IJCSE International Journal of Computer Sciences and Engineering Open Access

Review Paper Vol.-6, Issue-6, June 2018 E-ISSN: 2347-2693

A Model Driven Approach for Risk Reduction in Insulin Pump

Vishal Bhatt[1] , Kapil Kumar Gupta[2] , Nitin Goel[3]

[1,2,3] CSE Department, Shri Ramswaroop Memorial University, Barabanki, UP, India

**Corresponding Author: kapilkumargupta2007@gmail.com*

Available online at: www.ijcseonline.org

Accepted: 11/Jun/2018, Published: 30/Jun/2018

Abstract—This paper presents our effort of using model-driven engineering to establish insulin pump software based on the generic PCA reference model. The reference model was first translated into a network of timed automata using the UPPAAL tool. We applied the TIMES tool to automatically generate platform-independent code as its preliminary implementation. The code is then interface with pump hardware, software and deployed onto a real PCA pump. Experiments show that the code worked correctly and effectively with the real pump. To check the compatibility and rules violation we have also developed a test stub to check the consistency between the proposed model and the code through conformance testing. Challenges faced and their resolution during our work is also discussed in this paper.

Keywords— Model based engineering, Code Synthesis, Insulin Pump, TIMES Tool.

I. INTRODUCTION

Diabetes, which is caused by the unbalance of incretion and the kidney also cannot bear the glucose in blood, is an endocrinopathy with a diabetic from his birth to death. In fact, the most terrible thing is not diabetes itself, but the syndrome, via which the harm of diabetes is incarnated. Insulin has taken a most important place in the treatment for the disease just from the discovery of insulin and has been used for diabetes since 1921. The injection of insulin via a common injector is the only way of treatment in the past. However this method brings a great deal of pain to patients.

During 1960s a concept of uninterrupted injection appeared and replaced the traditional injection approach. In 1970s, the concept was advanced by simulating the exudation of insulin from pancreas. Moreover, an insulin pump was also proposed and developed in our study. With the PZT insulin pump integrated with a silicon micro needle array [1], an intelligent insulin pump was invented. Painless injection; automatic injection has been realized with this equipment.

The control of insulin dosage in the traditional pump appears in our proposed pump, a RF module is added so that a communication can be established. The RF module can accept a signal, sent out by a glucose meter, which denotes blood sugar. A MCU (micro-control unit) in the pump will process the signal and provide a result to the insulin pump. Then pump will adjust the insulin dosage according to the result.

An integrated, forming a closed-loop control system. MDA is proposed in this background. This theory divides the system into three parts by implementation level: PIM, PSM and CODE. PIM is the abstract of system logic, which doesn't include information about implementation technology. PSM is a model of given platform. CODE is implementation code, which is generated by MDA tool or auto program creating software based on PSM. Using MDA method, the system could be realized, integrated, maintained and tested easily. In the process of transforming the PIM into PSM with transformation tool, the validity of PIM is checked by "model checker" firstly, that make sure the PIM's grammatical and semantic is correct and no contradiction or ambiguity. Then, the corresponding relation of UML graph element is checked, the suitable PSM template is selected according to different platform, the PSM that is expressed by XML is generated referring to the format and content of PSM template. In the process of transforming the PSM into CODE with transformation tool, the code according to different platform is generated referring to the format and content of CODE template.
Source Model + Transformation rules = Target Model

II. THE INSULIN MODEL

The **insulin pump** is a medical device used for the administration of insulin in the treatment of diabetes mellitus, also known as continuous subcutaneous insulin infusion therapy. The device includes:

- the pump itself (including controls, processing module, and batteries)
- a disposable reservoir for insulin (inside the pump)
- a disposable infusion set, including a cannula for subcutaneous insertion (under the skin) and a tubing system to interface the insulin reservoir to the cannula.

An insulin pump is an alternative to multiple daily injections of insulin by insulin syringe or an insulin pen and allows for intensive insulin therapy when used in conjunction with blood glucose monitoring and carb counting.

To use an insulin pump, the reservoir must first be filled with insulin. Some pumps use prefilled cartridges that are replaced when empty. Most, however, are filled with the insulin prescribed for the user.

Setting up includes:

1. Opening a new (sterile) empty pump reservoir;
2. Withdrawing the plunger;
3. Inserting the needle into a vial of insulin;
4. Injecting the air from the reservoir into the vial to prevent a vacuum forming in the vial as insulin is withdrawn;
5. Drawing insulin into the reservoir with the plunger, and then removing the needle;
6. Squirting out any air bubbles from the reservoir, and then removing the plunger;
7. Attaching the reservoir to the infusion set tubing;
8. Installing the assembly into the pump and priming the tubing (pushing insulin and any air bubbles through the tubing). This is done with the pump disconnected from the body to prevent accidental insulin delivery;
9. Attaching to the infusion site (and priming the cannula if a new set has been inserted).

The system includes a micro-pump, a drug reservoir, a MCU, a RF module and a silicon needle array shows the schematic structure. There are two work modes of pancreas. One is the basic mode that pancreas secretes insulin uninterrupted with little dose, the other is added mode that insulin dosage is added before each dinner. And a program has been made to simulate the both modes. When the system works on the free inject model. RF receiver obtains much significant information, such as time and the concentration of blood sugar. MCU will count the daily dosage in time and make the system work. All the information will be stored in the EEPROM, which will be provided to doctors later on.[2]

With the Micro-Electro-Mechanical-Systems, a silicon micro needle array has been designed and manufactured. According to the anatomical features of skin, a suitable length in micro-needle array is 150-25um.

The insulin pump consists of the piezoelectric pump, silicon needles array and RF module. It can be conveniently operated from a remote control glucose-meter through a secured RF communication. The application of this insulin pump demonstrates the improved system, convenience and painlessness to the patients.[3]

III. OUR APPROACH

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

A UPPAAL model was constructed using a manual translation process. Along with functional and architecture requirements, they are 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 requirements. Once 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. 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.

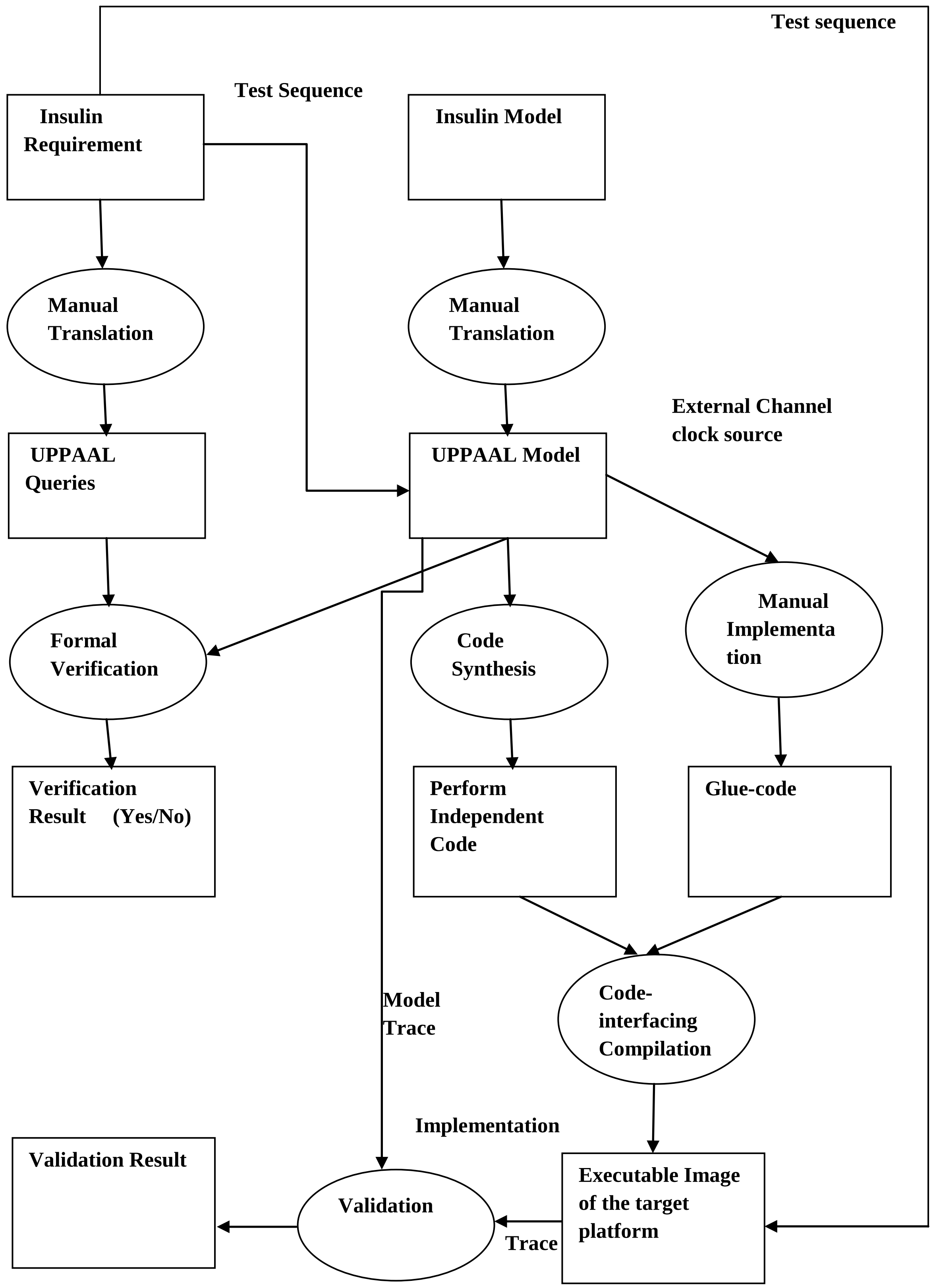

Figure 2: A Model-Driven Development for the Insulin Prototype.

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 insulin code and its platform environment. The glue code is described in order to validate the synthesized insulin 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.

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. They 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.

IV. FORMAL MODELING AND VERIFICATION

A. Formalization of the INSULIN model

We transformed the INSULIN 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 INSULIN pump.

The INSULIN 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. 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 INSULIN 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 INSULIN 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 INSULIN pump 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 INSULIN model.

B. The POST Session

The INSULIN 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.

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 INSULIN model, which includes an alarm state in the POST session that is entered by a timeout transition.

C. The Check Drug Routine

This automation 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.

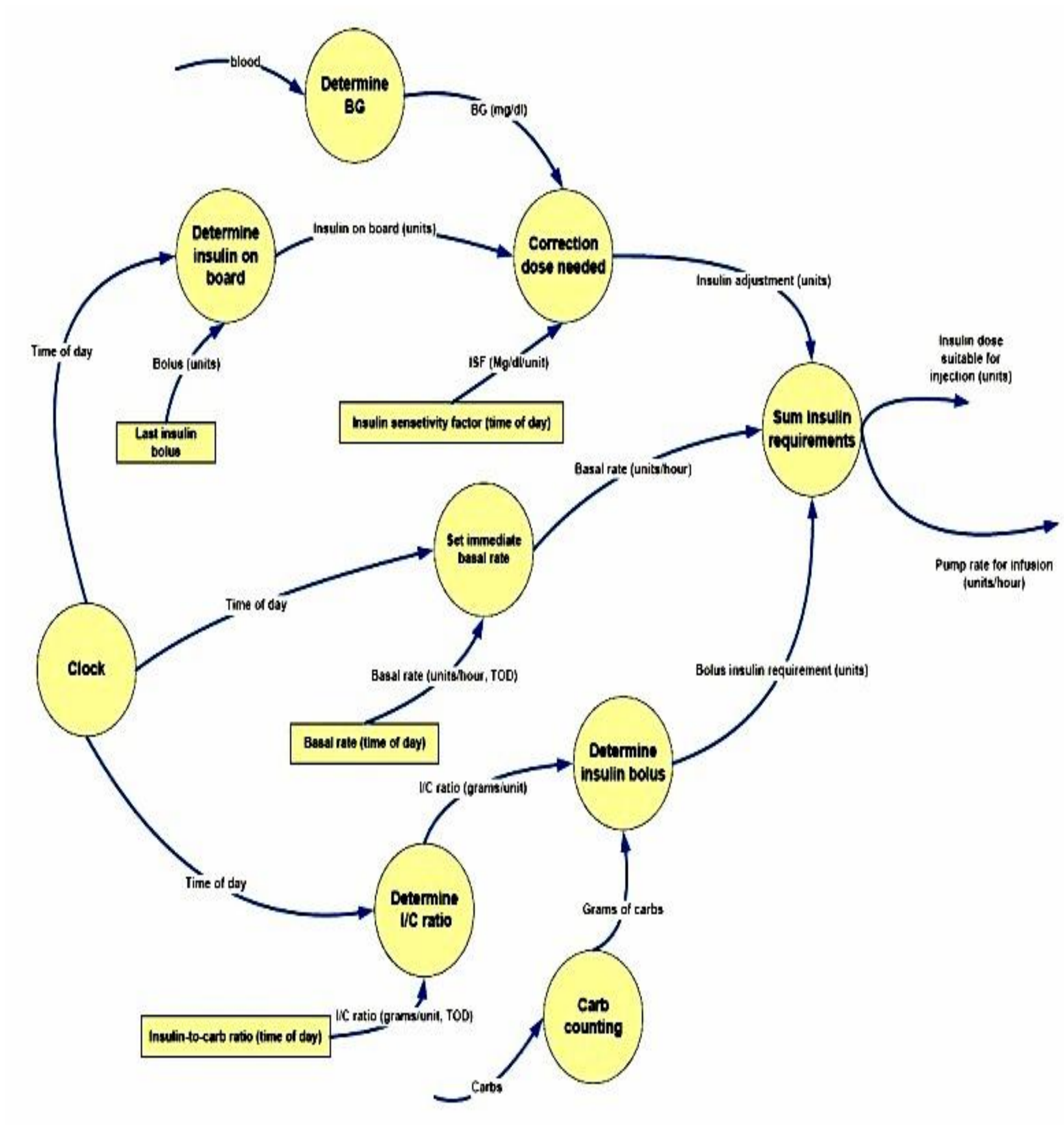

Figure 3: The Insulin Configuration Routine

V. CODE SYNTHESIS AND ADAPTATION

Applying the automatic code synthesis to the INSULIN 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.

A. Implementation with TIMES Tool

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 implementation improves the quality of embedded software in that it reduces human errors while retaining the benefits of model verification. 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 [4]. 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.

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.

B. Interaction 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.

We used UPPAAL channels to capture communication between the INSULIN 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.

Of course, the INSULIN implementation should not contain code for the environmental model. One approach is to generate the code with both the environmental model and the INSULIN 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 INSULIN code.

Listing 1: pseudo-code of *check-trans*

```
1 function check−t r ans
2 for each t r a n s i t i o n t ∈ trans-t ar ray
3 if t i s a c t i v e and eval-guard( t ) i s t rue
4 if t c ont a ins a channel s ynchr oni z a t i on .
5 if the r e e x i s t s a t ’ s complement t r ans i−
6 −t ion , t' , and eval-guard ( t' ) i s t rue
7 assign( t ) and assign( t' )
8 endif
9 else if t has no channel s ynchr oni z a t i on }
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.

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 INSULIN model. We generated source code only for the INSULIN model by replacing the environmental channels with state variables.

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. Listing 2 shows the pseudo-code of Impl(Ain). After receiving a user request, this front-end thread passes the request to the model thread, denoted as Impl(M), through shared variables.

In general, this may lead to desynchronization between the system and its environment; in the case of handling user input, this is appropriate.

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 INSULIN 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

```
function ext e rna l−channel−thread
    ext e rna l−event ← recv−ext e rna l−channel ( )
    if ( ext e rna l−event i s a bolus r eque s t )
        if (Impl (M) i s in the Infus ionNormal−
        Operat ion s t a t e )
            E−BolusRequest ← t rue
        endif
    endif
endfunction
```

C. TESTBED: THE INSULIN PROTOTYPE

Insulin 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. 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 INSULIN software implementation is running on an OMAP3530 processor running Linux OS. POSIX threads are used for parallel executions of the INSULIN State Controller Impl (M), the front-end of the environmental channel Imp (Ain), event logging, and RS-232 communication with the sensors and the motor controller. Although automatic code generation procedures ensured that our INSULIN 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 system.

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 INSULIN system. The tester communicates with the INSULIN 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 INSULIN system. Through this communication protocol, the tester is fully capable of observing the outputs of the INSULIN system and providing any stimulus that the INSULIN system may expect. The stimuli include both user actions and hardware conditions. In addition, the INSULIN system reports the current state to the tester at regular intervals. All states of the INSULIN model are encoded as one-byte values in the communication protocol.

Test scenarios are selected based on the INSULIN pump. Suppose, the, 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 INSULIN 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 can start, we query the UPPAAL tool to verify the property A[ ](!V-Init) against the model. If V-Init is reachable from the initial state of the INSULIN model, UPPAAL would return a counterexample, which can be used to infer an input sequence to drive the system to V-Init. With such a technique, we acquire an input sequence and use it to drive the INSULIN system to the Infusion Paused state.

Once in the Infusion Paused state, the tester delays for t1 minutes, where t1 > t and watches if the system transitions to the desired Alarm-Too Long Infusion Pause state . The validation result proves that the INSULIN system hits the Alarm-Too Long Infusion Pause state after the delay and hence conforms. Although our current tester implementation relies on a manual testing procedure, many aspects of it can be automated. [5]

VI. DISCUSSIONS AND FUTURE WORK

This section discusses a few issues that we faced while applying formal methods to the development of a Insulin infusion pump implementation, and proposes future directions. Looking at the future of insulin sets and canulas is pretty exciting BD working on smaller Pen needles can eventually transfer to infusion sets. [1]

Clinicians are using more intensive insulin strategies to achieve glycemic control. At Childrens Hospital Los Angeles, the use of multiple daily injections (MDI) began increasing in 2001, with the introduction of the first long-acting insulin analog, insulin glargine. Although it has been in use for more than a decade, “intensive” insulin therapy involving the use of insulin pumps has also increased during this time period. At Childrens Hospital Los Angeles, we have been able to achieve lower glycosylated hemoglobin (A1C) levels with fewer hypoglycemic episodes in regular clinical practice using continuous subcutaneous insulin infusion when compared with MDI (although MDI was not always defined as basal-bolus insulin). An advantage that pump users have when compared with patients using MDI is the ability to fine-tune insulin delivery and the flexibility to start or stop insulin delivery on demand with relative discretion.

However, studies have shown that pump therapy has the ability to improve glycemic control and reduce the development of hypoglycemia in pediatric patients with diabetes [9]. More frequent blood glucose monitoring increases the potential for detecting hyperglycemic and hypoglycemic episodes. Furthermore, frequent glucose monitoring is linked to improved glycemic control and has created a demand for advances in the area of glucose monitoring.

Another advantage is the ability to monitor blood glucose readings using an alarm system, which will signal users when readings are low (or high) to prevent significant hypoglycemic events (or to alert patients of significantly elevated glucose levels). Pump therapy has improved glycemic control and reduced hypoglycemic episodes. Despite major improvements, therapeutic goals have not been met. The fear of hypoglycemic events prevents aggressive lowering of A1C levels in pediatric patients with diabetes.[10]

VII. CONCLUSION

Researchers present a case study which applies model-driven development to insulin model. This case study illustrates how model-driven development can improve the risk reduction in insulin pumps. Future work is to extend the current INSULIN UPPAAL model to capture the entire INSULIN model. 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. The sensor-augmented insulin pump was well tolerated by these subjects with minor complaints about tapes and alarms. It provided patients and health care team members with insight into glycemic patterns, which facilitated treatment changes. This work has been supported from the resources of object management group for the development of model driven architecture.
In summary, the insulin pump can offer real-time information for “in-the-moment” diabetes management, as well as feedback for carbohydrate counting and adjustments for physical activity. Treatment can be optimized based on historical data from the sensor-augmented insulin pump downloads, allowing for adjustment of pump settings, calculation of the insulin-to-carbohydrate ratio, and calculation of insulin sensitivity factors. Additional trials are needed to confirm long-term clinical benefits of the sensor-augmented insulin pump system.

ACKNOWLEDGEMENT

This work is supported by resources from Associate Professor Dr. Shalini Agarwal department of Computer Science and Engineering SRMU Lucknow. They would also thank the anonymous reviewers for their significant and constructive critiques and suggestions, which substantially improved the quality of this paper.

REFERENCES

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

[2] Bin Ma and Sheng Liu, "A PZT Insulin Pump Integrated with a Silicon Micro Needle Array for Transdermal Drug Delivery" Fifty sixth IEEE ECTC Conference, San Diego, 2006: 677-681.

[3] The Insulin Model http://www.endocrineweb.com/conditions/diabetes/diabetes-what-insulin

[4] 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.

[5] 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.

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

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

[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] Intensified Treatment of Diabetes in Pediatrics Today http://general-medicine.jwatch.org/cgi/content/full/2011/712/1

[10] Continuous Real-Time Glucose Monitoring for Better Control. http://www.medtronicdiabetes.net/treatmentoptions/continuousglucosemonitoringto

[11] BaekGyu Kim, Anaheed Ayoub, Oleg Sokolsky, Insup Lee, Paul Jones, Yi Zhang, and Raoul Jetley. Safety-Assured Development of the GPCA Infusion Pump Software.The International Conference on Embedded Software 475-487, 2011.

[12] Zhi Xu, Sheng Liu , Zhiyin Gan, Bin Ma, Guojun Liu, Xinxia Cai, Honghai Zhang, Zhigang Yang. An Integrated Intelligent Insulin Pump Electronic Packaging technology, ICEPT 2006.

[13] Frank Truyen, “The Fast Guide to Model Driven Architecture The Basics of Model Driven Architecture (MDA)”, Cephas Consulting Corp January 2006.

[14] Christen Rees et. al., “Recommendations for Insulin Dose Calculator Risk Management”, Journal of Diabetes Science and Technology 2014 Jan; 8(1): 142–149.

[15] Brooke H. McAdams and Ali A. Rizvi, “An Overview of Insulin Pumps and Glucose Sensors for the Generalist”, Journal of Clinical Medicine2016 Jan; 5(1): 5.

[16] Levon Gevorkov , Anton Rassõlkin , Ants Kallaste , Toomas Vaimann, “Simulink based model for flow control of a centrifugal pumping system”, 25th International Workshop on Electric Drives: Optimization in Control of Electric Drives (IWED), PP-1-4, Feb, 2018.

[17] Alessio Bucaioni, Lorenzo Addazi, Antonio Cicchetti, Federico Ciccozzi, Romina Eramo, Saad Mubeen, Mikael Sjödin, “MoVES: A Model-Driven Methodology for Vehicular Embedded Systems” , IEEE Access PP- 6424-6445, Jan 2018.

[18] B. Bakariya, G.S. Thakur, “Effectuation of Web Log Preprocessing and Page Access Frequency using Web Usage Mining”, International Journal of Computer Sciences and engineering, Vol.1 , Issue.1 , pp.1-5, Sep-2013.

[19] K. J. Modi, D.P. Chowdhury, “A Framework for Management and Monitoring of QoS-based Cloud Services”, International Journal of Computer Sciences and engineering, Vol.5, Issue.5, pp.115-119, May-2017.

Authors Profile

Mr. Vishal Bhatt obtained his B. Tech (CS) degree in 2006 from UPTU, Lucknow, Uttar Pradesh, India . He did M. Tech (CSE) from GLA University, Mathura and Astt. Professor in Department of Computer Science, Shri Ramswaroop Memorial University Barabanki,. Uttar Pradesh, India. His main research interest is in the field of Data Mining, Software Engineering approaches etc..

Mr. Kapil Kumar Gupta obtained his B. Tech (IT) degree in 2009 from JSS academy of technical education, Noida, Uttar Pradesh, India . He did M. Tech (CSE) from Integral University, Lucknow and Astt. Professor in Department of Computer Science, Shri Ramswaroop Memorial University Barabanki,. Uttar Pradesh, India. His main research interest is in the field of Image Processing, Software Engineering approaches and Design and analysis of algorithms, etc.. He is a Member of IEEE and CSTA.

Mr. Nitin Goel obtained his B. Tech (CSE) degree in 2006 from ABES, Ghaziabad, Uttar Pradesh, India . He did M. Tech (Advance Network) from IIIT, Gwalior and Astt. Professor in Department of Computer Science, Shri Ramswaroop Memorial University Barabanki,. Uttar Pradesh, India. His main research interest is in the field of Data Mining, Software Engineering etc. He is a Member of IEEE.