Applying SOFL to a Generic Insulin Pump Software Design

Chung-Ling Lin, Wuwei Shen, and Dionysios Kountanis

Department of Computer Science,
Western Michigan University

Abstract. Software embedded into medical devices demands a higher standard on its safety, as compared to most commercial software. One of the most important reasons is that the safety issue should be thoroughly investigated. In the United States, Food and Drug Administration (FDA) is entitled to scrutinize medical devices to ensure they are safe to the public before they enter the market. However, the review of medical device software has been quite challenging because not only the design of medical device software is complicated and error-prone but also the validation of the software system against regulatory requirements is notoriously difficult. Thus, some methodologies based on formal methods have been proposed to alleviate the pain faced by both software developers and regulators such as FDA staff. In this paper, we study how to use the Structured-Object-Based-Formal Language, which is called SOFL to develop a software system controlling an insulin pump, called the *Generic Insulin Infusion Pump* (GIIP). This case study facilitates the understanding of how SOFL can be applied to software systems related to medical devices in terms of the design and review aspects.

1 Introduction

One of the most challenging issues facing the software engineering community is how to develop a software system that, software engineers can guarantee, satisfies the specified requirements. To support any type of guarantee, there is an implicit need to establish sufficient *evidence* that the system will perform dependably, as intended. The quality of this evidence can be a key factor in regulator and third party assessments of dependability claims. This need is particularly important for safety and security critical products such as medical devices.

To complicate matters further, as systems evolve to meet new demands, it is essential to be able to establish evidence that any changes made to the system do not affect the integrity of existing dependability properties and introduce new errors in the process.

Traditional software development methodologies emphasize process oriented practices as a means of assuring design artifacts are complete and consistent. This practice often leads to ambiguities within and between requirements and subsequent specifications. The software development landscape is littered with failures rooted in this practice.

Clearly, it is essential to establish complete and consistent requirement and specifications if a product is to have any chance at meeting it intended use needs. While a quality process is essential to developing a quality product, practices within this process can have a direct bearing on the outcome. Over past decades many technologies have been developed, ranging from structured design and object oriented design, to formal methods based design. Interest in mathematically based design has been a constant and much progress has been made in facilitating development in a practicable manner.

However, in reality, formal methods cannot marry well-established industrial process due to lack of the affordability and efficiency to handle a large scale of specification and proof. The divorce between practical software processes and formal methods has been extensively investigated in the past decades.

Formal engineering method (FEM) [1] has been proposed to bridge the gap between software engineering and formal methods. FME addresses the issue of adapting formal methods for industrial software process so that software engineers can easily grasp formal methods and, at the same time, they do not lose the power of the mathematical support. In this paper, we apply the Structured-Object-based-Formal Language (SOFL) [2] to a case study of the Generic Insulin Infusion Pump (GIIP) [3]. SOFL targets on the unification of mathematical notations and industrial software processes via the application of structured method for requirements analysis and specification and an object-oriented approach for design and implementation.

The software design of a generic insulin infusion pump has been illustrated as a case study to demonstrate how a medical device-based software system can be designed, implemented, and finally reviewed [4]. In the medical device industry, each manufacturer not only has its own requirements on a product such as an insulin infusion pump but also meets regulatory requirements such as safety requirements imposed by FDA. Otherwise, the product cannot be approved for the market. In this case study, we will concentrate on the basal management, part of the GIIP system, to demonstrate how SOFL can be applied to design a software system for a medical device. With the application of SOFL methodology, we help illustrate how a medical device-based software system can be leveraged in terms of design, development, validation and finally regulatory review.

The paper is organized as follows. Section 2 introduces SOFL and GIIP application. A software design based on SOFL for GIIP application is given in Section 3. We draw a conclusion in Section 4.

2 Preliminary

In this section, we briefly introduce SOFL, and then describe the GIIP model.

2.1 Introduction to SOFL

The SOFL is a formal framework that unites formal methods with industrial software development processes. In general, SOFL establishes a structured way to specify the requirements of a software system using an object-oriented approach for subsequent design and implementation based on these requirements. Formal methods can be applied across the entire SOFL-driven development process, to assure high quality specifications and verification at different levels of the intended software system. For example, SOFL allows software engineers to reason about the completeness of software specifications.

A SOFL specification includes a hierarchical condition data flow diagram (CDFD) that links a hierarchy of *specification modules* together. A CDFD is a *directed graph* consisting of *data flows*, *data stores* and *condition processes*. A CDFD describes the static interfaces between components and the dynamical interaction between these components and corresponding data flow. Figure 1 illustrates the basic components of CDFD. The *condition process* is specified with a pre- and post-condition. A *data flow* descriptor identifies how data is exchanged between condition processes. A *data store* defines a variable of a specific type. The *specification module* (s-module) describes the precise functionality of the condition processes in terms of their inputs and outputs. The *s-module* also provides a static definition of all components and details of the system in a textual form.

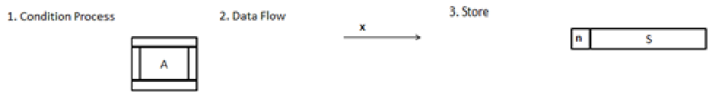


Fig. 1. CDFD Components

2.2 Introduction to GIIP

The Generic Insulin Infusion Pump (GIIP) (safety) model [5] was developed by FDA to be an open system research platform that establishes safety properties generic to insulin infusion pumps. It was envisioned that academics and manufacturers would experiment with the model and share improvements on its design details and experiment with it to help establish new or improved innovative development technologies. All the requirements of the system can be attributed to two categories: functional requirements given in the GIIP Functional Specification Document [6] and safety requirements given in the Safety Document [7].

The software design of GIIP allows a user to program a time period and an insulin infusion rate so a patient receives the administration of insulin via an insulin infusion pump. Based on the specification [6], the software model of GIIP consists of three primary functional modules: delivery control logic, time management, and interface to User Interface (UI) devices. The design of the system concentrates on the delivery control logic module, which includes several major components as illustrated in Figure 2. The Delivery Control Logic is composed of Basal Management, Bolus Management, Pump Delivery Mechanism Interface (PDMI), and Alarm Handler. Also, all relevant events are recorded via Data/Event Logging (DEL).

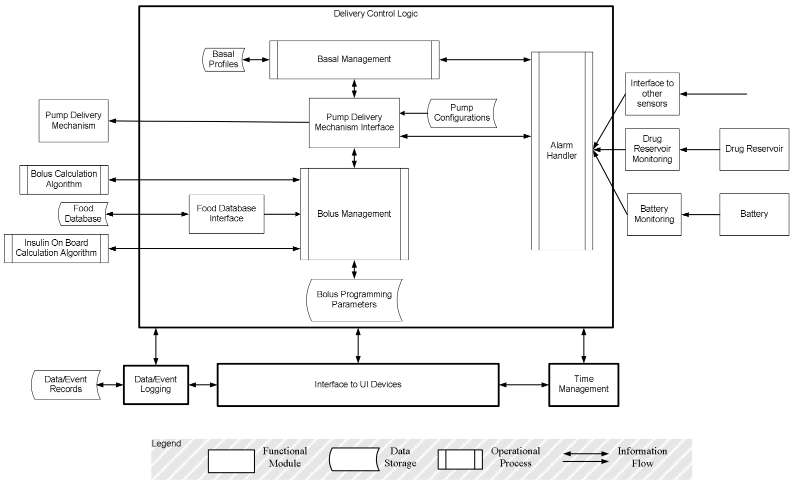


Fig. 2. GIIP Architecture

In this paper, we focus our design on the Basal Management component of the GIIP system. The basal management component is to allow a user/patient to program different insulin infusion rates within 24 hours in a day. Each insulin infuse rate should be given by an effective period– the start time, end time– and the corresponding basal rate, called a *segment*. All insulin infuse rates of a day are programmed to a file, called a *Basal Profile*.

However, sometimes a patient may require special administration of insulin due to some reasons. In this case, the component provides a user with a mechanism to program a high-priority temporary profile, called a *Temporary Basal*, which consists of the duration time and the basal rate. In summary, the basal management component should accomplish two major functionalities: 1) manage basal profiles, and 2) produce the correct information such as the insulin infuse rate based on (normal/temporary) basal profiles to the corresponding component [6].

2.3 GIIP Safety Requirements

The objective of developing the GIIP system is to assure its compliance to a set of core safety requirements, which are articulated to mitigate previous insulin pump failures and other significant safety issues [7]. Throughout this paper, we consider several safety requirements from [7] that govern safe basal administration in GIIP. For the convenience of readers, we reiterate these requirements as follows:

Safety Requirement 1: The pump shall allow the user to program a basal profile with a set of basal rates, ranging from 0.05 to x Units/hour.

Safety Requirement 2: For each basal rate in the profile, the user shall define the duration of the particular rate.

Safety Requirement 3: The pump shall allow the user to set at least two basal profiles at the same time, and require the user to activate no more than one profile at any single point in time.

Safety Requirement 4: The programmed infusion rate of a temporary basal shall not exceed x Units/hour and the duration of a temporary basal shall not exceed y hours.

Safety Requirement 5: The pump shall allow a user to stop a temporary basal while it is in administration.

3 Design of Basal Management Component

In this section, we first outline the structure of the Basal Management component, and then explain in detail how its design is refined to lower levels in a top-down style.

3.1 Top Level Design

In general, the Basal Management component takes the responsibility of managing basal administration according to requests received from the user. The user is allowed to send the component two types of requests regarding basal administration: 1) the BP-req requests, which allow the user to program and manage a basal profile, and 2) the TB-req requests, which allow the user to program and manipulate a temporary basal. Based on such requests, the component decides the current basal rate and outputs it to the insulin delivery mechanism (abstracted as the PDMI component) of the system for delivering insulin. The component also needs to log any changes made to basal administration and report them to the event logging mechanism in the system (abstracted as the DEL component). Figure 3 summarizes, in the format of a CDFD, the interaction that the Basal Management component has with other components in order to fulfill its functionalities.

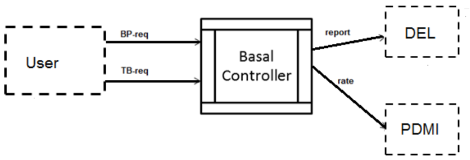
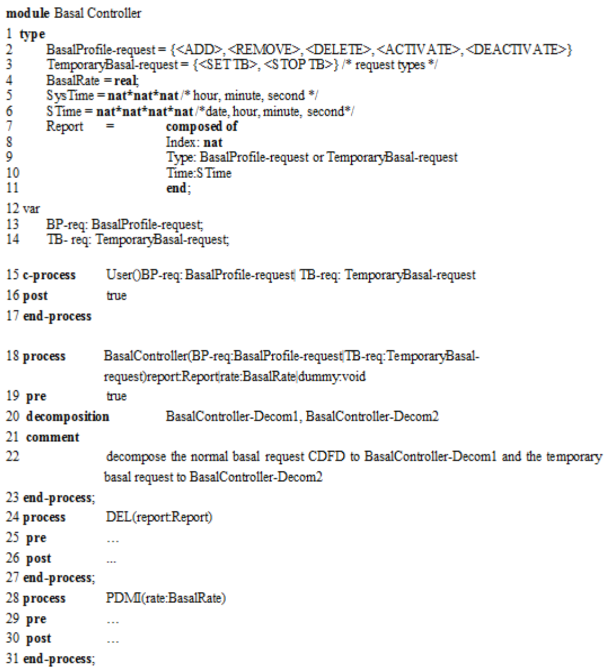


Fig. 3. Top Level CDFD for Basal Management

In Figure 3, a process, Basal Controller, is introduced to represent the Basal Management component. Textual SOFL specifications, as illustrated in Figure 4, are embedded in this process to explicitly define the functionalities of the Basal Management component. As shown in Figure 4, the textual specifications of a process include declaration of variables and data types to be used by the process and its functionalities specified in an object-oriented style.

The user manages basal administration mainly by instructing the Basal Management component to manipulate basal profiles. In particular, the user can request the component to add, delete, update, activate, or deactivate a selected basal profile. To manipulate requests from the user, the component declares an enumeration data type, *BasalProfile-request*, to define the types of operation on basal profiles (see line 2 in Figure 4); and a variable of this type, *BP-req*, to store such request(s) from the user (see line 12 in Figure 4).



```
module Basal Controller
1 type
2       BasalProfile-request = {<ADD>, <REMOVE>, <DELETE>, <ACTIVATE>, <DEACTIVATE>}
3       TemporaryBasal-request = {<SET TB>, <STOP TB>} /* request types */
4       BasalRate = real;
5       SysTime = nat*nat*nat /* hour, minute, second */
6       STime = nat*nat*nat*nat /*date, hour, minute, second*/
7       Report     =          composed of
8                             Index: nat
9                             Type: BasalProfile-request or TemporaryBasal-request
10                            Time:STime
11                            end;
12 var
13      BP-req: BasalProfile-request;
14      TB- req: TemporaryBasal-request;

15 c-process      User()BP-req: BasalProfile-request| TB-req: TemporaryBasal-request
16 post           true
17 end-process

18 process        BasalController(BP-req:BasalProfile-request|TB-req:TemporaryBasal-
                  request)report:Report|rate:BasalRate|dummy:void
19 pre            true
20 decomposition           BasalController-Decom1, BasalController-Decom2
21 comment
22                decompose the normal basal request CDFD to BasalController-Decom1 and the temporary
                  basal request to BasalController-Decom2
23 end-process;
24 process        DEL(report:Report)
25 pre            ...
26 post           ...
27 end-process;
28 process        PDMI(rate:BasalRate)
29 pre            ...
30 post           ...
31 end-process;
```

Fig. 4. Module of Top Level CDFD

With regard to managing temporary basal, the user can instruct the component to either start or stop a temporary basal. Thus, the Basal Controller process declares an enumerate data type *TemporaryBasal-request* , which consists of two possible values *SET TB* and *STOP TB*, to represent the operations on temporary basal administration. A variable *TB-req*, with the type *TemporaryBasal-request*, is declared to record the request(s) from the user regarding temporary basal manipulation.

As aforementioned, any changes to the basal administration, normal or temporary, need to be logged. In Figure 4, lines 7-11, a data type *Report* is declared for such logs. The *Report* type consists of three fields: field *Index* corresponds to the unique index number of the basal profile being affected by the change; field *type* indicates what type of change, *BasalProfile-request* or *TemporaryBasal-request*, happens on the selected basal profile; and field *Time* records the exact time when the change occurs. Notably, field *Time* has the type of *STime*, a quadruple recording the date, hour, minute, and second elements of the time.

3.2 Basal Profile Requests (BP-Req)

It is worth noting that, the textual specifications of a process also needs to define the expected way of how the process's functionalities are decomposed, if the complexity of these functionalities justifies further refinement. Take the Basal Controller process for instance. The functionalities of this process can be generally decomposed into those for normal basal management and for temporal basal manipulation. As shown in the second part of Figure 4[1], lines 20-21 explicitly define such decomposition. We first explain in this section how a lower-level CDFD is designed for managing normal basal profiles, and then explain that for temporary basal in section 3.3.

Figure 5 enumerates variables and data types used in managing normal basal administration. Firstly, a data type, Profile, is declared for basal profiles. The *Profile* type is composed of a set of segments, each of which has the type *Segment*, A basal profile with the type Profile distinguishes itself from others with a unique index number, stored in its *key* field.

Each segment in a basal profile is a combination of its effective period (field *EffectivePeriod*) and the associated basal rate (field *basalrate*), where the effective period is defined as the start and end time of the period. Type *EffectivePeriod* is thus declared as a production of two *Systime*-typed elements. Note that type *Systime* is different from *STime* in that the former has only three fields for hour, minute and second elements, while the latter has an extra field for date.

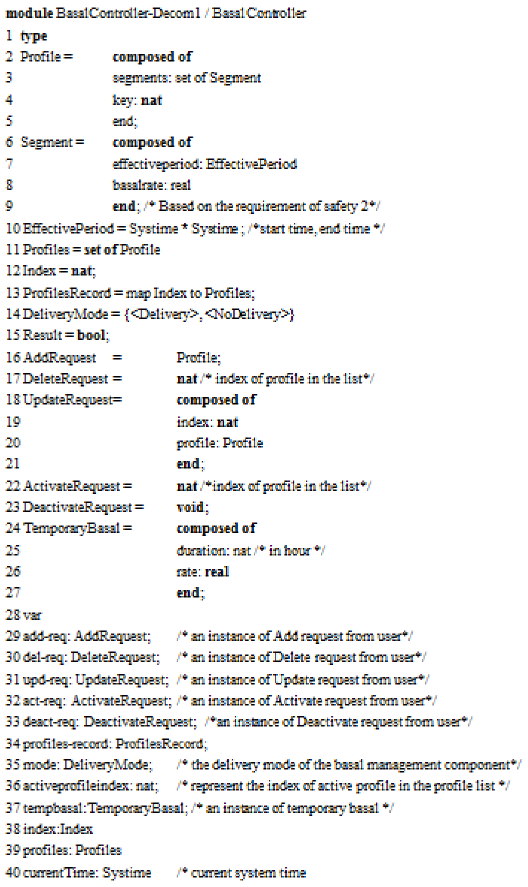
In the system, each basal profile stored is assigned with a unique index number, through which this profile can be fetched, edited, and removed. So, we define type *Profiles* as a *set* of *profile* at line 11 and type *Index* as nature number (denoted as *nat*) to represent the index of a profile at line 12. The basal profiles in the system are stored in variable *profiles,* which is declared at line 39 with type *Profiles.* To map an index number to the corresponding basal profile, a data type, *ProfilesRecord*, is declared at line 13 A a variable *profiles-record* typed as *ProfilesRecord*, is declared at line 34 to store the basal profile fetched based on a user-indicated index number.

With regard to basal delivery, the system takes one of two possible modes at any point of time: Delivery and No Delivery. Thus, line 13 of Figure 5 declares an enumeration type *DeliveryMode* with two values*: Delivery* and *NoDelivery*, and line 35 declares a variable *mode* to store the system's current delivery mode, with the type of *DeliveryMode*.

To eliminate the possible confusion in basal administration, safety requirement 3 enforces that no more than one basal profile be activated at any single point of time. To implement this requirement, we define the variable *activeprofileindex* (line 36 in Figure 5) to maintain the index of the basal profile currently being activated. Apparently, *activeprofileindex* can take only one value at any point of time.

A temporary basal is defined by its duration and its associated temporary basal rate. The data type *TemporaryBasal*, a record type, defined at lines 24-27 of Figure 5, is introduced to represent this fact. The *TemporaryBasal* has a *nat*-typed field

[1] The second part of Figure 4 also includes declaration for the *DEL* and *PDMI* components. Since these two components are not the focus of this paper, we skip their details here.



```
module BasalController-Decom1 / Basal Controller
1 type
2 Profile =         composed of
3                   segments: set of Segment
4                   key: nat
5                   end;
6 Segment =         composed of
7                   effectiveperiod: EffectivePeriod
8                   basalrate: real
9                   end; /* Based on the requirement of safety 2*/
10 EffectivePeriod = Systime * Systime ; /*start time, end time */
11 Profiles = set of Profile
12 Index = nat;
13 ProfilesRecord = map Index to Profiles;
14 DeliveryMode = {<Delivery>, <NoDelivery>}
15 Result = bool;
16 AddRequest    =          Profile;
17 DeleteRequest =          nat /* index of profile in the list*/
18 UpdateRequest=           composed of
19                          index: nat
20                          profile: Profile
21                          end;
22 ActivateRequest =        nat /*index of profile in the list*/
23 DeactivateRequest =      void;
24 TemporaryBasal =         composed of
25                          duration: nat /* in hour */
26                          rate: real
27                          end;
28 var
29 add-req: AddRequest;      /* an instance of Add request from user*/
30 del-req: DeleteRequest;   /* an instance of Delete request from user*/
31 upd-req: UpdateRequest;   /* an instance of Update request from user*/
32 act-req: ActivateRequest; /* an instance of Activate request from user*/
33 deact-req: DeactivateRequest;  /*an instance of Deactivate request from user*/
34 profiles-record: ProfilesRecord;
35 mode: DeliveryMode;       /* the delivery mode of the basal management component*/
36 activeprofileindex: nat;  /* represent the index of active profile in the profile list */
37 tempbasal:TemporaryBasal; /* an instance of temporary basal */
38 index:Index
39 profiles: Profiles
40 currentTime: Systime      /* current system time
```

Fig. 5. Type and Variable Declarations for BP Request

duration to define the duration of a temporary basal in hours, and the *real*-typed field *rate* documents its temporary basal rate (in Unit/Hour). Any temporary basal input by the user is stored in the variable *tempbasal* (line 37 in Figure 5), the type of which is *TemporaryBasal*. The current system time is represented by variable *currentTime,* whose type is *SysTime* given at line 40.

The *BP-Req* type of requests can be further refined to the following five types [6], based on what action the user intends to perform on the basal profiles:

- Request to add a basal profile (denoted as AddBasal)
- Request to delete an existing basal profile (denoted as DeleteBasal)
- Request to update an existing basal profile (denoted as UpdateBasal)
- Request to activate an existing basal profile (denoted as ActivateBasal)
- Request to deactivate the currently activated basal profile (denoted as Deactivate-Basal)

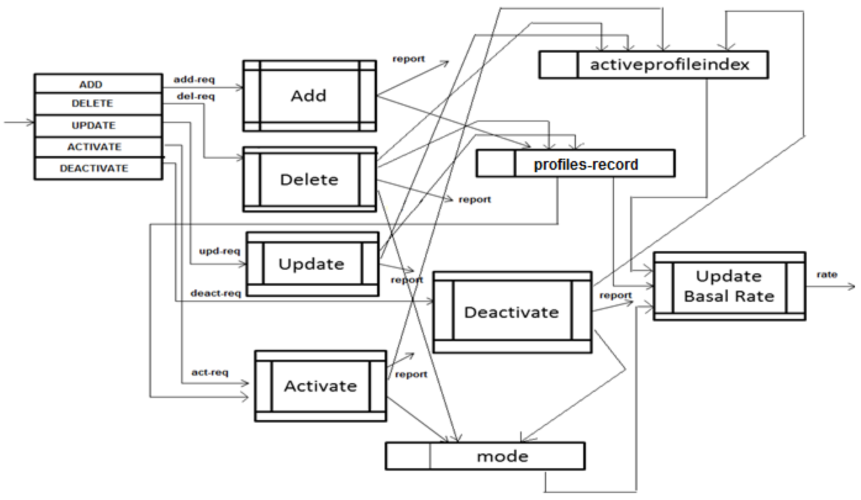


Fig. 6. Decomposition of Basal Controller (BP-Request, Partial)

These five types of requests are declared from line 16 to line 23 in Figure 5. Note that AddBasal requests can be distinguished by the new basal profile(s) they address. Thus, we declare the data type *AddRequest* as equivalent to type *Profile.* The actual AddBasal requests received from the user are stored by variable *add-req*.

DeleteBasal requests, on the other hand, refer to the basal profiles to be deleted by their index numbers. Thus, we declare type *DeleteRequest* as *nat*, while all DeleteBasal requests are stored in variable *del-req*, which is declared at line 30 of Figure 5.

UpdateBasal requests require a user to provide the index of the profile to be updated, as well as the new profile to replace it. Therefore, a record type *UpdateRequest* is declared at line 15 of Figure 5 representing such a request, which consists of two fields: a *nat*-typed field *index* for the index*,* and a *Profile*-typed field *profile* for the new profile. In addition, all UpdateBasal requests from the user are stored in variable *upd-req*.

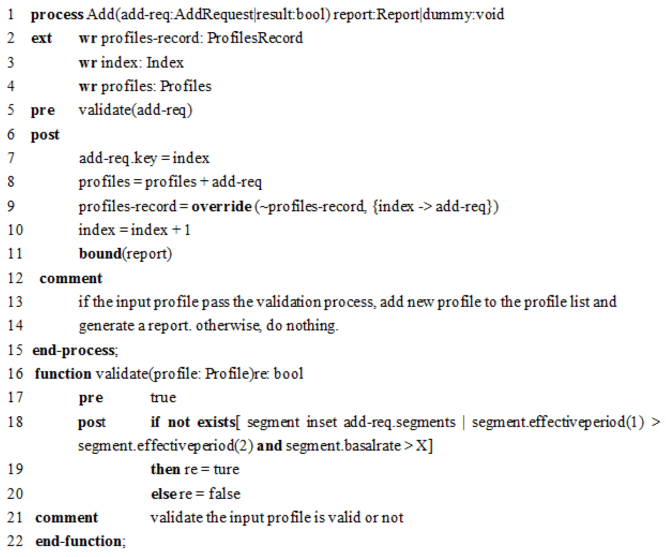
ActivateBasal requests are similar to DeleteBasal requests. Thus, a type *ActivateRequest* is declared for these requests, which is also *nat* to indicate the index of the basal profile under concern. ActivateBasal requests are stored in variable *act-req*.

DeactivateBasal requests do not require any additional parameters. Thus, the type *DeactivateRequest* for these requests is declared as a void type. A variable *deact-req* with type *DeactivateRequest* is declared to store DeactivateBasal requests from the user.

In our GIIP design, the *Basal Controller* component manipulates *BP-Req* requests from the user based on the types of such requests. That is to say, the behavior of the component is decomposed several subsets, each of which corresponding to a particular type of *BP-Req* requests. The CDFD in Figure 6 illustrates such decomposition. Moreover, a process is defined for manipulating each type of requests. The rest of this section discusses the details of all these processes.

Figure 7 depicts the process for handling AddBasal requests. The user can create multiple basal profiles that, if valid, are stored in the basal profiles record in the system. A basal profile is valid if it complies with safety requirements 1-3. Thus, the process in Figure 7 first validates whether an input profile against these safety requirements using the precondition at line 5. The precondition calls the method *validate,* which is defined by lines 16-22. The *validate* method first compares the start time (parameter 1 of *effectiveperiod*) and end time (parameter 2 of *effectiveperiod*) of the effective period of each segment in the basal profile under concern. If the former is before the later, then the method checks if the basal rate associated with this segement is greater than 0 Unit/Hour and less than a thereshold specified by the user (denoted as *X* in Figure 7). Any basal profile failed in these checks is considered as invalid and will be discarded.

If an input profile is valid, the process adds it into the profiles record through the following steps: 1) Add the new basal profile to the set *profiles* at line 8. 2) Point the current index to the new profile and override the previous profiles record using the keyword *override*; 3) Update the index of the next profile by increasing *index* by 1, as shown at line 10, and 4) generate a report on this action and send it to the DEL component.



```
1  process Add(add-req:AddRequest|result:bool) report:Report|dummy:void
2  ext    wr profiles-record: ProfilesRecord
3         wr index: Index
4         wr profiles: Profiles
5  pre    validate(add-req)
6  post
7         add-req.key = index
8         profiles = profiles + add-req
9         profiles-record = override (~profiles-record, {index -> add-req})
10        index = index + 1
11        bound(report)
12  comment
13        if the input profile pass the validation process, add new profile to the profile list and
14        generate a report. otherwise, do nothing.
15 end-process;
16 function validate(profile: Profile)re: bool
17        pre      true
18        post     if not exists[ segment inset add-req.segments | segment.effectiveperiod(1) >
          segment.effectiveperiod(2) and segment.basalrate > X]
19                 then re = ture
20                 else re = false
21 comment         validate the input profile is valid or not
22 end-function;
```

Fig. 7. Module for Add Profile

Figure 8 shows the process responding to DeleteBasal requests. This process first checks the presence of the profile to be deleted in the *profiles* record, as enforced by the pre-condition at line 6. If the profile does not exist in the *profiles* record, the process simply discards the request. Otherwise, it locates the index of the profile and removes it from the record (line 8). The process, like others, also generates a report if a profile is deleted, and sends the report to the DEL component.

Furthermore, if a request of deleting one basal profile is valid, and the profile to be deleted is currently the active profile (i.e., *index* of the request equals to *acitveprofileindex*), then the component needs to conduct the following tasks (as shown from line 9 to line 12 in Figure 8):

1. Deactivate the profile to be deleted without activating another one (i.e., setting the *acitveprofileindex* to -1);
2. If there is no temporary basal currently in process, set the delivery mode to No Delivery, indicating that there is no basal, in any form, currently under administration.

```
1  process Delete(del-req: DeleteRequest)report:Report and
2  ext    wr profiles-record: ProfilesRecord
3         wr activeprofileindex: nat
4         wr mode: DeliveryMode
5         rd tempbasal: TemporaryBasal
6  pre    exists[ r inset dom (profiles-record) | r = del-req]
7  post   let pfile: rng(profiles-record) | pfile.key = del-req in
8         profiles-record = rngdl (~profiles-record, {pfile}) and domdl ({r}, profiles-record)
9         if del-req = activeprofileindex
10        then activeprofileindex = -1 and
11        if tempbasal = nil and activeprofileindex = -1
12        then mode = NoDelivery
13        bound(report)
14 comment        remove the selected profile from profile list and switch delivery mode if necessary
15 end-process
```

Fig. 8. Module for Delete Profile

The *Update* process shown in Figure 9 is defined to manipulate UpdateBasal Requests. Similar to handling DeleteBasal requests, this process first checks if a basal profile with the index indicated by the user exists in the *profiles* record. If not, the process simply ignores the request. Otherwise, the process validates the new basal

```
1  process Update(upd-req:UpdateRequest)report:Report|dummy:void
2  ext    wr profiles-record: ProfilesRecord
3         wr activeprofileindex: nat
4         wr profiles: Profiles
5  pre    exists[ r inset dom (profiles-record) | r = upd-req.index] and
          validate(upd-req.profile)
6  post   let x: dom (profiles-record) | x = upd-req.index in
7         let pfile: rng (profiles-record) | pfile.key= upd-req.index in
8         profiles-record = rngdl (~profiles-record, {pfile})
9         profiles = profiles + upd-req.profile
10        profiles-record = override (~profiles-record, {x -> ~profiles-record(x)})
11        bound(report)
12  comment
13        if the input profile pass the validate process, update new profile to the profile list and
14         generate a report
15        otherwise, do nothing.
16 end-process;
```

Fig. 9. Module for Update Profile

profile by calling the *validate* method. If the new basal profile is valid, the process replaces the previous profile in the *profiles* record that has the user-specified index number with the new basal profile. If this happens, a report is generated and sent to the DEL component for logging.

Since the profiles record may contain multiple profiles, a user can activate one of them and use it to decide the output basal rate. The *Activate* process, as illustrated in Figure 10, is crafted to assist the user in dosing so. This process first checks whether or not the selected basal profile, i.e., parameter *act_req*, exists in the profiles record (line 5). If yes, the process updates the active profile variable *activeprofileindex* to the index of the selected profile, *act-req,*(line 6). After this, the system switches to the *Delivery* mode (line 7), and generates a log and feeds it to the DEL (line 8).

```
1  process Activate(act-req:ActivateRequest)report:Report
2   ext    wr activeprofileindex: nat
3          wr mode:DeliveryMode
4          rd profiles-record: ProfilesRecord
5   pre    exists [pfile inset rng (profiles-record) | act-req = pfile.key]
6   post   activeprofileindex = act-req
7          mode = “Delivery”
8          bound(report)
9   comment
10         set the active profile.
11 end-process
```

Fig. 10. Module for Activate Profile

The system allows a user to deactivate an active profile, a feature implemented by the *Deactivate* process in Figure 11. This process first voids the *activeprofileindex* variable by settings it to -1 (line 6) , and then, if there is no temporary basal defined, switches to the *No delivery* mode (line 8). The entire process is recorded in a log sent to the DEL (line 9).

```
1  precess Deactivate(decatc-req:DeactivateRequest)report:Report
2   ext    wr activeprofileindex:nat;
3          wr mode:DeliveryMode
4          rd tempbasal:TemporaryBasal
5   pre    true
6   post   activeprofileindex = -1
7          if tempbasal = nil
8          then mode = “NoDelivery”
9          bound(report)
10   comment
11         Switch the delivery mode based on the safety requirement
12   end-process
```

Fig. 11. Module for Deactivate Basal Rate

In order to decide the actual basal rate, we define the *UpdateBasalRate* process that calculates the actual basal rate continuously during the system execution. The process is shown in Figure 12 based on the following rules:

1. If the delivery mode is *NoDelivery,* the basal rate is 0;
2. Otherwise, if there is a temporary basal in progress, then the basal rate is set as the temporary basal rate;
3. Otherwise, the process should find the active profile (if any) from the *profiles* record, decide the right segment in the active profile that covers the current time, and use the basal rate associated with this segment as the actual basal rate.

```
process UpdateBasalRate()rate:BasalRate
ext     rd profiles-record: ProfilesRecord
        rd tempbasal:TemporaryBasal
        rd activeprofileindex: nat
        rd mode:DeliveryMode\
        rd currentTime
pre     true
post    if mode = "NoDelivery"
        then rate = 0;
        else if tempbasal != nil
        then rate = tempbasal.rate
        else if activeprofileindex = 0
        then rate = 0
        else let pfile: rng(profiles-record) | pfile.key = activeprofileindex
        rate = getrate(pfile)
        bound(rate)
        comment
        Calculate the basal rate and send the rate to PDMI component
end-process
function getrate(pfile: Profile) re: basalrate
        if exists[ segment inset pfile.segments | segment.effectiveperiod(1) <= currentTime and
        segment.effectiveperiod(2) > currentTime]
        then        re = segment.basalrate
end-function
```

Fig. 12. Module for Update Basal Rate

3.3 Temporary Basal Requests (TB-Req)

A user can instruct the system to start or stop a temporal basal. In particular, the user can send to the system a *SET TB* request for programming and start a temporary basal. Or he/she can request to stop the currently ongoing temporary basal with *STOP TB* requests. In particular, a *SET TB* request carries a temporal basal (duration and temporary rate) as the parameter, while a STOP TB request has no parameter.

Figure 13 illustrates how the GIIP system should respond to these types of request, where a Set TB process is introduce to manipulate *SET TB* requests and a Stop TB process to manipulate *STOP TB* requests.

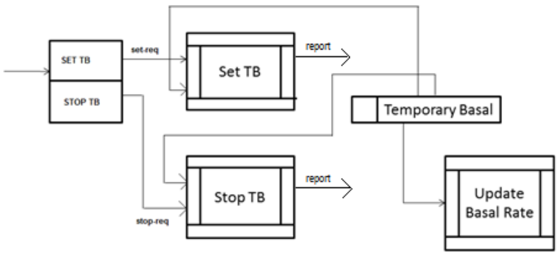
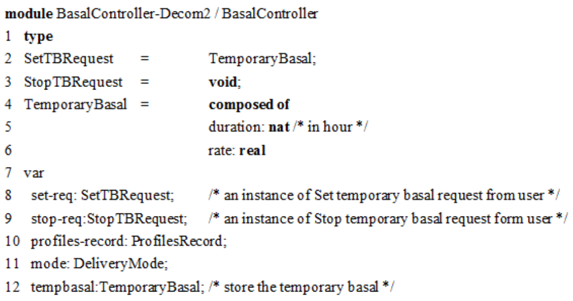


Fig. 13. Basal Controller Design for Temporary Basal Requests

Figure 14 provides type and variable declarations for the GIIP design with regard to managing temporary basal requests, where the types *SetTBRequest* and *StopTBReques* are defined for the two types of requests related to temporary basal, respectively. Note that type *StopTBRequest* is actually a void type, as it does not require any parameter to stop the current temporary basal. Variables *set-req* and *stop-req* are declared to stop the temporary basal requests from the user.



```
module BasalController-Decom2 / BasalController
type
SetTBRequest     =        TemporaryBasal;
StopTBRequest    =        void;
TemporaryBasal   =        composed of
                          duration: nat /* in hour */
                          rate: real
var
 set-req: SetTBRequest;      /* an instance of Set temporary basal request from user */
 stop-req:StopTBRequest;     /* an instance of Stop temporary basal request form user */
 profiles-record: ProfilesRecord;
 mode: DeliveryMode;
 tempbasal:TemporaryBasal; /* store the temporary basal */
```

Fig. 14. Type and Variable Declarations for TB Request

A process SetTB, as shown in Figure 15, is defined to specify the process for the GIIP to respond to a SET TB request from the user. As imposed by safety requirement 4, any new temporary basal that the user intends to initial, should have a temporary basal rate not greater than x Units/hour and a duration not greater than y hours (both x and y are thresholds pre-specified by the user). The *SetTB* process first checks to assure that no other temporary basal is currently stored in the system (see line 3 of Figure 15), and then checks whether the input temporary basal is valid by calling the *validateTB* method. The *validateTB* method, defined from line 8 to line 12 in Figure 15, checks whether or not the configuration of the input temporary basal is within the ranges prescribed in safety requirement 4. . If the input temporary basal is valid, the component stores it in variable *tempbasal*, generates a report indicating that the input temporary basal is initiated, and feeds the report to the DEL componet . Once variable *tempbasal* is set, the *UpdateBasalRate* process will update the output basal rate according to the temporary basal.

```
process SetTB(set-req:SetTBRequest)report:Report
ext    wr tempbasal:TemporaryBasal
pre    tempbasal = nil and
       validateTB(set-req)
post   tempbasal = set-req
end-process
function validateTB(tempbasal:TemporaryBasal)re: bool
pre    ture
post   if 0.05<= tempbasal.rate and tempbasal.rate < X and
       tempbasal.duration < Y
       then re = true
       else re = false
       bound(report)
comment
       Validates the temporary basal based on safety requirement 4
end-function
```

Fig. 15. Module for SetTB

In terms of stopping the current temporary basal, the *StopTB* process, shown in Figure 16, simply clears variable *tempbasal* by setting it to *nil*.

```
process StopTB(stop-req:StopTBRequest)report:Report
ext    wr       tempbasal:TemporaryBasal
pre    ture
post   tempbasal = nil
       bound(report)
end-process
comment         allow the user to stop the temporary basal(safety 5)
```

Fig. 16. Module for Stop TB Request

4 Related Work

To assure the correctness of safety-critical software systems, many formal methods based approaches have been proposed in last few decades, including Alloy [8], ASM [9], B [10], and Z [11]. All of these approaches are built on solid mathematical foundation. Although such solid mathematical foundation enable formal verification to be conducted on software systems thus developed, it also restricts the applicability of these approaches in industrial development practices. That is to say, developers who intend to take advantage of these approaches, they have to first become familiar with the mathematical foundation underlying them. Consequently, the extra learning curve and sophisticate mathematical background hinder engineers from applying these approaches in real industrial practices.

5 Conclusion

The application of formal methods in industrial practices has been hurdled by both the steep learning curve to master these methods and computational expressiveness underlying these methods. The SOFL methodology intends to overcome these hurdles by integrating together a formal representation framework and an object-oriented development process. It has been proven as effective when applied to various applications [12].

In this paper, we applied the SOFL methodology to develop the Generic Insulin Infusion Pump design. While concentrating on the Basal Management component in the GIIP system, our work can easily be extended to the rest of the GIIP system. More importantly, this case study helps us understand how SOFL leverages the development of a complex software system, such as medical device software, from the following aspects:

1. A software development process driven by the SOFL helps to capture the traceability [13] among different software artifacts, such as SOFL specifications and SOFL implementation modules. Good quality traceability information can greatly improve the correctness and maintainability of complex software systems. For example, the missing trace information from a requirement to a design element indicates that the requirement is more likely missed in the system [14]. Moreover, good traceability information can help third-party reviewers understand such systems with less effort and more accuracy.
2. The formal and practical aspects of the SOFL methodology introduce more rigorousness to the development of complex software systems. Formal verification can be applied to inspect the correctness and consistency of SOFL specifications for these systems, which in turn helps developers as well as third-party reviewers establish higher confidence in these systems.

As the future work, we plan to apply the SOFL methodology to the rest of the GIIP system, and to investigate effective verification techniques for inspecting SOFL specifications.

Acknowledgements. The authors would like to thank Drs. Yi Zhang and Paul Jones at FDA who provided valuable and helpful suggestions about the GIIP project, and the great comments and feedback about this paper. Last, the first author appreciates their guidance of the GIIP project during his internship in summer 2011.

References

[1] Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. Springer (2004) ISBN 3-540-20602-7

[2] Liu, S., Offutt, J., Ho-Stuart, C., Sun, Y., Ohba, M.: SOFL: A Formal Engineering Methodology for Industrial Applications. IEEE Transactions on Software Engineering 24(1), 24–45 (1998)

[3] Zhang, Y., Jones, P., Jetley, R.: A Hazard Analysis for a Generic Insulin Infusion Pump. Diabetes Science and Technology 4(2) (2010)

[4] Vogel, D.: Medical Device Software Verification, Validation, and Compliance. Artech House (2011)

[5] Generic Infusion Pump Project, `http://rtg.cis.upenn.edu/gip.php3`

[6] FDA, GIIP Functional Specifications (2011)

[7] Zhang, Y., Jetly, R., Jones, P., Ray, A.: Generic Safety Requirements for Developing Safe Insulin Pump Software. Diabetes Science and Technology 5(6), 1403–1419 (2011)

[8] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006) ISBN 978-0-262-10114-1

[9] Gurevich, Y.: Evolving Algebras. In: Specification and Validation Methods, pp. 9–36. Oxford University Press (1995) ISBN 0-521-49619-5

[10] Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996) ISBN 0-521-49619-5

[11] Schuman, S.A., Meyer, B., Abrial, J.-R.: A Specification Language. In: McKeag, R.M., Macnaghten, A.M. (eds.) On the Construction of Programs. Cambridge University Press (1980)

[12] Liu, S., Stavridou, V., Dutertre, B.: The Practice of Formal Methods in Safety Critical Systems. Journal of Systems and Software 28(1), 77–87 (1995)

[13] Spanoudakis, G., Zisman, A.: Software Traceability: A Roadmap. In: Handbook of Software Engineering and Knowledge Engineering. World Scientific Publishing (2004)

[14] Yadla, S., Huffman Hayes, J., Dekhtyar, A.: Tracing Requirements to Defect Reports: An Application of Information Retrieval Techniques. Innovations in Systems and Software Engineering: A NASA Journal 1, 116–124 (2005)