1-1-2015

Safety-Assured Model-Based Development of Real-Time Embedded Software for the Gpca Infusion Pump

Baekgyu Kim
University of Pennsylvania, bgkim0110@gmail.com

Safety-Assured Model-Based Development of Real-Time Embedded Software for the Gpca Infusion Pump

Abstract
Many safety-critical embedded systems must meet safety requirements associated with timing constraints. Not only shall a system read/write correct input or output values, but also those operations shall be performed with the right timing. Failing to meet those timing constraints results in serious safety issues (e.g., medical device malfunctions may harm patients). It is difficult to develop complex embedded software in a correct way without rigorous and systematic handling of various sources that affect the timed behavior of a system.

We propose the model-based development framework that enables timing aspects of a system to be formally modeled, verified, and further implemented in a systematic way.

The fundamental idea is to separate the timing concerns of the platform-independent and the platform-dependent aspects of a system. In the platform-independent development phase, input and output timed interactions between a system and its environment is modeled and verified using state-transition formalism (e.g., UPPAAL) by hiding platform-dependent timing details. In the platform-dependent development phase, such platform-dependent timing details are modeled using architectural modeling languages (e.g., AADL) that are necessary to execute the platform-independent code on a particular platform, such as internal interactions among software components (e.g., threads) and hardware components (e.g., sensors and actuators). The platform-independent code and the platform-dependent code are independently developed from the different levels of timing abstractions, and composed together in the integration phase. In this phase, we propose a way to systematically extend the platform-independent model into different platform-specific models, which formally characterize the implementation-level timed behavior that can be verified for timing requirement conformance. In case this verification step fails, we propose a way to adjust the timing parameters of the platform-independent code by compensating for the platform-dependent processing delays in such a way that the resulting implementation meets the timing requirements verified in the platform-independent model.

Applicability of this development approach was demonstrated by developing software running on several Patient-Controlled Analgesia (PCA) infusion pump systems. We hope that this approach is also applicable to other safety-critical domains where generic software needs to be developed independently of a particular platform, and integrated with many different platforms in a way that conforms to timing requirements.

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Computer and Information Science

SAFETY-ASSURED MODEL-BASED DEVELOPMENT OF REAL-TIME EMBEDDED SOFTWARE FOR THE GPCA INFUSION PUMP

BaekGyu Kim

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2015

Supervisor of Dissertation

Insup Lee
Professor of
Computer and Information Science

Co-Supervisor of Dissertation

Oleg Sokolsky
Research Associate Professor of
Computer and Information Science

Graduate Group Chairperson

Lyle Ungar, Professor of Computer and Information Science

Dissertation Committee
Linh T.X. Phan, Research Assistant Professor of Computer and Information Science
Rahul Mangharam, Associate Professor of Electrical and Systems Engineering
Joseph Devietti, Assistant Professor of Computer and Information Science
Mats Heimdahl, Professor of Computer Science (University of Minnesota)
Paul Jones, Senior Systems/Software Engineer (U.S. Food and Drug Administration)

SAFETY-ASSURED MODEL-BASED DEVELOPMENT OF
REAL-TIME EMBEDDED SOFTWARE FOR THE GPCA INFUSION PUMP

ACKNOWLEDGEMENTS

This dissertation could be completed based on the advice, feedback and support from the University of Pennsylvania (UPenn) faculty and students, external collaborators and my family.

First and foremost, I would like to express my deep appreciation to my advisor Prof. Insup Lee, who gave me the chance to study here at UPenn. Without this opportunity, this dissertation would not even have been possible. Throughout my Ph.D research, not only did he provide me with valuable advice in a way to keep my research on the right track, but he also patiently waited for me in spite of my slow progress for a long time. I believe that the way he has helped, guided me and taught me largely influence me in tackling more difficult problems that I will encounter in the future.

I am also grateful to my co-advisor Prof. Oleg Sokolsky, who guided me to complete my dissertation. He was willing to give me specific feedback on what I proposed, and guided me to approach the solution step by step. He was able to pinpoint exactly what was right or wrong in my research topic approach, so that I could adjust my research direction accordingly. Without his help, I believe the completion of this dissertation would have been further delayed.

The final dissertation was able to be greatly improved through the labor of my dissertation committee members, Linh, Rahul, Joseph, Mats and Paul. They guided me to identify the missing parts and highlighted the major contribution of this work in a better way. I feel that the final version became much closer to what I originally intended, compared to its initial version. This was possible through reflecting on the valuable feedback of my committee members, which I truly appreciated.

I would like to thank everyone else at UPenn who gave me valuable feedback on my work. Especially, I appreciate all the Ph.D students who provided me with feedback during our

weekly meetings so that I could look at my research from different angles that I could not think of myself. In addition, through the contribution of each Ph.D. student who presented their work on a regular basis, I could accumulate and expand my knowledge of relevant issues in the medical devices domain. I also want to thank Linh who was always willing to provide detailed feedback on my ideas and guided me to write papers in a more precise and unambiguous way. Anaheed and Lu, to whom I want to express my appreciation, are the two postdoctoral researchers who collaborated with me to develop several research ideas through valuable discussions. Thanks to David Arney who helped me to accumulate background knowledge of the infusion pump safety issues in my early Ph.D studies. I also appreciate several masters' students who added their implementation efforts to build the research prototype platform to demonstrate the research ideas presented in this dissertation.

I also appreciate several external researchers who worked hard to collaborate with me for a long time. The interaction with the researchers at the University of Minnesota including Mats, Sanjay, Anitha, John and Dongjiang helped me to expand the way to approach the same problem from many different perspectives. I also had the chance to collaborate with the researchers at DGIST. I especially appreciate the Master's student, Hyeon I Hwang, at DGIST who spent hard time to complete this collaborative work. I also appreciate his co-advisors, Prof. Sang H. Son and Prof. Taejoon Park, for their patience that allowed me to continue working with Hyeon I.

I would like to express my special appreciation to the researchers at the U.S. Food and Drug Administration (FDA), especially to Paul Jones and Yi Zhang in the Center for Devices and Radiological Health (CDRH). Not only did I interact with Paul and Yi through email and conference calls, but also traveled to CDRH four times (Oct. 2010, Feb., July., Dec. 2011), and stayed several days for each visit for in-depth discussion about infusion pump safety issues through the support of the NSF/FDA Scholar in Residence at FDA program (NSF/FDA SIR). This series of interactions enabled me to understand the safety issues of medical devices from the perspectives of the industry and government regulators.

In addition, Paul and Yi were very supportive in explaining the research conducted in their software research lab, and provided necessary information that helped me to apply my academic research to the infusion pump domain. The GPCA safety model and safety requirements provided by their research lab were a good starting point for me to accumulate the domain knowledge of infusion pumps and its safety issues from a technical point of view. The technical knowledge that I obtained from them has played an important role not only in applying the model-based development approach to the infusion pump domain to enhance software safety throughout my Ph.D studies, but also in continuing my future career in broader safety-critical domains.

I also want to show my deep appreciation to my former advisor Prof. SoonJu Kang at Kyungpook National University, who advised me to complete my master's degree. The hands-on implementation experience for embedded systems that I learned during this period greatly helped me to build the research prototype system in my Ph.D studies as well. In addition, he encouraged me to continue my research studying abroad, which eventually enabled me to study at UPenn.

Finally, all my Ph.D research could not have been completed without the support from my family. I truly appreciate my family's endeavors which allowed me to fully focus on my research throughout my Ph.D studies.

This research was supported in part by the NSF CNS-1035715, NSF FDA Scholar In Residence 1042829, and the DGIST Research and Development Program of the Ministry of Science, ICT and Future Planning of Korea (CPS Global Center).

ABSTRACT

SAFETY-ASSURED MODEL-BASED DEVELOPMENT OF REAL-TIME EMBEDDED SOFTWARE FOR THE GPCA INFUSION PUMP

BaekGyu Kim

Insup Lee

Oleg Sokolsky

Many safety-critical embedded systems must meet safety requirements associated with timing constraints. Not only shall a system read/write correct input or output values, but also those operations shall be performed with the right timing. Failing to meet those timing constraints results in serious safety issues (*e.g.,* medical device malfunctions may harm patients). It is difficult to develop complex embedded software in a correct way without rigorous and systematic handling of various sources that affect the timed behavior of a system.

We propose the model-based development framework that enables timing aspects of a system to be formally modeled, verified, and further implemented in a systematic way. The fundamental idea is to separate the timing concerns of the platform-independent and the platform-dependent aspects of a system. In the platform-independent development phase, input and output timed interactions between a system and its environment is modeled and verified using state-transition formalism (*e.g.,* UPPAAL) by hiding platform-dependent timing details. In the platform-dependent development phase, such platform-dependent timing details are modeled using architectural modeling languages (*e.g.,* AADL) that are necessary to execute the platform-independent code on a particular platform, such as internal interactions among software components (*e.g.,* threads) and hardware components (*e.g.,* sensors and actuators). The platform-independent code and the platform-dependent code are independently developed from the different levels of timing abstractions, and com-

posed together in the integration phase. In this phase, we propose a way to systematically extend the platform-independent model into different platform-specific models, which formally characterize the implementation-level timed behavior that can be verified for timing requirement conformance. In case this verification step fails, we propose a way to adjust the timing parameters of the platform-independent code by compensating for the platform-dependent processing delays in such a way that the resulting implementation meets the timing requirements verified in the platform-independent model.

Applicability of this development approach was demonstrated by developing software running on several Patient-Controlled Analgesia (PCA) infusion pump systems. We hope that this approach is also applicable to other safety-critical domains where generic software needs to be developed independently of a particular platform, and integrated with many different platforms in a way that conforms to timing requirements.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1 : Introduction

1.1. Motivation

Meeting timing constraints is crucial in many safety-critical embedded systems, such as medical, avionic and automotive systems, and failing to meet those constraints may lead to catastrophic situations. Such systems interact with their intended environments (*e.g.,* patients for medical devices), hence, timing constraints are typically expressed in terms of the timing of the interaction between the systems and their environments. For example, when the environment generates an input, the system shall produce an output within a bounded time. Therefore, it is important to build a system in such a way that its internal processing - reading sensor input from the environment, computing output, and writing actuator output to the environment - meets the timing constraints.

Implementing software executing as a part of such systems is non-trivial. One of the major challenges comes from the fact that the timed behavior of the software is closely tied to a platform's specific timing aspects (*e.g.,* scheduling mechanisms provided by operating systems, and timing overhead of device drivers for sensors and actuators) where the software is running; in addition, their interactions occur in a complicated pattern. Therefore, an ad-hoc software development approach without rigorous analysis of the complex interaction between software and platforms is unsuitable when building high assurance safety-critical systems.

We believe that a good software development approach should enable one to abstract such timed behavior of systems precisely toward rigorous verification and analysis of timing constraint preservation in the early development stage. Furthermore, an implementation should be essentially derived from the abstraction as systematically as possible to preserve the verified timing constraints. The motivation of this dissertation is to find such a rigorous and systematic software development approach that is based on formal methods, and applicable in the safety-critical domains.

Note that many researchers have spent significant efforts in formal verification/analysis methods and tools to assure timing constraints of systems that can be applicable to realizing such a development approach. We believe that applying these prior outcomes in concert to building a concrete safety-critical application which goes through the full development cycle - requirements, system design and verification, implementation, testing and validation - would introduce another level of problem spaces and challenges related to timing aspects that are necessary to be solved. For this purpose, we introduce an infusion pump system that is used as an example safety-critical system to which these techniques are applied, giving focus on timing aspects.

1.2. Background: Infusion Pumps Safety and GIP Project

An infusion pump is a safety-critical embedded system that injects drugs into patients' bodies in a controlled manner for various medical treatment purposes. In order to increase the effectiveness of traditional infusion therapies manually performed by caregivers, modern infusion pumps are equipped with mechanical structures that enable a drug administration process to be precisely performed in an automatic fashion by entering infusion parameters such as dose rate and volume-to-be-infused. The mechanical process of infusion pumps involves pumping mechanisms in which the rotation of pump-motors generates physical forces to move a syringe (or a drug bag) so that drugs can flow from a syringe to a patient through an intravenous tube. Furthermore, the mechanical structure is designed in such a way that alarming conditions, such as empty-reservoir, occlusion and air-in-line, can be detected and informed to caregivers so that patients can avoid serious conditions such as under or over-infusion.

Software plays an important role in infusion pumps in order to make such mechanical processes happen by interacting with sensors and actuators attached to the mechanical structures. In modern infusion pumps, software complexity is growing to implement advanced features in order to provide more effective infusion therapies; for example, network capabilities are included in modern infusion pumps to enable downloading drug libraries

and auto-programming of infusion parameters to reduce efforts and errors of manual entry. According to the FDA, such infusion pump softwares may contain more than 100,000 lines of code [4].

It is necessary that infusion pump software meets high safety standards in order to minimize harms to patients while performing infusion therapies. However, according to FDA's infusion pump improvement initiative [53], the FDA received numerous reports of adverse events associated with the use of infusion pumps, including serious injuries or deaths. In particular, the 87 infusion pump recalls conducted from 2005 to 2009 due to safety problems have raised awareness of safety issues from stakeholders such as government agencies (*e.g.,* the FDA), device manufacturers, caregivers, and patients. Software defects are identified as one of the most common types of reported problems. For example, software fails to activate pre-programmed alarms when problems occur; software triggers alarms when no problem is present and, thus, interrupts normal infusion therapies; or software interprets a single key stroke as multiple keystrokes, a so called key bounce problem, causing over-infusion to the patients [5].

To address the safety concerns of infusion pump systems, the FDA initiated the Generic Infusion Pump (GIP) project [2] aiming at creating software safety models and reference specifications that can be used by manufacturers and research communities to improve infusion pump software safety. Among a range of different types of infusion pumps, Patient Controlled Analgesia (PCA) infusion pumps are used for controlling the post-surgery pain of patients by injecting pain medication such as morphine. The FDA released two resources that address safety issues of PCA infusion pump systems to the public: (1) Generic PCA pump safety requirements (GPCA safety requirements), and (2) a Generic PCA pump model (GPCA model). The GPCA safety requirements include requirements that are expected to hold in general PCA infusion pumps to ensure a minimum degree of safety. On the other hand, the GPCA model is an abstract representation of common software behaviors found in typical PCA infusion pumps. This model was built using Mathworks Simulink [47] and

Stateflow [48].

While these FDA resources focus more on extracting functional requirements and modeling the functional behavior of PCA pump software, we argue that timing aspects are worth being considered more explicitly throughout the software development process. This is important because, when it comes to safety, the functional aspects cannot be separately considered from the timing aspects, since critical functionality of PCA pumps are often timing dependent. For example, a bolus infusion is the basic functionality of PCA pumps to provide additional drugs upon patients' request by pressing a button. Meeting this requirement also implicitly implies that the functionality should be implemented by conforming to some type of timing constraints; for example, upon requesting a bolus, a bolus infusion should start and finish within a bounded time after delivering the expected volume of drugs. If this timing constraint associated with the functionality does not hold, then the infusion therapy will start late or finish prematurely, thus, giving too much or too little drugs, which may harm the patient due to over or under-infusion. One goal of this dissertation is to study how such timing aspects can be explicitly modeled and verified along with the functional aspects of PCA pumps throughout the software development cycles.

In addition, in spite of the availability of the safety requirements and the software model, there is not much reported on how PCA pump software can be implemented in a safe manner. In comparison to modeling and verification of timed behavior of the software, implementing a software from a verified model, particularly one that can operate on a PCA pump hardware platform, raises different dimensions of timing-related problems that might be interesting to both academic and industrial researchers and developers. By building a concrete system from abstract models, we envision to identify and formalize these problems in order to provide solutions.

Even though a PCA pump is one type of safety-critical application, we believe that the timing issues that we address in this dissertation have common ground with other safety-critical systems, such as avionics and automotive vehicles, that also need to guarantee

timing constraints with different granularity to assure safety. Therefore, we hope that our PCA pump case-study is used as an exemplar by demonstrating how the timing aspects of safety-critical domains can be modeled, verified, and implemented to achieve high safety assurance of systems.

1.3. The Research Goal and Challenges

This dissertation aims at systematically developing source code from formal models of PCA pumps that can operate on a range of PCA pump hardware platforms by focusing on timing aspects. Figure 1 illustrates the overview of this research project: during Phase a, the platform-independent timed behavior of PCA pump systems is abstracted using modeling languages that can be formally verified against timing requirements. During Phase b, source code (*e.g.,* C code) is automatically generated from the verified timed models through the code generation process; and the generated code is expected to operate on a range of PCA pump platforms; by platforms, we mean it is a collection of hardware (*e.g.,* sensors and actuators) and accompanied system software routines (*e.g.,* operating systems or device drivers) that are not compatible with other device manufacturers. During Phase c, the generated code is integrated with a PCA pump platform that is manufactured by a particular device manufacturer through the platform integration process. During Phase d, the implemented system (*i.e.,* the integration of the source code and a chosen platform) is tested against the timing requirements by generating test cases from the models. This research road-map provides us with benefits as well as research challenges in addressing timing concerns in developing PCA infusion pump software.

Note that the timed behavior of PCA pump systems is abstracted without clear distinction between software and platforms. That is, the timing aspects of a system are expressed independently of a specific target platform. Such a platform-independent timing abstraction of a system is useful and necessary from the software development perspective in the following sense:

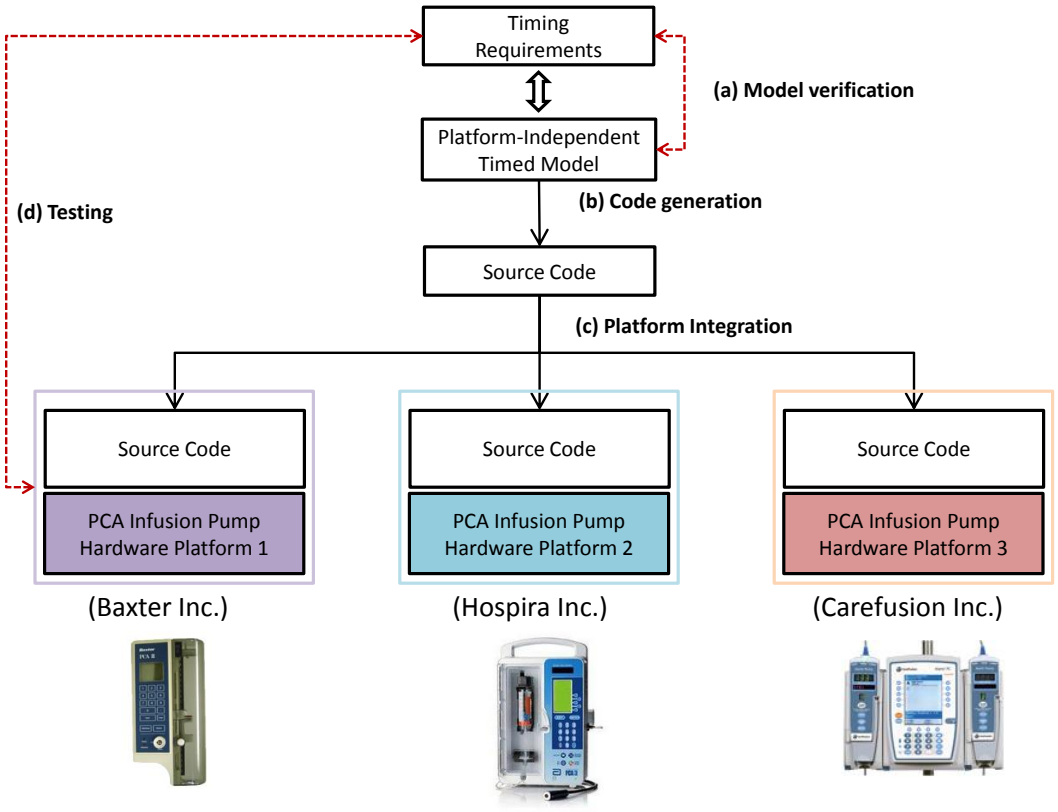


Figure 1: The overview of the GPCA reference implementation project

(1) *Lack of platform-dependent timing information*: When a software development is started, the design decision for the timing critical parts should be made by having sufficient platform-dependent timing information. For example, software design complexity to implement a certain control law may vary depending on how fast a chosen platform can execute the software. However, such detailed platform-dependent timing information may not be available in the early development stage for several reasons. For example, hardware platforms may not be ready at the moment when a software design needs to be started, or developing hardware platforms may be performed in parallel with the software development process. Therefore, it is necessary to have a system model that is independent of any particular platform in order to initiate a software development process in the absence of platform-dependent timing information; accordingly, the integration decision can be deferred to a

later stage when the platform-dependent timing information becomes available.

(2) *Reducing model complexity*: A formal verification is used to check whether a system model conforms to timing requirements or not. In some formal verification techniques, as the number of state variables in the model increases, the size of the system state space grows exponentially resulting in a significant increase of verification time [22]. In the initial development stage, modeling platform-dependent timing details, such as real-time scheduling algorithms and I/O processing delays of device drivers, may be less important, and incorporating such timing aspects adds unnecessary complexity to the model. Therefore, it is needed to abstract away such platform-dependent timing details to make a system model simplified in order to perform the formal verification at a reasonable cost.

(3) *Reusability of formally verified software*: Building a system model independently of a particular platform enables its generated code to be reused for a range of target platforms. Therefore, after gaining timing assurance about a platform-independent source code through formal verification and code generation processes, it is more cost-effective to use the same source code across different platforms instead of redoing the modeling, verification and code generation processes whenever platforms are changed. For example, PCA infusion pumps released by different device manufacturers have common software aspects to deliver drugs upon the patient's request, and to raise alarms upon detecting abnormal conditions. Such common aspects can be abstracted as a platform-independent software model, and source code generated from the model can be shared by different device manufacturers.

In spite of such benefits by adopting a platform-independent abstraction of a system, one naturally encounters challenges as to how to incorporate the platform-dependent part, and how to link it to the platform-independent abstraction toward the system-wide timing assurance. These challenges are as follows:

- What kind of timing information should be considered in designing the interface of the platform-independent code in a way that it can be integrated with different types

of platform-dependent code?

- What kind of platform-dependent timing information should be considered to operate the platform-independent code on a particular platform, and how should the timing information be abstracted and implemented on a platform?

- Upon integration, how should the platform-independent timing abstraction be mapped to the platform-dependent timing abstraction?

- How to reason about the timing conformance of an implemented system from the formal verification result of the platform-independent model?

We address the above challenges in our safety-assured model-based implementation framework to develop PCA infusion pump systems.

1.4. The Approach Overview

Our approach relies on reconciling two different levels of timing abstraction in developing final implemented systems: behavioral and architectural timing abstractions.

The behavioral-level timing abstraction intends to capture changes of internal system statuses through a series of input/output interactions with the system's environment; in particular, timing constraints (*e.g.,* min/max bounds) are associated with each input/output interaction so that the timed behavior of a system can be expressed in terms of expected timing bounds where each input and output may occur. Several transition-system based formalisms, such as timed automata or Stateflow, adopt such behavioral-level timing abstractions.

On the other hand, the architectural-level timing abstraction intends to capture the input/output information flow among different hardware components (*e.g.,* sensors or actuators) and software components (*e.g.,* threads or processes). Each component is associated with component-level timing properties that determine how a component is scheduled to read input or write output to/from another component (*e.g.,* time or event-trigger

mechanism). Several modeling languages such as AADL or UML MARTE, adopt such architectural-level timing abstractions.

The key idea of our approach is to use the two timing abstractions to separately model the platform-independent timing aspects (with behavioral-level timing abstraction) and the platform-dependent timing aspects (with architectural-level timing abstraction). Such separation enables the platform-independent code to be developed from the I/O behavioral perspective independently of particular platform-dependent architectural timing aspects. Next, we need to check whether the platform-independent code can be integrated with a particular platform in a way that conforms to the timing requirements. Our integration strategy provides several systematic mechanisms to check such compatibility between the platform-independent code and a particular platform.

Such separation of concerns is realized in three development phases illustrated in Figure 2: the platform-independent development phase (PI-Phase), the platform-dependent development phase (PD-Phase), and the integration phase (ITG-Phase).

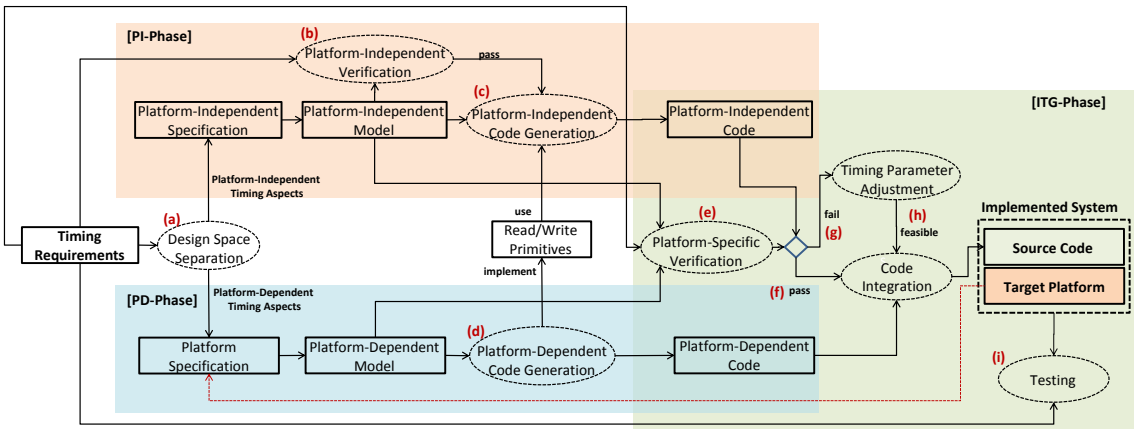


Figure 2: The approach overview of the safety-assured model-based implementation

Platform-Independent Development Phase (PI-Phase): The objective of the PI-Phase is to model and verify the input and output timing interactions between a system and its environment by abstracting details of the platform-dependent architectural timing aspects.

By abstracting such timing details, the developed platform-independent code is not tied to a particular platform architecture, instead it is expected to operate on a range of target platforms that may have different platform-specific timing aspects. The PI-Phase uses the following timing abstractions in constructing the platform-independent model: (1) the platform-independent abstraction of the input and output processing mechanisms: the platform-independent model expresses timed behavior of a system in terms of input and output events that are observable at the boundary of a system and its environment; however, it hides the internal details of the input and output processing mechanisms, such as how a platform reads and writes sensor inputs and actuator outputs (*e.g.,* polling or interrupt-based mechanisms), at this level of abstraction; (2) the platform-independent abstraction of execution schemes: source code generated from a platform-independent model needs to be executed (or scheduled) by a platform in order to realize the interaction between a system and its environment. Such an internal execution scheme is typically implemented by encapsulating the code behavior in a higher level software component such as threads or processes that can be executed according to a certain real-time scheduling mechanism. However, such a platform-dependent execution scheme is not represented at this level of abstraction.

Here is the development process of PI-Phase that follows this level of timing abstraction. Two inputs are given to the PI-Phase: (1) the platform-independent specification of PCA infusion pumps, and (2) the timing requirements. In this dissertation, for example, the FDA's GPCA model and safety requirements are used to extract a platform-independent specification and timing requirements respectively. Given a platform-independent specification, a UPPAAL model is created to abstract the input and output timed behavior of a system. According to the chosen level of timing abstraction, this platform-independent model hides the details of the internal architectural information flow of a system. In other words, from the perspective of the platform-independent model, it is sufficient to model that a system should produce an output within a bounded time once it receives an input from the environment; however, it hides the details about what kind of internal architectural

aspects are necessary to realize such bounded delays. In the verification process (b), the UPPAAL model is verified using model-checking to see whether a system conforms to the timing requirements during the interaction with its environment. In the code generation process (c), the platform-independent code (*e.g.,* C code) is automatically generated from the verified UPPAAL model using read/write/trigger primitives. These primitives are the implementation level interface that allows the platform-independent code to interact with an arbitrary platform that realizes a certain architectural option.

Platform-Dependent Development Phase (PD-Phase): The objective of the PD-Phase is to model and develop the platform-specific timing aspects of a system that are hidden at the PI-Phase to achieve the platform-independency, but necessary to support execution of the platform-independent code on a platform. This platform support includes mechanisms as to how the code is scheduled along with other types of platform-dependent codes (*e.g.,* device drivers) by operating system schedulers, and how the code reads data from sensors (*e.g.,* polling or interrupt-based mechanisms), and how the code writes data to actuators. The timing abstraction of the PD-Phase intends to express such platform-specific mechanisms to support the execution of the platform-independent code.

Here is the development process of the PD-Phase that follows this level of timing abstraction. Suppose a platform is given. In this dissertation, for example, a platform is considered to be PCA infusion pump hardware (manufactured by a certain company) and accompanies real-time operating systems and I/O device drivers. The information of a platform is used to construct the platform specifications, and further create a platform-dependent model. The platform-dependent model enumerates necessary hardware (*e.g.,* sensors and actuators) and software components (*e.g.,* threads and processes) that are required to create the final implemented system, and define dependency among these components to realize the internal information flow of a system. For this purpose, AADL is used to abstract such platform-dependent architectural information. In the code generation process (d), a platform-dependent code (*e.g.,* C code) is automatically generated to implement the inter-

nal architectural information that is expressed by the AADL model. Note that it is not necessary to have the same architectural information for any platform. Each platform may have its own architectural information to support the execution of the platform-independent code. Hence, the resulting platform-dependent code may also be different to accommodate such platform-dependent variation.

Integration Phase (ITG-Phase): The objective of the ITG-Phase is to build a final implemented system by composing a platform-independent and a platform-dependent code that have been developed from different levels of timing abstraction. An implemented system is considered correct if it conforms to the timing requirements. Note that not all compositions result in correct implementations since it has not been analyzed yet as to how the two levels of timing abstraction is correlated with each other in terms of the timing requirement conformance. Therefore, a major concern in the ITG-phase is to check whether a platform-independent code can be composed with a platform-dependent code in a way so that timing requirements are satisfied.

Our ITG-Phase consists of two steps in order to check the timing requirement conformance in the final implemented system: platform-specific timing verification and timing parameter adjustment. The platform-specific timing verification (e) is to formally verify (using model-checking) whether an implemented system conforms to the timing requirements in case the platform-independent code is implemented using the same timing parameters - such as min/max timing bounds of input and output occurrences - of the platform-independent model without compensating the platform processing delays. This verification step is realized by constructing a platform-specific model (PSM) whose timed behavior is close to that of such an implemented system. We first abstract platform processing delays (originally modeled as AADL models) using the UPPAAL semantics that can then be systematically composed with the platform-independent model (originally modeled as UPPAAL). This platform-specific model is used to perform the platform-specific timing verification; it may pass (f) if additional platform processing delays do not result in the timing requirement

violation; it may fail (g) if the platform processing delays result in the timing requirement violation.

The major reason of the verification failure is due to the fact that the platform-independent code does not account for the platform-processing delays during the PI-Phase. Hence, the timing parameter adjustment step is proposed to find new timing parameters that can be used in the platform-independent code to compensate the platform-processing delays, and if such a parameter adjustment is feasible (h), the platform-independent code generated with the new timing parameters can be integrated with the platform by conforming to the timing requirements.

Finally, in order to validate the timing requirement conformance in the implemented system, the testing process (i) generates test cases that are necessary to validate whether an implemented system conforms to the timing requirements; then, the test cases are fed into a final implemented system in order to validate the timing requirement conformance.

1.5. Contribution

We have made the following contributions:

- We formalized the system boundary of implemented systems based on Parnas' four variable model to establish the mapping from the input/output timing semantics of the model to that of implemented systems (Chapter 4);
- We proposed the primitive-based code generation that enables the platform-independent code to be generated in the absence of the timing information as to how a platform schedules the platform-independent code, and how a platform reads input or writes output to/from the environment, while at the same time allowing the code to be integrated with many different platforms that may implement those primitives in a different way (Chapter 4);
- We proposed the platform-dependent code generation algorithm that systematically

generates source code needed to support execution of the platform-independent code on a particular platform by characterizing its timing aspects using AADL models and code snippet repositories (Chapter 5);

- We proposed the platform-specific timing verification framework that enables us to construct the platform-specific models (PSM) by systematically extending the platform-independent model; PSM can then be used to formally verify the timing requirement conformance of an implemented system, and to quantify the deviation of the implementation from the platform-independent model with respect to the timing requirements (Chapter 6);

- We developed a method that automatically adjusts timing parameters used for determining the timing of reading input and writing output in the platform-independent code by compensating the platform-specific input/output processing delays. This strategy enables us to check whether it is feasible to execute the platform-independent code on a platform by conforming timing requirements verified in the platform-independent model; if feasible, it automatically finds the new timing parameters of the platform-independent code that need to be used upon integration with the platform (Chapter 7);

- We proposed a layered approach of testing timing requirements that enables us to not only check the timing requirement conformance of the implemented system, but also to measure the specific delay-segments that contribute to the timing constraint violation (Chapter 6).

1.6. Dissertation Outline

The rest of this dissertation is organized as follows: Chapter 2 explains the related work, Chapter 3 provides a general information about infusion pump systems in order to help understanding the basic operations and hardware platforms of actual infusion pump systems. In addition, we introduce the GPCA hazard analysis and safety requirements and the

GPCA model provided by the FDA. Chapter 4 introduces the platform-independent development phase; starting from categorizing the GPCA safety requirements, we introduce how platform-independent timed behavior is formally verified against the timing requirements, and further generated into platform-independent source code that is expected to operate on a range of platforms. Chapter 5 introduces the platform-dependent development phase. Given a particular target platform, we introduce how platform-dependent timing aspects are abstracted using modeling languages, and generated into platform-dependent source code that is expected to operate the platform-independent code. Chapter 6 introduces two different approaches to check the timing constraint conformance of the final implemented system: the first approach is the platform-specific timing verification that can be used when the timing information of a chosen platform can be formally abstracted along with the platform-independent model; the second approach is the platform-specific timing testing that can be used when the timing information is unknown, but has to be validated for the timing constraint conformance. Chapter 7 introduces the delay adjustment algorithm for the platform-independent code when it has to be integrated with a particular platform by conforming the timing constraints that have been verified in the platform-independent model. Finally, we give a summary and conclusion of this dissertation in Chapter 8.

CHAPTER 2 : Related Work

This dissertation proposes approaches that enable a system to be developed by separating timing concerns between the platform-independent and the platform-dependent aspects. The basic idea is to decompose the design space of a system that needs to be separately handled due to several constraints, such as model complexity and information availability; and then to compose the produced artifacts that have been separately developed by reasoning about the correctness of the composition with respect to timing requirements. In this chapter, we explain the related works from the following four different problem spaces, and argue about relevances and differences with our work:

- (Category 1) The software development processes
- (Category 2) The formal verification techniques for platform integration
- (Category 3) The code generation techniques
- (Category 4) The testing techniques

2.1. The software development processes

The work proposed by Thompson et al. [50, 49] indicates the importance of software prototyping in the early development stage and proposed an approach to the specification-based prototyping in which a detailed software specification is repeatedly refined from a formal executable model of the system requirements. In order to achieve such a prototyping flow, the Requirement State Machine Language (RSML) is designed to precisely describe the system specification based on the variables and state-transition formalism. Then, the NIMBUS environment enables the software specification to be structured by incrementally refining the system requirements. The case studies of developing software for the altitude switch system in the avionics domain [50] and the mobile robotics system [49] show the applicability of the proposed approach. The major concern in the work of Thompson et al. is (1) how to capture a system specification in a reusable way that evolves throughout the development cycles,

and (2) how to get the software specification from such a system specification. Similarly, we are also concerned about how to construct a formal model of a system in case the timing information of platforms (sensors and actuators in Thompson's work) is not precisely known in the early development stage; in addition, when that information becomes available later, we study how to develop software that will be integrated with a range of platforms that may have different timing characteristics. Even though the problem space is similar, we give more focus on timing aspects by choosing modeling languages such as UPPAAL and AADL that can express timing aspects of systems, which is difficult to express in RSML and NIMBUS.

Parnas proposed the four variable models [43] with which system requirements are formally described using four variables, namely, *monitored* (m), *controlled* (c), *input* (i) and *output* (o). This formalism enables a system to be expressed by decomposing software systems from input/output devices so that the system requirements can be formally represented through the mapping of those variables to the boundary of the subsystems. We utilize Parnas' formalism as a means to define the information flow occurring at the two system boundaries (*i.e.,* this dissertation calls them *mc*-boundary and *io*-boundary) that separate the platform-independent and the platform-dependent parts. Such a scoping of the two different aspects enables us to precisely define the timing relationship among various models constructed in our development process.

Unified Modeling Language (UML) has been extensively used to model a system from various abstraction levels and to help stakeholders to understand the system requirements and design. In particular, Gomaa's work [24] introduces the software engineering process for the development of software product lines using UML notations. The major concern in Gomaa's work is to separate common functionality from variable functionality to achieve software reuse when developing multiple systems that are similar to each other. Gomaa's work extends the UML-based modeling methods that are used for single systems to address software product lines, and claims that the proposed approach is more cost-effective

than developing each system individually from scratch. Such a product line approach is similar to our development process in a sense that it separates the common functionality (matched to our platform-independent aspects) from the variable functionality (matched to our platform-dependent aspects) to improve software reuse so that a series of similar systems (*e.g.,* infusion pump systems) can be constructed in a cost-effective manner. However, UML lacks formal semantics that makes it difficult to automatically verify whether a system actually meets the system requirements. Therefore, it typically plays a role of having common understanding about the system requirements and designs among different stakeholders without providing formal guarantees as to whether the constructed designs actually conform to the requirements. In addition, UML gives more focus on expressing functional requirements and design, not for timing requirements, which are one of the most important aspects that we want to capture and verify through using UPPAAL and AADL.

Sangiovanni-Vincentelli et al. introduced the contract-based design for cyber-physical systems [44]. A contract is a pair of assumptions and promises that are properties satisfied by the set of all inputs and all outputs of a design. In order to handle high complexity of systems, Sangiovanni-Vincentelli argues that it is necessary to apply such an assume/guarantee reasoning throughout all steps of the design process. Sangiovanni-Vincentelli also showed how the concept of the contract-based design is realized in the platform based design (PBD), where functionality (what the system is supposed to do) is strictly separated from architecture (how the functionality could be implemented) at each abstraction layer. We believe that our work is also aligned with the design philosophy proposed by Sangiovanni-Vincentelli. Our platform-independent model aims at capturing the functionality independently of a particular architecture adopted by a platform, and such architectural aspects are then composed with the platform-independent code in order to build the final implementation.

2.2. Formal verification techniques for the platform integration

One challenge that arises by separating timing concerns is to reason about the correctness of the integration of the platform-independent and the platform-dependent codes. In par-

ticular, when it comes to timing properties, this reasoning can be challenging due to the complex timing interactions that come from different levels of timing abstraction. There have been a number of approaches to bridge such timing gaps between a high-level model and its implementation. Here are a few works that are closely related to our work:

The work proposed by Altisen et al. [11] studied whether timed automata can be implemented on a given platform satisfying a desired property using its standard semantics. Our work also relies on timed automata semantics to abstract the platform-independent timed behavior, but with three major differences between Altisen's and our work: firstly, their implementation methodology that transforms a timed automaton into a program is similar to our implementation schemes in a sense that both study the possible ways to implement a timed automata on a given platform. However, Altisen's work studied only a restricted set of implementation methodologies, whereas we provide more general sets of implementation schemes. For example, (a) the buffering scheme for the communication between a platform and a program, and (b) the polling scheme for reading environmental signals, are commonly used implementation schemes that we consider; however, their work only considers the communication scheme using a shared-variable, and the interrupt-based scheme in reading inputs from the environment. Secondly, Altisen's work does not give details about how one can systematically transform an original timed automata model into a program model. Our work focuses on providing a systematic algorithm so that the transformation from the platform-independent model to the platform-specific model can be automated given an implementation scheme. Thirdly, Altisen's work focuses more on studying the property preservation under platform refinements; that is, when a slower platform is replaced with a faster one, they want to check whether the same property holds or not. On the other hand, our work focuses on providing a framework that enables one to compose a platform-independent model with a particular implementation scheme so that the composed platform-specific model can be used to verify timing properties.

The work proposed by Sifakis et al. [10, 51] also studied how platform-specific timing aspects

can be combined with the platform-independent model. These works introduced how an abstract model (that is matched to our *PIM*) can be transformed into a physical model (that is matched to our *PSM*). The basic idea of their transformation is to construct the physical model by assigning the execution time to each action transition of the abstract model (*i.e.,* transitions that are associated with input or output synchronization); more specifically, a single transition in the abstract model is decomposed into two separate transitions in the physical model. At the first transition, the clocks used in the abstract model freeze while performing the synchronization (*i.e.,* time does not flow while performing synchronizations), then, at the second transition, it compensates the execution time of the input (or the output) by adding timing guards associated with additional clock variables, which enables the transition to be completed after a constant execution time. However, this transformation simplifies several platform-specific aspects on which our work wants to focus. To the best of our knowledge, for example, their transformation does not consider a notion of event buffering or invocation mechanisms. Hence, the following scenarios cannot be captured and verified in their physical model: (1) *"even though an input event is read by the platform, the input may not lead to a transition in the model due to a buffer over-run condition*, (2) *"when the generated code invoked is too slow while the frequency of the input generated from the environment is relatively faster, the code may not process all the inputs"*.

Wulf et al. [54] proposed an approach to tackle the timed automata implementability problem based on a new semantics called *Almost ASAP* that is parameterized by a platform-processing delay. It presented a way to check whether a model that is constructed using this semantics can be implemented with a sufficiently fast platform; and if a faster platform replaces a slower one, the property still remains hold. However, Wulf's work didn't consider what constitutes the platform-specific delays specifically, wheres our work focuses on the source of timing delays that affects the platform-processing delays.

Bonakdarpour et al. [18] proposed a way to desynchronize distributed components modeled as the BIP semantics (*Behavior*, *Interaction*, *Priority*) in a way that the behavior of the

transformed BIP components is *observationally equivalent* with that of the original BIP model. During the desynchronization process, they inserted a notion of *manager* that mediates the communication among distributed BIP components in an asynchronous manner. Even though the distributed setting of their work is different from the stand-alone setting of ours, the problem of the interaction desynchronization is similar to our problem. However, they did not consider timing issues, whereas our main focus is to desynchronize the timed behavior of the platform-independent model from its environment by inserting a platform.

2.3. The code generation techniques

Lublinerman et al. [39, 38] introduces a code generation technique using the modeling language Synchronous Block Diagram (SBD). In SBD, the relationship between input and output is precisely defined at every synchronous round, and its semantic extension is currently being used in the Simulink/Stateflow model. Lublinerman's work considers the following implementation-specific criteria when generating code from a SBD: (1) internal structures of the generated code need to be sometimes hidden from the outside (*e.g.,* for intellectual-property concerns) and (2) a part of the generated code needs be reused in different contexts. To take into account these criteria, it introduces two quantifiable metrics, (1) modularity and (2) reusability, that can be used to measure how well the generated code from a block diagram can be used along with other code generated from different block diagrams in order to form a particular context. Then, several code generation algorithms are introduced, each of which maximize different metrics so that the generated code meets the respective implementation criteria while preserving the SBD semantics.

Lublinerman's work is similar to our code generation on how to generate a code that can be reused in different contexts (*c.f.,* different platforms in our work), with several differences. One major difference is on the level of abstraction to model implementations and how Lublinerman's work and ours model them. Lublinerman's work models an implementation based on the single modeling language (SBD) to abstract the relationship among different components characterized by a set of input and output dependencies. Our work

models more detailed platform-dependent interactions among thread and sensor/actuator components that comprises an implementation using AADL. In addition, our platform-independent modeling intends to capture behavioral aspects that should be implemented independently from such platform-dependent aspects. Therefore, our modeling strategy consequently makes our code generation generate source code that covers more details of the final implementation compared to Lublinerman's work.

Another difference comes from the metrics that each work wants to characterize in judging the composition. In order to consider the trade-off between reusability and modularity, Lublinerman's work manipulates the number of false dependencies in a way that each aspect can be maximized. However, Lublinerman's work does not consider timing aspects as to how the composition of different codes generated from different blocks impact timing requirement conformance; these timing aspects are the major concern in our work.

Henzinger et al. [28, 27] introduces a modeling language called Giotto, that can describe timed behavior of hard real-time applications. Logical Execution Time (LET) is the main abstraction of the Giotto timing semantics in order to implement deterministic timed behavior of the control application; that is, given a sequence of inputs, the output is produced at a pre-determined timing. When generating code from a Giotto model, Henzinger's work considers the following implementation-specific criteria: (1) the generated code should be ported on a range of platforms, and (2) the execution of the generated code should preserve the deterministic timing semantics of the Giotto model. However, achieving both code portability and deterministic timing semantics is challenging, since each platform may have different platform-dependent timing semantics (*e.g.,* different real-time scheduling policies), which will lead to different timed behavior of the generated code on its respective platforms. Henzinger et al. proposed a way to mediate the Giotto timing semantics and the platform-dependent timing semantics by separating the code generation into two processes. In the platform-independent process, E-code is generated in which the timing information is encoded independently of a particular platform's timing semantics; the generated E-code is

expected to operate on a virtual machine called the E-machine. In the platform-dependent process, the time-safety property is checked to see if platform-dependent timing semantics can operate the E-code preserving the Giotto semantics. If such a time-safety check is passed, the E-code can be run on a target platform preserving the Giotto timing semantics.

Henzinger's work is similar to ours in a sense that it mainly focuses on separating the two timing concerns in the code generation process - the platform-independent and the platform-dependent timing aspects. The verification flow to check correctness of the composition is also similar; both works generate the platform-independent code independently from a particular platform; upon integration, Henzinger's work performs *time-safety* checking that verifies whether a platform has enough performance to execute E-code according to the Giotto semantics. Similarly, our approach also constructs the platform-specific model (PSM) for the purpose of verifying whether the composition of the two types of code meets the timing requirement.

However, the scope of the code that our work generates is different from theirs. Henzinger's work generates E-code (*c.f.,* the platform-independent code in our work) that is compatible to platforms that run an E-machine, but our work does not require a platform to run a middleware to execute the platform-independent code. This is because, in addition to generating the platform-independent code, our work also aims at generating a platform-dependent code that can support the execution of the platform-independent code by constructing a more refined platform model to explicitly capture the platform-dependent timing aspects. As a result, our platform-independent code can be executed on a platform that runs the platform-dependent code that are characterized in the platform-dependent modeling stage (*i.e.,* no need for our platforms to run a particular middleware to execute the platform-independent code).

Such differences can also be highlighted when comparing our work to the AADL code generation using Ocarina [37] which requires platforms to run PolyORB middleware to execute the generated code. The assumption that any platform will run the same middleware makes the

code generation algorithm more simple, but we believe that designing a platform-dependent code generation algorithm that allows us to explicitly describe only the necessary parts of a platform will result in producing more efficient code, especially in the embedded systems domain.

Functional Mockup Interface (FMI) is a tool independent standard for the exchange of dynamic models and for co-simulation [17]. This standard intends to enable the composition of model components that are designed using distinct modeling tools. The standard is currently evolving and being extended through collaboration between simulation tool vendors and research institutes. One of the extension efforts has been made by Broman et al. [20, 21]. Broman proposed a way to extend the standard to enable deterministic execution for a much broader class of models [20], and defined a suite of requirements for hybrid co-simulation standards [21]. In our work, we utilized multiple modeling languages, such as UPPAAL, Stateflow, and AADL. Thus, if FMI standard supports the model exchange and co-simulation environment for the modeling languages that we have used in this work, we believe that some of the manual process (e.g., the model translation from Stateflow to UPPAAL) would be more efficiently performed.

There are also other works that studied separating concerns between specification and hardware-dependent details in generating source code. In particular, Schirner et al. [46] proposed automatic generation of hardware dependent software for MPSoCs platforms from abstract system specifications. The approach of Schirner's work is similar to ours in the sense that one can write specifications of embedded systems hiding the details of implementation, and later mapping to an actual platform to generate code separately. However, Schirner's work uses a modeling language, Transaction Level Models (TLM), as an input to the process as opposed to AADL which we use. In addition, they do not consider dealing with heterogeneous aspects of programming interfaces exposed by different platforms, which are characterized by our parametrized code snippet repositories.

2.4. The testing techniques

Our development process aims at gaining timing assurance of the software by using formal methods and systematic code generation approaches throughout the development cycles, ideally to reduce (or remove) the cost spent in testing. However, the models (*e.g.,* the platform-independent and the platform-dependent model) that we create are an *abstraction* of an implementation by hiding some details of the implementation. In practice, such an abstraction is necessary in order to keep the model size manageable for formal verification due to several limitations that come from a chosen technique (*e.g.,* scalability issues from the model checking). This makes testing necessary in concert with using formal methods.

There have been several attempts at using UPPAAL to test implementations. Larsen et al. [36] introduced a tool for online black-box testing for real-time embedded systems using UPPAAL specifications. In Larsen's work, the system behavior (*e.g.,* the PCA pump behavior) is modeled using an UPPAAL automaton that is composed with another UPPAAL automaton that models the behavior of the environment (*e.g.,* the patients) with which the system will interact. This information is fed into a UPPAAL-TRON for checking input-output conformance of the system. This tool can be used in our work to automatically generate test cases and to execute the tests in order to systematically check the correctness of the integration between the platform-independent and the platform-dependent code. Hessel et al. [30, 29] also used UPPAAL to derive test suites specifying the expected behavior of the implemented systems based on structural coverage criteria. We expect that these systematic online testing approaches can reduce test costs compared to manual testing.

Gay et al. [25, 23] studied the testing problem of using a model as an oracle to test implemented systems. Gay's work highlights the challenges of using a model as an oracle due to several aspects that are not properly abstracted in the model, but should be considered at the implementation level to perform more accurate testing such as input processing delays, execution time fluctuation, and hardware inaccuracy. Neglecting such gaps leads to excessive amounts of false positive testing results that need to be inspected further by adding

extra costs. The proposed solution is to adapt the model behavior to the implementation by comparing the state of the model to that of the implementation. In order to realize such an adaptation, Gay's work introduces a notion of the tolerance constraints that defines allowable changes to certain variables of the model. Next, the dissimilarity function is used to compare the state of the model to the observable state of the System-Under-Test (SUT). Finally, using the SMT-based bounded model checking technique, the framework finds a solution to the constraint that gives a new model state that is more similar to the behavior of the SUT; as a result, the new model can be used as a better testing oracle.

In comparison to our work, Gay's work addresses the similar issue in a sense that, in order to reason about the correctness of an implementation, there should be additional consideration about the impact of the timing gaps between the model and the implementation. One major difference is that our work focuses on constructing a new model (*i.e., PSM*) for model checking; on the other hand, Gay's work creates a new model for testing. More specifically, our approach relies on abstracting the platform-specific portion in a composable way so that it can be integrated with the platform-independent model. The resulting *PSM* is eventually used to formally verify the correctness of the integration between the platform-independent and the platform-dependent code. On the other hand, Gay's work gives more focus on transforming a model into another model through a steering process so that the resulting model behavior becomes closer to the implementation; the resulting model can then be used as a better testing oracle by reducing false positive testing results. We believe that these two works can complement each other to add more timing assurance to reason about the correctness of the composition of the platform-independent and the platform-dependent aspects of the implementations.

Mathwork introduces a concept of Model-in-the-Loop (MIL), Software-in-the-Loop (SIL) and Processor-in-the-Loop (PIL) testing [19] to validate the constructed artifacts from Simulink/Stateflow models (*e.g.,* models, source code, and microprocessors running the source code) throughout the development cycles in a systematic way. This multi-layer

testing approach enables a system designer to develop a model of a system in the early development stage from which a test suite is automatically generated to perform MIL testing; this test suite is used to check whether the model conforms to the requirements. Next, SIL testing is performed to check whether the behavior of the source code generated from the model matches that of the model by elaborating the test suite that was used in MIL. After that, PIL testing enables the microprocessor that is running the source code to be checked for the requirement conformance by further elaborating the test suite.

Such a layered approach is useful in our work to test the behavior of three different artifacts: (1) the platform-independent model (using MIL) (2) the platform-independent code (using SIL) (3) the final implemented system that is composed of the platform-independent and the platform-dependent code (using PIL). However, this approach does not explicitly explain how a test suite used in MIL should be refined to perform the next level of testing (*e.g.,* SIL and PIL). Particularly, such a test suite refinement is quite challenging when the notion of input and output used in the test suite of MIL are not well matched to those used in SIL and PIL for some reasons. For example, such scenarios may include a single input (*e.g.,* alarm condition) in MIL which is matched to two different inputs (*e.g.,* empty reservoir and occlusion condition) in SIL and PIL. Feeding an input (or observing an produced output) at SIL and PIL takes non-zero execution time as opposed to MIL which assumes zero delay for processing such input and output.

CHAPTER 3 : Background

3.1. Infusion Pump Preliminary

3.1.1. Infusion Pump Classification

An infusion pump can be classified based on its intended use in various medical treatments. The GIP class diagram shown in Figure 3 [2] classifies infusion pumps based on two criteria: (1) how an infusion pump physically contacts with patients, and (2) what types of medical treatment is targeted using an infusion pump.

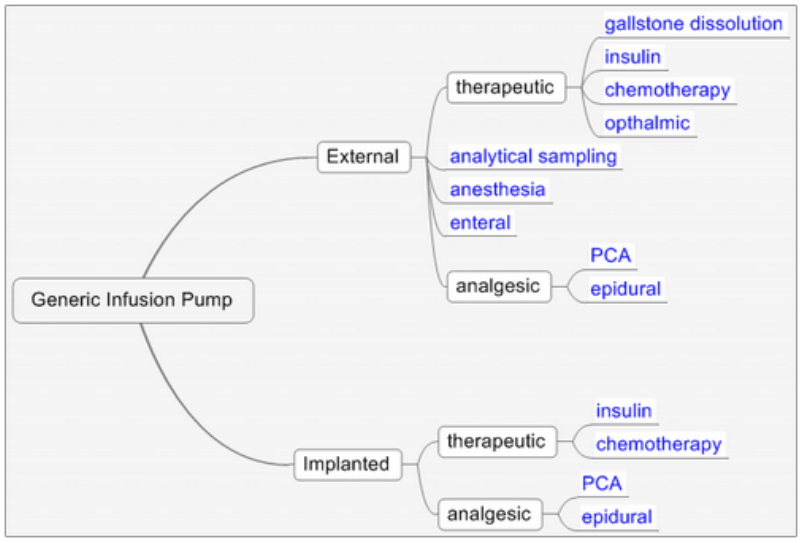


Figure 3: Generic Infusion Pump (GIP) Class Diagram

An infusion pump is classified as an external or an implanted pump depending on the way a pump physically contacts with patients. An external pump provides patients with drugs through an administration set that is connected to a patient intravenous system. This type of pump is typically located in close proximity to patients so that patients or caregivers may start infusion by setting a dose rate and Volume-To-Be-Infused (VTBI) through a user-interface whenever a short-term infusion therapy is needed. On the other hand, an implanted pump is surgically placed inside the patient body by threading a catheter into the desired position. A programmer, a separate device from a pump, is used to program

infusion parameters; after programming infusion parameters by caregivers, a pump is able to provide a constant-rate infusion at the location where the pump is implanted. This type of pump is typically battery-operated, and aims at providing long-term drug infusion in order to cure chronic diseases.

An infusion pump is also classified according to its targeting medical therapies. A pump may be used for therapeutic purposes. For example, an insulin pump is used to provide insulin for the treatment of diabetes patients in order to control a blood glucose level of patients' bodies; an infusion pump is also used for chemotherapy by delivering drugs to help eliminate cancer cells and keep those cells from multiplying in patients' bodies. On the other hand, a pump may also be used for analgesic purposes in order to help post-surgical patients be relieved from pain. For example, a Patient-Controlled Analgesia (PCA) pump is used to deliver morphine to patients upon their request by pressing a button (called a bolus infusion) or at a continuous rate.

Infusion pumps belonging to different classes may introduce different hazards so that different safety requirements may be required to mitigate those hazards. Among a range of infusion pumps explained above, this thesis considers external PCA infusion pumps to demonstrate the software development approach.

3.1.2. Hardware Platforms of PCA Infusion Pumps

A PCA pump hardware platform is equipped with mechanical structures that enable pain-relief treatment to be automatically performed based on infusion parameters set by patients or caregivers. We give a brief description of PCA pump hardware platforms here, and additional explanation will be provided later on when necessary.

A PCA pump hardware platform is basically designed to perform three major operations: (1) drug administration (2) detection of alarming conditions and (3) user interactions. We introduce hardware platforms of syringe-type PCA pumps. Figure 4 shows two syringe-type PCA pump hardwares produced by different device manufacturers, which are also used as

target platforms of our case study.

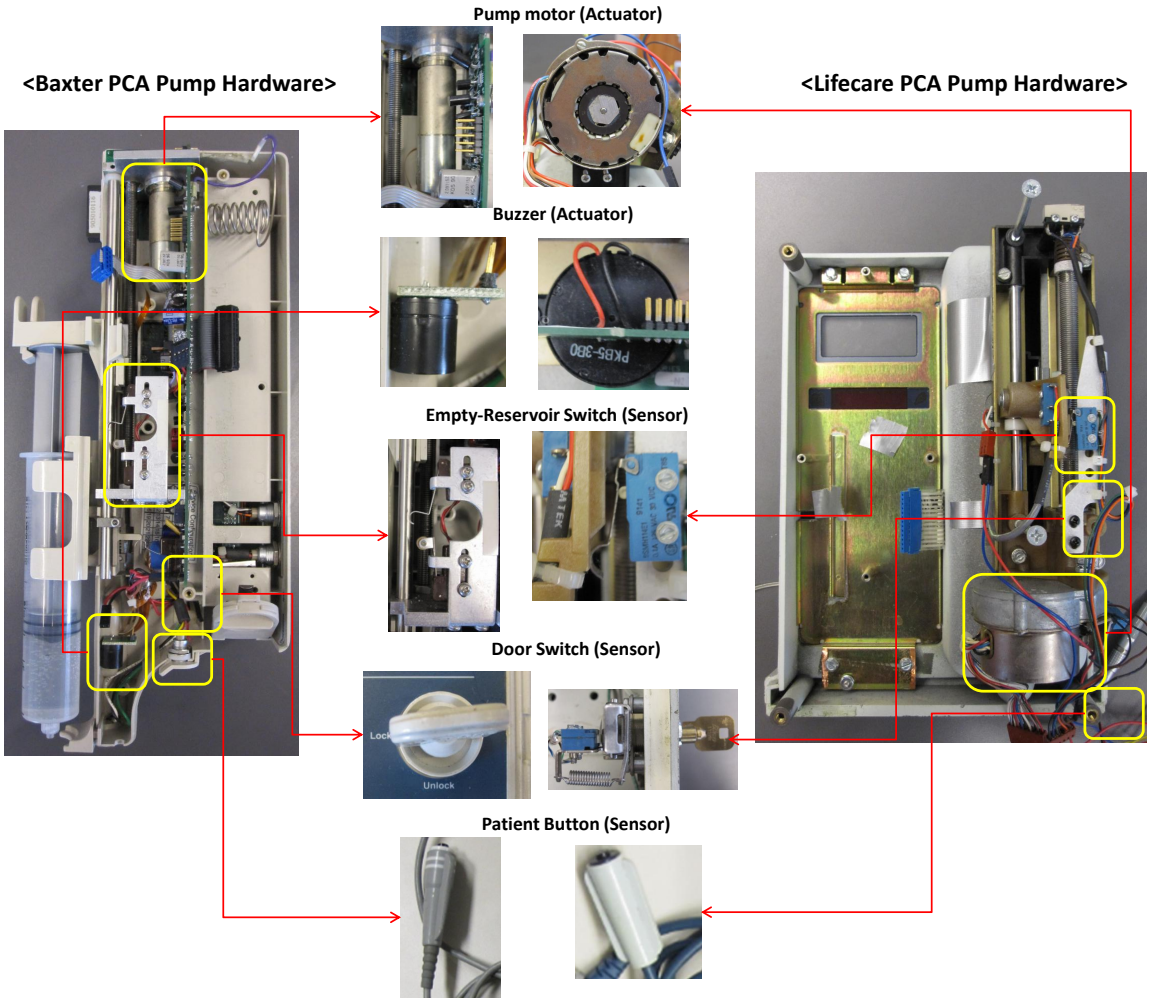


Figure 4: PCA Infusion Pump Hardware Platforms

(1) Drug administration: in order to realize infusion administration, a PCA pump hardware platform is designed to take a particular form of drug reservoir, such as syringes or drug-bags, where drugs are filled in a liquid form. A syringe-type pump is equipped with a syringe slot in order to firmly locate a syringe. A syringe bar mechanically connected to a lead screw is used to press the syringe located in the slot so that drugs can flow from a syringe to an administration set tube, and from the tube to the patient. Rotation of a lead screw generates a physical force to the syringe bar so that the bar can press a syringe vertically or horizontally. A pump motor (actuator) is used to realize the rotation of a lead

screw.

(2) Detection of alarming conditions: a pump is designed to detect alarming conditions in order to mitigate hazards that may harm to a patient during a drug administration. For example, an empty reservoir switch (sensor) is attached to a particular position where a syringe bar cannot move beyond that position. Therefore, the switch is pressed when no drugs are left in the syringe, so a syringe-empty condition can be detected by monitoring the empty reservoir switch status (pressed or released). Some pumps are equipped with additional switches in order to detect a syringe-low condition using a similar mechanism with the empty-reservoir detection. An accidental removal of a syringe from a slot during infusion is prevented by closing and locking a door with a key. A person is only able to access a syringe located in the syringe slot by opening this door for the purpose of removal or insertion of syringes. A door switch (sensor) is used to detect the status of a door (locked or unlocked). A pressure sensor is used to detect occlusion conditions that typically occur when a tube is twisted which results in an irregular flow of drugs. Such a pressure sensor is turned on or off by comparing occlusion pressures delivered to a pump motor to an occlusion pressure threshold that has been set a priori. Upon detecting these alarming conditions, a pump should trigger visible and audible alarm signals to patients and caregivers so that those alarming conditions can be removed by human intervention. An LED display or a buzzer (actuators) are used to trigger such a visible and audible alarm.

(3) User interactions: A pump is designed to interact with patients and caregivers in order to set infusion parameters and to initiate various types of infusions. A keypad is a means for caregivers to enter infusion parameters such as dose rate and VTBI in order to start drug administration. According to Oladimeji's work [42], typical keypads found in infusion pumps are based on either a serial number entry or incremental number entry mechanism. A keypad supporting serial number entry has numbered keys (from 0 to 9) and a decimal point that enables users to enter infusion parameters serially, digit by digit. A keypad supporting incremental number entry is only able to increment or decrement the number

typically using a pair of up/down keys or rotary dials. In addition to keypads, a PCA pump has a patient-pendant that is used for patients to initiate a bolus infusion by pressing the button. Whenever a patient feels pains, a patient may press this button in order to receive additional amounts of drugs during infusion.

3.2. Generic PCA Pump Safety Resources

We give the following definitions found in [52] in order to explain the safety resources by the FDA.

- *Hazard*: A possible source of danger or a condition which could result in human injury.
- *Hazard Analysis*: Identification of hazards and their initiating causes.
- *Hazard Mitigation*: Reduction in the severity of the hazard, the likelihood of the occurrence, or both.

3.2.1. Hazard Analysis for the Generic PCA Pump

The GPCA hazard analysis includes a list of hazards and their causes when patients use the PCA infusion pump systems. Table 1 shows example hazards and their causes from the GPCA hazard analysis [3].

Table 1: Example Hazards from GPCA Hazard Analysis

ID	Hazard	Cause	Mitigated by
1.1.	Over-infusion	Programmed flow rate too high	Drug library
1.2.	Over-infusion	Dose limit exceeded due to too many bolus requests	Flow sensor
1.8.	Under-infusion	Reservoir Empty	Flow sensor, Drug library
1.7.	Under-infusion	Occlusion	Flow sensor
1.10	Under-infusion	Flow rate does not match programmed rate	Flow sensor
1.16	Improper flow	Inaccurate flow rate; Infusion intermittent	Flow sensor

Over-infusion and under-infusion are major hazards that may result in a patient's injury by injecting drugs more or less than expected. Over-infusion is caused in several different scenarios. For example, patients are allowed to press a bolus-request button in order to ad-

minister additional amounts of doses due to severe pain during infusion. However, referring to the hazard ID-1.1. in Table 1, if a pump does not limit the maximum number of doses that patients can receive by pressing a bolus-request button, it will result in delivering a larger amount of drugs than patients are expected to receive. Under-infusion occurs as well as a result of several different scenarios. For example, when a syringe becomes empty during infusion, caregivers need to replace the empty syringe with a new one so that the infusion therapy can continue. However, referring to the hazard ID-1.8. in Table 1, if a caregiver fails to detect empty-syringe conditions, which leads to failure to replace the empty syringe with a new one, a patient may receive a smaller amount of drugs than the patient is expecting to receive during the infusion session.

The identified hazards in the GPCA hazard analysis are expected to be avoided or adequately mitigated when patients are using the PCA infusion pump systems.

3.2.2. Safety Requirements for the Generic PCA Pump

The GPCA safety requirements include a list of requirements that should be guaranteed in general PCA infusion pumps in order to avoid or mitigate the hazards identified in the GPCA hazard analysis. Table 2 shows example GPCA safety requirements found in [6].

For example, the safety requirement 1.4.11. in Table 2 intends to mitigate the over-infusion hazard 1.2 in Table 1 by setting a maximum permissible limit of bolus requests for a certain time period. Therefore, a pump can raise an alarm when the number of bolus requests by a patient exceeds this limit, which informs the patient or caregivers that the patient is taking an excessive amount of drugs. As another example, the safety requirement 1.5.6. in Table 2 intends to mitigate the under-infusion hazard 1.8 in Table 1 by monitoring the remaining volume of drugs; a pump should raise an alarm when the volume reaches a certain threshold, denoted as a parameter *y ml*. Therefore, caregivers who recognize the alarm can intervene in the infusion therapy by finishing the current infusion, removing an administration set from a patient, or by continuing the infusion, replacing the empty syringe with a new one.

Table 2: Example GPCA Safety Requirements

ID	Safety Requirements
1.4.1.	*A bolus dose shall be given when requested by the patient (normal bolus) or programmed to be administered over a period of time (square bolus).*
1.2.2.	*If the pump is equipped with a flow rate sensor and the flow rate exceeds the programmed rate setting by more than n% over a period of more than t minutes, or if the pump goes into free flow, the pump shall issue an alarm to indicate over-infusion of the patient.*
1.4.11.	*If a bolus request causes the bolus dose to exceed the maximum permissible limit (for a given time period), the pump shall issue a dose limit exceeded alarm.*
1.4.10.	*No normal bolus doses should be administered when the pump is alarming (in an error state).*
1.5.6.	*If the calculated volume of the reservoir is y ml, and an infusion is in progress, an Empty Reservoir alarm shall be issued.*
2.2.4.	*If the pump is idle for t minutes while programming a dose setting, the pump shall issue an alert to indicate that the user needs to finish programming and start infusion.*

As shown in these example safety requirements, all requirements are specified in natural language and may contain symbolic parameters, *e.g., y* in the requirement 1.5.6., to meet the needs of a wide range of PCA pump classes.

3.2.3. Generic PCA Pump Model

The GPCA model, provided by the FDA, abstracts software behavior that is commonly found in typical PCA pump software. The model is built using Mathworks Simulink and Stateflow. Figure 5 illustrates a high-level block diagram of the GPCA model. The *Alarm Detecting Component* and *State Controller* are two major state machines that consist of the GPCA model.

The Alarm Detecting Component serves as an interface to receive alarm signals, *e.g.,* empty reservoir, occlusion, and abnormal flow rate, from hardware sensors. These alarm signals are categorized into three levels based on their levels of severity - level 1 alarms (*e.g.,* CPU failure), level 2 alarms (*e.g.,* empty reservoir) and warnings (*e.g.,* infusion paused for too long). It also notifies the State Controller of any alarm signals so that the State Controller can react to the alarming conditions appropriately based on the level of severity.

On the other hand, the primary purpose of the State Controller is to regulate the rest

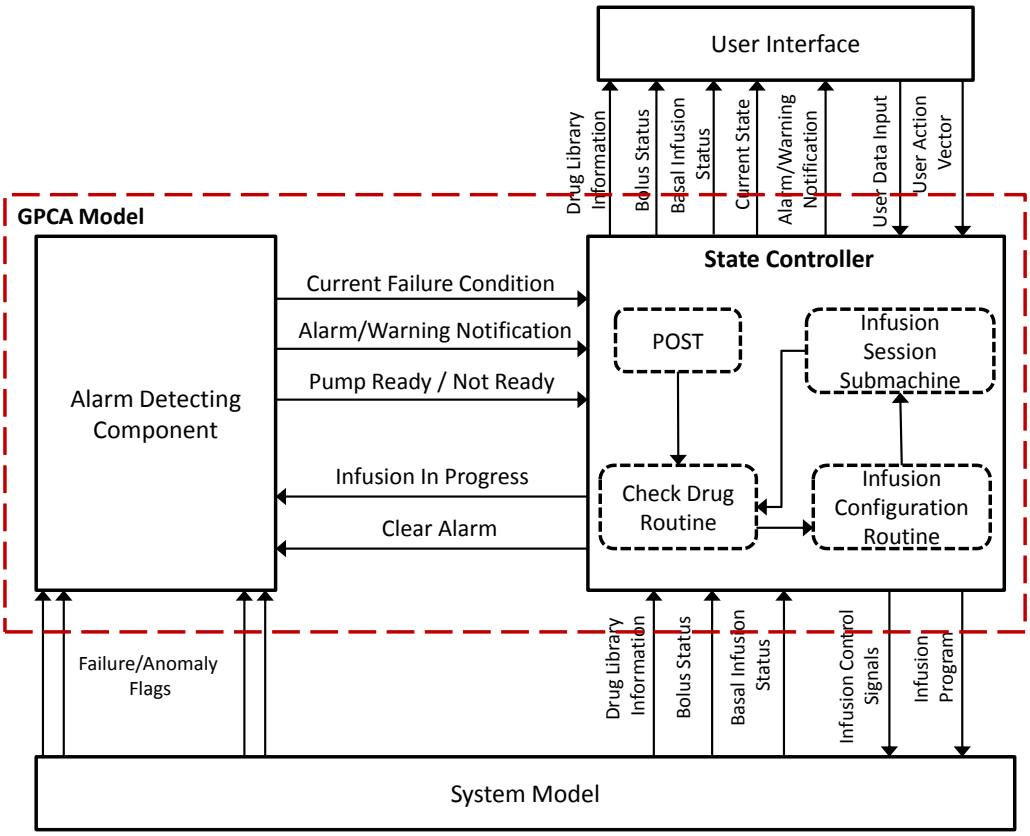


Figure 5: System Architecture of the FDA's GPCA Pump Model

of the pump to fulfill its expected functionality, *i.e.,* administering the right drug to the right patient at a right rate and dosage. The State Controller receives infusion requests from the user through a user interface and instructs the pump motor to deliver medication accordingly. It also provides additional functions to ensure the correct operation of the pump, including checking patient information, checking the correctness of infusion parameters, guiding the user on how to use the device, notifying the user of unsafe conditions via alarms, and so on.

This thesis uses the State Controller as a functional abstraction of the platform independent model, and shows how this model is extended to incorporate platform-independent and platform-dependent timing aspects toward building final implemented systems.

CHAPTER 4 : Platform-Independent Development Phase

4.1. The Problem Statements and Challenges

The FDA's GPCA model written in the Stateflow semantics is claimed to be the general system behavior found in a range of PCA infusion pumps. The main problem of the PI-Phase is to systematically develop a platform-independent code from the GPCA model that meets the GPCA safety requirements with formal assurance. Here are several challenges in developing such a platform-independent code.

Challenge 1: The GPCA model is expressed using the informal semantics provided by Stateflow, which makes it difficult to directly apply formal verification techniques. Even though the GPCA model abstracts the system behavior that is commonly found in PCA infusion pumps using the Stateflow semantics, its conformance to the safety requirements has not been formally verified. Here, the formal verification means that a model is checked against a formal query translated from a requirement by systematically exploring all possible cases occurring in the whole state space of the model; in other words, if the formal verification result proves the conformance, there is no such case that violates the requirement in any behavior introduced from the model; otherwise, the model may contain behavior that can lead to a requirement violation. [1] In spite of the fact that Stateflow has become popular in designing system behaviors in many industrial domains, it has several semantical issues that originate from its absence of formal semantics as indicated in several literatures [45] [26]. This becomes a big obstacle for formal verification of the system design modeled in Stateflow.

Challenge 2: Not all the safety requirements are written in a way that formal verification can be performed in the context of the GPCA model. Ideally, all the safety requirements are to be formally verified so that the subsequent development phases

[1] We note that this kind of formal verification technique is distinguished from testing or simulation-based techniques that are typically used to check a subset of the whole system space by providing a finite sequence of input and comparing the observed output with its expected output.

can inherit the verified property from the model. However, it is challenging to formally verify every requirement in the GPCA model for several reasons. We observed that the level of abstraction adopted in some safety requirements is different from that of the GPCA model. For example, some requirements include detailed functionality of PCA infusion pumps, whereas the GPCA model does not explicitly model such details; some requirements explain functionality belonging to the environment (*e.g.,* the user interface) with which the system expressed in the GPCA model should interact (*i.e.,* those functionalities are out of the scope of the GPCA model). We also observed that some requirements are vaguely written from the perspective of formal verification. These aspects that we observed impede the formal verification of all the safety requirements at the PI-Phase.

Challenge 3: Designing a well-defined interface of the platform-independent code is difficult since it is not directly matched to the timing semantics of the platform-independent model. The platform-independent code constitutes only a part of a final implemented system in our model-based implementation; in addition, the platform-independent code should be integrated with many different platforms in the later stage. Therefore, the platform-independent code should be generated along with a well-defined interface that allows it to be integrated with a range of platforms. Designing such a code-level interface needs correct translation of the timing semantics of the platform-independent model, but such a translation is complex. For example, an input transition semantics of a model requires multiple actions to be performed at the code-level: reading sensor input, making a transition decision by comparing the current clock value to the guard condition associated with the transition, and choosing the next transition from a transition table. In addition, this series of actions may be tightly coupled with the platform-dependent timing aspects (e.g., implementation of clock semantics is different across different platforms). Therefore, designing such an interface requires consideration in terms of what aspects should be decoupled as platform-independent and platform-dependent timing aspects, and how those decoupled aspects should interact with each other. Nonetheless, state-of-the-art code generators do not utilize this concept well enough: existing code generators produce a code

that is specific to a particular platform, which makes it difficult to be integrated with other platforms (*i.e.,* the generated code is tightly coupled with a particular platform-dependent code), or they do not provide a standard way about how the generated code should be integrated with a range of platform-dependent codes. These aspects of existing code generators requires the addition of a glue code in an ad-hoc fashion, which makes the final implemented system difficult to reason about how timing semantics at the boundary between the platform-independent and platform-dependent code are implemented.

We give the overview of our approach to tackle these challenges in the next section.

4.2. The Approach Overview of the PI-Phase

The approach for challenge 1: In order to tackle the informal semantics issue of the GPCA model written in Stateflow, we translate the GPCA model into another model that has formal semantics. UPPAAL is a modeling language whose semantics are the extension of timed automata semantics [12]. UPPAAL has a formal semantics, thus system behaviors modeled using UPPAAL result in no ambiguous interpretation. We transform the GPCA model into the UPPAAL model by defining a set of rules. The transformation rules capture the structure of the GPCA model in the UPPAAL model through mapping semantics of locations, transitions, input and output. The resulting UPPAAL model formally expresses the platform-independent system behavior that is similar to that of the GPCA model. [2] The UPPAAL model is used for the formal verification with the safety requirements, and for the platform-independent code generation.

The approach for challenge 2: Due to the abstraction gaps between the safety requirements and the GPCA model, it is necessary to figure out what requirements are formally verifiable at the level of abstraction of the UPPAAL model that has been translated from the GPCA model. For this purpose, the safety requirements are categorized into four cate-

[2] We note that the UPPAAL model is not a precise replication of the GPCA model in a strict sense due to the semantics differences originating from the two languages, but our mapping rules enable the functionalities intended from the GPCA model to be translated into the UPPAAL model.

gories: (1) safety requirements that can be formalized and verified in the UPPAAL model, (2) safety requirements that can be formalized, with the UPPAAL model requiring additional information to verify them, (3) safety requirements that cannot be formalized, but can be validated at the implementation level; (4) safety requirements that cannot be formalized because they address issues related to the environment of the GPCA model or vague in description. We translate the requirements belonging to the category 1 into UPPAAL queries, and these queries are fed into the model checker along with the UPPAAL model for the formal verification purpose. The verification result produces a binary output (*i.e.,* pass or fail). Informally speaking, the verification announces *pass* only if it cannot find a trace that violates the query by exploring all possible traces generated from the UPPAAL model; otherwise, it announces *fail* producing counter-examples that are traces generated from the UPPAAL model, which lead to the requirement violation. In case a system design defect exists in the UPPAAL model (that presumably comes from the original GPCA model), counter-examples can be used as useful information to fix the defect before starting implementation. The code generation process proceeds only if the formal verification announces pass for all safety requirements belonged to category 1.

The approach for challenge 3: In order to exploit the separation of concern at the code generation level, we first formally define the system boundary of the final implemented system using Parna's four-variable model [43] by precisely specifying the scope of the platform-independent code and the platform-dependent code, respectively. We compare how existing code generators (RealTimeWorkshop [40] and TIMES [14]) generate source code that is mapped to the scope of the platform-independent part; then, we extract common platform-dependent supports that need to be integrated for the execution of the platform-independent code on a target platform. Using these platform-dependent supports, we define primitives that are expected to be generated during the code generation process. The proposed code generation strategy generates the platform-independent code using primitive signatures only in the PI-Phase (*i.e.,* its internal implementation has not been completed yet); on the other hand, the internal implementation of such primitives are completed in the later platform-

dependent code generation process conducted in the PD-Phase. Such primitive-based code generation exploits clear separation of the scope of the platform-independent and platform-dependent code, and enables systematic integration of those codes in the later stage.

Figure 6 illustrates the overview of the PI-Phase, and a detailed explanation of our approach is given in the following sections.

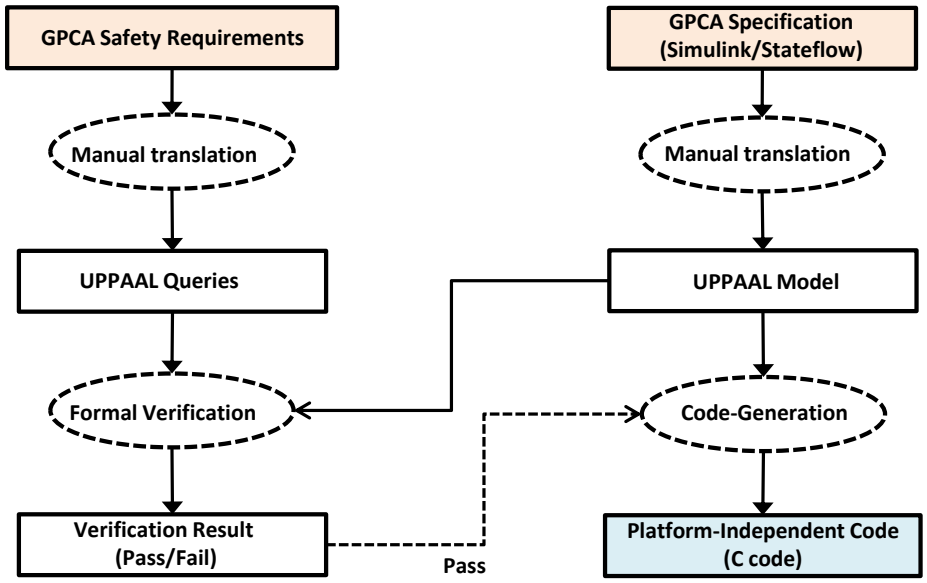


Figure 6: The platform-independent phase of the safety-assured model-based implementation

4.3. Platform-Independent Modeling Process

The platform-independent modeling process aims at developing a formal model of the PCA pump system using formal modeling languages. One criteria of such a formal model is to have a level of timing abstraction that is independent from any particular PCA pump platform. For the purpose of constructing such a formal model, we utilize the FDA's GPCA model as a specification, which is claimed to be a typical real-world PCA pump functionality that is common to the broad class of PCA pumps.

In particular, the State Controller that we utilized to demonstrate our model-based imple-

mentation framework consists of four parts as illustrated in Figure 5: (1) *Power-On-Self-Test* (*POST*), (2) *Check Drug Routine*, (3) *Infusion Configuration Routine*, (4) *Infusion Session Submachine*. The behavior of the State Controller is represented as Stateflow state-transition charts, which in total consists of more than 50 states and 100 transitions. The control flows of these charts depend on about 50 user events and hardware conditions. We explain how such behavior is formally modeled and verified, and further generated into source code.

Criteria for choosing a modeling language: We choose a modeling language that can be used for formally modeling the GPCA model with consideration from the perspective of both a design stage and its implementation stage. Here is our criteria in choosing a modeling language for the platform-independent modeling process:

- *Appropriate expressiveness to capture the specification*: The FDA's GPCA model, that is used as our specification, is described as a state-transition system where behavioral aspects of systems are represented as a finite set of states; transitions are accompanied with input that triggers a transition and output that is expected to be produced upon taking a transition. A modeling language used in our framework should have a similar expressiveness in describing such behavioral aspects that are used in the platform-independent specification in order to minimize gaps in translating a specification into a formal model.

- *Formally defined semantics*: The FDA's GPCA model relies on a Simulink/Stateflow modeling language; however, it has a lack of formal semantics, which may lead to inconsistent model behavior from what the model was originally intended for. For example, as indicated in the literature [45], Stateflow has several semantical issues in processing a run-to-completion semantics of event broadcast leading to stack overflow; junction backtracking without restoring the previous values of the variables, and so on. We believe that choosing a modeling language that has precise formal semantics is crucial in reasoning the correctness, not only of a specification, but also its

implementation.

- *Formal verification capability*: The conformance of a developed model with respect to the timing requirements should be verified before starting its implementation. Such verification can be categorized into a simulation-based verification or a formal verification [35]. In simulation-based verification, one generates input vectors from a timing requirement, and feeds them into the model, and compares its output timing with its expected output timing in order to judge the conformance of the requirement. On the other hand, the formal verification expresses the desired output behavior in a form of queries, and then lets a formal checker prove or disprove the queries. In writing such a query, it is not necessary to consider how many input vectors should be fed since a formal checker will automatically explore all possible cases in order to verify the query. In our framework, a modeling language that is accompanied with such formal verification capability is required in order to make sure the timing requirements are verified in a model by exploring all possible cases.

- *Code generation capability*: We envision that a developed model is automatically transformed into source code in order to preserve the properties that have been verified in the verification process. A code generator takes a model as an input, and produces source code (*e.g.,* C code) as an output. In our framework, a modeling language that is supported by such code generation capability is required in order to obtain source code automatically instead of obtaining it through error-prone manual coding process.

UPPAAL [16] is an integrated tool environment for modeling, validation and verification of real-time systems. Its modeling language is based on timed automata [12] extended with data types such as bounded integers and arrays. UPPAAL supports a model checking capability with which one can write properties using UPPAAL query language, and verify whether a model satisfies those properties. If a model satisfies a property, the model checker announces "pass", otherwise "fail" by showing counter examples that show why the property is not satisfied in the model.

We use UPPAAL to model the GPCA specification, and to verify the timing requirements since it meets the above criteria. However, we note that UPPAAL is not the only modeling language that can be used in our model-based implementation framework; other modeling languages that meet the above criteria are also equally good.

Transformation of the FDA's GPCA Specification into the UPPAAL model: We transform the State Controller part of the GPCA model expressed in Simulink and Stateflow into a network of UPPAAL automata through a manual translation process. The State Controller is organized as four sequentially connected state machines. Each of these four state machines separately abstracts a series of system behavior that interacts with its environment from the moment when a PCA pump is turned on, until the moment when an infusion administration finishes. Each of these four state machines has a final state that is associated with an event that results in a transition to an initial state of another state machine.

To retain as much of the syntactic structure of the Stateflow model as possible, the transformation maintains one-to-one mapping between states, conditions, actions, and transitions in the two models. It is noted that our transformation process is not intended to have a precise replication of the Simulink/Stateflow model by overcoming all the semantics differences between the two models. Instead, we reconstructed the general functions of the Simulink/Stateflow model in the UPPAAL model, which can be formally verified against the GPCA safety requirements. While it is possible to combine the four state machines into a single UPPAAL automaton, we chose to model them separately, preserving the model structure. As a result of the transformation, most states and transitions in the UPPAAL model corresponds to those of the State Controller of the GPCA model.

During the transformation, we also had to introduce quantitative timing information into the UPPAAL model. The State Controller of the GPCA model contains timeout transitions, but constraints triggering timeout transitions are not specified. We introduced a clock shared by all UPPAAL automata to capture the progress in time. Then, we added invariants to

the automata locations and extended transition guards to enforce timeout constraints. The timeout constraints were derived from the GPCA safety requirements and instantiated with specific values when used in UPPAAL models.

As illustrated in Figure 7, the transformed UPPAAL model consists of four automata that correspond to the four parts of the State Controller of the GPCA model. The details of each automata is explained in Appendix A.1, and we here briefly explain each with its relevant GPCA safety requirements that should be guaranteed for the safety operation of a PCA pump.

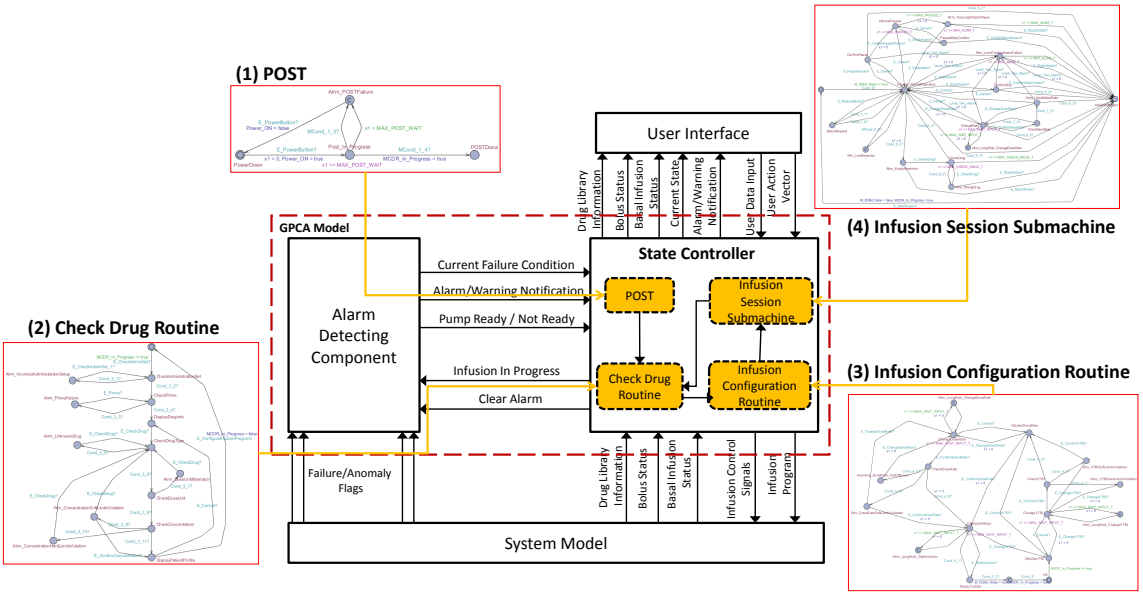


Figure 7: The mapping between the GPCA model and the UPPAAL model

The POST Session. The *Power-On-Self-Test* (*POST*), triggered by turning the power on, includes self-tests of processors and memory, critical circuitry, indicators, displays and alarms to ensure that a device is ready for use. The GPCA model abstracts relevant testing procedures into a state, called *POST*, which is mapped to the *POST-In-Progress* state in the UPPAAL model as shown in Figure 43 of Appendix A.1. An alarm state is entered if the POST check fails or stalls for a certain period of time. The GPCA safety requirements related to the POST session are:

- *No bolus dose shall be possible during the POST.*
- *The POST shall take no longer than t seconds.*

We note that the second requirement cannot be checked at the model level, since the details of actual POST operations and times they take are abstracted away. Instead, we interpreted the requirement to mean that if POST does not complete within t seconds, the pump enters into an alarm state. This interpretation is consistent with the GPCA model, which includes an alarm state in the *POST session* that is entered by a timeout transition.

The Check Drug Routine. The *Check Drug Routine* checks drug type and concentration to make sure that the right drug is loaded. The result of each check is decided by the user, and can take one of the two possible outcomes: a successful outcome will move the automaton to a state where the next check can be performed, while an unsuccessful one raises an alarm to be displayed by the user interface. The corresponding UPPAAL automata is shown in Figure 44 of Appendix A.1.

The Infusion Configuration Routine. The *Infusion Configuration Routine* describes a workflow that a caregiver goes through to setup an infusion administration program (*i.e.,* prescription). Dose rate and volume-to-be-infused (VTBI) are typical infusion parameters that should be entered before starting any infusion session. To reduce potential dose errors, this submachine also checks the entered infusion parameters against a pre-loaded drug library. If the entered parameter values exceed the soft or hard limits specified in a drug library, the submachine would prompt the user to either reconfigure them or abort the infusion by raising alarms. Figure 45 of Appendix shows the UPPAAL model of the *Infusion Configuration Routine*. Relevant GPCA safety requirements are:

- *The pump shall include a programmable drug library configurable according to patient type (adult, pediatric, etc.) and care area (home care, ambulatory, clinic, etc.).*
- *If the programmed infusion parameters exceed the upper or lower hard/soft limits, the*

pump shall issue an alarm and prompt the user to revise the parameters.

- *If the pump is idle for t minutes while programming an infusion prescription, the pump shall issue an alert to indicate that the user needs to finish programming and start infusion.*

We note that the GPCA model does not specify a timeout on the states that require user inputs such as *ChangeDoseRate* or *ChangeVTBI*. This is one of the places that we added timeout values and extended transition guards to capture the timing requirements.

The Infusion Session Submachine. The *Infusion Session Submachine* abstracts how a system coordinates the rest of the pump to complete the infusion process. The user may change the pump administration process, such as canceling or suspending the infusion, requesting boluses, adjusting infusion parameters and so on. In addition, upon receiving alarm signals from the Alarm Detecting Component during infusion, a pump should handle those alarm signals appropriately by raising an alarm or pausing the current infusion.

Figure 46 of Appendix A.1 is the UPPAAL model of the *Infusion Session Submachine*. The *Infusion Session Submachine* is entered after acceptable infusion parameters have been set. The pump performs the infusion in the *Infusion-NormalOperation* state in which it reacts to multiple user requests or failure conditions. In particular (1) a patient can request a bolus during the ongoing infusion as reflected by the *E-RequestBolus* event; (2) the caregiver can pause a current infusion by pressing a pause button that triggers the event *E-PauseInfusion*; (3) an empty-reservoir condition (condition *Cond-6-3*) occurs if the remaining volume of the drug reservoir is less than a pre-specified threshold; and (4) the condition *Level-Two-Alarm* should be processed if a hardware failure such as the drug reservoir door is open or an occlusion is detected. Transitions caused by failure conditions must take precedence over those caused by user events. To model this rule, we gave higher priority to the transitions triggered by failure conditions such as *Cond-6-3* or *Level-Two-Alarm* over the transitions triggered by user events such as *E-RequestBolus* or *E-PauseInfusion*.

Relevant GPCA safety requirements are:

- *The pump shall issue an alert if paused for more than t minutes.*
- *If the calculated volume of the reservoir is y ml, and an infusion is in progress, an Empty-Reservoir alarm shall be issued.*

Note that the second requirement affects both the *Infusion Session Submachine* and the *Alarm Detecting Component*, which performs volume calculations and sets the low-volume condition (*Cond-6-3*). Since the latter component is not modeled in our case study, the requirement is restated as: if condition *Cond-6-3* is set and an infusion is in progress, an *Empty Reservoir* alarm shall be issued.

The Environment Model. The above UPPAAL model, translated from the State Controller part of the GPCA model, abstracts the platform-independent system behavior. In order to perform formal verification, an environmental behavior needs to be specified in the form of a model, which interacts with the platform-independent model. Therefore, one can check whether the system behavior specified in the platform-independent model meets the safety requirements when its environment behaves according to the environment model.

For example, the transition from the location *Infusion-NormalOperation* to the location *BolusRequest* in Figure 46 is taken when an input called *E-BolusRequest* is received from the environment. Next, an environmental behavior needs to be defined to determine how such an input is generated in order to verify, for example, that the *E-BolusRequest* input leads to a transition from the location *Infusion-NormalOperation* to the location *BolusRequest*.

Unfortunately, it is challenging to precisely define such environmental behavior in the beginning of the development process. Therefore, we take a conservative approach in creating such an environment model; that is, it is assumed that the environment is able to provide the platform-independent model with inputs at any point of time. If the platform-independent model has different behavior on a particular state, depending on inputs received from the

environment, the environment generates those inputs in a non-deterministic manner.

According to the scope of the GPCA model illustrated in Figure 5, the *User Interface* and the *System Model* are considered as an environment with which the GPCA model is expected to interact. The User Interface is an environment that displays various messages, and allows users to program the pump. On the other hand, the System Model is an environment that abstracts signals that hardware components might provide to the pump software, such as alarming signals. Each environment is modeled as a separate UPPAAL automaton as illustrated in Figure 8, and the environment model is composed with the UPPAAL model that is translated from the State Controller part of the GPCA model. The formal verification is performed in the context of such environmental behavior at the platform-independent development phase.

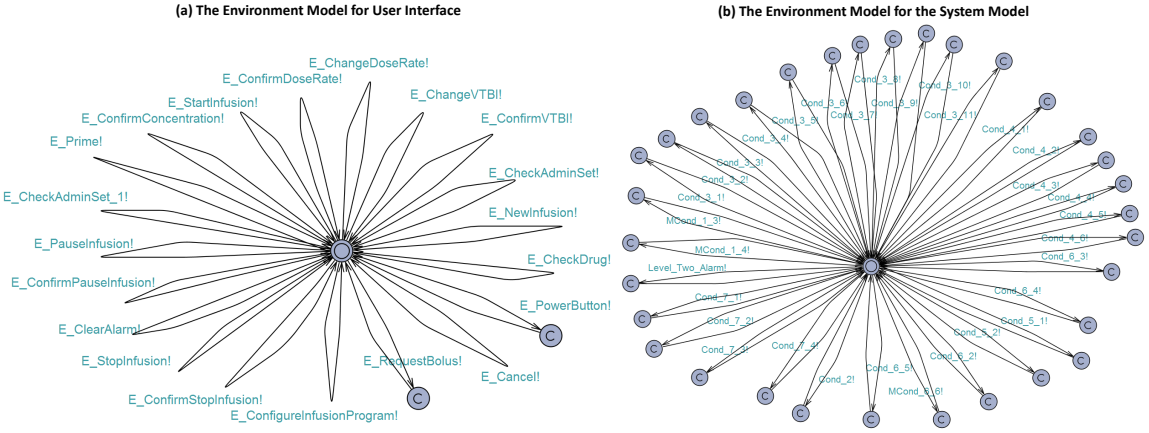


Figure 8: The Environment Model

4.4. Platform-Independent Verification Process

The platform-independent verification process aims at verifying the conformance of a formal model of PCA pump systems with respect to the safety requirements. Ideally, we expect all the GPCA safety requirements to be verified using the UPPAAL model, so that the rest of the development processes can inherit sufficient safety assurance from the beginning. However, we were only able to formally verify some of the GPCA safety requirements using the UPPAAL model for the following reasons.

Firstly, the GPCA model is not detailed enough to verify some of the safety requirements. This is because the GPCA model relies on a different level of abstraction from that of the safety requirements. For example, the GPCA model does not describe detailed functions related to the calculation of the reservoir time remaining, which is mentioned in the following requirement: *the reservoir time remaining shall be re-calculated at the beginning of every bolus dose*. Such functions are considered platform-dependent since different platforms may have different ways of calculating the reservoir time remaining using platform-dependent parameters such as availability of sensors to detect a certain threshold of remaining volume. As another example, the following requirement couldn't be verified for a similar reason since the GPCA model does not detail the behavior of the pump stroke: *if the suspend occurs due to a fault condition, the pump shall be stopped immediately without completing the current pump stroke*.

Secondly, some requirements could not be formalized regardless of the abstraction level of the GPCA model. Even if these requirements were formalized in any form, the meaning of the formal property would be difficult to verify in the context of the safety requirements. For example, the following requirement -*the flow rate for the bolus dose shall be programmable* - requires a system to have the functionality to maintain flow-rate information in memory so that it can be modified on demand. Since this requirement describes implementation-specific functionality, developers need to assure this requirement at the implementation level through validation instead of formal verification.

Thirdly, some safety requirements are too vague from the formal verification perspective, so they can neither be formalized for verification nor be validated at the implementation level. For example, the following requirement -*flow discontinuity at low flows (f ml/hr or less) should be minimal* -, or the requirement - *a clear indication should be displayed any time the drug library is not in use* - contains unclear definitions of *minimal* and *a clear indication*, respectively. These requirements need to be clarified and improved before they can be formally verified at the model level or validated at the implementation level.

Table 3: Categorization of GPCA Safety Requirements

Category 2	
SR 1.5.4.	Reservoir amount remaining shall be re-calculated at the beginning of every bolus dose.
SR 1.6.2.	If the suspend occurs due to a fault condition, the pump shall be stopped immediately without completing the current pump stroke.
Category 3	
SR 1.4.2.	The flow rate for the bolus dose shall be programmable.
SR 1.11.3.	Each log entry shall be stamped with a corresponding date/time value.
Category 4	
SR 1.1.3.	Flow discontinuity at low flows (f ml/hr or less) should be minimal.
SR 5.1.7.	A clear indication should be displayed any time the drug library is not in use.

Based on this observation, we divided these safety requirements into four categories:

- Category 1 : Safety requirements that can be formalized and verified in the UPPAAL model.
- Category 2 : Safety requirements that can be formalized, but the GPCA Simulink/Stateflow model needs additional information to verify them.
- Category 3 : Safety requirements that cannot be formalized, but can be validated at the implementation level.
- Category 4 : Safety requirements that cannot be formalized because they address issues related to the ambient environment of the pump or they are vague in description.

Out of 97 safety requirements, we identified 20 requirements as Category 1; 23 as Category 2; 31 as Category 3; and 23 as Category 4. Table 3 shows example safety requirements belonging to Categories 2, 3, and 4; we introduce how the safety requirements belonging to Category 1 are verified using the UPPAAL model.

Formal Verification of Safety Requirements in Category 1: The GPCA safety requirements belonging to Category 1 are manually translated into a temporal logic formula expressed in the UPPAAL query language. The UPPAAL query language consists of *state formula* and *path formula* [15]. State formula describes an expression that can be evaluated

for a state without looking at the behavior of the model; on the other hand, the path formula quantifies over paths or traces of the model, which can be classified into *reachability*, *safety* and *liveness*. Even though there is no systematic way to translate informal requirements into formal queries, our translation process roughly follows the strategy: (1) expressing the state formula using relevant UPPAAL locations and variable conditions, (2) extending the state formula with the path formula so that the context of the safety requirement is appropriately represented in the form of *reachability* or *safety* or *liveness* properties.

For example, consider the safety requirement, *No bolus dose shall be possible during the POST*. This requirement can be captured by the following UPPAAL query:

$$A[\,]\ (!\ (POST.POST\text{-}In\text{-}Progress\ \&\&\ ISSM.BolusRequest))\ ,$$

where the word *No* is mapped to the logic operator !(not), *POST.POST-In-Progress* is used to indicate a location that abstracts that the POST checking is being performed, and *ISSM.BolusRequest* is used to indicate the location that abstracts that a bolus infusion is in progress. This forms a state formula that is mapped to the safety requirement. Then, the state formula is further extended with the path formula $A[\,]$ (invariantly) to express a safety property that enforces the above state formula to be satisfied by the model in all its executions.

Consider another requirement, *the pump shall issue an alert if paused for more than t minutes*. In order to write a state formula for this requirement, three locations are identified from the UPPAAL model: *ISSM.InfusionPaused* is a location that abstracts a pump is in a paused state after a user presses a *confirm-pause* button in order to pause the current infusion for a while. *ISSM.PausedStopConfirm* is another location that can be reached from the *ISSM.InfusionPaused* location by pressing a *stop-infusion* button; *ISSM.PausedStopConfirm* location abstracts that a pump is in a paused state waiting for a user to press a *confirm-stop* button in order to completely stop the current infusion. If a pump is in either *ISSM.InfusionPaused* or *ISSM.PausedStopConfirm* location, the pump is expected to be in a

state where the current infusion is paused. On the other hand, *ISSM.Alrm-TooLongInfusion-Pause* is a location that abstracts that a pump is raising an alarm due to the fact that the pump is paused for more than a certain time-limit that is defined as t minutes in the safety requirement.

One can write a state formula, *ISSM.InfusionPaused* && *x1* > *MAX-PAUSED-T*, where *x1* is a clock variable reset to 0 when it enters the *ISSM.InfusionPaused* location, to indicate a condition that the pump stays in the location *ISSM.InfusionPaused* for more than t minutes. Here, t is instantiated as a constant variable *MAX-PAUSED-T*; and this state formula is extended with the path formula $\rightarrow$ (lead to) to express that if the former state formula holds, the later state formula, *ISSM.Alrm-TooLongInfusionPause*, will eventually satisfy. The corresponding UPPAAL query is:

$$(\textit{ISSM.InfusionPaused} \;\&\&\; x1 > \textit{MAX-PAUSED-T}) \rightarrow \textit{ISSM.Alrm-TooLongInfusionPause}$$

Similarly, *ISSM.PausedStopConfirm* should be extended with the path formula as follows:

$$(\textit{ISSM.PausedStopConfirm} \;\&\&\; x1 > \textit{MAX-PAUSED-T}) \rightarrow$$
$$\textit{ISSM.Alrm-TooLongInfusionPause}$$

Several more examples of requirement formalization are shown in Table 4. The current GPCA safety requirements contain 97 requirements. We translated 20 requirements into UPPAAL queries, and verified them in the UPPAAL model. More details of the platform-independent verification can be found in [33].

In order to give intuition as to how such formal verification can be used to detect defects of software designs, we give an example of the verification result and analysis here. In fact, the last query turns out to be *not* satisfied in the UPPAAL model that is translated from the GPCA model. Provided with a counter example produced as a result of the verification, the value of the clock variable *x1* can diverge forever in the *ISSM.PausedStopConfirm* location; and the root cause is that the original GPCA model did not have a transition

Table 4: Mapping between Safety Requirements and UPPAAL queries

Category	**Safety Requirement(SR) / Safety Property(SP)**
SR 1.4.3	No normal bolus doses should be administered when the pump is alarming (in an error state).
Query	A[](! (ISSM.BolusRequest && CDR.Alrm-UnknownDrug))
SR 3.4.3	The POST shall take no longer than t seconds.
Query	(POST.Post-In-Progress && x1 > MAX-POST-WAIT) → POST.Alrm-POSTFailure
SR 1.5.6	If the calculated volume of the reservoir is y ml, and an infusion is in progress, an Empty Reservoir alarm shall be issued.
Query	(ISSM.Infusion-NormalOperation && Cond-6-3 == true) → (ISSM.Alrm-EmptyReservior)
SR 2.2.4	If the pump is idle for t minutes while programming a dose setting, the pump shall issue an alert to indicate that the user needs to finish programming and start infusion.
Query	(ICR.ChangeDoseRate && x1 > MAX-WAIT-INPUT-T) → (ICR.Alrm-LongWait-ChangeDoseRate)

from *ISSM.PausedStopConfirm* to *ISSM.Alrm-TooLongInfusionPause* that should be taken upon time-out; consequently, the UPPAAL model, translated from the GPCA model, does not have the corresponding transition either. We fixed this design defect (*i.e.,* adding a missing transition to the UPPAAL model)[3], and moved to the next development process after verifying all safety requirements belonging in Category 1. One may manually find such a type of design defect in an ad-hoc manner, but we believe that identifying such defects in an automatic fashion using formal verification will result in a higher safety assurance of systems.

4.5. Platform-Independent Code Generation Process

The platform-independent code generation process aims at automatically generating source code from a verified system model, which is expected to operate on a range of target platforms. Such a systematic transformation process, typically implemented as a part of a code generator, reduces error-prone aspects of a manual implementation conducted in an ad-hoc fashion. A code generator is able to generate source code (*e.g.,* C code) that implements transition tables, boolean (or integer) variables to represent input and output

[3]We also reported this verification result to the FDA, and expect it to be reflected in the next version of the GPCA model.

occurrences and execution logic (switch-case of if-then-else statements), which maps to the model structure. In this section, we formally define the scope of the platform-independent code in the final implemented system (Subsection 4.5.1), and explore how existing code generators generate source code for such a structural correspondence between a model and source code (Subsection 4.5.2), and propose additional support required for its better integration with different platform-dependent code (Subsection 4.5.3).

4.5.1. System boundaries of the final implemented system

The platform-independent code generated from the model will be executed as a part of the final implemented system. We formally define a system boundary of the final implemented system in order to precisely specify the scope of the platform-independent code.

Figure 9-(a) illustrates the platform-independent model that interacts with its environment model. Such model-level interactions are expressed using input and output semantics provided by modeling languages; for example, UPPAAL provides channel synchronizations or condition variables that can abstract input and output interactions among different UPPAAL automata. Once verifying the timing requirements at the model level, the platform-independent code generation process, illustrated in Figure 9-(c), transforms the platform-independent model into source code. As illustrated in Figure 9-(b), the platform-independent code constitutes only a part of the final implemented system, and it is expected to be integrated with a particular platform-dependent code that is not yet known at the moment when the platform-independent code is produced.

We argue that the relationship between (1) how the platform-independent model interacts with its environment model and (2) how its generated code interacts with its real environment should be precisely defined in order to reason about the timed behavior of the final implemented system w.r.t. its model behavior. However, due to the fact that the platform-independent code forms only a part of the final implemented system, there is a mismatch in mapping the system boundary. That is, the input and output interaction occurring at the

model level can be matched to the interaction between the platform-independent code and the platform-dependent code; or it can be matched to the interaction between the hardware platforms and the real environment. We believe that a uniform interpretation is necessary for precise reasoning about the relationship between the platform-independent model and the implemented system that executes the generated code from it.

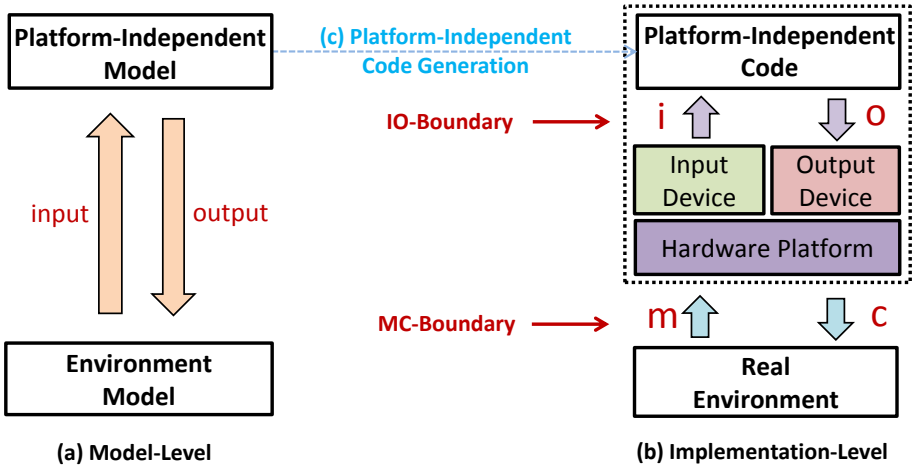


Figure 9: The mapping between the model and the implementation using Parnas' four variables

We propose a mapping of the system boundary between the platform-independent model and its implemented system using Parnas' four-variable model. The Four-variable model originally proposed by Parnas is a well-known technique in requirement engineering and has been used to precisely describe safety-critical system requirements [43]. In this formalism, the separation among three subsystems - *Software*, *Input Device*, and *Output Device* - is explicitly defined using four variables, namely, *monitored* (m), *controlled* (c), *input* (i) and *output* (o). In our context, the *Software* refers to the platform-independent code that is generated from the model; the *Input-Device* and *Output-Device* refer to the platform-dependent code that needs to be integrated for the execution of the platform-independent code on a particular platform. We give a brief description of the role of each variable:

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 that 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, the *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 the platform-independent code. An input variable (i) characterizes events that are read by the platform-independent code. For example, the platform-independent code may have *i*-variables, *i-BolusReq*. *Input-Device* (a part of the platform-dependent code) is responsible for converting the events in *m*-variable that is changed by the real environment into the events in *i*-variable that is read by the platform-independent code. Sensors and their accompanied device drivers are the example of *Input-Device*. An output variable (o) characterizes events that are written by the platform-independent code. For example, the platform-independent code may have *o*-variables, *o-PumpMotor*. *Output-Device* (a part of the platform-dependent code) 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.

Two aspects need to be considered in reasoning about the relationship between the platform-independent model and an implemented system:

1. How the timing of the input/output synchronizations in the model-level is mapped to the implementation-level?

2. How the instantaneous synchronization semantics in the model-level is mapped to the implementation-level?

Here is our mapping considered in this dissertation: the timing of the input or output synchronization occurrence in the platform-independent model is mapped to the timing of the corresponding input and output event occurrences at the *mc*-boundary. That is, an implemented system (i.e., composition of the platform-independent code and a platform) should guarantee the same timing constraints at the *mc*-boundary in order to conform to the timing constraints verified at the model level.

The processing delays of the synchronization semantics at the model level is considered instantaneous, and this instantaneous timing semantics is mapped to non-zero processing delays incurred from a platform. That is, a platform will add certain amounts of input/output processing delays; an input event occurred from the *mc*-boundary requires input processing delay until it gets delivered to the platform-independent code, and an output produced by the platform-independent code at the *io*-boundary requires output processing delay until it gets written to the environment. In Chapter 5, we explain what aspects contribute to these platform-processing delays, and in Chapter 6 and 7, we explain how to reason about the impact of such platform-processing delays to the timed behavior of the implemented system. In this chapter, we focus on how existing code generators generate code from the platform-independent model, and what kind of considerations is required to design a better code generator for the platform-independent code, while deferring the discussion about the platform-dependent timing aspects to the later chapters.

4.5.2. Code Generation Mechanisms of Existing Code Generators

RealTimeWorkshop is a code generator that produces C source code from Simulink/Stateflow models [40], and TIMES is a code generator that produces C source code from UPPAAL models [14]. Even though our platform-independent model is created using UPPAAL, comparing source code generated from both code generators helps us to understand the common

issues that arise at the integration stage. Therefore, we give two example models, one written in Simuilink/Stateflow and the other written in UPPAAL, and explore common aspects that need to be considered for the integration with different platform-dependent codes.

Figure 10 and Figure 11 illustrate the two models that are constructed using Simulink/Stateflow and UPPAAL, respectively. The two models illustrate the software behavior of the PCA infusion pump during an infusion session that is in progress, which is motivated from the *Infusion Session Submachine* of the GPCA model. It is noted that even though the two models looks syntactically similar, the semantics of their behavior are different from each other. We give informal description of their model semantics here that is enough to understand the generated code from each model.

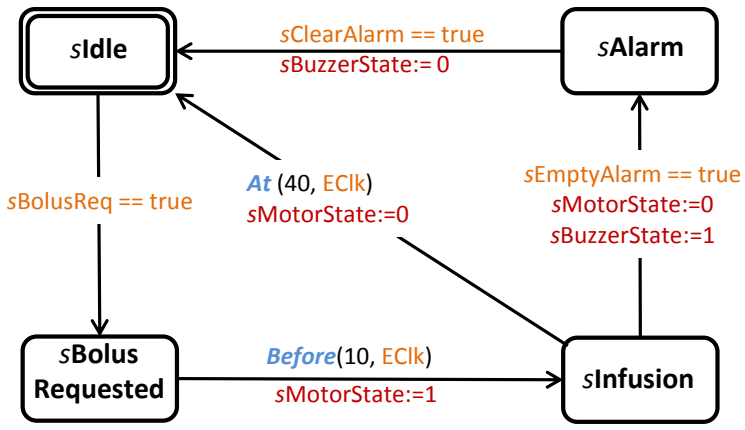


Figure 10: The example Stateflow model

For a clear explanation of the two models, the syntax of the Stateflow model is annotated with a prefix '*s*' and the syntax of the UPPAAL model is annotated with a prefix '*u*'. Figure 10 is a Stateflow model that abstracts the timed behavior of a PCA infusion pump. This Stateflow model is invoked at every *EClk* event that is periodically triggered from a Simulink model that is not shown here; upon invocation, it executes the transition system based on the Stateflow semantics. It has four locations and five variables to express input and output; *sIdle*, which is an initial location, represents a pump that is waiting for a bolus request from a patient; *sBolusRequested* is entered when a patient requests a bolus by

pressing a bolus request button, which is expressed as a variable condition *sBolusReq* == *true*; a transition from *sBolusRequested* to *sInfusion* is taken before an *EClk* event occurs 10 times since *sBolusRequested* is entered; *sInfusion* illustrates that a pump is providing a bolus infusion, and is entered by updating a variable *sMotorState* := 1 where 1 implies that a pump motor is in a rotating status; a transition from *sInfusion* to *sIdle* is taken when no empty-alarm occurs until *EClk* triggers 40 times; otherwise, a transition from *sInfusion* to *sEmptyAlarm* is taken by updating *sMotorState* := 0 and *sBuzzerState* := 1; the *sEmptyAlarm* location illustrates that a pump is alarming since a drug reservoir becomes empty, and when a caregiver presses a *clear-alarm* button, which is implied in the update *sClearAlarm* == *true*, it takes a transition to *sIdle* by updating a variable *sBuzzerState* := 0.

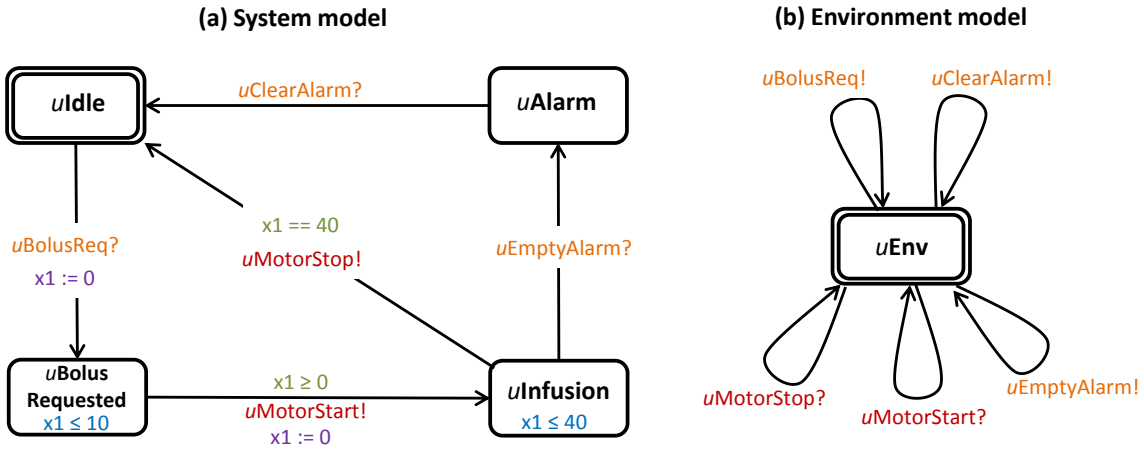


Figure 11: The example UPPAAL model

Figure 11 is a UPPAAL model that abstracts the similar timed behavior with the Stateflow model in Figure 10. In comparison to the condition variables used in the Stateflow model, channel synchronizations are used in the left side UPPAAL automaton to express input (*e.g., uBolusReq?*) and output (*e.g., uMotorStart!*) behavior with its environment on the right side. In addition, a clock variable x_1 is used to express a progress of continuous time; this clock variable is used in clock reset statements (*e.g.,* $x_1 := 0$) or invariants (*e.g.,* $x_1 \leq 10$) or clock guard conditions (*e.g.,* $x_1 == 40$) in order to represent the timed behavior.

We briefly explain the automatically generated code from the two example models. Listing 4.1 and Listing 4.2 are the code snippets that are automatically generated from the Stateflow model in Figure 10 using RealTimeWorkshop. Three input variables used in the Stateflow model are mapped to three boolean variables that are encapsulated in U, which is a C-structure type. In addition, *EClk* is generated as a part of input variables whose value changes results in the execution of the logic in Listing 4.2. The execution logic of the transition system, so called *step* function, is generated to perform the following actions in order; (1) checks the current location using a *locID* variable, (2) checks guard conditions associated with its temporal counter (*e.g., tCnt*) or input variables (*e.g., U.sEmptyAlarm*), (3) perform updates on temporal counters or output variables (*e.g., Y.sMotorState*), and (4) updates a new location using a *locID* variable.

Listing 4.1: (RTW) Mapping of input and output variables

```
/* External inputs */
typedef struct {
  boolean_T sBolusReq;
  boolean_T sEmptyAlarm;
  boolean_T sClearAlarm;
  real_T EClk;
} U;

/* External outputs */
typedef struct {
  boolean_T sMotorState;
  boolean_T sBuzzerState;
} Y;
```

Listing 4.2: (RTW) Step function logic

```
function step (void)
  tCnt := tCnt + 1;
  if EClk is triggered and a chart is active
    switch (locID)
      case sBolusRequested:
        if tCnt < 10
          Y.sMotorState := True;
          locID := sInfusion;
          tCnt := 0;
        endif
        break;
      case sEmptyAlarm:
        if U.sClearAlarm
          Y.sBuzzerState := False;
          locID := sIdle;
        endif
        break;
      case sIdle:
        if U.sBolusReq
          locID := sBolusRequested;
          tCnt := 0;
        endif
        break;
      case sInfusion:
        if U.sEmptyAlarm
          Y.sMotorState := False;
          Y.sBuzzerState := True;
          locID := sEmptyAlarm;
        else
          if tCnt == 40
            Y.sMotorState := False;
            locID := Idle;
          endif
        endif
        break;
    endswitch
  endif
endfunction
```

Listing 4.3 and Listing 4.4 and Listing 4.5 and Listing 4.6 are the code snippets that are automatically generated from the UPPAAL model in Figure 11 using TIMES. The transition table shown in Listing 4.3 encodes information in the *trans-t*, which is a C structure type, as to (1) whether a transition is active or not, (2) a source location, (3) a destination location, (4) an associated channel synchronization. Note that Figure 11 has a total of 10 transitions, so the corresponding *trans-t* contains the same number of items. The *check-trans* function in Listing 4.4, which encodes the execution logic and whose purpose is similar to the *step* function in Listing 4.2, evaluates transitions defined in the model using the *eval-guard* function in Listing 4.5. Note that if a transition t in the UPPAAL model contains channel synchronization such as *uEmptyAlarm?*, *check-trans* tests its complement transition t', which in this case would be defined in the environmental model. Also note that evaluating the clock guard conditions (*e.g.,* $x_1 == 40$) or assigning a new value to the clock variables (*e.g.,* $x_1 := 0$) are performed through calling API *rdClock* and *setClock* as shown in Listing 4.5 and Listing 4.6, respectively.

Listing 4.3: (TIMES) pseudo-code of *transition table*

```
1  trans_t TRANS[NB_TRANS] = {
2    {true, uIdle, uBolusRequsted, uBolusReqS},
3    {false,uBolusRequsted,uInfusion,uMotorStartR},
4    {false, uInfusion, uEmptyAlarm, uEmptyAlarmS},
5    {false, uEmptyAlarm, uIdle, uClearAlarmS},
6    {false, uInfusion, uIdle, uMotorStopR},
7    {true, uEnv, uEnv, uBolusReqR},
8    {true, uEnv, uEnv, uMotorStopS},
9    {true, uEnv, uEnv, uEmptyAlarmR},
10   {true, uEnv, uEnv, uClearAlarmR},
11   {true, uEnv, uEnv, uMotorStartS}
12 };
```

Listing 4.4: (TIMES) pseudo-code of *check-trans*

```
1  function check-trans
2    for each transition t ∈ trans-t array
3      if t is active and eval-guard(t) is true
4        if t contains a channel synchronization.
5          if there exists a t'’s complement
           transition, t', and eval-guard (t') is true
6            assign(t) and assign(t')
7          endif
8        else if t has no channel synchronization
9          assign(t)
10       endif
11     endif
12   endfor
13 endfunction
```

Listing 4.5: (TIMES) *eval-guard* function

```
1  bool eval_guard(int trn) {
2    switch(trn) {
3    case 1: return (rdClock(x1)>=0);
4    case 4: return (rdClock(x1)==40);
5    case 0:
6    case 2:
7    case 3:
8    case 5:
9    case 6:
10   case 7:
11   case 8:
12   case 9:
13     return true;
14   }
15   return false;
16 }
```

Listing 4.6: (TIMES) *assign* function

```
1  void assign(int trn) {
2    switch(trn) {
3    case 0:
4      setClock(x1,0); break;
5    case 1:
6      setClock(x1,0); break;
7    }
8  }
```

This amount of source code that constitutes the platform-independent code in Figure 9-(b) correctly translates the model structure into that of source code. However, such platform-independent code cannot be executed by itself without the appropriate platform-dependent support. In Subsection 4.5.3, we summarize the necessary platform-dependent supports that need to be considered during the platform-independent code generation toward its integration with different platform-dependent code.

4.5.3. Separating concerns for the interface implementation of the platform-independent and platform-dependent code

Based on the observation from the two existing code generators in Section 4.5.2, we categorize the necessary parts to be considered for the integration between the platform-independent and dependent code. Next, we propose a way to generate code that implements those parts as a set of primitives. The implementation of these primitives are separated as a platform-independent part and a platform-dependent part as illustrated in Figure 12. That is, signatures of such primitives are generated as a part of the platform-independent

code, and the implementation of those primitives is generated as a platform-dependent code. In this Subsection, we introduce signatures of such primitives that need to be generated along with the platform-independent code. In Chapter 5, we introduce how such platform-dependent details to implement such primitives can be represented in the form of the AADL model, and automatically generated in a platform-dependent manner.

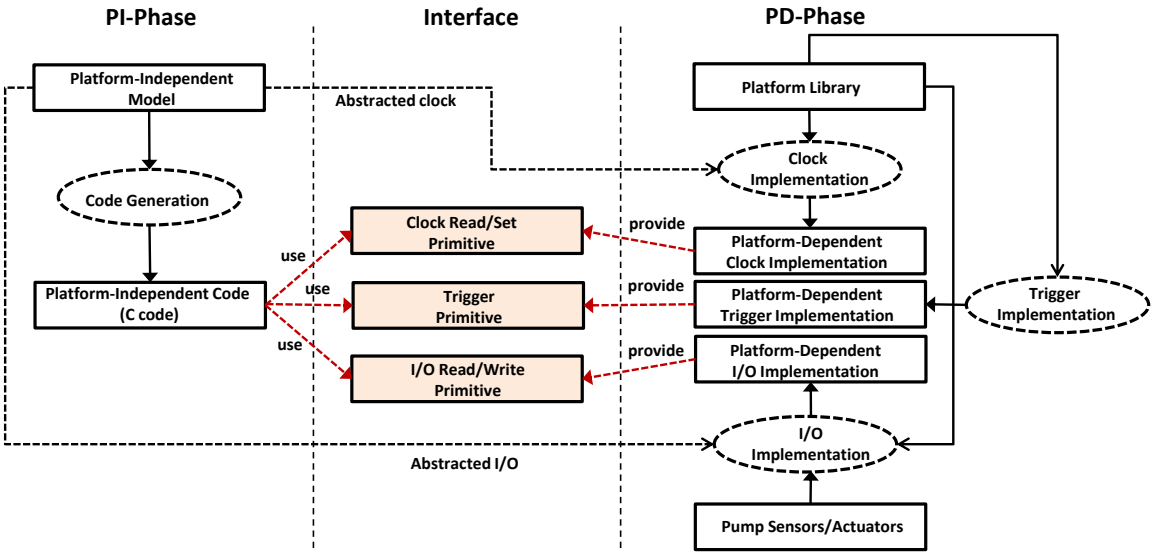


Figure 12: The primitives for the interaction between the platform-independent and platform-dependent code

(Category 1) Interfacing platform-independent I/O operations: The platform-independent code shown in Subsection 4.5.2 includes a set of input and output variables that are expected to be integrated with the platform-dependent code that processes the input and output of its environment. For example, the boolean variables in Listing 4.1 are mapped to input and output variables of the Stateflow model in Figure 10. The value of each variable is examined by the execution logic (*step* function) in Listing 4.2 in order to make a decision for relevant input or output transitions. Similarly, Listing 4.3 includes channel synchronization between the platform-independent model and its environment model in the form of a transition table which is used in *check-trans* function to perform relevant input or output transitions.

The level of abstraction of these input and output variables are matched to the *IO-boundary* in Figure 9 that has yet to be integrated with the platform-dependent code that handles input and output at the *mc*-boundary level. For example, the value of *sBolusReq* in Listing 4.1 is expected to change from *false* to *true* when a patient presses a bolus request button. The platform-dependent code accepts such sensor input at the *mc*-boundary and processes it in a way that the value of *sBolusReq* is changed accordingly at the *io*-boundary. Similarly, when the *step* function in Listing 4.2 updates the output value of *sMotorState* from *false* to *true*, this output update needs to result in the rotation of the pump-motor. The platform-dependent code accepts such platform-independent output at the *io*-boundary and processes it in a way that the value changes become visible to the real environment at the *mc*-boundary (*e.g.,* by providing electrical signal to the pump motor).

In order to implement such I/O interactions, we define *read*(*i-variable*) and *write*(*o-variable*) primitives. The platform-independent code is generated with *read*(*i-variable*) signatures that return values associated with an input *i-variable*; the value is written by the platform-dependent code by the transformation of the *m-event* (at the *mc*-boundary) into an *i-event* (at the *io*-boundary). For example, three signatures - *read*(*uBolusReq*), *read*(*uEmptyAlarm*), *read* (*uClearAlarm*) - are generated from the UPPAAL model in Figure 11 as a part of the platform-independent code; but, it is platform-dependent how actually the internal operation of the read primitives works on different target platforms. Similarly, *write*(*o-variable*) signatures that are used to write output values, which is expected to be read by the platform-dependent code. For example, two signatures - *write*(*uMotorStart*), *write*(*uMotorStop*) - are generated from the same UPPAAL model as a part of the platform-independent code, leaving its internal implementation to the platform-dependent code. The I/O interaction between the platform-independent and platform-dependent code can only be performed through these *read* and *write* primitives.

(Category 2) Interfacing platform-independent clock operations: In case a platform-independent model deals with a notion of time to express timed behavior of a system, its

corresponding code should have a way to implement such timing semantics. For example, the UPPAAL model in Figure 11 uses a continuous clock variable *x1* that is proportionally increased over real-time; this clock variable is reset to zero or its current value is compared with constant values (*e.g.,* 0 or 40) to represent timed behavior. As a result, its generated code includes a way to implement such timing semantics. For example, *eval-guard* function in Listing 4.5 uses a generated *rdClock*(*x1*) function that returns the current value of clock variable *x1*; a returned value is compared with two constant values in line 3 and 4 in order to check whether the timing guard conditions associated with the two transitions are satisfied. Similarly, the *assign* function in Listing 4.6 uses a generated *setClock*(*x1*, 0) function in order to reset the clock to a new value 0.

In order to implement such clock operations, we define *readClock*(*clk*) and *setClock*(*clk*, *val*) primitives. The platform-independent code is generated with *readClock*(*clk*) signatures that return the current value of a clock whose identification is *clk*; the value of *clk* is increased by the platform-dependent code in proportion to real-time. Similarly, *setClock*(*clk*, *val*) signature is generated as a part of the platform-independent code, which resets the current value of a clock with a specific value, *val*; it is platform-dependent how such a reset operation is implemented. The clock operation between the platform-independent and platform-dependent code can only be performed through these *readClock* and *setClock* primitives.

(Category 3) Interfacing invocation mechanisms of the platform-independent execution logic: The platform-independent code includes an execution logic that implements transition semantics based on the current inputs and clock values, and produces outputs under the assumption that category 1 and 2 are correctly interfaced with the platform-dependent code. For example, the *step* function in Listing 4.2 and *check-trans* function in Listing 4.4 are examples of such an execution logic. These functions are expected to be invoked according to some invocation mechanisms (*e.g.,* periodic invocation or aperiodic invocation); upon invocation, transition conditions are examined to be taken.

In order to implement such invocation mechanisms, we define a *trigger* primitive. The platform-independent code is generated with a *trigger* primitive whose internal is an implementation of an execution logic that maps to a model structure; but, it is platform-dependent how this trigger primitive is invoked on a target platform.

In [32], we demonstrated how these primitive signatures generated as a part of the platform-independent code can be integrated with two different PCA pump platforms that implement the internal of the primitives in a different way.

4.6. Summary of the PI-Phase

In the PI-Phase, we showed how the timed behavior of a system can be modeled independently of a particular platform. The basic idea of the platform-independent timing abstraction is to express the timing and dependency of input and output events occurring at the *mc*-boundary (*i.e.,* the system boundary between a platform and the environment). On the other hand, this level of timing abstraction hides internal platform-specific details of how such input and output can be processed by a platform and how a platform internally interacts with software at the *io*-boundary (*i.e.,* the system boundary between a platform and the platform-independent code). In order to define such an abstraction level of PCA pumps, we utilized the FDA's GPCA model that expresses generic behavior of PCA pumps; and we showed the formal verification result by translating it into the UPPAAL model.

It is also challenging to obtain the platform-independent code from the model since we do not know yet what platforms will be integrated with the platform-independent code in this development phase. In order to facilitate the integration process with unknown platforms in the later stage, we defined three types of primitives, by comparing two existing code generators, that are necessary to operate the platform-independent code, but need to be implemented in a platform-dependent way. The platform-independent code is generated using the primitive signatures only while having their internals implemented by a particular platform.

In the next chapter, we explain what kind of platform-dependent timing information needs to be considered to generated platform-dependent code, and how the primitive signatures used in the platform-independent code can be implemented in a platform-dependent manner.

CHAPTER 5 : Platform-Dependent Development Phase

5.1. The Problem Statements and Challenges

A *platform* is defined as a hardware (*e.g.,* sensors and actuators) and a software stack (*e.g.,* accompanied device drivers and operating systems) on which the platform-independent code is expected to be operating. The platform-independent code that has been developed in the PI-Phase is not sufficient by itself to operate on a platform; for example, the internal implementation of the primitive signatures (*e.g.,* trigger primitives or I/O primitives) has not been completed yet. Roughly speaking, the platform-dependent code is defined as a source code that implements particular architectural timing aspects that enable the platform-independent code to interact with the hardware through APIs provided from the software stack. Here are two objectives of the PD-Phase:

- (G1) the PD-Phase should provide a *systematic* way to implement its platform-dependent code that is compatible with a given hardware platform,
- (G2) the PD-Phase should provide a *flexible* way to implement the platform-dependent code for a range of platforms.

We note that achieving (G1) does not necessarily mean achieving (G2), because a systematic approach that implements a platform-dependent code targeting a particular platform may not equally apply to another platform. For example, if the PD-Phase has a systematic way to implement the trigger primitive for the platform A, it may not apply to the platform B, that has a different software stack. We also note that achieving (G2) does not necessarily mean achieving (G1) either, because one can implement different platform-dependent codes in a non-systematic way; for example, the PD-phase may not place any restriction in the way of implementing the platform-dependent code, leaving one to implement it in an ad-hoc manner. There are several challenges to achieve both (G1) *systemization* and (G2) *flexibility* in order to implement the platform-dependent code.

Challenge 1: Among a range of abstraction levels that are relevant to characterizing a platform, one needs to choose an appropriate level of abstraction to describe a platform for systematic generation of the platform-dependent code. The range of the level of abstraction for defining a platform is wide. For example, in order to control sensors and actuators that are physically interfaced to a micro-controller, initializing appropriate values for corresponding registers (*e.g.,* general purpose I/O) of the micro-controller is of an important concern. In case multiple threads are expected to run on a platform, designing a real-time scheduling algorithm to schedule those threads is also of an important concern. Using appropriate code patterns imposed by the underlying software stack is also of an important concern. In order to systematically generate the platform-dependent code, however, considering all levels of platform abstraction is practically difficult, and there is no consensus about which level of abstraction is most appropriate. The challenging part is to decide a right abstraction level in describing a platform, and to provide the means to formally express such aspects from which source code can be systematically generated that can support the execution of the platform-independent code.

Challenge 2: As a platform changes from one to another, different styles of code should be generated due to heterogeneous aspects of platforms. In general, as a platform changes, its platform-dependent code is likely to change accordingly for several reasons. Each platform adopts its own architectural option to support the execution of the platform-independent code. For example, a platform may implement the trigger primitive using a periodic invocation mechanism (*i.e.,* the platform-independent code is executed every pre-defined period); others may use an aperiodic invocation mechanism (*i.e.,* the platform-independent code is executed whenever events arrive). A platform may have multiple options for reading environmental input (*e.g.,* polling or interrupt-based mechanisms). In addition, each platform has its own software stack that provides a set of APIs to implement a certain architectural option. For example, source code that implements the periodic invocation mechanism using real-time operating system A is different from the source code using real-time operating system B since the two operating systems may provide different

sets of APIs and code patterns that need to be followed to implement the same architectural option. In order to support flexibility that allows choosing a range of platforms, one needs to deal with such variations that appear whenever a platform changes.

We give the overview of our approach to tackle these challenges in the next section.

5.2. The Approach Overview of the PD-Phase

We propose a way to abstract the timing aspects of a platform using Architectural Analysis Description Language (AADL) model and a code snippet repository from which the platform-dependent code is systematically generated.

AADL is a modeling language for describing real-time embedded systems from an architectural perspective. In particular, we abstract architectural information flows that need to be implemented as a platform-dependent code using a subset of AADL; that is, an AADL model captures how a platform reads sensor values (*e.g.,* polling or interrupt-based mechanism), how a platform processes and stores the retrieved sensor values to be read by the platform-independent code (*e.g.,* shared variable or buffer mechanism), how a platform invokes the platform-independent code for its computation (*e.g.,* periodic or aperiodic invocation), and how the produced output from the platform-independent code is delivered back to a platform, which can be used to control actuators. From the systematization point of view (G1), the AADL provides a means to formally express such architectural aspects from which the platform-dependent code can be systematically generated. From the flexibility point of view (G2), as a platform changes from one to another, the modular aspects of AADL enable such architectural changes to be easily accommodated.

On the other hand, the proposed code snippet repository captures code patterns that implement an AADL model using a programming interface provided by a software stack of a particular platform. From the systematization point of view (G1), a code snippet repository is categorized according to their functions to implement AADL components from which a code can be systematically generated. In other words, one can implement a code generation

algorithm such that, given an AADL model, the algorithm finds a corresponding category that contains a particular code snippet in order to systematically generate respective platform-dependent codes. From the flexibility point of view (G2), the category is general enough so that each platform can fill the categories with different code snippets that are compatible with their respective programming interfaces. In other words, the categorization of the code snippet repository remains the same across different platforms, but the contents of each category changes to accommodate the different code patterns required to implement an AADL model on respective platforms.

Given a platform description using an AADL model and a code snippet repository, the proposed code generation algorithm systematically composes the two resources to systematically generate the platform-dependent code. Figure 13 illustrates the overview of the PD-Phase, and a detailed explanation of our approach is given in the following sections.

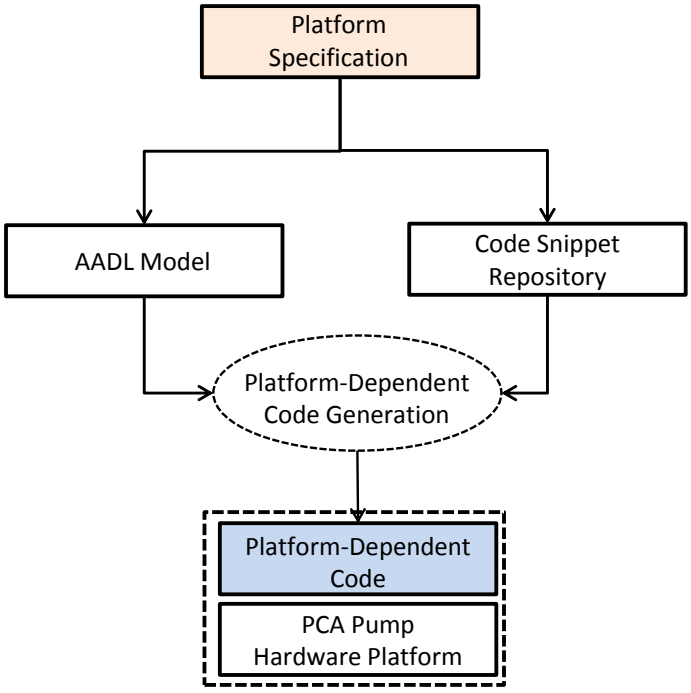


Figure 13: The platform-dependent phase of the safety-assured model-based implementation

5.3. Platform-Dependent Software Aspects

Consider the following Category 1 and Category 3 of the primitives explained in Subsection 4.5.3 of the PI-Phase:

- (Category 1) Interfacing platform-independent I/O operations,
- (Category 3) Interfacing invocation mechanisms of the platform-independent execution logic.

The platform-independent code is generated using primitives belonging to each category expecting that the internals of these primitives are to be implemented in a platform-dependent manner. This chapter explains how such primitives can be systematically implemented in a platform-dependent manner.

Figure 11 is a platform-independent model in a sense that it does not explicitly express the timing information on how it should be executed on target platforms. Hence, its generate code (*i.e.,* the platform-independent code) cannot be executed without the appropriate platform-dependent supports. Two orthogonal aspects need to be considered toward implementation of such platform-dependent supports:

Platform-dependent architectural aspects: Each platform has its own platform dependent mechanism to execute the same platform-independent computation. Regarding Category 1, different platforms can implement the semantics of *uBolusReq?* differently: one platform may implement a periodic thread that samples the status of the electrical signal level of the patient-controlled button, wheres another platform may implement an aperiodic thread that is invoked upon the interrupt-trigger when it detects the change of the signal level of the button. These two platforms equally have a capability to read the input from the environment, but in a different way. Regarding Category 2, different platforms can invoke the platform-independent code differently: one platform may execute the platform-independent computation as a periodic thread, which periodically reads input from

the platform, computes transitions, and writes output to the platform. However, another platform may execute the same computation as an aperiodic thread, which performs the computation only when input arrives. For example, the platform-independent model in Figure 11 may periodically read inputs for *uBolusReq?*, *uEmptyAlarm?*, *uClearAlarm?*, and then take the corresponding outgoing transitions and write outputs. However, the model may also read those inputs aperiodically; that is, the platform-independent computation is performed only if any of those events occurs.

We call these aspects the platform-dependent architectural timing aspects. Such differences in executing the platform-independent computation are captured using AADL models, from which source code that is compatible to each platform can be generated.

Platform-dependent programming interfaces: Each platform has its own software stack (*e.g.,* real-time operating system) that provide APIs to access to the platform-dependent mechanisms. As an example, Listing 5.1 shows the code snippet that implements a periodic thread for empty reservoir detection running on FreeRTOS [8]. The code snippet includes several API calls that are provided by FreeRTOS. *TaskCreate* API (Line 8) is called to register necessary information to the OS kernel such as thread's priorities and callback functions. *vTaskDelayUntil* API (Line 17) is called to block the callback function *cbEmptyRsv* until the next invocation period (in this example, the period is 500 ms).

We call these aspects the platform-dependent programming interfaces. Such different code snippets are categorized according to their functions to implement AADL components such as periodic/aperiodic threads and their interactions. This code snippet category is used to generate different platform-dependent source code along with AADL models.

Listing 5.1: Code snippet of a periodic task in FreeRTOS

```
1 //Declaration part
2 const portTickType periodEmptyRsv=500;
3 const portBASE_TYPE prioirtyEmptyRsv=2;
4 const portBASE_TYPE stacksizeEmptyRsv=500;
5
```

```
6  //Initialization part
7  void init_EmptyRsv( void ){
8   TaskCreate(cbEmptyRsv,''EmptyRsv'',stacksizeEmptyRsv, NULL,prioirtyEmptyRsv,NULL);
9  }
10
11 //Thread callback function part
12 void cbEmptyRsv (void* pvParameters){
13  portTickType xLastWakeTime;
14  xLastWakeTime = xTaskGetTickCount();
15  for(;;){
16   //Wait for the next cycle
17   vTaskDelayUntil(&xLastWakeTime,periodEmptyRsv);
18   //Perform action here
19   //(1)Read
20   //(2)Compute
21   //(3)Write
22  }
23 }
```

5.4. Platform-Dependent Modeling using AADL

This section explains how to capture the platform-dependent architectural timing aspects using AADL models. The differences in the hardware/software architectures are highlighted from the code generation perspective.

AADL is a modeling language for describing real-time embedded systems from an architectural perspective. It provides an abstraction of software components (*e.g.,* periodic/aperiodic threads) and hardware components (*e.g.,* devices and processors). The interactions among such components are abstracted using ports and port connections. We note that only a subset of AADL components and their semantics is used to explain the idea underlying the platform-dependent code generation; a broader scope of AADL components and their semantics can be found in [1].

Figure 14 shows the graphical representations of two AADL models, M1 (top model) and M2 (bottom model), that specify the hardware/software architectures of two different in-

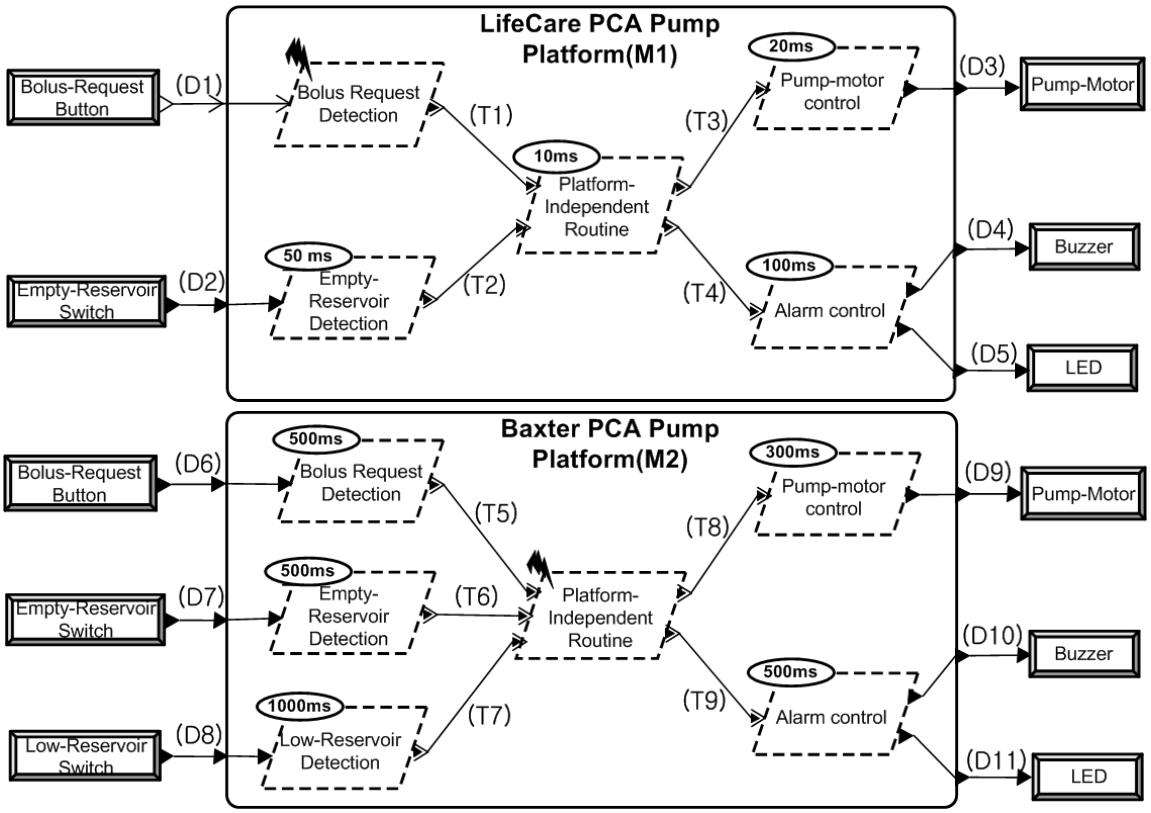


Figure 14: AADL models of two infusion pump platforms

fusion pump systems. The big rounded box in the center of each model denotes a *system component*, which represents the scope of the software system. Each system component contains several *thread components*, represented by the dotted rectangles. Some thread components are connected to *device components*, represented by the double-lined rectangles outside the system component. Thread and device components are interconnected to each other using *port connections*, denoted by the different types of directional lines in the figure. The informal semantics of each component is as follows.

Threads: A thread is a concurrent schedulable unit of sequential computation, with one or more assigned properties. There are five threads in M1 and six threads in M2. Each thread has a property of *dispatch protocols*. For instance, the M1.BolusRequestDetection thread has an *aperiodic* dispatch protocol: any event that arrives on its input ports can invoke the thread to perform its execution; upon completion, the thread becomes idle until the next

event occurrence on its input ports. In contrast, the M2.BolusRequestDetection thread has a *periodic* dispatch protocol: the thread is invoked periodically (*e.g.,* every 500ms in the example) and independent of the occurrences of events on its input ports. There are several other properties that are not shown in the graphical format, which we will explain as is needed.

Devices: A device, such as a sensor or an actuator, is an abstraction of the physical device that exposes only its input/output ports to the external environment. For instance, the M1.BolusRequestButton and M2.EmptyReservoir-Switch in Figure 14 represent sensor-type devices that provide output ports. These output ports are connected to the input ports of the threads, M1.Bolus-RequestDetection and M2.EmptyReservoirDetection, through the port connections, D1 and D7, respectively.

Port and Port Connections: A port connection represents the relationships among *ports* that enable the directional exchange of data and events. The interactions among components can be expressed using port connections. There are nine and eleven port connections (with identifiers T1–T9 and D1–D11) in M1 and M2, respectively. There are three different types of port connections (represented by different shapes in the figure), as detailed below.

- *Data* port connections express interactions between components without queuing, and a type of data message can be defined. For instance, M2.D6 is a data port connection between the device M2.BolusRequestButton and the periodic thread M2.BolusRequest-Detection. The data on this connection may represent the status of the bolus request button (*e.g.,* the button is pressed or released);

- *Event* port connections are used to deliver events among components with queuing. For example, M1.D1 is an event port connection between the device M1.Bolus-RequestButton and the aperiodic thread M2.Bolus-Request-Detection. A button-pressed event may be delivered to the aperiodic thread via a queuing mechanism;

- *Event-Data* port connections are used for event transmissions with queuing, and each event may be associated with data. For example, M2.T7 is an event-data port connection between the two periodic threads, M2.Low-ReservoirDetection and M2.PlatformIndependentRoutine. The status of a low reservoir condition may be delivered via this port connection through a queuing mechanism.

As was for the threads, there are several port properties that are not shown in the graphical format, which we will explain as needed.

This example implies that (1) the platform-independent code requires being composed with a platform-dependent architectural option for its execution; (2) different platforms may provide different architectural options, hence a different platform-dependent code is required.

Here is the summary of the architectural differences between M1 and M2, which will result in different types of source code:

- Different thread types lead to different source code. For example, the thread M1.Bolus-RequestDetection is an aperiodic thread that is invoked only if an event is generated by the thread M1.BolusRequestButton; however, the thread M2.BolusRequestDetection is a periodic thread that is invoked every 500ms, regardless of when such an event occurs.

- Different types of port connections lead to different source code. For example, M1.T1 is an event-data port connection between two threads, M1.BolusRequestDetection and M1.PlatformIndependent-Routine. In contrast, M2.D10 is a data port connection between the M2.PumpMotorControl thread and the M2.PumpMotor device.

- Different numbers of AADL components lead to different source code. For example, M2 has a LowReservoirSwitch (device), LowReservoirDetection (thread), D8 (data port connection), and T7 (event-data port connection) that do not appear in M1. The more information is given in the AADL model, the more source code needs to be

generated.

5.5. Platform-Dependent Code Snippet Repositories

This section explains how to capture the platform-dependent programming interfaces using code snippet repositories.

Intuitively speaking, the code generation algorithm works as follows: to generate code for a particular platform, it first finds an AADL component (in the AADL model of the platform capability) to be generated into source code (*e.g.,* the periodic thread of M1.AlarmControl or the event-data port connection M2.T9 in Figure 14). Next, it looks up appropriate code snippets in the corresponding code snippet repository that can implement the chosen component. Finally, it generates the source code for the platform based on these snippets.

To automate the code generation process, it is necessary to construct a mapping between AADL components and code snippet repositories. Our approach is to construct code snippet repositories that are categorized according to the functions to implement the AADL components. Consequently, each category can be filled with different code snippets that are written using programming interfaces compatible with each platform. The code generation algorithm uses this categorization to find mappings between the AADL models and the code snippets.

Table 15 shows the categorization of code snippet repositories. The first column provides the code generation algorithm with the information for checking whether a target platform has the programming support to implement AADL components. The second column provides a more detailed level of categorization that guides the code snippets of each programming support to be written in a particular format. For example, consider the periodic thread component M2.EmptyReservoirDetection in Figure 14. To generate the code that is mapped to this periodic thread component, the code generation algorithm refers to the periodic thread category in the programming support column in Table 15 to check whether the code snippets of each subcategory of the programming support exist, and if so, it uses the

Table 5: Categorization of code snippets

Programming support	**Code snippet category**	
Dispatch mechanism	Declaration Initialization Dispatch invocation function	
Periodic thread	Declaration Initialization Thread callback function	
Aperiodic thread	Declaration Initialization Thread callback function	
Device-to-Thread port connection	Data port	Declaration Initialization Get primitives
	Event and Event Data port	Declaration Initialization Interrupt callback function
Thread-to-Device port connection	Declaration Initialization Set primitives	
Thread-to-Thread port connection	Data port	Declaration Initialization (shared variables) Read primitives Write primitives
	Event and Event Data port	Declaration Initialization (FIFO queues) Read primitives Write primitives

corresponding code snippets to generate the code.

5.5.1. Case Study of Code Snippet Repositories: FreeRTOS vs. bare platform

We demonstrate the applicability of the categorization of code snippet repositories in Table 15 using a case study of two different platforms, denoted by *FreeRTOS* and *BarePlatform*, which have different programming interfaces. The *FreeRTOS* platform runs the FreeRTOS operating system, which supports a preemptive scheduler with which programmers implement periodic/aperiodic threads. The *BarePlatform* platform is a platform that does not run any operating system; therefore, one needs to implement a dispatch mechanism that can invoke periodic/aperiodic threads. We explain each category of the categorization shown in Table 15 using example code snippets implemented on these two platforms.

Dispatch mechanism: A programming interface should provide a dispatch mechanism, *i.e.,* periodic or aperiodic, for threads to be scheduled. Some platforms may already have such a dispatch mechanism implemented; for example, *FreeRTOS* provides the API vTaskStartScheduler(), and it is sufficient to call this function to start the dispatch mechanism. However, other platforms (*e.g.,BarePlatform*) may not have a dispatch mechanism, in which case the platforms should add code snippets that implement dispatch mechanisms following the Dispatch mechanism category in Table 15.

Listing 5.2: Code snippets of Dispatch mechanism on the *BarePlatform*

```
1  //Declaration
2  const int Dispatch_Invocation_Interval = 10;
3
4  //Initialization
5  void init_dispatch( void ){
6   hardware_timer_init(Dispatch_Invocation_Interval, cbDispatchInvocation);
7  }
8
9  //Dispatch invocation
10 void cbDispatchInvocation (void* pvParameters){
11  Disable_interrupt;
12   Update_Dispatch_Flag();
```

```
13    Dispatch_Aperiodic_Threads();
14    Dispatch_Periodic_Threads();
15   Enable_interrupt;
16  }
```

Listing 5.2 gives an example code snippet of the dispatch mechanism implemented on *Bare-Platform*, which belongs to the code snippet category of Dispatch mechanism in Table 15. The operation of the dispatch mechanism is as follows. In Lines 4–7, the code snippet initializes a hardware timer of the microprocessor with a fixed millisecond-basis period (Dispatch_Invocation_Interval) defined in Line 2 of the *Declaration* part, and a pointer to the callback function (cbDispatchInvocation) in the *Initialization* part. This enables cbDispatchInvocation, implemented in the *Dispatch invocation* part, to be called every period (*i.e.,* 10 ms in this example). Upon being activated, the invocation function checks the list of periodic and aperiodic threads[1] that need to be invoked at the current invocation period; this is implied in Update_Dispatch_Flag() in Line 12. Then, the invocation function executes all checked threads in Lines 13–14, and it completes the current dispatch round.

Periodic/Aperiodic thread implementation: A programming interface should provide a mechanism to implement periodic and aperiodic threads that can be scheduled by the dispatch mechanism explained above. Code snippets that implement such threads vary across platforms that expose different programming interfaces.

Listing 5.3: Parametrized code snippet of Aperiodic thread in FreeRTOS

```
1  //Declaration part
2  const portBASE_TYPE prioirty#F_tid(ext)#=#F_priority(ext)#;
3  const portBASE_TYPE stacksize#F_tid(ext)#=#F_stack(ext)#;
4
5  //Initialization part
6  void init_#F_tid(ext)#( void ){
7   TaskCreate(cb#F_tid(ext)#,''#F_tid(ext)#'',stacksize#F_tid(ext)#, NULL,prioirty#F_tid(ext)#,NULL);
8  }
9
10 //Thread callback function part
```

[1]This information is implemented in the declaration part, but not shown here for clarity

```
11  void cb#F_tid(ext)# (void* pvParameters){
12   for(;;){
13    #F_input(ext)# //Wait for the input
14    //Compute
15    #F_output(ext)# //Write output
16   }
17  }
```

Listings 5.3 and 5.4 show the code snippets for aperiodic threads that can be executed on *FreeRTOS* and *BarePlatform*, respectively. Special functions (*e.g.,* F_{tid}(ext)) enclosed with two sharp signs (#) are used to specify parametrized code snippets that need to be replaced with some other codes, which we will explain in Subsection 5.5.2. Both code snippets implement aperiodic threads on each platform; however, they are different in the following sense.

Listing 5.4: Parametrized code snippet of Aperiodic thread in BarePlatform

```
1  //Declaration part
2  const int prioirty#F_tid(ext)#=#F_priority(ext)#;
3
4  //Initialization part
5  void aperiodic_#F_tid(ext)#_init( void ){
6   register_aTask(cb#F_tid(ext)#, prioirty#F_tid(ext)#);
7  }
8
9  //Thread callback function part
10 void cb#F_tid(ext)# (void* pvParameters){
11    #F_input(ext)# //Read input
12    //Compute
13    #F_output(ext)# //Write output
14 }
```

The two programming interfaces require different information to initialize aperiodic threads. For example, the *FreeRTOS* code snippet in Listing 5.3 specifies the stack size (Line 3) of the maximum amount of memory that a thread can occupy at run time. This parameter is passed to the *TaskCreate* API (Line 7), so that the OS kernel triggers exceptions in case of stack overflows at run time. On the contrary, *BarePlatform* does not require explicit stack

sizes of threads, since the platform is incapable of specifying stack sizes and capturing stack overflows.

The two programming interfaces provide different ways of implementing interactions between their thread callback functions and dispatch mechanisms. For example, the *FreeRTOS* code snippet in Listing 5.3 implements the infinite for-loop (Line 12–16), in which blocking functions are used to interact with the scheduler (the function $\#F_{input}(\text{ext})\#$ in Line 13 will be replaced with such blocking functions in our framework). However, the *BarePlatform* code snippet in Listing 5.4 does not have such a for-loop: its dispatch mechanism is invoked periodically, and the called thread callback function will be returned in the current dispatch invocation without looping.

The two programming interfaces provide different names of APIs to perform similar functions. For instance, to register the thread information to the kernel, the *FreeRTOS* uses the *TaskCreate* API (Line 7), whereas the *BarePlatform* uses the *register_aTask* API (Line 6).

Note that the code snippet category of aperiodic threads in Table 15 is filled with different code snippets of Listing 5.3 and Listing 5.4 for *FreeRTOS* and *BarePlatform*, respectively.

Port connection implementation: Each programming interface should provide a mechanism to implement the port connections that enable thread and device components to interact with one another (*e.g.,* D1–D11 and T1–T7 in Figure 14). Different types of port connections (e.g, data or event-data ports) lead to different code snippets. Moreover, different instances of a port connection can be implemented by different code snippets, depending on whether they cut through the software system scope (that need to be generated into source code) or not. For example, M1 and M2 in Figure 14 contain system components (big rounded boxes in the middle) that represent the scope of the software system; therefore, thread components within the scope are subject to code generation. In contrast, device components outside the scope are not subject to code generation. Finally, the port connections of M1.{D1-D5} and M2.{D6-D11} cut through the scope of software system.

In our code snippet repository, such port connections are separately categorized from port connections that have both of their source and destination components residing inside the scope (*e.g.,* M1.{T1-T4} and M2.{T5-T9})

Based on the above observation, the code snippet repository in Table 15 distinguishes three different categories of port connections:

- The *Device-to-Thread* port connection category stores code snippets that are used to implement directional port connections from *device* to *thread* components. This port connection is typically used by thread components to read sensor values. For example, the implementations of M1.{D1, D2}, M2.{D6, D7, D8} use code snippets from this category. This category includes code snippets that process *input only* (*i.e.,* it does not have a code snippet that writes outputs). Inputs can be read from devices in two different ways. First, a thread component may read values from a device component through data ports (*e.g.,* M1.D2 and M2.{D6, D7, D8}); in this case, the code snippet provides *Get* primitives that can be called by thread components to retrieve data from device components. Second, a thread component may read values from device components through event or event-data ports (*e.g.,* M1.D1); in this case, the code snippet provides the *Interrupt* callback function that is called when events from device components occur.

- The *Thread-to-Device* port connection category stores code snippets that are used to implement directional port connections from *thread* to *device* components. This port connection is typically used by thread components to actuate actuators by writing values. For example, the implementations of M1.{D3, D4, D5}, M2.{D9, D10, D11} use code snippets from this category. This category includes code snippets that process *output only* (*i.e.,* it does not have a code snippet that reads inputs). Thread components can write outputs to devices by calling the *Set* function.

- The *Thread-to-Thread* port connection category stores code snippets that are used

to implement port connections between two thread components. For example, the implementations of T1-T9 in M1 and M2 use code snippets from this category. Unlike the above port connection types, this category includes code snippets that process *both* inputs and outputs. Two threads can communicate with each other through either data port or event-data port. In the former case, a shared variable is used with the associated read/write primitives; in the latter, a FIFO queue is used with the associated read/write primitives.

We next give an example of the code snippet for thread-to-thread port connection implementation on *FreeRTOS*.

Listing 5.5 shows the code snippet of (event-data port) thread-to-thread port connection of *FreeRTOS*. We note that this code snippet is written following the four categories related to the (event-data port) thread-to-thread port connection shown in Table 15.

Declaration: Lines 1–6 implement the declaration part, which lists the necessary variables to implement the port connection using a FIFO queue: the handler of the FIFO queue (Line 2), the queue size (Line 3), the dequeue policy (Line 4), the blocking mode (Line 5), and the overflow handling policy (Line 6).

Initialization: Lines 8–11 implement the initialization function that creates a FIFO queue using FreeRTOS API, *xQueueCreate*; the variables in the declaration part is passed to the API call.

Read primitive: Lines 13–28 implement the read primitives that can be used by threads to read items from the FIFO queue. Note that two different read primitives are implemented, and which one is generated depends on the dequeue policy that is specified in Line 4. This is intended to capture the AADL property *Dequeue_Protocol*, whose value can be either *OneItem* (read a single item from the queue) or *AllItems* (read all items from the queue).

xQueueReceive is a FreeRTOS API that dequeues items from the queue. In case the queue

is empty, the blocking mode, specified as edQBlockMode#F_{pid}(ext)# in Line 5, decides whether a caller of this API should be blocked until any item arrives, or it should be timed-blocked (*i.e.,* blocked for only a certain amount of time), or non-blocked (*i.e.,* never blocked). The *uxQueueMessagesWaiting* API in Line 24 returns the number of items in the queue, which is needed to read all the items from the queue (Lines 23–25).

Write primitive: Lines 30–49 implement the write primitive that can be called by threads to insert items to the FIFO queue. The *xQueueSend* API is used to insert an item to the FIFO queue in Line 32. Lines 34–45 show the three different ways for handling the queue overflow exception, depending on the value of the AADL property *Overflow_Handling_Protocol* (*i.e.,* DROP_OLDEST or DROP_NEWEST or ERROR). The code that handles such an overflow handling protocol is not detailed here.

Listing 5.5: Parametrized code snippet of Thread-to-Thread port connection in FreeRTOS

```
1  //Declaration part
2  static xQueueHandle edQHandle#F_pid(ext)#;
3  const portBASE_TYPE edQSize#F_pid(ext)# = #F_qsize(ext)#;
4  const portBASE_TYPE edDQPolicy#F_pid(ext)# = #F_rpolicy(ext)#;
5  const portBASE_TYPE edQBlockMode#F_pid(ext)# = #F_wmode(ext)#;
6  const portBASE_TYPE edQOverflowHandling#F_pid(ext)# = #F_wpolicy(ext)#;
7
8  //Initialization part
9  void ed#F_pid(ext)#_Init( void ){
10   edQHandle#F_pid(ext)# = xQueueCreate(edQSize#F_pid(ext)#, sizeof(#F_itemtype(ext)#));
11 }
12
13 //Read primitive
14 #if edDQPolicy#F_pid# == OneItem
15 portBASE_TYPE Read_#F_pid(ext)# (#F_itemtype(ext)#* buf){
16   xQueueReceive(edQHandle#F_pid(ext)#, buf, edQBlockMode#F_pid(ext)#);
17   return TRUE;
18 }
19 #elif edDQPolicy#F_pid# == AllItem
20 portBASE_TYPE Read_#F_pid# (#F_itemtype(ext)#* buf){
21   portBASE_TYPE item_count = 0;
22   item_count = uxQueueMessagesWaiting(edQHandle#F_pid(ext)# ) );
```

```
23   for(int i = 0 ; i < item_count ; i++){
24     xQueueReceive(edQHandle#F_pid(ext)#, buf + i*sizeof(#F_itemtype(ext)#), edQBlockMode#F_pid(ext)#);
25   }
26   return item_count;
27 }
28 #endif
29
30 //Write primitive
31 portBASE_TYPE Write_#F_pid(ext)# (#F_itemtype(ext)#* buf){
32   portBASE_TYPE result = xQueueSend(edQHandle#F_pid(ext)#, buf, edQBlockMode#F_pid(ext)# );
33   if(result == FALSE){
34     switch(edQOverflowHandling#F_pid(ext)#){
35     case DROP_OLDEST:
36         //Drop oldest and enqueue
37         break;
38     case DROP_NEWEST:
39         //Drop newest and enqueue
40         break;
41     case ERROR:
42         //Raise an exception
43         break;
44     default:
45     }
46     return FALSE;
47   }
48   return TRUE;
49 }
```

5.5.2. Parametrized Code Snippets

As described in Subsection 5.5.1, the AADL models and the code snippet repositories are created independently of one another. Hence, it is necessary to inform the code generation algorithm of the scope of the code that should be related to the information of the AADL model of a platform. This is done via parametrized code snippets, which specify the placeholders that later can be filled by the code generation algorithm based on the AADL model. Therefore, our framework separates concerns between how the code snippets are written and how they are actually used in a certain architectural context. For example, the code

snippet for aperiodic threads in Listing 5.3 is written independently of how it is used to implement M1.BolusRequestDetection or M2.PlatformIndependentRoutine in Figure 14.

Parametrized code snippets can be written using functions that take a set of input parameters and return a piece of code. Such functions enable a parametrized code snippet to be instantiated into several different pieces of code; for instance, the code snippets in Listing 5.3, 5.4, 5.5 use the parametrized code snippets that are enclosed with two sharp signs(#). These functions also specify the rules for instantiating the code using external information that is passed through *ext*. For example, F_{tid}(ext) in Line 11 of Listing 5.3 specifies how a thread identification should be represented in the aperiodic thread callback function using external information *ext*. Suppose the string "BolusRequest" is passed as *ext*, then one may define a function F_{tid} that converts the string into "BolusReq" and returns it as a piece of code. Then, the code snippet after resolving the parameter of Line 11 becomes "*void cbBolusReq (void* pvParameters)*."

As another example, one may define a function, F_{period}(ext), to specify the rule to convert *ext* into a value that represents the period of a periodic thread in the code snippet. In Figure 14, the period property of M2.EmptyReservoirDetection is represented as a string of "500 ms". This string is passed as a parameter of *ext*. The internal of F_{period} converts "500 ms" into some appropriate values. Suppose, we want to get the code in Line 2 of Listing 5.1. Then, the function simply removes "ms" from the string, resulting in the value 500. However, the returned value is not always necessarily 500 to represent "500 ms" of period in the code snippet. Because different programming interfaces may require to scale the numeric value differently to represent "500 ms" of period. For example, the dispatch mechanism of BarePlatform in Listing 5.2 invokes the dispatch mechanism every 10 ms as specified in line 2. The code snippet for periodic threads (not shown here) requires to specify periods of thread relative to the dispatch invocation interval. Then, F_{period}(ext) should convert "500 ms" into 50 instead of 500. This example shows different platforms interpret the same information coming from the AADL model in a different way. Therefore, having parametrized

code snippets provides flexibility to deal with such heterogeneous programming interfaces.

In Section 5.6, we explain the proposed code generation algorithm that co-relates an AADL model (explained in Section 5.4) and a code snippet repository (explained in this section), in order to produce the platform-dependent code for different platforms.

5.6. Platform-Dependent Code Generation Process

The constructed AADL model and the code snippet repository of a particular platform are used together with the platform-independent code as inputs to the code generation algorithm to generate the platform-dependent code. In this section, we explain how the algorithm processes and correlates the AADL model and code snippet repository to produce the platform-dependent code.

The AADL model is expressed in textual form. Listing 5.6 shows a textual representation of M2.EmptyReservoir-Detection in Figure 14. The scope starting with **thread** (Line 2) and ending with **end** (Line 15) characterizes M2.Empty-ReservoirDetection. Lines 4–9 characterize the input and output ports that are associated with the thread. Lines 11–14 specify the properties that characterize the periodic thread.

Listing 5.6: Textual representation of the periodic thread component M2.EmptyReservoirDetection in Figure 14

```
1  -- Define thread type of EmptyReservoirDetection
2  thread thd_empty_rsv
3    features
4     D7: in data port;
5     T6: out event data port{
6      Overflow_Handling_Protocol => Error;
7      Dequeue_Protocol => AllItems;
8      Queue_Size => 5;
9     };
10   properties
11    Dispatch_Protocol => Periodic;
12    Period => 500 Ms;
13    SEI::Priority => 2;
```

```
14      Source_Stack_Size => 500 B;
15  end thd_empty_rsv
```

We note that the AADL standard provides a rich set of properties to characterize AADL components. In addition, one may also define custom properties associated with AADL components. As a result, it is difficult to design a code generation algorithm that takes into account all possible properties. Instead, we provide a finite set of AADL properties that are sufficient for our case study; we expect that extensions of this property set can easily be done as are needed.

Table 6 summarizes the list of AADL properties of thread and port connection components that are used by our code generation algorithm in order to find appropriate code snippets in the code snippet repository.

Table 6: Information extracted from the AADL model

AADL component	**Property**	**Example value**
Thread	Thread ID	EmptyReservoirDetection
	Thread type	Periodic/Aperiodc
	Thread period	100ms or 10sec
	Thread priority	3
	Source stack size	500 B
	Input port connection IDs	D1, D2, T1
	Output port connection IDs	D1, D2, T1
Port connection	Connection ID	D1, D2, T1
	Interaction type	Device-to-Thread, Thread-to-Thread
	Source component ID	EmptyReservoirDetection
	Destination component ID	PlatformIndependent-Routine
	Port connection type	Data, Event, Event-Data
	Queue size	5
	Read policy	Read one item, Read all item
	Write policy	Drop oldest, Drop newest, Error

Listing 5.7 gives the pseudo code of the code generation algorithm that generates platform-dependent code from an AADL model and a code snippet repository. The parameters M and C that are passed as input to the function *CodeGen* in Line 16 are the abstracted representation of the information obtained from the AADL model in Table 6 and the code

snippet repository in Table 15, respectively.

Listing 5.7: Pseudo code of the platform-dependent code generation algorithm

```
16 function CodeGen(M, C)
17 exp_scope
18   //Generating code for thread components
19   for each thread, Thread[i] ∈ M
20    if M.Thread[i].type == PERIODIC
21     SnippetHandle := C.PeriodicThreadSnippet;
22    else if M.Thread[i].type == APERIODIC
23     SnippetHandle := C.APeriodicThreadSnippet;
24    endif
25    for each parametrized function, F_k ∈ SnippetHandle
26     SnippetHandle.F_k(M.Thread[i]);
27    endfor
28    Generate(SnippetHandle);
29   endfor
30  //Generating code for port connection components
31   for each port connection, PortConn[j] ∈ M
32    if M.PortConn[j].type == Device-to-Thread
33     SnippetHandle := C.Device-to-ThreadSnippet
34    else if M.PortConn[j].type == Thread-to-Device
35     SnippetHandle := C.Thread-to-DeviceSnippet
36    else if M.PortConn[j].type == Thread-to-Thread
37     SnippetHandle := C.Thread-to-ThreadSnippet
38    endif
39    for each parametrized function, F_l ∈ SnippetHandle
40     SnippetHandle.F_l(M.Thread[i]);
41    endfor
42    Generate(SnippetHandle);
43   endfor
44 exception(No matched code snippets)
45   //Exception handling
46 exception(No matched parameters)
47   //Exception handling
48 endexp_scope
49 endfunction
```

The algorithm generates the platform-dependent code for thread components of M in Lines

19–29. The for-loop in these lines finds a match between thread components in M and the code snippets in C. Using the *dispatch protocol* property of M, the algorithm finds different code snippets from C. After such a match is found, the algorithm resolves the parametrized code snippets (explained in Subsection 5.5.2) of the matched code snippet in Lines 25–27. The properties of the matched thread in M are passed as a parameter to the function of the parametrized snippets. This function converts the parameter into a piece of code using the rules specified in the function. After resolving all parametrized code snippets, the algorithm generates the platform-dependent code of the thread components in M, which is implied in Line 28; here, *Generate* is a simple function that copies and pastes the code snippet into some output files. The code generation for port connection components (Lines 31–43) is similar to the generation for thread components, except that the generation algorithm uses the *interaction type* property of the port connection to find a match between M and C.

We note that the code generation algorithm deals with two types of exceptions implied in the exception scope (Lines 17–48). Specifically, the *No matched code snippets* exception is raised when the algorithm cannot find a matched code snippet in C to generate a component in M. For example, the code snippet repository may not contain code snippets that implement periodic threads, but the AADL model has periodic thread components to be generated into code. One should handle such an exception appropriately, *e.g.,* by registering a code snippet to C that implements periodic threads. The *No matched parameters* exception is raised when the algorithm cannot find a match of a conversion function in any parametrized code snippets. There are two cases to trigger this exception: (1) M does not have the information that is needed for C to generate code; for example, C requires the blocking mode on the FIFO queue ($\#F_{wmode}(\text{ext})\#$) to be specified, so as to implement read/write primitives for port connection components in Line 5 of Listing 5.5, but M does not have such a property in Table 6, and (2) C does not have the capability to implement the properties of M; for example, M has a property of source stack size, but C does not have the code snippet that contains this information. Both exceptions should be handled by users (*e.g.,* by adding missing information in M or by using the default value of C).

Composition with Platform-Independent Code: As explained in Section 5.3, the platform-independent code (*e.g.,* Listing 4.3 4.4 4.5 4.6) generated from the platform-independent model (*e.g.,* Figure 11) should be composed with the platform-dependent code by (1) interfacing platform-independent I/O operations, and (2) interfacing invocation mechanisms of the platform-independent execution logic. We explain several places that need to be considered in our framework for the composition. Consider the example platform-independent model in Figure 11. The model can be systematically transformed into source code that repeats the following sequential operations, which is also used in several code generators [7][14]:

1. *Read* inputs from some variables, PI_{input}, that is updated by some external piece of code

2. *Compute* the next state using transition tables (encoded as switch-case statements or array structures) based on PI_{input}

3. *Write* outputs to some variables, PI_{output}, that are read by some external piece of code

The AADL model utilizes port connections to express input/output relationship among different AADL components such as threads and devices. On the other hand, the code snippet repository in Table 15 contains code snippets of the port connection that implement read and write primitives using the target programming interfaces. An example of read/write primitives can be found in Listing 5.5. In order to interface I/O operations, one needs to resolve (1) read dependencies between PI_{input} and the read primitives, and (2) write dependencies between PI_{output} and the write primitives. In Listing 5.4, $\#F_{input}$(ext)# (line 11) and $\#F_{output}$ (ext)# (line 13) specify such placeholders in the form of parametrized code snippets (*i.e.,* one should provide the implementations of such functions to resolve input/output dependencies).

In addition, the semantics of *repeated* execution of (1), (2), (3) in the platform-independent

code needs to be mapped to the platform-dependent code as well. A thread component of the AADL model is an abstraction of sequential computation in which input is read from input port connections and output is written to output port connections. Therefore, one may implement such a mapping using either periodic or aperiodic thread components. In case of periodic threads, the execution of (1), (2) and (3) can be performed periodically. In case of aperiodic threads, the execution of (1), (2) and (3) can be performed only if any input is available from one of the input ports. Either case equally implements the platform-independent model by executing (1) (2) (3) repeatedly.

5.7. Summary of the PD-Phase

In the PD-Phase, we showed how a platform-dependent code can be systematically produced in a way that it can be composed with the platform-independent code on a particular platform.

Our approach is to selectively generate platform-dependent code by characterizing architectural timing aspects and programming interfaces of a platform. An AADL model is used as a means to capture platform-dependent architectural timing aspects, and a code snippet repository is used as a means to capture programming interfaces of a platform. The specified platform-dependent timing information is used to systematically generate code to interface I/O operations and invocation mechanisms required to operate the platform-independent code on a particular platform.

In comparison to the PI-Phase, where the input and output timing occurring at the *mc*-boundary is modeled, the PD-Phase abstracts internal platform delays required for processing input and output events, such as polling or interrupt-based interactions with sensors or thread scheduling mechanisms. Such platform processing delays vary across a range of platforms as each platform may adopt different architectural options that can be composed with the same platform-independent code. This implies that the observable delays of input and output events at the *mc*-boundary may also vary depending on what platforms will be

composed with the platform-independent code.

Thus, even though the platform-independent model has been verified against timing requirements, it cannot guarantee that the composition of the platform-independent code and the platform-dependent code also meets the same timing requirements at the implementation level. Therefore, it is necessary to analyze whether the composition is actually performed in a way that the implementation meets the timing requirements that have been verified in the platform-independent model. In Chapter 6, we explain two different approaches (*i.e.,* testing and formal verification approaches) to check the correctness of such a composition in terms of timing requirement conformance; in case the composition does not conform to the timing requirements, in Chapter 7, we explain how to optimize the platform-independent code in a way that the implementation meets the timing constraints.

CHAPTER 6 : Integration Phase (Part 1)

6.1. The Problem Statements and Challenges of the ITG-Phase

In order to gain benefits from the separation of concerns, the platform-independent model is created and verified independently from any particular platform-specific timing aspects. The platform-independent code generated from the model is thus an incomplete version of the final implemented system; it can only be completed when the platform-independent code is integrated with a particular platform-dependent code. Therefore, in order to claim the timing requirement conformance of the final implemented system, it is necessary to show that the composition of the two codes have been performed in a way that conforms to the timing requirements. The main objective of the ITG Phase is to establish a systematic framework that enables one to check the correctness of such a composition in terms of the timing requirement conformance. We are particularly concerned about (1) how the timed behavior of an implemented system is changed after the composition from that of the platform-independent model, and (2) how such changes impact to the requirement conformance. Here is the challenge in checking the correctness of such a composition:

Challenge: The relationship of timed behavior between the two codes has not been expressed in their respective development process, but this information is necessary to check the correctness of the composition.

In order to check a composition of different aspects of a system, it is necessary to precisely express their relationship. However, the platform-independent code has been developed from the model that describes the timed behavior of a system using the behavioral semantics, whereas the platform-dependent code has been developed from the model with the architectural semantics. For example, the platform-independent model uses a timed transition system where a system state changes upon input and output synchronization, time-invariants and guard conditions to express timed behavior of a system. On the other hand, the platform-dependent model expresses the platform's timed behavior using interac-

tions among different architectural components (*e.g.,* threads, buffers) and their associated properties (*e.g.,* periodic invocation, buffer size). In order to check the correctness of the composition to reason about the timed behavior of a system as a whole, it is necessary to understand how the two levels of abstraction is related to each other.

6.2. The Approach Overview of the ITG-Phase

We propose two processes conducted in the ITG phase that enable the implemented system to be checked for timing requirement conformance - (1) the platform-specific timing verification process (formal verification approach), (2) the platform-specific timing testing process (testing approach). The two processes are performed under different assumptions about the platform-specific timing information. Timing verification is performed when the timing aspects of the platform-dependent code are known, and can be formally abstracted along with the platform-independent model. On the other hand, timing testing is performed when the platform-independent code is composed with the platform-dependent code, though, the timing aspects of the platform-dependent code are not known. Two processes can be used together or independently depending on the availability of the platform-specific timing information. Here are the overviews for each approach:

Formal verification approach: We propose the platform-dependent timing verification framework as illustrated in Fig. 15. In order to build an implemented system, the platform-independent code is assumed to be composed with the platform-dependent code that implements a particular architectural option. We propose the implementation scheme category that separately categorizes such possible architectural options that a platform may choose, based on (1) how a platform-dependent code interacts with its real environment, and (2) how a platform interacts with the platform-independent code. Each platform may choose its own implementation scheme that matches with the architectural option implemented to support the execution of the platform-independent code. Then, we propose a transformation algorithm that systematically transforms a platform-independent model (*PIM*) into a platform-specific model (*PSM*) using UPPAAL semantics. This transformation algorithm

composes the platform-independent model with the chosen implementation scheme in a modular way; that is, when composed, the internal structure of the platform-independent model remains unchanged. The transformed platform-specific model is now a new abstraction of the timed behavior of a particular implemented system. Therefore, the same timing requirements verified in the platform-independent model can still be used to verify the platform-specific model in order to check the timing requirement conformance of the implemented system through model-checking. The details of this approach is given from Section 6.3 to Section 6.8.

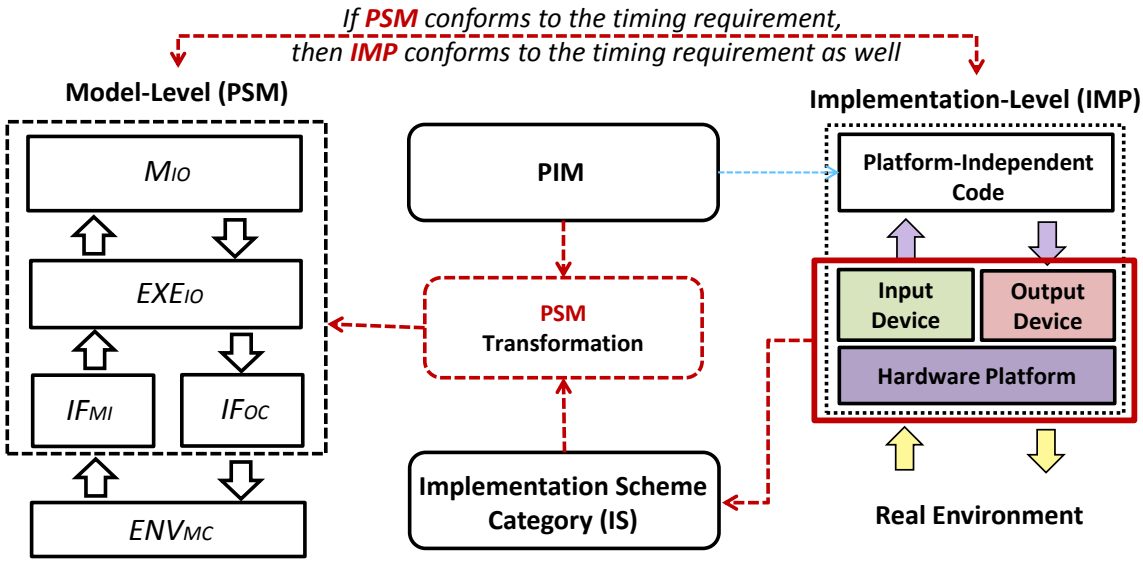


Figure 15: The Formal Verification Approach in the Integration Phase

Testing approach: Testing is a widely used technique to show the requirement conformance when a part of the internal of an implemented system is not known explicitly. We propose a layered testing approach that enables the implemented system to be checked for timing requirement conformance in a systematic way as illustrated in Fig. 16. Under the assumption that the timing aspects of the platform-dependent code is unknown, this approach defines the testing interface using Parnas' four-variables (m, i, o, c) that can separately measure the delays occurring at the platform-independent and the platform-dependent code respectively. This interface is used by two different types of testing -

R-testing and *M-testing*. *R-testing* checks whether the implemented system conforms to the timing requirements. In this testing, test cases that consists of a pair of *m-stimulus* and *c-response* with the maximum allowable delay are generated using the testing interface at the *mc*-boundary. If the testing result shows that a timing requirement is violated in the implemented system, then, M-testing is followed. In *M-testing*, testing points are generated at both the *mc*-boundary and *io*-boundary to measure several delay segments, such as input delay or output delay, in order to quantify timing deviation between the platform-independent model and its implemented system. The details of this approach is given from Section 6.10 to Section 6.12.

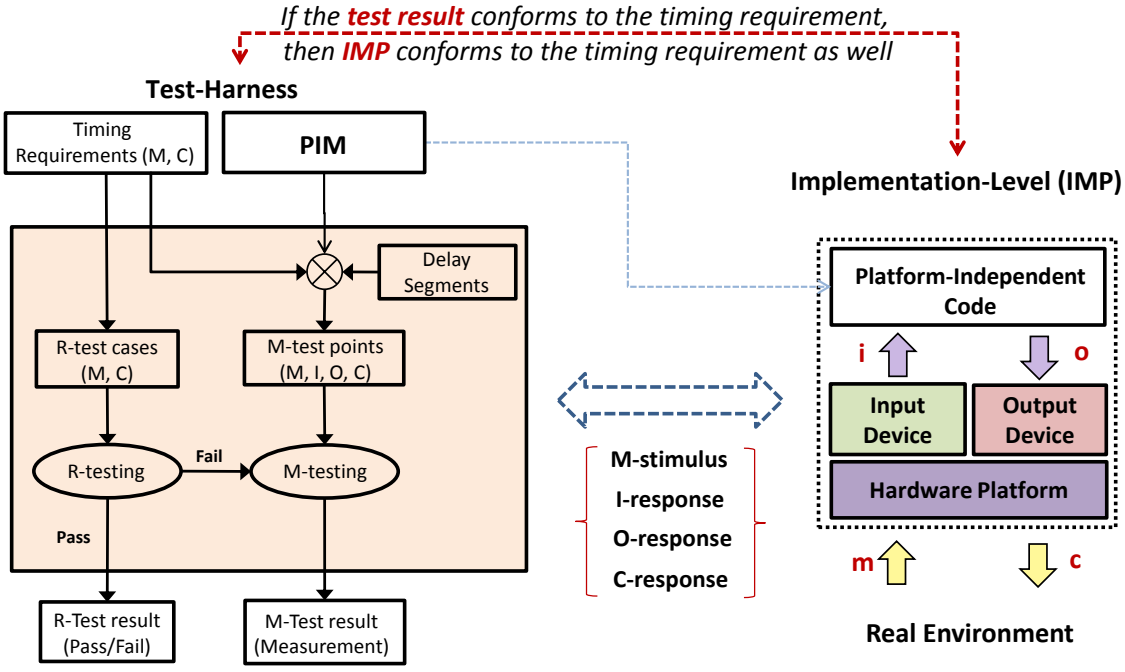


Figure 16: The Testing Approach in the Integration Phase

6.3. The Problem Statement (*PSM* Verification)

6.3.1. Motivating Example

We first give a running example motivated from the GPCA safety requirements and the GPCA model explained in Section 3.2 in order to explain the platform-dependent verifica-

tion process.

Consider the following informal timing requirement that is obtained from the GPCA safety requirements [6]:

- (REQ1) *"When a patient requests a bolus, a bolus infusion should start within 500ms."*[1]

In the PI-Phase, a platform-independent code is developed by going through the two processes:

Platform-independent modeling and verification: the timed behavior of a system is first formalized using timed modeling languages. Fig. 17 shows a UPPAAL model that abstracts the timed behavior of the infusion pump system and its environment. This model is a parallel composition of two automata, *M* and *ENV*. *M* models the system using a clock variable (x), input synchronizations (*m-BolusReq* and *m-EmptySyringe*), and output synchronizations (*c-StartInfusion* and *c-StopInfusion* and *c-Alarm*). *ENV* models the environment using a clock variable (*env-x*) and complementary synchronizations with that of *M*.

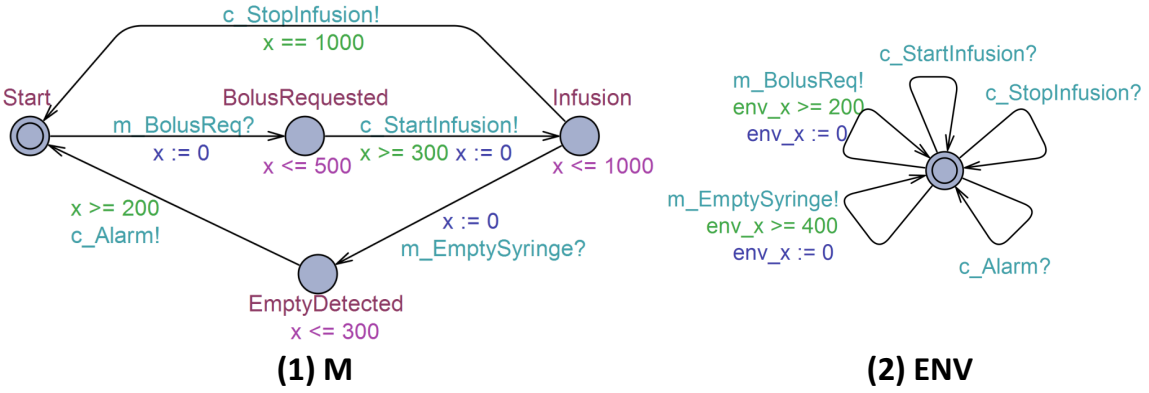


Figure 17: The example PIM

The informal semantics of the model is as follows: *ENV* can perform synchronizations, *m-BolusReq* and *m-EmptySyringe*, with its respective minimum inter-arrival times (200 and

[1]Note that the specific timing parameter (*500ms*) is added to the original requirement to explain our work.

400 time-units). In case the *m-BolusReq* synchronization is enabled while M is in the *Start* location, M makes a transition to *BolusRequested* location and then performs the output synchronization *c-StartInfusion* in between 300 and 500 time-units; similarly, M makes a transition either to the *Start* or *EmptyDetected* location depending on the timing guards and the available synchronization.

We can show that REQ1 is satisfied in this model (by describing REQ1 as a logic formula, which states that the maximum delay of two successive synchronizations – *m-BolusReq* followed by *c-StartInfusion* – does not exceed 500 time-units, and verifying it using model checking).

Platform-independent code generation: a code generation process automatically generates the platform-independent code (*e.g.,* C code) from the verified model (*e.g.,* the model in Fig. 17). Listing 6.1 gives the pseudo code of the generated code that is typically produced by existing code generators [14] [40].

Listing 6.1: Abstraction of Code(PIM)

```
1 Code(PIM){
2   loop forever
3     await();
4     local_i:=readInput(i);
5     Compute-InputTrans(local_i, getClock());
6     local_o:=Compute-OutputTrans(getClock());
7     writeOutput (local_o);
8   end loop
9 }
```

The pseudo code interacts with a *platform* via several platform-specific primitives (which must be implemented by any platform that executes the generated code) as follows: First, the code waits for a trigger (*await* in Line 3) from a platform so that the subsequent blocks of the code can be executed. Once being triggered, it reads inputs (*readInput* in Line 4) from the platform, and uses the obtained values to make a decision for input transitions

(*Compute-InputTrans* in Line 5) along with the current clock time (which can be read from a platform through *getClock*). Subsequently, it performs a transition decision for outputs (*Compute-OutputTrans* in Line 6) along with the current clock time. Finally, it writes the produced output to the platform (*writeOutput* in Line 7), and repeats this series of actions.

6.3.2. Problem Statement

The example *platform-independent model* (*PIM*) in Fig. 17, which is one of the artifacts produced in the PI-Phase (Chapter 4), does not explicitly capture the platform-specific timing information. For example, it only describes that M is synchronized with ENV over the *m-BolusReq* channel for the input interaction, but it does not express information about how such synchronization is to be implemented on a particular platform. On the other hand, the PD-Phase (Chapter 5) aims at expressing such platform-dependent information from which the platform-dependent code can be generated to support the execution of the platform-independent code. This information includes, *e.g.,* how a platform reads a bolus request input from a sensor (*e.g.,* interrupt or sampling), how a platform processes the sensor readings and delivers them to the software (*e.g.,* shared variable or buffering), and how tasks are scheduled by a platform (*e.g.,* periodic or aperiodic invocation) to read the input values and to take transitions.

The above separation of timing concerns is necessary in the model-based implementation, because these platform-specific timing details are not typically available or less concerned in the early modeling/verification stage, and the generated code is also expected to be re-used on a range of target platforms through platform-specific customizations. However, it also leads to timing gaps between a model and its implementation. In other words, the testing result of the timing requirements on the final implementation may not be consistent with the timing verification result of the platform-independent model, mainly because of the missing platform-specific timing details in the early modeling stage.

To describe the problem more precisely, we introduce some notations used in this chap-

ter. *Code*(*PIM*) denotes the *platform-independent code* generated from a *PIM*. This code needs to be composed with platform-specific primitives (*e.g.,* read/write API) to realize the platform-specific interactions. *IS* denotes an *implementation scheme* that is used for such a composition. The implemented system is denoted by IMP, which indicates that the resulting implementation is the platform-independent code executed with the support of an implementation scheme *IS*. Finally, $\mathcal{P}(\Delta_{mc})$ denotes a timing requirement that the delay between an input m and an output c must be within Δ_{mc} time units (*e.g.,* the delay between the bolus request and the start infusion must be within *500ms*).

As explained in the infusion pump example, the timing information of a chosen *IS* is not explicitly modeled in the *PIM* and hence, the following claim may *not* always hold:

$$\mathit{PIM} \models \mathcal{P}(\Delta_{mc}) \textbf{ implies } \mathsf{IMP} \models \mathcal{P}(\Delta_{mc}).$$

To better describe the timed behavior of the implementation, we need a platform-specific version of the *PIM* that captures the timed behavior of the *IS*, such that if this platform-specific model (*PSM*) is verified to meet a timing requirement, then its implementation also satisfies the requirement. In addition, as was discussed earlier, the *IS* may introduce an additional delay to the response time between an input m and an output c, which can lead to a violation of the timing requirement $\mathcal{P}(\Delta_{mc})$ in the implementation. To quantify how close the implementation is from satisfying $\mathcal{P}(\Delta_{mc})$, we would like to compute a new delay bound Δ'_{mc} for which $\mathcal{P}(\Delta'_{mc})$ holds in the implementation. In other words, we need to derive a platform-specific model (*PSM*) and a bound Δ'_{mc} such that

$$\mathit{PSM} \models \mathcal{P}(\Delta'_{mc}) \textbf{ implies } \mathsf{IMP} \models \mathcal{P}(\Delta'_{mc}).$$

To this end, our goals are (1) to identify the necessary information to obtain such a *PSM*, (2) to develop a method for systematically transforming a *PIM* into a *PSM* based on the identified information, and (3) to compute the bound Δ'_{mc}.

In the next section, we introduce a platform-dependent verification framework that achieves these goals.

We assume that a platform consists of several building blocks to support the execution of *Code*(*PIM*), including the *Input-Device* that processes inputs generated from the environment, the *Output-Device* that processes outputs generated from *Code*(*PIM*), and the *Code-Execution* that invokes *Code*(*PIM*) to perform transitions based on the environmental inputs and the clocks' values. Fig. 18-(a) shows the block diagram that illustrates these blocks in an implemented system.

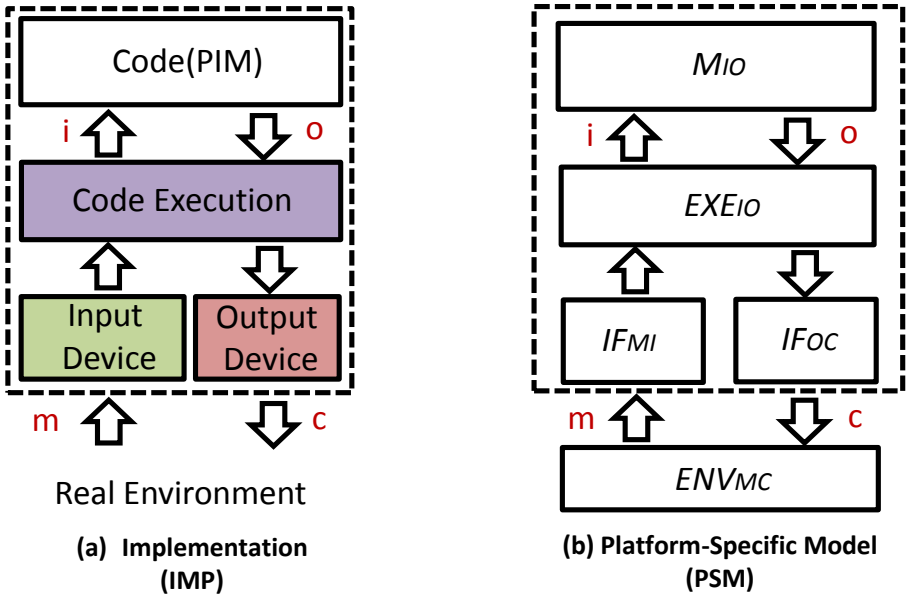


Figure 18: The mapping between (a) the implemented system and (b) its platform-specific model

To model the platform-specific information, we propose a general category of implementation schemes that lists possible mechanisms to implement each interaction of the platform with the environment and with the *Code*(*PIM*). A platform can select a particular combination of mechanisms from the category as its implementation scheme. Based on a chosen implementation scheme *IS*, we can systematically transform a *PIM* into a *PSM*. The transformed *PSM* is the parallel composition of the UPPAAL models shown in Fig. 18-(b) (*i.e.*, $M_{IO} \| IF_{MI} \| IF_{OC} \| EXE_{IO} \| ENV_{MC}$). Here, IF_{MI} and IF_{OC} model the platform-specific input and output processing mechanisms for interacting with the environment (modeled

as ENV_{MC}). EXE_{IO} models the platform-specific invocation mechanism to schedule the platform-independent code so as to receive (deliver) the code's inputs (outputs) from (to) a platform. Finally, M_{IO} models the timed behavior of the platform-independent code executed with the support of the implementation scheme IS. Each of these models is matched to its respective building blocks shown in Fig. 18-(a).

Our transformation algorithm is modular and preserves the original structure of the PIM. The algorithm ensures that the resulting PSM has a similar timed behavior[2] to that of the IMP. Based on the obtained PSM, we can verify whether the delay between an input m and an output c is bounded; if so, we derive the delay bound Δ'_{mc} from the total processing delays of the *Input-Device*, the *Code-Execution*, and the *Output-Device*. By verifying the PSM against the timing requirement $\mathcal{P}(\Delta'_{mc})$, we can formally check whether the implementation meets this relaxed timing requirement.

6.5. Implementation Schemes

We define the system boundaries of an implemented system using Parnas' *four-variables* [43] that is similar to the mapping in Subsection 4.5.1. In this mapping, *Code*(*PIM*) is integrated with a *platform*, and their interaction occurs at the *io*-boundary. *Code*(*PIM*) is (1) invoked by a platform, (2) reads input from a platform in the form of *i*-variables, (3) makes transition decisions, (4) writes outputs to the platform in the form of *o*-variables, and (5) waits for another invocation from a platform. A *platform* is composed of the *Input-Device* and *Output-Device*, and it interacts with the *environment* at the *mc*-boundary. The *Input-Device* (1) reads inputs from the environment in the form of *m*-variables, (2) processes the inputs, and (3) delivers the processed inputs to *Code*(*PIM*) in the form of *i*-variables. Similarly, the *Output-Device* reads (1) outputs from *Code*(*PIM*) in the form of *o*-variables, (2) processes the outputs, and (3) delivers the processed outputs to the environment in the form of *c*-variables. Fig. 18 illustrates the mapping of an implemented system (*IMP*) with

[2]We use the term "*similar* timed behavior" since it requires further assumptions to argue that the implementation shows exactly the "*same* timed behavior" with the *PSM* in a strict sense.

the four-variables model.

An implementation scheme defines a mechanism for implementing each interaction at the two system boundaries:

Definition 1 (Implementation Scheme)**.** An implementation scheme is a pair $\{MC, IO\}$, where

- *MC* specifies a reading (writing) mechanism and associated parameters for each variable $v \in m \cup o$ ($v' \in i \cup c$);
- *IO* specifies a reading (writing) mechanism and associated parameters for each variable $v \in i$ ($v' \in o$), as well as an invocation mechanism for the *Code*(*PIM*).

We next explain the implementation scheme in detail.

6.5.1. The mc-boundary interactions

An implementation scheme for the *mc*-boundary interaction is categorized in Table 7. An implementation scheme for the *Input-Device* specifies (1) what types of input signals are generated from the environment (in the form of *m*-variables), (2) how the *Input-Device* reads these input signals and delivers the processed inputs to *Code*(*PIM*) (in the form of *i*-variables), and (3) the minimum and maximum delays – represented by the platform-specific parameters $\text{delay}_{\min}$ and $\text{delay}_{\max}$ – that the *Input-Device* takes to transform an input signal to a program value that can be read by the *Code*(*PIM*). Similar information can be defined by an implementation scheme for the *Output-Device*.

Implementation scheme MC_s: The input interaction subcategory includes implementation schemes that a platform reads the *m*-variables whose signals (or values) are changed by the environment. The schemes are categorized based on the types of signals generated from the environment. Fig.19-(a) illustrates three types of signals considered in the input interaction.

Table 7: Implementation Scheme for Environment-to-Platform Interaction (m, c)

Interaction Type	Signal Type	Read Policy	Platform Specific Parameters (P_{mc})
Input Interaction (Env->Platform)	Pulse Signal (Type 1)	Interrupt (Rising Edge)	(1) Min/Max input processing delay (2) Input minimum inter-arrival time (Env)
	Sustained Signal (Type 2)	Interrupt (Rising Edge)	(1) Min/Max input processing delay (2) Input minimum inter-arrival time (Env) (3) Input sustained duration (Env)
		Interrupt (Falling Edge)	
		Polling	(1) Min/Max input processing delay (2) Input sustained duration (Env) (3) Input minimum inter-arrival time (Env) (4) Polling Interval
	Sustained Signal Until Read (Type 3)	Interrupt (Rising Edge)	(1) Min/Max input processing delay (2) Input minimum inter-arrival time (Env)
		Polling	(1) Min/Max input processing delay (2) Polling Interval (3) Input minimum inter-arrival time (Env)
Output Interaction (Platform->Env)	Pulse Signal (Type 1)	-	(1) Min/Max output processing delay
	Sustained Signal (Type 2)	Non-Blocking	
		Blocking	
	Sustained Signal Until Read (Type 3)	Non-Blocking	
		Blocking	

The *type-1* signal is a pulse signal. This signal does not have a sustained duration; more precisely, its sustained duration is too short to be captured through a polling-based scheme. Therefore, a platform is only able to read the *type-1* signal through an *interrupt-based* policy, in which an interrupt-service routine is automatically called for processing whenever any signal change is detected on a sensor. For example, an infusion pump should detect drug-drops using a drop sensor in order to precisely calculate the volume of drugs infused. Such a drug-drop passes the sensor too fast, so that it is typically detected through an interrupt-based scheme.

The *type-2* signal has a non-zero sustained duration; once the signal is triggered, it is sustained for a certain amount of time-duration, and then disappears. Therefore, a platform has options to read the *type-2* signal, either as an *interrupt-based* or a *polling-based* scheme. In the interrupt-based scheme, a platform has two additional options in choosing the timing

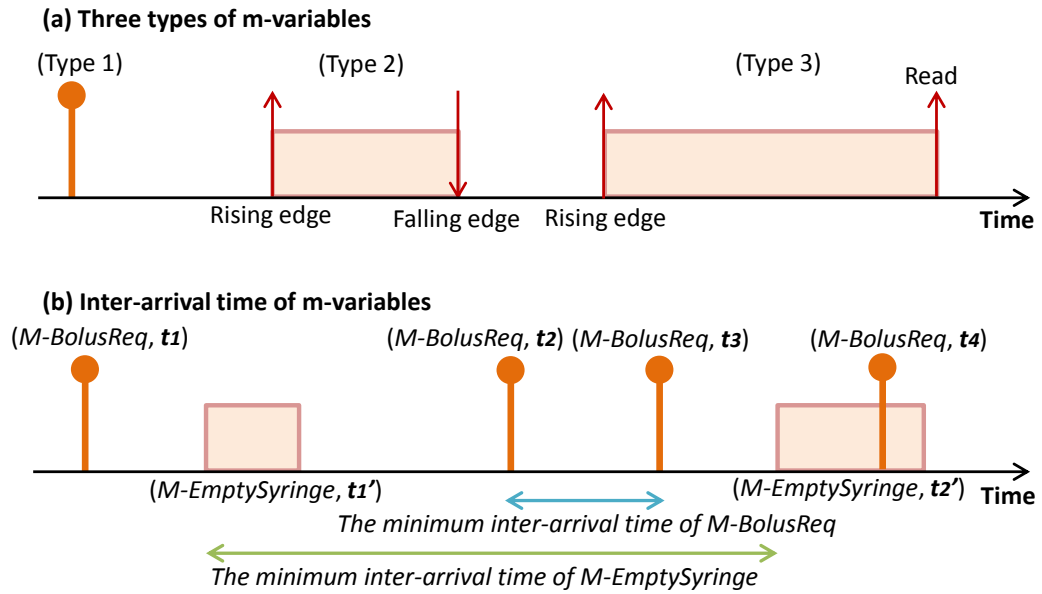


Figure 19: The environmental signal types

of signal reading, a *rising-edge* or a *falling-edge*. For example, an infusion pump should detect a pressed state of a bolus request button or a closed state of an empty syringe sensor; the granularity of the sustained time for both signals are far greater than a drug-drop so that they can be detected through a polling-based scheme.

The *type-3* signal is sustained until the signal is read by a platform. Like the *type-2* signal, a platform can read the *type-3* signal through either an interrupt-based scheme or a polling-based scheme. However, in case of the interrupt-based scheme, a platform is only able to read the signal at its rising-edge since a falling-edge is triggered immediately when a platform reads it. For example, the environment is another system that keeps an item in a buffer until a platform reads the item.

The output interaction subcategory includes implementation schemes that a platform writes signals to the environment, which is typically performed through actuators. Like the input interaction, the output interaction considers three different types of environmental signals. Unlike the input interaction, however, a platform can be blocked or non-blocked over the outputs that are currently being produced. In the former case, a platform cannot produce

another output until the current output is read by the environment or disappears due to a time-out. In the latter case, a platform can produce another output at any time, regardless of the output currently produced.

We believe that the three types of signal categories are sufficiently general enough in describing discrete environment[3] and implemented systems, so any platform may choose one of the implementation schemes to read inputs from and to write outputs to the environment.

Platform Specific Parameters P_{mc}: These implementation schemes require *platform-specific parameters* to be implemented on a platform. Once an input signal is read either through an interrupt or a polling-based scheme at the *mc*-boundary, the *Input-Device* requires a processing delay until it is transformed into a program value that can be read by *Code*(*PIM*) at the *io*-boundary. This processing delay is characterized as ($\text{delay}_{\text{min}}$, $\text{delay}_{\text{max}}$); a processed input value will be available for *Code*(*PIM*) in between this time interval. In case of using a polling-based scheme to read *type-2* or *type-3*, the *Input-Device* specifies a polling interval at which it samples the availability of the signals. Note that the environmental signals also have parameters. For the *type-2* signal, its sustained duration is characterized as (*min-sus*, *max-sus*); that is, once a signal is triggered, it is sustained for at least *min-sus* time, but it disappears before *max-sus* time. In addition, each input signal generated from the environment is associated with a minimum inter-arrival time that specifies the minimum duration in which a particular input never occurs. For example, Fig.19-(b) shows that two inter-arrival time defined over *m-BolusReq* and *m-EmptySyringe* inputs, respectively. Even though these parameters are related to the environment, changes on these parameters lead to different timed behaviors of a platform. For example, if the sustained duration of *type-2* signal is too short compared to the polling interval of a platform, a platform may miss the input. If the inter-arrival time of input is too short compared to the processing delay of a platform, a platform may suffer buffer-full conditions.

[3]Our categorization does not consider the aspects of the continuous environment. If we take into account the continuous signals generated from the environment, the categorization needs to be more elaborated.

6.5.2. The io-boundary interactions

An implementation scheme for the *io*-boundary interaction is categorized in Table 8. An implementation scheme for the interactions at the *io*-boundary specifies (1) how the *Code*(*PIM*) is invoked for its execution by a platform, and (2) how the *Code*(*PIM*) receives inputs from the *Input-Device* and delivers outputs to the *Output-Device*.

Implementation scheme IO_s: The invocation type subcategory includes implementation schemes about how a platform triggers *Code*(*PIM*) (its example is Listing 6.1). In case of the *periodic* invocation, *Code*(*PIM*) is periodically triggered (*e.g.,* the line 3 in Listing 6.1 is unblocked periodically), and reads inputs from a platform (*e.g.,* the line 4 in Listing 6.1).

Assume that a platform finished processing an input that had been read from the environment; for example, electrical signal changes on the bolus-request button (*m*-variable) is detected at the *mc*-boundary, and the *Input-Device* converts it in the form of a program value (*e.g.,* true or false in *i*-variable) that needs to be passed through *io*-boundary. Then, there are two communication data structures to pass the processed input (*i.e., i*-variables) to *Code*(*PIM*); one is through shared variables, and another is through buffers that can accommodate a finite size of items. When using a buffer, *Code*(*PIM*) has two additional options in reading the buffer upon every invocation: (1) a single input is read from a buffer, (2) all inputs are read from a buffer.

Fig. 20 illustrates the timed behavior of the implementation scheme among the environment (*ENV*), a platform, and *Code*(*PIM*). Here, in the *mc*-boundary, *ENV* generates three *type-2* signals, m_1, m_2, m_3 in order; and, a platform reads them using an interrupt-based scheme on their rising edges. In the *io*-boundary, a buffer is used to deliver the processed inputs to *Code*(*PIM*); *Code*(*PIM*) is invoked periodically (five consecutive invocations are shown here). Depending on the choice of the read policy from a buffer, *Code*(*PIM*) uses different sets of inputs to make transition decisions, which is performed at the line 5 in Listing 6.1. For example, when *Code*(*PIM*) makes a transition decision at $4th$ invocation, it uses a single

Table 8: Implementation Scheme for Platform-to-Code Interaction (i, o)

Invocation Type	Interaction Type	Communication Mechanism	Read Policy	Platform-Specific Parameters (P_{io})
Periodic	Input Interaction (Platform-> Code)	Shared variable	-	(1) Invocation period
		Buffers	Read One (Code)	(1) Invocation period (2) Input buffer size
			Read All (Code)	
	Output Interaction (Code -> Platform)	Shared variable	-	(1) Invocation period (2) Polling interval
		Buffers	Read One (Platform)	(1) Invocation period (2) Output buffer size (3) Polling interval
			Read All (Platform)	
Aperiodic	Input Interaction (Platform→ Code)	Shared variable	-	(1) Aperiodic trigger event
		Buffers	Read One (Code)	(1) Aperiodic trigger event (2) Input buffer size
			Read All (Code)	
	Output Interaction (Code -> Platform)	Shared variable	-	(1) Aperiodic trigger event (2) Polling interval
		Buffers	Read One (Platform)	(1) Aperiodic trigger event (2) Output buffer size (3) Polling interval
			Read All (Platform)	

input value (i_2) in case the *read-one* policy is used; on the other hand, it uses two input values (i_2 and i_3) in case the *read-all* policy is used. Using different read-policies results in different timed behaviors of $Code(PIM)$.

As another implementation scheme, $Code(PIM)$ can be triggered in an *aperiodic* manner; that is, a set of events is defined that can trigger $Code(PIM)$; $Code(PIM)$ is triggered only if an event in this set occurs. A *shared variable* can also be used to deliver inputs from a platform to $Code(PIM)$; $Code(PIM)$ can read a shared variable, and a platform overwrites it whenever a new input comes in.

Platform Specific Parameters P_{io}: This implementation scheme requires *platform-specific parameters* to be implemented on a platform. If a buffer-scheme is used to implement

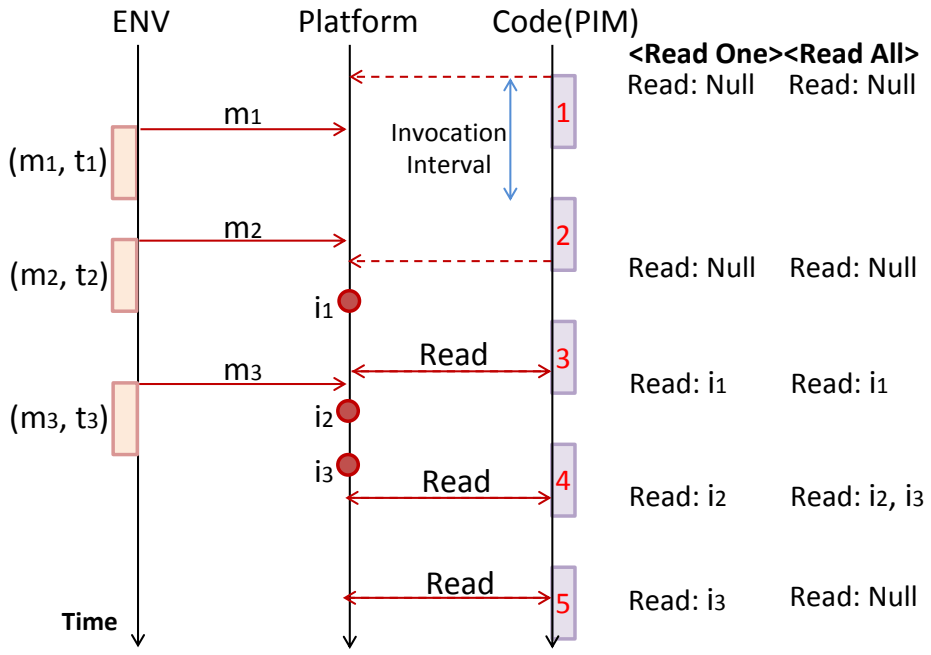


Figure 20: The illustration of the interactions at the *io*-boundary

a communication medium, a platform should specify its maximum buffer size.

Code(*PIM*) should check the communication medium to read inputs written by a platform. *Code*(*PIM*) is able to read the communication medium only if it is being invoked. Under periodic invocation, a platform should specify the invocation period; every expiration of the invocation period, *Code*(*PIM*) reads inputs from the communication medium, performs transitions, and writes outputs. Similarly, a platform should check the communication medium too to read outputs written by *Code*(*PIM*). A platform can check the communication medium through either an interrupt or a polling-based mechanism. In the interrupt-based mechanism, a platform is informed when outputs are written by *Code*(*PIM*). In the polling-based mechanism, a platform periodically checks the communication medium; in this case, it should specify the polling interval.

Different implementation schemes lead to different delays from the instant the environment generates an input signal until the instant the *Code*(*PIM*) reads the processed input. For example, using a polling mechanism for detecting the environmental input (*m*-variables)

can prolong the reading up to the next polling time, and using an aperiodic invocation for the *Code*(*PIM*) can reduce the delay by invoking *Code*(*PIM*) immediately whenever the processed input is inserted to the buffer. In Section 6.6, we introduce a modular transformation algorithm that transforms a *PIM* into a *PSM* for a particular combination of implementation schemes as an example.

6.6. Modular Transformation from *PIM* to *PSM*

The transformation algorithm takes a *PIM* and an implementation scheme (*IS*) as inputs, and produces a *PSM* as an output. A resulting *PSM* introduces a similar timed behavior close to the timed behavior of an actual implementation, IMP. The algorithm is modular, as it preserves the structure of the *PIM* for any implementation scheme defined in Section 6.5. This modularity makes the platform-specific timing verification possible for a range of implementation schemes that an arbitrary target platform may choose to execute the *Code*(*PIM*). Before describing the algorithm, we first discuss the timed behavior of the *PSM* that the algorithm constructs from the *PIM* and the *IS*.

We give the definitions of *PIM*, *PSM*, *IS* below to explain the transformation algorithm.

Definition 2 (*PIM*)**.** A *PIM* is defined as a network of UPPAAL automata of $M \parallel ENV$, and M is defined as $M = (L, l_0, C, A, E, I)$, where L is a set of locations, l_0 is an initial location, C is a set of clocks, A is a union of a set of input synchronizations $A_m = \{m_1, ... , m_k\}$, and a set of output synchronizations $A_c = \{c_1, ... , c_j\}$, and E is a set of transitions, I is a set of invariants.

Note that input and output synchronizations in Definition 2 are expressed using m and c variables only, which implies that the interactions between M and *ENV* occur at the *mc*-boundary. This means that the interactions at the *io*-boundary is not expressed at the *PIM* level of abstraction.

Definition 3 (*PSM*)**.** A *PSM* is defined as a network of UPPAAL automata of $M_{IO} \parallel IF_{MI_1}, ... , IF_{MI_k} \parallel IF_{OC_1}, ... , IF_{OC_j} \parallel EXE_{IO} \parallel ENV_{MC}$

Note that a subscript of each automaton in Definition 3 specifies a system boundary where the interactions occur. For example, M_{IO} and EXE_{IO} model the interactions at the *io*-boundary; IF_{MI_k} and IF_{OC_j} model the interactions at both the *mc*-boundary and *io*-boundary; and ENV_{MC} models the interactions at the *mc*-boundary. Such interactions are modeled using either channel synchronizations or variables in UPPAAL semantics.

The transformation algorithm is compatible with the following implementation scheme IS_1. Algorithms compatible for other implementation schemes can be similarly designed preserving the modular structures.

Example 6.6.1 (IS_1)**.** The implementation scheme 1 is given by $IS_1 = \{MC_1, IO_1\}$, where: (1) $MC_1(v) = \langle$(*pulse* signal, interrupt, rising-edge); (delay$_{\min}$= 1, delay$_{\max}$= 3)$\rangle$ and $MC_1(v') = \langle$(*pulse* signal); (delay$_{\max}$= 1, delay$_{\max}$= 3)$\rangle$ for all $v \in m \cup o$, and $v' \in i \cup c$; and (2) $IO_1(v) = \langle$(Buffers, Read-all); (buffer-size= 5)$\rangle$ for all $v \in i \cup o$, and $IO_1(invoke)$ = $\langle$(Periodic invocation); (period = 100)$\rangle$.

Fig. 21 illustrates the different timed behaviors of the *PIM* and the *PSM*. As shown in the figure, in the *PIM*, M is directly synchronized with *ENV* at the *mc*-boundary: whenever an input is triggered ($m_k!$), M can immediately accept it ($m_k?$); similarly, whenever M produces an output ($c_k!$), the output is immediately visible to the environment ($c_k?$).

However, in the *PSM*, M_{IO} (which will be constructed from M) is indirectly synchronized with the environment ENV_{MC} via a *platform* whose behavior is abstracted as the parallel composition of IF_{MI} (input interface), IF_{OC} (output interface), and EXE_{IO} (code execution) automata. Specifically, when an input is triggered ($m_k!$), it is first read ($m_k?$), processed, and enqueued to a buffer by IF_{MI}. The buffered input is then dequeued by EXE_{IO}, which also performs the synchronization with M_{IO} ($i_k!$). Subsequently, M_{IO} produces the corresponding output and enqueues the output to a buffer at the *io*-boundary. Finally, IF_{OC} dequeues the output, processes it, and makes the processed output visible to ENV_{MC} ($c_k!$).

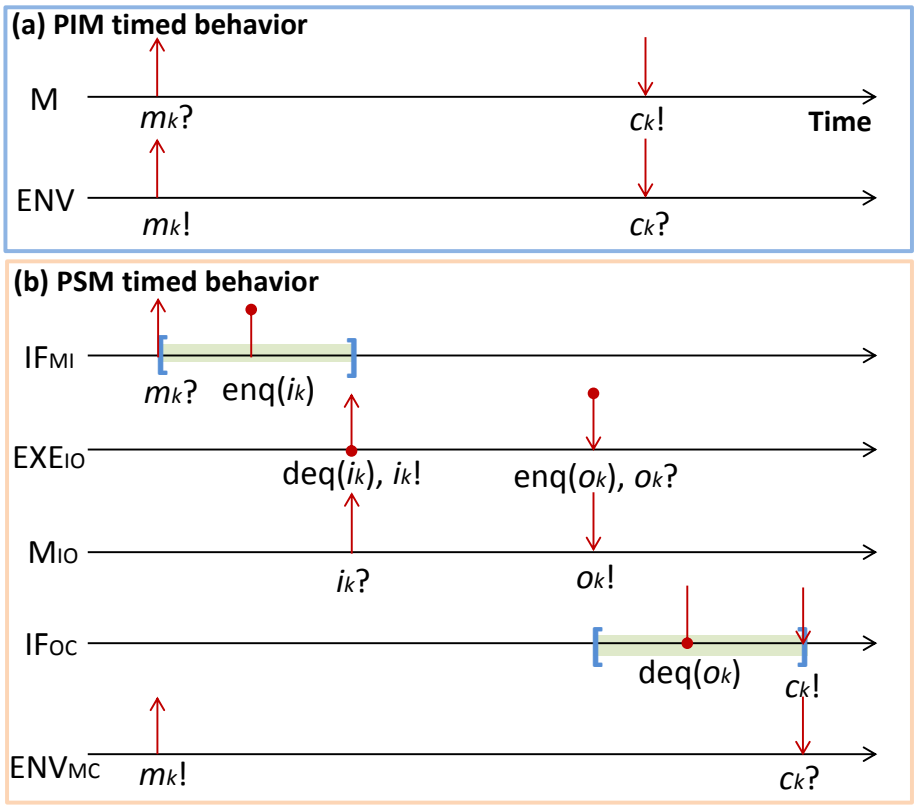


Figure 21: The illustration of the timed behavior of PIM and PSM under the implementation scheme

We next explain the detailed algorithm for constructing a *PSM* with the above timed behavior. We focus on a transformation algorithm that is compatible with the implementation scheme IS_1 in Example 6.6.1; algorithms compatible for other schemes can be designed similarly.

We first define the necessary operations that are used for the transformation algorithm:

- *Rename*(s_{src}, s_{dst}): returns an item whose name is changed from s_{src} into s_{dst},
- *Copy*(S_{src}, S_{dst}): a pair-wise copy of items in a set S_{src} into another set S_{dst},
- *Insert*(s_{src}, S_{dst}): inserts an item s_{src} into a set S_{dst},
- *CreateL*(l_k, inv_k): creates a new location l_{src} whose invariant is inv_k,

- *CreateT*(e, l_{src}, l_{dst}): creates a new transition e whose source location is l_{src}, and destination location is l_{dst},
- *Associate*(e, g, a, u): Associate a transition e with a guard condition (g), and an input or an output synchronization (a), and an update statement (u),
- *ID*(s): returns an identification of s, which is either a location or a synchronization.

The next three subsections explain (1) $M_{IO} \| ENV_{MC}$, (2) $IF_{MI} \| IF_{OC}$, and (3) EXE_{IO} that comprises the *PSM*.

6.6.1. Construction of M_{IO} and ENV_{MC}

In contrast to the M synchronizations with the environment over m and c variables, M_{IO} is desynchronized from the environment. This implies that *Code*(*PIM*) should directly interact with a platform at the *io*-boundary to communicate with the environment (see Fig. 18). The purpose of this transformation is to obtain M_{IO} that abstracts a part of such timed behavior of *Code*(*PIM*) when interacting with its platform.

M_{IO} is defined as $M_{IO} = (L_{IO}, l_{IO0}, C_{IO}, A_{IO}, E_{IO}, I_{IO})$, and Listing 6.2 is a pseudo algorithm (*Transform1*) that constructs M_{IO} from M. Intuitively, M_{IO} remains syntactically the same as M except its synchronizations are renamed from m to i, which implies the interaction between *Code*(*PIM*) and a platform at the *io*-boundary. *curLoc* is an integer variable that keeps track of current locations of M_{IO}; this variable is updated every transition to record the identification of the current location. The value of *curLoc* is used to perform necessary synchronizations in EXE_{IO} that will be introduced in Subsection 6.6.3.

Listing 6.2: The pseudo algorithm for M to M_{IO} transformation

```
function Transform1(M, M_IO)
 Copy(L, L_IO);
 Copy(l, l_IO0);
 Copy(C, C_IO);
 Copy(I, I_IO);
 for each m_k ∈ A_m and c_k ∈ A_c
```

```
  Insert(Rename(m_k, i_k), A_IO);
  Insert(Rename(c_k, o_k), A_IO);
endfor
for each e_k ∈ E
  CreateT(e_IOk, e_k.l_src, e_k.l_dst);
  Associate(e_IOk, e_k.g, e_k.a, (e_k.u ∪ curLoc:=ID(e_IOk.l_dst)));
  Insert(e_IOk, E_IO);
endfor
endfunction
```

Fig. 22-(1) is the example of M_{IO} transformed from M in Fig. 17-(1) using the algorithm *Transform1*; Fig. 22-(2) is the example of ENV_{MC} transformed from ENV in Fig.17-(2). Note that the environment model remains exactly the same under the current implementation scheme where only *type1* signal is considered. Now, the two input synchronizations in M_{IO}, *i-BolusReq* and *i-EmptySyringe*, are not synchronized with the output synchronizations of ENV_{MC}, *m-BolusReq* and *m-EmptySyringe* as implied in their different names. Similarly three output synchronizations in M_{IO}, *o-StartInfusion* and *o-StopInfusion* and *o-Alarm*, are not synchronized with their complementary synchronizations of ENV_{MC} for the same reason. Such desynchronization between M_{IO} and ENV_{MC} results in some missing information in connecting the input and output data flow across the two system boundaries. We introduce the input and output interface automata that fill such missing information in Subsection 6.6.2.

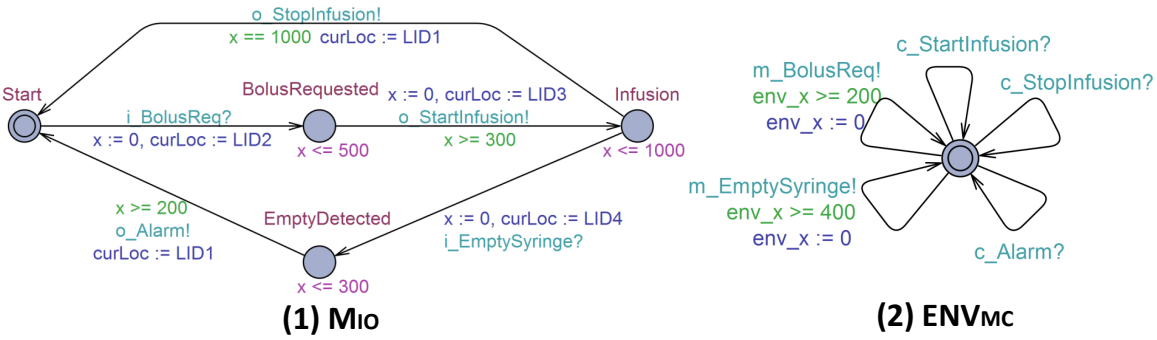


Figure 22: The M_{IO} and ENV_{MC} of the *PSM*

6.6.2. Construction of IF_{MI} and IF_{OC}

The transformed M_{IO} is desynchronized from its environment according to the *Transform1* algorithm in Listing 6.2. IF_{MI} (IF_{OC}) is an input (output) interface automaton that models the data flow from m to i (from o to c), which is performed by the *Input-Device* (*Output-Device*). This automaton models (1) the platform-specific delays introduced when converting an *environmental* input to a *program* input (a *program* output to an *environmental* output), and (2) the communication mechanism used to deliver a program input to the *Code*(*PIM*) (a program output to the platform).

Construction of IF_{MI}: IF_{MI} is an input interface automaton that abstracts the data flows from m to i that are performed by the *Input-Device*. IF_{MI} is synchronized with ENV_{MC} over m-variables to read environmental input signals (*e.g., type1* signal). Then, it finishes input processing in between minimum and maximum processing delay defined in P_{mc1} of IS_1. The processed inputs in the form of i-variables are enqueued in a *buffer* since a buffer scheme is used in IS_1 for the input interaction between the *Input-Device* and *Code*(*PIM*). The buffered inputs are ready for being read by *Code*(*PIM*). The purpose of this transformation is to obtain IF_{MI} that abstracts such timed behavior of the *Input-Device*.

IF_{MI} is a network of UPPAAL automata defined as $IF_{MI} = IF_{MI_1} \parallel IF_{MI_2} \parallel ... \parallel IF_{MI_k}$, where k is the number of input synchronizations in A_m of M. An IF_{MI_k} automaton is defined as $IF_{MI_k} = (L_{MI}, l_{MI_0}, C_{MI}, A_{MI}, E_{MI}, I_{MI})$. Listing 6.3 is the pseudo algorithm (*Transform2*) that constructs IF_{MI} from M and IS_1.

Listing 6.3: The pseudo algorithm for IF_{MI} construction

```
function Transform2(IS1, M, IFMI)
 for j = 1 to k
  Insert(iClk, IFMIj.CMI);
  Insert(CreateL(lidle, null), IFMIj.LMI);
  Insert(CreateL(lproc, iClk ≤ IS1.MaxDelayj), IFMIj.LMI);
  Insert(mj ∈ Am, IFMIj.AMI);
  for i = 1 to 4
   case i == 1:
```

```
    Create(e_i, l_idle, l_proc);
    Associate(e_i, null, m_k, iClk := 0);
   case i == 2:
    Create(e_i, l_proc, l_idle);
    Associate(e_i, (iClk ≥ MINDelay_j ∧ ilen < IS_1.iN), null, ienqueue(ID(m_k)) ∪ iClk := 0);
   case i == 3:
    Create(e_i, l_proc, l_idle);
    Associate(e_i, (iClk ≥ MINDelay_j ∧ ilen == IS_1.iN), null, null);
   case i == 4:
    Create(e_i, l_proc, l_proc);
    Associate(e_i, null, m_j, null);
   Insert(e_i, IF_MI_j.E_MI);
  endfor
  IF_MI = IF_MI ∥ IF_MI_j;
 endfor
endfunction
```

Here are intuitions of the timed behavior of the IF_{MI_j}. Regarding to the two locations, the l_{idle} location implies the *Input-Device* is ready to read an input m_j from ENV_{MC}; the l_{proc} location implies the *Input-Device* is currently processing the input m_j that has been read from ENV_{MC}. Once the input m_j is read by making a transition e_1, a processed input is ready in between ($IS_1.MinDelay_j$, $IS_1.MaxDelay_j$) that is defined in P_{mc1}. A processed input is delivered to $Code(PIM)$ through a finite size of buffer whose size is $IS_1.iN$ that is defined in P_{io1}. Therefore, there are two cases when the processed input needs to be inserted into the buffer, (1) the buffer has an empty slot (implied in the transition e_2), or (2) the buffer is full (implied in the transition e_3).

Note that it is possible for different types of inputs to have different processing delays; for example, *m-BolusReq* may have a processing delay (300, 500), while *m-EmptySyringe* has a processing delay (200, 600). Also, while processing an input m_j, the same type of input (*i.e.,* m_j) is ignored as implied in the transition e_4. However, processing an input m_j does not block processing other types of inputs m_k where $k \neq j$; for example, while $IF_{MI_{BolusReq}}$ is in the l_{proc} location, it is possible for a platform to read an input *m-EmptySyringe* for processing in $IF_{MI_{EmptySyringe}}$. Fig. 23 shows the $IF_{MI_{BolusReq}}$ and $IF_{MI_{EmptySyringe}}$

constructed from M in Fig. 17 and IS_1.

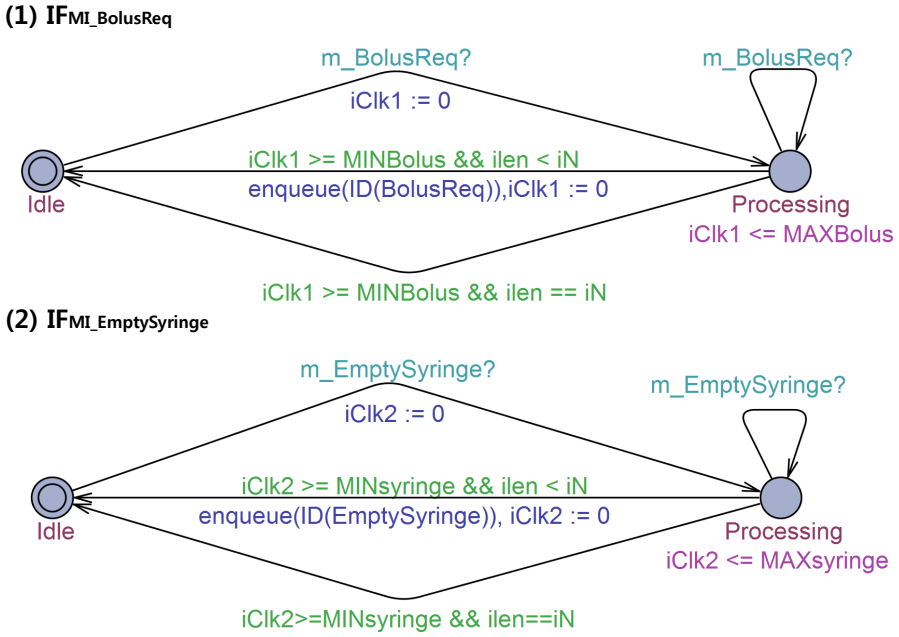


Figure 23: The input interface automata of *PSM*

Construction of IF_{OC}: IF_{OC} is an output interface automaton that abstracts the data flows from o to c that are performed by the *Output-Device*. IF_{OC} is synchronized with ENV_{MC} over c-variables to read outputs from $Code(PIM)$. Then, it finishes output processing in between minimum and maximum processing delay that is defined in P_{mc1} of IS_1. The processed outputs in the form of c-variables are ready to be written to ENV_{MC}. The purpose of this transformation is to obtain IF_{OC} that abstracts such timed behavior of the *Output-Device*.

Here is the construction algorithm of IF_{OC}. Similar to IF_{MI}, IF_{OC} is a network of UPPAAL automata defined as $IF_{OC} = IF_{OC_1} \parallel IF_{OC_2} \parallel ... \parallel IF_{OC_j}$, where j is the number of output synchronizations in A_c of M. An IF_{OC_j} automaton is defined as $IF_{OC_j} = (L_{OC}, l_{OC_0}, C_{OC}, A_{OC}, E_{OC}, I_{OC})$. Listing 6.4 is the pseudo algorithm (*Transform3*) that constructs IF_{OC} from M and IS_1.

Listing 6.4: The pseudo algorithm for M to IF_{OC} transformation

```
function Transform3(IS_1, M, IF_OC)
 for j = 1 to n
  Insert(oClk, IF_OC_j.C_OC);
  Insert(CreateL(l_idle, oClk≤IS_1.MaxDQDelay_j), IF_OC_j.L_OC);
  Insert(CreateL(l_proc, oClk≤IS_1.MaxDelay_j), IF_OC_j.L_OC);
  Insert(c_j ∈ A_c, IF_OC_j.A_OC);
  for i = 1 to 3
    case i == 1:
     Create(e_i, l_idle, l_proc);
     Associate(e_i, (olen > 0 ∧ ofind(ID(c_j)) == true), null, odequeue(ID(c_j)) ∪ oClk := 0);
    case i == 2:
     Create(e_i, l_proc, l_idle);
     Associate(e_i, oClk ≥ IS_1.MinDelay_j, c_j, oClk := 0);
    case i == 3:
     Create(e_i, l_idle, l_idle);
     Associate(e_i, ((olen == 0 ∨ ofind(ID(c_k)) == false) ∧ oClk == IS_1.MAXDQInterval), null, oClk :=
       0);
   Insert(e_i, IF_OC_j.E_MI);
  endfor
  IF_OC = IF_OC ‖ IF_OC_j;
 endfor
endfunction
```

The timed behavior of the IF_{OC} is similar to that of IF_{MI} except the data flows in an opposite direction. Since a buffer scheme is used in IS_1 for the output interaction between the *Output-Device* and *Code*(PIM), IF_{OC} now dequeues buffered outputs in the form of o-variables; then, it converts the buffered outputs into the outputs in the form of c-variables in between ($IS_1.MinDelay_j$, $IS_1.MaxDelay_j$) that can be written to the environment.

We make a note for the platform-specific parameter $IS_1.MaxDQDelay_j$ that is the polling interval defined in P_{io1}. This parameter specifies the upper time limit until the *Output-Device* is forced to check the output buffer to read outputs for processing. If this parameter is set to infinite, there is no timing constraint enforced to the *Output-Device* in reading the output buffer. In this case, it is possible for *Output-Device* not to read the output buffer forever so that output values are never visible at the mc-boundary even though Code(PIM)

keeps writing outputs to the buffer at the *io*-boundary. If it is set as an integer number, the *Output-Device* has to read the output buffer before the timeout. Fig. 24 shows the example $IF_{OC_{StartInfusion}}$ and $IF_{OC_{StopInfusion}}$ and $IF_{OC_{Alarm}}$ transformed from M in Fig. 17 and IS_1.

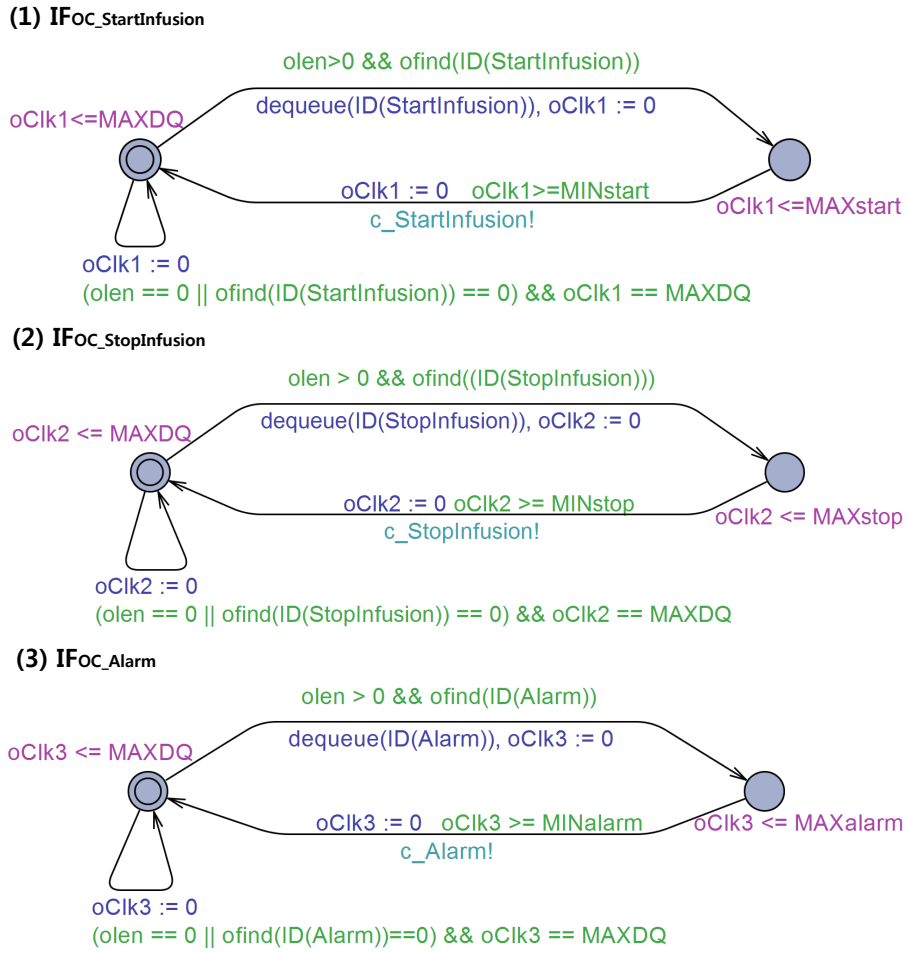


Figure 24: The output interface automata of *PSM*

6.6.3. Construction of EXE_{IO}

M_{IO} is transformed from M according to the *Transform1* algorithm in Listing 6.2. Occurrences of synchronizations in A_{IO} of M_{IO} imply that *Code*(*PIM*) has read inputs from the *Input-Device*, or Code(*PIM*) has written outputs to the *Output-Device*. However, even though IF_{MI} and IF_{OC} abstract the behavior of the *Input-Device* and the *Output-Device*,

there is no direct synchronizations in between M_{IO} and these automata; for example, the $IF_{MI_{BolusReq}}$ automaton in Fig. 23-(1) is synchronized with ENV_{MC} through the *m-BolusReq* channel, but there is no synchronization over the *i-BolusReq* channel with M_{IO}. This is because the chosen communication scheme (*i.e.,* a buffer-scheme) prevents M_{IO} from directly synchronizing with IF_{MI} and IF_{OC}. In other words, once an input is written to the input buffer by the *Input-Device*, *Code*(*PIM*) does not need to read it immediately; similarly, once an output is written to the output buffer by *Code*(*PIM*), the *Output-Device* does not need to read it immediately. These situations are illustrated in Fig. 21 as the two timing gaps (1) the time passage between *enq*(i) and *deq*(i), and (2) the time passage between *enq*(o) and *deq*(o).

EXE_{IO} automaton performs such synchronizations with M_{IO} only if when M_{IO} reads inputs from the buffer, and when M_{IO} writes outputs to the buffer. Such timing for buffer read-/write operations is closely tied to the invocation implementation scheme of *Code*(*PIM*); in other words, buffer read/write operations can occur while *Code*(*PIM*) is being invoked by a platform. In IS_1, *Code*(*PIM*) is invoked periodically (*i.e., await* function is periodically unblocked in Listing 6.1). Upon invocation, *Code*(*PIM*) first dequeues all inputs from the buffer (since IS_1 uses *read-all* policy), and performs a single input synchronization (since IS_1 uses *a single input transition* per invocation policy); an output synchronization is performed afterward when an output is enqueued to the buffer in a similar manner. After finishing the read and write operations, EXE_{IO} becomes idle waiting for the next periodic invocation. The purpose of this transformation is to obtain EXE_{IO} that abstracts such timed behavior of periodic invocation and synchronizations with M_{IO}.

EXE_{IO} is an UPPAAL automaton that is defined as $EXE_{IO} = (L_{EXE}, l_{EXE0}, C_{EXE}, A_{EXE}, E_{EXE}, I_{EXE})$. The construction of EXE_{IO} from M_{IO} and IS_1 consists of the construction for non-buffer and buffer operations.

Construction of the non-buffer operation part: *Transform4* in Listing 6.5 is the construction algorithm of the non-buffer operation part of EXE_{IO}. It creates six locations that

correspond to the execution of the *Code*(*PIM*) abstraction in Listing 6.1 as follows; l_{wait} implies that *Code*(*PIM*) is waiting for an invocation (*i.e., await* is being blocked); l_{active} location implies that *Code*(*PIM*) has been invoked, and ready for the execution of the rest of the code (*i.e., await* is unblocked); $l_{readready}$, $l_{readcomplete}$, $l_{writeready}$, $l_{writecomplete}$ locations imply that the respective computations -*readInput*, *Compute-InputTrans*, *Compute-OutputTrans*, *writeOutput*- occur. The two platform-specific parameters, $IS_1.InvokePrd$ and $IS_1.WCET$, are associated with the invariants of these locations. $IS_1.InvokePrd$ is the invariants of l_{wait} that specify the periodic invocation interval defined in P_{io1}; $IS_1.WCET$ is the invariant of the rest of the locations that specifies the worst case execution time of *Code*(*PIM*) defined in P_{cd1}. Fig. 25 is the EXE_{IO} constructed from M_{IO} in Fig .22-(1) where these six locations are represented.

Listing 6.5: The pseudo algorithm for construction EXE_{IO} *from* M_{IO} *to* transformation (Non-buffer operation part)

```
function Transform4(IS_1, M_IO, EXE_IO)
 Insert(eClk, C_EXE);
 Insert(CreateL(l_wait, eClk ≤ IS_1.InvokePrd), L_EXE);
 Insert(CreateL(l_active, eClk ≤ IS_1.WCET), L_EXE);
 Insert(CreateL(l_readready, eClk ≤ IS_1.WCET), L_EXE);
 Insert(CreateL(l_readcomplete, eClk ≤ IS_1.WCET), L_EXE);
 Insert(CreateL(l_writeready, eClk ≤ IS_1.WCET), L_EXE);
 Insert(CreateL(l_writecomplete, eClk ≤ IS_1.WCET), L_EXE);
 Copy(A_IO, A_EXE);
 for i = 1 to 8
   case i == 1:
    Create(e_i, l_wait, l_active);
    Associate(e_i, eClk == IS_1.InvokePrd, null, eClk := 0);
   case i == 2:
    Create(e_i, l_active, l_readready);
    g := null;
    for each l_k ∈ L_IO
     if l_k has at least one e_k ∈ E_IO such that e_k is associated with an input synchronization i_k ∈
     A_IO
       g := g ∨ curLoc == ID(l_k);
     endif
```

```
 endfor
 Associate(e_i, g, null, null);
case i == 3:
 Create(e_i, l_active, l_readcomplete);
 g := null;
 for each l_k ∈ L_IO
  if l_k has no e_k ∈ E_IO such that e_k is associated with an input action i_k ∈ A_IO
   g := g ∨ curLoc == ID(l_k);
  endif
 endfor
 Associate(e_i, g, null, null);
case i == 4:
 Create(e_i, l_readready, l_readcomplete);
 Associate(e_i, ilen == 0, null, null);
case i == 5:
 Create(e_i, l_readcomplete, l_writeready);
 g := null;
 for each l_k ∈ L_IO
  if l_k has at least one e_k ∈ E_IO such that e_k is associated with an output synchronization o_k ∈
  A_IO
   g := g ∨ curLoc == ID(l_k);
  endif
 endfor
 Associate(e_i, g, null, null);
case i == 6:
 Create(e_i, l_readcomplete, l_writecomplete);
 g := null;
 for each l_k ∈ L_IO
  if l_k has no e_k ∈ E such that e_k is associated with an output synchronization o_k ∈ A_IO
   g := g ∨ curLoc == ID(l_k);
  endif
 endfor
 Associate(e_i, g, null, null);
case i == 7:
 Create(e_i, l_writeready, l_writecomplete);
 Associate(e_i, olen == oN, null, null);
case i == 8:
 Create(e_i, l_writecomplete, l_waiting);
 Associate(e_i, null, null, eClk := 0);
Insert(e_i, E_EXE);
```

```
endfor
endfunction
```

This algorithm also generates transitions that are selectively taken depending on the presence of synchronizations associated with the current locations of M_{IO}. Therefore, transitions to $l_{readready}$ and $l_{writeready}$ locations can be skipped if no input and output synchronizations are possible in the current location of M_{IO}, respectively. For this purpose, the algorithm generates two outgoing transitions, e_2 and e_3, from the l_{active} location; e_2 is taken only if M_{IO} is in locations whose outgoing transitions include at least one input synchronization; for example, *Start* and *Infusion* are such locations in Fig. 22-(1). e_3 is taken only if M_{IO} is in locations whose outgoing transitions include at least one output synchronization; for example, *Bolus* and *Infusion* and *Empty* are such locations in Fig. 22-(1).

It also additionally generates the two transitions, e_4 and e_7, that are taken when the input buffer is empty and the output buffer is full in which case no synchronizations can happen (*i.e.,* M_{IO} has nothing to read an input from the empty buffer, and cannot write an output due to the fact that the buffer is full).

Construction of the buffer operation part:

Transform5 in Listing 6.6 is the transformation algorithm that constructs the buffer operation part of EXE_{IO}. Given the constructed locations using the non-buffer operation part, this algorithm basically adds necessary input and output transitions according to the following rules. For an input synchronization i_k associated with a transition e_k in M_{IO}, it creates two types of transitions from $l_{readready}$ to $l_{readcomplete}$ in EXE_{IO}. The first transition e'_{k1} implies that there is a buffered input associated with i_k, and the guard condition of the transition e_k is satisfied; then, it dequeues the input performing the input synchronization between M_{IO} and EXE_{IO} over i_k. The second transition e'_{k2} implies that the guard condition of the transition e_k is satisfied; therefore, it cannot synchronize over i_k even though there is an associated input in the buffer.

Transitions associated with output synchronizations are constructed in a similar way. Fig. 25 is the EXE_{IO} constructed from M_{IO} in Fig. 22-(1).

Listing 6.6: The pseudo algorithm for construction EXE_{IO} from M_{IO}(Buffer-operation part)

```
function Transform5(IS_1, M_IO, EXE_IO)
 for each e_k ∈ E_IO such that e_k is associated with an input synchronization i_k ∈ A_IO
   //Create the first transition.
   CreateT(e'_k1, l_readready, l_readcomplete);
   g' := null;
   Copy(e_k.g, g');
   Associate(e'_k1, (ilen > 0) ∧ g' ∧ ifind(ID(i_k)), i_k, idequeue(ID(i_k)));
   Insert(e'_k1, E_EXE);
   //Create the second transition.
   Create(e'_k2, l_readready, l_readcomplete);
   g' := null;
   Copy(negate(e_k.g), g');
   Associate(e'_k2, (ilen > 0) ∧ g', null, null);
   Insert(e'_k2, E_EXE);
 endfor

 for each e_k ∈ E_IO such that e_k is associated with an output synchronization o_k ∈ A_IO
   //Create the first transition.
   Create(e'_k1, l_writeready, l_writecomplete);
   g' := null;
   Copy(e_k1.g, g');
   Associate(e'_k1, (olen < IS_1.oN) ∧ g' ∧ curLoc == ID(e_k.src) , o_k, oenqueue(ID(o_k)));
   Insert(e'_k1, E_EXE);
   //Create the second transition.
   Create(e'_k2, l_writeready, l_writecomplete);
   g' := null;
   Copy(negate(e_k.g), g');
   Associate(e'_k2, (olen < IS_1.oN) ∧ g' ∧ curLoc == ID(e_k.src), null, null);
   Insert(e'_k2, E_EXE);
 endfor
endfunction
```

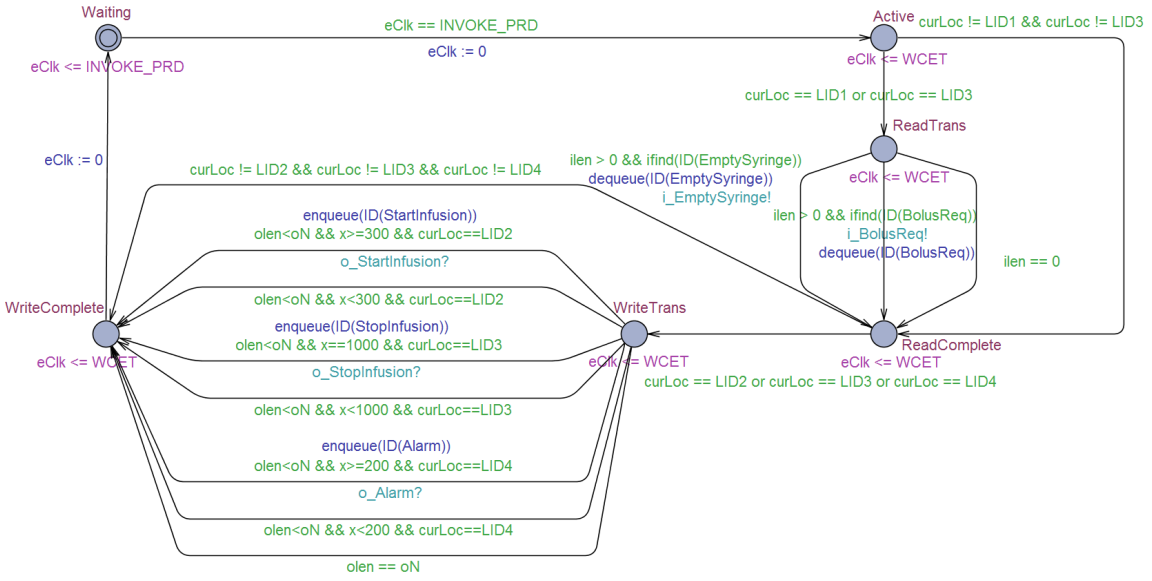


Figure 25: The EXE_{IO} synchronized with M_{IO} in Fig.22-(1)

6.7. The Property of the *PSM*

We explain the property of the *PSM* that has been transformed from the *PIM* according to the proposed algorithm. Here are definitions of three types of maximum delays of a pair (j) of input and output:

1) M-C Delay Δ_{mc}: specifies the maximum time passage from the instant the environment triggers an input (m_j, t_{m_j}) until the instant the environment observes an output (c_j, t_{c_j}) at the *mc*-boundary, *i.e.*, $\Delta_{mc} = t_{c_j}$ - t_{m_j}; this delay is illustrated in Fig. 21 as the time passage between the two synchronizations ($m_k!$ and $c_k?$) of ENV_{MC}.

2) Input-Delay Δ_{mi}: specifies the maximum time passage from the instant the environment triggers an input (m_j, t_{m_j}) at the *mc*-boundary until the instant the $Code(PIM)$ reads the input (i_j, t_{i_j}) at the *io*-boundary, *i.e.*, $\Delta_{mi} = t_{i_j}$ - t_{m_j}; this delay is illustrated in Fig. 21 as the guarded section of IF_{MI}.

3) Output-Delay Δ_{oc}: specifies the maximum time passage from the instant $Code(PIM)$ produces an output (o_j, t_{o_j}) at the *io*-boundary until the instant the environment observes

the output (c_j, t_{c_j}) at the *mc*-boundary, *i.e.,* $\Delta_{oc} = t_{c_j}$ - t_{o_j}; this delay is illustrated in Fig. 21 as the guarded section of IF_{OC}.

Given $PIM \models \mathcal{P}(\Delta_{mc})$, our goal is to find Δ'_{mc} such that,

$$PSM \models \mathcal{P}(\Delta'_{mc}),\ and\ \Delta'_{mc} \geq \Delta_{mc}.$$

Δ'_{mc} is a relaxed timing constraint from the original constraint Δ_{mc}. Such relaxation should be considered in terms of the *Input-Delay* and the *Output-Delay.* The *Input-Delay* is a function of ENV_{MC}, IF_{MI}, EXE_{IO}; that is, once the environment triggers an input, the sustained duration of the input signal, the input interaction scheme (*e.g.,* polling or interrupt-scheme), invocation-scheme (*e.g.,* periodic or aperiodic scheme) constitute the delay. On the other hand, the *Output-Delay* is a function of ENV_{MC}, IF_{OC}, EXE_{IO}; that is, once $Code(PIM)$ produces an output, the output interaction scheme, invocation-scheme, and the environmental behavior constitute the delay.

Remark 1. In general, we cannot determine the bound of Δ'_{mc} because some combinations of platform-specific parameters (P_{mc}, P_{io}) make Δ'_{mc} unbounded. However, we can derive timing constraints on implementation schemes that make Δ'_{mc} bounded. If they are satisfied, we can find such a bound.

We now show the four constraints that make Δ'_{mc} bounded:

- (Constraint 1) *Detection of all input signals*: Given an input pattern generated from ENV_{MC}, (1) IF_{MI} can detect all input signals, and (2) the maximum input processing delay of IF_{MI} is shorter than the minimum inter-arrival time.

- (Constraint 2) *No overflow of the input buffer*: The invocation interval of EXE_{IO} should be small enough w.r.t. the input processing speed of IF_{MI} so that each input can be read by EXE_{IO} before the input buffer overflows.

- (Constraint 3) *No overflow of the output buffer*: Given an output pattern generated by M_{IO}, (1) IF_{OC} has sufficient processing speed to process all outputs before the

output buffer overflows, and make them visible to ENV_{MC}, and (2) ENV_{MC} can read the produced outputs by IF_{OC} fast enough.

- (Constraint 4) *No internal transition occurrences*: Since ENV_{MC} generates an input, M_{IO} does not take internal transitions until the input is processed by IF_{MI}, and read by M_{IO}.

Lemma 1. *If the system constraints are verified in PSM, then (1) the Input-Delay is bounded by the function of all maximum platform-specific parameters that are used for ENV_{MC}, IF_{MI}, EXE_{IO}, (2) the Output-Delay is bounded by the function of all maximum platform-specific parameters that are used for ENV_{MC}, IF_{OC}, EXE_{IO},*

Proof. Here is the proof for the *Input-Delay*: Since the constraint 1 holds, the triggered input is read by IF_{MI} until:

- (Case 1) *type-1* signal is immediately accepted;
- (Case 2) *type-2* signal is accepted up to *max-sus* time;
- (Case 3) *type-3* signal is accepted up to *polling-interval*.

Once an input signal is read by IF_{MI}, since the constraint 2 holds, it takes up to $\text{delay}_{\max}$ input processing delay until it is inserted into the input-buffer; the inserted input is waiting to be read by $Code(PIM)$ whose behavior is modeled in EXE_{IO}. There are two different invocation mechanisms, and each introduce different maximum delays.

In case of the *periodic* invocation:

- (Case 1: Read-All policy) the buffered input is read within *2*invocation-interval* since EXE_{IO} will read the buffered input either in the current invocation or the next invocation;
- (Case 2: Read-One policy) the buffered input is read within (*input-buffer-size*)*2* *invocation-interval* since a buffered input needs to wait until other inputs that have

been previously inserted are read by *Code*(*PIM*); the buffer size determines the bound of the maximum waiting delay.

In case of the *aperiodic* invocation:

- (Case 1: Read-All policy) the delay takes up to 2*WCET, since if the current execution stage of *Code*(*PIM*) already passes, the buffered input will be read immediately after completing the current invocation;
- (Case 2: Read-One policy) the delay takes up to (*input-buffer-size*)*WCET, which constitutes the delays taken for *Code*(*PIM*) to read other inputs that have been already buffered.

Finally, since the constraint 4 holds, there is no internal transition occurring up to this point since the input is triggered from ENV_{MC}; therefore, M_{IO} can synchronize with the input read from the buffer. Therefore, *Input-Delay* is bounded by the summation of these platform-specific timing parameters, and the upper bound of the *Output-Delay* can be proved similarly. □

Recall that Δ_{mi} and Δ_{oc} are the upper bounds of *Input-Delay* and *Output-Delay* as Δ_{mi} and Δ_{oc}. Let $\Delta_{io-internal}$ be the maximum internal delay of the *PIM* for processing the input and output pair (i, o). The following lemma holds:

Lemma 2. *If the system constraints are verified in PSM, then, we can determine* Δ'_{mc} *such that,* $PSM \models \mathcal{P}(\Delta'_{mc})$, *and* $\Delta'_{mc} = \Delta_{mi} + \Delta_{oc} + \Delta_{io-internal}$.

Proof. The rest of the delays that contribute to Δ'_{mc} is the internal delay of *PIM* for processing the input i and the output o. Therefore, the maximum possible *M-C delay* is bounded by the summation of these three types of delays. □

We make the following claim:

Theorem 1. *If PSM verifies the system constraints, and if a platform is correctly described*

using the implementation scheme, then PSM $\models \mathcal{P}(\Delta'_{mc})$ *implies* IMP*(Code(PIM), IS)* $\models \mathcal{P}(\Delta'_{mc})$.

We can validate the assumption: *"if a platform is correctly described using the implementation scheme"* through testing, and show the case-study in Section 6.8.

6.8. Case Study of the *PSM* Verification

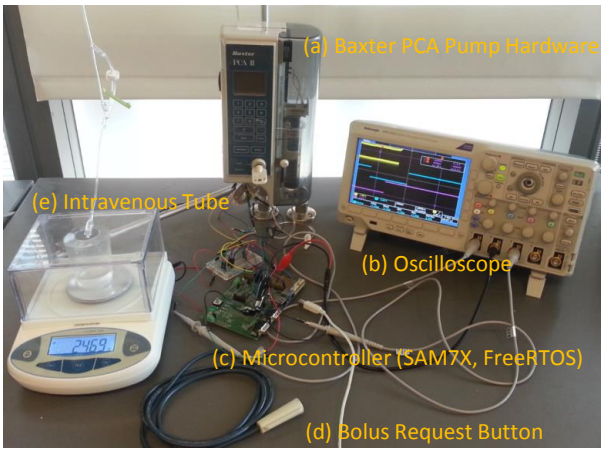
The case study is intended to show how the proposed timing verification framework can be used.

Case study setting: We created a *PIM* that meets REQ1 using UPPAAL; that is, it verifies that the *PIM* starts infusion within *500ms* when a patient requests a bolus (*i.e.,* $PIM \models \mathcal{P}(\Delta_{mc})$). Then, the *PIM* is automatically generated into C-code using TIMES tool [14] to obtain *Code*(*PIM*). The *Code*(*PIM*) is then integrated with the Baxter PCA infusion pump platform as shown in Fig. 26-(a), and all timing information is measured using the oscilloscope. The *Code*(*PIM*) is integrated with the target platform following IS_1 except the polling scheme is used to read the bolus request input. Fig. 26-(b) shows the platform-specific parameters used in this implementation. The *PSM* is transformed from the *PIM* and IS_1 according to the algorithm introduced in Section 6.6.

Under this setting, we performed 60 times of the bolus request scenarios on the Baxter PCA infusion pump platform, and measured the timing delays using the oscilloscope. Some of these parameters are obtained from testing (*e.g.,* input processing delay, WCET), and some of these parameters are set by ourselves (*e.g.,* polling interval, invocation period). Throughout the 60 times of bolus request testing, the *M-C delay*, the *Input-Delay* and the *Output-Delay* are measured together, and their average, maximum, minimum time delays are summarized in the *Measured Delay-*(*IMP*) row in Table 9.

PSM **Verification over** $\mathcal{P}(\Delta_{mc})$: REQ1 is *not* satisfied under the *PSM* (*i.e.,* $PSM \not\models \mathcal{P}(\Delta_{mc})$). In other words, when the *PIM* is composed with the implementation scheme IS_1,

(a) the experimental platform setup



(b) the platform-specific parameters

Category	Parameter	Value
P_{mc1}	Sustained duration (for *m-BolusReq*)	Min: 100ms Max: 300ms
	Input processing delay (for *m-BoulsReq*)	Min: 3ms Max: 10ms
	Input polling interval	50ms
	Output processing delay (for *c-StartInfusion*)	Min: 3ms Max: 10ms
P_{io1}	Invocation period	100ms
	Dequeue interval	300ms
	Input-buffer size	5
	Output-buffer size	5
P_{cd1}	Minimum inter-arrival time (for *m-BolusReq*)	2000ms
	WCET	30ms

Figure 26: The experimental setup (PCA Infusion Pump System)

it takes more than *500ms* until the infusion is started from the moment when a bolus is requested. This is because the additional platform-dependent delays originated from the composition of *PIM* and IS_1 contribute to the prolonged delay. From this *PSM* verification result, we can conclude that the actual implementation may introduce a case where the actual delay (Δ'_{mc}) is greater than the delay (Δ_{mc}) that has been verified in *PIM*. Among 60 test results, we observed 53 times of the timing requirement violations (*i.e.*, only 7 cases showed the delay is less than *500ms*); this implies $\mathsf{IMP} \not\models \mathcal{P}(\Delta_{mc})$.

Table 9: The experiment result

		M-C delay	Input Delay	Output Delay	Buffer Overflow
Verified Upper Bound (PSM)		1430ms	490ms	440ms	X
Measured Delay (IMP)	Avg.	610ms	97ms	215ms	X
	Max	748ms	152ms	304ms	
	Min	456ms	48ms	100ms	

PSM **Verification over** $\mathcal{P}(\Delta'_{mc})$: Now, we verify a more relaxed timing requirement $\mathcal{P}(\Delta'_{mc})$. Given the measured *M-C delay*, we chose a Δ'_{mc} ($1430ms$) determined by Lemma 1

and 2 based on the platform-specific parameters in Fig. 26-(b); here, the verified *M-C delay* and *Input-Delay* and *Output-Delay* are 1430ms (490ms + 440ms + 500ms), 490ms, 440ms, respectively. We also verified the system constraints under the *PSM*. Thus, assuming the validated platform-specific parameters through testing are correct, we can conclude $\mathsf{IMP} \models \mathcal{P}(\Delta'_{mc})$. Note that this verified result is also consistent with the testing result; in other words, all measured time-delays are bounded by the verified time-bound (*i.e.,* $1430ms$).

6.9. Summary of the *PSM* Verification

The integration phase aims at obtaining a final implemented system by (1) composing the platform-independent code (produced in the PI-Phase) and the platform-dependent code (produced in the PD-Phase), and by (2) checking whether the composition conforms to the timing requirements.

In particular, Sections 6.3 to Section 6.8 introduced the platform-dependent verification process that particularly focuses on the timing aspects of the implemented system; that is, this verification process enables the timed behavior of the implemented system to be formally verified against the timing requirements. The general category of implementation schemes are proposed from which a platform can choose a particular combination for the execution of the platform-independent code. Then, the proposed transformation algorithm systematically transforms a *PIM* into a *PSM* given an implementation scheme. If a platform is correctly described using the implementation scheme and meeting the system constraints, the resulting *PSM* introduces the timed behavior that is close to the timed behavior of the actual implemented system.

6.10. The Problem Statement (Timing Testing)

The formal verification approach introduced from Section 6.3 to Section 6.8 can be used under the assumption that the platform-specific timing information required to construct an implementation scheme are known a priori, and can be formally abstracted along with the platform-independent model. In other words, if such platform-specific timing informa-

tion is not known for some reason (*e.g.,* the internal of the platform-dependent code is considered proprietary information from the device manufacturer's perspective), one needs to consider a different approach to show the timing requirement conformance of the implemented system. In the rest of the sections, a timing testing approach is proposed, which can be alternatively used to show timing requirement conformance when such platform-specific timing information is not known.

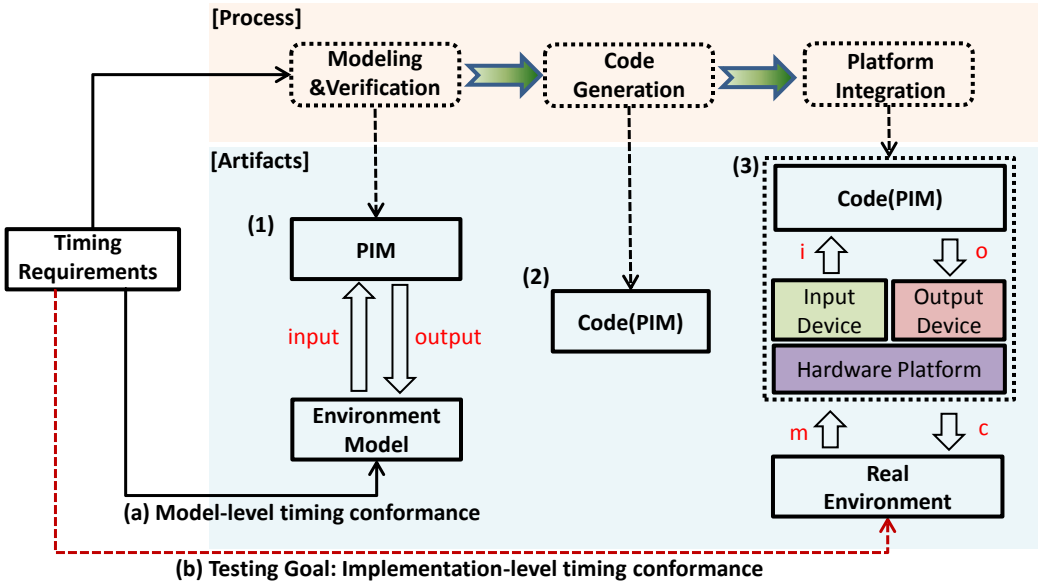


Figure 27: The goal of the testing framework in the model-based implementation.

Fig. 27 shows the model-based implementation process that we have used to develop infusion pump software. Consider the following timing requirement:

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

The modeling and verification phase (1) aims at creating a model that interacts with the environment model. For example, Fig. 28 is a Stateflow model that captures the timed behavior of the infusion pump system, and the timing requirement of REQ1 can be verified; the details of this model are explained later.

Code generation (2) aims at automatically generating source code that preserves the model behavior. Note that the generated code (denoted as $Code(PIM)$) is assured to conform to

the model structure through this process. For example, the code generator used in [40] 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. 28.

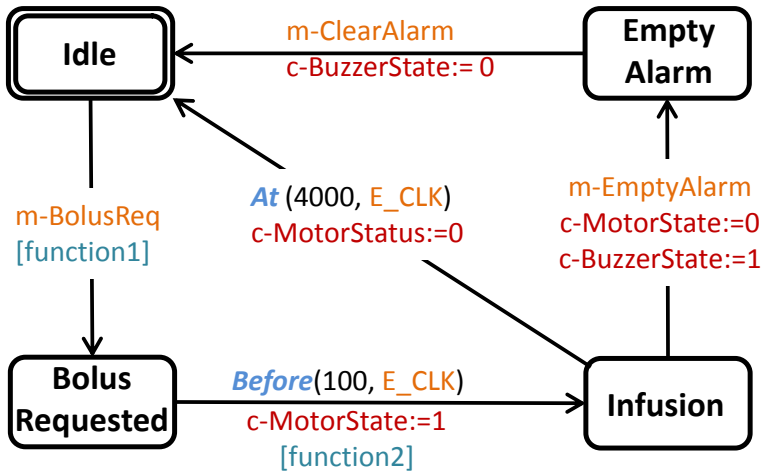


Figure 28: The example Stateflow model for infusion pump system

Platform integration (3) aims at adding interfacing code that is necessary for $Code(PIM)$ to be executed on a platform. For example, input/output interfacing code bridges physical input/output (denoted as m and c variables) and abstracted input/output of $Code(PIM)$ (denoted as i and o variables). In this example, input interfacing code converts pressing the bolus request button, which generates an electrical signal change (that is mapped to the *m-BolusReq* input in Fig. 28), into updating the corresponding generated boolean variable of $Code(PIM)$.

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.

6.11. The Approach Overview (Timing Testing)

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.

6.11.1. Mapping the four-variables to the implemented system

To test the timing requirements of an implemented system, we identify the relevant input and output of the implemented system with associated timing constraints. One assumption made in the platform-independent model 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 $Code(PIM)$ reads the input event that is pre-processed by some input processing mechanism (*e.g.,* sampling routines) executing on the hardware platform.

In order to have a uniform interpretation of the input and output of the implemented system, Parnas' four-variables model (that were used in the formal verification approach introduced from Section 6.3 to Section 6.8) is also used for this testing approach to formally define different abstraction boundaries of the implemented system. Fig. 27-(3) illustrates the implemented system that shows the mapping with the four variables (m, i, o, c). Here is the implication of the mapping of the four variables to the implemented systems in the testing approach:

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* in Fig. 28 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-MotorState* variable may have a range of integer values in order to specify the speed associated with the pump motor (*e.g.,* [*c-MotorState* == 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*(*PIM*). An input variable (i) characterizes events that are read by *Code*(*PIM*). For example, *Code*(*PIM*) that is generated from the model in Fig. 28 has three i-variables; *i-BolusReq*, *i-EmptyAlarm*, *i-ClearAlarm*. *Input-Device* is responsible for converting the corresponding 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*(*PIM*). For example, *Code*(*PIM*) has two o-variables; *o-MotorState* and *o-BuzzerState*. *Output-Device* 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*(*PIM*) and the platform (*i.e., io*-boundary) from those in the boundary between the platform and the physical environment (*i.e., mc*-boundary). Also note that the platform-independent model expresses the timing of input and output at the *mc*-boundary; on the other hand, its generated code *Code*(*PIM*) will be composed with a platform at the *io*-boundary. We next explain the testing framework based on the four-variable mapping.

6.11.2. Testing Objectives and R-M testing

In the model-based implementation as shown in Fig. 27, the timing requirements that were verified in the model (Fig. 27-(1)) may be violated in the implemented system (Fig. 27-(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 timed behavior of the platform-independent 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) $\{(m\text{-}BolusReq, t_{m1}), (c\text{-}BolusStart, t_{c1})\}$
- (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:

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

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 platform-independent model does not hold in the implemented system that executes *Code*(*PIM*). 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.

Given the timing requirement of REQ1, Fig. 29 illustrates the timed behavior of the model (Fig. 29-(a)), and its implemented system using four-variables (Fig. 29-(b),(c),(d)). In Fig. 29-(a), when *m-BolusReq* event is provided to the model of Fig. 28, the *c-MotorState* event is produced within 100 ms. Fig. 29-(b) shows the timed 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. 29-(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*(*PIM*) actually reads the input (after being processed by *Input-Device*).

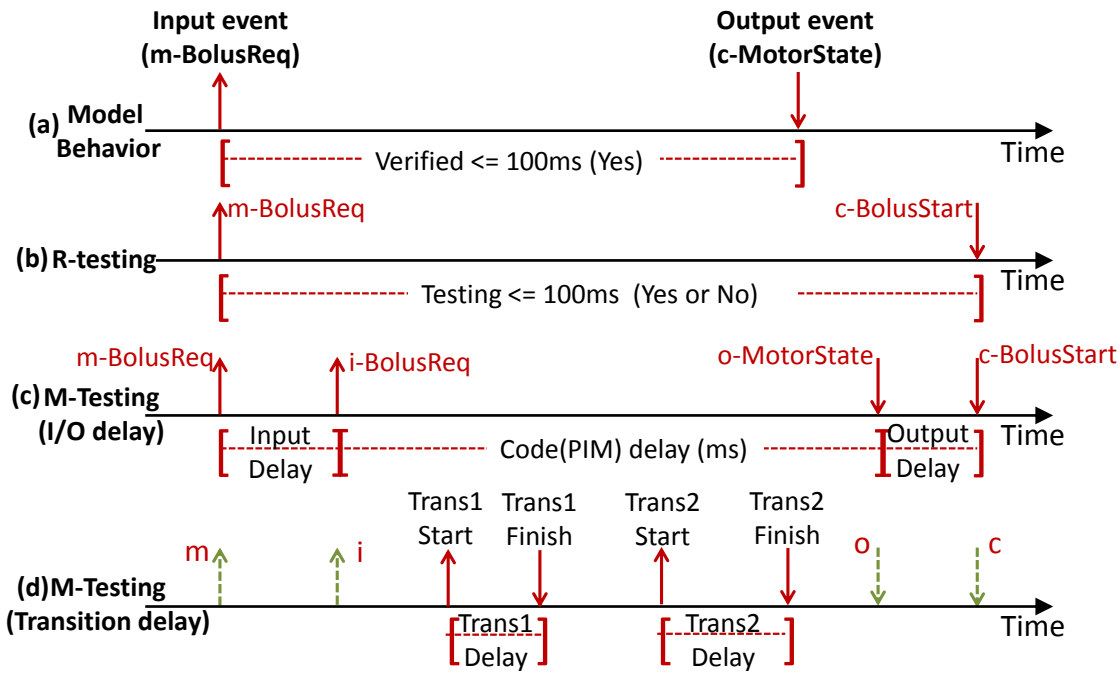


Figure 29: The illustration of the timing testing in the R-M testing framework

For example, Input-Delay in Fig. 29-(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*(*PIM*) writes the output until the moment the output becomes actually visible to the physical environment. For example, Output-Delay in Fig. 29-(c) illustrates the time delay associated with the (*o-MotorState*, *c-BolusStart*) pair.

(3) *Code(PIM)-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*(*PIM*) reads an *i*-event until the moment an *o*-event is produced. For example, *Code*(*PIM*)-Delay in Fig. 29-(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. 28: transition from *Idle* to *BolusRequested* and transition from *BolusRequested* to *Infusion*. Fig. 29-(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.

6.12. Case Study of the Timing Testing

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 platform-independent model.

Case-Study Setting: We consider REQ1 to be a timing requirement that needs to be satisfied in both the model and the implemented system. A model (Figure 27-(1)) is created using Stateflow, and a part of this model is shown in Figure 28. The timing requirement is verified in the model using the Simulink Design Verifier [41]. 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 [40] is used to automatically generate C source code (Fig. 27-(2)) from the verified model.

The generated code (*Code*(*PIM*)) 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 the 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 the timing deviation of different implemented systems. We consider three representative implementation schemes to

integrate *Code*(*PIM*) with the target platform. The three implementation schemes are as follows:

Implementation Scheme 1 (Single-threaded implementation): The implementation, *Code*(*PIM*), is executed by a single thread that is invoked periodically. In our case study, *Code*(*PIM*) 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*(*PIM*) 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*(*PIM*) 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*(*PIM*) execution. In our case study, the summation of the thread periods along the path of sensing-*Code*(*PIM*)-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*(*PIM*) 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*(*PIM*) 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*(*PIM*) thread, and the other two threads have a higher and a lower priority than the *Code*(*PIM*) thread respectively. These threads do not communicate with the *Code*(*PIM*), 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 10: Testing Results: Measured time-delays for the bolus request scenario in REQ1

Table 10 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 the *m-BolusReq* event and the *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.

6.13. Summary of the Timing Testing

Section 6.10 to Section 6.12 introduced the platform-dependent timing testing process that

can be applied to the implemented system in which the timing information of the platform-dependent code is not explicitly known. This testing process systematically measures the delay occurring during the interaction between the platform-dependent code and the platform-dependent code through R-testing and M-testing. R-testing measures the time difference of the input and output events occurring at the boundary of the platform and the environment (*i.e., mc*-boundary); this type of 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 platform-independent model using the input and output events occurring at the boundary of the platform-independent code and the platform-dependent code (*i.e., io*-boundary); this type of testing can be used to quantify timing deviation of implemented systems from the platform-independent model. This timing testing process can be alternatively used with the platform-dependent timing verification process introduced in Sections 6.3 to Section 6.8 depending on the availability of the platform-dependent timing information of an implemented system in order to show the timing requirement conformance.

CHAPTER 7 : Integration Phase (Part 2)

7.1. The Problem Statements and Challenges

In Chapter 6, we introduced an issue that an implemented system may experience prolonged delays from that of the platform-independent model. This is mainly because the platform-independent code is directly generated from the platform-independent model without compensating the platform-processing delay. Fig. 30 illustrates this issue. When the platform-independent model is constructed, it only models input/output timing that can be observed at the *mc*-boundary by hiding the internal timing details as to how the platform-independent code interacts with a particular platform. Accordingly, when the platform-independent code is generated using existing code generators [40] [40], the model-level timing parameters are directly translated into the code-level timing parameters in order to determine the input/output timing at the *io*-boundary. This is problematic because the platform-processing delay will be again added to the delays implemented by the platform-independent code. This situation may result in the implemented systems that do not conform to the timing requirements that have been verified in the platform-independent model. The techniques proposed in Chapter 6 aim at analyzing how such deviations impact the timing requirement conformance.

In this chapter, we introduce a way to *optimize* the platform-independent code by adjusting the timing parameters of the platform-independent model in case those prolonged delays result in the timing requirement violation. In other words, we want to appropriately compensate the platform-processing delay by adjusting the timing parameters of the platform-independent model; hence, if such a timing parameter adjustment is successful, it will result in an implemented system that conforms to the timing requirements verified in the platform-independent model.

Here is the challenge in adjusting the timing parameters to achieve the desired compensation of the platform-processing delay:

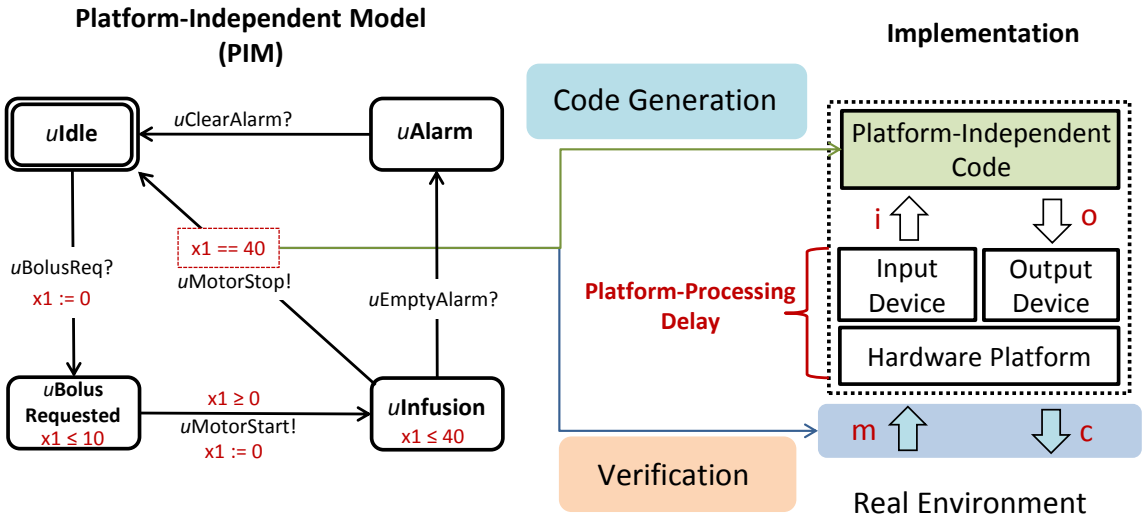


Figure 30: The source of the timing deviation between the platform-independent model and its implementation

Challenge: It is not straightforward how to calculate how much timing parameters of a platform-independent model should be adjusted to achieve such a compensation; this is because several I/O semantics mismatches between the model and the implementation makes those timing parameters sometimes *decreased* or *increased* in a complicated pattern in order to realize the compensation.

The delays implemented by the platform-independent code should compensate the additional delay required from a given platform. However, such a compensation cannot be obtained only by *reducing* the values of all timing parameters of the platform-independent model. For example, the timing semantics of the synchronization in UPPAAL applies the same regardless of whether the transition type is an input or output synchronization; that is, the transitions associated with any types of synchronization should occur taking zero time when it should happen. On the other hand, the implemented system processes input and output differently as illustrated in the information flow of Fig. 30; that is, the platform-independent code can read an input (from a buffer) only after the environment has generated the input, followed by the platform that processes it by placing the input

value in the buffer; on the contrary, an output can be visible in the environment only after the platform-independent code produced output by placing it in the buffer, followed by the platform that has processed the output. Due to such differences, the values of the timing parameters sometimes has to be *reduced* or *increased* in order to obtain the platform-independent code that meets the desirable timed behavior of the implemented system.

7.2. The Approach Overview

We aim at addressing this challenge by proposing a model transformation approach for the code generation. We argue that the platform-independent models are not appropriate representations of the code-level timed behavior, especially when the generated code has to be integrated with platforms that have non-zero I/O processing delays. We transform a platform-independent model into a software model by explicitly characterizing (non-zero) platform-processing delays and appropriately compensating those delays. The software model is then used to generate code, which would be implemented on the platform whose processing delays are compensated during the model transformation process. The resulting implemented system is guaranteed to meet the timing constraints that have been verified in the platform-independent model.

Our model transformation approach involves two steps. First, check whether it is feasible to compensate the processing delays of a given platform while preserving the bounds on the delays between observable events in the platform-independent model and implemented systems. Second, adjust the timing parameters in the platform-independent model to obtain a software model that compensates the platform-processing delays. We formulate and solve the problem using Integer Linear Programming (ILP). We define the objective function of ILP based on the goal of obtaining a software model with the *least* timed behavior perturbation from the platform-independent model. The linear constraints of ILP are given to quantify the delay-bound differences between the platform-independent model and an implemented system, taking into account the model's I/O delays accumulated over paths (between pairs of I/O transitions) and the platform-processing delays. By solving the ILP

problem, we obtain a set of timing parameter assignments that can be used to transform the platform-independent model into a software model.

We demonstrate the usefulness of our approach via a case study of infusion pump systems. Experimental results show that systems implemented with the code generated from our transformed software models have better a performance in terms of timing constraints preservation compared to the systems implemented with the code directly generated from the platform-independent model.

7.3. Problem Formulation

7.3.1. Motivating Example

We first explain, through the following example, why we need to transform platform-independent models for representing the code-level timed behavior. Consider a platform-independent model represented as an event-clock automaton [13] (cf. Definition 5) shown in Figure 31. Each transition in the model is labelled with either an input (a_1? and a_3?) or an output (a_2!) event. Each event is associated with a clock, which is automatically reset to zero whenever a transition associated with the corresponding event is taken. For example, the clock x_{a_1} is reset to zero when the transition labelled with the input event a_1? is taken. Transitions are also annotated with clock guard conditions. For example, the guard condition $x_{a_1} \geq 2 \wedge x_{a_1} \leq 10$ means that the transition can be taken anytime in between 2 and 10 time-unit after the previous transition. It is straightforward to verify that the example model satisfies the following timing constraint:

(**REQ0**): *"a system shall produce the output* a_2! *within 2 and 10 time-units since the input* a_1? *event occurs from the environment".*

The timing parameters in the clock guard conditions do not distinguish code-level delays and platform-processing delays. Suppose the platform-processing delay is 2 time units. Then a system implemented using the code generated from the model shown in Figure 31 would

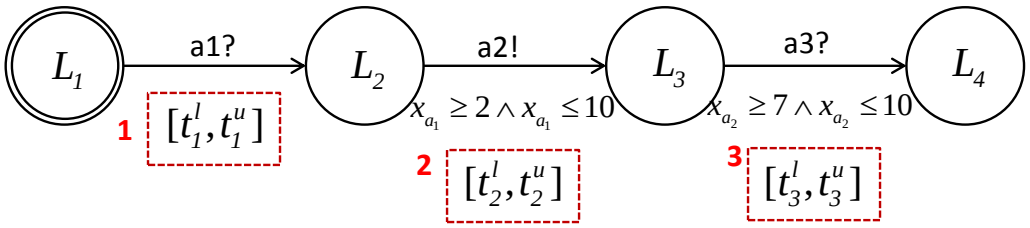


Figure 31: Model 1 with variable assignment (Sequential Pattern)

not preserve REQ0, because the delay of output a_2! is now bounded by 4 and 12 time units. This implies that the platform-independent model shown in Figure 31 is not an appropriate representation for the code generation. Thus, there is a need for model transformation. In practice, the given platform-processing delays are often in a range (*i.e.,* minimum and maximum bounds) rather than a single constant, which makes the model transformation challenging. Because we need to consider the intermix of min/max platform-processing delays and code-level delays, while the latter may also be affected by the model structure (*e.g.,* loops). Once we know what the ranges of platform delays are, timing guards in the model can be adjusted so that the generated code, running on the platform, would exhibit correct system-level behavior.

7.3.2. Problem Statement

Figure 32 shows an overview of relations between platform-independent model, software model, code, platform and implementation. We adopt Parnas' four-variable model [43] to define the boundaries of an implemented system (shown in the right side of the figure). Based on this variable mapping, we define the boundaries of an implemented system to formalize the problem: *io*-boundary that separates a platform and the code, and *mc*-boundary that separates an environment and a platform. Suppose that, for a given platform, the minimum and maximum delays of processing each input/output event is known (cf. Definition 6). The goal is to transform a platform-independent model ($\mathsf{M_s}$) into a software model ($\mathsf{M_c}$) by compensating the platform-processing delays (P), in order to preserve the timing

constraints in the implemented system.

We first define three functions: $f_{\mathsf{M_s}}$, f_{SOF}, and f_{IMP}, which quantify the min/max delay-bounds of the occurrence of an I/O event succeeding another event in the platform-independent model, *io*-boundary (software model), and *mc*-boundary, respectively. Formally, we define $f_{\mathsf{M_s}}(i,j)$ as the min/max delay-bounds of *simple paths* (*i.e.,* those without cycles) starting from the transition i and ending with the transition j in the platform-independent model $\mathsf{M_s}$. For example, the timing constraint REQ0 can be formally denoted by $f_{\mathsf{M_s}}(1,2)$=[2,10], representing that the output transition 2 shall be taken in between 2 and 10 time-units after taking the input transition 1 of Model 1 shown in Figure 31. Given a simple path p, we can also write $f_{\mathsf{M_s}}(p)$. We define f_{SOF} and f_{IMP} in a similar fashion as for $f_{\mathsf{M_s}}$. In the model transformation from $\mathsf{M_s}$ to $\mathsf{M_c}$, we only adjust the timing parameters (*i.e.,* clock guards) and do not change the model structure. Therefore, there are one-to-one mappings between transitions i and j for $f_{\mathsf{M_s}}(i,j)$, $f_{\mathsf{SOF}}(i,j)$ and $f_{\mathsf{IMP}}(i,j)$.

Definition 4 (*Delay-Bound Inclusion Constraint*)**.** A system implementation preserves the delay-bound inclusion constraint with respect to the corresponding platform-independent model iff:

- the minimum delay bound of f_{IMP} for any pair of I/O events at the *mc*-boundary is *no less* than that of $f_{\mathsf{M_s}}$,
- the maximum delay bound of f_{IMP} for any pair of I/O events at the *mc*-boundary is *no greater* than that of $f_{\mathsf{M_s}}$.

Formally, we denote the constraint satisfaction by $f_{\mathsf{IMP}}(i,j) \in f_{\mathsf{M_s}}(i,j)$ iff $f^{\mathsf{min}}_{\mathsf{IMP}}(i,j) \geq f^{\mathsf{min}}_{\mathsf{M_s}}(i,j)$ and $f^{\mathsf{max}}_{\mathsf{IMP}}(i,j) \leq f^{\mathsf{max}}_{\mathsf{M_s}}(i,j)$ for any pair of I/O transitions (events) i and j.

Example 7.3.1. Model 1 shown in Figure 31 has three pairs of I/O transitions, where $f_{\mathsf{M_s}}(1,2)$=[2,10], $f_{\mathsf{M_s}}(2,3)$=[7,10], $f_{\mathsf{M_s}}(1,3)$=[9,20]. The delay-bound inclusion constraint holds for a system implementation with $f_{\mathsf{IMP}}(1,2)$=[4,6], $f_{\mathsf{IMP}}(2,3)$=[8,9], $f_{\mathsf{IMP}}(1,3)$=[15,19]}, because $f^{\mathsf{min}}_{\mathsf{IMP}} \geq f^{\mathsf{min}}_{\mathsf{M_s}}$ and $f^{\mathsf{max}}_{\mathsf{IMP}} \leq f^{\mathsf{max}}_{\mathsf{M_s}}$ for all possible pairs of I/O transitions. □

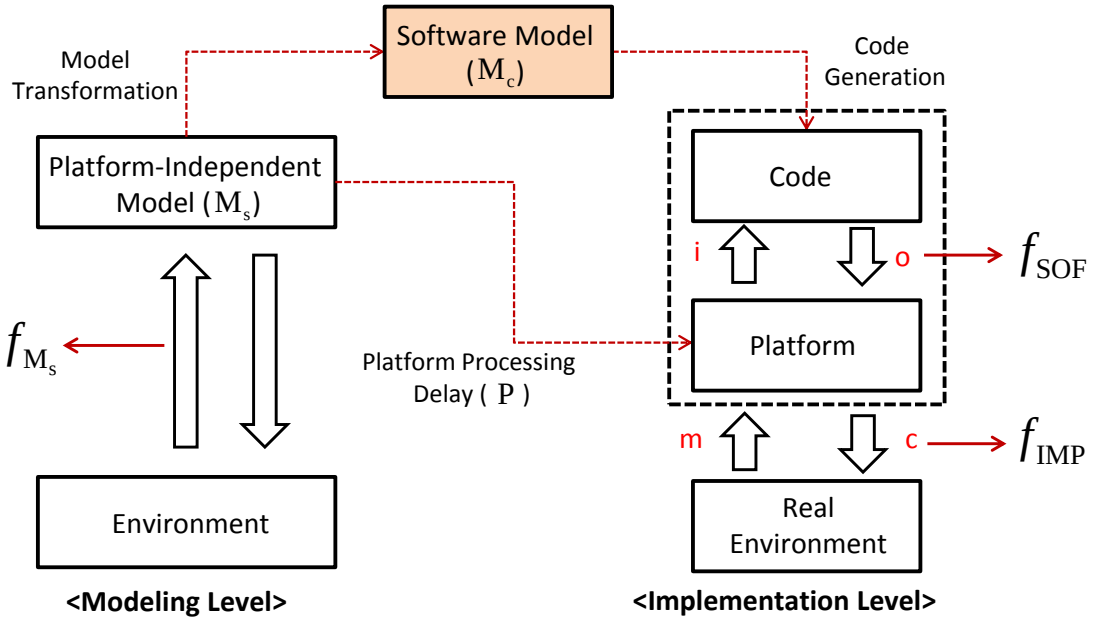


Figure 32: Mapping from system models to implementations

The key of model transformation from a platform-independent model M_s into a software model M_c lies in solving the research problem of finding a suitable function f_{SOF} such that the induced function f_{IMP} (based on the known platform-processing delay P) preserves the delay-bound inclusion constraint with respect to the function f_{M_s} (determined by M_s). We also need to show that any implemented system meeting the delay-bound inclusion constraint is guaranteed to satisfy the timing requirements that are verified in the platform-independent model M_s.

7.3.3. Approach Overview

Finding a suitable function f_{SOF} is a challenging problem. On the one hand, we need to consider its dependency to the function f_{IMP} and the platform-processing delay P. On the other hand, we need to make sure that the derived function f_{IMP} and the platform-independent model function f_{M_s} satisfy the delay-bound inclusion constraint. The simple shrinking or expanding timing guards of f_{M_s} would not give us a satisfying f_{SOF}, because the change of timing guards in one transition (*e.g.*, with the intent of decreasing f_{SOF}) may actually increase f_{SOF} for another transition/path, due to the complex dependency among

various I/O transitions and the mixture of minimum/maximum delays.

Our approach is to formalize those dependencies in terms of a set of linear constraints to automatically find timing parameter assignments for the function f_{SOF} using the ILP. First, we propose algorithmic procedures to compute $f_{\mathsf{M_s}}$ and f_{IMP}. The computation of $f_{\mathsf{M_s}}$ is based on the timing parameters (*i.e.,* clock guard constants) in the platform-independent model $\mathsf{M_s}$, while the computation of f_{IMP} is based on the timing parameters of the software model $\mathsf{M_c}$ (represented as variables in f_{SOF}) and the platform-processing delays P. Then, we formalize the delay-bound inclusion constraint between $f_{\mathsf{M_s}}$ and f_{IMP} as a set of linear inequality constraints for ILP. A satisfying solution of the ILP problem gives us a set of variable assignments for f_{SOF}, which can be used to parameterize the timing guards of the software model $\mathsf{M_c}$. Finally, we show by Theorem 2 that the system implemented using the code generated from the software model $\mathsf{M_c}$ is guaranteed to satisfy the timing requirements (*i.e.,* bounded delays between a pair of I/O events).

7.4. Computing $f_{\mathsf{M_s}}$ and f_{IMP}

In this section, we develop algorithmic procedures for computing $f_{\mathsf{M_s}}$ and f_{IMP}, which are functions that quantify the min/max delay-bound of any pair of I/O transitions and events in the platform-independent model and the implementation, respectively. In this paper, we consider platform-independent models that are represented as *event-clock automata* [13].

Definition 5 (*Event-Clock Automata*)**.** An event-clock automaton is a tuple $\mathsf{M}= (\mathcal{L}, L_0, L_f, \Sigma, E)$, where $\mathcal{L}$ is a set of locations; $L_0 \in \mathcal{L}$ and $L_f \in \mathcal{L}$ are an initial and a final location, respectively; $\Sigma = \Sigma_{in} \cup \Sigma_{out}$ is an alphabet with a set of input (*resp.* output) events Σ_{in} (*resp.* Σ_{out}); $E = \{(L, L', a, \varphi)\}$ is a fine set of transitions with each transition connecting a starting location L and an ending location L', labelled with an input/output event $a \in \Sigma$, and associated with a clock constraint φ over the clocks C_Σ.

We refer to [13] for the formal operational semantics of event-clock automata. An informal semantic description for an example model could be found in Section 7.3.1. In this paper, we

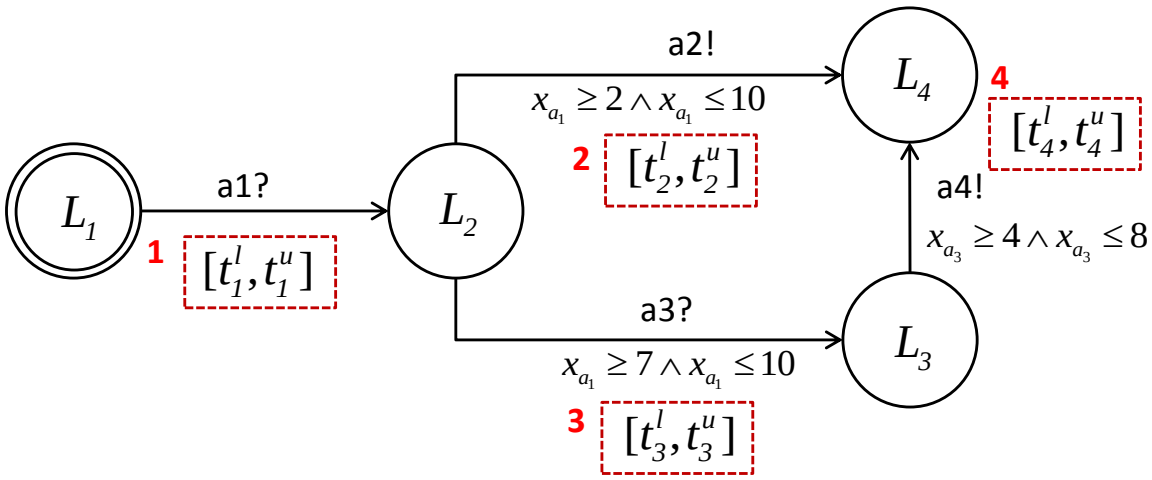


Figure 33: Model 2 with variable assignment (Alternative Pattern)

consider three different model structure patterns, namely, *sequential* (*e.g.,* Model 1 shown in Figure 31), *alternative* (*e.g.,* Model 2 in Figure 33), and *cyclic* (*e.g.,* Model 3 in Figure 34).

7.4.1. Computing $f_{\mathsf{M_s}}$

The computation of the function $f_{\mathsf{M_s}}$, which represents the minimum and maximum delay-bounds between two I/O transitions in the platform-independent model, is non-trivial. Firstly, there can be many different paths in between two transition occurrences. In this case, the path that has the minimum delay can be different from the one that has the maximum delay. Therefore, we need to examine all possible paths between the two transitions in order to compute $f_{\mathsf{M_s}}$. Secondly, paths connecting two transitions may include cycles. Since we assume non-negative guard conditions, the minimum delay bound is obtained by considering only a simple path. However, when it comes to the maximum delay bound, it differs depending on how many cycles are taken between the two transitions. For example, in Model 3 (cf. Figure 34), the maximum delay-bound between transitions 1 and 3 increases as more cycles are taken.

We first show how to compute $f_{\mathsf{M_s}}$ for a pair of transitions which are connected by *simple paths* (*i.e.,* paths that do not contain any repeating locations) only. To this end, we adapt

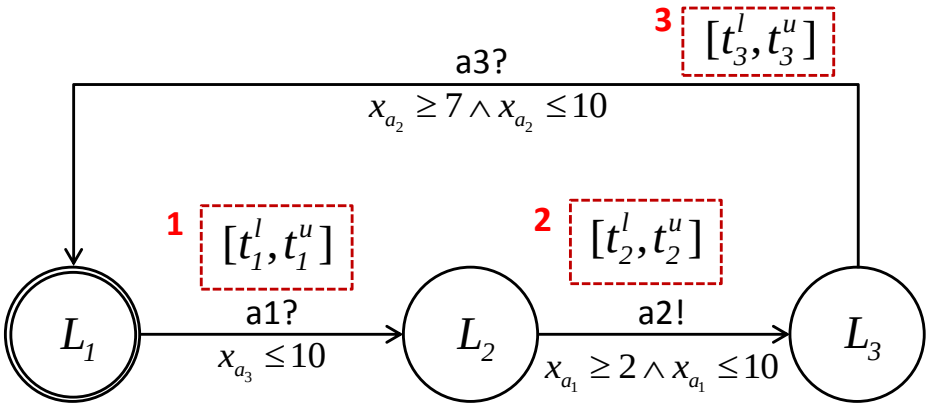


Figure 34: Model 3 with variable assignment (Cyclic Pattern)

the Floyd-Warshall algorithm. The Floyd-Warshall algorithm computes the minimum and maximum timing intervals between the entering of locations L_i and L_j in a model with n locations, denoted by $D^{\mathsf{min}}(L_i, L_j, n)$ and $D^{\mathsf{max}}(L_i, L_j, n)$, respectively. We cannot use the Floyd-Warshall algorithm to compute $f_{\mathsf{M_s}}$ directly, because the function represents the delay-bound between two transitions rather than locations. Instead, we define:

$$f_{\mathsf{M_s}}^{\mathsf{min}}(i,j) = D^{\mathsf{min}}(L_i^{\mathsf{post}}, L_j^{\mathsf{pre}}, n) + \nu_j^l \tag{7.1a}$$

$$f_{\mathsf{M_s}}^{\mathsf{max}}(i,j) = D^{\mathsf{max}}(L_i^{\mathsf{post}}, L_j^{\mathsf{pre}}, n) + \nu_j^u \tag{7.1b}$$

where L_i^{post} is the ending location of transition i, L_j^{pre} is the starting location of transition j, ν_j^l and ν_j^u are the lower and upper bounds of clock valuations associated with transition j. For example, Figures 35(a)-(b) show the results of applying the Floyd-Warshall algorithm to Model 3, while Figures 35(c)-(d) show the computation results of $f_{\mathsf{M_s}}$ for Model 3. Note that the values of the main diagonal entries are undefined in Figures 35(c)-(d), because the method described above is only applicable for transitions connected with simple paths.

Now, we generalize the computation of $f_{\mathsf{M_s}}$ for paths with cycles (*i.e.,* non-simple paths).

Lemma 3. *The min/max delay-bound over a (non-simple) path p in a platform-independent*

(a)

$D^{\min}$	L_1	L_2	L_3
L_1	0	0	2
L_2	9	0	2
L_3	7	7	0

(b)

$D^{\max}$	L_1	L_2	L_3
L_1	0	10	20
L_2	20	0	10
L_3	10	20	0

(c)

$f^{\min}_{M_s}$	1	2	3
1	-	2	9
2	7	-	7
3	0	2	-

(d)

$f^{\max}_{M_s}$	1	2	3
1	-	10	20
2	20	-	10
3	10	20	-

Figure 35: The results of applying the Floyd-Warshall algorithm and computing $f_{\mathsf{M_s}}$ for Model 3

model $\mathsf{M_s}$*, denoted by* $f_{\mathsf{M_s}}(p)$*, is the summation of the min/max delay-bounds of all simple paths that comprise* p*.*

Proof. Suppose p consists of a series of transitions $t_i \ldots t_j$, where t_i is the starting transition of p, and t_j is the ending transition of p. Let $t^p_1 \ldots t^p_k$ be the ending (and starting) transitions in between t_i and t_j that comprises all simple paths $\in p$; that is, $p = t_i \ldots t^p_1 \ldots t^p_2 \ldots t^p_k \ldots t_j$ with a set of simple paths $t_i \ldots t^p_1$, and $t^p_1 \ldots t^p_2, \ldots$, and $t^p_{k-1} \ldots t^p_k$, and $t^p_k \ldots t_j$. In an event-clock automaton, the min/max interval of a particular simple path can be calculated independently of the min/max interval of its adjacent simple paths (*i.e.,* the min/max interval of a prior simple path immediately followed by this simple path, and the post simple path immediately following this simple path) using an event-recording clock that is reset to zero upon taking a starting transition of a simple path. Therefore, the minimum interval of $f_{\mathsf{M_s}}(p)$ is calculated as follows: $f^{\mathsf{min}}_{\mathsf{M_s}}(p) = f^{\mathsf{min}}_{\mathsf{M_s}}(t_i, t^p_1) + f^{\mathsf{min}}_{\mathsf{M_s}}(t^p_1, t^p_2) + \ldots + f^{\mathsf{min}}_{\mathsf{M_s}}(t^p_{k-1}, t^p_k) + f^{\mathsf{min}}_{\mathsf{M_s}}(t^p_k, t_j)$; and the maximum interval of $f_{\mathsf{M_s}}(p)$ is also similarly calculated. □

Note that Lemma 3 is only applicable to event-clock automata where clocks reset on every transition, which is sufficient for this paper. The computation for more general cases (*e.g.,* clocks reset in arbitrary transitions) was studied in [31] using the reachability graph, but that algorithm may not terminate in the presence of cycles.

Example 7.4.1. Consider a path p in Model 3 that starts with the transition t_1 and ends with t_3 after repeating two cycles; that is, p=t_1,t_2,t_3,t_1,t_2,t_3. The path p consists of the following simple paths: p_1=t_1,t_2,t_3; p_2=t_3,t_1,t_2; and p_3=t_2,t_3. From Figures 35(c)-(d), we know that $f_{\mathsf{M_s}}(\mathit{1},\mathit{3})$=[9,20], $f_{\mathsf{M_s}}(\mathit{3},\mathit{2})$=[2,20], and $f_{\mathsf{M_s}}(\mathit{2},\mathit{3})$=[7,10]. Based on Lemma 3, we obtain that $f^{\mathsf{min}}_{\mathsf{M_s}}(p)$=$f^{\mathsf{min}}_{\mathsf{M_s}}(\mathit{1},\mathit{3})$+$f^{\mathsf{min}}_{\mathsf{M_s}}(\mathit{3},\mathit{2})$+$f^{\mathsf{min}}_{\mathsf{M_s}}(\mathit{2},\mathit{3})$ and $f^{\mathsf{max}}_{\mathsf{M_s}}(p)$=$f^{\mathsf{max}}_{\mathsf{M_s}}(\mathit{1},\mathit{3})$+$f^{\mathsf{max}}_{\mathsf{M_s}}(\mathit{3},\mathit{2})$+ $f^{\mathsf{max}}_{\mathsf{M_s}}(\mathit{2},\mathit{3})$. Thus, we have $f_{\mathsf{M_s}}(p)$=[18,50]. □

7.4.2. Computing f_{IMP}

In the following, we describe how to compute the function f_{IMP} based on the platform-processing delay P and the code-level delay f_{SOF}.

Definition 6 (*Platform-Processing Delay*)**.** For any I/O event $a_k \in \Sigma$, the platform-processing delay $\mathsf{P}(a_k) = [\delta^{\mathsf{min}}_k, \delta^{\mathsf{max}}_k]$, where δ^{min}_k and δ^{max}_k are the minimum and maximum times needed for the platform to process the event a_k, respectively.

The platform-processing delay characterizes the min/max delays consumed by a given platform (independently of the code-level delays) to process each I/O event. In Figure 32, this delay is considered as the input delay consumed over the information flow from m to i, and the output delay consumed over the information flow from o to c.

Example 7.4.2. There are three I/O events $\{a_1?, a_2!, a_3?\}$ in Model 1 shown in Figure 31. Suppose the platform-processing delay is given by P={$\mathsf{P}(a_1)$=[1,2],$\mathsf{P}(a_2)$=[3,4],$\mathsf{P}(a_3)$=[2,5]} That is, it takes the platform at least 1 and at most 2 time-units from the moment it reads the input event a_1 (mapped to m variable in Figure 32) from the environment at the *mc*-boundary until the moment the code reads the processed input event (mapped to i variable) at the *io*-boundary. Similarly, it takes the platform at least 3 and at most 4 time-units from the moment the code writes the output event a_2 (mapped to o variable) at the *io*-boundary

until the moment the platform writes the processed output event (mapped to c variable) to the environment at the *mc*-boundary.

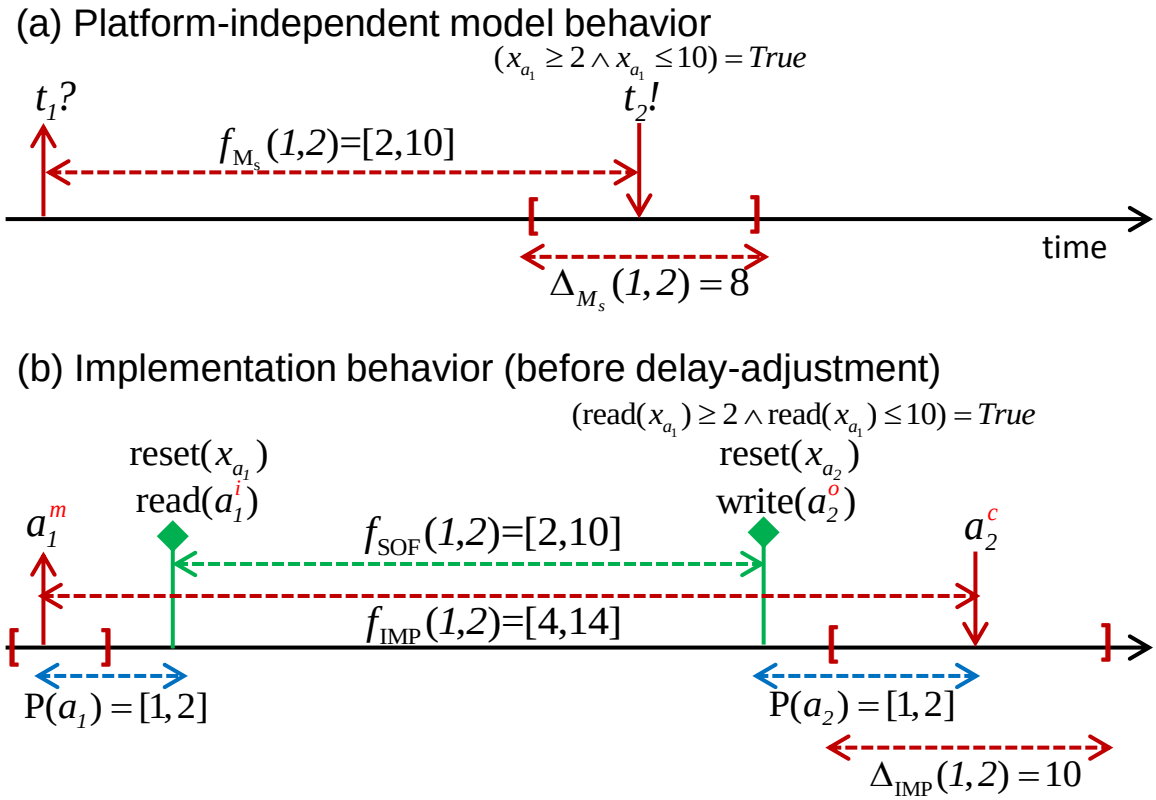


Figure 36: The timed behavior comparison between the platform-independent model (M_s) and the implementation ($P(a_1)$=[1,2] and $P(a_2)$=[1,2]); the arrows imply the events of the *mc*-boundary, while the diamond polls imply the events of the *io*-boundary

Now, we explain the relation between the system implementation delay f_{IMP}, the code-level delay f_{SOF}, and the platform-processing delay P. For example, Figure 36 illustrates how the platform-independent model and the implementation behave differently when processing a pair of the *input* transition 1 (t_1) and the *output* transition 2 (t_2) of Model 1. In the system model, t_2 is taken (lower-direction arrow) in between 2 and 10 time-units since t_1 has been taken (upper-direction arrow); therefore, $f_{M_s}(1,2)$=[2,10]. In the implementation, the delay-bound $f_{IMP}(1,2)$ may differ from $f_{M_s}(1,2)$ if the same timing parameters (T_s) of M_s is used to implement the code-level delay (f_{SOF}). Suppose the code is generated using T_s (*i.e.,* 2 and 10), and this code is to be integrated with a platform that has the

processing delay: P=[1,2] (assuming that the same min/max bound is applied to all I/O events). Assume that the code interacts with the platform through a set of primitives: *read* primitive to read the processed input values from the platform or to read current clock values; *reset* primitive to set the clock values to zero; *write* primitive to write the output values to the platform. Note that the time instances when these primitives are called by the code are different from the times when the corresponding I/O occurs in the environment, due to the platform delays.

This implemented system behaves as follows: when the input (denoted as a_1^m) associated with the transition t_1 of $\mathsf{M_s}$ is generated from the environment, (1) the platform reads a_1^m first at the *mc*-boundary (red upper arrow); (2) the code reads the processed input (denoted as a_1^i) at the *io*-boundary sometime after, in between 1 and 2 time-units (first green diamond poll) due to $\mathsf{P}(a_1)$, and reset the associated clock (x_{a_1}); (3) the code produces the output (denoted as a_2^o) at the *io*-boundary (second diamond poll) in between 2 and 10 time-units after reading the previous input (a_1^i); (4) the platform processes and writes the output (denoted as a_2^c) to the environment at the *mc*-boundary (red lower arrow) in between 1 and 2 time-units due to $\mathsf{P}(a_2)$. Intuitively, $f_{\mathsf{IMP}}(1, 2)$ will be larger than $f_{\mathsf{M_s}}(1, 2)$ (i.e., $f_{\mathsf{IMP}} \notin f_{\mathsf{M_s}}$) in this case; this deviation occurs since the I/O information flow and P have not been compensated in implementing the code-level delay (f_{SOF}) (*i.e.,* the code-level delay should have been *shrinked* for the compensation in this case).

In general, f_{IMP} can be larger or smaller than $f_{\mathsf{M_s}}$ depending on the event type and the amount of the platform-processing delay and the dependency among different transitions; the four possible I/O patterns and their implementation behavior are illustrated in Figure 37. Therefore, it is problematic to generate the code using the same parameters (T_{s}) of $\mathsf{M_s}$. Instead, we want to find a new timing parameter assignment (T_{c}) that can be used for the code so that the delay-bound inclusion constraint holds.

In order to find T_{c}, we derive the equation of f_{IMP} by separating two parts: we consider the part that constitutes the platform-processing delay (P) as *known* parameters, while we

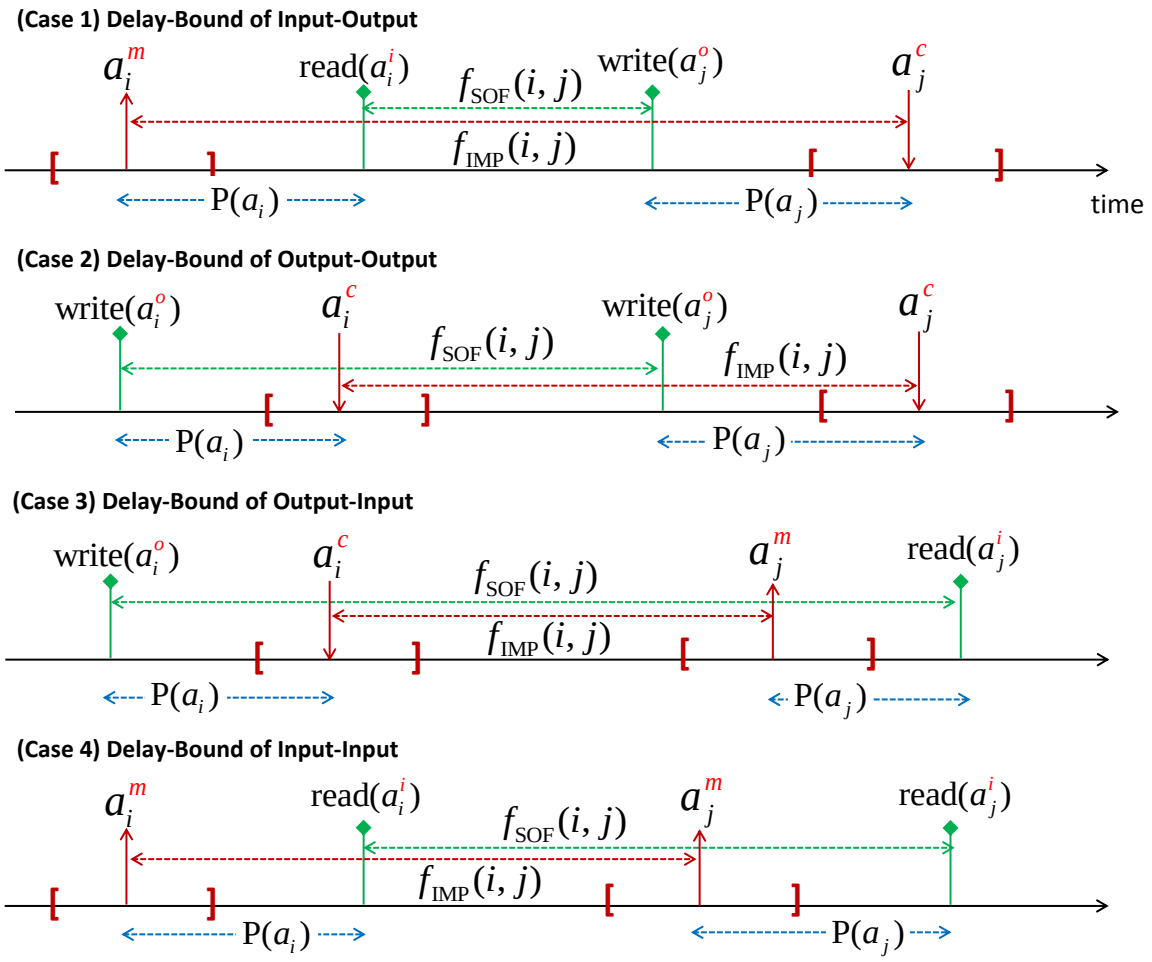


Figure 37: The four cases of the delay bound computation at the implementation level

leave the part that constitutes the code-level delay (f_{SOF}) as *unknown* variables; now, f_{IMP} for any pair of I/O events can be calculated as follows:

Each event (i, j) occurring at the *mc*-boundary is either input or output, hence there are four combinations for a pair of events that can be calculated as follows:

- Case 1: i is an input event and j is an output event:

$$[f_{\mathsf{SOF}}^{\mathsf{min}}(i,j)+(\mathsf{P}^{\mathsf{min}}(a_j)+\mathsf{P}^{\mathsf{min}}(a_i)),\ f_{\mathsf{SOF}}^{\mathsf{max}}(i,j)+(\mathsf{P}^{\mathsf{max}}(a_j)+\mathsf{P}^{\mathsf{max}}(a_i))]$$

- Case 2: i is an output event and j is an output event:

$$[f_{\mathsf{SOF}}^{\mathsf{min}}(i,j)+(\mathsf{P}^{\mathsf{min}}(a_j)-\mathsf{P}^{\mathsf{max}}(a_i)),\ f_{\mathsf{SOF}}^{\mathsf{max}}(i,j)+(\mathsf{P}^{\mathsf{max}}(a_j)-\mathsf{P}^{\mathsf{min}}(a_i))]$$

- Case 3: i is an output event and j is an input event:

$$[f^{\mathsf{min}}_{\mathsf{SOF}}(i,j) - (\mathsf{P}^{\mathsf{max}}(a_j) + \mathsf{P}^{\mathsf{max}}(a_i)),\ f^{\mathsf{max}}_{\mathsf{SOF}}(i,j) - (\mathsf{P}^{\mathsf{min}}(a_j) + \mathsf{P}^{\mathsf{min}}(a_i))]$$

- Case 4: i is an input event and j is an input event:

$$[f^{\mathsf{min}}_{\mathsf{SOF}}(i,j) - (\mathsf{P}^{\mathsf{max}}(a_j) - \mathsf{P}^{\mathsf{min}}(a_i)),\ f^{\mathsf{max}}_{\mathsf{SOF}}(i,j) - (\mathsf{P}^{\mathsf{min}}(a_j) - \mathsf{P}^{\mathsf{max}}(a_i))]$$

These equations are obtained by the straightforward case analysis illustrated in Figure 37. We only show Case 1 since the rest of the equations are similarly derived.

(**Case 1: input t_i and output t_j**) Refer to the information flow of Case 1 in Figure 37. Suppose the code reads the input i at τ_i and the code writes the output j at τ_j at the *io*-boundary. The possible range of the input occurrence at the *mc*-boundary is $[\tau_i - \mathsf{P}^{\mathsf{max}}(a_i),\ \tau_i - \mathsf{P}^{\mathsf{min}}(a_i)]$ because the input at the *mc*-boundary has to occur before reading the input at the *io*-boundary. The possible range of the output occurrence at the *mc*-boundary is $[\tau_j + \mathsf{P}^{\mathsf{min}}(a_j),\ \tau_j + \mathsf{P}^{\mathsf{max}}(a_j)]$ because the output at the *mc*-boundary has to occur after writing the output at the *io*-boundary. Then, the minimum interval of these two events is $(\tau_j + \mathsf{P}^{\mathsf{min}}(a_j)) - (\tau_i - \mathsf{P}^{\mathsf{min}}(a_i))$; the maximum interval of these two events is $(\tau_j + \mathsf{P}^{\mathsf{max}}(a_j)) - (\tau_i - \mathsf{P}^{\mathsf{max}}(a_i))$. By rewriting, the min/max interval of the two event is $[\tau_j - \tau_i + \mathsf{P}^{\mathsf{min}}(a_j) + \mathsf{P}^{\mathsf{min}}(a_i),\ \tau_j - \tau_i + \mathsf{P}^{\mathsf{max}}(a_j) + \mathsf{P}^{\mathsf{max}}(a_i)]$. Note that, the term $\tau_j - \tau_i$ is the code-level delay-bound that is *unknown*; the minimum interval of the two events can be obtained by having the minimum interval of the code-level delay; hence, the term $\tau_j - \tau_i$ of the minimum interval is rewritten as $f^{\mathsf{min}}_{\mathsf{SOF}}(i,j)$ that indicates the *unknown* minimum interval of the code-level delay. The maximum interval of the two events can be obtained by having the maximum interval of the code-level delay; hence, the term $\tau_j - \tau_i$ of the maximum interval is rewritten as $f^{\mathsf{max}}_{\mathsf{SOF}}(i,j)$ that indicates the *unknown* maximum interval of the code-level delay. As a result, we obtain the final equation of $f_{\mathsf{IMP}}(i,j)$ for Case 1 as follows:

$$[f^{\mathsf{min}}_{\mathsf{SOF}}(i,j) + (\ \mathsf{P}^{\mathsf{min}}(a_j) + \mathsf{P}^{\mathsf{min}}(a_i)),\ f^{\mathsf{max}}_{\mathsf{SOF}}(i,j) + (\mathsf{P}^{\mathsf{max}}(a_j) + \mathsf{P}^{\mathsf{max}}(a_i))]$$

7.5. Delay-Bound Adjustment using Integer Linear Programming

In this section, we explain how to find the unknown variables (T_{c}) that will be used to implement the code-level delay (f_{SOF}) using the integer linear programming (ILP).

7.5.1. Intuition of the Delay-Bound Adjustment

We formalize the ILP problem to find T_{c} to generate the code that can be integrated with a platform preserving the delay-bound inclusion constraint with respect to the system model ($\mathsf{M_s}$). Before explaining the details, the intuition behind this formalization is given.

Consider the timed behavior of $\mathsf{M_s}$ and the implemented system in Figure 36. In case the code is generated using T_{s}, two orthogonal aspects result in the deviation of the timed behavior between $\mathsf{M_s}$ and the implementation.

(1) the deviation of the uncertainty range: the uncertainty range of a pair of I/O transition t_i and t_j of $\mathsf{M_s}$ and the uncertainty range of a pair of the corresponding I/O events of an implemented system is defined as follows:

$$\Delta_{\mathsf{M_s}}(i,j) = f^{\mathsf{max}}_{\mathsf{M_s}}(i,j) - f^{\mathsf{min}}_{\mathsf{M_s}}(i,j) \tag{7.2a}$$

$$\Delta_{\mathsf{IMP}}(i,j) = f^{\mathsf{max}}_{\mathsf{IMP}}(i,j) - f^{\mathsf{min}}_{\mathsf{IMP}}(i,j) \tag{7.2b}$$

This range implies that the time interval where the second transition (t_j) is allowed to occur following the first transition (t_i). However, if the code is generated using the original timing parameter assignment (T_{s}), the platform-independent model uncertainty ($\Delta_{\mathsf{M_s}}$) is directly implemented as a code as well. The issue is that the chosen platform will add another uncertainty that comes from the platform-processing delay (P). For example, in Fig. 36-(b), the uncertainty range of the implementation (Δ_{IMP}) becomes 10 that is larger than the platform-independent model uncertainty ($\Delta_{\mathsf{M_s}}$) by 2; this additional amount comes from P that results in violation of the delay-bound inclusion constraint. To remedy this, we

introduce the *Shrinking* operation to adjust *either* upper-bound or lower-bound of the guard condition of the code to compensate P; this will make Δ_{IMP} fit into that of $\Delta_{\mathsf{M_s}}$ by either increasing the lower-bound or decreasing the upper-bound (*i.e.*, $\Delta_{\mathsf{IMP}}(i,j) \leq \Delta_{\mathsf{M_s}}(i,j)$). In Figure 38-(c), for example, the uncertainty range of the implementation is reduced by changing the guard condition associated with t_2 from [2,10] to [4,10] (*i.e.*, the lower bound of the original guard condition is increased by 2).

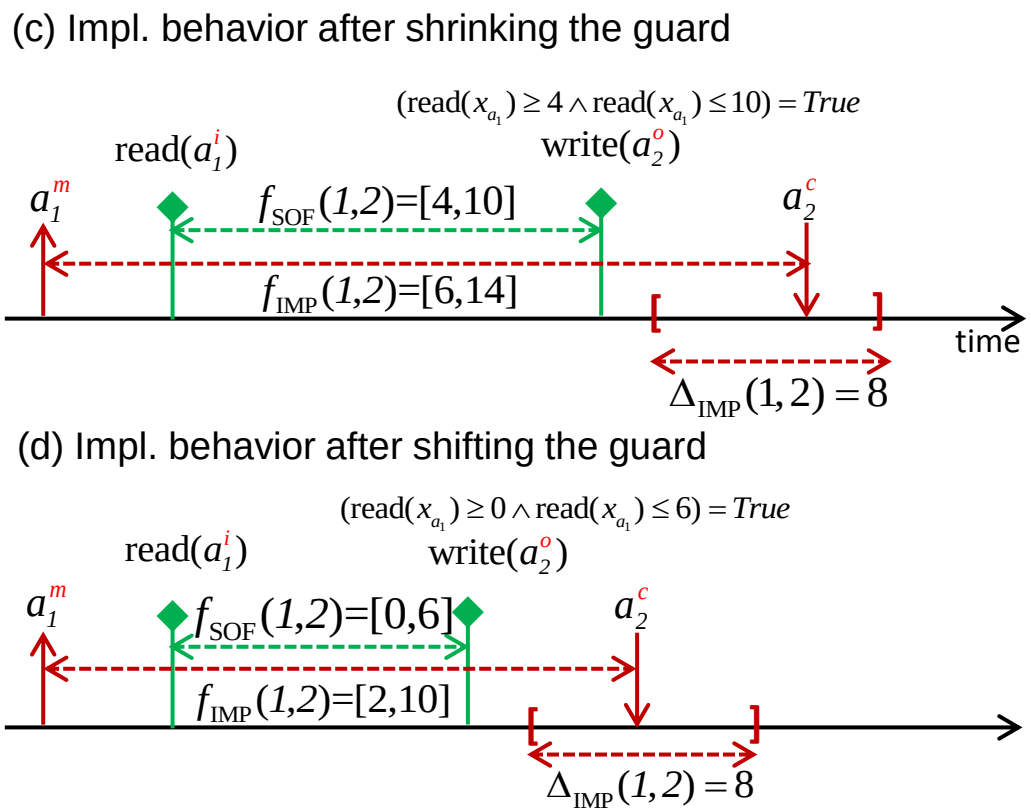


Figure 38: The illustration of the delay adjustment algorithm

(2) I/O information flow directions: In spite of the shrinking operation in Figure 38-(c), the resulting $f_{\mathsf{IMP}}(1,2)$ is [6,14] that is not included in $f_{\mathsf{M_s}}(1,2)$=[2,10] (*i.e.*, $f_{\mathsf{IMP}}(1,2) \notin f_{\mathsf{M_s}}(1,2)$). This implies that it is not sufficient to perform the *Shrinking* operation only in order to obtain T_{c}. The main reason is that the shrinking operation only makes Δ_{IMP} fit into $\Delta_{\mathsf{M_s}}$, but it does not consider the aspects originating from different combinations of I/O information flows.

Depending on the types of I/O event pairs, the delay-bound of an implementation is deviated from $\mathsf{M_s}$ in four different ways as illustrated in Figure 37. To remedy this, we introduce the *Shift* operation that moves the *relative* timing of the second event occurrence back and forth by adjusting *both* the upper and lower guard condition of the code (*c.f.,* the shrinking operation only adjusts either guard condition). In Figure 38-(d), for example, both the lower and upper guard condition of the code are decreased by 4; the result of this shift operation is that the code should now produce the output in between 0 and 6 time-units; then the implementation preserves the delay-bound inclusion constraint with respect to $\mathsf{M_s}$ (*i.e.,* $f_{\mathsf{IMP}}(1,2)$=[2,10] $\in$ $f_{\mathsf{M_s}}(1,2)$=[2,10]).

Applying the shrinking/shift operations for all possible pairs of I/O transitions is challenging since there are many different aspects intertwined with each other in a complicated pattern, such as the platform-processing delay, the I/O information flows and the dependency among different I/O transitions. We explain how these conditions are formalized in terms of the ILP so that the two operations (*i.e.,* shrinking and shifting) can be automatically performed for all possible I/O transitions to adjust the code-level delay.

7.5.2. Formalization of the ILP Problem for Acyclic Models

We first explain how the problem is formalized for acyclic models, and then explain how it can be extended to cyclic models.

Integer Variable Mapping to the Model: To formalize the ILP problem, we first map integer variables to the system model ($\mathsf{M_s}$) that express the *unknown* min/max code-level delay (f_{SOF}). Here is our mapping: two integer variables, t_k^l and t_k^u, are mapped to each transition of $\mathsf{M_s}$; these variables represent the lower and upper bounds, respectively, of the guard condition associated with the transition. In general, this mapping requires $2n$ variables if $\mathsf{M_s}$ has n transitions. For example, six variables (t_1^l,t_1^u,t_2^l,t_2^u,t_3^l,t_3^u) are used to represent Model 1. This variable mapping enables the min/max code-level delay (f_{SOF}) for any pair of I/O transitions to be represented in terms of a linear combination of variables.

We will use this representation to define constraints as follows:

Defining Linear Constraints: For all pairs of I/O transitions in $\mathsf{M_s}$, we formalize the following constraints using the linear combination of the integer variables:

- (Type 1) The minimum delay-bound of the implemented system should be equal or greater than that of $\mathsf{M_s}$ (*i.e.,* $f^{\mathsf{min}}_{\mathsf{IMP}} \geq f^{\mathsf{min}}_{\mathsf{M_s}}$);
- (Type 2) The maximum delay-bound of the implemented system should be equal or less than that of $\mathsf{M_s}$ (*i.e.,* $f^{\mathsf{max}}_{\mathsf{IMP}} \leq f^{\mathsf{max}}_{\mathsf{M_s}}$);
- (Type 3) Each min/max delay-bound of the code should be non-negative;
- (Type 4) The minimum delay-bound of the code should be equal or less than the maximum delay bound.

Type 1 and type 2 characterize the delay-bound inclusion constraints of Definition 4; we need the constraints of type 3 and type 4 to obtain only non-negative guard conditions whose minimum delay bound is less than the maximum bound.

Example 7.5.1 (Linear Constraints for Model 1)**.** Model 1 has three pairs of I/O transitions: (1,2), (2,3), (1,3), and here are the linear constraints:

- (C1a) $t^l_2 + \mathsf{P}^{\mathsf{min}}(a_2) + \mathsf{P}^{\mathsf{min}}(a_1) \geq 2$
- (C1b) $t^u_2 + \mathsf{P}^{\mathsf{max}}(a_2) + \mathsf{P}^{\mathsf{max}}(a_1) \leq 10$
- (C2a) $t^l_3 - (\mathsf{P}^{\mathsf{max}}(a_3) + \mathsf{P}^{\mathsf{max}}(a_2)) \geq 7$
- (C2b) $t^u_3 - (\mathsf{P}^{\mathsf{min}}(a_3) + \mathsf{P}^{\mathsf{min}}(a_2)) \leq 10$
- (C3a) $t^l_2 + t^l_3 - (\mathsf{P}^{\mathsf{max}}(a_3) - \mathsf{P}^{\mathsf{min}}(a_1)) \geq 9$
- (C3b) $t^u_2 + t^u_3 - (\mathsf{P}^{\mathsf{min}}(a_3) - \mathsf{P}^{\mathsf{max}}(a_1)) \leq 20$
- (C4) $t^l_1 \geq 0 \wedge t^u_1 \geq 0 \wedge t^l_2 \geq 0 \wedge t^u_2 \geq 0 \wedge t^l_3 \geq 0 \wedge t^u_3 \geq 0$

- (C5) $t_1^l \leq t_1^u \wedge t_2^l \leq t_2^u \wedge t_3^l \leq t_3^u$

Suppose a platform is characterized as P=[1,2], and the code is assumed to be generated from Model 1. In this case, the linear constraints are obtained by plugging in P as follows: (C1a) $t_2^l \geq 0$; (C1b) $t_2^u \leq 6$; (C2a) $t_3^l \geq 11$; (C2b) $t_3^u \leq 12$; (C3a) $t_2^l \geq 10$; (C3b) $t_2^u + t_3^u \leq 19$; (C4) $t_1^l \geq 0 \wedge t_1^u \geq 0 \wedge t_2^l \geq 0 \wedge t_2^u \geq 0 \wedge t_3^l \geq 0 \wedge t_3^u \geq 0$; (C5) $t_1^l \leq t_1^u \wedge t_2^l \leq t_2^u \wedge t_3^l \leq t_3^u$ □

The constraints (C1a,b), (C2a,b), (C3a,b) are relevant to the pairs of transitions (1,2), (2,3), (1,3), respectively. The constraints (C1a)-(C3a) and (C1b)-(C3b) are the type 1 and type 2 constraints, respectively; the right hand side of these constraints come from the calculated $f_{\mathsf{M_s}}$ according to Lemma 3, while the left hand side are obtained based on the calculations of f_{IMP} in Section 7.4 by replacing $f_{\mathsf{SOF}}^{\mathsf{min}}(i,j)$ and $f_{\mathsf{SOF}}^{\mathsf{max}}(i,j)$ with the linear combination of (1) the relevant integer variables that express the *unknown* code-level delay, and (2) the *known* platform-processing delay. On the other hand, the constraint (C4) and (C5) are the type 3 and type 4 constraints needed independently of the platform-processing delay (P).

The linear constraints for Model 2 are also similarly defined as follows:

Example 7.5.2 (Linear Constraints for Model 2)**.** Model 2 has four pairs of I/O transitions: (1,2), (1,3), (1,4), (3,4), and here are the linear constraints:

- (C1a) $t_2^l + \mathsf{P}^{\mathsf{min}}(a_2) + \mathsf{P}^{\mathsf{min}}(a_1) \geq 2$
- (C1b) $t_2^u + \mathsf{P}^{\mathsf{max}}(a_2) + \mathsf{P}^{\mathsf{max}}(a_1) \leq 10$
- (C2a) $t_3^l - (\mathsf{P}^{\mathsf{max}}(a_3) - \mathsf{P}^{\mathsf{min}}(a_1)) \geq 7$
- (C2b) $t_3^u - (\mathsf{P}^{\mathsf{min}}(a_3) - \mathsf{P}^{\mathsf{max}}(a_1)) \leq 10$
- (C3a) $t_3^l + t_4^l + (\mathsf{P}^{\mathsf{min}}(a_4) + \mathsf{P}^{\mathsf{min}}(a_1)) \geq 11$
- (C3b) $t_3^u + t_4^u + (\mathsf{P}^{\mathsf{max}}(a_4) + \mathsf{P}^{\mathsf{max}}(a_1)) \leq 18$
- (C4a) $t_4^l + (\mathsf{P}^{\mathsf{min}}(a_4) + \mathsf{P}^{\mathsf{min}}(a_3)) \geq 4$

- (C4b) $t_4^u + (\mathsf{P}^{\mathsf{max}}(a_4) + \mathsf{P}^{\mathsf{max}}(a_3)) \leq 8$
- (C5) $t_1^l \geq 0 \wedge t_1^u \geq 0 \wedge t_2^l \geq 0 \wedge t_2^u \geq 0 \wedge t_3^l \geq 0 \wedge t_3^u \geq 0 \wedge t_4^l \geq 0 \wedge t_4^u \geq 0$
- (C6) $t_1^l \leq t_1^u \wedge t_2^l \leq t_2^u \wedge t_3^l \leq t_3^u \wedge t_4^l \leq t_4^u$

Suppose a platform is given characterized as P=[2,4]; and the code is assumed to be generated from Model 2. The corresponding linear constraints are as follows: (C1a) $t_2^l \geq -2$; (C1b) $t_2^u \leq 2$; (C2a) $t_3^l \geq 9$; (C2b) $t_3^u \leq 8$; (C3a) $t_3^l \geq 7$; (C3b) $t_3^u + t_4^u \leq 10$; (C4a) $t_4^l \geq 0$; (C4b) $t_4^u \leq 0$; (C5) $t_1^l \geq 0 \wedge t_1^u \geq 0 \wedge t_2^l \geq 0 \wedge t_2^u \geq 0 \wedge t_3^l \geq 0 \wedge t_3^u \geq 0 \wedge t_4^l \geq 0 \wedge t_4^u \geq 0$; (C6) $t_1^l \leq t_1^u \wedge t_2^l \leq t_2^u \wedge t_3^l \leq t_3^u \wedge t_4^l \leq t_4^u$ □

Defining Objective Functions for Optimization: Our goal is to find the parameter assignments (T_{c}) for the *unknown* variables that satisfy these linear constraints. Note that there can be many possible parameter assignments that satisfy these linear constraints. For example, in Figure 38, the guard condition (lower bound: 0 and upper bound: 6) for the pair of I/O transition allows the code to write an output (a_2^o) in between 0 and 6 time-units since it has read the input (a_1^i) with the result of $f_{\mathsf{IMP}}(1, 2)$=[2,10]. Another possible assignment is (lower bound: 0, upper bound: 1) that results in $f_{\mathsf{IMP}}(1, 2)$=[2,5], which still preserves the delay-bound inclusion property with respect to Model 1.

To this end, we define optimality in choosing T_{c} by considering the following aspect: apart from the platform-processing delay (P), the code also requires some computation time for its internal processing; for example, the code computes the output based on the input according to the control law that can be a complex function. Therefore, a better solution is to find T_{c} that *maximizes* the room for the internal computation of the code. In the above example, the assignment (lower bound: 0, upper bound: 6) is a better solution than the other assignment (lower bound: 0, upper bound: 1) since the code can have more computation time before producing the second event (*i.e.,* 6 time-unit versus 1 time-unit). Such an optimal assignment can be obtained by defining the objective function that maximizes the uncertainty range of the implementation (Δ_{IMP}) as close as to that of the

system model ($\Delta_{\mathsf{M_s}}$); this will allow us to find the largest room for the code computation while preserving the linear constraints.

Suppose a system model ($\mathsf{M_s}$) has n pairs of I/O transitions. Then, we also have n uncertainty ranges for the platform-independent model ($\Delta_{\mathsf{M_s}}$) and an implemented system (Δ_{IMP}) that corresponds to each pair of I/O transitions. $\Delta_{\mathsf{M_s}}$ is known since it is calculated from the known T_{s}, but Δ_{IMP} is unknown since it is calculated from the unknown T_{c}. Let $\Delta_{\mathsf{IMP}}(k)$ be the uncertainty range of an implemented system for a particular pair (k) of I/O transitions. Then, the general form of our objective function is as follows:

$$\textbf{maximize} \quad \sum_{i=1}^{n} \Delta_{\mathsf{IMP}}(i) \tag{7.3}$$

For example, Model 1 has three uncertainty ranges that correspond to the three pairs of I/O transitions.

$$\Delta_{\mathsf{IMP}}(1,2) = (t_2^u + P^{\mathsf{max}}(a_2) + P^{\mathsf{max}}(a_1)) - (t_2^l + P^{\mathsf{min}}(a_2) + P^{\mathsf{min}}(a_1))$$
$$\Delta_{\mathsf{IMP}}(2,3) = (t_3^u - (P^{\mathsf{min}}(a_3) + P^{\mathsf{min}}(a_2))) - (t_3^l - (P^{\mathsf{max}}(a_3) + P^{\mathsf{max}}(a_2)))$$
$$\Delta_{\mathsf{IMP}}(1,3) = (t_2^u + t_3^u - (P^{\mathsf{min}}(a_3) - P^{\mathsf{max}}(a_1))) - (t_2^l + t_3^l - (P^{\mathsf{max}}(a_3) - P^{\mathsf{min}}(a_1)))$$

The following is the objective function for Model 1:

$$\textbf{maximize} \quad \Delta_{\mathsf{IMP}}(1,2) + \Delta_{\mathsf{IMP}}(2,3) + \Delta_{\mathsf{IMP}}(1,3)$$

Our solver will find the parameter assignment for the six variables (t_1^l,t_1^u,t_2^l,t_2^u,t_3^l,t_3^u) by maximizing the summation of the uncertainty range of the implementation.

Example 7.5.3 (Optimal parameter assignment of Model 1)**.** Suppose the code is to be generated from Model 1, and integrated with a platform characterized as P=[1,2]; then the optimal parameter assignment for the code is (t_1^l,t_1^u,t_2^l,t_2^u,t_3^l,t_3^u) = (0, INF, 0, 6, 11, 12), where INF implies the absence of the upper bound. Consider another P=[1,3]; then there is no possible parameter assignment for the code generation. □

In Example 7.5.3, the solver finds *no* feasible parameter assignment for the code in case the platform with P=[1,3] has to be used. This implies that no code can be generated from Model 1 for this platform by preserving the delay-bound inclusion constraint. However, suppose another platform P=[1,2] is chosen whose maximum I/O processing delay is 1 time-unit faster than the previous platform. In this case, the solver can find the optimal parameter assignment that can be used to generate the code running on this platform.

The objective function for Model 2 is also similarly defined. Model 2 has four uncertainty ranges that correspond to the four pairs of I/O transitions.

$$\Delta_{\mathsf{IMP}}(1,2) = (t_2^u + P^{\mathsf{max}}(a_2) + P^{\mathsf{max}}(a_1)) - (t_2^l + P^{\mathsf{min}}(a_2) + P^{\mathsf{min}}(a_1))$$
$$\Delta_{\mathsf{IMP}}(1,3) = (t_3^u - (P^{\mathsf{min}}(a_3) - P^{\mathsf{max}}(a_1))) - (t_3^l - (P^{\mathsf{max}}(a_3) - P^{\mathsf{min}}(a_1)))$$
$$\Delta_{\mathsf{IMP}}(1,4) = (t_3^u + t_4^u + (P^{\mathsf{max}}(a_4) + P^{\mathsf{max}}(a_1))) - (t_3^l + t_4^l + (P^{\mathsf{min}}(a_4) + P^{\mathsf{min}}(a_1)))$$
$$\Delta_{\mathsf{IMP}}(3,4) = (t_4^u + (P^{\mathsf{max}}(a_4) + P^{\mathsf{max}}(a_3))) - (t_4^l + (P^{\mathsf{min}}(a_4) + P^{\mathsf{min}}(a_3)))$$

The following is the objective function for Model 2:

$$\textbf{maximize } \Delta_{\mathsf{IMP}}(1,2) + \Delta_{\mathsf{IMP}}(1,3) + \Delta_{\mathsf{IMP}}(1,4) + \Delta_{\mathsf{IMP}}(3,4)$$

We believe that one can define more refined notions of optimality in terms of the internal computation time for the code. One possible notion is to give more internal computation time for a particular I/O transition than the other in case adjusting both Δ_{IMP} equally conforms linear constraints. Accommodating such refined optimality will result in more complex objective functions than Equation 7.3. Exploring all possible notions of optimality and comparing them to each other is out of the scope of this dissertation.

7.5.3. Handling Cyclic Models

Note that, if a platform-independent model contains cycles, there can be an infinite number of paths between a pair of transitions. For example, Model 3 contains a cycle closed by transition 3. Consider a pair of transitions (t_1, t_3); there are an infinite number of paths,

corresponding to the number of times the cycle has been followed. Yet, we want to preserve the delay-bound inclusion property for each of these paths. To this end, we show that it is sufficient to consider only the simple paths through the model to guarantee that the delay-bound inclusion property for arbitrary paths, as follows:

Theorem 2. *Given a platform processing delay* P *and a platform-independent model* $\mathsf{M_s}$*, if* $f_{\mathsf{IMP}}(i,j) \in f_{\mathsf{M_s}}(i,j)$ *for every pair of transitions* t_i *and* t_j*, then* $f_{\mathsf{IMP}}(p) \in f_{\mathsf{M_s}}(p)$ *for any path* p *that starts with* t_i *and ends with* t_j*.*

Proof. (Proof by contradiction) Suppose $\exists\ p$ such that $f_{\mathsf{IMP}}(p) \notin f_{\mathsf{M_s}}(p)$, where $p = t_i \ldots t_1^p \ldots t_2^p \ldots t_k^p \ldots t_j$ with a set of simple paths $t_i \ldots t_1^p$, and $t_1^p \ldots t_2^p$, ..., and $t_{k-1}^p \ldots t_k^p$, and $t_k^p \ldots t_j$. By Lemma 3, $f_{\mathsf{M_s}}(p)$ is calculated as follows:

$$f_{\mathsf{M_s}}^{\mathsf{min}}(p) = f_{\mathsf{M_s}}^{\mathsf{min}}(t_i, t_1^p) + f_{\mathsf{M_s}}^{\mathsf{min}}(t_1^p, t_2^p) + \ldots + f_{\mathsf{M_s}}^{\mathsf{min}}(t_{k-1}^p, t_k^p) + f_{\mathsf{M_s}}^{\mathsf{min}}(t_k^p, t_j)$$

$$f_{\mathsf{M_s}}^{\mathsf{max}}(p) = f_{\mathsf{M_s}}^{\mathsf{max}}(t_i, t_1^p) + f_{\mathsf{M_s}}^{\mathsf{max}}(t_1^p, t_2^p) + \ldots + f_{\mathsf{M_s}}^{\mathsf{max}}(t_{k-1}^p, t_k^p) + f_{\mathsf{M_s}}^{\mathsf{max}}(t_k^p, t_j)$$

Since $f_{\mathsf{IMP}}(p) \notin f_{\mathsf{M_s}}(p)$, either of the following cases must be satisfied:

$$\text{(Case 1) } f_{\mathsf{IMP}}^{\mathsf{min}}(p) < f_{\mathsf{M_s}}^{\mathsf{min}}(p)$$

$$\text{(Case 2) } f_{\mathsf{IMP}}^{\mathsf{max}}(p) > f_{\mathsf{M_s}}^{\mathsf{max}}(p)$$

Suppose (Case 1) holds; then,

$$f_{\mathsf{IMP}}^{\mathsf{min}}(t_i, t_1^p) + f_{\mathsf{IMP}}^{\mathsf{min}}(t_1^p, t_2^p) + \ldots + f_{\mathsf{IMP}}^{\mathsf{min}}(t_{k-1}^p, t_k^p) + f_{\mathsf{IMP}}^{\mathsf{min}}(t_k^p, t_j) <$$
$$f_{\mathsf{M_s}}^{\mathsf{min}}(t_i, t_1^p) + f_{\mathsf{M_s}}^{\mathsf{min}}(t_1^p, t_2^p) + \ldots + f_{\mathsf{M_s}}^{\mathsf{min}}(t_{k-1}^p, t_k^p) + f_{\mathsf{M_s}}^{\mathsf{min}}(t_k^p, t_j).$$

However, this is not possible due to the assumption that $f_{\mathsf{IMP}}^{\mathsf{min}}(t_i, t_j) \geq f_{\mathsf{M_s}}^{\mathsf{min}}(t_i, t_j)$ for $\forall\ t_i, t_j$.

Suppose (Case 2) holds; then,

$$f_{\mathsf{IMP}}^{\mathsf{max}}(t_i, t_1^p) + f_{\mathsf{IMP}}^{\mathsf{max}}(t_1^p, t_2^p) + \ldots + f_{\mathsf{IMP}}^{\mathsf{max}}(t_{k-1}^p, t_k^p) + f_{\mathsf{IMP}}^{\mathsf{max}}(t_k^p, t_j) >$$
$$f_{\mathsf{M_s}}^{\mathsf{max}}(t_i, t_1^p) + f_{\mathsf{M_s}}^{\mathsf{max}}(t_1^p, t_2^p) + \ldots + f_{\mathsf{M_s}}^{\mathsf{max}}(t_{k-1}^p, t_k^p) + f_{\mathsf{M_s}}^{\mathsf{max}}(t_k^p, t_j).$$

However, this is not possible due to the assumption that $f_{\mathsf{IMP}}^{\mathsf{max}}(t_i, t_j) \leq f_{\mathsf{M_s}}^{\mathsf{max}}(t_i, t_j)$ for $\forall\ t_i$,

t_j.

This contradicts the fact that either (Case 1) or (Case 2) must be satisfied. □

Example 7.5.4 (Linear Constraints for Model 3)**.** According to Theorem 2, we only need to consider a finite number of simple paths in Model 3, and the corresponding linear constraints are listed below:

- (C1a) $t_2^l + \mathsf{P}^{\mathsf{min}}(a_2) + \mathsf{P}^{\mathsf{min}}(a_1) \geq 2$
- (C1b) $t_2^u + \mathsf{P}^{\mathsf{max}}(a_2) + \mathsf{P}^{\mathsf{max}}(a_1) \leq 10$
- (C2a) $t_2^l + t_3^l - (\mathsf{P}^{\mathsf{max}}(a_3) - \mathsf{P}^{\mathsf{min}}(a_1)) \geq 9$
- (C2b) $t_2^u + t_3^u - (\mathsf{P}^{\mathsf{min}}(a_3) - \mathsf{P}^{\mathsf{max}}(a_1)) \leq 20$
- (C3a) $t_3^l + t_1^l - (\mathsf{P}^{\mathsf{max}}(a_2) + \mathsf{P}^{\mathsf{max}}(a_1)) \geq 7$
- (C3b) $t_3^u + t_1^u - (\mathsf{P}^{\mathsf{min}}(a_2) + \mathsf{P}^{\mathsf{min}}(a_1)) \leq 20$
- (C4a) $t_3^l - (\mathsf{P}^{\mathsf{max}}(a_3) + \mathsf{P}^{\mathsf{max}}(a_2)) \geq 7$
- (C4b) $t_3^u - (\mathsf{P}^{\mathsf{min}}(a_3) + \mathsf{P}^{\mathsf{min}}(a_2)) \leq 10$
- (C5a) $t_3^l - (\mathsf{P}^{\mathsf{max}}(a_1) - \mathsf{P}^{\mathsf{min}}(a_3)) \geq 0$
- (C5b) $t_3^u - (\mathsf{P}^{\mathsf{min}}(a_1) - \mathsf{P}^{\mathsf{min}}(a_3)) \leq 10$
- (C6a) $t_1^l + t_2^l + (\mathsf{P}^{\mathsf{min}}(a_3) + \mathsf{P}^{\mathsf{min}}(a_2)) \geq 2$
- (C6b) $t_1^u + t_2^u + (\mathsf{P}^{\mathsf{max}}(a_3) + \mathsf{P}^{\mathsf{max}}(a_2)) \leq 20$
- (C7) $t_1^l \geq 0 \wedge t_1^u \geq 0 \wedge t_2^l \geq 0 \wedge t_2^u \geq 0 \wedge t_3^l \geq 0 \wedge t_3^u \geq 0$
- (C8) $t_1^l \leq t_1^u \wedge t_2^l \leq t_2^u \wedge t_3^l \leq t_3^u$

□

Now, the objective function can be similarly defined as Model 1 and Model 2; the following is the list of uncertainty ranges that correspond to the pairs of I/O transitions in Model 3:

$$\Delta_{\mathsf{IMP}}(1,2) = (t_2^u + P^{\mathsf{max}}(a_2) + P^{\mathsf{max}}(a_1)) - (t_2^l + P^{\mathsf{min}}(a_2) + P^{\mathsf{min}}(a_1))$$

$$\Delta_{\mathsf{IMP}}(1,3) = (t_2^u + t_3^u - (P^{\mathsf{min}}(a_3) - P^{\mathsf{max}}(a_1))) - (t_2^l + t_3^l - (P^{\mathsf{max}}(a_3) - P^{\mathsf{min}}(a_1)))$$

$$\Delta_{\mathsf{IMP}}(2,1) = (t_3^u + t_1^u - (P^{\mathsf{min}}(a_2) + P^{\mathsf{min}}(a_1))) - (t_3^l + t_1^l - (P^{\mathsf{max}}(a_2) + P^{\mathsf{max}}(a_1)))$$

$$\Delta_{\mathsf{IMP}}(2,3) = (t_3^u - (P^{\mathsf{min}}(a_3) + P^{\mathsf{min}}(a_2))) - (t_3^l - (P^{\mathsf{max}}(a_3) + P^{\mathsf{max}}(a_2)))$$

$$\Delta_{\mathsf{IMP}}(3,1) = (t_3^u - (P^{\mathsf{min}}(a_1) - P^{\mathsf{max}}(a_3))) - (t_3^l - (P^{\mathsf{max}}(a_1) - P^{\mathsf{min}}(a_3)))$$

$$\Delta_{\mathsf{IMP}}(3,2) = (t_1^u + t_2^u + (P^{\mathsf{max}}(a_3) + P^{\mathsf{max}}(a_2))) - (t_1^l + t_2^l + (P^{\mathsf{min}}(a_3) + P^{\mathsf{min}}(a_2)))$$

The following is the objective function for Model 3:

$$\textbf{maximize} \quad \Delta_{\mathsf{IMP}}(1,2) + \Delta_{\mathsf{IMP}}(1,3) + \Delta_{\mathsf{IMP}}(2,1) + \Delta_{\mathsf{IMP}}(2,3) + \Delta_{\mathsf{IMP}}(3,1) + \Delta_{\mathsf{IMP}}(3,2)$$

Example 7.5.5 (Optimal parameter assignment of Model 3)**.** Suppose the code is to be generated from Model 3, and integrated with a platform characterized as P=[0,1]; then the optimal parameter assignment for the code is (t_1^l,t_1^u,t_2^l,t_2^u,t_3^l,t_3^u) = (0, 10, 2, 8, 9, 9). Consider another P=[1,2]; then there is no possible parameter assignment for the code generation. □

7.6. Case Study: Infusion Pump Systems

In order to show the applicability of the proposed approach, we generate the code for the Baxter II Syringe PCA infusion pump platform, and validated several delay-bound inclusion constraints (the equipment setup is shown in Appendix). Here are the details of the case study:

(1) Modeling and verification for the infusion pump systems: Model 4 in Figure 39 is the system model ($\mathsf{M_s}$) used for this case study, and its informal semantics and the implication in the actual PCA pump implementation are as follows: in the initial location (L_1), the pump waits for the patient's bolus request (in the implementation, the input

$mBolusReq$ is generated by pressing a button). However, the pump shall not accept a bolus request occurring earlier than 5000 ms [1] since the previous infusion has been either normally finished by providing the expected amount of drugs or abnormally finished due to the alarming condition (*i.e.,* those premature bolus requests should be ignored). Otherwise, the pump shall process any valid bolus request by taking the transition from L_1 to L_2. In L_2, the pump shall initiate the bolus infusion in between 150 ms and 470 ms by taking the transition from L_2 to L_3 (in the implementation, the output $cStartInfusion$ is produced by rotating the pump motor at a calculated speed). Any bolus infusion will finish in between 300 ms and 750 ms by taking the transition from L_3 to L_1 (in the implementation, the output $cStopInfusion$ is produced by stopping the pump motor rotation), unless it encounters the empty syringe condition. During the infusion, if the empty syringe is detected (in the implementation, the input $mEmptySyringe$ is detected from the switch sensor that is pressed when the syringe hits the bottom), the pump takes a transition from L_3 to L_4 to prepare alarms. In L_4, the pump raises an alarm in between 200 ms and 500 ms by taking the transition from L_4 to L_1 (in the implementation, the output $cAlarm$ is produced by providing signals to a buzzer and alarm LEDs).

Here are three timing constraints considered in this case study:

- (REQ1) *When a patient requests a bolus, the bolus infusion should start in between 150 ms and 470 ms*;
- (REQ2) *The bolus infusion should be active at least 300 ms, and at most 750 ms*;
- (REQ3) *When a patient requests a bolus, the bolus infusion should finish in between 450 ms and 1220 ms in the absense of the empty reservoir alarm.*

(2) Obtaining the platform-processing delay (P): We measured the platform-processing delay (P) of each input and output of the Baxter PCA pump according to the measurement method introduced in [34], and here is the brief explanation:

[1] We assume 1 time unit in the model is mapped to 1 ms in the implementation

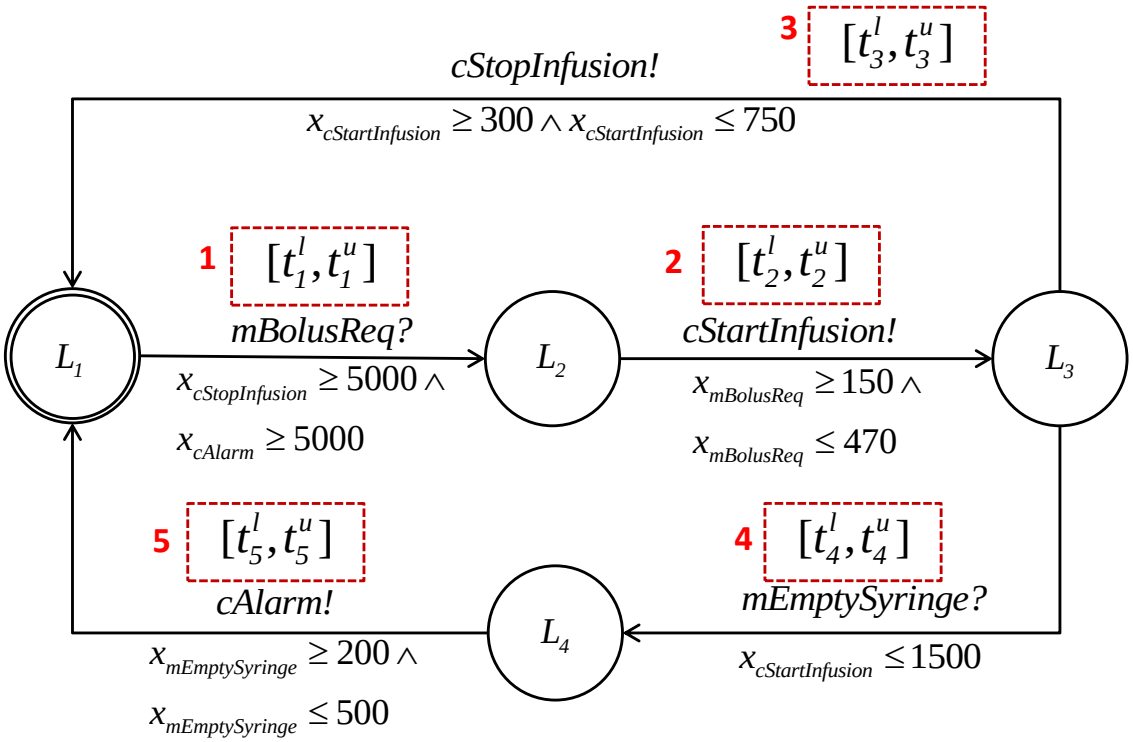


Figure 39: Model 4 with variable assignment (Simplified GPCA model)

To measure the input processing delay, we obtained the timestamp (t_m) when the input is generated from the environment (*mc*-boundary), and obtained another timestamp (t_i) when the code reads the processed input (*io*-boundary); the input processing delay is calculated by subtracting t_m from t_i. To measure the output processing delay, we obtained the timestamp (t_o) when the output is generated from the code (*io*-boundary), and another timestamp (t_c) when the platform actually writes the processed output to the environment (*mc*-boundary); the output processing delay is calculated by subtracting t_o from t_c.

We obtained each timestamp using the oscilloscope that captures the signal changes of the sensors and actuators of the PCA pump. The measurement has been performed 20 times to measure each I/O delay (two inputs and three outputs), and the min/max bound and the average of the measured platform processing delay are summarized in Table. 11.

From this measurement, we characterize the platform-processing delay of the Baxter PCA pump platform as P={P(*mBolusReq*)=[50,151],P(*cStartInfusion*)=[100,303],

I/O Event	Type	P^{Δ}_{min} (ms)	P^{Δ}_{max} (ms)	Avg. (ms)
mBolusReq?	Input	50	151	105.27
cStartInfusion!	Output	100	303	225.97
cStopInfusion!	Output	98	302	213.64
mEmptySyringe?	Input	104	200	142.8
cAlarm!	Output	298	303	300.05

Table 11: The measured platform processing delay of the Baxter II syringe pump platform

$\mathsf{P}(cStopInfusion)$=[98,302], $\mathsf{P}(mEmptySyringe)$=[104,200], $\mathsf{P}(cAlarm)$=[298,303]}.

(3) Formalizing the ILP problem: According to the procedure explained in Section 7.5, we formalized the ILP problem to find the optimal parameter assignment to generate the code. Firstly, two integer variables are mapped to each I/O transition of the model, which are unknown variables that need to be solved to implement the code-level delay; a total of 10 integer variables are mapped to the model (t^l_1,t^u_1,t^l_2,t^u_2,t^l_3,t^u_3,t^l_4,t^u_4,t^l_5,t^u_5). Secondly, $f_{\mathsf{M_s}}$ has been calculated for the simple paths between all pairs of I/O transitions; a total of 20 pairs of $f_{\mathsf{M_s}}$ are calculated. Thirdly, based on the calculated $f_{\mathsf{M_s}}$, 42 linear constraints have been formalized that need to be satisfied for delay-bound inclusion constraint. Finally, the objective function is defined with the given P, and the solver finds the optimal parameter assignment as summarized in Table 12. This parameter assignment is used to implement the guard conditions in generating the code that will be executing on the given platform.

	t^l_1	t^u_1	t^l_2	t^u_2	t^l_3	t^u_3	t^l_4	t^u_4	t^l_5	t^u_5
T^{M_s}	5000	INF	150	470	300	750	0	1500	200	500
T^{M_c}	5454	INF	0	16	505	548	503	1704	0	202

Table 12: The parameter assignment (T_{s}) of $\mathsf{M_s}$ and the parameter assignment (T_{c}) of $\mathsf{M_c}$

(4) Validating the optimal solution: The goal of the validation is to compare how well the two different codes generated from the platform-independent model ($\mathsf{M_s}$) and the software model ($\mathsf{M_c}$) preserve the delay-bound inclusion constraint. We generated the two

different codes from each model using the TIMES tool [14]. In order to obtain the implemented system, the generated codes are interfaced with the same platform-specific I/O processing routines whose min/max processing delay-bound is measured as shown in Table 11.

In order to validate the delay-bound inclusion constraint, we measured the delay between a pair of I/O events associated with each timing constraint at the *mc*-boundary. For example, in order to validate REQ1, we obtained the timestamp ($t_{mBolusReq}$) for the signal change occurring at the bolus request button (its signal changes from 5 volts to 0 volts if the button is pressed by the patient); and we obtained another timestamp ($t_{cStartInfusion}$) for the signal change occurring at the pump motor (its signal changes from 0 volts to 5 volts if the motor starts rotating); the delay between the two signal changes are measured by subtracting $t_{mBolusReq}$ from $t_{cStartInfusion}$.

For each code, the delay measurement has been performed 20 times for each timing constraint; the trends of the delay-bound for each constraint throughout the testing are shown in Figure 40 41 42; their min/max/average delays are summarized in the two separate f_{IMP} columns of Table 13; at the same time, $f_{\mathsf{M_s}}$ is also shown in Table 13 as well for comparison.

REQ	f^{M_s}		$f^{IMP}(T^{M_s}, P^{\Delta})$			$f^{IMP}(T^{M_c}, P^{\Delta})$		
	Min (ms)	Max(ms)	Min(ms)	Max(ms)	Avg.(ms)	Min(ms)	Max(ms)	Avg.(ms)
REQ1	150	470	603	843	695.8	155	459	283.4
REQ2	300	750	603	907	845	595	619	606.6
REQ3	450	1220	1400	1710	1537	747	1070	887.55

Table 13: The validation result of the delay-bound inclusion constraint of the GPCA reference implementation

Analysis: Meeting the delay-bound inclusion constraint means that all the measured implementation delays of each timing constraint should be within the time interval allowed by the platform-independent model ($\mathsf{M_s}$) (*i.e.*, $f^{\mathsf{min}}_{\mathsf{IMP}} \geq f^{\mathsf{min}}_{\mathsf{M_s}}$ and $f^{\mathsf{max}}_{\mathsf{IMP}} \leq f^{\mathsf{max}}_{\mathsf{M_s}}$). If at least one delay measurement is out of the interval allowed by $\mathsf{M_s}$, the implementation does not

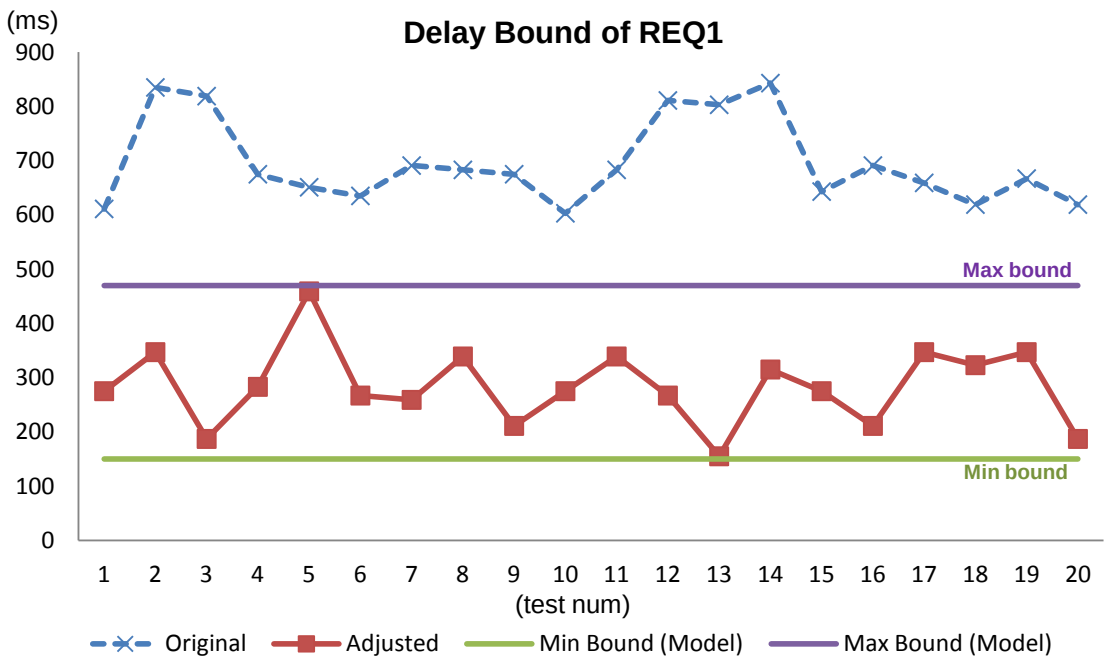


Figure 40: Validation Result for REQ1

conform to the delay-bound inclusion constraint.

Figure 40, 41, 42 show the validation result of REQ1, REQ2, REQ3, respectively. The purple line and and green line represent $f_{M_s}^{min}$ and $f_{M_s}^{max}$, respectively. Hence, the implementation delay (f_{IMP}) is expected to stay within these min/max bounds. The blue line represents the actual delay of the implemented system that executes the code generated from the system model (M_s) without any delay adjustment; on the other hand, the red line represents the actual delay of the implemented system that executes the code generated from the software model (M_c) that has been constructed according to the proposed approach.

According to our testing results, all measured implementation delays of the system running the code generated from M_c are within the delay-bound allowed by M_s for the three timing constraints (*i.e.,* the delay-bound inclusion constraint holds under this testing set). On the other hand, most measured implementation delays of the system running the code generated from M_s are out of the interval allowed by M_s for all three timing constraints (*i.e.,* the delay-

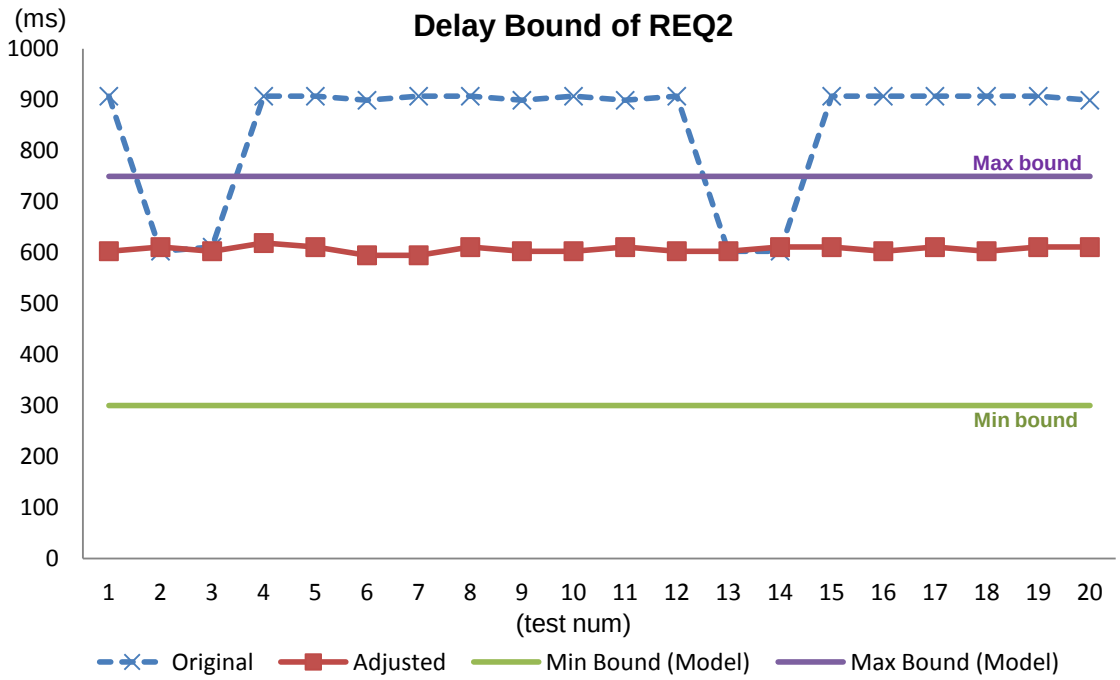


Figure 41: Validation Result for REQ2

bound inclusion constraint does not hold under this testing set). This result shows that a platform-independent model (M_s) is not an appropriate model to generate the code for the timing constraint conformance; instead, a software model (M_c) has to be used to generate the code toward the timing constraint conformance.

7.7. Summary of the Platform-Specific Code Generation

This chapter introduced a way to compensate the platform-processing delay by adjusting the timing parameters of the platform-independent model during code generation. In particular, we introduced a notion of the delay-bound inclusion constraint that is well matched to formally describe the timing constraints among occurrences of input and output. In order to meet the delay-bound inclusion constraint in the implementation level, we formalized the integer linear programming (ILP) problem in order to identify how much model delay should be adjusted in a way that it can be used to generate the code. This formalization enables the timing parameters to be automatically adjusted using the ILP solver by compensating

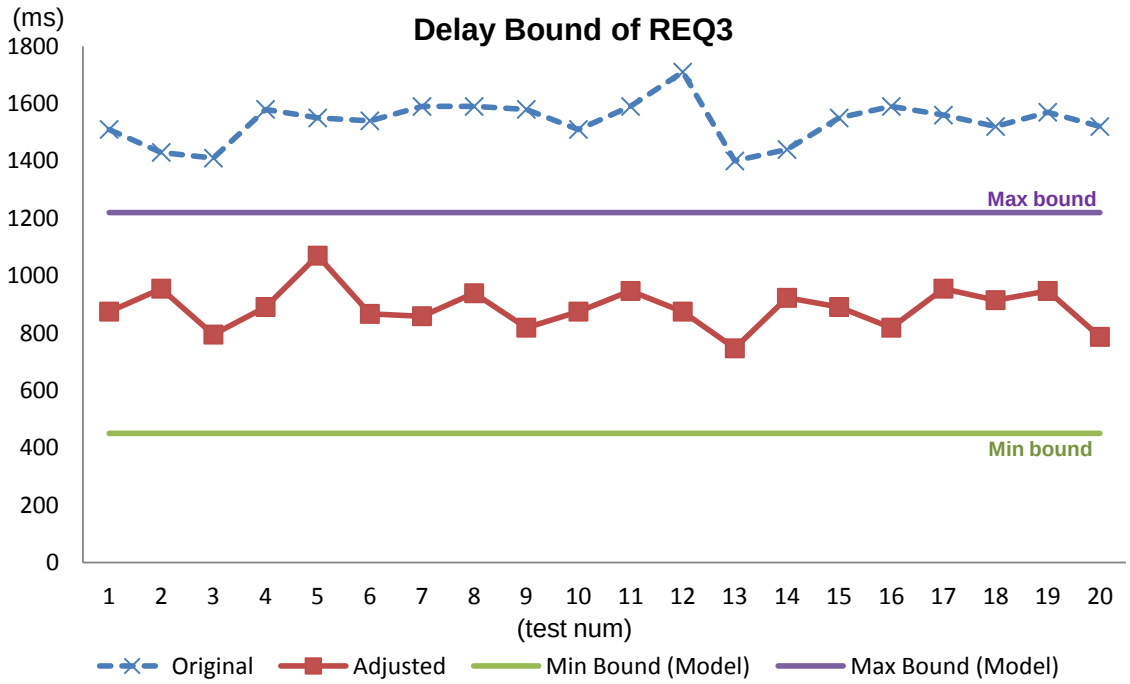


Figure 42: Validation Result for REQ3

the platform-processing delay. In addition, if a platform does not have sufficient processing power to run code generated from the platform-independent model, the solver gives output that says that there is no feasible adjustment strategy for the given platform. On the other hand, as long as the solver finds any valid timing parameter adjustment, then the solution guarantees the code to use the maximum computation time on a given platform while preserving the delay-bound inclusion constraint.

One benefit of this approach is that, by adjusting only the timing parameters of a platform-independent model, we can still use the capability of existing code generation tools that can automatically produce source code based on the model structure. In addition, if the solver finds a valid timing parameter assignment to obtain such a code, no additional verification or testing process is required to check how much timing deviation exists in the implemented system with respect to the platform-independent model due to the platform-processing delay; because, the delay has been already compensated while generating the code in a way that the implemented system conforms to the timing constraints.

CHAPTER 8 : Conclusions

Summary: This dissertation introduced how timing aspects of safety-critical systems can be modeled, verified, and implemented by demonstrating it through the PCA pump case study. By separating concerns of the platform-independent and the platform-dependent timing aspects, timed behavior of a system can be modeled, verified and implemented independently of platform-specific timing aspects. Such a development framework enables software not only to be reused across a range of platforms, but also to be developed in the absence of the platform-specific timing information. In addition, such decoupled development processes facilitate a system to be modeled through different levels of abstraction using modeling languages that can appropriately express each abstraction layer. In our work, we used state-transition formalism (*e.g.,* UPPAAL) to abstract the input/output timed behavior of the platform-independent aspect, while the architectural modeling (*e.g.,* AADL) is used to abstract the platform-dependent aspects, such as the thread components and interactions with sensors and actuators.

Besides, since software is developed independently of a particular platform, software may not be integrated with any platform in a way that the final implementation preserves timing constraints that have been verified in the platform-independent model. This is because how the timing overhead, originating from a chosen platform, impacts the timed behavior of the platform-independent model is not guaranteed. Therefore, our integration stage provides two different approaches to check whether such integration can be performed by conforming the timing constraints. The first approach is to systematically extend the platform-independent model into the platform-specific model that explicitly models the timing overhead of a platform. Hence, the platform-specific model has a similar timed behavior with that of the final implementation, so one can formally verify it to check whether the timing constraints are satisfied at the implementation level. The second approach is to optimize the platform-independent code by adjusting the timing parameters of the platform-independent model in a way that the platform-specific timing overhead can be appropriately compen-

sated to hold the timing constraints in the final implementation. Hence, as long as the code is integrated with a platform that is used to adjust the timing parameters, the implementation guarantees the timing constraints that has been verified in the platform-independent model.

Perspective: This dissertation gave special focus on the safety assurance associated with the timing aspects. Apart from the timing aspects, we believe that it is also worthwhile to discuss the advantages of the model-based development in building safety critical systems from a more general perspective.

The complexity of safety critical systems is growing fast. Hence, it becomes more challenging to reason about system-wide safety properties as a result of various internal interactions occurring across many different system layers that affect those properties in a complicated pattern. Unless complexity is handled in a way that a system can be developed in a traceable way, it is hard to expect the final implementation to meet high safety standards.

We believe that the model-based development - one example of such an application was demonstrated in this dissertation - is promising in a sense that a system can be formally modeled so that rigorous safety analysis is performed in the early development stage; consequently, implementations are systematically constructed from the verified model. The resulting implementation has a higher safety assurance in comparison to those developed in an adhoc fashion.

Applying this technique throughout the complete development cycle of building complex systems is challenging. One reason is that many modeling languages have different semantics, expressiveness and verification capability that may fit better than others to abstract certain aspects of systems. The associated modeling, verification and code generation techniques need to be used in concert to reason about the safety of the whole systems in order to gain the benefit from the model-based development to the fullest extent. We believe that more research has to be conducted especially regarding such interoperability for the

seamless connection among different modeling languages.

Development cost is an important issue from the industrial perspective. The development cost includes not only the money spent for the system development, but also the time duration until the system is released to the market and the reviewing efforts by the government authorities to approve the systems to be sold in the market. Even though we have not quantified such a cost in this PCA pump case study, we believe that the overall development cost can be reduced or at least worth spending on improving system safety in comparison to other development processes (*i.e.,* non model-based development process) for the following reasons:

Development Cost: Modeling and verification will add more costs to the early development stage since one needs an additional step of modeling systems to be developed. However, those additional modeling costs can be well compensated by the rest of the development: automatic code generation will reduce the implementation cost in comparison to the manual coding; automatic test case generation will reduce the cost in comparison to the manual test case generation from informal requirements and specifications.

Maintenance Cost: The model-based development can adapt to users' requirements that may change over time in a cost-effective way. Accommodating a minor requirement change can be costly in the safety-critical domain since one needs to argue how the change affects the safety claims that have been established in the existing implementation. In particular, if a system is complex, it is non-trivial how such minor changes will impact the whole system behavior without any means to trace them from the specification level down to the implementation level. In the model-based development, by modifying existing models, one can trace how such changes impact the system behavior through the model verification process. In addition, those modified aspects of the models can be systematically reflected to the implementation and testing process which will reduce the maintenance cost.

Reviewing cost: Safety-critical systems are typically reviewed by authorities, such as govern-

ment agencies, before they are released to the market. For example, infusion pumps require 510 (k) approval from the FDA, by demonstrating it is at least as safe and effective as a legally marketed device before they are sold. Such a reviewing process is challenging from both the reviewer and device manufacturer side. Device manufacturers need to convince the authorities by demonstrating the relationship of developed artifacts, such as design specification, source code, and testing results, in order to make the higher level safety claim of the device to be marketed. Reviewers need to understand and investigate the linkage of the artifacts to identify if there exists some holes between the safety claim and those artifacts. Without traceable artifacts, the review process can be challenging for both sides. In this sense, the model-based development can benefit both the device manufacturer and the reviewing authorities by enabling those artifacts to be produced in a traceable way so that it facilitates the reviewing process.

To conclude this dissertation, we hope that our case study of applying the model-based development to PCA pumps become a good exemplar so that it can promote the use of the model-based development in many other safety critical domains.

APPENDIX

A.1. The GPCA UPPAAL model

The four UPPAAL automata are translated from POST, Check Drug Routine, Infusion Configuration Routine, Infusion Session Submachine of State Controller of the GPCA Stateflow model that is explained in Section 4.3. We follow the naming convention of the GPCA Stateflow model to define locations and input/output synchronizations in the UPPAAL model.

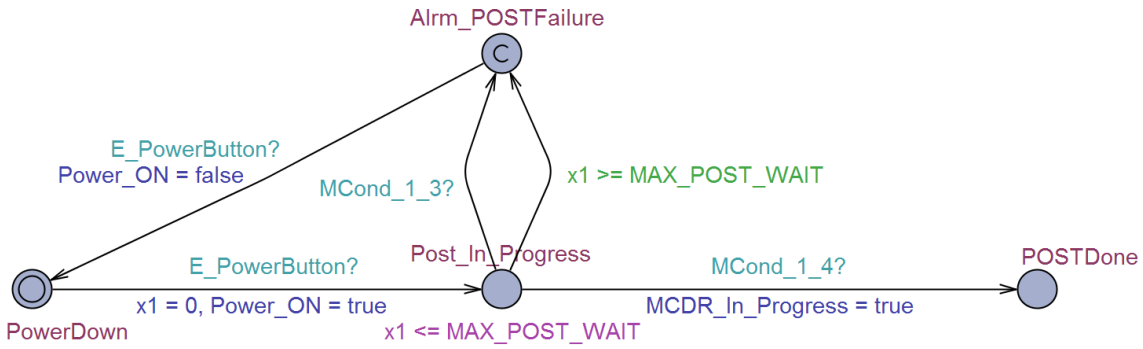


Figure 43: The POST Session

A.2. Appendix: Code Generation with Timing

A.2.1. Notations used in Chapter 7

Table 14 summarizes the notations used in Chapter 7.

A.2.2. ILP Formalization of Model 4

Model 4 has total 42 linear constraints as follows:

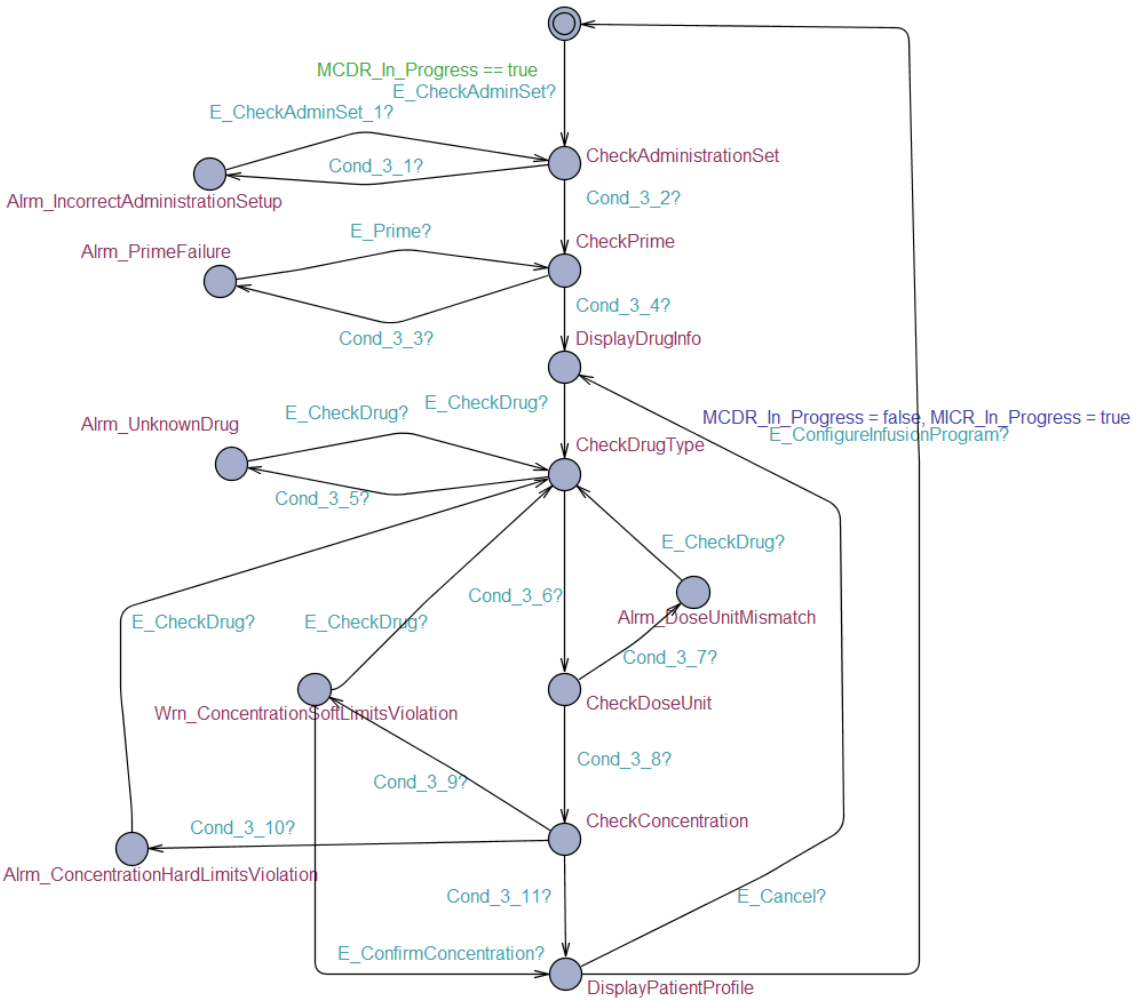


Figure 44: The Check Drug Routine

A.3. The Experimental Platforms for the GPCA Reference Implementations

A.3.1. Introduction to the GPCA experimental platforms

We introduce the experimental hardware platforms used for the GPCA infusion pump reference implementations. The model-based implementation methodology introduced in the previous chapters is validated by executing the developed source codes on these real PCA infusion pump hardware platforms. The intention of the case study using real PCA pumps is to provide a better understanding about how the model-based implementation

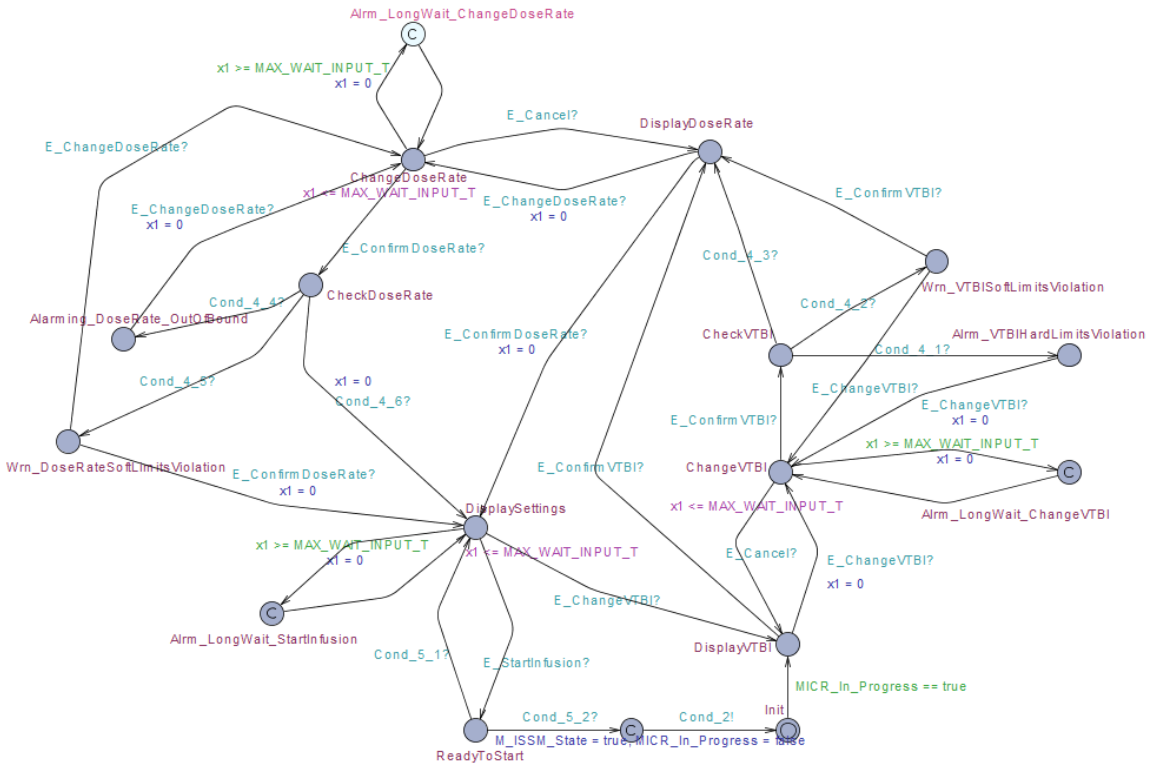


Figure 45: The Infusion Configuration Routine

can be applied in practice not only to the academic field, but also to the industrial field. We aim at demonstrating the proposed idea on a range of platforms that have different hardware structures. In order to meet this aim, the following criteria were considered in selecting PCA infusion pump hardware platforms:

- (**Criteria 1**) Each PCA pump hardware platform should be produced by different device manufacturers,

- (**Criteria 2**) In case PCA pumps are produced by the same device manufacturer, those should belong to different model categories.

We believe that those criteria will guide us to choose good candidate platforms whose internal hardware structures are different from each other enough for demonstrating our model-based implementation methodology. Figure 47 shows the two PCA infusion pump

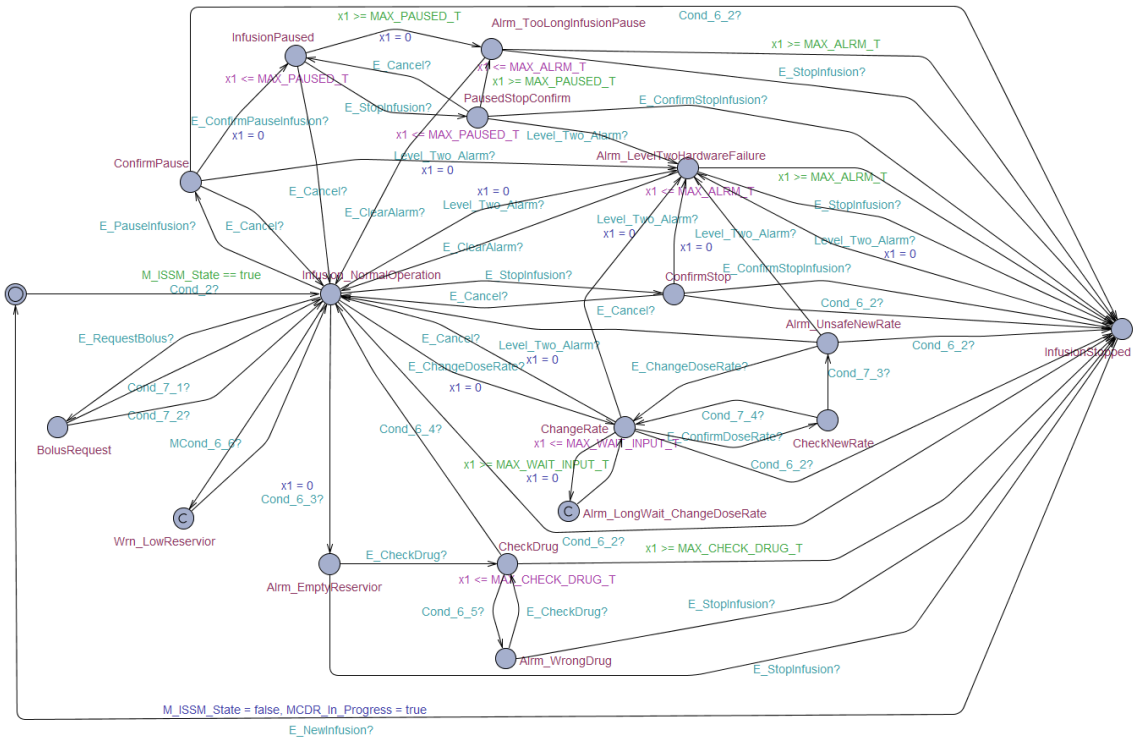


Figure 46: The Infusion Session Submachine

hardware platforms that have been obtained based on the criteria; their internal hardware structures are compared in Figure 4 in Chapter 3. We dismantled these hardware platforms, and implemented device drivers for sensors and actuators; such device drivers are considered as platform-dependent code that should be integrated with the platform-independent code. We note that since device manufacturers do not reveal information about how to interface their sensors and actuators publicly, we identified functionality of hardware pins connected to each sensor and actuator in a brute-force way (*i.e.,* examining all possible combinations of the hardware pins). All case studies and experiments were conducted on both or either hardware platforms depending on the research context introduced in the previous chapters.

A.3.2. Comparison of commercial PCA pump hardware platforms

In order to develop an embedded system - like infusion pumps - that meets safety requirements, understanding its hardware platform (*e.g.,* sensors and actuators) is as important as

(a) The evaluation hardware platforms (Baxter and Lifecare PCA pump)

(b) The test setup for the flow-rate accuracy testing

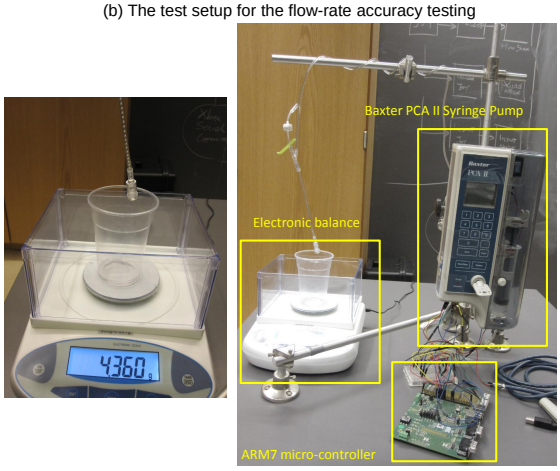


Figure 47: (a) The evaluation hardware platforms for the GPCA reference implementation (Baxter PCA Syringe Infusion Pump and Lifecare 4100 PCA Infusion Pump) (b) The test setup for the flow-rate accuracy testing based on the IEC 60601-2-24 standard [9]

understanding the software which will operate on the hardware platform. This is because the operation of such software is closely tied to the way the hardware platform is structured to interact with its physical environment. Therefore, some safety requirements may not be implemented using only software unless its platform has the necessary hardware supports, or the software should be implemented differently depending on the way its hardware platform is configured. These considerations are necessary in the development of PCA infusion pump systems as well. To understand the hardware aspects of the PCA infusion pump system, we compared three PCA infusion pump hardware platforms. Table 16 summarizes the comparison of sensors and actuators equipped in these PCA infusion pumps, and we had the following observations:

(Observation 1) Each platform has different sets of sensors and actuators, which requires different roles of software in implementing safety requirements:

Consider the following safety requirement:

- *If the current value / calculated volume of the reservoir is less than x ml, and an infusion is in progress, a Low Reservoir alert shall be issued.*

In order to implement this safety requirement in the Baxter PCA II Syringe platform, the software can utilize the hardware input read from the reservoir low sensor; if the sensor reading is true (*i.e.,* the syringe is close to empty), then the software can produce an alarm output. However, in order to implement the same requirement in the LifeCare 4100 PCA pump or the Baxter AP II PCA pump that are not equipped with such a sensor, the software should detect the low reservoir condition differently (*e.g.,* using the software time-out mechanism). From this observation, we expect that the platform-independent code will be integrated with quite different platform-dependent codes on different hardware platforms in order to fulfill the safety requirements depending on the availability of hardware supports. Hence, the software development approach should take into account such a variation when exploiting separation of concerns between the platform-independent and dependent aspects.

(Observation 2) Some safety requirements mandate a platform to have a certain sensor realize the safe operation, but some platforms are not equipped with the sensor:

Consider the following safety requirements:

- *If the pump is equipped with a flow rate sensor and the flow rate exceeds the programmed rate setting by more than n% over a period of more than t minutes, or if the pump goes into free flow, the pump shall issue an alarm to indicate overinfusion of the patient*
- *If the pump is equipped with a flow rate sensor and the flow rate is less than n% of the programmed rate setting over a period of t minutes, the pump shall issue an alarm to indicate underinfusion of the patient.*

In order to implement these requirements, a hardware platform must be equipped with a flow rate sensor (or a drop sensor) in order to calculate the actual flow rate of drug passing through intravenous tubes; in other words, software cannot calculate the actual flow rate without such a sensor. However, we observed that none of the three PCA infusion pumps in

Table 16 is equipped with a drop sensor. Therefore, a PCA pump system cannot satisfy these safety requirements no matter what software is implemented on these hardware platforms. This implies that formal verification and testing techniques introduced in this thesis can be applied to assure safety requirement conformance, but they may not be sufficient to assure some other safety requirements due to practical limitations of the system that we want to develop. In this case, we believe that the conformance of the safety requirements should be considered from the environmental context where the developed system will operate. For example, caregivers (*i.e.,* environment) should monitor the actual flow rate often enough so that such over-infusion or under-infusion situations can be reasonably detected, which could not be detected by the system itself.

Notations	Definitions
$\mathrm{M_s}$	The system model
$\mathrm{M_c}$	The software model
T_s, T_c	The timing parameter assignments to the system model and the software model
t_k^l, t_k^u	The lower/upper bound of clock values associated with a transition *k* of the software model
v_k^l, v_k^u	The lower/upper bound of clock values associated with a transition *k* in the system model
$\mathrm{P}(a_k)$ $[\mathrm{P}^{\min}(a_k), \mathrm{P}^{\max}(a_k)]$	The min/max platform processing delay of the event *ak*; By adding min and max superscripts, each indicates minimum delay bound and maximum delay bound separately.
$f_{\mathrm{M_s}}(i, j)$ $[f_{\mathrm{M_s}}^{\min}(i, j), f_{\mathrm{M_s}}^{\max}(i, j)]$	The min/max delay bound of a transition *j* succeeding a transition *i* in the system model; By adding min and max superscripts, each indicates minimum delay bound and maximum delay bound separately
$f_{\mathrm{M_s}}(p)$ $[f_{\mathrm{M_s},}^{\min}(p), f_{\mathrm{M_s}}^{\max}(p)]$	The min/max delay bound of a path *p* in the system model; By adding min and max superscripts, each indicates minimum delay bound and maximum delay bound separately
$f_{\mathrm{IMP}}(i, j)$ $[f_{\mathrm{IMP}}^{\min}(i, j), f_{\mathrm{IMP}}^{\max}(i, j)]$	The min/max delay bound of an event *j* (corresponding to transition *j* in the system model) succeeding an event *i* (corresponding to transition *i* in the system model) at the *mc*-boundary; By adding min and max superscripts, each indicates minimum delay bound and maximum delay bound separately
$f_{\mathrm{IMP}}(p)$ $[f_{\mathrm{IMP}}^{\min}(p), f_{\mathrm{IMP}}^{\max}(p)]$	The min/max delay bound of a final event (corresponding to the final transition of *p* in the system model) succeeding a start event (corresponding to the start transition of *p* in the system model) at the *mc*-boundary; By adding min and max superscripts, each indicates minimum delay bound and maximum delay bound separately
$f_{\mathrm{SOF}}(i, j)$ $[f_{\mathrm{SOF}}^{\min}(i, j), f_{\mathrm{SOF}}^{\max}(i, j)]$	The min/max delay bound of an event *j* (corresponding to transition *j* in the system model) succeeding an event *i* (corresponding to transition *i* in the system model) at the *io*-boundary; By adding min and max superscripts, each indicates minimum delay bound and maximum delay bound separately
$a_k?, a_k!$	The system model-level input/output event (identifier: *k*)
a_k^m, a_k^i	The implementation-level input event: the input (*m*) generated from the environment at the *mc*-boundary; the input (*i*) processed by the platform at the *io*-boundary (identifier: *k*)
a_k^o, a_k^c	The implementation-level output event: the output (*o*) generated by the code at the *io*-boundary; the output (*c*) generated by the platform at the *mc*-boundary (identifier: *k*)
x_{a_k}	The event clock associated with the event *ak*
$\Delta_{\mathrm{M_s}}(i, j)$	The uncertainty range of the event *j* occurrence following the event *i* occurrence in the system model
$\Delta_{\mathrm{IMP}}(i, j)$	The uncertainty range of the event *j* occurrence following the event *i* occurrence in the implemented system

Table 14: Notations used in Chapter 7

Table 15: The Linear Constraints for Model 4 (Figure 39 in Chapter 7)

ID	Constraint	ID	Constraint
C1a	$t_2^l+P_{min}^2+P_{min}^1 \geq 150$	C1b	$t_2^u+P_{max}^2+P_{max}^1 \leq 600$
C2a	$t_2^l+t_3^l+P_{min}^3+P_{min}^1 \geq 900$	C2b	$t_2^u+t_3^u+P_{max}^3+P_{max}^1 \leq 2100$
C3a	$t_2^l+t_4^l-(P_{max}^4-P_{min}^1) \geq 150$	C3b	$t_2^u+t_4^u-(P_{min}^4-P_{max}^1) \leq 2100$
C3a	$t_2^l+t_4^l-(P_{max}^4-P_{min}^1) \geq 150$	C3b	$t_2^u+t_4^u-(P_{min}^4-P_{max}^1) \leq 2100$
C4a	$t_2^l+t_4^l+t_5^l+P_{min}^5+P_{min}^1 \geq 350$	C4b	$t_2^u+t_4^u+t_5^u+P_{max}^5+P_{max}^1 \leq 2600$
C5a	$\min[t_3^l+t_1^l-(P_{max}^1+P_{max}^2), t_4^l+t_5^l+t_1^l-(P_{max}^1+P_{max}^2)] \geq 5200$	C5b	$\max[t_3^u+t_1^u-(P_{min}^1+P_{min}^2), t_4^u+t_5^u+t_1^u-(P_{min}^1+P_{min}^2)] \leq$ INF
C6a	$t_3^l+P_{min}^3-P_{max}^2 \geq 750$	C6b	$t_3^u+P_{max}^3-P_{min}^2 \leq 1500$
C7a	$t_4^l-(P_{max}^4+P_{max}^2) \geq 0$	C7b	$t_4^u-(P_{min}^4+P_{min}^2) \leq 1500$
C8a	$t_4^l+t_5^l+(P_{min}^5-P_{max}^2) \geq 200$	C8b	$t_4^u+t_5^u+(P_{max}^5-P_{min}^2) \leq 2000$
C9a	$t_1^l-(P_{max}^3+P_{max}^1) \geq 5000$	C9b	$t_1^u-(P_{min}^3+P_{min}^1) \leq$ INF
C10a	$t_1^l+t_2^l+(P_{min}^3-P_{max}^2) \geq 5150$	C10b	$t_1^u+t_2^u+(P_{max}^3-P_{min}^2) \leq$ INF
C11a	$t_1^l+t_2^l+t_4^l-(P_{max}^4+P_{max}^3) \geq 5150$	C11b	$t_1^u+t_2^u+t_4^u-(P_{min}^4+P_{min}^3) \leq$ INF
C12a	$t_1^l+t_2^l+t_4^l+t_5^l+(P_{min}^5-P_{max}^3) \geq 5350$	C12b	$t_1^u+t_2^u+t_4^u+t_5^u+(P_{max}^5-P_{min}^3) \leq$ INF
C13a	$t_5^l+t_1^l-(P_{max}^1-P_{min}^4) \geq 5200$	C13b	$t_5^u+t_1^u-(P_{min}^1-P_{max}^4) \leq$ INF
C14a	$t_5^l+t_1^l+t_2^l+P_{min}^2+P_{min}^4 \geq 5350$	C14b	$t_5^u+t_1^u+t_2^u+P_{max}^2+P_{max}^4 \leq$ INF
C15a	$t_5^l+t_1^l+t_2^l++t_3^l+P_{min}^3+P_{min}^4 \geq 6100$	C15b	$t_5^u+t_1^u+t_2^u++t_3^u+P_{max}^3+P_{max}^4 \leq$ INF
C16a	$t_5^l+P_{min}^5+P_{min}^4 \geq 200$	C16b	$t_5^u+P_{max}^5+P_{max}^4 \leq 500$
C17a	$t_1^l-(P_{max}^5+P_{max}^1) \geq 5000$	C17b	$t_1^u-(P_{min}^5+P_{min}^1) \leq$ INF
C18a	$t_1^l+t_2^l+(P_{min}^2-P_{max}^5) \geq 5150$	C18b	$t_1^u+t_2^u+(P_{max}^2-P_{min}^5) \leq$ INF
C19a	$t_1^l+t_2^l+t_3^l+(P_{min}^3-P_{max}^5) \geq 5900$	C19b	$t_1^u+t_2^u+t_3^u+(P_{max}^3-P_{min}^5) \leq$ INF
C20a	$t_1^l+t_2^l+t_4^l-(P_{max}^4-P_{max}^5) \geq 5150$	C20b	$t_1^u+t_2^u+t_4^u-(P_{min}^4+P_{min}^5) \leq$ INF
C21	$t_1^l \geq 0 \wedge t_1^u \geq 0 \wedge t_2^l \geq 0 \wedge t_2^u \geq 0 \wedge t_3^l \geq 0 \wedge t_3^u \geq 0 \wedge t_4^l \geq 0 \wedge t_4^u \geq 0 \wedge t_5^l \geq 0 \wedge t_5^u \geq 0$	C22	$t_1^l \leq t_1^u \wedge t_2^l \leq t_2^u \wedge t_3^l \leq t_3^u \wedge t_4^l \leq t_4^u \wedge t_5^l \leq t_5^u$

Table 16: The comparison of three commercial PCA infusion pump hardware platforms

Type	Hardware Part	LifeCare 4100 PCA pump	Baxter PCA II Syringe pump	Baxter AP II PCA pump
Sensor	Door sensor	O	O	O
	Reservoir Empty Sensor	O	O	X
	Reservoir Low Sensor	X	O	X
	Drop sensor	X	X	X
	Pressure sensor	O	O	O
	Air-in-line sensor	X	X	X
	Humidity sensor	X	X	X
	Temperature sensor	X	X	X
	Voltage sensor	O	O	O
	Patient-Pendant	O	O	O
	Drug-Loaded Sensor	O	X	X
	-	Shaft Sensor	X	X
Actuator	Pump Motor	O (Stepper Motor)	O (DC Motor)	O (DC motor)
	Buzzer	O	O	O

BIBLIOGRAPHY

[1] Architecture Analysis and Design Language. http://www.aadl.info.

[2] The generic infusion pump project. http://rtg.cis.upenn.edu/gip.php3.

[3] The generic patient controlled analgesia pump hazard analysis. http://rtg.cis.upenn.edu/gip.php3.

[4] Infusion pump software safety research at fda. http://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/ucm202511.htm.

[5] Medical device safety - medical device recalls. http://www.fda.gov/MedicalDevices/Safety/ListofRecalls/default.htm.

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

[7] Simulink Coder: Generate C and C++ code from simulink and stateflow models. http://www.mathworks.com/products/simulink-coder.

[8] Using the freertos real-time kernel. http://www.freertos.org.

[9] *International Standard-Medical electrical equipment Part 2-24: Particular requirements for the basic safety and essential performance of infusion pumps and controllers*. International Electrotechnical Commission, 2012.

[10] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Model-based implementation of real-time applications. In *EMSOFT*. ACM, 2010.

[11] K. Altisen and S. Tripakis. Implementation of timed automata : an issue of semantics or modeling. *FORMATS*, 2005.

[12] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[13] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata: a determinizable class of timed automata. *Theoretical Computer Science*, 1999.

[14] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *FORMATS*, 2003.

[15] Gerd Behrmann, Alexandre David, and KimG. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[16] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. *UPPAALâĂŤa tool suite for automatic verification of real-time systems*. Springer, 1996.

[17] Torsten Blochwitz, Martin Otter, Johan Åkesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, Hans Olsson, and Antoine Viel. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *Proceedings of the 9th International Modelica Conference*, pages 173–184. The Modelica Association, 2012.

[18] Borzoo Bonakdarpour, Marius Bozga, and Jean Quilbeuf. Model-based implementation of distributed systems with priorities. *Design Automation for Embedded Systems*, pages 1–26, 2012.

[19] Jon Friedman Brett Murphy, Amory Wakefield. Best practices for verification, validation, and test in model-based design. 2008.

[20] David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Determinate composition of fmus for cosimulation. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT '13, pages 2:1–2:12, Piscataway, NJ, USA, 2013. IEEE Press.

[21] David Broman, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Requirements for hybrid cosimulation standards. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, HSCC '15, pages 179–188, New York, NY, USA, 2015. ACM.

[22] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics*. 2001.

[23] Gregory Gay, Sanjai Rayadurgam, and Mats P. E. Heimdahl. Steering model-based oracles to admit real program behaviors. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 428–431, New York, NY, USA, 2014. ACM.

[24] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[25] Sanjai Rayadurgam Gregory Gay and Mats P.E. Heimdahl. Improving the accuracy of oracle verdicts through automated model steering. In *The Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014)*, 2014.

[26] Grégoire Hamon. A denotational semantics for stateflow. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 164–172, New York, NY, USA, 2005. ACM.

[27] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[28] Thomas A. Henzinger and Christoph M. Kirsch. The embedded machine: Predictable, portable real-time code. *ACM Trans. Program. Lang. Syst.*, 29(6), October 2007.

[29] A. Hessel and P. Pettersson. A test case generation algorithm for real-time systems. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 268–273, 2004.

[30] Anders Hessel, KimG. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal real-time test case generation using uppaal. In *Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes in Computer Science*, pages 114–130. Springer Berlin Heidelberg, 2004.

[31] Inhye Kang and Insup Lee. An efficient state space generation for analysis of real-time systems. *SIGSOFT Softw. Eng. Notes*, 21(3):4–13, May 1996.

[32] Baek Gyu Kim, L.T.X. Phan, Insup Lee, and O. Sokolsky. A model-based i/o interface synthesis framework for the cross-platform software modeling. In *Rapid System Prototyping (RSP), 2012 23rd IEEE International Symposium on*, pages 16–22, Oct 2012.

[33] BaekGyu Kim, Anaheed Ayoub, Oleg Sokolsky, Insup Lee, Paul Jones, Yi Zhang, and Raoul Jetley. Safety-assured development of the gpca infusion pump software. In *EMSOFT*, 2011.

[34] BaekGyu Kim, Hyeon I Hwang, Taejoon Park, Sang H. Son, and Insup Lee. A layered approach for testing timing in the model-based implementation. In *DATE '14*, pages 1–4, March 2014.

[35] William K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall, 2005.

[36] KimG. Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In *Formal Approaches to Software Testing*, pages 79–94. 2005.

[37] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and JÃľrome Hugues. Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. In *Reliable Software Technologiesï£¡ Ada-Europe 2009*, volume 5570 of *Lecture Notes in Computer Science*, pages 237–250. Springer Berlin / Heidelberg, 2009.

[38] R. Lublinerman, C. Szegedy, and S. Tripakis. Modular code generation from synchronous block diagrams: Modularity vs. code size. In *ACM Symposium on Principles of Programming Languages (POPL '09)*, 2009.

[39] R. Lublinerman and S. Tripakis. Modularity vs. reusability: Code generation from synchronous block diagrams. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 1504–1509, 2008.

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

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

[42] Patrick Oladimeji, Harold Thimbleby, and Anna Cox. Number entry interfaces and their effects on error detection. In *Proceedings of the 13th IFIP TC 13 international conference on Human-computer interaction - Volume Part IV*, INTERACT'11, pages 178–185, Berlin, Heidelberg, 2011. Springer-Verlag.

[43] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25:41–61, 1995.

[44] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. Taming dr. frankenstein: Contract-based design for cyber-physical systems*. *European Journal of Control*, 18(3):217 – 238, 2012.

[45] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of simulink/stateflow into lustre. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 259–268, New York, NY, USA, 2004. ACM.

[46] G. Schirner, A. Gerstlauer, and R. Domer. Automatic generation of hardware dependent software for mpsocs from abstract system specifications. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pages 271–276, March 2008.

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

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

[49] Jeffrey M. Thompson, Mats P.E. Heimdahl, and Debra M. Erickson. Structuring formal control systems specifications for reuse: Surviving hardware changes. Technical Report TR 00-004, University of Minnesota, January 2000. Submitted to the *Fifth NASA Langley Formal Methods Workshop*.

[50] Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.

[51] Ahlem Triki, Jacques Combaz, Saddek Bensalem, and Joseph Sifakis. Model-based implementation of parallel real-time systems. In *FASE*, Lecture Notes in Computer Science. 2013.

[52] U.S. Food and Drug Administration, Center for Devices and Radiological Health. *Guidance for Industry, FDA Reviewers and Compliance on Off-The-Shelf Software Use in Medical Devices*, September 1999.

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

[54] Martin Wulf, Laurent Doyen, and Jean-Francois Raskin. Almost asap semantics: From timed models to timed implementations. In *HSCC*. 2004.