develop concurrent programs even in an environment having limitations in the number of threads.

## References

[1] W. T. Tsai, Z. Cao, X. Wei, R. Paul, Q. Huang and X. Sum, "Modelling and Simulation in Service-Oriented Software Development", Simulation, vol. 83, **(2011)**.

[2] Y. H. Kim, T. Y Lee, Y. R. Seong and H. R. Oh, "Design and Realization of Multi-Thread Structure for an LLRP Server", Applied Mathematics & Information Sciences, vol. 6, no. 3, **(2012)**, pp. 1117-1123.
[3] M. Emmi, Q. Shaz and R. Zvonimir, "Delay-bounded scheduling", ACM SIGPLAN Notices, ACM, vol. 46, no. 1, **(2011)**, pp. 411-422.
[4] B. P. Zeigler, "Theory of Modelling and Simulation", John Wiley, New York, **(1984)**.
[5] B. P. Zeigler, "DEVS formalism: A framework for hierarchical model development", IEEE Transactions on Software Engineering, vol. 2, **(1988)**, pp. 228-241.
[6] B. P. Zeigler, "Object-oriented simulation with hierarchical, modular models: intelligent agents and endomorphic systems", Academic Press, **(2014)**.
[7] Y. H. Kim, H. R. Oh and Y. R. Seong, "Software Development Method Using the Concurrency Control Approach Based on DEVS Simulation", Proceeding of the 2014 FTRA International Conference on ACS, vol. 11, no. 7, **(2014)**, pp. 553-558.
[8] T. G. Kim, "DEVS Formalism: Reusable Model Specification in an Object-Oriented Framework", International Journal in Computer Simulation, vol. 5, no. 4, **(1995)**, pp. 397-415.
[9] T. G. Kim and S. B. Park, "The DEVS Formalism: Hierarchical Modular Systems Specification in C++", Proceeding of 1992 European Simulation Multi-conference, **(1992)**, pp. 152-156.
[10] T. G. Kim, "DEVSim++ User's Manual: C++ Based Simulation with Hierarchical Modular DEVS Models", **(1994)**.
[11] D. Whitgift, "Methods and tools for software configuration management", John Wiley, **(1991)**.
[12] S. W. Randy, "Prototyping and the Systems Development Life Cycle", Journal of Information Systems Management, vol. 8, no. 2, **(1991)**, pp. 47-53.
[13] J. J. Labrosse, "uC/OS:The Real-Time Kernel", R&D Publications, **(1992)**.
[14] J. H. Im, H. R. Oh and Y. R. Seong, "Development of Concurrent Programs based on a Modelling and Simulation Formalism", Proceedings of International Workshop Ubiquitous Science and Engineering, Jeju island, Korea, August 19-22, **(2015)**.

## Authors

**Jung Hyun Im**, he received the B.S degree in Department of Electrical Engineering from Kookmin University and the M.S degree in Department of Secured-Smart Electric Vehicle from Kookmin University. He is currently a student in Department of Secured-Smart Electric Vehicle from Kookmin University. His research interests are in the areas of discrete event system modeling and simulation, and embedded system.

**Ha-Ryoung Oh**, he was born in Busan, Korea, in 1961. He received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 1983 and the M.S. and Ph.D. degrees in electrical engineering from Korea Advanced Institute of Science and Technology, Daejeon, Korea, in 1988 and 1992, respectively. Since 1992, he has been a professor with Kookmin University, Seoul. His current research interests include RFID system, wireless sensor network, and embedded system.

**Yeong Rak Seong**, he received the B.S degree in electronics engineering from Hanyang University, Seoul, Korea, in 1989 and the M.S and Ph.D. degrees in electrical engineering from Korea Advanced Institute of Science and Technology, Daejeon, Korea, in 1991 and 1995, respectively. Since 1996, he has been with Kookmin University, Seoul, where he is currently a professor. His current research interests include real-time system, wireless sensor networks, discrete event system modeling and simulation, and embedded system.

 