Master’s Thesis

석사 학위논문

Timing Testing in Model-Based Development of Safety-Assured Software for Medical Devices

Hyeon I Hwang (황 현 이 黃 鉉 伊)

Department of Information and Communication Engineering

정보통신융합전공

DGIST

2014

Timing Testing in Model-Based Development of Safety-Assured Software for Medical Devices

Advisor: Professor Sang Hyuk Son

Advisor: Professor Taejoon Park

Co-Advisor: Professor Soon Ju Kang

by

Hyeon I Hwang

Department of Information and Communication Engineering

DGIST

A thesis submitted to the faculty of DGIST in partial fulfillment of the requirements for the degree of Master of Science in the Department of Information and Communication Engineering. The study was conducted in accordance with Code of Research Ethics[1].

. . 2014.

Approved by

Professor Sang Hyuk Son (Signature)
(Advisor)

Professor Taejoon Park (Signature)
(Advisor)

Professor Soon Ju Kang (Signature)
(Co-Advisor)

1) Declaration of Ethical Conduct in Research: I, as a graduate student of DGIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

Timing Testing in Model-Based Development of Safety-Assured Software for Medical Devices

Hyeon I Hwang

Accepted in partial fulfillment of the requirements for the degree of Master of Science

. . 2014.

Head of Committee _________________ (Signature)

Prof. Sang Hyuk Son

Committee Member _________________ (Signature)

Prof. Taejoon Park

Committee Member _________________ (Signature)

Prof. Soon Ju Kang

MS/IC
201222018

황 현 이, Hyeon I Hwang, Timing Testing in Model-Based Development of Safety-Assured Software for Medical Devices. Department of Information and Communication Engineering, 2013, 42 p.
Advisors: Professor Son, Sang Hyuk and Professor Park, Taejoon
Co-Advisor: Professor Kang, Soon Ju

ABSTRACT

Medical devices are safety-critical systems that have been gradually becoming more complicated and software issues are increasingly on the rise. To solve this problem, the quality of the software for the medical devices should be thoroughly guaranteed through verification. Every possible state and path should be verified.

To enhance the quality of the software-related safety issues model-based development can be useful. In model-based development the purpose of formal modeling is to permit precise understanding, specification, and analysis of the system. Model-based development is a software development approach to design the system model and verify if the designed model meets every system requirement [1]. After verifying all of the safety requirements are satisfied, a C source code can be generated through the code generation tool.

However, there is a challenge for meeting the timing requirements in model-based development due to timing semantics mismatch between the model and the implemented system. Even though the model conforms to the timing requirement through the verification process, the implemented system from the model may not conform to the requirement. A layered approach for timing testing in the model-based development is proposed in this thesis. A four-variables model is used to test and measure the timing gap between the model and the implemented system. The R-testing is performed in order to detect the timing requirement violation and M-testing is performed in order to measure the timing gap between the abstract model and the implemented system. An infusion pump is used for the case study.

Keywords: model-based development, safety-assured software, medical devices, infusion pump, timing semantics mismatch.

Contents

List of Figures

List of Tables

Ⅰ. INTRODUCTION

With embedded systems getting smarter and more complicated, the amount of software for the systems is also increasing. This leads to more frequent system faults due to software errors and the importance of the software-related safety issues is on the rise. The issue is the lack of a systematic development process. Informal system design, typically relies on documentations and engineering practices. It is difficult to verify and validate the correctness of the early development stages in this case. In addition, the code is typically written from the informal system design in a manual fashion. This is error-prone, and requires extensive testing. This development process leads to lack of safety and inefficiency.

To enhance the quality of the software, model-based development is widely used in the industry and research institutions because the system is designed in a way that formal verification can be performed in the early stage of the development process. This approach enhances the safety and efficiency of the software development process.

However, there is a challenge of the timing testing in the model-based development. Testing methodologies have been studied in the model-based development [2], but the timing testing approach has not been studied well. In addition, model-based development is strong in the functional aspects, but weak in the timing aspects when verification is performed at the abstract model level because timing behavior is abstracted to avoid the verification complexity [3]. Furthermore, another reason why model-based development is weak in the timing aspects is that timing semantics mismatches between the abstract model and the implemented system. For example, UPPAAL [4] and Stateflow/Simulink [5, 6], which are the modeling tools, have instantaneous transition semantics. That is, the input and output transition occurs simultaneously taking zero time. The implemented system requires non-zero computation time to implement such semantics because some computation phases such as reading time, reading input, input-transition, writing output, and output-transition are taken. Even though the timing requirements are satisfied at the model level, they might not be satisfied at the implemented system level.

Hence, timing testing is essential to verify and validate that the timing requirements are satisfied not only in the model, but also in the implemented system. A layered approach is proposed in this thesis. Our approach has two levels testing which are R-testing and M-testing. R-testing checks if the implemented system meets the timing requirements. In this testing, only input (sensor) and output (actuator) information from the target platform is used. If the result of R-testing shows that the implemented system does not conform to the timing requirements, M-testing is followed. M-testing is performed in order to measure how much timing deviation occurs due to the semantics mismatches. In this testing, not only input (sensor) and output (actuator) information from the target platform, but also abstracted input and output of the automatically generated code from the abstract model are used. To perform the tests, testing points are necessary in terms of both physical environment and the automatically generated code from the model. In order to express the abstraction boundary of the implemented system Parnas' four-variables model is used [7]. *Monitored (m), input (i), output (o),* and *controlled (c)* variables are used in the four-variables model. Only *monitored (m)* and *controlled (c)* two variables are used for the R-testing. All four variables, *monitored (m), input (i), output (o),* and *controlled (c),* are used for the M-testing. The results of the testing provide a tester with a way to optimize the implemented system.

The proposed approach is applied to an infusion pump for the case study. The infusion pump is developed based on the model-based development. R-M testing is performed to check the conformance between the timing requirements and the implemented system.

The main contributions of this thesis are as follows:

- We present the test design using the four-variables model that expresses the abstraction boundary of the implemented system.
- We propose the layered approach for the timing requirements testing in the implemented system developed by the model-based development.
- We apply the layered approach to an infusion pump system for a case study in order to show the applicability.

II. BACKGROUND

1. Model-Based Development

First of all, safety requirements from the system experts should be documented. Formal modeling can be started from the understanding of the safety requirements. There are some modeling tools such as UPPAAL [4], Stateflow [5], Simulink [6], TIMES TOOL [8] or SCADE, but each of them has different modeling properties. Among them Simulink / Stateflow is chosen for our study. After finishing the modeling, verification should be performed thoroughly. Verification can be performed through the query language or simulation. If any safety requirement is not sufficient in the model, the model should be modified until all safety requirements are adequate. Once this step is finished, a C source code can be obtained by using the code generation tool. We first need to understand how the code works and then attach some interfacing code to interact with the hardware. In our case, a Patient Controlled Analgesic (PCA) infusion pump is used for the case study. Figure 1 shows the brief process of model-based development.

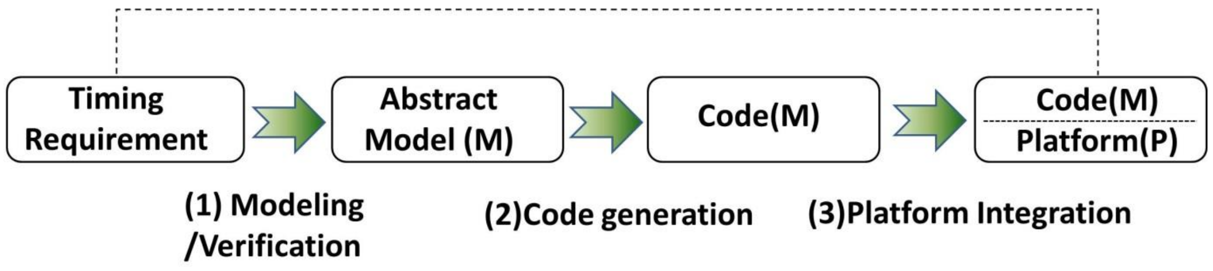


Figure 1. Process of model-based development

2. Patient Controlled Analgesic (PCA) Infusion Pump

There are many types of medical devices such as a pulse oximeter, pacemaker, and infusion pump. Among them, we have chosen an infusion pump. Infusion pumps are medical devices that deliver fluids, including nutrients and medications, into a patient’s body in a controlled manner. Infusion pumps are used worldwide in hospitals as well as in home care. Nevertheless, more than 56,000 reports of adverse events associated with the use of infusion pumps, including serious injuries and deaths were reported from 2005 through 2009 [9]. Medical devices are safety-critical systems because they are related to the patient’s life. The Food and Drug Administration (FDA) has recognized that this is a serious problem and launched the *Infusion Pump Improvement Initiative* [9]. Through this initiative, the FDA will take broad steps such as establishing additional requirements for infusion pump manufacturers, proactively facilitating device improvements, and increasing user awareness to prevent infusion pump problems. These are the reasons why infusion pumps have been chosen for our research. A PCA (Patient Controlled Analgesic) infusion pump has been selected specifically because it is one of the most common infusion pumps. The purpose of the PCA infusion pump is pain-relief treatment (e.g., morphine). The patient may request additional doses (called bolus) by pressing the “request” button attached to the pump. Then a small amount of drugs will be injected into the patient. The infusion pump mentioned in this thesis is the PCA infusion pump.

III. TIMING SEMANTICS MISMATCHES IN THE MODEL-BASED DEVELOPMENT

In this section, each step of the model-based development is handled through the infusion pump case study.

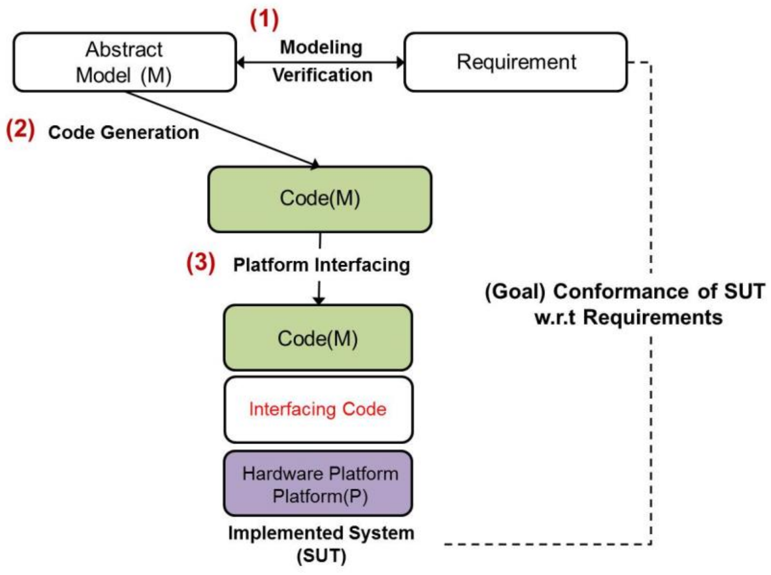


Figure 2. The goal of the layered approach in the model-based development

Figure 2 illustrates the goal of the proposed approach in the model-based development. In the step Figure 2-(1), modeling is performed with regard to the system requirements using the modeling tool such as UPPAAL or Simulink/Stateflow (Like mentioned in the BACKGROUND section, Simulink/Stateflow is chosen for our research). System requirements are provided as a document and some requirements of the infusion pump are written below.

- *(REQ1) A bolus dose shall be started within 100ms when requested by the patient.*
- *(REQ2) If the syringe becomes empty during infusion, an empty reservoir alert shall be issued and the current infusion should stop within 50ms.*

For example, Figure 3 shows the way to model the *REQ1*.

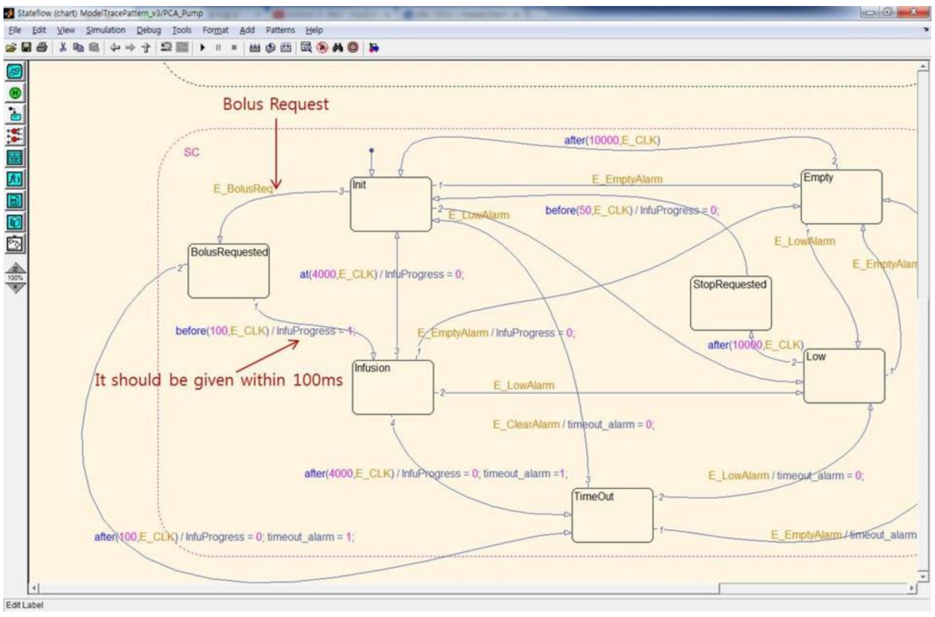

Stateflow (chart) ModelTracePattern_v3/PCA_Pump
File Edit View Simulation Debug Tools Format Add Patterns Help
Bolus Request
SC
E_BolusReq
Init
BolusRequested
at(4000,E_CLK) / InfuProgress = 0;
before(100,E_CLK) / InfuProgress = 1;
It should be given within 100ms
Infusion
E_EmptyAlarm / InfuProgress = 0;
after(4000,E_CLK) / InfuProgress = 0; timeout_alarm =1;
after(100,E_CLK) / InfuProgress = 0; timeout_alarm = 1;
after(10000,E_CLK)
E_EmptyAlarm
E_LowAlarm
before(50,E_CLK) / InfuProgress = 0;
Empty
E_LowAlarm
E_EmptyAlarm
StopRequested
after(10000,E_CLK)
Low
E_LowAlarm
E_ClearAlarm / timeout_alarm = 0;
E_LowAlarm / timeout_alarm = 0;
TimeOut
E_EmptyAlarm / timeout_alarm
Edit Label

Figure 3. Abstract model of the infusion pump developed by Simulink/Stateflow

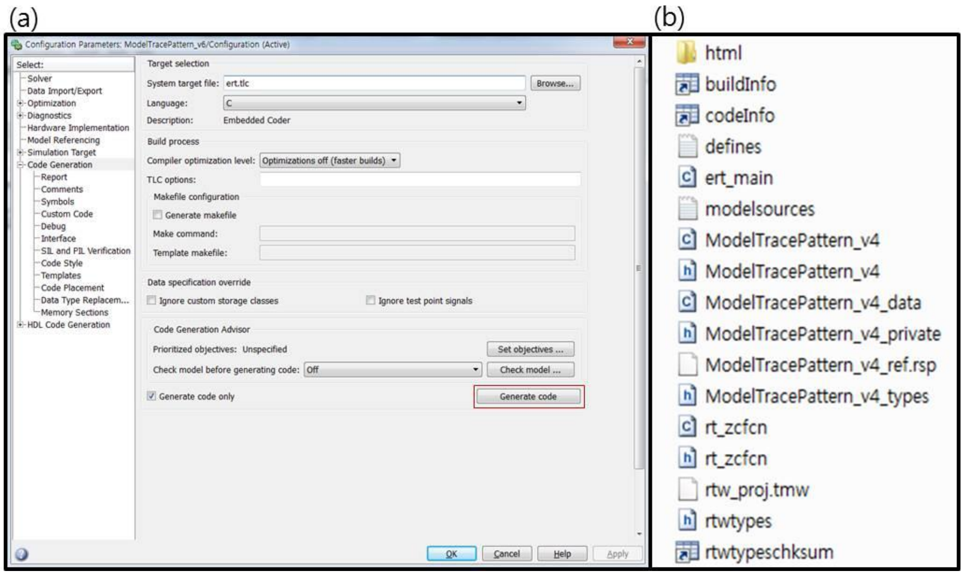
Figure 3 is a part of the abstract model of the infusion pump with regard to the system requirements using Simulink/Stateflow. When an event, *E_BolusReq*, is triggered from the environment (i.e., patients), state transition from *Init* state to *BolusRequested* state occurs. After this state transition from *BolusRequested* state to *Infusion* state occurs subsequently. At this transition the value of an output variable, *InfuProgress*, is changed from 0 to 1. *E_CLK* event implies the digital clock and *before(100, E_CLK)* is one of the syntax in the Simulnk/Stateflow to model the temporal logic. Once modeling is finished, the model can be verified through Simulink Design Verifier or model simulation. The model should be modified until the model verifies that it satisfies all the requirements. In our case-study 5 input events are fed into the Stateflow model from the external Simulink block and 4 outputs are produced from the external Simulink block. The variables of the 5 input events and the 4 outputs are defined here.

- *E_CLK event: This input event indicates the digital clock. It is used for the temporal* *logic such as before, after, or at in the Stateflow [10].*

- *E_BolusReq event: This input event indicates that the bolus request button is pressed from the environment (i.e., patients). When this event happens, the expected output is to operate the infusion pump motor so that the bolus is injected into the patient's body through the syringe.*
- *E_ClearAlarm event: This input event indicates that clear alarm button is pressed from the environment (i.e., patients). When this event happens, the expected output is to clear the buzzer alarm.*
- *E_EmptyRsv event: This input event indicates that there is no more of the drug in the syringe and that it is empty. When this event happens, the expected output is to raise the empty reservoir alarm and stop the infusion pump motor.*
- *E_LowRsv event: This input event indicates that the amount of the drug in the syringe is low. When this event happens, the expected output is to raise the low reservoir alarm and stop the infusion pump motor.*
- *Y.InfuProgress: This output variable interacts with the infusion pump motor. When the value of the Y.InfuProgress is 0, it implies that the motor does not operate. When the value of the Y.InfuProgress is 1, it implies that the motor operates.*
- *Y.empty_alarm, Y.low_alarm: These output variables interact with the buzzer alarm. When the value of the Y.empty_alarm or Y.low_alarm is 0, it implies that the empty reservoir alarm or low reservoir alarm is not raised. When the value of the Y.empty_alarm or Y.low_alarm is 1, it implies that the empty reservoir alarm or low reservoir alarm is raised. Y.empty_alram == 1 or Y.low_alarm == 1 is the expected output of the input event E_EmptyRsv or E_LowRsv respectively.*
- *Y.timeout_alarm: This output variable interacts with the buzzer alarm. The timeout alarm is raised when the violation of the timing requirements occurs.*

The empty reservoir alarm, low reservoir alarm, and timeout alarm are distinguished by the pulse width modulation associated with the alarming speed.

As shown in Figure 2-(2), automatic code generation is performed to generate source code (denoted as Code(M)) that preserves the model structure from the verified model by using Real-Time Workshop. The number and contents of the generated source and header files depend on which syntax is used in the model.



Target selection
System target file: ert.tlc
Language: C
Description: Embedded Coder

Build process
Compiler optimization level: Optimizations off (faster builds)
TLC options:
Makefile configuration
[ ] Generate makefile
Make command:
Template makefile:

Data specification override
[ ] Ignore custom storage classes
[ ] Ignore test point signals

Code Generation Advisor
Prioritized objectives: Unspecified
Check model before generating code: Off
[✓] Generate code only

html
buildInfo
codeInfo
defines
ert_main
modelsources
ModelTracePattern_v4
ModelTracePattern_v4
ModelTracePattern_v4_data
ModelTracePattern_v4_private
ModelTracePattern_v4_ref.rsp
ModelTracePattern_v4_types
rt_zcfcn
rt_zcfcn
rtw_proj.tmw
rtwtypes
rtwtypeschksum

Figure 4. (a) Real-Time Workshop, (b) generated header and source files from the model through Real-Time Workshop

Figure 4-(a) shows Real-Time Workshop window. There are some platform-dependent options to apply the generated code to the target platform (denoted as Platform(P)) properly. Figure 4-(b) shows the generated files from the model through automatic code generation. In our study 5 header and 4 source files are generated from the model.

Code(M) cannot execute alone on the Platform(P) without adding interfacing code to Code(M). As shown in Figure 2-(3), adding the interfacing code to Code(M) is performed. Contents of the interfacing code are dependent on the target platform, operating system, implementation method such as the dependency between tasks and the way to trigger the input event (e.g., sampling or interrupting method) and so forth. For example, the input/output interfacing code connects the input/output of the physical environment with the input/output variables of Code(M). To amplify the input/output interfacing code, when the patient presses the bolus request button, input interfacing code converts the electrical signal change into updating the input variable of Code(M) (i.e., *E_BolusReq*) from false to true. Likewise, when the output variable of Code(M) (e.g., *Y.InfuProgress*) is updated from false to true, the output interfacing code converts the update into generating the electrical signal change to operate the physical actuator of the infusion pump. In our case study the model consists of 4 inputs and 4 outputs. Therefore, 4 input interfacing codes and 4 output interfacing codes are added to Code(M).

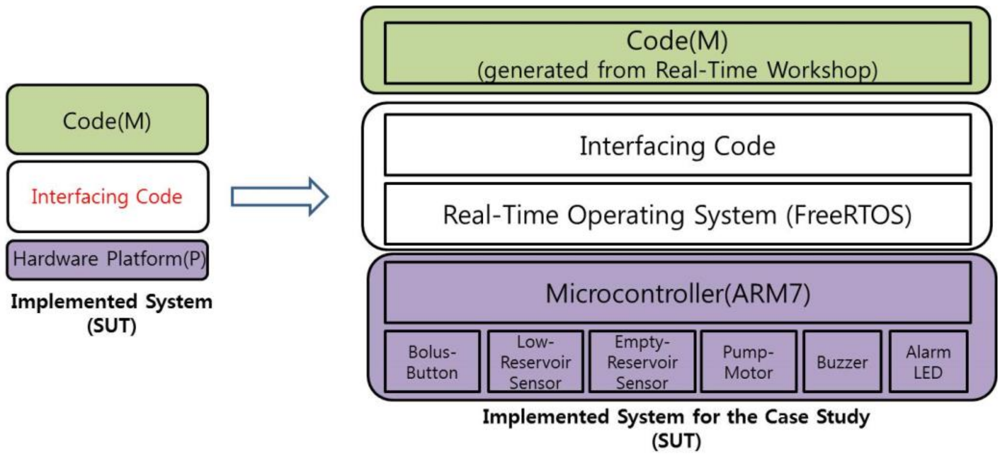
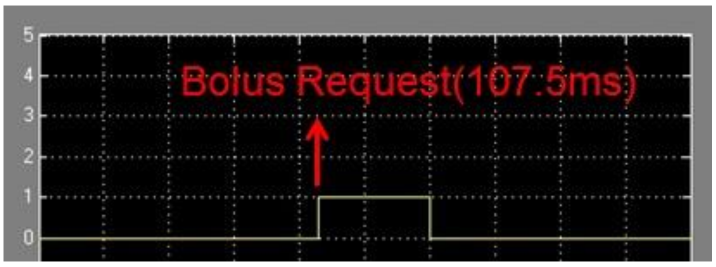


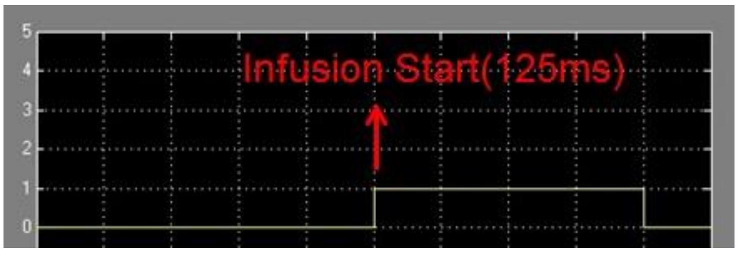
Figure 5. Integrated system for the infusion pump case study

Figure 5 illustrates the integrated system for the infusion pump case study. Sensors (e.g., bolus request button, low reservoir detecting sensor, empty reservoir detecting sensor, alarm clear button) and actuators (e.g., pump motor, buzzer) of the infusion pump hardware are interfaced with the micro-controller(ARM7) that operates FreeRTOS [11].

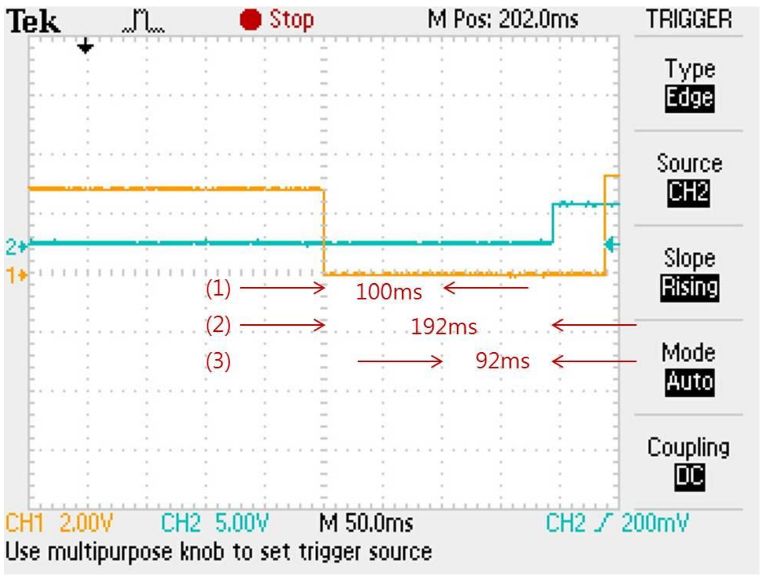
In the model-based development process, timing requirements might not be satisfied in the implemented system level even though the conformance of the timing requirements is verified in the model level. Figure 6 shows one example case.



(a)



(b)



(c)

Figure 6. (a) Time scope when event *E_BolusReq* is triggered in the model, (b) Time scope when bolus infusion starts in the model, (c) Time scope of event *E_BolusReq* and output infusion pump motor in the implemented system

Figure 6-(a) and (b) illustrate the system behavior in the model with regard to the timing requirement that is *a bolus dose shall be started within 100ms when requested by the patient*. When event *E_BolusReq* is triggered, the related output value (i.e., *Y.InfuProgress*) should be changed from false to true within 100ms. Figure 6-(a) and (b) show that it takes 17.5ms (125ms – 107.5ms = 17.5ms) from the event occurrence to the output value change, which means the timing requirement is satisfied in the model. However, Figure 6-(c) shows the violation of the timing requirement in the implemented system by using an oscilloscope. The orange and blue lines indicate the bolus request button and infusion pump motor respectively. In Figure 6-(c), section (1) (i.e., 100ms) indicates the constraint of the timing requirement. Section (2) (i.e., 192ms) indicates measured time from the bolus request to the infusion start and section (3) (i.e., 92ms) indicates the timing deviation between the timing requirement constraint and the implemented system. This timing assurance gap is made due to the abstraction of the timing aspects using a modeling language. It takes zero time to transit the states and produce the related output in the model, but it is hard to realize this timing semantics because some computation phases are necessary such as read input, input-transition, write output, and output-transition in the implemented system.

Due to the timing semantics mismatch between the model and the implemented system it is necessary to perform timing testing in order to assure if the timing requirements are also satisfied in the implemented system. It is not efficient if all testing with regard to the system requirements is performed because the conformance of the generated with regard to the functional requirement aspect is assured by automatic code generation. That is, we have confidence on Code(M) with regard to the functional requirement, so testing for the functional requirement aspect does not need to be performed. On the other hand, there is less confidence on the timing requirement aspect in the implemented system. Our goal is to check conformance of the implemented system with regard to the timing requirements and measure the timing deviation precisely so that the results of the test give some clue to the tester in order to optimize the final implemented system.

IV. THE LAYERED APPROACH FOR THE TIMING TESTING

In this section the four-variables model and two different levels of timing testing, R-testing and M-testing, are explained.

1. Mapping the four-variables to the implemented system

The uniform separation at the boundary between the hardware platform and the real environment is necessary to perform the timing testing precisely. The four-variables model is a famous technique in order to map the system requirements [7]. The four-variables model consists of monitored (*m*), input (*i*), output(*o*), and controlled (*c*) variables. The four variables are interfaced with the sensor device, actuator device, and the software of the system. The four-variables model is used for our layered approach for the timing testing.

Monitored (*m*) and Controlled (*c*) variables: monitored (*m*) and controlled (*c*) variables interact with the physical environment and the hardware platform. Monitored (m) variable is interfaced with the sensor or input devices such as bolus request button, clear alarm button, empty reservoir detecting sensor, or low reservoir detecting sensor in the infusion pump system. For example, once the bolus request button is pressed as an input event, the value of the relevant monitored (*m*) variable is changed from false to true. Another example is that when the event *E_EmptyRsv* is triggered, which means the syringe reservoir is empty, the value of the relevant monitored (*m*) variable is changed from false to true. Likewise, the controlled (*c*) variable is interfaced with the actuator or output devices such as the infusion pump motor or buzzer in the infusion pump system. For example, once the infusion pump motor as an output of an event *E_BolusReq* operates, the value of the relevant controlled (*c*) variable is changed from false to true. Contrariwise, once the infusion pump motor operation stops, the value of the controlled (*c*) variable is changed from true to false. Another example is that when the alarm as an output of an event *E_EmptyRsv* goes off, the value of the relevant controlled (*c*) variable is changed from false to true.

Input (*i*) and Output (*o*) variables: input (*i*) and output (*o*) variables interact with the software Code(M) that is automatically generated from the Real-Time Workshop. Code(M) includes each input variable related with the input events. That is, every input event, *E_CLK*, *E_BolusReq*, *E_ClearAlarm*, *E_EmptyRsv*, *E_LowRsv*, has their own variable in Code(M). Code(M) also includes the output variables. In our case-study the infusion pump system model produces 4 outputs for three types of alarms and the pump motor actuator. The value of the input and output variables in Code(M) is changed from false to true when the relative input event is triggered. For example, when event *E_BolusReq* is triggered, the input variable *U.E_BolsReq* and output variable *Y.InfuProgress* is changed from false to true. The value of the output variable *Y.InfuProgress* is true means that the infusion pump motor should operate for the bolus infusion. It is possible for the abstract model level to operate the pump motor once the event *E_BolusReq* is triggered. However, this is not possible for the implemented system because several computation phases such as reading input / output, writing input / output, state transition are necessary in the implemented system.

Four variables are useful to distinguish the part between the physical environment (e.g., infusion pump hardware platform) and the software (e.g., Code(M)).

2. Testing Objectives and Testing Ports

Even though the model conforms to the timing requirements through the verification process, the implemented system from the model may not conform to the requirement due to the assurance gap between the model and the implemented system.

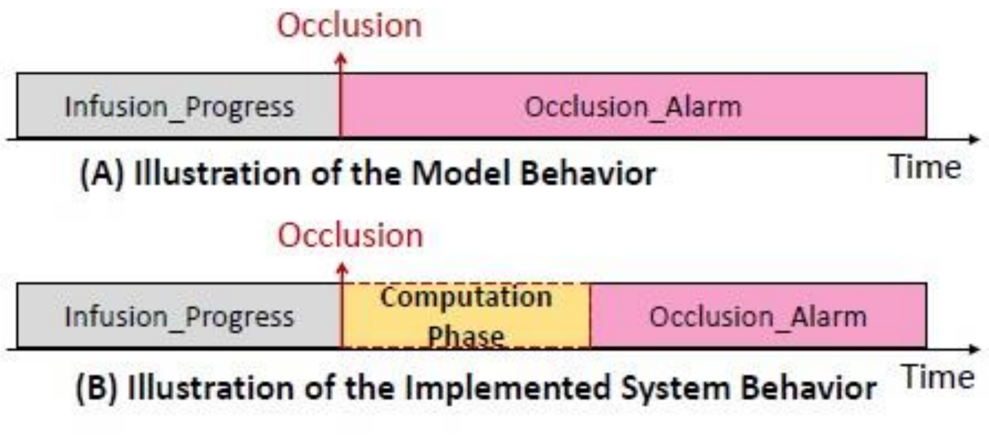


Figure 7. Assurance gap between the model and the implemented system

Figure 7 illustrates one brief example of the assurance gap between the model and the implemented system. If occlusion event is triggered during infusion, the pump motor operation should stop and raise the alarm immediately (i.e., Figure 7-(A), illustration of the model behavior). However, computation phase is necessary for reading time / input, input / output transition, writing output in the implemented system. So, the alarm cannot be raised immediately from the occlusion event. (i.e., Figure 7-(B), illustration of the implemented system behavior).

Therefore, even if the timing requirements are verified in the model, testing for the timing requirements should be performed also in the implemented system. A layered approach for the timing testing is proposed and explained in detail in section C. Our proposed timing testing has two main objectives. *Objective1* is to check whether the timing requirements are satisfied in the implemented system or not (i.e., R-testing) and *objective2* is to measure how much deviation exists in the implemented system and analyze the source of the deviation (i.e., M-testing). The result of the *objective1* is a yes or no according to the timing requirements constraints. On the other hand, the result of the *objective2* is three types of quantitative measurement (i.e., *input delay, output delay, Code(M) delay, transition delay*).

To perform this testing approach, four variables need to be defined as the port. Monitored, controlled, input, and output variables are defined as *m*-port, *c*-port, *i*-port, and *o*-port respectively. These four ports are used for the precise timestamp for the timing testing.

***m*-port:** *m*-port is interfaced with the hardware platform to detect the input events. For example, once event *E_BolusReq* is triggered (i.e., bolus request button is pressed by the patient), *m*-port generates the electrical signal changes.

***c*-port:** *c*-port is interfaced with the hardware platform to detect the operation of the output actuators. For example, once the infusion pump motor operates or stops, *c*-port generates the electrical signal changes.

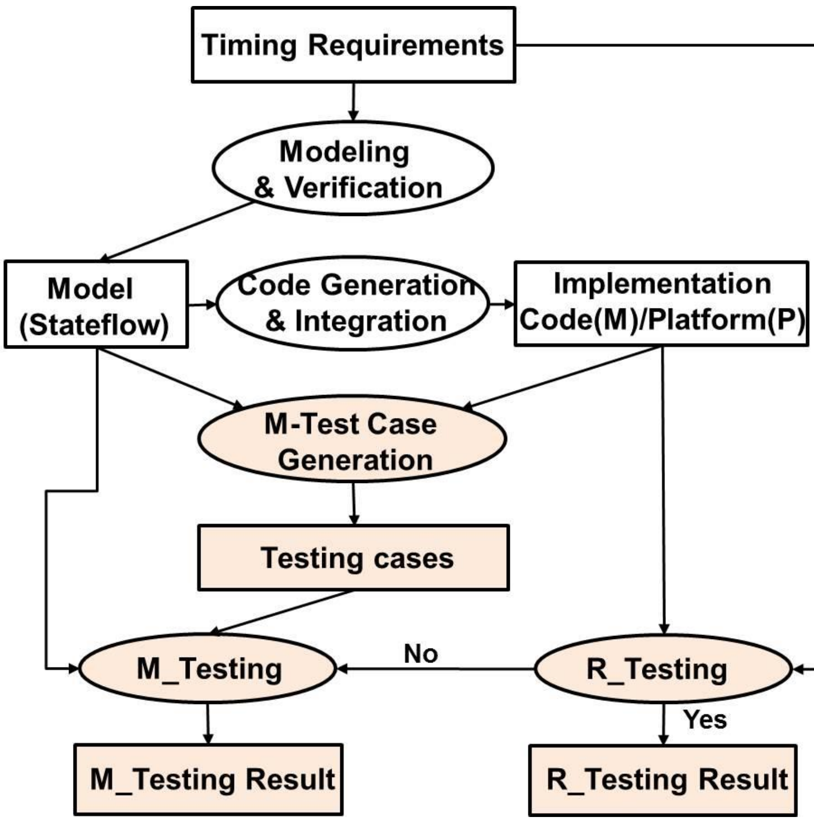
***i*-port:** *i*-port is interfaced with Code(M) to detect the input variable changes. For example, once the value of the input variable *U.E_BolusReq* is changed, *i*-port generates the electrical signal changes.

***o*-port:** *o*-port is interfaced with Code(M) to detect the output variable changes. For example, once the value of the output variable *Y.InfuProgress* is changed, *o*-port generates the electrical signal changes.

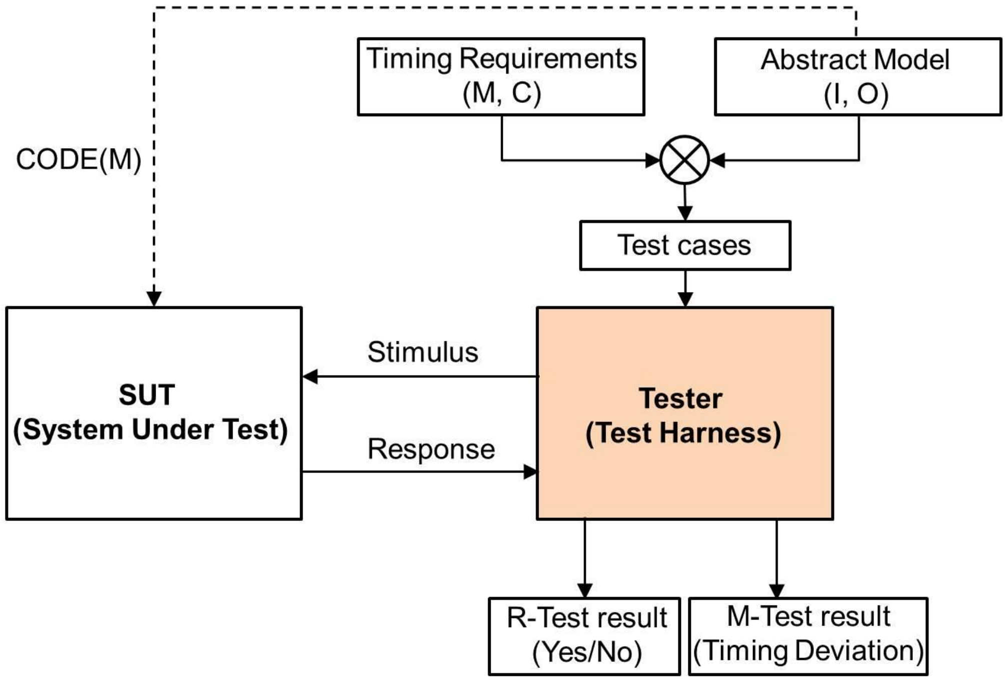
Timestamps from all ports are measured by using an oscilloscope.

3. R-testing and M-testing

A layered approach for the timing testing performs two different levels testing to achieve the *objective1* and *objective2*.



(a)



(b)

Figure 8. Experiment framework: (a) the overall testing framework (b) the R-M testing framework

R-testing: R-testing is performed before the M-testing. R-testing is performed to check whether the conformance of the implemented system with regard to the timing requirements is satisfied or not. If the result of the R-testing is a yes, M-testing does not need to be performed. On the other hand, if the result of the R-testing is a no, M-testing needs to be followed. Figure 8-(a) illustrates that only timing requirements information is provided to the tester for the R-testing. In terms of four variables, Figure 8-(b) illustrates that only monitored and controlled variables information is provided to the tester for the R-testing. That is, R-testing only utilizes the input / output of Platform(P) to test the timing requirements. To perform the R-testing some assumptions are needed. The assumptions are:

- *The R-tester should be able to change m variables.*
- *The R-tester should be able to observe the changes in c variables.*
- *The R-tester should be able to timestamp on the events associated with m and c variables.*

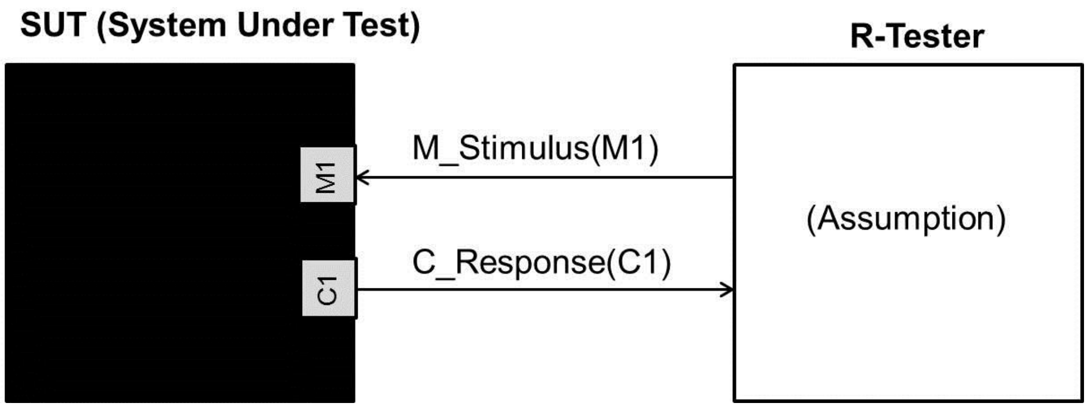
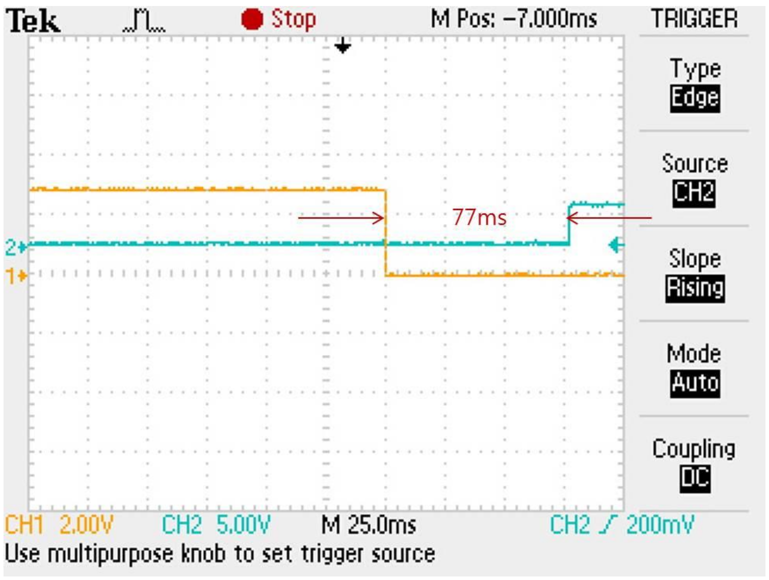


Figure 9. R-tester assumption and SUT (System Under Test)

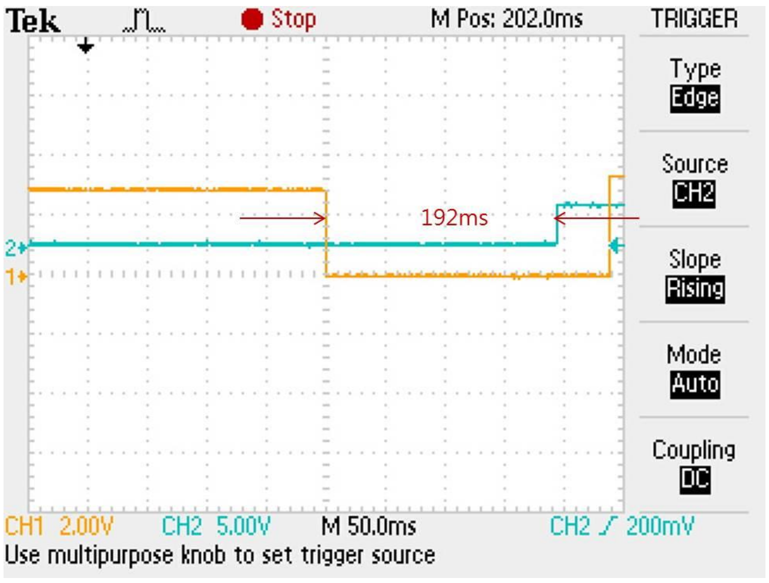
Figure 9 illustrates the R-testing framework. The R-tester has a *m* variable and *c* variable information as the stimulus and the response respectively. For example, the event *E_BolusReq* is defined as the *m* variable and the operation of the infusion pump motor is defined as the *c* variable in the *REQ1*. According to the timing requirement of the *REQ1*, the time difference between the timestamp of *m-stimulus* (t_{m1}) and *c-response* (t_{c1}) has to be less than 100ms. That is, $t_{c1} - t_{m1} \leq$ 100ms should be satisfied. One result among two cases can be generated according to the value of $t_{c1} - t_{m1}$.

- (*Result1*) *The result of the R-testing is a yes.*
- (*Result2*) *The result of the R-testing is a no.*

For example, if the time difference (i.e., $t_{c1} - t_{m1}$) is less than 100ms, it means that the timing requirement is satisfied (i.e., *Result1*) and the M-testing does not need to be followed. On the other hand, if the time difference is more than 100ms, it means that the conformance of the implemented system with regard to the timing requirement is not satisfied (i.e., *Result2*). Therefore, M-testing needs to be followed to measure the computation sections (e.g., *input / output delay*, *Code(M) delay*, *transition delay*) and analyze the main source of the deviation.



(a)



(b)

Figure 10. (a) Example of *Result1* of *REQ1*, (b) Example of *Result2* of *REQ1*

Figure 10 shows the two different results of the R-testing of *REQ1*. The orange and blue lines indicate the bolus request button (i.e., event *E_BolusReq*) and operation of infusion pump motor respectively. When the orange line shows a falling trigger, it is the time that the bolus request button is pressed (i.e., t_{m1}). When the blue line shows a rising trigger, it is the time that the infusion pump motor operates (i.e., t_{c1}). Figure 10-(a) shows the *Result1* of *REQ1* because $t_{c1} - t_{m1}$ = 77ms is less than 100ms. On the other hand, Figure 10-(b) shows the *Result2* of *REQ1* because $t_{c1} - t_{m1}$ = 192ms is more than 100ms and M-testing is followed.

M-testing: M-testing is performed if the result of R-testing is a *Result2*. M-testing is performed to measure the computation sections and analyze the main source of the deviation. Figure 8-(a) illustrates that abstract model information is provided to the tester for the M-testing. In terms of four variables, Figure 8-(b) illustrates that not only monitored and controlled variables but also input and output variables information is provided to the tester for the M-testing. That is, M-testing utilizes the input / output of Code(M) and Platform(P) to test the timing requirements. To perform the M-testing some assumptions are needed. The assumptions are:

- *The M-tester should be able to change m variables.*
- *The M-tester should be able to observe the changes in c variables.*
- *The M-tester should be able to timestamp on the events associated with m and c variables.*
- *The M-tester should be able to observe the changes in i variables.*
- *The M-tester should be able to observe the changes in o variables.*
- *The M-tester should be able to timestamp on the events associated with i and o variables.*

The first to the third assumptions are the same with R-testing assumptions and the fourth to the sixth assumptions are the additional M-testing assumptions.

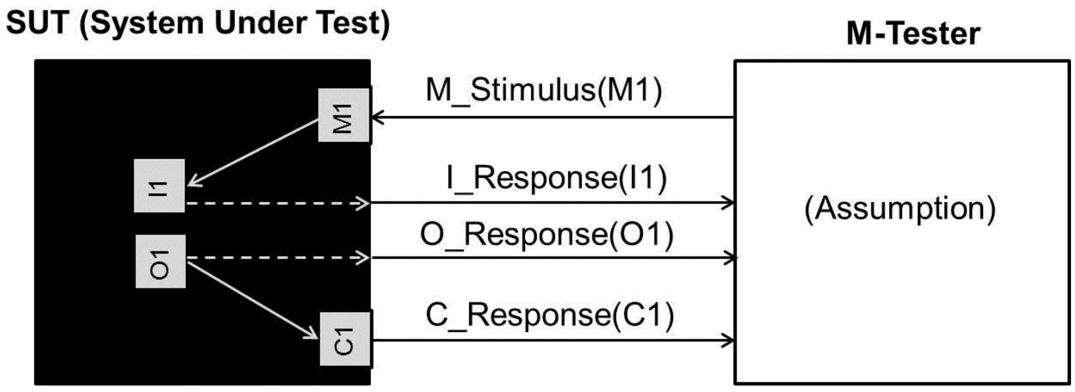


Figure 11. M-tester assumption and SUT (System Under Test)

Figure 11 illustrates the M-testing framework. The M-tester has a *m* variable and *i*, *o*, and *c* variables information as the stimulus and the response respectively. For example, event *E_BolusReq* is defined as the *m* variable and the operation of the infusion pump motor is defined as the *c* variable in the *REQ1*. And also, input variable *U.E_BolusReq* in Code(M) is defined as the *i* variable and output variable *Y.InfuProgress* in Code(M) is defined as the *o* variable in the *REQ1*. These four variables information is provided to the M-tester. The timestamp of *m-stimulus*, *c-response*, *i-response*, and *o-response* is denoted as t_{m1}, t_{c1}, t_{i1}, and t_{o1} respectively. M-testing has four categories which are *input delay*, *output delay*, *Code(M) delay*, and *transition delay*.

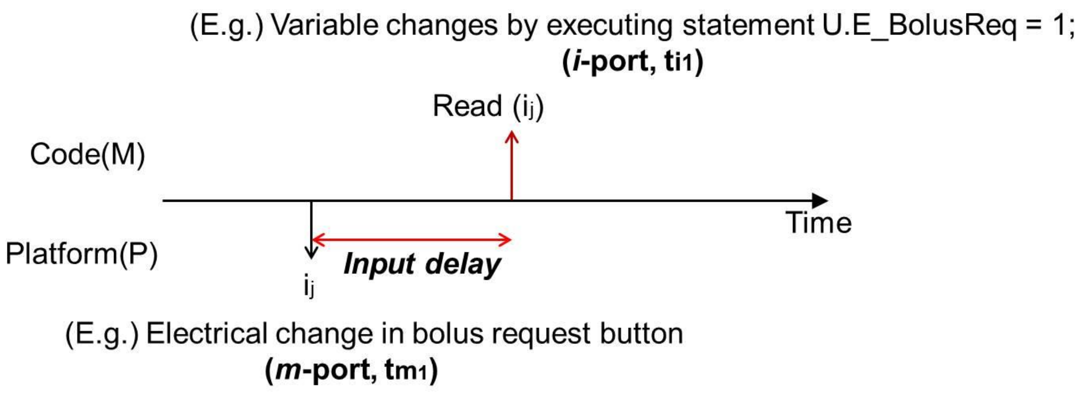
(1) ***Input delay***: In the abstract model, it takes zero time to read input and take the input-transition. However, in the implemented system, computation phase is necessary to read the input and this computation phase is denoted as *input delay* in this thesis. *Input delay* can be expressed by t_{i1} - t_{m1}. Figure 12-(a) illustrates the *input delay* in *REQ1* testing. *Input delay* is a time gap between the *m*-port and the *i*-port.

(2) ***Output delay***: In the abstract model, it takes zero time to write output and take the output-transition. However, computation phase is necessary to write the output in the implemented system and this computation phase is denoted as *output delay* in this thesis. *Output delay* can be expressed by t_{c1} - t_{o1}. Figure 12-(b) illustrates the *output delay* in *REQ1* testing. *Output delay* is a time gap between the *o*-port and the *c*-port.

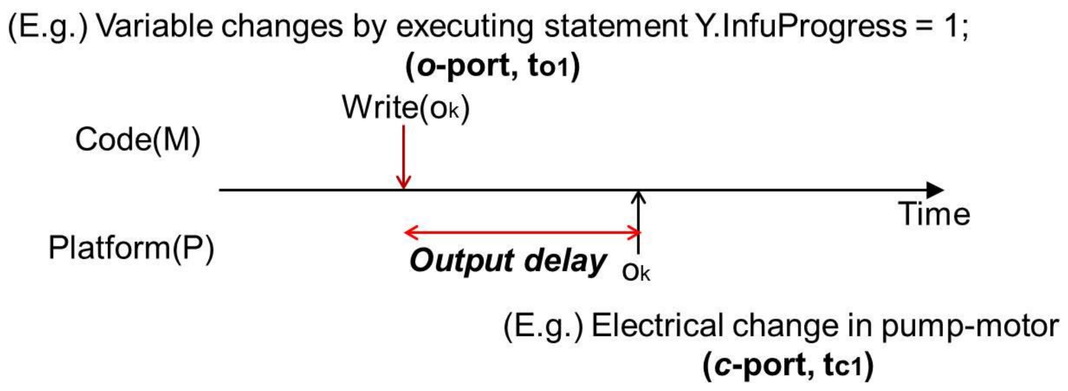
(3) ***Code(M) delay:*** In the abstract model, it takes zero time to process data such as reading time and input, writing output and taking input / output transition. However, computation phase is necessary to process data in the implemented system and this computation phase is denoted as *Code(M) delay* in this thesis. *Code(M) delay* can be expressed by t_{o1} - t_{i1}. Figure 12-(c) illustrates the *Code(M) delay* in *REQ1* testing. *Code(M) delay* is a time gap between the *i*-port and the *o*-port.

(4) ***Transition delay:*** In the abstract model, it takes zero time to take state transition. Furthermore, the model does not consider the computation phase of the function like an initialization function executed during the state transition process (denoted as transition function). So it also takes zero time to compute the function. However, computation phase is necessary to take state transition and perform the transition function and this computation phase is denoted as *transition delay* in this thesis. Figure 12-(d) illustrates

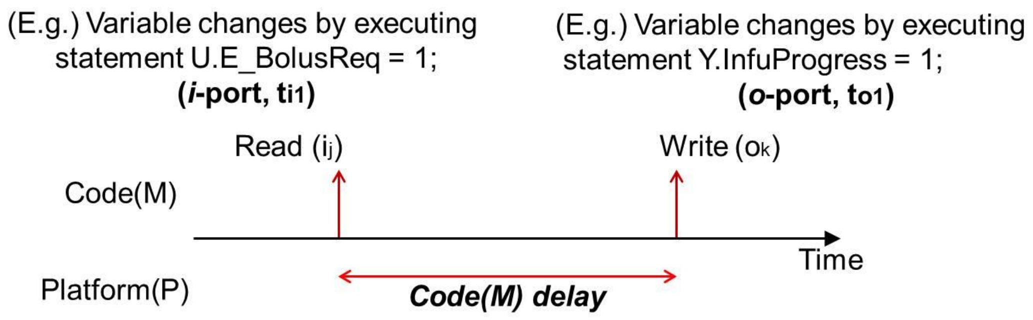
the *transition delay*. *Transition delay* is a time gap between the previous state and the current state.


(E.g.) Variable changes by executing statement U.E_BolusReq = 1;
(*i*-port, ti1)
Read (ij)
Code(M)
Time
Platform(P)
ij
Input delay
(E.g.) Electrical change in bolus request button
(*m*-port, tm1)

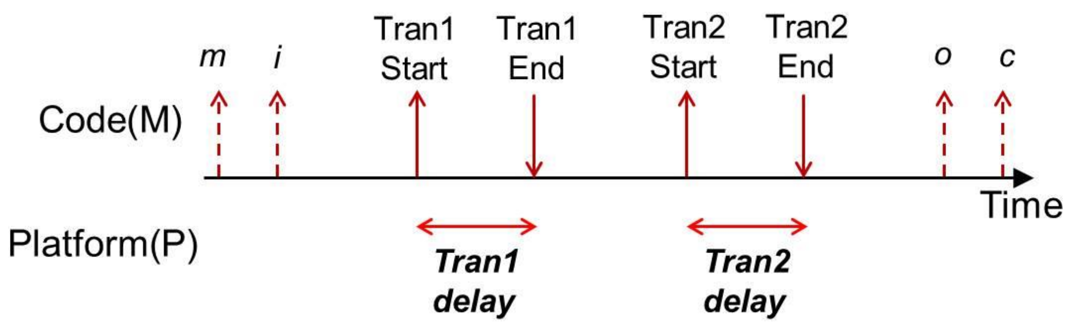
(a)


(E.g.) Variable changes by executing statement Y.InfuProgress = 1;
(*o*-port, to1)
Write(ok)
Code(M)
Time
Platform(P)
Output delay
ok
(E.g.) Electrical change in pump-motor
(*c*-port, tc1)

(b)


(E.g.) Variable changes by executing statement U.E_BolusReq = 1;
(*i*-port, ti1)
(E.g.) Variable changes by executing statement Y.InfuProgress = 1;
(*o*-port, to1)
Read (ij)
Write (ok)
Code(M)
Time
Platform(P)
Code(M) delay

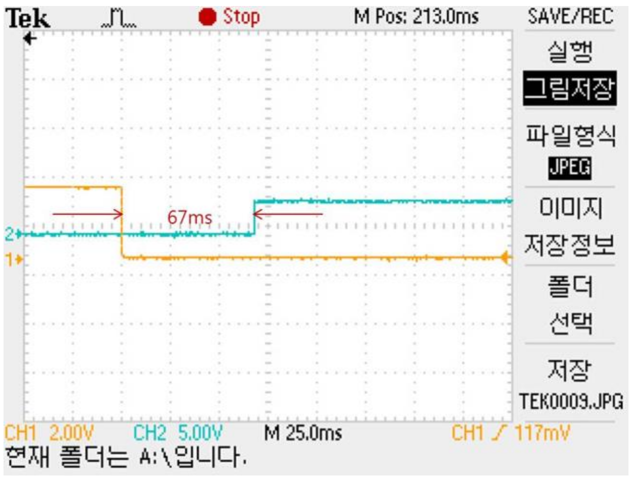
(c)



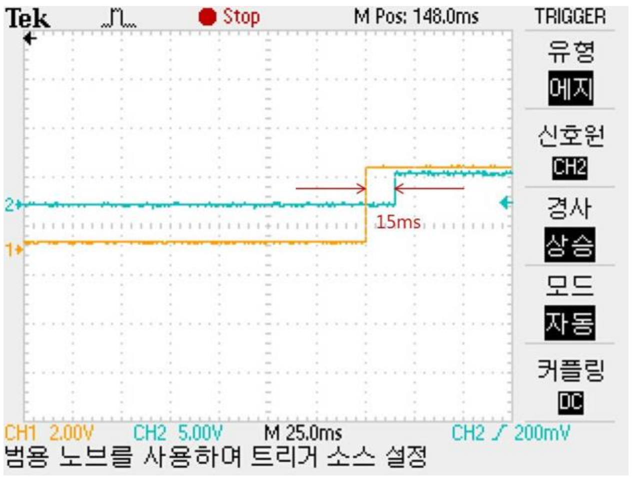
(d)

Figure 12. Four categories of M-testing: (a) input delay (b) output delay (c) Code(M) delay (d) transition delay

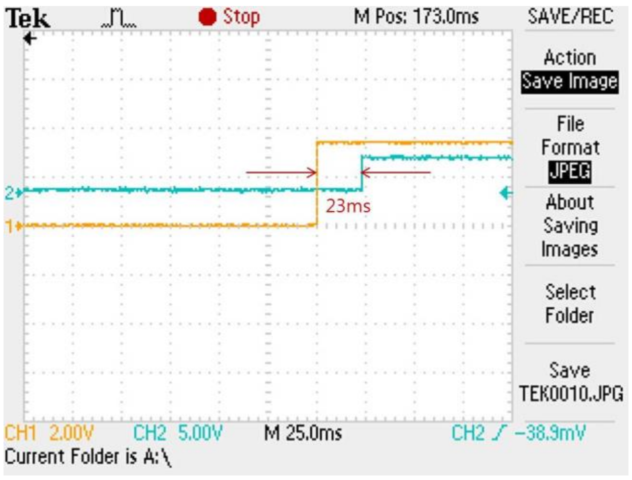
Figure 13 shows the measurement example of *input delay*, *output delay*, and *Code(M) delay* in the M-testing for *REQ1*. In Figure 13-(a) the orange line and blue line indicate the bolus request button (i.e., *m*-port) and an input variable (i.e., *i*-port) respectively. When the orange line shows a falling trigger, it is the time that the bolus request button is pressed. When the blue line shows a rising trigger, it is the time that the value of the input variable (i.e., *U.E_BolusReq*) is changed from false to true. In Figure 13-(b) the orange line and blue line indicate an output variable (i.e., *o*-port) and operation of the infusion pump motor (i.e., *c*-port) respectively. When the orange line shows a rising trigger, it is the time that the value of the output variable (i.e., *Y.InfuProgress*) is changed from false to true. When the blue line shows a rising trigger, it is the time that the infusion pump motor operates. In Figure 13-(c) the orange line and blue line indicate an input variable (i.e., *i*-port) and an output variable (i.e., *o*-port) respectively. The meaning of the orange line and blue line's trigger is the same with the explanation for *input delay* and *output delay*.



(a)



(b)



(c)

Figure 13. Measurement of M-testing for *REQ1*: (a) measurement of *input delay* (b) measurement of *output delay* (c) measurement of *Code(M) delay*

V. CASE STUDY: TIMING TESTING FOR INFUSION PUMP SYSTEMS

In this section, a proposed layered approach for the timing testing is performed for an infusion pump systems in order to show the applicability. We show how to detect the timing requirement violation through R-testing, and how to measure the timing deviation compared with the abstract model through M-testing.

Case-Study Setting: *REQ1* is considered as the timing requirement that needs to be satisfied not only in the abstract model, bus also in the implemented system. The infusion pump model is made through Stateflow/Simulink language [5, 6]. The final model consists of 15 states, 31 transitions, 5 inputs, and 4 outputs. The explanation for each input and output is handled in *timing semantics mismatches in the model-based development* section. Figure 14 shows a Simulink block diagram for the infusion pump system. It consists of 5 inputs, 4 outputs, and 1 chart (i.e., a block called PCA_Pump). The main functions of the model are implemented in the chart and the part of the model is shown in Figure 3. Verification of the model is performed through Simulink Design Verifier [12]. The model is modified until the conformance of all requirements is verified. Once the verification step is finished, the C source code can be automatically generated through the Real-Time Workshop. Five header and 4 source files are generated for our case study. In order to interact with the infusion pump hardware interfacing code is added into the Code(M) that is the generated code (this step is called integration step) and then Code(M) is ported into the micro-controller (ARM7). Finally, the sensors and actuators of the infusion pump hardware are interfaced with the micro-controller that operates FreeRTOS [11]. Baxter PCA Syringe Pump shown in Figure 15 is used as an infusion pump hardware. This pump provides 18 infusion modes, which fall into two major categories: continuous and timed infusions [13]. Timed infusion mode is used for our experiment.

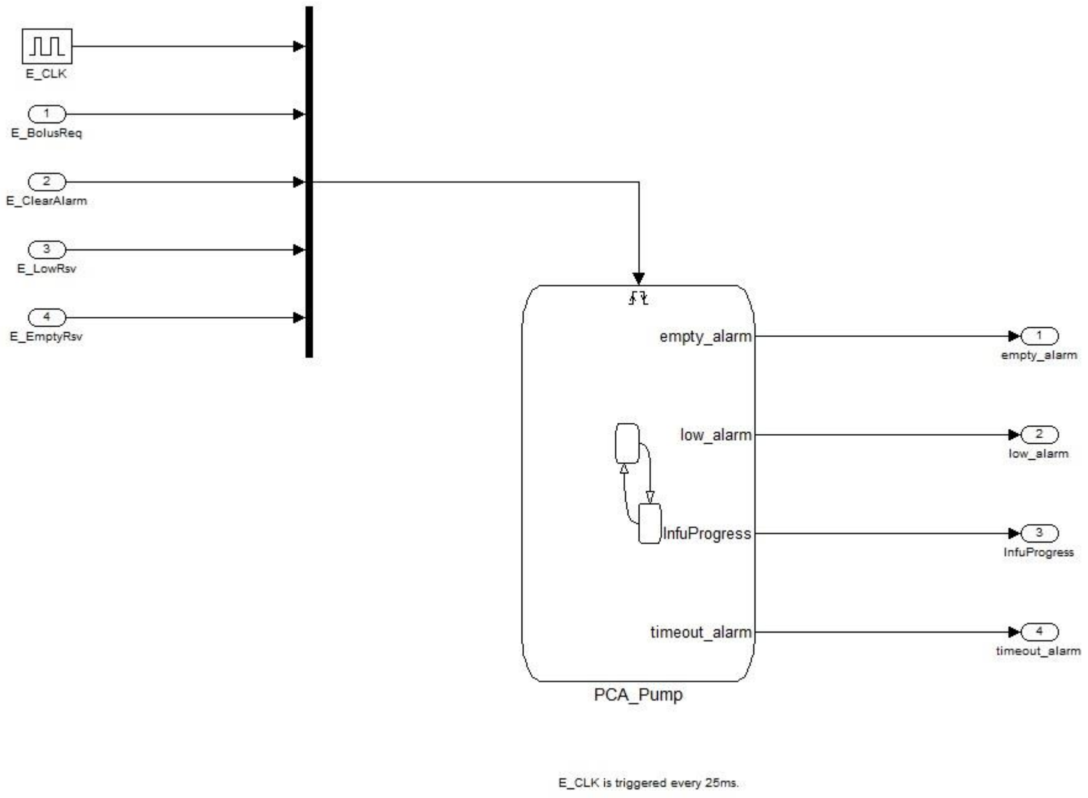


Figure 14. Simulink block diagram for infusion pump system

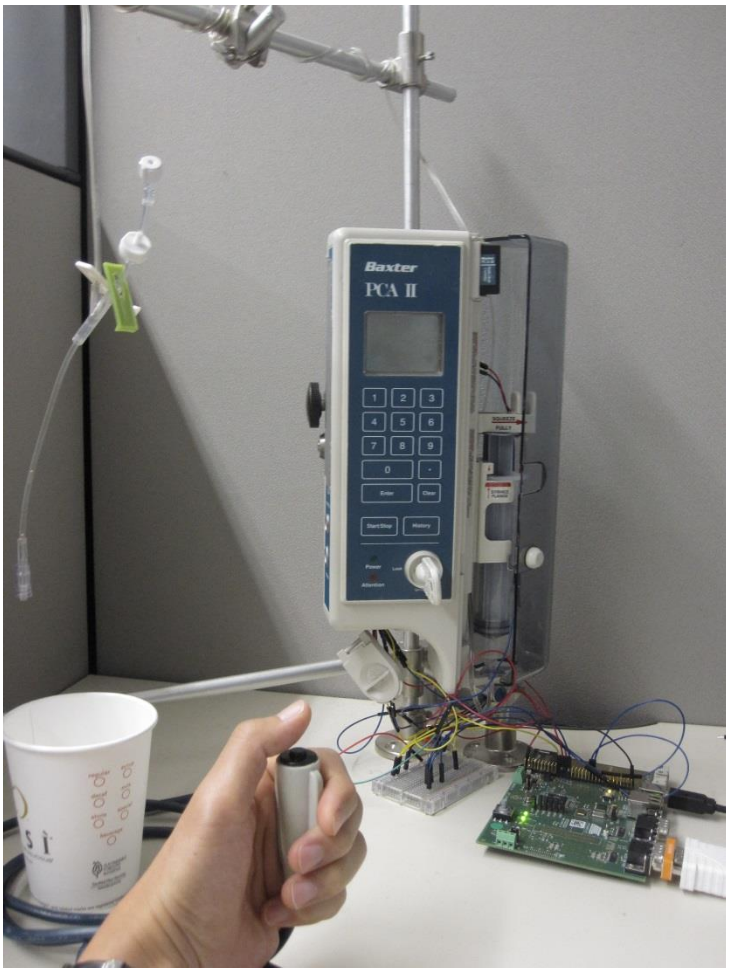
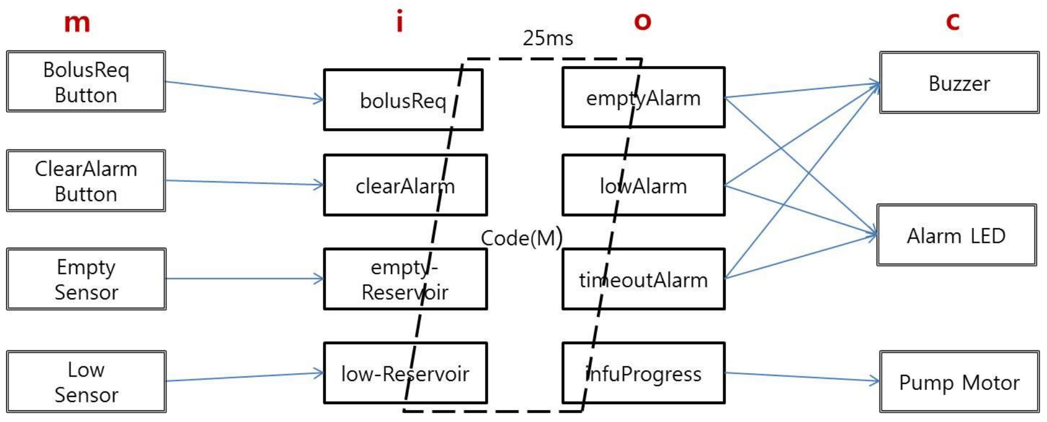
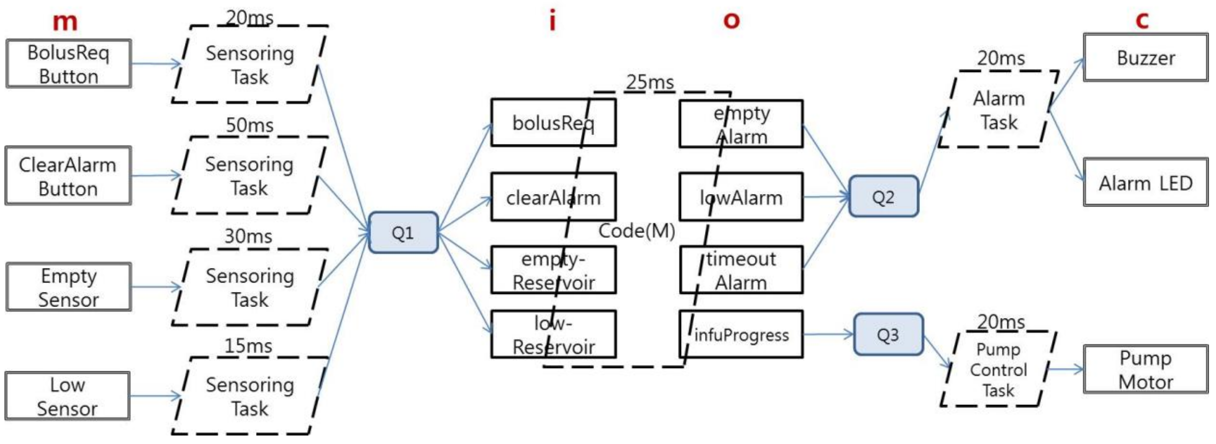


Figure 15. The experimental platform

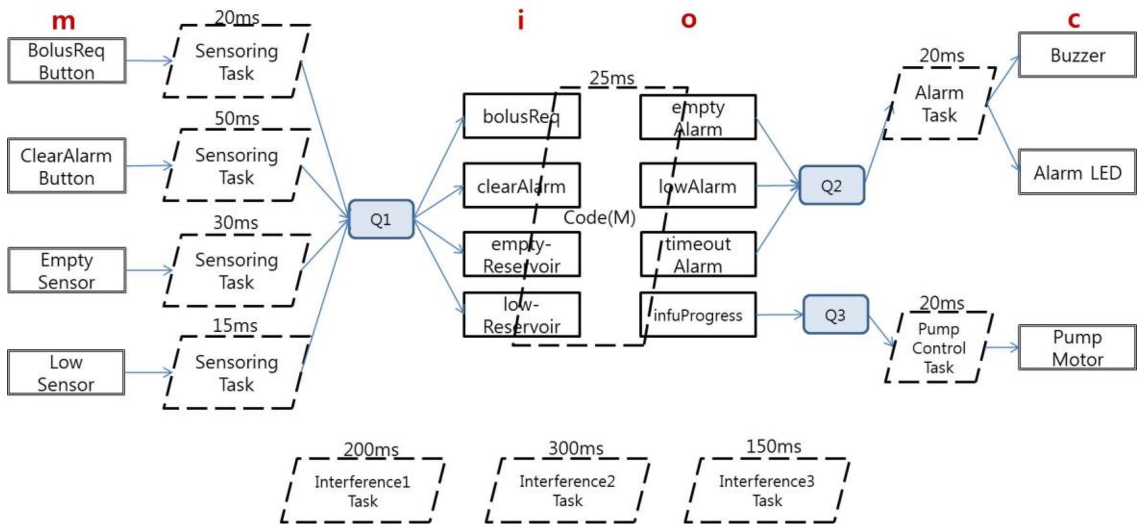
Case-Study Scenarios: In the integration step, there are various ways to add the interfacing code in terms of reading the input event, the number of thread, an operating system, and so forth. For example, an input event can be read by a periodic thread or an interrupt service routine. Similarly, software can be executed by a single thread or multiple threads. Therefore, the system can be implemented in various ways through the integration step. It is challenging to apply R-M testing to a wide range of different implemented systems. So, we create four different implemented systems and apply R-M testing to each of them. The ways to achieve these four different implemented systems are common in the integration step. An explanation of the four implemented systems follows:



(a)



(b)



(c)

Figure 16. The structure of the implemented systems: (a) *implementation1* (b) *implementation2* (c) *implementation3*

Implementation1: Figure 16-(a) illustrates the *implementation1*. This implemented system is executed by a single thread. That is, the sensing process to read inputs and the actuation process to write outputs are included in a single thread. The thread is executed every 25ms on FreeRTOS. Input events are read periodically, not by interrupt service routine.

Implementation2: Figure 16-(b) illustrates the *implementation2*. This implemented system is executed by the multiple threads. Sensing and actuation process come away from the Code(M) and create their own thread. Four sensing threads to detect *E_BolusReq*, *E_ClearAlarm*, *E_EmptyRsv*, and *E_LowRsv* events and 2 actuation threads to operate the infusion pump motor and alarming buzzer are created in the *implementation2*. So, there are total 7 threads including the thread executing Code(M) called Code(M) thread. In Figure 16-(b) the numbers mean the trigger periods of each thread. The threads are scheduled by FreeRTOS and Code(M) thread is executed every 25ms like *implementation1*. The communication between the sensing / actuation threads and the Code(M) thread is achieved by FIFO queues. Three queues are used. One is used to transmit input events from the sensing threads to the Code(M) thread, another is used to transmit output data for the infusion pump motor (i.e., the value of *Y.InfuProgress*) from the Code(M) thread to the motor actuation thread, the other is used to transmit output data for the alarming buzzer (i.e., the value of *Y.empty_alarm*, *Y.low_alarm,* and *Y.timeout_alarm*) from the Code(M) thread to the buzzer actuation thread. The summation of the period of sensing, the Code(M), and actuation threads is less than 100ms in order to make sure the value of the *c* variable is changed within 100ms after the value of the *m* variable is changed.

Implementation3: Figure 16-(c) illustrates the *implementation3*. This implemented system is the same as *implementation2* except 3 additional threads, called interference threads, that do not interact with Code(M), are added. The reason why the interference threads are added to *implementation2* is that our infusion pump system is simpler than today's smart infusion pump. Furthermore, model-based development is not concerned with the whole system structure, such as communication with other devices, due to the model complexity. Therefore, interference threads are added in order to assume that they make the implemented system more complicated. The interference threads execute their own tasks, not associated with Code(M). One of the threads has the same priority with the Code(M) thread, and the other two threads have a higher and a lower priority than the Code(M) thread respectively. Two different testing scenarios are made in *implementation3*. One is the testing in *REQ1* that are the same with the *implementation1*, *2*, and *4* (denoted as *scenario1*). A time constraint 100ms is the deadline to meet the timing requirement. On the other hand, a time constraint of another testing scenario (denoted as *scenario2*) is the time duration, not the deadline. The *scenario2* is explained later in detail.

Implementation4: This implemented system is the same as *implementation3* except the additional functions executed during the state transition process. The reason why the functions are added during the state transition process is to measure the *transition delay*. Graphical functions [14] in Stateflow are added into the state transitions in the abstract model. Figure 17 illustrates one of the added graphical functions that execute the loop for a certain time. State transitions occur twice from *m* variable == 1 to *c* variable == 1 in *REQ1* testing. So, two graphical functions are added into the abstract model. In the model, execution time of graphical function is not considered in the verification step. Execution time of graphical functions that have effects on the *transition delay* might lead to the violation of the timing requirement in the implemented system.

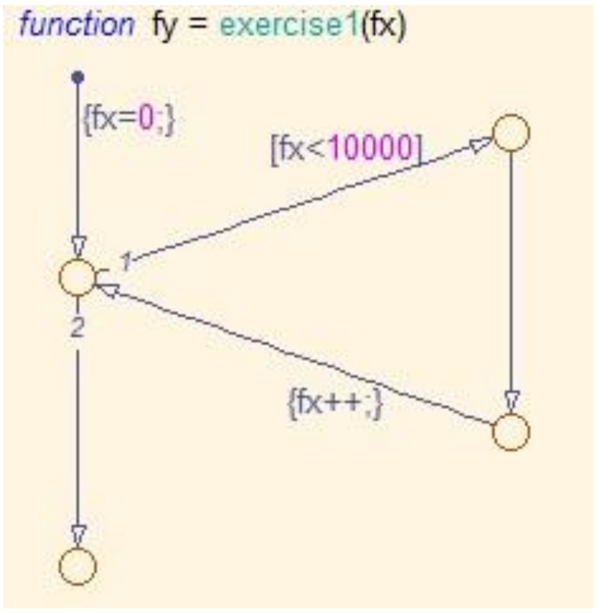


Figure 17. Graphical function in Stateflow

Test num	Implementation 1		Implementation 2				Implementation 3			
				M-Testing				M-Testing		
	R-Testing (m, c)	M-Testing	R-Testing (m, c)	Input delay (m, i)	CODE(M) delay (i, o)	Output delay (o, c)	R-Testing (m, c)	Input delay (m, i)	CODE(M) delay (i, o)	Output delay (o, c)
1	44	-	77	38	25	15	228	190	23	15
2	44	-	81	41	25	15	192	156	23	16
3	43	-	61	22	25	15	Max	MAX	-	-
4	49	-	MAX	MAX	-	-	105	67	23	15
5	30	-	220	181	25	15	205	170	22	15
6	49	-	123	82	26	15	168	181	23	15
7	45	-	104	64	25	15	219	181	23	16
8	49	-	79	40	25	15	149	108	23	15
9	31	-	89	47	25	15	Max	MAX	-	-
10	27	-	MAX	MAX	-	-	119	80	24	15

Table 1. R-M testing results in *REQ1*

1. Scenario1

Table 1 is the experimental result of the R-M testing in *REQ1*. The time is measured by an oscilloscope that detects the electrical signal change of *m*, *c*, *i*, and *o* port. The time unit is millisecond. If the result of the R-testing is more than 100ms, M-testing is followed in order to measure *input delay*, *Code(M) delay*, and *output delay*. Timing testing is performed for *implementation1*, *2*, *3*, and *4* (testing result of *implementation4* is explained separately). Red numbers in the R-testing columns indicate that the conformance of the implemented system with regard to *REQ1* is not satisfied. In other words, a bolus dose is not started within 100ms when requested. MAX in the R-testing columns indicates event loss. The infusion pump motor does not operate even if the bolus request button is pressed. More than 10 tests are performed, but table 1 includes only 10 selected tests in order to show the proper results. However, 10 tests samples are selected in order to maintain the probability of the violation of the timing requirement.

There is no violation of the timing requirement in *implementation1*. Half of the R-testing results for *implementation2* show the timing requirement violation and then M-testing is followed in order to measure *input delay*, *Code(M) delay*, and *output delay*. *Input delay* has a bigger effect on the requirement violation than *Code(M) delay* and *output delay*. Most R-testing results for *implementation3* show the requirement violation and then also M-testing is followed. The main source of the violation is similar to the one for *implementation2*. *Input delay* also has a bigger effect on the requirement violation than *Code(M) delay* and *output delay*. The results of M-testing can give some clues to optimize the implemented system. For example, *input delay* can be reduced by increasing the sampling rate of the sensing thread. And also, queue size needs to be extended in order to remove the event loss such as 4 and 10 of *implementation2* and 3 and 9 of *implementation3* cases.

The R-M testing result for *implementation4* is explained separately because *transition delay* needs to be measured only when the transition function is included in the abstract model. The results of R-testing and M-testing (input delay, Code(M) delay, output delay) for *implementation4* is similar to the results for *implementation3*. Most R-testing results show the requirement violation. Table 2 includes only one test sample because the results of the *transition delay* are consistent. The time unit is millisecond. In table 2, transition1 indicates the transition from *Init* state to *BolusRequested* state and transition2 indicates the transition from *BolusRequested* state to *Infusion* state in Figure 3. If the transition function executed during state transition process is more complicated, *transition delay* would be more increased. In this case, *transition delay* might have a bigger effect on the timing requirement violation than *input delay*. Therefore, *transition delay* needs to be measured when the transition function is included in the model.

Transition1 delay	Transition2 delay
11	20

Table 2. Transition delay of M-testing in REQ1

2. Scenario2

The time constraint of the R-M testing in *REQ1* (i.e., *scenario1*) is the deadline, 100ms. On the other hand, the time constraint of the *scenario2* is the time duration. For example, let us assume that a bolus dose shall be infused for 2 seconds when requested by the patient (duration of the infusion is set before the treatment according to the patient’s condition, the type of drug and so forth). As shown in Figure 18, the conformance of the abstract model with regard to the duration of the infusion (i.e., 2 seconds) is guaranteed. Nevertheless, the timing requirement might not be satisfied in the implemented system due to the *input delay*, *Code(M) delay*, and *output delay*. In order to perform the *scenario2* experiment, the followed assumptions are needed:

- *A bolus dose shall be infused for 2 seconds when requested by the patient.*
- *The above duration of the infusion is set based on the patient's condition and the type of drug.*

These assumptions are used as the timing requirement for the R-M testing in the *scenario2* experiment.

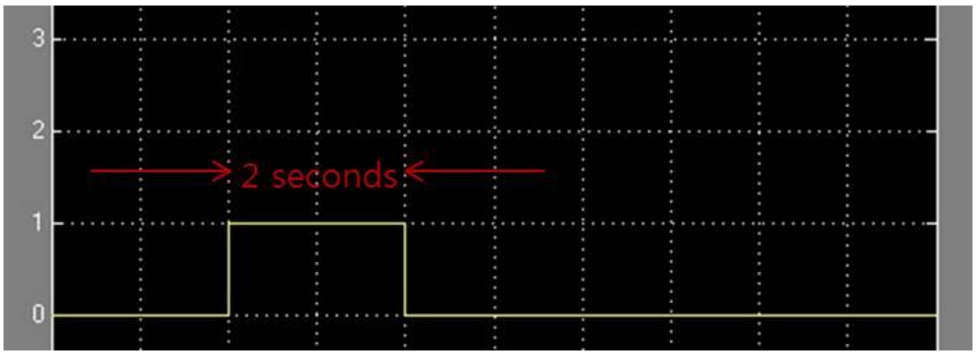


Figure 18. Time scope for the duration of the infusion in the abstract model

The timing gap between the model and the implemented system, especially in terms of the time duration, might be small. However, if the small timing gap is accumulated for a long time, it might cause some problem. For example, it is assumed that a bolus dose shall be infused for 2 seconds when requested by the patient. A bolus dose might not be infused for 2 seconds due to *input delay*, *Code(M) delay*, and *output delay* in the implemented system. The more a bolus request button is pressed, the more the difference of amount of injection between the model and the implemented system is increased. This would lead to the over or under-infusion. Over-infusion and under-infusion are considered as the common sources of the infusion pump accidents [15]. Definition of the over / under-infusion is provided for the accurate understanding [16].

- *Over-infusion: A situation in which more fluid or medication than is intended is* *delivered to the patient.*

- *Under-infusion: A situation in which less fluid or medication than is intended is delivered to the patient.*

If the drug is critical such as morphine, over or under infusion has great effect on the patient. Therefore, testing for the time duration needs to be performed. Table 3 is the experimental result of the R-M testing in *scenario2*. Time unit is millisecond. If the result of the R-testing is not 2 seconds, the M-testing is followed in order to measure *input delay*, *Code(M) delay*, and *output delay*. The M-testing is performed twice for one R-testing in *scenario2*. One is performed at the start of the actuation (actuation start section in Table 3) and another is performed at the end of the actuation (actuation end section in Table 3). Figure 19 illustrates the reason why the M-testing needs to be performed twice for one R-testing in *scenario2*. A and B implies the timing gap between the model and the implemented system at the start of the actuation (e.g., operation of an infusion pump motor starts) and at the end of the actuation (e.g., operation of an infusion pump motor ends) respectively. As shown Figure 19, the R-testing result for the time duration requirement depends on the value of the | A - B |. If the value of the | A – B | is 0, it means that the time duration requirement is satisfied in the implemented system. On the other hand, if the value of the | A – B | is not 0, it means that the time duration requirement is violated and there is a timing gap between the model and the implemented system as much as the value of the | A – B |. Therefore, the delay A and B need to be measured by M-testing.

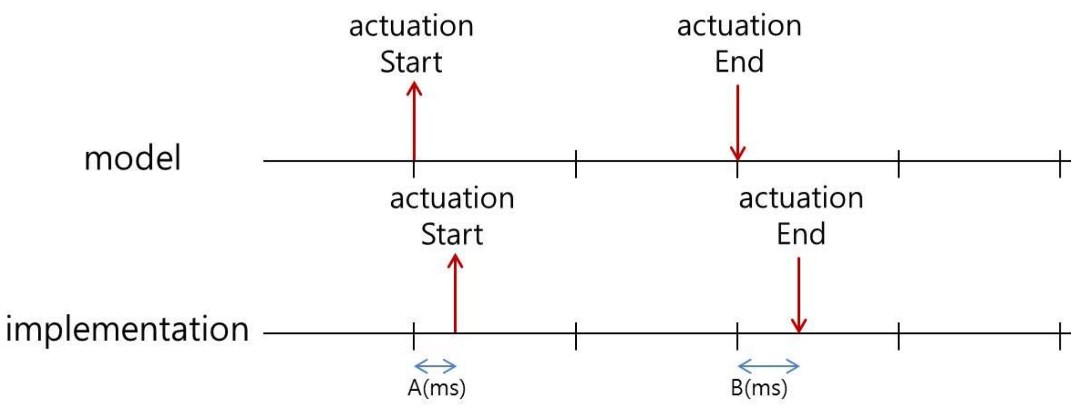


Figure 19. Timing gap between the model and the implemented system in scenario2

As shown in Table 3, R-testing result show the timing requirement violation (i.e., the value of the | A – B | in Figure 19 is 25ms and this is an under-infusion case) and two M-testing are followed in order to measure *input delay*, *Code(M) delay*, and *output delay* at the actuation start and end section. Both of the *input delay* are almost 0. In order to meet the time duration requirement the value of the A and B in Figure 19 have to be same. *Code(M) delay* in the actuation start / end section are quite different. Therefore, *Code(M) delay* in the actuation start section needs to be reduced in order to optimize the implemented system.

R-Testing	Actuation start section			Actuation end section		
	M-Testing			M-Testing		
	Input delay (m, i)	CODE(M) delay (i, o)	Output delay (o, c)	Input delay (m, i)	CODE(M) delay (i, o)	Output delay (o, c)
1975	0	25	17	0	3	14

Table 3. R-M testing results in scenario2

In order to measure the accumulated difference of amount of injection between the model and the implemented system a bolus is requested 150 times. As shown in Table 4, total accumulated difference of amount of injection is 4.6979g for 150 times bolus request. This under-infusion is considered as the common sources of the infusion pump accidents. Therefore, the R-M testing for the time duration needs to be performed in the implemented system.

	Amount of injection (g)
Model	18.7272
Implemented system	14.0293
The difference between the model and the Implemented system	4.6979

Table 4. Accumulated amount of injection for 150 times bolus request

VI. RELATED WORK

Abstraction is also used for the testing in the model-based development. In [3], abstractions of the model such as the functional abstraction, data abstraction, communication abstraction, and temporal abstraction are used for the testing in order to reduce the model complexity. However, they also point out the limitations of the temporal abstraction.

Test case generation technologies have been developed well in the various ways [17, 18, 19]. And also, some tools to test the systems are developed [20]. However, most of the test case generation technologies and the testing tools do not consider the non-functional requirements, especially the timing aspects even if testing real-time requirements is an important issue, in particular in embedded systems.

The Linux Trace Toolkit (LLT) is a set of tools that is designed to log program execution details from a patched Linux kernel and then perform various analyses on them, using console-based and graphical tools. LLT allows the user to see in-depth information about the processes that were running during the trace period, including when context switches occurred. Especially, in terms of time analysis, LLT also allows the user to see how long the processes were blocked for, and how much time the processes spent. However, LLT does not provide the sources of the timing requirement violation.

[21] proposed a tool for online testing using UPPAAL. The system is modeled using UPPAAL, and the test utilizes the model in order to generate test cases that can be fed into the implemented system. This work also focuses on testing the timing aspects. However, it does not consider the segmented delays of the implemented system such as input and output delay.

In [22], the boundary cases of timing constraints are considered in generating test cases to detect timing requirement violation. These works can be used and extended in order to generate the R-M testing cases in our proposed framework.

VII. CONCLUSIONS AND FUTURE WORK

We proposed a layered approach for the timing requirements testing in the implemented system developed by the model-based development. A four-variables model is used to express the abstraction boundary of the implemented system and measure the timing gap between the model and the implemented system. The R-testing is performed in order to check the timing requirement violation. If the R-testing result shows that the timing requirement is not satisfied, the M-testing is followed in order to measure the timing deviation of the implemented system with regard to the abstract model. We apply the R-M testing to the infusion pump system for the case study. The infusion pump system is implemented in the four different ways and the R-M testing is applied to the each of them. And also, two different types of the timing requirements (time deadline and time duration) are used for the scenarios.

Input events are read by the periodic threads in our case study. For the future work, we consider the different way to read the input events. That is, an interrupt service routine is considered as the input reading way. When the execution of the automatically generated code from Real-Time Workshop is interrupted, the behavior of the implemented system is different from the behavior of the abstract model. Furthermore, the behavior of the implemented system depends on where the execution of the generated code is interrupted at. The MathWorks also concerns this case and released a document [23]. Overall, through my future researches, I want to contribute to the development of the timing testing in the model-based development.

References

[1] France, R. and B. Rumpe, Model-driven development of complex software: A research roadmap. IEEE Computer Society, 2007: p. 37-54.

[2] J. F. Brett Murphy, Amory Wakefield, “Best practices for verification, validation, and test in model-based design,” 2008.

[3] Wolfgang, P. and P. Alexander, Abstractions for Model-Based Testing. Electronic Notes in Theoretical Computer Science, 2005. 116.

[4] G. Behrmann, A. David, and K. Larsen, “A tutorial on UPPAAL,” in Formal Methods for the Design of Real-Time Systems (revised lectures), ser. LNCS, vol. 3185, 2004, pp. 200–237.

[5] Mathworks Stateflow. http://www.mathworks.com/products/stateflow.

[6] Mathworks Simulink. http://www.mathworks.com/products/simulink.

[7] Parnas, D.L. and J. Madey, Functional documents for computer systems. Science of Computer programming, 1995. 25(1): p. 41-61.

[8] Amnell, T., et al., TIMES: a tool for schedulability analysis and code generation of real-time systems. Springer, 2004: p. 60-72.

[9] U.S. Food and Drug Administration, Center for Devices and Radiological Health. White Paper: Infusion Pump Improvement Initiative, April 2010.

[10] Mathworks Temporal logic.
http://www.mathworks.co.kr/kr/help/stateflow/ug/using-temporal-logic-in-state-actions-and-transitions.html#brh91yy-9_2

[11] http://www.freertos.org, “Using the freertos real-time kernel.”

[12] Mathworks Simulink Design Verifier.
http://www.mathworks.com/products/datasheets/pdf/simulink-design-verifier.pdf

[13] Operator’s Manual Auto Syringe AS50.
http://www.lhsc.on.ca/Health_Professionals/CCTC/eduquiz/as50.pdf

[14] Mathworks Graphical function.
http://www.mathworks.co.kr/products/stateflow/examples.html?file=/products/demos/shipping/stateflow/sf_gfdemo.html

[15] Keay, S., The safe use of infusion devices. Continuing Education in Anaesthesia, Critical Care & Pain, 2004. 4.

[16] U.S. Food and Drug Administration, definition of the over / under-infusion. http://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/ucm202502.htm

[17] Mark, U., P. Alexander, and L. Bruno, A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability, 2012. 22.

[18] Dalal, S.R., et al., Model-based testing in practice. ACM, 1999: p. 285-294.

[19] Pretschner, A., et al., One evaluation of model-based testing and its automation. ACM, 2005: p. 392-401.

[20] De Resyste, C., TorX: Automated Model Based Testing. Citeseer.

[21] K. Larsen, M. Mikucionis, and B. Nielsen, “Online testing of real-time systems using uppaal,” in Formal Approaches to Software Testing, 2005, pp. 79–94.

[22] D. Clarke and I. Lee, “Testing real-time constraints in a process algebraic setting,” in Proceedings of the 17th international conference on Software engineering, ser. ICSE ’95. ACM, 1995, pp. 51–60.

[23] Mathworks Dealing with Task Overruns. http://www.mathworks.com/tagteam/22845_temporary_overruns_scheduler.pdf

요 약 문

모델 기반 개발에서의 시간 요구사항 만족 여부 검증 테스트

의료 장비는 높은 안전성이 요구되는 시스템임에도 불구하고 소프트웨어의 결함으로 인해 사고가 일어난다. 안전성이 보장되는 소프트웨어의 개발을 위해서 모델 기반 개발법을 이용하는 추세이다. 모델링 툴을 이용해 시스템 모델을 요구사항에 맞게 모델링을 한 뒤, 모델 단계에서 요구사항 만족 여부에 대한 검증을 한다. 검증을 마친 후에는 툴의 기능을 이용해 모델로부터 자동적으로 소스 코드를 생성할 수 있다. 그러나 모델과 구현된 시스템 간의 timing semantics mismatch 가 존재하기 때문에 최종적으로 구현된 시스템에서도 시간 요구사항을 역시 만족시키는지에 대한 검증이 필요하다. 왜냐하면 비록 시간 요구사항이 모델에서는 만족되었을지라도 구현된 시스템에서는 만족되지 않을 수 있기 때문이다. 시스템의 시간 요구사항 검증을 위해 이 논문에서는 R-testing 과 M-testing 이라는 이중 레이어로 된 테스트를 제안하였다. 모델과 구현된 시스템 간의 시간 차이를 측정하기 위해서 four-variables 모델이 이용되었다. R-testing 은 구현된 시스템에서 시간 요구사항의 만족 여부에 대한 확인을 위해서 수행된다. 만약에 R-testing 의 결과가 시간 요구사항이 위반되었다고 나온다면 M-testing 을 추가적으로 수행한다. M-testing 을 통해 모델과 구현된 시스템 간의 시간 차이와 그 원인들을 세부적으로 측정하게 되고, 측정값들을 통해 시스템을 더욱 최적화시킬 수 있다. 사례 연구로 약물 주입 펌프 의료기기에 제안한 R-M testing 을 직접 적용함으로써 그 적용가능성과 중요성을 확인할 수 있었다.

황현이