Safety-Assured Development of the GPCA Infusion Pump Software *

BaekGyu Kim Anaheed Ayoub
Oleg Sokolsky Insup Lee
Computer and Information Science Dept.,
University of Pennsylvania
Philadelphia
Pennsylvania
USA
{baekgyu,anaheed,
sokolsky,lee}@cis.upenn.edu

Paul Jones Yi Zhang Raoul Jetley
OSEL, Center for Devices and Radiological Health
U.S. Food and Drug Administration
Silver Spring
Maryland
USA
{PaulL.Jones, Yi.Zhang2,
Raoul.Jetley}@fda.hhs.gov

ABSTRACT

This paper presents our effort of using model-driven engineering to establish a safety-assured implementation of Patient-Controlled Analgesic (PCA) infusion pump software based on the generic PCA reference model provided by the U.S. Food and Drug Administration (FDA). The reference model was first translated into a network of timed automata using the UPPAAL tool. Its safety properties were then assured according to the set of generic safety requirements also provided by the FDA. Once the safety of the reference model was established, we applied the TIMES tool to automatically generate platform-independent code as its preliminary implementation. The code was then equipped with auxiliary facilities to interface with pump hardware and deployed onto a real PCA pump. Experiments show that the code worked correctly and effectively with the real pump. To assure that the code does not introduce any violation of the safety requirements, we also developed a testbed to check the consistency between the reference model and the code through conformance testing. Challenges encountered and lessons learned during our work are also discussed in this paper.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: [Formal methods, Model checking, Validation]

General Terms

Design and Verification

Keywords

PCA infusion pump, model-based engineering, formalization, verification, code synthesis, timed automata

1. INTRODUCTION

Infusion pumps are medical devices that deliver medicine such as antibiotics, chemotherapy drugs, and pain relievers, into a patient's body in prescribed amounts for therapeutic purposes. According to the Infusion Pump Improvement Initiative of U.S. Food and Drug Administration (FDA) [18], numerous adverse events have been reported that are associated with the use of infusion pumps; some of the cases resulted in serious injuries and deaths. Software defects are implicated in many of these adverse events.

Researchers at the FDA have launched a Generic Infusion Pump (GIP) project, to help address safety problems associated with infusion pump software. The goal of the project is to develop a set of generic safety reference models that can be used as reference standards and test harnesses to verify the safety of infusion pump software [5, 11]. As an extension to the GIP project, we concentrated on safety issues related to Patient Controlled Analgesic (PCA) infusion pumps [4]. A PCA infusion pump allows patients to request additional doses (called boluses) of pain-relief medication - beyond a preset base dose (rate), by pressing a "request" button attached to the pump. A preliminary hazard analysis, safety requirements, and reference model, called the Generic PCA (GPCA) model, are already released to the public [1, 2].

Model-driven development (MDD) is a software development approach in which abstract models of software systems are created, analyzed for correctness, and systematically transformed to a concrete implementation [9]. In this paper, we use MDD to establish an implementation of a PCA infusion pump prototype based on the GPCA model and safety requirements [1, 2]. Our approach is to: 1) formalize the model and the requirements using the UPPAAL tool [7], 2) formally verify that the model satisfies the safety requirements, and 3) use the TIMES tool [3] to generate code out of the verified model. The platform-independent C code produced by the TIMES tool is then extended with glue code and installed on a real PCA pump hardware. The correctness of the extended code is verified by conformance testing on a testbed, which monitors the execution of the code and compares it with the corresponding model execution. The overall scheme of our approach is shown in Figure 2 and further discussed in Section 3.

The main contributions of this paper are as follows:

- We present a case study of immediate practical importance. The FDA Infusion Pump Improvement Initiative provides specific guidelines to infusion pump manufacturers. We believe that our methodology may provide an exemplar for the medical device industry towards satisfying these guidelines.

*This research is supported in part by NSF CNS-0834524, NSF CNS-0930647, NSF CNS-1035715, and NSF CNS-1042829 (FDA SIR).

- We discuss the challenges encountered applying a MDD approach to the case study. In particular, the TIMES tool generates code for closed systems which raises a practical issue of interfacing an open system with its environment at the deployment stage. We introduce our approach to overcoming this issue by adding glue code in order to keep the semantics of environmental channels unchanged in the implementation.

- We categorize the informally stated GPCA safety requirements into classes that require different handling in the development process. Some requirements can be formalized and verified at the model level. Verification of some other requirements, on the other hand, cannot be performed given the level of model detail. The rest of the requirements are not formalizable, but can be validated at the implementation level. We discuss handling these different categories of requirements in our development methodology.

This paper is organized as follows. First, the GPCA safety requirements and GPCA model are introduced in Section 2. Our approach is given in Section 3. Formal verification and automated implementation are detailed in Sections 4 and 5, respectively. Our testbed of the model-driven implementation and the testing results are shown in Section 6. Related work is introduced in Section 7, followed by discussion and future work in Section 8.

2. THE GPCA PROJECT

This section describes the GPCA Safety Requirements and the GPCA model that our work relies on.

2.1 The GPCA Safety Requirements

The GPCA safety requirements [2] were derived from an analysis of hazards encountered in the use of PCA infusion pumps on the market. They serve to establish a minimum degree of safety for these devices. The hazards, and hence safety requirements, were developed primarily from an abstracted software system perspective, permitting total freedom in actual device implementation.

Two examples of the GPCA safety requirements are defined as follows:

- No normal bolus doses should be administered when the pump is alarming.

- If the calculated volume of the reservoir is y *ml*, and an infusion is in progress, an Empty Reservoir alarm shall be issued.

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

Our development process needs to ensure that the GPCA implementation satisfies the safety requirements. The first step is to formalize the requirements as temporal formulae in the syntax of the UPPAAL model checker. Formalized safety requirements are used not only to verify the GPCA model but also to build test cases for the GPCA implementation in the validation phase.

Unfortunately, not all safety requirements can be formalized as temporal logic formulae or directly verified against the GPCA model. We discuss this in later sections.

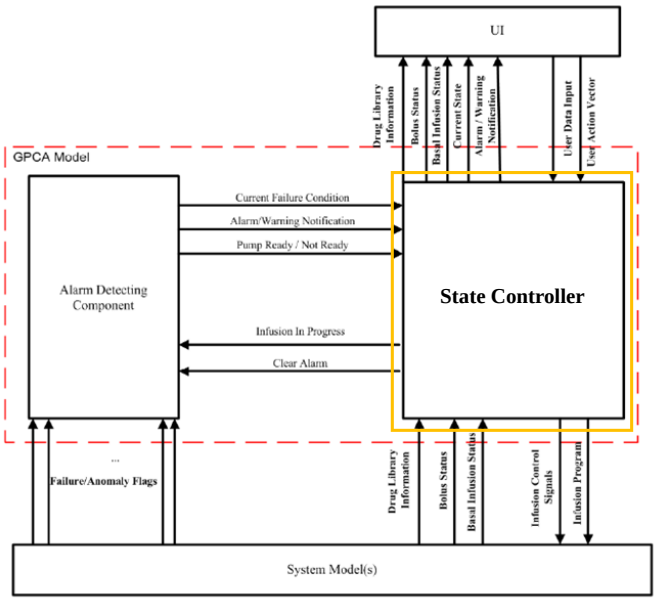

Figure 1: System Architecture of the GPCA Model

2.2 The GPCA Model

The GPCA model is an abstract representation of common behaviors shared by typical PCA pump software. The model is built using Mathworks Simulink [16] and Stateflow [17]. As shown in Figure 1, the GPCA model consists of two state machines - the *State Controller* and the *Alarm Detecting Component*. The primary purpose of the *State Controller* is to regulate the rest of the pump to fulfill its expected functionality, i.e., administering the right drug to the right patient at a right rate and dosage. On the other hand, the *Alarm Detecting Component* serves as an interface to receive surveillance signals, e.g., ambient temperature, from hardware sensors. It should also notify the *State Controller* of any anomaly in the signals received, so that the *State Controller* can react to the anomaly promptly.

In this paper, we concentrate on the *State Controller* because it embodies the core functions of PCA pump software. 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 smooth and correct operation of the pump, including checking patient information, checking the correctness of infusion programs, guiding the user on how to use the device, notifying the user of unsafe conditions via alarms, and so on. A detailed description of the model is given in [1].

The GPCA *State Controller* consists of four parts - *Power-On-Self-Test* (POST), *Check Drug Routine*, *Infusion Configuration Routine* and *Infusion Session Submachine* - corresponding to four typical steps in infusion processes.

- The *POST*, triggered by turning the power on, includes self-tests of processors and memory, critical circuitry, indicators, displays, and alarms to ensure that the device is ready for use.

- The *Check Drug Routine* checks drug type and concentration to make sure that the right drug is loaded.

- The *Infusion Configuration Routine* enables the user to input and adjust infusion parameters. A typical infusion programming instance requires the user to define the dose rate and dosage, i.e., volume to be infused (VTBI). To reduce potential dose errors, this submachine also checks the input infusion parameters against a pre-loaded drug library. If these parameters are deemed unsafe based on the drug library, the submachine would prompt the user to either reconfigure them or abort the infusion.

- The *Infusion Session Submachine* abstracts how software 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, resetting or disabling alarms, etc. The *Infusion Session Submachine* has to correctly interpret user inputs and adjust pump operation accordingly.

The behavior of these subsystems within the GPCA *State Controller* is expressed in Stateflow state-transition charts, which in total consist of more than 50 states and 100 transitions. The control flows of these charts depend on about 50 user events and hardware conditions.

3. OUR APPROACH

We pursued a model-driven approach in the development of the GPCA prototype, which relies on formal modeling and analysis tools. Through this approach, we expect to find any incompleteness in the GPCA model or any of its violations to the safety requirements, and further to automatically generate code from the verified model. In this section, we introduce how the GPCA safety requirements and the GPCA model are used in developing the GPCA prototype.

Figure 2 shows our approach; given the GPCA Simulink/Stateflow model, a UPPAAL model was constructed using a manual translation process. Along with functional and architecture requirements, safety requirements were also manually translated into temporal logic formulae using the UPPAAL query language. The UPPAAL model was then formally verified, by using the UPPAAL model checker, to assure that it satisfies all the formalized safety requirements.

Once the safety of the UPPAAL model was assured, we used the TIMES tool to synthesize it into C code. An advantage of using the TIMES tool is that it guarantees behavioral consistency between the synthesized code and the UPPAAL model. The TIMES tool generates either BrickOS platform code or platform independent code[1]. We chose to generate platform-independent code, and then customized it for our particular target platform. We introduced glue code that invokes platform-dependent system calls to interface with the platform-independent code on our target platform. In

[1]We note that the TIMES tool is for generating real-time task scheduling where timed automata are used for task arrival, execution time, deadline. However, we are only using the timed behavior of automata and synchronization. Our proposed methodology is equally applicable with other code generators/synthesizers from timed automata.

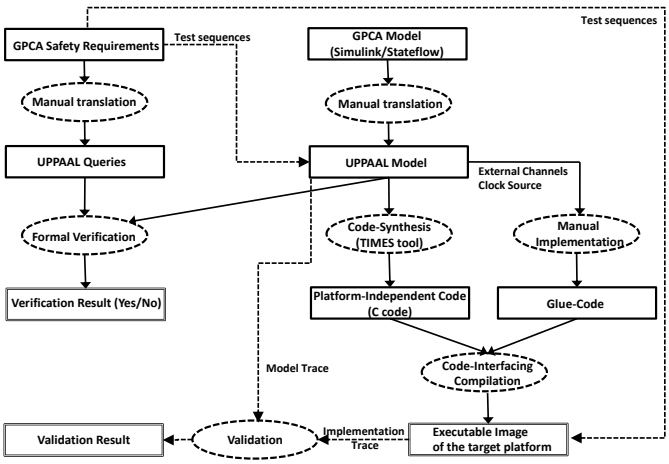

Figure 2: The Model-Driven Development for the GPCA prototype.

particular, the glue code provides a clock source implementation that provides the timing semantics of timed automata. Further, the glue code is used to implement communication channels between the GPCA code and its platform environment. The glue code is described in more detail in Section 5.

In order to validate the synthesized GPCA implementation, we developed a tester that consists of two primary parts: 1) an input generator that fed the implementation with environmental stimulus, such as user inputs or hardware conditions, and 2) a monitor that observed the runtime behavior of the implementation relative to the particular stimulus. The observed runtime behavior was then compared with the execution of the UPPAAL model using the same stimulus (conformance testing) illustrated as dotted lines in Figure 2. The safety requirements were also used to produce testing scenarios that were used as input to the tester.

The next three sections explain the steps of our model-driven approach: formal verification, automated implementation, and validation.

4. FORMAL MODELING AND VERIFICATION

Formalization of the GPCA model. We transformed the GPCA model expressed in Simulink and Stateflow into a network of UPPAAL automata through a manual process. To retain as much of the syntactic structure of the Stateflow model as possible, the transformation maintained 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 semantic differences between two models. Instead, we reconstructed the general functions of the Simulink/Stateflow model in the UPPAAL model, which was formally verifiable against the GPCA safety requirements.

The GPCA Stateflow model is organized hierarchically as four sequentially connected state machines. Each of these four state machines has a final state that sets a special condition variable when it is entered. This condition variable

triggers the execution of the model transition from this state machine to the next. 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. Condition variables are also used in the Stateflow model to represent inputs. These variables are set by the model environment and trigger transitions between states. Each of such variables was kept in the UPPAAL model in the format of communication channels that triggers the corresponding transitions in the model.

Most states and transitions in both the GPCA Stateflow model and UPPAAL automata have accompanied actions. For example, if one of the models is in the *Alarm-Empty-Reservoir* state, it is expected to launch an alarm to inform the empty-reservoir condition. Such accompanied actions also need to be implemented when code is generated from the models. For example, the action of raising an alarm when the model is in the *Alarm-Empty-Reservoir* state should be implemented as sending an electric signal to the pump's buzzer to make an appropriate alarming sound. Fortunately, since we forced the generated code to inherit the structure of the UPPAAL automata, it became easier to implement the accompanied actions.

During the transformation, we also had to introduce quantitative timing information into the UPPAAL model. The GPCA Stateflow 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.

We now describe the four UPPAAL automata that correspond to the four parts of the GPCA model. Due to space limitation, we only present the time automata for the *Infusion Configuration Routine* and the *Infusion Session Submachine*.

The POST Session. 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. An exception 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 noted 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. This automaton goes through a series of checks such as checking drug types. 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.

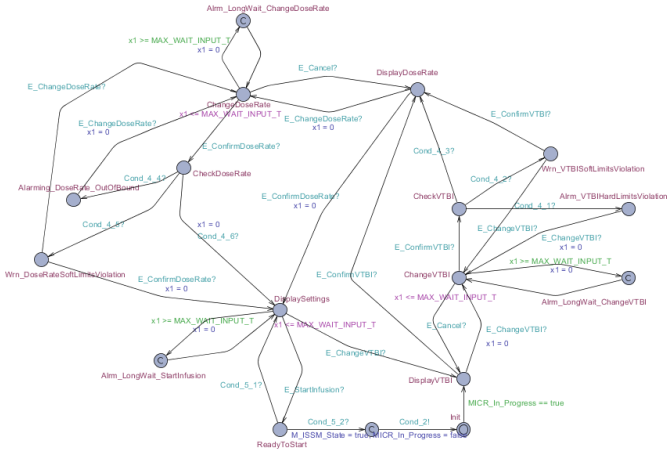

Figure 3: The Infusion Configuration Routine.

The Infusion Configuration Routine. Figure 3 shows the UPPAAL model of the *Infusion Configuration Routine*. This model describes a workflow that a caregiver goes through to setup an infusion administration program (i.e. prescription). Two infusion parameters are specified in the *Infusion Configuration Routine*: dose rate and VTBI, both of which must be set before the model enters into the *Infusion Session Submachine*. The routine raises alarms if the entered parameter values exceed the soft or hard limits specified in a drug library. 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.

The first requirement cannot be formalized, but can be validated by design review. The remaining two requirements, on the other hand, can be formalized and verified in the UPPAAL model. 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 safety requirements.

The Infusion Session Submachine. Figure 4 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. Code to control the pump 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 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 implement 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 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.

Formalization of safety requirements. The GPCA safety requirements are translated into temporal logic formulae expressed in the UPPAAL query language. For example, consider the safety requirement, *No bolus dose shall be possible during the POST*. This requirement can be captured by the following temporal logic formula:

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

where the word *No* is mapped to the logic operator !(not), *POST.POST-In-Progress* is a variable reflecting whether or not the model stays in the POST state, and the variable *ISSM.BolusRequest* is used to indicate if a bolus request is issued. Lastly, the temporal quantifier $A[\,]$ (invariantly) enforces the above formula to be satisfied by the model in all its executions.

The requirement, *The pump shall issue an alert if paused for more than t minutes*, is formalized as a *leads to* formula in UPPAAL. The *leads to* form can express a property like whenever a certain condition is satisfied, then eventually another condition will be satisfied. The following temporal logic formula captures this safety requirement.

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

where *MAX-PAUSED-T* is the instantiated time unit from t minutes in the safety requirement. Several more examples of requirement formalization are shown in Table 1. The current GPCA safety requirements contain 97 requirements. We translated 20 requirements into temporal logic formulae, and verified them in the UPPAAL model. All of these requirements are satisfied by the GPCA UPPAAL model. The remaining requirements either could not be expressed as temporal logic formulae or could not be verified on the GPCA model. In Section 8, we present our categorization of GPCA requirements and discuss the manipulation of different categories of these requirements.

Table 1: Mapping between Safety Requirements and Safety Properties

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).
SP	A[](! (ISSM.BolusRequest && CDR.Alrm-UnknownDrug))
SR 3.4.3	The POST shall take no longer than t seconds.
SR	(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.
SP	(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.
SP	(ICR.ChangeDoseRate && x1 > MAX-WAIT-INPUT-T) → (ICR.Alrm-LongWait-ChangeDoseRate)

5. CODE SYNTHESIS AND ADAPTATION

Applying the automatic code synthesis to the GPCA implementation is introduced in this section. The TIMES tool is used to generate source code from the formal model. The generated code uses glue code to interface with the target platform. The glue code for the environmental channel is explained.

5.1 Automated Implementation with TIMES

Manual implementation of embedded system software is error prone due to the large number of control states and variety of events that the code needs to react to. An automated implementation improves the quality of embedded software in that it reduces human errors while retaining the benefits of model verification. The current UPPAAL model has more than 50 states and 100 transitions, and reacts to over 50 conditions and user events. Even a moderately complex application such as this is difficult to implement manually without introducing significant errors.

TIMES is a tool suite for symbolic schedulability analysis and synthesis of executable code with predictable behavior for real-time systems [3] . We use its code-synthesis function to translate the behavior of the UPPAAL model into source code. The tool generates C code that is either platform independent or specific to brickOS operating system running on the LEGO Mindstorm platform. We adopted the platform-independent version and then instrumented it to run on our target platform. We briefly explain the code-synthesis scheme of the TIMES tool to help in understanding the glue code in the next subsection.

In the code-synthesis scheme of the TIMES tool, transitions in a timed-automata model are stored in an array of type *trans-t*. The data structure *trans-t* contains four fields to represent transitions: an active transition flag, a *source-location-id*, a *destination-location-id*, and a *synchronization-id*. The active transition flag is an indicator that the transition needs to be evaluated in the current state. For example, in the *InfusionPaused* state in Figure 4, two transitions are active: a transition to the *Infusion-NormalOperation* state

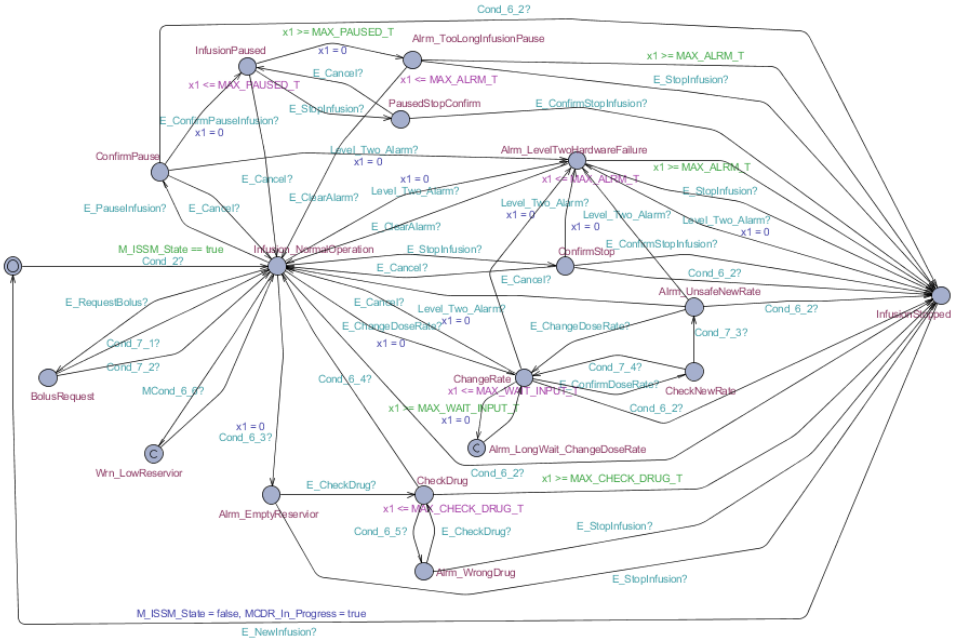

Figure 4: The Infusion Session Submachine.

and a transition to the *Alrm-TooLong-InfusionPause* state. The *eval-guard* function evaluates guards of active transitions. If guards for multiple transitions from a same state are satisfied, one of these transitions will be taken according to their order in the *trans-t* array: a transition with a lower index has a higher priority. The *source-location-id* and the *destination-location-id* are used to specify the origin and destination states of a single transition, respectively. Once a transition t is taken, transitions indexed by t's *source-location-id* are deactivated by setting their active fields to false. In contrast, transitions indexed by t's *destination-location-id* are activated so that the new active transitions can be processed in the next iteration of the evaluation of guards. The *synchronization-id* indicates that a transition contains a channel synchronization with another complement transition. The *check-trans* function shown in Listing 1, automatically generated by the TIMES tool, implements the behavioral flow of timed automata based on the *trans-t* data structure.

5.2 Communication with the environment

The platform-independent code generated by the TIMES tool needs to be ported to the target platform in a way that preserves the semantics of timed automata. Two kinds of glue code are needed to interface with the platform independent code: code implementing the clock source for timed automata and code implementing synchronous channels for communication with the environment. For the clock source, we introduced a platform-specific system call to implement the notion of time that can be used by the platform independent code. In this section, we concentrate on the glue code for external communication, addressing a practical issue in using the TIMES tool. As mentioned earlier, TIMES generates code for a closed system, but we are working with an open system that communicates with its environment.

As in common practice, we modeled the GPCA environment as another automaton in order to close the model. We used UPPAAL channels to capture communication between the GPCA system and its environment. We used a very general environment model, which can send an input action at any moment and is always ready to accept any output action. This allows us to verify whether the safety requirements are satisfied regardless of the user behavior or hardware events.

Of course, the GPCA implementation should not contain code for the environmental model. One approach is to generate the code with both the environmental model and the GPCA model using the TIMES tool, and then eliminate the code of the environmental model. However, it turned out that the code generated by TIMES was tightly coupled, and it was difficult to manually separate out the environment code without affecting the correctness of the GPCA code.

Listing 1: 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 transi-
6            -tion, t', and eval-guard (t') is true
7            assign(t) and assign(t')
8          endif
9        else if t has no channel synchronization}
10         assign(t)
11       endif
12     endif
13   endfor
14 endfunction
```

The *check-trans* function in Listing 1 evaluates transitions defined in the model using the *eval-guard* function. Note that if a transition t in the GPCA model contains channel

synchronization such as *E-RequestBolus*?, *check-trans* tests the complement transition t', which in this case would be defined in the environmental model. This process is described in lines 3-8 of Listing 1. The generated code would have to be modified in many different places to account for communication channels, which communicate with the environment.

We took an alternative approach to overcome the difficulty of decoupling the environment model from the GPCA model. We generated source code only for the GPCA model by replacing the environmental channels with state variables. For example, synchronization of *E-RequestBolus*? on a transition was replaced with a guard, *E-RequestBolus*$== true$, and an update action, *E-RequestBolus*$:= false$. The evaluation of the guard was done in line 3, and the update action was evaluated in lines 9-11 of Listing 1. Then, the environment automaton was removed from the model before code generation. Note that channels internal to the model were not affected and were processed according to the TIMES logic.

In addition, we needed to implement a software routine to receive user events, and interface this routine to the platform-independent code. For this part, another thread $Impl(A_{in})$, where A_{in} defines user input, was created to process a user input at the front end. Listing 2 shows the pseudo-code of $Impl(A_{in})$. After receiving a user request, this front-end thread passes the request to the model thread, denoted as $Impl(M)$, through shared variables. For example, when a user presses the bolus request button, $Impl(A_{in})$ sets *E-RequestBolus* to true. In parallel to $Impl(A_{in})$, $Impl(M)$ waits until a guard of the communication transition becomes true, and resets *E-RequestBolus* to false after processing the bolus request event. However, according to the synchronization semantics, a transition with the synchronization *E-RequestBolus*! can be taken only if the GPCA model is in a state where it can perform *E-RequestBolus*? synchronization. If a pump is not in such a state, the request is ignored. In general, this may lead to desynchronization between the system and its environment; in the case of handling user input, this is appropriate. The $Impl(M)$ thread informs the $Impl(A_{in})$ thread of its state using a shared variable, and checks it in line 4 of Listing 2.

We argue that the generated code preserves the behaviors of the UPPAAL model. Indeed, the interaction with the environment using the shared-variable implementation may happen only when synchronization over communication channels is used in the model. Checking the state of the GPCA model before accepting the request ensures this. By atomically resetting the shared variable, we ensure that no events that should be responded to are missed, and that "old" events that have been already processed would not affect the system again.

Ideally, code synthesis tools should support modular generation. If so, separating the system from its environment, after the closed-system verification, would be straightforward. In our experience, this is the most error-prone aspect of a model-based development that relies on systematic code generation.

Listing 2: glue code for environmental channels

```
1  function external-channel-thread
2    external-event ← recv-external-channel()
3    if (external-event is a bolus request)
4      if (Impl(M) is in the InfusionNormal-
5          Operation state)
6        E-BolusRequest ← true
7      endif
8    endif
9  endfunction
```

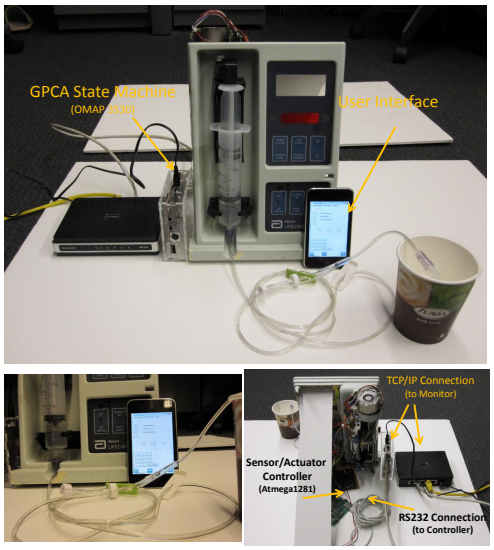

Figure 5: The GPCA Prototype System.

6. TESTBED : THE GPCA PROTOTYPE

Infusion pumps rely on hardware components to reduce the risk of harm, such as stepper motors to administer precise amounts of drugs to patients, sensors to detect an empty reservoir, air-in-line sensors to detect air bubbles in the drug flow, and so on. We built a testbed in order to test our UPPAAL implementation of the GPCA Simulink/Stateflow model. Figure 5 shows the current version of the GPCA infusion pump prototype. This prototype is equipped with software routines that control sensors and actuators. To build it, we obtained a used infusion pump and reused its hardware. The pump contains a stepper motor, with the Atmega1281 processor performing low-level control of the motor. The pump hardware also contains a buzzer that sounds alarms and sensors that detect environmental conditions such as temperature and humidity. In our future work, more sensors will be attached to detect additional infusion problems. Our GPCA software implementation is running on an OMAP3530 processor running Linux OS. POSIX threads are used for parallel executions of the GPCA State Controller $Impl(M)$, the front-end of the environmental channel $Impl(A_{in})$, event logging, and RS-232 communication with the sensors and the motor controller.

Although automatic code generation procedures ensured that our GPCA pump implementation inherited the structure of the UPPAAL model, the existence of the glue code, including code implementing environmental channels and control over actual hardware peripherals, required the final implementation to be comprehensively validated. Since not all of the requirements could be formalized and directly verified against the final implementation, it became necessary to use testing to achieve sufficient confidence in our sys-

tem. Based on this observation, we implemented a tester in a physically isolated system, which facilitated conformance testing in an Internet environment to check the runtime behavior of the GPCA system.

The tester communicates with the GPCA system using a communication protocol over a TCP/IP connection. The protocol provides compact encoding of signals and values exchanged between the tester and the GPCA system. Through this communication protocol, the tester is fully capable of observing the outputs of the GPCA system and providing any stimulus that the GPCA system may expect. The stimuli include both user actions and hardware conditions. In addition, the GPCA system reports the current state to the tester at regular intervals. All states of the GPCA model are encoded as one-byte values in the communication protocol.

Our testing strategy is shown in Figure 2 with dotted lines. Test scenarios are selected based on the GPCA safety requirements. Suppose, the safety requirement, *The pump shall issue an alert if paused for more than t minutes*, is to be tested. This scenario is used to build a test sequence for the model and the implementation. First, the tester drives the GPCA system to a particular state from which the validation of the requirement can start. In particular, if state *V-Init* is needed so that the validation of a certain safety requirement can start, we query the UPPAAL tool to verify the property $A[\ ](!\textit{V-Init})$ against the model. If *V-Init* is reachable from the initial state of the GPCA model, UPPAAL would return a counterexample, which can be used to infer an input sequence to drive the system to *V-Init* [10]. With such a technique, we acquire an input sequence and use it to drive the GPCA system to the *InfusionPaused* state. Once in the *InfusionPaused* state, the tester delays for t_1 minutes, where $t_1 > t$, and watches if the system transitions to the desired *Alrm-TooLongInfusionPause* state. The validation result, as shown in Figure 6, proves that the GPCA system hits the Alrm-TooLongInfusionPause state after the delay and hence conforms to the safety requirement.

Although our current tester implementation relies on a manual testing procedure, many aspects of it can be automated. In Section 8, we explain our future work to extend the manual testing procedure to an automated one with a tool support using UPPAAL-TRON.

7. RELATED WORK

General issues of a model-driven approach are summarized in [15]. There are several approaches that have been proposed to overcome these issues. JAHUEL [6], for example, was proposed as a compilation tool chain to generate code that bridges the semantic gap between platform-independent modeling and platform-specific implementation. However, the system description language of JAHUEL, called FXML, does not support verification of behavioral aspects. A UML-based approach was studied in [8] to model and synthesize code for hard real-time systems. This work proposes to use UML notations to describe a system independent of its particular platform, and then map the description to a platform-specific model before synthesizing code. In particular, this methodology focuses on specifying hard real-time constraints like WCET and upper bounds for reaction times. However, this methodology is not suitable for our case study, because we want to ensure that safety properties are satisfied by all possible executions of the model.

On the other hand, some other approaches strived to enhance the safety in medical devices from the perspective of software product lines. One example is the work presented in [14], which integrates product-line safety analysis with model-based development. This work uses the Rhapsody tool to develop executable UML models for the purpose of modeling and verifying medical device software. Although this work involved modeling, verification and code-synthesis that we discussed in our paper, it did not describe a methodology for how a concrete implementation can be obtained and validated from the model.

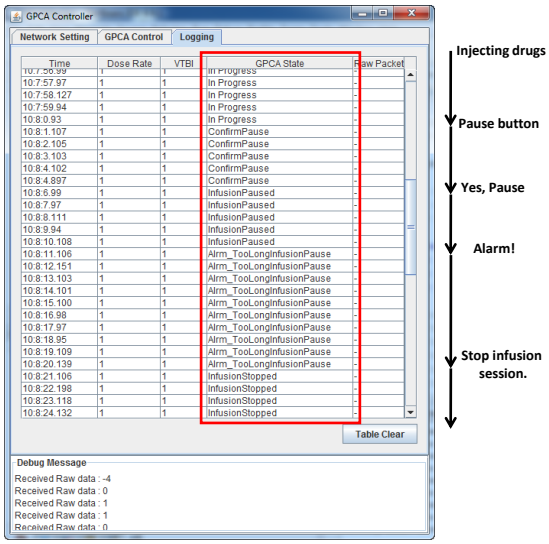

Figure 6: The implementation trace of the GPCA prototype.

8. DISCUSSIONS AND FUTURE WORK

This section discusses a few issues that we faced while applying formal methods to the development of a GPCA infusion pump implementation, and proposes future directions.

The categorization of safety requirements. Our work focused on applying formal techniques to establish a safety-assured implementation from the GPCA reference model (Simulink and Stateflow) and GPCA pump safety requirements, which were provided by FDA, and can be found at [1]. During the process of formalizing the GPCA pump Simulink/Stateflow model, we faced a significant challenge in that the GPCA model lacks sufficient details. This makes it somewhat impractical to verify the model against the generic safety requirements.

As mentioned earlier, not all GPCA requirements are directly formalizable and verifiable. 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 for-

Table 2: Categorization of GPCA Safety Requirements

Category 2	
SR 1.5.4.	Reservoir amount remaining shall be recalculated 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.

malized, 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. This means that we cannot verify all safety requirements just using formal methods.[2] The safety requirements in Category 1 were addressed in Table 1. We discuss Categories 2 through 4 with examples of the safety requirements shown in Table 2.

For safety requirements in Category 2, the GPCA Simulink/Stateflow model is not detailed enough to verify them. A major reason for this is that the GPCA Simulink/Stateflow model relies on a different level of abstraction from that of the safety requirements. For example, the GPCA Simulink/Stateflow model does not describe detailed functions related to the calculation of the remaining reservoir amount, which is mentioned in SR 1.5.4. However, the model can specify a placeholder that guides a programmer to fill the functionality at the implementation level. In particular, the model can provide a function interface, *calculate-remaining-amount()*, as a placeholder on a certain transition or state, and then a detailed calculation procedure can be implemented in this placeholder. SR 1.6.2 cannot be captured by the GPCA model since the behavior of pump stroke is not detailed enough. Unlike SR 1.5.4, the GPCA model should have multiple placeholders that have dependencies on each other in different places to guide a programmer to implement details of the pump stroke and suspension. In this case, rather than using the placeholder approach directly, it makes more sense to extend the model to a lower level, e.g., providing states that describes the pump stroke, so that a placeholder can guide a programmer to implement the corresponding features independent from other placeholders.

Safety requirements in Category 3 cannot be formalized regardless of the abstraction level of the GPCA Simulink/Stateflow 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 the safety requirements in Category 3, developers will need to spend time validating them at the implementation level rather than trying to formalize them. For example, SR 1.4.2 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.

Lastly, safety requirements in Category 4 are too vague, so they can neither be formalized for verification nor be validated at the implementation level. For example, the definitions of *minimal* and *a clear indication* are not clear. These requirements need to be clarified and improved before they can be formally verified at the model level or validated at the implementation level.

Uncertainties in GPCA requirements. As mentioned in Section 4, all safety properties checked on the formalized GPCA UPPAAL model were satisfied. However, we had to make several state and transition changes to the GPCA model to ensure that verification was possible. Interestingly, this does not necessarily mean that the GPCA model is incorrect with respect to the requirements. Instead, it points to the incomplete requirements traceability in the existing GPCA Simulink/Stateflow model. Consider, for example, the following requirement: *If the pump is in a state where user input is required, the pump shall issue periodic alerts every t minutes until the required input is provided.* For each user input, this requirement can be addressed in two ways. The first way is to handle the requirement completely in the user interface component of the pump. In this case, the *State Controller* component will passively wait until the input is provided. Thus the above requirement becomes irrelevant for the case study since the user interface was not modeled. The second way to handle the above requirement is to handle it in the *State Controller*, as the *State Controller* is responsible for keeping track of the pump's status. However, the GPCA model does not provide this functionality for some user events such as $E - ChangeDoseRate$ and $E - StartInfusion$ in its *Infusion Configuration Routine*. To capture this requirement in the GPCA UPPAAL model, we had to add timeout transitions in the respective states. These timeout transitions lead to a new state where the user interface component would be notified to display the alert, and the *State Controller* would return to waiting for the required input.

The necessity of online testing. Model checking cannot completely replace testing, since deployed systems not only interact with real environments, but also contain many factors that cannot be formalized, such as platform-dependent APIs. In general, test suites that form input sequences to the target system are generated based on system requirements or specifications. Then, test-case execution validates whether the system output agrees with the predicted one [13].Our testing methodology introduced in Section 6 manually generates test suites based on the safety requirements. We believe that automation in testing can enhance the contribution to safety in infusion pump systems.

One of the tools that facilitates automated testing is the UPPAAL-TRON tool. It performs conformance testing of timed systems [12] and is based on the same formalism that

[2]We plan to integrate our study into a future release of the GPCA safety requirements and the GPCA model.

we used in our development approach. The environment, e.g., users, is modeled along with the specification, e.g., the GPCA model; this information is fed into UPPAAL-TRON for establishing input-output conformance with the target system, e.g., the GPCA implementation. The UPPAAL-TRON checks whether the GPCA implementation conforms with the model when the implementation is running under a certain environmental assumption defined in the user model.

We expect that this systematic online testing can reduce test costs while providing reasonable test coverage, and ultimately contribute to the safety of infusion pumps.

Improvements for manual translation. Our translation procedure from the GPCA Simulink/Stateflow model to the UPPAAL model is manual because there is no translation tool from Simulink and Stateflow to UPPAAL models. If there were a tool that automated translation between the two tool systems, it could lead to more reliable modeling and verification processes.

9. CONCLUSION

We present our case study which applies model-driven development to GPCA infusion pump safety requirements and the GPCA Simulink/Stateflow model. This case study serves to illustrate how model-driven development can improve the safety of infusion pump systems. Our future work is to extend the current GPCA UPPAAL model to capture the entire GPCA Simulink/Stateflow model. In particular, the *Alarm Detecting Component* in the GPCA model is not discussed in this paper. Another direction for study is to develop a systematic method that can be used to close the gap of code generation/synthesis that is based on the notion of a closed system. This is an important gap to fill since many life-critical systems like medical devices operate in an open environment.

10. ACKNOWLEDGMENT

We would like to thank David Arney for his contribution on the GPCA model and safety requirements.

11. REFERENCES

[1] The generic patient controlled analgesia pump model. http://rtg.cis.upenn.edu/gip.php3.

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

[3] 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.

[4] D. E. Arney, R. Jetley, P. Jones, I. Lee, A. Ray, O. Sokolsky, and Y. Zhang. Generic infusion pump hazard analysis and safety requirements version 1.0. Technical report, University of Pennsylvania, February 2009. Department of Computer and Information Science Technical Report No. MS-CIS-08-31.

[5] D. E. Arney, R. Jetley, P. Jones, I. Lee, and O. Sokolsky. Formal methods based development of a PCA infusion pump reference model: Generic infusion pump (GIP) project. In *Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability (HCMDSS-MDPnP 2007)*, pages 23 – 33, 2007.

[6] I. Assayad, V. Bertin, F. X. Defaut, P.Gerner, O. Quevreux, and S. Yovine. Jahuel: A formal framework for software synthesis. *ECMDA-FA*, 2005.

[7] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems (revised lectures)*, volume 3185 of *LNCS*, pages 200–237, 2004.

[8] S. Burmester, H. Giese, and W. Schafer. Model-driven architecture for hard real-time systems: From platform independent models to code. *ECMDA-FA*, 2005.

[9] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE*, 2007.

[10] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *ICSE*, pages 232–243, 2003.

[11] R. Jetley and P. Jones. Safety requirements based analysis of infusion pump software. In *SMDS*, 2007.

[12] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing real-time embedded software using UPPAAL-TRON - an industrial case study. In *The 5th ACM International Conference on Embedded Software*, 2005.

[13] M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, pages 293–303, 2009.

[14] J. Liu, J. Dehlinger, and R. Lutz. Safety analysis of software product lines using state-based modeling. *The Journal of Systems and Software*, 80:1879–1892, 2007.

[15] D. C. Schmidt. Model-driven engineering. *IEEE Computer Magazine*, February 2006.

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

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

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