Use of Tabular Expressions for Refinement Automation

Neeraj Kumar Singh[1(✉)], Mark Lawford[2], Thomas S.E. Maibaum[2], and Alan Wassyng[2]

[1] INPT-ENSEEIHT/IRIT, University of Toulouse, Toulouse, France
nsingh@enseeiht.fr
[2] McMaster Centre for Software Certification, McMaster University, Hamilton, Canada
{lawford,wassyng}@mcmaster.ca, tom@maibaum.org

Abstract. We aim to develop sound and effective techniques to automate formal modelling and refinement from tabular expressions using a *correct-by-construction* approach. In this work, we present a refinement strategy to generate formal models from tabular expressions, as they can be used in the Event-B modelling paradigm. The proposed refinement strategy permits us to develop an abstract model using tabular expressions and a series of Event-B models using refinement from the set of tabular expressions. Further the proofs associated with the refinement strategy used to generate the model are examined through the Rodin tools. Our work is an important step towards eliciting patterns of automatic refinement for Event-B models from tabular expressions and to meet the properties of *completeness* and *disjointness* in a rigorous manner. To assess the effectiveness of our proposed approach, we use a medical device case study: the *Insulin Infusion Pump (IIP)*.

Keywords: Tabular expression · Event-B · Refinement · Formal methods · Verification · Validation · Insulin Infusion Pump

1 Introduction

Requirement engineering (RE) provides a framework for a better understanding of system requirements by simplifying system complexity using formal and informal techniques. It plays an important role in analyzing system requirements, and functional and non-functional system behaviours to achieve the properties of consistency, unambiguity and completeness. Tabular expressions [1] support a technique for requirement engineering that uses (potentially complex) relations for documenting and analysing system requirements, in order to define them precisely and concisely. It is a visual representation of functions in a tabular layout that has a precise semantics and a formal notation. Moreover, this tabular representation of system requirements satisfies the important properties of *disjointness* and *completeness*.

Y. Ouhammou et al. (Eds.): MEDI 2017, LNCS 10563, pp. 167–182, 2017.
DOI: 10.1007/978-3-319-66854-3_13

On the other hand, formal methods have been applied successfully to design and develop critical systems, such as avionics, medical and automotive [2,3]. In particular, formal methods have been used to check functional requirements and safety requirements by developing system models. In formal modelling, refinement plays an important role for handling a large complex system by developing the whole system incrementally, in which each incremental step can be used to introduce new functionalities while preserving the required safety properties.

Practicalities of performing automatic refinements are largely an open problem. It is, clearly, unrealistic to carry out such refinements entirely by hand, which is well illustrated by the complexity of the examples in [4,5]. Some refinement steps are, however, inherently difficult to automate. Our work highlights how automation is feasible to guide the refinement process.

Our primary contribution in this paper is to propose a refinement strategy that can help automate the process of formalizing system requirements from tabular expressions using a *correct-by-construction* approach. We show how the refinement strategy can be used to transform tabular expressions into formal models that aid in determining the correctness of functional behaviour, and the modelling structure of a system. The refinement approach allows us to build a formal model incrementally, where the first model represents only abstract behaviour, and the incremental models are enriched by more concrete behaviours. The generated formal models are used later to define safety properties and to check system consistency using formal verification. To achieve our goal, we select the Event-B modelling language that allows incremental refinement based on a *correct-by-construction* approach for generating formal models from tabular expressions.

To assess the proposed incremental refinement strategy in Event-B, we apply it to design and to formally specify an *Insulin Infusion Pump (IIP)*. First, the informal IIP requirements are described in tabular expressions that are used later for producing the formal models. In the IIP case study, we verify functional behaviours including various system operations, that are required to maintain insulin delivery, user profile management, and the calculation of required amounts of insulin. The complete formal development builds incrementally-refined models of IIP, formalizing the required functional behaviour by preserving its required safety properties. We also use the Rodin [6] tools to check the generated formal models. The added contributions of this article can be summarised as follows: (1) proposing a refinement strategy for generating formal models from tabular expressions; (2) discussing in detail opportunities and ramifications for automating the application of the refinement strategy; (3) presenting validation of the proposed refinement strategy by discharging the proof obligation of the generated models using Rodin tools; and (4) applying the refinement strategy to the *Insulin Infusion Pump (IIP)* case study.

The structure of the article is as follows. In Sect. 2, we review preliminary material: tabular expression and the modelling framework. Section 3 presents a refinement strategy for generating the formal models from tabular expressions. Section 4 presents an example that illustrates the application of the refinement

strategy: the *Insulin Infusion Pump (IIP)*, including model analysis. Section 5 presents related work, and in Sect. 6, we conclude the paper and discuss with future work.

2 Preliminaries

2.1 Tabular Expressions

In the late 1970s, Parnas et al. [1,7] used tables to specify the software system requirements for expressing complex behaviours through organizing the relation between input and output variables. These tables were used simply to describe system requirements unambiguously. Parnas formally defined ten different types of tables for different purposes using functions, relations, and predicates [1]. Parnas also called these tables tabular expressions because the tables use mathematical expressions and recursive definitions. Some foundational works reported on formal semantics, table transformation, and composition of tables [8,9]. The formal semantics of tables specify the precise meaning that helps to maintain the same level of understanding when tables are used by various stakeholders. Similarly, table transformation can be used to derive a desired system behaviour under various system situations, and the composition of tables can be used to integrate different tables to obtain the final complex behaviour. These tables have been used in several safety-critical projects such as by Ontario Hydro for the Darlington Nuclear Shutdown Systems [10,11], and the US Naval Research Laboratory [12], etc.

Tabular expressions [7] are not only effective visually and as a simple approach to documenting system requirements by describing conditions and relations between input and output variables, they also facilitate preserving essential properties like *completeness* and *disjointness*, which are described as follows:

- ***Disjointness:*** *requires that the conditions (c_i) in columns (rows) do not overlap, which can be formalised as* $\forall i, \forall j (i \neq j \Rightarrow \neg(c_i \wedge c_j))$.
- ***Completeness:*** *requires that the conditions in columns (rows) cover all the input possibilities, which can be formalised as* $(c_1 \vee c_2 \ldots \vee c_n) \equiv TRUE$.

In our work, for generating formal models from tabular expressions using our refinement strategy, we use *horizontal condition tables (HCT)*. An HCT table contains of a group of columns for input conditions and a group of columns for output results. However, the input column may be sub-divided to specify multiple sub-conditions. The tabular structure highlights the structure of predicates, and adjoining cells are considered to be ANDed so that can be interpreted in the tabular structure as a list of "*if-then-else*" predicates.

2.2 The Modelling Framework

In this section, we summarize the Event-B modelling language [13]. The Event-B language has two main components: *context* and *machine*. A *context* describes

the static structure of a system, namely *carrier sets* and *constants* together with *axioms* and *theorems* stating their properties. A *machine* defines the dynamic structure of a system, namely *variables, invariants, theorems, variants* and *events*. Terms like *refines, extends*, and *sees* are used to describe the relation between components of Event-B models. Events are used in a *machine* to modify state variables by providing appropriate *guards*.

Modelling Actions over States. The event-driven approach of Event-B is borrowed from the B language [14]. An Event-B model is characterized by a list of *state variables* possibly modified by a list of *events*. An invariant $I(x)$ expresses required safety properties that must be satisfied by the variable x during the activation of events. An event is a state transition in a dynamic system that contains *guard(s)* and *action(s)*. A *guard*, a predicate built on the state variables, is a necessary condition for enabling an event. An *action* is a generalized substitution that describes the ways one or several state variables are modified by the occurrence of an event. There are three ways to define an event e. The first is BEGIN $x : |(P(x, x')$ END, where the *action* is not guarded and the action is always enabled. The second is WHEN $G(x)$ THEN $x : |(Q(x, x'))$ END, where the *action* is guarded by G, and the *guard* must be satisfied to enable the *action*. The last is ANY t WHERE $G(t, x)$ THEN $x : |(R(x, x', t))$ END, where the *action* is guarded by G, and it depends on the local state variable t for describing non-deterministic events. Event-B supports several kinds of proof obligations like invariant preservation, non-deterministic action feasibility, guard strengthening in refinements, simulation, variant, well-definedness etc.

Invariant preservation (see INV1 and INV2 below) ensures that each invariant is preserved by the INITIALIZATION event $Init(x)$ and other model events $BA(e)(x, x')$; non-deterministic action feasibility (FIS) shows the feasibility of the event e with respect to the invariant I; guard strengthening in a refinement ensures that the concrete guards in a refining event are stronger than the abstract ones; and simulation ensures that each action in a concrete event simulates the corresponding abstract action.

$$
\begin{array}{l}
INV1 : Init(x) \;\Rightarrow\; I(x) \\
INV2 : I(x) \;\wedge\; BA(e)(x, x') \;\Rightarrow\; I(x') \\
FIS \;\; : I(x) \;\wedge\; \mathsf{Grd}(e)(x) \;\Rightarrow \exists y.BA(e)(x, y)
\end{array}
$$

Model Refinement. A model can be refined to introduce new features or more concrete behaviour of a system. The Event-B modelling language supports a stepwise refinement technique to model a complex system. The refinement enables us to model a system gradually and provides a way to strengthen invariants thereby introducing more detailed behaviour of the system. This refinement approach transforms an abstract model to a more concrete version by modifying the state description. The refinement process extends a list of state variables by refining each abstract event to a corresponding concrete version, or by adding new

events. These refinements preserve the relation between an abstract model and its corresponding concrete model, while introducing new events and variables to specify more concrete behaviour of the system. The abstract and concrete state variables are linked by *gluing invariants*. The generated proof obligations ensure that each abstract event is correctly refined by its concrete version. For instance, an abstract model AM with state variable x and invariant $I(x)$ is refined by a concrete model CM with variable y and gluing invariant $J(x, y)$. e and f are two events of the abstract model AM and concrete model CM, respectively. Event f refines event e. $BA(e)(x, x')$ and $BA(f)(y, y')$ are predicates of the events e and f, respectively. This refinement relation generates the following proof obligation:

$$I(x) \ \wedge \ J(x, y) \ \wedge \ BA(f)(y, y') \ \Rightarrow \ \exists x' \cdot (BA(e)(x, x') \ \wedge \ J(x', y'))$$

A set of new events introduced in a refinement step is viewed as hidden events, which are not visible to the environment of the system being modelled. These introduced events are outside of the control of the environment. Newly introduced events refine *skip* and are not observable in the abstract model. Any number of executions of an internal action may occur in between each execution of a visible action. This refinement relation generates the following proof obligation:

$$I(x) \ \wedge \ J(x, y) \ \wedge \ BA(f)(y, y') \ \Rightarrow \ J(x, y')$$

The refined model reduces the degree of non-determinism by strengthening the guards and/or predicates. The refinement of an event e by an event f means that the event f simulates the event e, which guarantees that the set of traces of the refined model contains (up to stuttering) the traces of the resulting model. The Rodin platform provides a set of tools to support project management, model development, proof assistance, model checking, animation and automatic code generation.

3 Refinement Strategy

A common way of constructing a formal specification is to start from a very simple abstract model that captures only basic system behaviour, and to add new features or system requirements to the abstract model to develop a concrete system by satisfying the additional requirements. An extension is a set of new features and system requirements that always zoom into a detailed system behaviour without changing the original abstract behaviour. We refer to this type of modelling method as *superpositioning* [15].

Superposition seems to be a good candidate in the field of formal modelling, because it allows us to construct a complicated formal specification by incremental refinement steps. Each new refinement step always focuses on a single design decision. In other words, it permits us to tackle one issue at a time, rather than having to make a joint design decision and settle a number of interrelated design questions [15].

Here, we describe a refinement strategy for generating formal models from documented system requirements. Our objective is to formalize tabular expressions and then define safety properties for the developed models to verify the documented system requirements. To produce formal models from tabular expressions is not an easy task due to no refinement relation among the tables, lack of techniques to support table compositions and implicit information about correct ordering of system behaviour. In order to produce formal models from tabular expressions, we propose a refinement strategy that allows us to construct a model progressively by traversing tabular requirements using a *correct-by-construction* approach. The proposed strategy can be suitable for any formal language that can support refinement based development. As mentioned, we use the Event-B modelling language, which supports refinement based progressive development. A formal definition of the transformation rule for the proposed refinement strategy is given below.

Definition 1. *Let T be a set of tabular expressions, in which each tabular expression satisfies the properties of disjointness and completeness. Then a transformation rule R is a function producing a new formal model M, defined according to the syntax of Event-B models for a given input model:*

$$R : T \times C \rightarrow M$$

where C contains a set of possible configurations (i.e., with/without refinement) of the transformation rule R.

Note that R is defined as a total function, i.e., it produces a new model for each input model t of tabular expressions and configurations c, i.e., when $(t, c) \in dom(R)$.

Figure 1 depicts a graphical layout of the refinement strategy. The refinement strategy for producing formal models from tabular expressions considers system requirements defined in tabular expressions to construct an abstract model and successive refinement models. A formal definition of the transformation rule using the refinement strategy for producing a formal model M from tabular expressions is defined below.

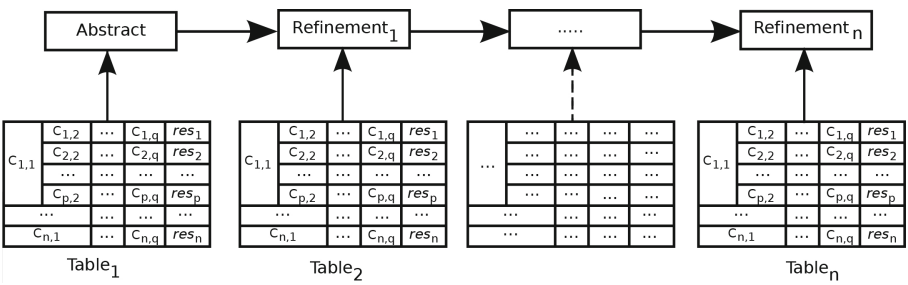

Fig. 1. Refinement strategy

Definition 2. *A refinement strategy is a transformation rule* $R : T \times C \rightarrow M$ *that constructs a model* M *for input tabular expressions and configuration (with/without refinement). The generated model* M *is defined as,*

$$
\begin{aligned}
&M = AM \sqsubseteq CM_1 \sqsubseteq CM_2 \sqsubseteq \cdots \sqsubseteq CM_n \\
&AM = t_1 \in T \wedge \Pi_{c_{1..n},res}(t_1) \\
&CM_1 = AM \cup (t_2 \in T \wedge \Pi_{c_{1..n},res}(t_2)) \\
&CM_2 = CM_1 \cup (t_3 \in T \wedge \Pi_{c_{1..n},res}(t_3)) \\
&\ldots \\
&CM_n = CM_{n-1} \cup (t_m \in T \wedge \Pi_{c_{1..n},res}(t_m))
\end{aligned}
$$

where AM *is an abstract model,* $CM_1, CM_2, \ldots, CM_n$ *are a series of concrete models,* Π *is a projection relation to select a table's column,* $t_1, t_2, t_3, \ldots, t_m$ *are a set of tables,* $c_{1..n}$ *is a set of columns of the table* T*,* res *is a set of output columns and* $\sqsubseteq$ *denotes a refinement relation. It is important to know that each table* t *of* T *contains a set of required variables, including type definitions, to describe system requirements in tabular form, which can be used during the process of model generation* $(AM, CM_1 \ldots CM_n)$*.*

The generated formal model is composed of an abstract model and a list of refinement models. An abstract model is important because it tells us exactly what the system is supposed to do without telling us how. In this refinement strategy, we can start to design an abstract model from any tabular expression, and then we can select other tabular expressions in a sequential order to introduce a new system behaviour by applying refinement laws, and preserving abstract behaviour. In Fig. 1, each tabular expression is introduced at a new refinement level that is defined in Definition 2 as $CM_1 = AM \cup (t_2 \in T \wedge \Pi_{c_{1..n},res}(t_2))$. The *skip* refinement allows us to introduce other events to maintain state variables. Importantly, a new refinement level allows to introduce a set of new events. In this refinement strategy, we do not use any guard strengthening and action simulation refinement laws. By using *skip* refinement, we introduce a new set of events corresponding to the tabular expressions. To design a formal model from tabular expressions, we traverse a tabular expression, in which condition columns are used for defining the guard predicates and output columns are used for defining actions of the events. Each row of a tabular expression creates an event. For example, we can select a row from Table 2 to create the first event, and the second event can be created from the second row of Table 2. In fact, in both rows the first row of the condition column is common and can be used as a common guard for both events. At each refinement level, we always select a new tabular expression to introduce new features and system requirements. It should be noted that the total number of refinements will depend on the total number of tabular expressions, and sometimes a few tables can be formalized together. Moreover, to satisfy the refinement relation between two consecutive models, to develop a consistent model, and to prove all the generated proof obligations related to refinement, we need to identify a dependency order between the tables that can be used further to generate the formal models.

4 Case Study: The Insulin Infusion Pump

To assess the proposed refinement strategy for producing formal models from tabular expressions, we select a medical device case study: the *Insulin Infusion Pump (IIP)*. An insulin pump is a complex and software-intensive medical device that delivers an appropriate amount of insulin to patients whenever required. An insulin pump is an integration of several hardware components: a physical pump, a disposable reservoir, and a disposable infusion set. The pump system is made of a controller and a battery. The disposable infusion set includes a cannula for subcutaneous insertion, and a tubing system to interface the insulin reservoir to the cannula. An insulin pump can be programmed to release small doses of insulin continuously (basal), or a one shot dose (bolus) before a meal, to control the rise in blood glucose.

As far as we know, there are no published system requirements for an IIP, but several research publications provide informal requirements [16,17]. We used such informal descriptions as a basis for this work to identify the system requirements by formulating *use cases* and *hazard analysis*. These system requirements focus on the functional behaviour of an IIP without addressing design requirements, and human computer interaction (HCI) requirements.

4.1 Generating Tabular Expressions

In this section, we describe how tabular expressions can be produced from the informal requirements. The IIP system requirements are described in several tabular expressions that are used to check the important properties of *completeness* and *disjointness*. Note that the tabular expressions for IIP are derived manually from the given informal requirements. In this development, we define 49 tabular expressions, that are further grouped into eight main functionalities: power status, user operations, basal profile management, temporary basal profile management, bolus management, bolus delivery, reminder management, and insulin output calculator. These main functionalities form a group of several small sub-functions, that are also defined using tabular expressions. For instance, Tables 1 and 2 describe *power status* and *power on self test (POST)*, in which condition columns contain required conditions and the results columns show associated outputs of the variables. In Table 1, we use natural language descriptions for describing the required conditions and in Table 2 the first condition column depends on the previous state of the variable *c_pwrStatus*. Similarly, in Table 2 the result column defines the value of control variable *c_pwrStatus*. We derive several tabular expressions from informal requirements to specify the system requirements so as to meet the properties of *disjointness* and *completeness*. Note that, in our tabular expression, we used a naming convention that uses prefixes to distinguish different types of variables to improve readability of the developed tabular expressions.

Table 1. Tabular expression for POST

Condition	*Result* POST
{ POST completed without problem }	Pass
{ POST completed and problems are detected }	Fail

Table 2. Tabular expression for power status

Condition		*Result* c_pwrStatus
c_pwrStatus$_{-1}$ = Standby	EXIST[M_pwrReq]	POST
	! EXIST[M_pwrReq]	NC
c_pwrStatus$_{-1}$ = POST	[POST] = Pass	Ready
	POST = Fail	Standby
c_pwrStatus$_{-1}$ = Ready	EXIST[M_pwrReq]	OffReq
	! EXIST[M_pwrReq]	NC
c_pwrStatus$_{-1}$ = OffReq	M_pwrResp = Accept	Standby
	M_pwrResp = Cancel	Ready

4.2 Formalizing the Insulin Infusion Pump

In the IIP case study, we use the refinement strategy to produce formal models from tabular expression requirements. In this development approach, we initially ignore most of the system complexity, including various functional behaviours. All the tabular expressions are progressively modelled using the refinement strategy, by providing required safety properties to make the operations safe. Note that sometimes there is no specific order required in which to apply the refinements. In this case, any order can be chosen after developing an abstract model. However, sometimes this is not true due to dependency between tables. In fact, we need to choose a specific order of tables during the system development. Each table of the system requirements is introduced in a new refinement level. In this article, we include all elementary steps for describing the model development and refinement steps of an IIP and the complete formal specification is available for inspection in the appendix of a report [18], which is more than 1500 pages long.

Abstract Model: Power Status. Our abstract model of the IIP specifies only power status and related functionalities that control the power status, i.e., turning the system *on/off*. The tabular expressions of *power status* and *power on self test (POST)* are defined in Tables 1 and 2, which are used for modelling an abstract model of IIP. In order to start the formalization process, we need to define static properties of the system. An Event-B context declares three enumerated sets *e_pwrStatus*, *e_basicResp*, and *e_postResult* defined using axioms (*axm*1–*axm*3) for power status.

$$
\begin{aligned}
&axm1 : partition(e\_pwrStatus, \{Standby\_pwrStatus\}, \{POST\_pwrStatus\}, \\
&\qquad \{Ready\_pwrStatus\}, \{OffReq\_pwrStatus\}) \\
&axm2 : partition(e\_basicResp, \{Accept\_basicResp\}, \{Cancel\_basicResp\}) \\
&axm3 : partition(e\_postResult, \{Pass\_postResult\}, \{Fail\_postResult\})
\end{aligned}
$$

An abstract model declares a list of variables defined by invariants (*inv1*–*inv5*). A variable *POST_Res* is used to state the result of *power-on-self-test (POST)*, where the result 'pass' (*Pass_postResult*) means system is safe to turn *on*, and the result 'fail' (*Fail_postResult*) means system is unsafe to start. The next variable *post_completed* is used to show successful completion of POST of an IIP. The variable *c_pwrStatus* shows the current power status of the system.

The variable *M_pwrReq* is used to model a request for power *on/off* from the user, and the last variable *M_pwrResp* is used for modelling user responses to system prompts.

$inv1 : POST\_Res \in e\_postResult$
$inv2 : post\_completed \in BOOL$
$inv3 : c\_pwrStatus \in e\_pwrStatus$
$inv4 : M\_pwrReq\_A \in BOOL$
$inv5 : M\_pwrResp \in e\_basicResp$

We introduce 10 events (derived from Tables 1 and 2) for specifying a desired functional behaviour for controlling the power status of the IIP. These events include guards for enabling the given actions, and the actions that define the changes to the states of power status (*c_pwrStatus*) and power-on-self-test (*POST_Res*). Here, we provide only two events related to the power status and power-on-self-test in order to demonstrate the basic formalization process. An event *POST_Completed* is used to assign the pass result (*Pass_postResult*) to *POST_Res*, when *post_completed* is *TRUE*. This event is generated from Table 1. The light grey colour of the condition and result columns of Table 1 shows the conditions and actions that are translated equivalently to event *POST_Completed*.

EVENT POST_Completed
WHEN
grd1 : $post\_completed = TRUE$
THEN
act1 : $POST\_Res := Pass\_postResult$
END

EVENT PowerStatus1
WHEN
grd1 : $c\_pwrStatus = Standby\_pwrStatus$
grd2 : $\exists x \cdot x \in BOOL \wedge x = M\_pwrReq$
THEN
act1 : $c\_pwrStatus := POST\_pwrStatus$
END

Similarly, another event *PowerStatus1* is used to set *POST_pwrStatus* to *c_pwr-Status*, when power status is *standby*, and there exists a power request from the user. The light grey colour of the condition and result columns of Table 2 presents the conditions and actions that are translated equivalently to event *PowerStatus1*. The remaining events are formalized in a similar way and are translated from the rows of Tables 1 and 2.

Since we do not have space for the detailed formalization, we summarise each refinement step of the IIP development in the following section.

4.3 A Chain of Refinements

For developing the whole system applying our refinement strategy, we used 7 main progressive development steps, which are defined as follows:

First Refinement: User Operations. This refinement introduces a set of operations, such as create, remove, activate and manage the basal profile, bolus profile, and reminders, performed by the user to program the IIP for delivering insulin. In this development, we cover all user interactions with the system, including user initiated commands and system responses. The formalised operations enable the delivery of a controlled amount of insulin according to the physiological needs of a patient.

Second Refinement: Basal Profile Management. This refinement introduces basal profile management to maintain a record and to store basal profiles defined by the user. In particular, we focus on the following operations: create a

basal profile; remove a basal profile; check the validity of a selected basal profile; activate a basal profile; and deactivate a basal profile. Note that whenever a new basal profile activates, then the old basal profile deactivates automatically.

Third Refinement: Temporary Basal Profile Management. This refinement introduces temporary basal profile management that is similar to the basal profile management, which allows for activating, deactivating and checking the validity of a selected temporary basal profile.

Fourth Refinement: Bolus Preset Management. This refinement introduces bolus preset management, which includes creating and checking the validity of a new bolus preset, removal operation of an existing bolus preset, and activation of the selected bolus preset.

Fifth Refinement: Bolus Delivery. In this refinement, we introduce a bolus delivery mechanism that allows us to start bolus delivery, to calculate the required dose for insulin delivery, and to check the validity of the calculated bolus and manually entered bolus. Moreover, this refinement also ensures that the IIP always delivers a correct amount of bolus at the scheduled time.

Sixth Refinement: Reminder Management. In this refinement, we introduce reminder management that allows us to create and validate a new reminder, and to remove an existing reminder. This refinement covers all the necessary elements for describing the reminder management, and to verify the requirements of reminder management.

Seventh Refinement: Insulin Output Calculator. The last refinement models the insulin output calculator. It calculates the insulin required over the course of the day, the appropriate time segment, and the time steps for delivering the insulin. It also keeps track of the insulin delivered within the time segment. The infusion flow rate can be 0, if the system is *off*, and there is no active profile or the maximum amount of insulin has been delivered already.

4.4 Safety Properties

In our IIP case study, we introduce several safety properties (i.e., see $spr1$–$spr9$) to make sure that the formalized IIP system is consistent and safe. The first safety property ($spr1$) ensures that when *EnteredBasProfValid* is *TRUE*, an entered basal delivery rate is within the safe range. Similarly, when *EnteredBasProfValid* is *TRUE*, $spr2$ ensures that the total amount of insulin delivered over a day is within the stated limit. $spr3$ and $spr4$ perform the same checks for the selected basal rate and amount when *SelectedBasalProfileIsValid* is *TRUE*. $spr5$ and $spr6$ perform the same checks for the temporary basal profile when *EnteredTemporaryBasalIsValid* is *TRUE*. $spr7$ states that when *SelectedPresetIsValid* is *TRUE*, the bolus rate of a selected bolus profile must be within the range of minimum bolus bound and maximum bolus bound. $spr8$ ensures that when *EnteredBolusIsValid* is *TRUE*, the bolus rate of an entered bolus profile must be within the range of minimum bolus bound and maximum bolus

bound. The last safety property ($spr9$) states that the total amount of insulin to output over the next time unit is less than or equal to the maximum daily limit of insulin that can be delivered.

$$
\begin{aligned}
&spr1: EnteredBasProfValid = TRUE \Rightarrow (\exists x, y \cdot x \mapsto y = M\_basProf \wedge \\
&\quad (\forall i \cdot i \in index\_range \wedge i \in dom(y) \Rightarrow y(i) \geq k\_minBasalBound \\
&\quad \wedge y(i) \leq k\_maxBasalBound)) \\
&spr2: EnteredBasProfValid = TRUE \Rightarrow (\exists x, y, insulin\_amount \cdot x \mapsto y = M\_basProf \wedge \\
&\quad insulin\_amount \in y\_insulinValue \wedge (\forall i \cdot i \in index\_range \wedge i \in dom(y) \Rightarrow \\
&\quad insulin\_amount = insulin\_amount + y(i) * k\_segDayDur) \wedge \\
&\quad insulin\_amount \leq k\_maxDailyInsulin) \\
&spr3: SelectedBasalProfileIsValid = TRUE \Rightarrow (\exists x, y \cdot x \mapsto y = M\_basActSelected \wedge \\
&\quad (\forall i \cdot i \in index\_range \wedge i \in dom(y) \Rightarrow y(i) \geq k\_minBasalBound \\
&\quad \wedge y(i) \leq k\_maxBasalBound)) \\
&spr4: SelectedBasalProfileIsValid = TRUE \Rightarrow \\
&\quad (\exists x, y, insulin\_amount \cdot x \mapsto y = M\_basProf \wedge insulin\_amount \in y\_insulinValue \wedge \\
&\quad (\forall i \cdot i \in index\_range \wedge i \in dom(y) \Rightarrow insulin\_amount = insulin\_amount + \\
&\quad y(i) * k\_segDayDur) \wedge insulin\_amount \leq k\_maxDailyInsulin) \\
&spr5: EnteredTemporaryBasalIsValid = TRUE \Rightarrow \exists x, y, z \cdot x \mapsto y \mapsto z = M\_tmpBas \wedge \\
&\quad y \geq k\_minBasalBound \wedge y \leq k\_maxBasalBound) \\
&spr6: EnteredTemporaryBasalIsValid = TRUE \Rightarrow \\
&\quad (\exists x, y, z \cdot x \mapsto y \mapsto z = M\_tmpBas \wedge y * z \leq k\_maxDailyInsulin) \\
&spr7: SelectedPresetIsValid = TRUE \Rightarrow (\exists x, y \cdot x \mapsto y = M\_bolSelected \wedge \\
&\quad y \geq k\_minBolusBound \wedge y \leq k\_maxBolusBound) \\
&spr8: EnteredBolusIsValid = TRUE \Rightarrow (\exists x, y \cdot x \mapsto y = M\_bolus \wedge \\
&\quad y \geq k\_minBolusBound \wedge y \leq k\_maxBolusBound) \\
&spr9: c\_insulinOut \leq k\_maxDailyInsulin
\end{aligned}
$$

4.5 Model Analysis

In this section, we present the proof statistics by presenting detailed information about generated proof obligations. Event-B supports *consistency checking* which shows that a list of events preserves the given invariants, and *refinement checking* which makes sure that a concrete machine is a valid refinement of an abstract machine. This complete formal specification of an IIP contains 263 events, 16 complex data types, 15 enumerated types, and 25 constants for specifying the system requirements. The system requirements are described using 49 tabular expressions. The formal development of the IIP is presented through one abstract model and a series of seven refinement models. In fact, the refinement models are decomposed into several sub refinements. Therefore, we have a total of 43 refinement levels for describing the system behaviour. In this paper, we have omitted the detailed description of the 43 refinements by grouping them into the main components we used to present the formal specification of the IIP by applying the second refinement strategy to the group of tabular expressions.

Table 3 shows the proof statistics of the development in the Rodin tool. To guarantee the correctness of the system behaviour, we provide a list of safety properties in the last refinement model. This development resulted in 444 (100%) proof obligations, of which 342 (77%) were proved automatically, and the remaining 102 (23%) were proved interactively using the Rodin prover (see Table 3).

Table 3. Proof statistics

Model	Total number of POs	Automatic proof	Interactive proof
Abstract model	3	3 (100%)	0 (0%)
First refinement	22	22 (100%)	0 (0%)
Second refinement	98	82 (83%)	16 (17%)
Third refinement	26	25 (100%)	1 (0%)
Fourth refinement	52	45 (87%)	7 (13%)
Fifth refinement	54	54 (100%)	0 (0%)
Sixth refinement	66	60 (91%)	6 (9%)
Seventh refinement	123	51 (42%)	72 (58%)
Total	444	342 (77%)	102 (23%)

These interactive proof obligations are mainly related to automated refinement based model generation and complex mathematical expressions, simplified through interaction to provide additional information for assisting the Rodin prover. Other proofs needed only to simplify predicates.

5 Related Work

Since the late 1950s, tables have been used for analyzing computer code, and documenting requirements. Tables first appeared in the software literature in the 1960s [19]. Early tables included decision tables, transition tables, etc. Parnas and others introduced tabular expressions for developing the requirements document for the A-7E aircraft [20,21] in work for the US Navy. Parnas was the most influential person to apply tabular expressions in documenting software [1]. Later, tables were used by many others, including at Bell Laboratories, and the US Air Force. Starting in the late 1980s tabular notations were applied by Ontario Hydro in developing the shutdown systems for the Darlington Nuclear Plant [22]. Formal semantics of tabular expressions have been proposed by Parnas [1] and other researchers [8]. A slightly outdated survey on tabular expressions is available in [8]. Nalepa et al., have proposed eXtended Tabular Trees (XTT) [23] and HeKatE [24] for developing a complex rule-based system, where these approaches are used to ensure high density and transparency of visual knowledge.

Refinement enables the incremental development of a system to ensure that a refined model retains all the essential properties of an abstract model. The foundational work of formal reasoning about correctness and stepwise development using refinement was established by Dijkstra [25] and Hoare [26] and further developed by Back and von Wright [27], and Morgan [28]. The refinement calculus provides a formal stepwise approach for constructing a program from an abstract program to a concrete program by preserving essential properties. There are a few papers published on automating the refinement pattern [4] and principles for refinement [5]. In [4], the authors propose refinement patterns using syntactic model transformation, pattern applicability conditions and proof obligations for verifying correctness preservation. To handle the design complexity of applying Event-B refinement and consistency rules, one paper [5], presents refinement planning from an informal/semi-formal specification.

6 Conclusion

We have presented a refinement strategy that can automate the process of formalizing system requirements from tabular expressions using a *correct-by-*

construction approach. We used a refinement strategy to transform tabular expressions into formal models that determine the correctness of functional behaviour and modelling structure of a system. We also highlighted challenges for automation: primarily, composition of tabular expressions, use of sequential ordering of tables, and table traversing complexities. Due to the variety of layouts of tabular expressions, there are still open issues related to the automation of tables that ought to be supported, and hence we do not claim completeness at this stage. On the other hand, our results showed that the proposed refinement strategy can largely be automated to generate formal models from tabular expressions. Moreover, the proposed approach is scalable to handle large and complex systems, in which system requirements are presented in tabular form.

In order to apply a refinement strategy, we selected the Event-B modelling language, which allows incremental refinement based on a *correct-by-construction* approach, for generating formal models from tabular expressions. Further, the Rodin tools can be used to verify formally the produced model. To assess the effectiveness of our proposed refinement strategy, we used the *Insulin Infusion Pump (IIP)* as a case study. The IIP requirements are described in tabular expressions, which we used to produce formal models using incremental refinement steps. In order to guarantee the 'correctness' of the system behaviour, we provided a list of safety properties in the generated model. Each refined model was proven to guarantee the preservation of those safety properties. This method of model generation and verification from the defined tabular expression requirements facilitates systematic modelling of a formal model using incremental refinement to guarantee formal designing of system requirements including required properties of completeness, disjointness, and safety. Our complete formal development of this IIP is available in a 1500 page report [18].

Our future goal is to develop a tool based on the proposed refinement strategy to automate the process for generating formal models from tabular expressions, and to apply this approach on several large and complex case studies to automate formal reasoning for tabular system requirements to verify a desired behaviour under relevant safety properties. This automation will allow us to produce formal models automatically from tabular requirements. In fact, if the original requirements are modified later, then we can use the automation tool to produce the new modified formal models. In addition, our intension is to use the generated and proved Event-B models for producing source code in many languages using EB2ALL [2,29].

References

1. Parnas, D.L.: Tabular representation of relations. Technical report, McMaster University (1992)
2. Singh, N.K.: Using Event-B for Critical Device Software Systems. Springer, New York (2013). doi:10.1007/978-1-4471-5260-6
3. Lee, I., Pappas, G.J., Cleaveland, R., Hatcliff, J., Krogh, B.H., Lee, P., Rubin, H., Sha, L.: High-confidence medical device software and systems. Computer **39**(4), 33–38 (2006)

4. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A.: Patterns for refinement automation. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 70–88. Springer, Heidelberg (2010). doi:10.1007/978-3-642-17071-3_4
5. Kobayashi, T., Ishikawa, F., Honiden, S.: Understanding and planning Event-B refinement through primitive rationales. In: Ait Ameur, Y., Schewe, K.D. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z. LNCS, vol. 8477, pp. 277–283. Springer, Heidelberg (2014). doi:10.1007/978-3-662-43652-3_24
6. Project RODIN: Rigorous open development environment for complex systems (2004). http://rodin-b-sharp.sourceforge.net/
7. Parnas, D.L., Madey, J., Iglewski, M.: Precise documentation of well-structured programs. IEEE Trans. Softw. Eng. **20**(12), 948–976 (1994)
8. Janicki, R., Wassyng, A.: Tabular expressions and their relational semantics. Fundam. Inform. **67**(4), 343–370 (2005)
9. Jin, Y., Parnas, D.L.: Defining the meaning of tabular mathematical expressions. Sci. Comput. Program. **75**(11), 980–1000 (2010). (Special Section on the Programming Languages Track at the 23rd ACM Symposium on Applied Computing)
10. Archinoff, G., Hohendorf, R., Wassyng, A., Quigley, B., Borsch, M.: Verification of the shutdown system software at the Darlington nuclear generating station. In: International Conference on Control and Instrumentation in Nuclear Installations, Glasgow, UK (1990)
11. Wassyng, A., Lawford, M.: Lessons learned from a successful implementation of formal methods in an industrial project. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 133–153. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45236-2_9
12. Heitmeyer, C., Kirby, J., Labaw, B., Bharadwaj, R.: SCR: a toolset for specifying and analyzing software requirements. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 526–531. Springer, Heidelberg (1998). doi:10.1007/BFb0028775
13. Abrial, J.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
14. Abrial, J.: The B-book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
15. Back, R., Sere, K.: Superposition refinement of reactive systems. Formal Aspects Comput. **8**(3), 324–346 (1996)
16. Masci, P., Ayoub, A., Curzon, P., Lee, I., Sokolsky, O., Thimbleby, H.: Model-based development of the generic PCA infusion pump user interface prototype in PVS. In: Bitsch, F., Guiochet, J., Kaâniche, M. (eds.) SAFECOMP 2013. LNCS, vol. 8153, pp. 228–240. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40793-2_21
17. Xu, H., Maibaum, T.: An Event-B approach to timing issues applied to the generic insulin infusion pump. In: Liu, Z., Wassyng, A. (eds.) FHIES 2011. LNCS, vol. 7151, pp. 160–176. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32355-3_10
18. Singh, N.K., Wang, H., Lawford, M., Maibaum, T.S.E., Wassyng, A.: Report 18: formalizing insulin pump using Event-B. Technical report 18, McSCert, McMaster University, October 2014. https://www.mcscert.ca/index.php/documents/mcscert-reports
19. Cantrell, H.N., King, J., King, F.E.H.: Logic-structure tables. Commun. ACM **4**(6), 272–275 (1961)
20. Heninger, K., Kallander, J., Parnas, D.L., Shore, J.E.: Software requirements for the A-7E aircraft. NRL Memorandum report 3876. Naval Research Laboratory (1978)

21. Parnas, D.L.: A generalized control structure and its formal definition. Commun. ACM **26**(8), 572–581 (1983)
22. Wassyng, A., Lawford, M., Maibaum, T.S.E.: Software certification experience in the Canadian nuclear industry: lessons for the future. In: EMSOFT, pp. 219–226 (2011)
23. Nalepa, G.J., Ligęza, A., Kaczor, K.: Formalization and modeling of rules using the XTT2 method. Int. J. Artif. Intell. Tools **20**(06), 1107–1125 (2011)
24. Nalepa, G.J., Ligęza, A.: The HeKatE methodology. Hybrid engineering of intelligent systems. Int. J. Appl. Math. Comput. Sci. **20**(1), 35–53 (2010)
25. Dijkstra, E.W.: A Discipline of Programming, 1st edn. Prentice Hall PTR, Upper Saddle River (1997)
26. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
27. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction, 1st edn. Springer-Verlag New York, Inc., New York (1998). doi:10.1007/978-1-4612-1674-2
28. Morgan, C.: Programming from Specifications. Prentice-Hall Inc., Upper Saddle River (1990)
29. Méry, D., Singh, N.K.: Automatic code generation from Event-B models. In: Proceedings of Second Symposium on Information and Communication Technology, pp. 179–188. ACM (2011)