Proceedings of the Korean Nuclear Spring Meeting
Gyeong ju, Korea, May 2003

Verification and Testing of the RTOS for safety-critical embedded systems

Na Young Lee[1], Jin Hyun Kim[2], Ah Young Sung[3], Byung Ju Choi [3], Jin Young Choi[2], Jang Soo Lee[4]

[1]Seoul National University, 56-1, Shillim, Kwanak, 151-742, Seoul, Korea
[2]Korea University, 5-1, Anam, Seongbuk, 136-701, Seoul, Korea
[3]Ewha Womans University, 11-1, Daehyun, Seodaemoon, 120-750, Seoul, Korea
[4]Korea Atomic Energy Research Institute, Daejon, Korea

Abstract

Development in Instrumentation and Control (I&C) technology provides more convenience and better performance, thus, adopted in many fields. To adopt newly developed technology, nuclear industry requires rigorous V&V procedure and tests to assure reliable operation. Adoption of digital system requires verification and testing of the OS for licensing. Commercial real-time operating system (RTOS) is targeted to apply to various, unpredictable needs, which makes it difficult to verify. For this reason, simple, application-oriented real-time OS is developed for the nuclear application. In this work, we show how to verify the developed RTOS at each development lifecycle. Commercial formal tool is used in specification and verification of the system. Based on the developed model, software in C language is automatically generated. Tests are performed for two purposes; one is to identify consistency between the verified model and the generated code, the other is to find errors in the generated code. The former assumes that the verified model is correct, and the latter incorrect. Test data are generated separately to satisfy each purpose. After we test the RTOS software, we implement the test board embedded with the developed RTOS and the application software, which simulates the safety critical plant protection function.

Testing to identify whether the reliability criteria is satisfied or not is also designed in this work. It results in that the developed RTOS software works well when it is embedded in the system.

1. Introduction

Instrumentation and Control (I&C) systems have made a big improvement during last decades, and are successfully adopted in many fields. The nuclear industry, known as one of the most safety-critical fields, is slow to adopt newly developed instrumentation and control (I&C) technologies. It is partly because of the conservative attitude generic to the safety

critical industry, but more because of the scarcity of new construction of nuclear power plants (NPPs), which results in little need for the newly developed I&C systems [1,2].

As the need for the digital I&C system becomes increasing, how to assure the correct behavior of the system becomes important. NIST recommends performing Verification and Validation (V&V) in every life-cycle stage for the safety critical application. It also recommends some properties to be verified, such as completeness, consistency, correctness, testability and understandability in each stage. These processes are also required for the licensing of nuclear applications.

In this paper, we focus on the V&V and testing procedure of the real time OS, which is to be used in a real-time, embedded system for the safety critical nuclear application. Since the operating system takes charge in controlling or scheduling processes, it needs to be verified and tested. We show how to perform V&V and also show how to generate test data to prove the correctness and the liveness of the OS.

There are not many studies reported on the OS verification, for the OS is too complicated and non-deterministic, which makes it difficult to verify. For the nuclear application, however, it is recommended to implement the system simple and deterministic to avoid errors. To be deterministic, the developed OS use fixed schedule determined during compiling. Thus, the OS in our application becomes verifiable.

Here, we specify and verify the requirements and the design using the STATEMATE. It supports functional compositions in the form of Activity charts, description of interactions between states by Statechart, and automatic code generation based on the Activity chart or Statecharts. The generated code needs testing, and it is performed for two purposes. One is to show whether verified properties of the model are inherited to the code. This assumes the verified model is correct. Model based test data including verification traces are generated for this purpose and then performed. The other is, however, to find errors of the code. This assumes that the code even though it is generated from the verified model, has errors. Structural coverage and functionality are checked for this purpose.

After testing the developed OS software, we embed it in the test board with the Digital Plant Protection System (DPPS) software, which simulates the safety critical reactor protection function. Developed system should satisfy 10^{-3} probability of failure on demand (pfd) to show it is reliable enough to the safety critical application [3]. We show that by performing the required number of tests in the test board.

2. Methodology

In this section, we show how to develop RTOS for the safety critical embedded system, which needs verification and validation for the licensing. Our approach is classified as three parts: specification and verification at each stage of the life cycle, test data generation for the software testing, the embedded system testing. The developed software and the hardware are integrated as an embedded system. As application software, abstract plant protection function is implemented and used. Testing is performed to identify if the developed RTOS software works well when it is integrated with hardware and application software. Our approach is shown in the figure 2.1.

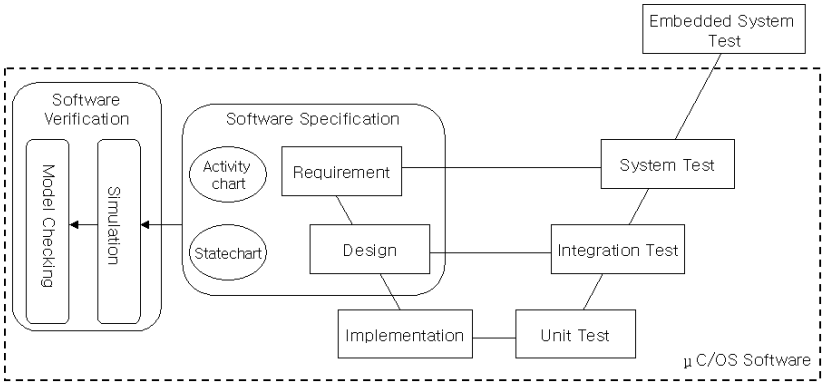

Figure 2.1 Overall approaches of the V&V and testing of the RTOS

2.1 S/W Specification and Verification using Formal methods

In this section, we specify RTOS with graphical formal language, and verify it with model checking. We have used STATEMATE MAGNUM toolset, I-Logix, as a formal language and the model-checker. STATEMATE MAGNUM is based on Activity-charts [4] and Statecharts [5].

The Activity-charts is used for the functional description and the Statecharts in the behavioral description of the system. Statecharts is a well-defined formal language with mathematical semantics [6]. Using Statecharts, we can describe behaviors of the requirements/design stage with states and transitions between them. Transitions in the Statecharts are five-tuple (source, target, event, action, condition). The arrow in the Statecharts goes from source to target and is labeled as e[c]/a, which means that the event e triggers the transition when condition c is true and then action ‘a’ is carried out when transition is taken. Action ‘a’ can be a list of actions.

Once the design of the model is specified, it needs V&V to prove it satisfies properties of the system. STATEMATE has a simulation tool [7] to validate system design. Using simulation, we can validate not only each module but also full systems.

STATEMATE MAGNUM also provides ModelChecker [8] as a formal verification tool. Model checking [9] can be used in checking the dynamic behavior of a design regarding specific properties. It is important to know that the model checking is different from the testing in the classical meaning. In contrast to the testing, model checking is complete in a mathematical sense. If the model checker proves that a specific property is satisfied in the model, the result is 100% correct. If the model checker shows that a basic state in the STATEMATE design is not reachable, it means that no one will ever find the simulation run (only input variables can be changed during execution), of which trace path includes the specific basic state. Thus, the model checking can be called exhaustive testing. All this design information is translated into the input language of the model checker kernel by the tool automatically. On the other hand, a user has to define the properties to be checked [8].

We can get an automatically generated code from the code generator of STATEMATE, which is expected to have fewer errors than by implemented by human.

2.2 S/W Testing

The RTOS testing is performed in two ways: during the software testing and the embedded system testing, as shown in the figure 2.1. The software testing focuses on the RTOS software itself, whereas the embedded system testing focuses on the integration with the application software embedded in the RTOS.

Unit testing, integration testing and system testing are performed. To select test data for each testing, we utilize results driven from each life cycle phase as shown in the table 2.1.

In case of unit testing, we select unit test data by applying the white-box testing criteria to the code, which is automatically generated from the STATEMATEMAGNUM in the implementation phase. We also select unit test data by applying the branch coverage criteria to the Statecharts specified in the design phase. In case of integration testing, we select integration test data by utilizing the Statecharts specified in the design phase. In case of system testing, we select system test data by utilizing the Activity charts specified in the requirement phase.

Table 2.1 Work products at each life cycle phase and Levels of Testing

	Requirement	Design	Implementation
System Testing	Activity chart		
Integration Testing		Statechart	
Unit Testing		Statechart	Automatic Code

2.3 Embedded System Testing

In this paper, we implement the embedded system, which includes simple plant protection software for the nuclear power plant and RTOS. Both are generated from the verified model specified using the STATEMATE. As test criteria for this testing, we set 10^{-3} failure on demand, which is used as reliability testing criteria for the safety-critical nuclear systems. As a test data, we generate random combinations of failure condition, until it satisfies to obtain the required number of failures.

3. Formal Specification and Verification Procedure

3.1 Requirements of RTOS

RTOS, unlike a generic OS, the logical correctness of an RTOS depends not only on the correctness of its output, but also on the timeliness of its output. In addition, like other OS, must provide at least the following three functions: task scheduling, task dispatching, and intertask communication. Scheduling is a process that determines which task controls CPU next. It is called when interrupt occurs, task finishes its job, and running task is suspended. When scheduler is called, it scans a list of ready state tasks, selects task according the

scheduling policy, and give CPU control to it. Dispatching is a process that switches, saves and restores context of running task and that of next running task. In this paper, a dispatcher creates tasks periodically and makes it ready state. Intertask communication enables tasks to exchange information and synchronize with each other. Message mailbox and message queue are implemented to perform communications between tasks.

User code is taken as a task is executed with predefined execution time and period. OS can interrupt it during execution of any user tasks.

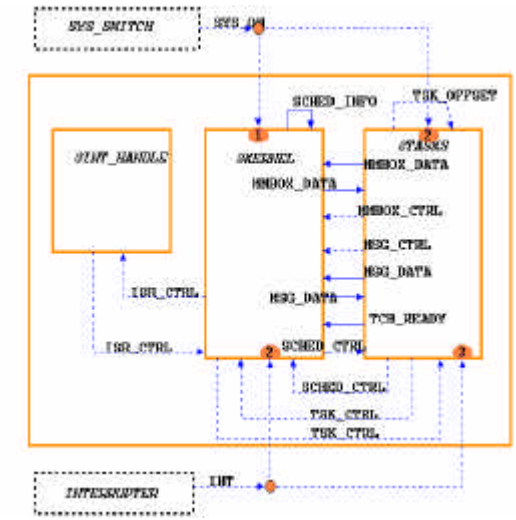

Figure 3.1 S/W Requirements Specification for the RTOS

3.2 RTOS Specification and Verification

Functional specification of the RTOS is shown in the figure 3.1. Each box includes functions of the RTOS. The lines represent communications between functions, and the dotted box represents external environment. RTOS contains kernel, interrupt handler, and tasks. Kernel includes scheduler, task manager, message mailbox, and message queue. ‘ Task’ module has seven tasks: five with the same, and one with a higher priority. The last one is assigned as an idle task.

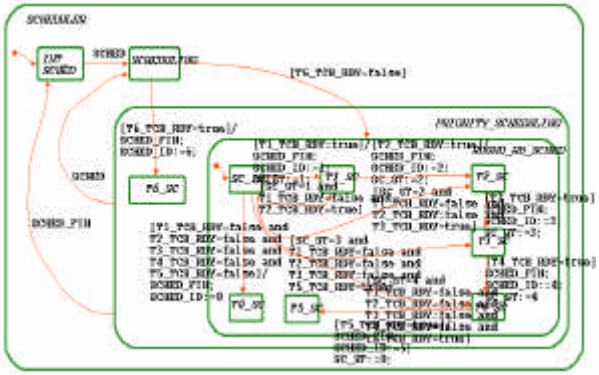

Figure 3.2 Behavioral Specification of Scheduler

Scheduler performs priority-based, or round robin scheduling. Embedded software in the RTOS has two different tasks in its application. One is monitoring and voting task, the other is trip task. Trip (shutdown) task has the highest priority and others have the same priority. Figure 3.2 shows behaviors inside the scheduler module.

For the intertask communication, our RTOS provides message mailbox and message queue.

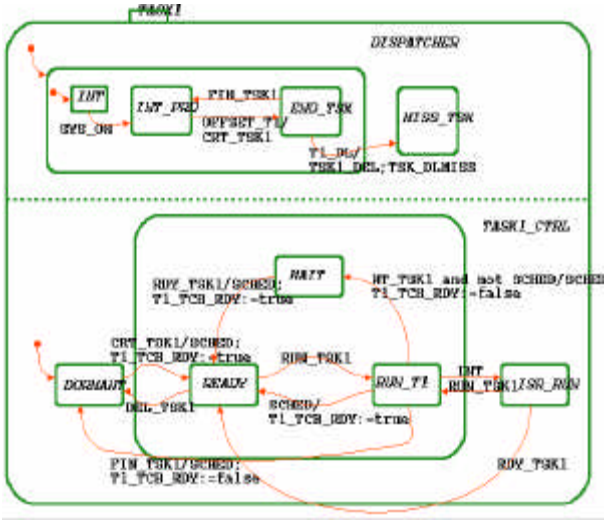

Figure 3.3 Behavioral Specification of the Task

Figure 3.3 shows behavior of tasks. Each task has basically five states: dormant, ready, running, waiting and ISR. Dormant state indicates that task is not created. Task in the dormant state can be created after task get into its period and spend offset time. Task in the ready state is waiting for the scheduling, which is ready to run. Task in the running state takes transition into the waiting state when it needs to use resource or to spend time. Transition into ISR state is taken when the running task is interrupted by the system. The dispatcher module is used to wake up the task.

After the specification, RTOS is simulated using STATEMATE, as shown in the figure 3.4.

Figure 3.4 Simulations of the RTOS

Verified properties by ModelChecker are as follows:

- Whether the specified RTOS has unreachable states or not.
- Whether the specified RTOS is deterministic or not.
- Whether two or more tasks can be in the running state simultaneously.

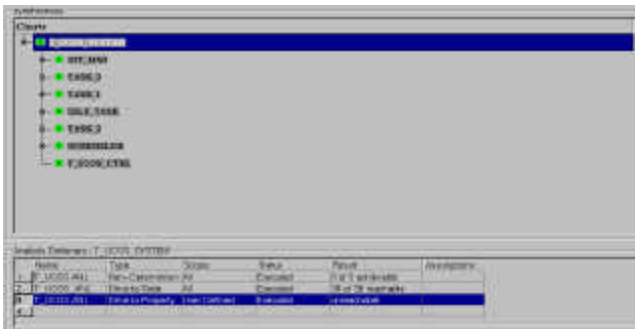

Figure 3.5 Verification Using Model checking

We show that the specified RTOS satisfies all the properties described above. The result is shown in the figure 3.5.

Design specification in Statecharts can be translated automatically into C code, which is tested in the later stage.

4. Software Testing

4.1 Unit Testing

In the unit testing, we define the unit as the smallest module that is testable independently. Unit testing is performed to ensure correct operations of these units.

In the RTOS unit testing, units are extracted from the Statecharts, and automatically converted as C codes, respectively. There are two purposes of unit testing: one is to identify whether the automatically converted C code is correct, and the other is to check whether the behavior of the code is consistent as we design.

Table 4.1 Test data for the ‘ Task4’

Number	State-based Unit test data
1	CRT_TSK=T & RUN_TSK=T
2	CRT_TSK=T & RUN_TSK=T & WT_TSK=T & RDY_TSK4=T
3	CRT_TSK=T & RUN_TSK T & SCHED=T & DEL_TSK4=T
4	CRT_TSK=T & RUN_TSK=T & FIN_TSK4=T
5	CRT_TSK=T & RUN_TSK=T & INT=T
6	CRT_TSK=T & RUN_TSK=T & INT=T & RDY_TSK4=T

4.2 State-based Unit Testing

We extract the state-based unit test data from units specified in the Statecharts by applying the branch coverage criteria. Branch coverage criteria, which need to cover all the transitions, are chosen, because the specification itself is based on states and transitions between them. The procedure to generate test data for the state-based unit testing is as follows.

(1) Identify units to be tested.
(2) Select test data for the state-based unit testing.
① Apply branch coverage criteria to units extracted from the design specification.
② Select test data to cover all the branches to satisfy the criteria.

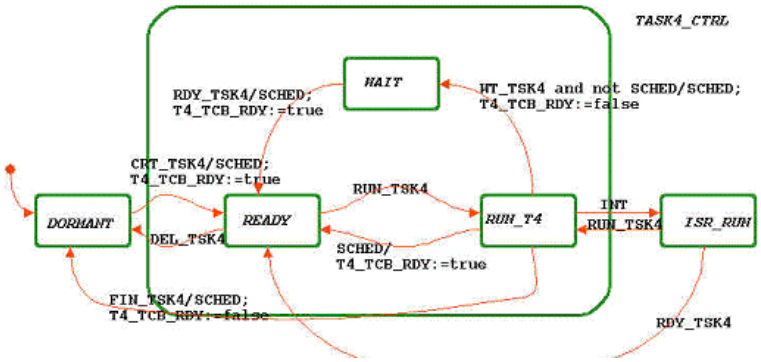

Figure 4.1 Statechart of the Task4

As an example, we apply branch coverage criteria to the ‘Task4’, shown in the figure 4.1, and then we select six test data for it, which is shown in the table 4.1.

4.1 Code-based Unit Testing

In this paper, we perform code-based testing using the structural testing tool, ATAC (Automatic Test Analysis for C) [11, 12]. We select two white-box testing criteria: statement coverage criteria and branch coverage criteria. Statement coverage criteria is chosen as recommended in the Reg.Guide 1.171 [13] for the nuclear application. According to the Reg.Guide 1.171, we should cover all statements in the code in the code-based testing. Branch coverage criteria is also chosen as a stronger criteria in this testing.

The procedure to generate the test data for the code-based unit testing is as follows.

(1) Identify units to be tested.
(2) Generate automatic codes for identified units one by one.
(3) Select test data for the code-based unit testing.
① Apply statement and branch coverage criteria to the code.
② Select test data to cover 100% of both criteria.

4.2 Integration Testing

The purpose of integration testing is to check whether there is any error when units are combined together. Thus, it focuses on interfaces between modules. We select test data that can detect interface messages between modules. The procedure to select integration test data is as shown below:

(1) Draw the state-composition diagram.

[Steps to draw up the state-composition diagram]

① Identify units to be combined into a module. This is shown in the figure 4.2. Modules and units are represented in rounded rectangles and ellipses, respectively. An initial unit is represented in an ellipse with a check mark and a final unit is represented with two ellipses.

② Relationships between units are represented in dotted lines, and between modules are represented in solid lines.

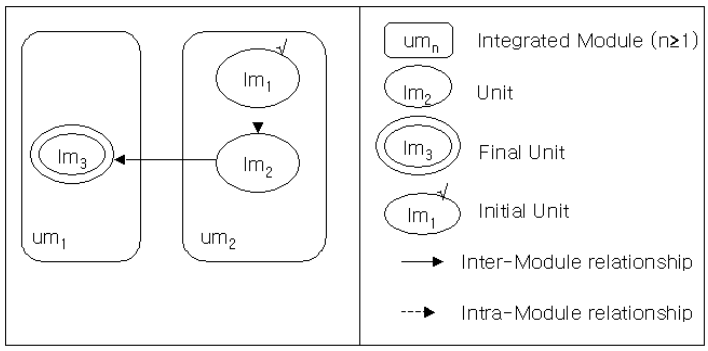

Figure 4.2 Notations of the state-composition diagram

(2) In the state-composition diagram, a state stands for a unit or a module whereas a transition stands for communication relation between states. By applying branch coverage criteria, we can cover all states and branches in the state-composition diagram. Therefore, selected test data by applying the branch coverage criteria cover all the notations in the state-composition diagram.

4.2.1 State-composition diagram of the RTOS

(1) Identify units that are combined into a module

Figure 4.3 shows the state-composition diagram of the RTOS. There are two modules: Kernel and Tasks module. Those modules are composed of two and eight units, respectively.

(2) Identify relationships

As shown in the figure 4.3, if the Tasks module gets the SYS_ON signal, the ‘Task’ units in the Tasks module send the TCB_RDY signal to the Kernel module. And then, the Kernel module sends the RUN_TSK signal to initiate the target ‘Task’.

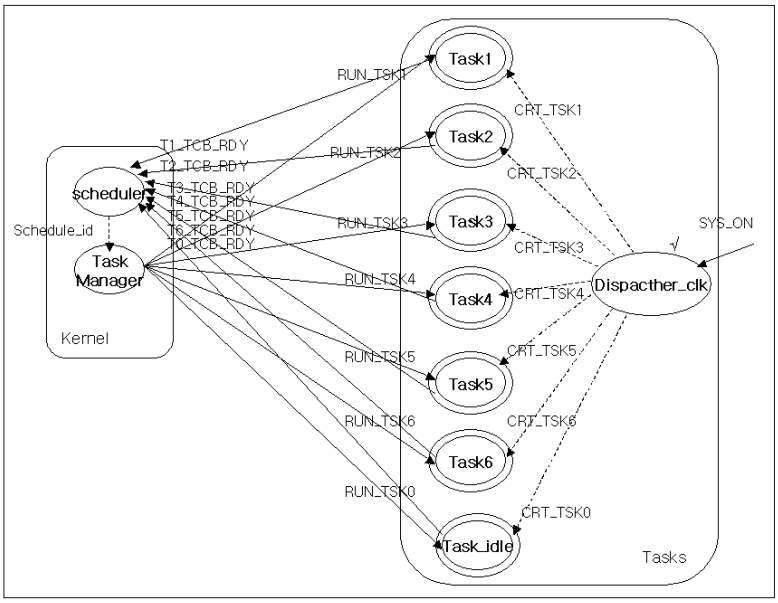

Figure 4.3 State-Composition Diagram of the RTOS

4.2.2 Test data for integration testing

We select integration test data by applying the branch coverage criteria to the transition relations between modules as shown in the figure 4.3. The bold lines in the figure 4.4 represent one example of applying the branch coverage criteria to the ‘Task1’ unit in the ‘Tasks’ module. If the ‘Task1’ takes a ‘SYS_ON = T’ signal, it gives out a ‘T1_TCB_RDY=T’ signal to the Kernel module. And then, the Kernel module sends out a ‘RUN_TSK1=T’ signal to initiate ‘Task1’.

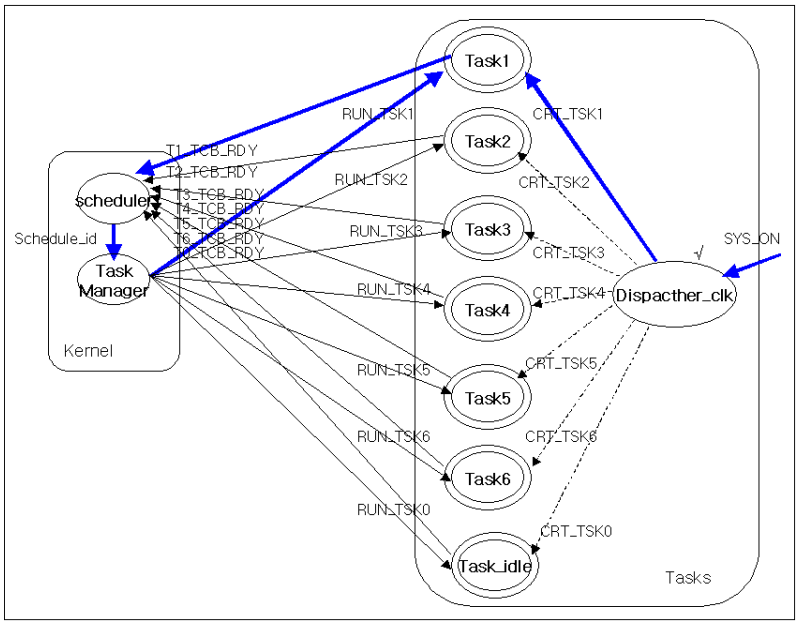

Figure 4.4 Applying the branch coverage criteria to the Task1

Table 4.2 Integration test data between the Kernel module and the Tasks module

Number	Integration test data		Expected output
1	SYS_ON = T	T1_TCB_RDY = T	RUN_TSK4 = T
2	SYS_ON = T	T1_TCB_RDY = T	RUN_TSK4 = T
3	SYS_ON = T	T1_TCB_RDY = T	RUN_TSK4 = T
4	SYS_ON = T	T1_TCB_RDY = T	RUN_TSK4 = T
5	SYS_ON = T	T1_TCB_RDY = T	RUN_TSK4 = T
6	SYS_ON = T	T1_TCB_RDY = T	RUN_TSK4 = T
7	SYS_ON = T	T1_TCB_RDY = T	RUN_TSK4 = T

As shown in the table 4.2, we generate seven test data between the Kernel module and the Tasks module by applying the branch coverage criteria, considering the final state in the sate-composition diagram.

4.3 System Testing

Purpose of the system testing is to check whether it shows behaviors as described in the requirement specification. In selecting system test data, we utilize the Activity chart. The procedure of selecting system test data is as follows:

(1) Extract scenarios from the Activity charts.

(2) Draw up the sequence diagrams according to the scenario.

[Converting the Activity chart to the sequence diagram]

① A Module of the Activity chart can be a class of the sequence diagram.

② Events and data of the Activity chart can be messages of the sequence diagram.

(3) Select system test data by identifying the input data according to the data flow of the sequence.

4.3.1 Scenarios

Scenarios extracted from the activity chart can be classified according to interrupt conditions. Here, we generate test data from the example without interrupt.

4.3.2 Sequence diagram

We compose the sequence diagram based on the Activity chart, which is shown in the figure 4.5.

4.3.3 Selection of the system test data

We select system test data by identifying input data according to the data flow in the sequence diagram. One case is shown in the figure 4.5, where input data is shown as a set of signals: SYS_ON, and T1_TCB_RDY ~ T4_TCB_RDY, which can be formalized as below:

System Test Data = (SYS_ON, T1_TCB_RDY, T2_TCB_RDY, T3_TCB_RDY,

T4_TCB_RDY)

Without an interrupt, there are totally sixteen system test data, as shown in the table 4.3. In case the 15th system test data defined as (T,F,T,T,T) is chosen, the expected output would be to initiate the ‘ Task 6’ .

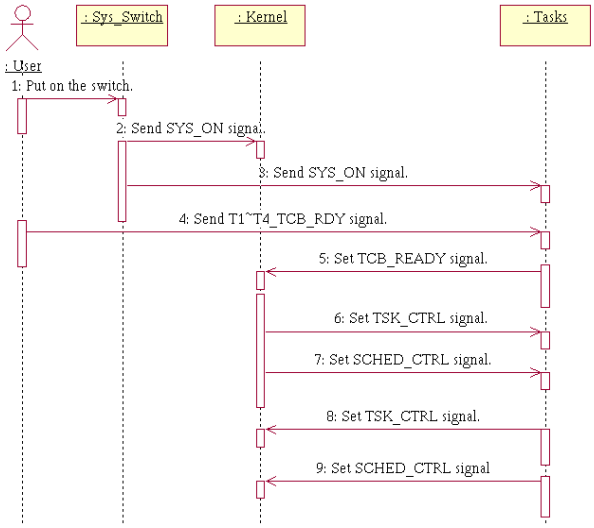

Figure 4.5 Sequence Diagram of the RTOS

Table 4.3 System test data without an interrupt

Number	System test data					Expected Output
	SYS_ON	T1_TCB_ RDY	T2_TCB_ RDY	T3_TCB_ RDY	T4_TCB_R DY	RUN_TSK6
1	T	F	F	F	F	F
2	T	F	F	F	T	F
3	T	F	F	T	F	F
4	T	F	T	F	F	F
5	T	T	F	F	F	F
6	T	F	F	T	T	T
7	T	F	T	F	T	T
8	T	T	F	F	T	T
9	T	F	T	T	F	T
10	T	T	F	T	F	T
11	T	T	T	F	F	T
12	T	T	T	T	F	T
13	T	T	T	F	T	T
14	T	T	F	T	T	T
15	T	F	T	T	T	T
16	T	T	T	T	T	T

5. EMBEDDED SYSTEM TESTING

5.1 System specification

Development procedure including verification of the RTOS software for the safety critical application is shown in previous sections. In this section, we show the testing to identify if it works well when it is embedded in the hardware, and also to check if there is any error when it is integrated with the application software. Test board is set up for this purpose. Used test board has an Intel 80C196kc CPU with RS-232 port as an interface

As the developed PLC is targeted for the Digital Plant Protection System (DPPS), we model the simplified DPPS function and generate code automatically as the same procedure shown for the RTOS software.

DPPS takes four redundant inputs of each variable, then, compares them with the defined set point. When the input exceeds the set point, it generates ‘ true’ value as an output. Results are transmitted to the next module, where results from each four redundant channels are gathered. If more than two values out of four monitored variables are shown as ‘ true’ , then it puts out a trip signal to initiate protection function. DPPS system software is simplified, and only one monitoring variable is modeled for the convenience of the embedded system testing.

5.2 Reliability assessment

In the nuclear industry, analog protection system has been operated over 30 years. Thus, reliability assessment methodology is well established for the analog systems. However, assessment of the reliability of digital systems with embedded software is different from that of analog system [3]. These days, it is suggested to include the development process as well as test results to assess reliability [14]. To show quantified reliability, we focus on the testing and analysis method of results. In order to ensure the reliability, the required number of fault-free test to satisfy requirement is calculated. If there occurs no failure, we can calculate the smallest number of required testing from the equation below [15];

$$\int_0^{p_0} \frac{(1-p)^n \, dp}{B(1,1+n)} \geq 1-\acute{a} \quad \ldots\ldots\ldots(1)$$

Here, α can be determined from the required confidence level, p_0 is the defined probability of failure on demand, which is set to 10^{-3} in our case, B (a,b) is a Beta function and n is the number of test executed. At first, we calculate the minimum required number of test assuming no error. Then, whenever we find error during the test run, we recalculate the required number of testing using the test results so far including failed run as suggested in the Bayesian approach [15].

Based on this, testing is performed to the implemented embedded system. Input data are consists of four truth-values and this determines whether the trip function would be initiated or not as an output.

Embedded System Test Data = (Bool_T1, Bool_T2, Bool_T3, Bool_T4)

Expected output of this testing, which is a demand for the trip function, can be shown as either true or false. We assign Boolean values randomly to generate system test data. If there are more than two ‘true’ values, it can be counted as one demand. To assure 10^{-3} pfd, we generate test data until the number of demand becomes as calculated according to the equation (1). In case it doesn’t show any error during testing, the number of required test would be n=4602, where ‘false’ case is excluded [5].

From this test, we can identify that the developed RTOS software satisfies reliability criteria when the application software is embedded in it and integrated with hardware.

6. Conclusion

In this work we show the overall procedure to specify, verify and test the RTOS for the safety-critical, nuclear applications. Requirements of the RTOS are analyzed and concluded as verifiable, since the required functions to the RTOS is simple and deterministic. After the verification, the software is automatically generated in C. This generated code is tested for each purpose. Unit testing, integration testing and system testing are performed as software testing. In the unit testing, we performed two types of testing; one is code-based testing to see whether there is any error, and the other is state-based testing to check if the developed software satisfies properties of the verified model. In the integration testing, we make out the state-composition diagram and generate test data by applying branch coverage criteria. In this test, we check whether any two or more modules communicate properly as we design in the model. In the system testing, we extract scenarios from the requirements specification to check overall system behavior. After verification and testing, we implement the embedded system as a prototype. We generate the simplified application software to be embedded in the RTOS, and then test the whole embedded system to check whether it works properly when integrated with the application software and also with the hardware. We calculate the required number of tests to satisfy reliability criteria, which will give us, quantified reliability.

References

[1] IAEA, “Harmonization of the licensing process for digital instrumentation and control systems in nuclear power plants”, IAEA-TECDOC-1327, 2002.

[2] EPRI TR-103291-Vol.1, Handbook for verification and validation of digital systems Vol. 1: Summary, 1994.

[3] U.S. NRC, “ Digital Instrumentation and control systems in nuclear power plants; safety and reliability issues”, National Academy Press, 1997.

[4] David Harel and Michal Politi, “Modeling Reactive Systems with Statecharts: The Statemate Approach”, I-Logix, 1999.

[5] David Harel, “STATECHART: A VISUAL FORMALISM FOR COMPLEX SYSTEMS”, Science of Computer Programming 8 (1987) pp231-274

[6] David Harel and Ammon Naamad, “The STATEMATE Semantics of STATECHARTs”, ACM Trans. Soft. Eng. Method. Oct. 1996

[7] “Simulation Reference Manual” , I-Logix, 2001.

[8] “ModelChecker Tutorial”, I-Logix, 2002.

[9] Edmund M.Clarke. jr, Orna Grumberg, Doron A. Peled. “Model Checking”, 1999, The MIT press

[10] Phillip A. Latlante. “Real-time System Design and Analysis an Engineer’s Handbook.” 2nd Edition, New York : IEEE press 1997

[11] M.R Lyu, J.R. Horgan and S. London, “A coverage analysis tool for the effectiveness of software testing”, in Proceeding of International Symposium on Software Reliability Engineering, 1993.

[12] J.R. Horgan and S. London, “ATAC : A data flow coverage testing tool for C”, in Proceedings of Symposium on Assessment of Quality Software Development Tools, pp2-10, New Orleans, LA, May, 1992.

[13] U.S.NRC., REGULATORY GUIDE, 1.171, Software Unit Testing for Digital Computer Software Used in Safety Systems of Nuclear Power Plants, National Academy Press , 1997.

[14] Debra S. Herrmann, “Sample implementation of the Littlewood holistic model for assessing Software quality, Safety and Reliability”, Proceedings annual reliability and maintainability symposium, pp.138-148, 1998.

[15] Bev Littlewood, David Wright, “Some conservative stopping rules for the operational testing of safety-critical software”, IEEE transactions on software engineering, Vol. 23, No. 11, pp. 673-683, 1997.