A Layered Approach for Testing Timing in the Model-Based Implementation

BaekGyu Kim* Hyeon I Hwang** Taejoon Park** Sang H. Son** Insup Lee*

*University of Pennsylvania
**Daegu Gyeongbuk Institute of Science & Technology

***Abstract*—**
The model-based implementation is to derive an implementation from a model that has been shown to meet requirements. Even though this approach can be used to guarantee that an implementation satisfies functional requirements that are shown to be correct at the model level, it is still challenging to assure timing requirements at the implementation level. We propose a layered approach in testing timing requirements conformance of implemented systems developed by model-based implementation. In our approach, the abstraction boundary of the implemented system is formally defined using Parnas' four-variables model. Then, the proposed approach tests timing aspects of the interaction between the auto-generated code and the target platform-dependent code based on the four-variables. This approach aims at not only detecting the timing requirement violation, but also at measuring delay-segments that contribute to the timing deviation of the implemented system w.r.t. the model. We show the case study of testing timing requirements of an infusion pump system to illustrate the applicability of the proposed framework.

I. INTRODUCTION

The model-based implementation is to derive an implementation from a model that has been shown to meet requirements. Even though this approach has been shown to be effective in correctly implementing functional requirements, it is still challenging to assure timing requirements at the implementation level. Such a timing assurance gap mainly originates from the fact that models abstract timing aspects of implementations and target platforms. Abstracting timing dependent details is necessary to facilitate system design and verification while keeping the size of state space reasonable for model checking. It is also necessary since platform details are not available during the modeling phase. This introduces a challenge in how to validate the timing behavior of an implementation derived from a model. For example, input and output actions in many timed modeling languages employ instantaneous transition semantics in which transitions associated with input and output are assumed to take zero time. We propose a testing framework that can be used to measure the actual execution time of an implementation derived from models with such semantics and to reason about how measured time-bounds affect the adherence of the timing requirement at the implementation level.

There are several existing works for testing the final implemented systems in the model-based implementation. Software-in-the-Loop (SIL) and Hardware-in-the-Loop (HIL) testing is used to test whether the source code matches the desired behavior developed and specified in the Simulink/Stateflow model [1]. Even though tools are available to test functional conformance, they lack an ability to test timing aspects of the code running on a target platform. [2] proposed a tool for online black-box testing for real-time embedded systems using UPPAAL specifications. This work also focuses on testing timing aspects by generating test cases from the UPPAAL model that describes the expected timing behavior of the implemented system under a certain environmental behavior; however, it lacks the ability to measure internal time-delays occurring in the implemented system such as input and output delay during the execution of test scenarios. In comparison, our work focuses on precisely defining segmented factors (e.g., input/output delays, transition delays) that contribute to the timing-deviation of an implemented system, and how to test the time-delay caused by these factors.

We propose a layered testing approach that enables such a timing analysis to be performed in a systematic way. In this approach, the implemented system is a composition of the automatically generated code (e.g., C code) from the model and the target platform (e.g., sensors, actuators and accompanied device drivers). Our approach is to consider these two aspects separately in the testing framework using Parnas' four-variables model [3], which we use to characterize the abstraction boundary of the implemented systems. In the four-variables model, *monitored* (m) and *controlled* (c) variables characterize changes of physical environmental quantities; on the other hand, *input* (i) and *output* (o) variables characterize software behavior that interact with the physical environment through input/output devices. In our proposed testing framework, the four-variables (m, i, o, c) are used separately to reason about timing aspects of the generated code and the target platform. Two different levels of testing are proposed using the four variables. R-testing checks whether the implemented system conforms to the timing requirements. In this phase, test cases are generated using the input and output provided from the target platform only. If the testing result shows that a timing requirement is violated in the implemented system, then, M-testing is followed. In this phase, testing points are generated by segmenting several delays, such as input delay or output delay, in order to quantify timing deviation between the model and its implemented system. The proposed framework is applied to the infusion pump system to illustrate the applicability of our framework in testing the timing requirements.

The rest of the paper is organized as follows: we explain the timing assurance gap in Section II. Our testing framework is detailed in Section III. Section IV shows the case study using the infusion pump system, and concludes in Section V.

II. TIMING ASSURANCE GAP IN THE MODEL-BASED IMPLEMENTATION

In this section, we introduce the overall process of the model-based implementation and highlight the timing assurance gap between a model and an implemented system. We use a PCA (Patient-Controlled Analgesia) infusion pump system as an example to explain the problem.

A PCA infusion pump system is a safety-critical medical device that physically interacts with a patient by injecting medication in a controlled manner for pain-relief purposes. A bolus is a small amount of medication that is injected when a bolus-request button is pressed. An infusion administration

*This research was supported in part by the DGIST Research and Development Program of the Ministry of Science, ICT and Future Planning of Korea (CPS Global Center) and NSF CNS-1035715.
*978-3-9815370-2-4/DATE14/©2014 EDAA

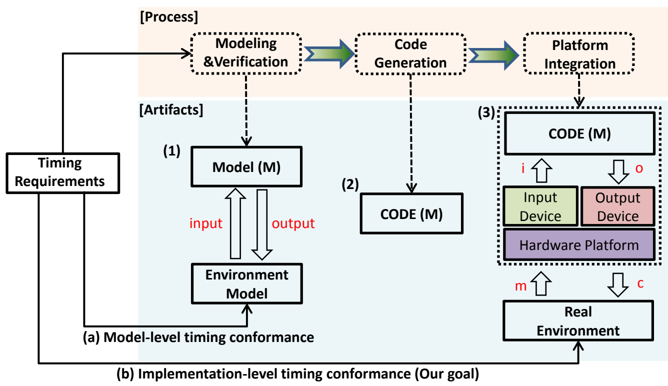

Fig. 1. The goal of the testing framework in the model-based implementation.

is controlled using several sensors and actuators. A pump-motor (actuator) is used to apply force to the syringe so that medication can flow from the syringe to the patient through intravenous tubes. Abnormal conditions, such as empty reservoirs and occlusions, are detected using various sensors and, when detected, caregivers are notified by buzzers and alarm LEDs (actuator).

Fig. 1 shows the model-based implementation process that we have used to develop infusion pump software. To illustrate the need for timing requirement validation, consider the following requirement from the GPCA safety requirements that list general requirements for the safe operation of PCA infusion pumps [4]; an explicit timing information is added to the original requirement in order to explain our work:

- *(REQ1) A bolus dose shall be started within 100ms when requested by the patient.*

The modeling and verification phase aims at creating a model (Fig. 1-(1)) that interacts with the environment model. For example, Fig. 2 is a Stateflow model that captures the software behavior of the infusion pump, and the timing requirement of REQ1 can be verified; the details of Fig. 2 are explained later in this paper.

Code generation aims at automatically generating source code that preserves the model behavior. Note that the generated code (denoted as CODE (M) in Fig. 1-(2)) is assured to conform to the model structure through this process. For example, the code generator used in [5] is able to generate C source code that implements transition tables, boolean (or integer) variables to represent input and output occurrences, and execution logic (*switch-case* or *if-then-else* statements), which maps to the model structure of Fig. 2.

Platform integration aims at adding interfacing code that is necessary for CODE (M) to be executed on the target platform. For example, input/output interfacing code bridges physical input/output (denoted as m and c variables in Fig. 1-(3)) and abstracted input/output of CODE (M) (denoted as i and o variables in Fig. 1-(3)). In this example, input interfacing code converts pressing the bolus request button, which generates an electrical signal change, into updating the generated boolean variable of CODE (M), that is mapped to the *i-BolusReq* input in Fig. 2. Our goal is to characterize such a potential source of timing gaps precisely so that the timing testing can be performed to identify timing violations in the final implemented system.

III. The Layered Approach in Timing Testing

This section explains Parnas' four-variables model that our testing framework relies on. We introduce a layered testing approach in which conformance to a timing requirement is first checked. Then, if the timing requirement is violated, then several delay-segments that contribute to the violation are measured. The measured delay-segments are used as useful information in debugging the timing requirement violation of the implemented systems.

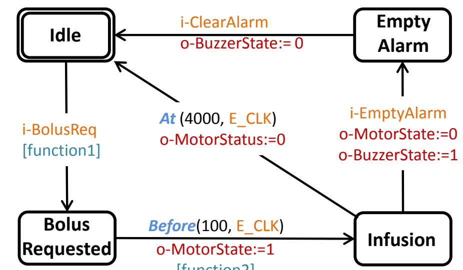

Fig. 2. The example Stateflow model for infusion pump software

A. Mapping the four-variables to the implemented system

To test the timing requirements in the implemented system (Fig. 1-(3)), we precisely identify the relevant input and output of the implemented system with associated timing constraints. One assumption made in the model in Fig. 1-(1) (and its example in Fig. 2) is that the model of the system and the environment processes their input and output instantaneously (i.e., zero processing time for input and output). However, this assumption creates uncertainty when reasoning about the exact timing of the input and output in the implemented system since several different interpretations are possible. For example, the input timing can be considered to be when a physical event happens at the boundary between the hardware platform and the real environment (e.g., electrical signal changes when pressing the bolus request button). Another possible interpretation is that the input timing is when CODE (M) reads the input event that is pre-processed by some input processing mechanism (e.g., sampling routines) executing on the hardware platform. We believe that uniform interpretation is necessary for precise timing testing of the input and output of an implemented system.

Parnas' four-variables model is a well-known technique in requirement engineering and has been used to precisely describe safety-critical system requirements [3]. The system requirements are described in the form of monitored (m), input (i), output (o), and controlled (c) variables with which interaction among *Input-Device*, *Output-Device*, and *Software* system can be captured. We use this concept in order to formally define different abstraction boundaries of the implemented system. Fig. 1-(3) illustrates the implemented system that shows the mapping with the four variables (m, i, o, c).

Monitored and Controlled variables: *monitored* variables (m) and *controlled* variables (c) are used to express physical environmental changes that can be observed and enforced by the hardware platform. A monitored variable (m) characterizes physical environmental changes and a hardware platform typically uses sensors to observe the status of m variable. For example, *m-BolusReq* is a monitored Boolean variable that captures the events, pressed or released, associated with the bolus request button (e.g., [*m-BolusReq==True*] implies the bolus request button is in a *pressed* state). A controlled variable (c) characterizes physical environmental changes, and a hardware platform uses actuators to enforce changes in physical dynamics. For example, *c-PumpMotor* variable may have a range of integer values in order to specify the speed associated with the pump motor (e.g., [*c-PumpMotor* == 10] implies the pump-motor rotates at a speed level of 10). From now on, we use m-event and c-event to refer to any changes in m-variable and c-variable, respectively.

Input and Output variables: *input* variables (i) and *output* variables (o) are used to express the input and output of CODE (M). An input variable (i) characterizes events that are read by CODE (M). For example, CODE (M) that is generated from the model in Fig. 2 has three i-variables; *i-BolusReq*, *i-EmptyAlarm*, *i-ClearAlarm*. *Input-Device* in Fig. 1-(3) is

responsible for converting the events in *m*-variable into the events in *i*-variable. Sensors and their accompanied device drivers are the example of *Input-Device*. An output variable (*o*) characterizes events that are written by CODE (M). For example, CODE (M) has two *o*-variables; *o-MotorState* and *o-BuzzerState*. *Output-Device* in Fig. 1-(3) is responsible for converting the events in *o*-variable into the events in *c*-variable. Actuators and their accompanied device drivers are examples of *Output-Device*. We use *i*-event and *o*-event to refer to any changes in *i*-variable and *o*-variable, respectively.

Note that the four-variable mapping enables the implemented system to separate the input and output in the boundary between CODE (M) and the target platform from those in the boundary between the target platform and the physical environment. We next explain the testing framework based on the four-variable mapping.

B. Testing Objectives and R-M testing

In the model-based implementation as shown in Fig. 1, the timing requirements that were verified in the model (Fig. 1-(1)) can become violated in the implemented system (Fig. 1-(3)). Such a violation can be due to many different possible sources of timing deviation in an implemented system. Our proposed testing framework is to deal with such timing deviation and aims at achieving the following two separate goals:

(G1) The implemented system is checked whether the timing requirements are violated or not.

(G2) The implemented system is measured as to how much it deviates from the timing behavior of the model.

The outcome from (G1) is a pass-fail testing result after performing a series of test cases extracted from a given timing requirement; we call this *R-testing*. The outcome from (G2) is a quantitative measurement (e.g., 10 ms or 100 ms) of delay-segments extracted from the model; we call this *M-testing*.

R-Testing: The conformance of the implemented system w.r.t. the timing requirements is checked through R-testing. In this testing, test cases are generated from the timing requirements in the form of *m*-variable and *c*-variable. For example, REQ1 can be expressed using a pair of *m* and *c* variables with its timing constraint:

- (REQ1-a) $\{$(*m-BolusReq*, t_{m1}), (*c-BolusStart*, t_{c1})$\}$
- (REQ1-b) t_{c1} - t_{m1} $\leq$ 100ms

where i) *m-BolusReq* is an *m*-event (value changes in *m*-variable) that can be observed from the hardware platform of the infusion pump; the timing of the *m*-event occurrence is denoted as t_{m1}, and ii) *c-BolusStart* is a *c*-event (value changes in *c*-variable) that is expected to be visible from the hardware platform upon receiving the prior *m*-event (*m-BolusReq*); the timing of the *c*-event occurrence is denoted as t_{c1}. The timing constraint required in REQ1 is specified by REQ1-b; that is, the time difference from t_{m1} to t_{c1} should be within 100 ms. Given the timing requirement, R-test cases are generated in order to check whether the implemented system conforms to the requirement using *m* and *c* variables only. For example, consider the following test sequence of input events generated from REQ1-a:

$\{$(*m-BolusReq*, *10ms*), (*m-BolusReq*, *300ms*), (*m-BolusReq*, *500ms*), $\}$

Then, the expected output timing of *c-BolusStart* event should be within 110 ms, 400 ms, 600 ms, ... according to REQ1-b. If all measured time differences from the implemented system conforms to this timing constraint, then R-testing passes; otherwise, R-testing fails.

M-Testing: If the R-testing result is *false*, the timing requirement verified at the model level does not hold in the implemented system that executes CODE (M) (i.e., the automatically generated code from the same model). For example, a bolus infusion is not started within 100 ms upon a patient's request in the infusion pump system even though it is shown to be satisfied by the model. The purpose of M-Testing is to measure delay-segments that constitute the timing deviation of the implemented system.

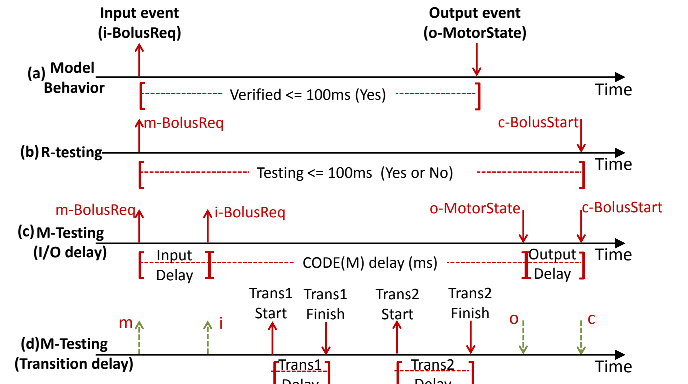

Fig. 3. The illustration of the timing testing in the R-M testing framework

Given the timing requirement of REQ1, Fig. 3 illustrates the timing behavior of the model (Fig. 3-(a)), and its implemented system using four-variables (Fig. 3-(b),(c),(d)). In Fig. 3-(a), when *i-BolusReq* event is provided to the model of Fig. 2, the *o-MotorState* event is produced within 100 ms. Fig. 3-(b) shows the timing behavior of the implemented system captured through R-testing. Suppose the R-testing result shows that REQ1 does not conform in the implemented system (i.e., the delay is greater than 100 ms). Fig.3-(c) and (d) illustrate several delay-segments that constitute the requirement violation, which are introduced below:

(1) *Input-Delay* is defined as a time passage from the occurrence of *m*-event to *i*-event. That is, it measures a delay from the physical input occurrence accepted by the hardware platform until CODE (M) actually reads the input (after being processed by *Input-Device*). For example, Input-Delay in Fig. 3-(c) illustrates the time delay associated with the (*m-BolusReq*, *i-BolusReq*) pair.

(2) *Output-Delay* is defined as a time passage from the occurrence of *o*-event to *c*-event. That is, it measures a delay from when CODE (M) writes the output until the moment the output becomes actually visible to the physical environment. For example, Output-Delay in Fig. 3-(c) illustrates the time delay associated with the (*o-MotorState*, *c-BolusStart*) pair.

(3) *CODE (M)-Delay* is defined as a time passage from the occurrence of *i*-event to *o*-event. That is, it measures a delay from when CODE (M) reads an *i*-event until the moment an *o*-event is produced. For example, CODE (M)-Delay in Fig. 3-(c) illustrates the time delay associated with the (*i-BolusReq*, *o-MotorState*) pair.

(4) *Transition-Delay* is defined as a time passage for executing transitions from the occurrence of *i*-event to *o*-event. Multiple transitions can occur during this period due to internal transitions, and each transition delay is separately measured in our testing framework. For example, two transitions constitute a pair of (*i-BolusReq*, *o-MotorState*) events in Fig. 2: transition from *Idle* to *BolusRequested* and transition from *BolusRequested* to *Infusion*. Fig. 3-(d) shows two transition delays $\{$Trans1-Delay (e.g., 11 ms), Trans2-Delay (e.g., 20 ms)$\}$. The time difference from the start to the end of each transition is measured and this set of delays is called the *transition delay* of the (*i-BolusReq*, *o-MotorState*) pair.

IV. Case Study: Timing Testing for Infusion Pumps

In this case study, we use the model-based implementation of an infusion pump system and apply the proposed testing framework to detect the timing requirement violation of the implementation, and how to measure the timing deviation from the model.

Case-Study Setting: The GPCA reference implementation project aims at improving software safety of PCA infusion pump systems by developing open-source artifacts through the model-based implementation [6]. In this project, the GPCA safety requirements are used, which contain general requirements that should be guaranteed for safe operation of PCA infusion pumps. The GPCA model written in Simulink/Stateflow is also used to capture generic software behavior of the infusion pumps from which the safety requirements can be formally verified. The code generation process automatically generates code that can work on infusion pump hardware platforms. The example requirements and the model used in our case study utilize a part of these open-artifacts in order to show the applicability of the proposed approach.

We consider REQ1 to be a timing requirement that needs to be satisfied in both the model and the implemented system. A model (Fig. 1-(1)) is created using Stateflow, and a part of this model is shown in Fig. 2. The timing requirement is verified in the model using the Simulink Design Verifier [7]. That is, the value of *o-MotorState* changes from zero to one within 100 ms when *i-BolusReq* is triggered while the system is in *Idle* state. RealTimeWorkshop [5] is used to automatically generate C source code (Fig. 1-(2)) from the verified model.

The generated code (CODE (M)) is then interfaced with the platform-dependent *Input-Device* and *Output-Device* on the infusion pump hardware used for the GPCA reference implementation. We use a Baxter PCA Syringe Pump as an infusion pump hardware and interface sensors and actuators to ARM7 micro-controller that runs the FreeRTOS real-time operating system.

Case-Study Scenarios: This case study shows how the proposed testing framework can be used to detect the requirement violation, and to measure timing deviation of different implemented systems.

We consider three representative implementation schemes to integrate CODE (M) with the target platform. The three implementation schemes are as follows:

Implementation Scheme 1 (Single-threaded implementation): The implementation, CODE (M), is executed by a single thread that is invoked periodically. In our case study, CODE (M) is invoked every 25 ms to read *m*-events from the sensors (e.g., bolus-request button); and to write *c*-events to the actuators at the end of CODE (M) computations (e.g., pump motor).

Implementation Scheme 2 (Multi-threaded implementation): This implementation uses multiple threads to read *m*-events from sensors and to write *c*-events to actuators. In addition, a thread that executes CODE (M) is separately run to read *i*-events from the sensing threads, and to write *o*-events to the actuation threads. Therefore, it is possible to sample sensor values, and to give commands to actuators at a different frequency from that of the CODE (M) execution. In our case study, the summation of the thread periods along the path of sensing-CODE (M)-actuation routines is less than 100 ms in order to make sure that any *c*-event is produced within 100 ms after an *m*-event is accepted by the sensing threads. The communication among sensing/actuation threads and CODE (M) threads is implemented using FIFO queues.

Implementation Scheme 3 (Multi-threaded implementation with other threads): Often, there are additional threads in addition to threads used by the model-based implementation (e.g., network drivers on infusion pump systems). This scheme aims to allow non-stand-alone implementation with additional functionalities executed by threads in addition to sensing, actuation, and CODE (M) threads of the implementation scheme 2. In our case study, three additional threads are scheduled. 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. These threads do not communicate with the CODE (M), but execute their own independent tasks.

No.	Impl. 1		Impl. 2				Impl. 3			
	R-Test	M-Test	R-Test	M-Test			R-Test	M-Test		
				Input Delay	CODE(M) Delay	Output Delay		Input Delay	CODE(M) Delay	Output Delay
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 I. TESTING RESULTS: MEASURED TIME-DELAYS FOR THE BOLUS REQUEST SCENARIO IN REQ1

Table I is the experimental results that show the time delays measured while each implemented system processed the bolus processing scenario in REQ1. Ten test samples obtained from each implemented system are shown in the table to explain how our testing framework works. The results of R-testing and M-testing are separately shown for each implemented system. Note that R-testing measures the time delay between *m*-event and *c*-event, and compares it to REQ1 in order to check the requirement violation; here, the numbers in R-testing columns imply the time delay between *m-BolusReq* event and *c-BolusStart* in milliseconds (ms). Red numbers in the R-testing columns imply that these test samples violate the timing requirement of REQ1 (i.e., the delays are greater than 100 ms). MAX implies that *c-BolusStart* was not observed until time-out after providing the *m-BolusReq* event. For those test cases that violate the timing requirement in R-testing, M-testing is followed to measure the specific delay-segments that constitute the requirement violation. The measured delay-segments can be used as useful information in debugging the timing requirement violation in the implemented system.

V. Conclusion

We propose a timing testing framework for the model-based implementation based on Parnas' four-variables model. The test framework measures the time delay between the auto-generated code and Input/Output devices. The four-variable model is used to partition timing testing into R-resting and M-testing. R-testing measures the time difference of the input and output events occurring at the boundary of the Input/Output devices and the environment. R-testing enables the implemented system to check a timing requirement violation. M-testing measures the delay-segments that constitute the timing deviation of the implemented system w.r.t. the model using the input and output events occurring at the boundary of the auto-generated code and Input/Output devices. This testing framework can be used to quantify timing deviation of implemented systems. In future work, we plan to study test coverage and test sufficiency from which test cases can be systematically generated in order to automate the proposed R-M testing.

References

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

[2] 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.

[3] D. L. Parnas and J. Madey, "Functional documents for computer systems," *Science of Computer Programming*, vol. 25, pp. 41–61, 1995.

[4] "Safety requirements for the generic patient controlled analgesia pump," http://rtg.cis.upenn.edu/gip.php3.

[5] MathWorks, "Simulink coder - generate c and c++ code from simulink and stateflow models."

[6] B. Kim, A. Ayoub, O. Sokolsky, I. Lee, P. Jones, Y. Zhang, and R. Jetley, "Safety-assured development of the GPCA infusion pump software," in *EMSOFT*, 2011.

[7] MathWorks, "Simulink design verifier - identify design errors, generate test cases, and verify designs against requirements," 2012.