A Tabular Expression to Event-B Language Transformation Tool

By
Yanjun Jiang, M.Eng.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Applied Science
Department of Computing and Software
McMaster University

MASTER OF APPLIED SCIENCE(2011) McMaster University
(Software Engineering) Hamilton, Ontario

TITLE: A Tabular Expression to Event-B Language Transformation Tool

AUTHOR: Yanjun Jiang, M.Eng.(Tianjin University)

SUPERVISOR: Dr. Tom Maibaum

NUMBER OF PAGES: vii, 113

Abstract

Tabular expressions have been proved in practice to be an excellent way to check the completeness (all possibilities are considered) and disjointness (no overlap among conditions) of software requirements described as functions. However, just these two properties alone are not sufficient for justifying the safety and efficacy of safety critical systems. Formal specification languages like Z, CSP and Event-B can also support mathematical reasoning required for these kinds of systems and have more effective tool support available. This thesis reports on a new transformation tool TX2EB (Tabular Expressions to Event-B) which combines the completeness and disjointness checks and mathematical reasoning needed for verification and classification of the documented requirements. Moreover, the proposed tool can generate formal specifications from the tabular expressions automatically and support subsequent refinement and analysis of the specifications. Therefore, it can liberate researchers from tedious and time-consuming manual application of the transformation.

Dedicated to my family

Acknowledgements

First of all, I want to thanks my supervisor Dr. Tom Maibaum for providing me such a good opportunity to do research in the McMaster Centre for Software Certification. His profound knowledge, rich experience in the software certification field and his approachable personality have greatly influenced me. As a Canada Research Chair in the Foundations of Software Engineering and Professor of the Computing and Software Department, Tom has tons of research and teaching activities to deal with every day. But even though he is so busy, he still manages to meet me every one or two weeks to discuss the progress and difficulties in my research. Without his supervision and help, this thesis could not be finished. I offer my sincerest thanks to him.

Secondly, I am extremely grateful to postdoc Neeraj Kumar Singh. As an expert in Event-B formal specification, he put forward the idea of automating the transformation process from Tabular Expressions to Event-B specifications. And whenever I had some technical problems or difficulties, he was always there to help me out. He also spent lots of time on proofreading my draft paper and thesis, as well as seminar slides. He also gave me a lot of help in my daily life. It is my honor to work together with such a great and kind man.

I also want to express thankfulness to the Insulin Infusion Pump group members : Alan Wassyng, Mark Lawford, Neeraj Kumar Singh, Hao Wang, Mischa Geven, Nicholas Proscia for their instructive and beneficial discussion at the regular meeting every Wednesday afternoon.

Finally, thanks all the committee members: Emil Sekerinski, Tom Maibaum and Mark Lawford for reading my thesis, joining my oral defense and giving me all the valuable comments and suggestions, which greatly improves the quality of the thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Research on rigorous software development has been carried out for many years, especially in the area of safety-critical software systems [16, 4], such as in the medical, nuclear and automotive domains. As one of the outstanding developments in this field, tabular expressions [36] present the software requirements as functions in the form of clear, readable and hierarchical, but still formal, tables; they have been used successfully in a number of safety-critical system projects. For example, [17] reported the usage of tabular methods for the specification and verification of a nuclear reactor shutdown system at Ontario Power Generation Inc. (OPG). More recently, tabular expression has been suggested for requirements specification in the FAA because it is easier to confirm that the requirements are complete and consistent [18]. However, the lack of tool support has greatly constrained its widespread usage. Fortunately, Eles et al. [8] developed a tool for parsing the tabular expressions in the Matlab Simulink environment. Using this tool, a user can check the completeness and disjointness of the requirements represented in tables with basic knowledge of the formal specification. This tool has the potential to help the software engineers and architects in formally documenting the software requirements. Therefore, it has a promising potential both in academia and industry.

The traditional testing method tries to verify the system after it has been built, which is called "laboratory execution" in [2]. Quite different from that, the Event-B model "blue-print" approach [2] can make available some formal mathematical reasoning to prove correctness and consistency of a system before it is built. Therefore, it can provide confidence to the developer that the system is correct. This kind of before-implementation verification is of crucial importance for some systems where "laboratory execution" is impossible or failure is unaffordable, like a nuclear plant.

On the other hand, manual transformation of requirements from tabular expressions of requirements to Event-B models suffers from several drawbacks. For example, we manually transformed Insulin Infusion Pump (IIP) requirements stated as tabular expressions to Event-B models to validate and verify their correctness and safety. The process was not only tedious, laborious and long lasting, but also it was error-prone. More importantly, it is possible to automate the manual process if proper table traversing and refining rules and generating algorithms are given.

To sum up, because of the merits of both tabular expressions and Event-B models and demerits of the human transformation process, an effective and efficient automatic tool is highly desirable to improve the quality and efficiency of the rigorous software development process.

1.2 Related Work

1.2.1 The TET Tool

As far as we know, our tool is the first one that combines both the tabular expression language and the Event-B modelling language. Nevertheless, there was some preliminary work done by Eles et al. [8]. In the TET tool, a Matlab GUI for building tabular expressions was created to facilitate user's development of requirements. Thus, the user can directly edit the system requirements in the tables. After finishing the tabular requirements, a PVS/CVC3 model

can be generated using the Matlab script or the jTET Java library [5]. Then a PVS/CVC3 prover can be leveraged to prove the required properties of completeness and disjointness of tabular requirements. Once these properties are proved, the user will be notified by a *Typechecked* status message. Otherwise, an error or a *counterexample* will be generated.

1.2.2 The jTET Tool

Matthew Dawson developed the jTET Java library [5], which can be use by tables from different kinds of tools (including TET, but not limited to TET). Basically, jTET takes some previous tabular expression processing code written in Matlab and rewrites it in Java. Or put another way, jTET separates the data logic processing part (a.k.a. data model in the MVC pattern) from the GUI so that it can be reused by not only the TET tables but some other table forms. Our TX2EB tool reuses the expression parsing and table traversing part of jTET.

1.2.3 The UML-B Tool

UML-B [31] is a Rodin Plug-in that can generate Event-B models from UML models. It is a fully integrated graphical frontend for Event-B. Various diagrammatic modelling notations and editors are available in UML-B to create models which are then translated into Event-B for verification. Specifically, the users have to create a UML diagram first. Once the drawing is finished and saved, the tool (the translator part) will automatically convert the diagram into its corresponding Event-B model. Then the Event-B verification tools (syntax checker and prover) will be called automatically to provide instant feedback about problems, which are then displayed on the relevant UML-B drawing. This tool has been applied on a failure management system (FMS) [32] and it emerged that UML-B is very suitable for this kind of problem.

Our TX2EB tool and UML-B tool have the common objective to produce

Event-B models. Our tool does not have graphical input like UML-B. But instead it uses some kind of tabular expression form of input and combines two types of verification together (the disjointness and completeness checks and the Event-B mathematical proof check), which proves that the specified system is correct from different perspectives.

1.2.4 The EB2ALL Tool

Singh et al. [7, 28] describes a code generator named EB2ALL. It is a set of translator tools that automatically generates multiple efficient target programming language source codes (C, C++, Java, and C#) from Event-B formal specifications. The EB2ALL contains four plugins: EB2C, EB2C++, EB2J, and EBC#. The goal of the tool is to generate verified source code that satisfies behavioral properties of the developed formal system. It is developed as a set of plugins for the Rodin Platform under the Eclipse framework.

In fact, the TX2EB tool can be combined with this EB2ALL code generation tool to generate the desired source code from tabular expression requirements. Specifically, this process can be illustrated in the following Fig. 1.1.

First, the users can create the tabulated requirements from the informal requirements manually and check the disjointness and completeness of the tabular expressions. And then an Event-B model can be generated from the type checked tabular expressions. The generated Event-B model is then verified in the Rodin platform. If the users want to generate source code, the EB2ALL tool is employed to generate C/C++/C#/Java code from the Even-B model.

1.3 Contributions

The transformation work in this thesis is an important step to automate the software development life cycle, particularly from requirement analysis to code generation. A software engineer only needs to develop the system requirements using tabular expressions and then the tabulated system requirements can be transformed into Event-B models automatically using the TX2EB tool.

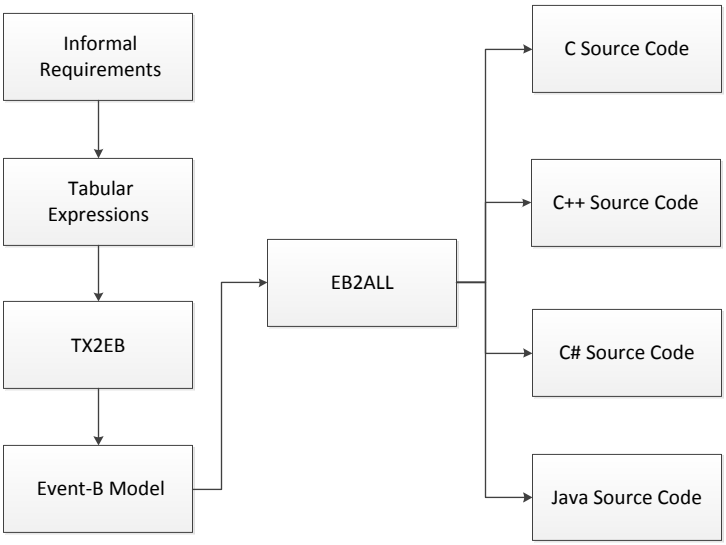


Figure 1.1: Tabular Expressions to Source Code

Particularly, the completeness and disjointness properties of the system can be checked first using the TET tool. And then more mathematical reasoning about the generated Event-B models can be done in the Rodin Platform to verify the correctness and consistency of the whole system. Furthermore, if the user wants to produce the source code, the tool EB2ALL [28] is available to generate C/C++/C#/Java source codes from Event-B models. Finally, we also implement 12 Function Blocks of the IEC-61131Standard as case study.

1.4 Overview

The remaining part of the thesis is organized as follows. Chapter 2 provides background information about tabular expressions and the Event-B language. Chapter 3 describes requirements, design, translation principles and rules, implementation, and limitations of the tool. The case study in Chapter 4 shows the applicability and usefulness of the tool. Finally, Chapter 5 concludes the thesis.

Chapter 2

Background

2.1 Tabular Expressions

2.1.1 History

Tabular expressions (a.k.a. function tables) were introduced to the world by Parnas and others at the U.S. Naval Research Laboratories in the late 1970s to document requirements [36]. They have been around for more than three decades. As the most important contributor, Parnas has published lots of papers about the use of tabular expressions in documenting software systems [23, 25, 27, 26, 12]. He also described 10 classes of tabular expressions and gave their syntax and semantics [24, 14], and has encouraged the development of software tools for tabular expressions [10].

2.1.2 Introduction

There are many varieties of tabular expressions (tables), such as Normal Function Tables, Inverted Function Tables, Vector Function Tables, Normal Relation Tables, Inverted Relation Tables, Vector Relation Tables, Mixed Vector Tables, Predicate Expression Tables etc. [24]. In our work, we mainly concentrate on the most common Horizontal Condition Tables (HCTs).

HCTs, also known as Program Function Tables (PFTs), were invented sim-

ply because it seemed that system engineers preferred to read tables from left to right. They are very suitable for documenting the requirements behavior. In the HCTs, the relation between adjoining cells is "conjunction" (logic and) [36]. The typical structure of a HCT table is illustrated in the following Fig. 2.1.

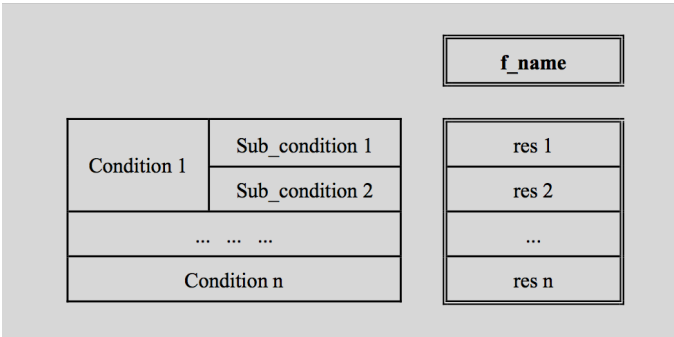


Figure 2.1: Horizontal Condition Table

The Semantics of HCT [22] can be depicted as shown in the following Fig. 2.2.

Condition		*Result* **f**
C_1	$C_{1.1}$	res_1
	$C_{1.2}$	res_2
	...	...
	$C_{1.m}$	res_m
...		...
C_n		res_n

IF C_1
  IF $C_{1.1}$ **THEN** f $= res_1$
  ELSEIF $C_{1.2}$ **THEN** f $= res_2$
  ...
  ELSEIF $C_{1.m}$ **THEN** f $= res_m$
ELSEIF ...
ELSEIF C_n **THEN** f $= res_n$

Figure 2.2: Semantics of HCT

Tabular expressions provide a useful way to document and analyze software systems. They are clearly readable but yet still formal, which can help software engineers to check the completeness and disjointness (definitions are given below) of the requirements in tables in a convenient way [14, 8, 22]. For example, Fig. 2.3 compares the formal logical specification of a function (the above one) with its semantically equivalent tabular expression (the bottom

one). The function has two input variables m and n and one output variable g. The value of g is determined by the input values of variables m and n. That is g is a binary function of m and n. It is obvious that the tabular expression is more readable and expressive than the logical description of the function.

$$g(m,n) = \begin{cases} m+n & \text{if } m \geq 2 \wedge n > 1 \quad (2.1) \\ 3m+2n & \text{if } m < 2 \wedge n > 1 \quad (2.2) \\ m/n & \text{if } m \geq 2 \wedge n = 1 \quad (2.3) \\ m-2n & \text{if } m < 2 \wedge n = 1 \quad (2.4) \\ m-n & \text{if } m \geq 2 \wedge n < 1 \quad (2.5) \\ mn & \text{if } m < 2 \wedge n < 1 \quad (2.6) \end{cases}$$

Condition		*Result* g
m ≥ 2	n > 1	m + n
	n = 1	m / n
	n < 1	m - n
m < 2	n > 1	3m + 2n
	n = 1	m - 2n
	n < 1	mn

Figure 2.3: The logical description of a function and its tabular expression

The formal definitions of completeness and disjointness are given as follows [17, 8].

A m-ary function $f : T_1, T_2, ...T_m$ to T_r may have a tabular expression representation:

$$f(x_1, x_2, \ldots x_m) = \begin{array}{|c|c|c|c|} \hline c_1 & c_2 & \ldots & c_n \\ \hline e_1 & e_2 & \ldots & e_n \\ \hline \end{array} \text{ or } \begin{array}{|c|c|} \hline c_1 & e_1 \\ \hline c_2 & e_2 \\ \hline \ldots & \ldots \\ \hline c_n & e_n \\ \hline \end{array}$$

In the above formula, every c_i corresponds to an e_i. Each c_i is a boolean expression and e_i is a term of type T_r. It means that when the boolean expression c_i is true, the function f returns the term e_i. The table has to satisfy the following two properties to define a total function properly.

Disjointness: requires that the conditions in columns (rows) do not overlap. i.e., $\forall i, \forall j (i \neq j \Rightarrow \neg(c_i \wedge c_j))$.

Completeness: requires that the conditions in columns (rows) cover all the input possibilities, i.e., there is no missing part. That is $(c_1 \vee c_2 \ldots \vee c_n) \equiv TRUE$.

Disjointness and completeness are important properties for safety-critical systems. They are necessary, but not always sufficient, conditions for the safety of the system whose requirements are being defined.

2.2 Event-B

2.2.1 Introduction

According to [13] and the Event-B wiki, Event-B is a formal method based on set theory and first order logic that is an extension of the B method. While the B method is focused on supporting formal development of software, Event-B is designed for analysis and modelling of systems that may consist of software, electronics and physical components. Key features of Event-B are the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify consistency between different refinement levels, as well as the correct use of set theory to describe models.

As an event-based formal specification and modelling language, Event-B is now widely used for formalizing system requirements. We only give the Event-B basic concepts and notations in this thesis. More detailed information can be found in [2].

The Event-B notations are described in detail by Abrial [2]. The introduction of Event-B notations in this paper are primarily based on his work. There are mainly two important components in an Event-B model, namely Context

and Machine. The Context is composed of carrier sets, constants, axioms, and theorems while the Machine is made up of variables, invariants, theorems, variants, and events. The components both in Contexts and Machines are all called modelling elements, and they correspond to the static and dynamic parts of a model, respectively.

There are three kinds of relationships between components of an Event-B model as depicted in Fig. 2.4 [2].

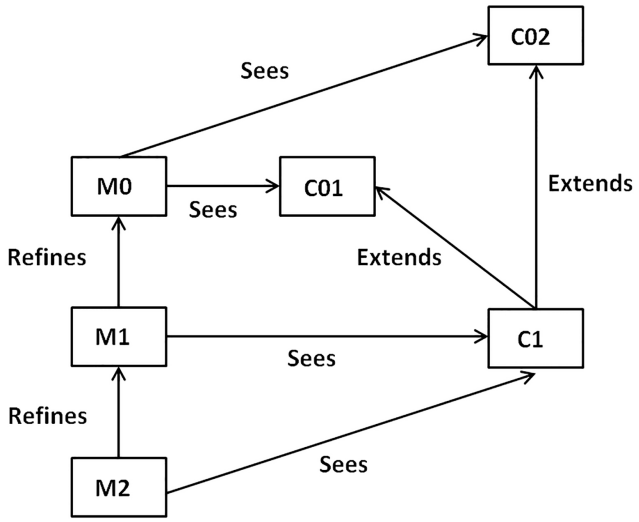


Figure 2.4: Contexts and Machines Relation

- A concrete Machine can only "refine" at most one more abstract Machine.
- A concrete Context can "extend" zero, one, or several more abstract Contexts.
- A Machine can "see" zero, one, or several Contexts.

If a Machine "sees" a Context, then all the components like constants, sets, and axioms defined in the Context and extended from other Contexts can be used by the Machine.

2.2.2 Event-B Context

The Context structure is illustrated in Fig. 2.5. The Context is made of predefined keywords and corresponding optional clauses. Each Context must have a unique name within the same model defined at the very beginning of its structure and it may extend zero, one, or several other existing Contexts which are listed after the keyword "extends". Clause "sets" defines a carrier set for the Context. Common carrier sets are the Natural Numbers set, the Integer Numbers set and user-defined Enumeration sets. A list of constants can be defined under the "constants" keyword. Some Contexts may have axioms and theorems. The axioms describe the various predicates that constants obey which act as hypotheses in the proof obligation while the theorems list all the theorems that need to be proved within the Context.

```
<Context name>
    extends
        <Contexts list>
    sets
        <sets list>
    constants
        <constants list>
    axioms
        <label>:<predicate>
        ...
    theorems
        <label>:<predicate>
        ...
    end
```

Figure 2.5: The Context Structure

2.2.3 Event-B Machine

Fig. 2.6 represents the structure of an EventB Machine. Similar to Context, the name is given at the beginning of the Machine structure, and it must be

different from all other component names in the same model. A Machine can "refine" zero, one, or several other Machines, and it may also "see" zero, one, or several other Contexts. A list of variables can be defined in the Machine "variables" keyword. Clause "invariants" introduces the various predicates which the variables must satisfy. All the theorems that must be proved in the Machine are listed in the "theorems" clause. So as to prove a theorem, the axioms and theorems in the seen Contexts and the invariants and theorems in the refined abstract Machines, as well as the local invariants, can all be treated as presumptions. If a Machine model needs to be proved convergent, the "variant" clause may be used. The "events" clause gives the various events within the Machine model, and it is the essential part of the EventB Machine.

```
<Machine name>
    refines
        <Machine name>
    sees
        <Contexts list>
    variables
        <variables list>
    invariants
        <label>:<predicate>
        ...
    theorems
        <label>:<predicate>
        ...
    variant
        <variant>
    events
        <events list>
    end
```

Figure 2.6: The Machine Structure

As the essential part of the EventB Machine, the event generally has the following structure described in Fig. 2.7. Generally, an event is made up of several clauses introduced by specific keywords. Of all the clauses in an event,

```
<event name>
    status
      {ordinary,convergent,anticipated}
    refines
      <events list>
    any
      <parameters list>
    where
      <guards list>
    with
      <witnesses list>
    then
      <actions list>
    end
```

Figure 2.7: The Event Structure

only the "status" clause is mandatory, all other clauses are optional.

- Clause "status" can be either **ordinary**, **convergent** or **anticipated**. By default, the event status is ordinary. Convergent means that the event has to decrease the variant. Anticipated ensures that the event cannot increase the variant.
- Clause "refines" defines the abstract event this event refines.
- Clause "any" describes the parameters of the event.
- Clause "where" lists all the guards of the event, which are the necessary conditions to enable the event. If the keyword Clause "any" is not provided in the event, then keyword **where** is substituted by keyword **when** in the pretty print of the Rodin Platform.
- Clause "with" lists the *witnesses* of the corresponding abstract event.
- Clause "then" contains the list of actions of the event. An action can be either *deterministic* or *non-deterministic*.

In the first case, the action is made up of a variable name, followed by :=, then followed by an expression. For example, **act1** : $x := x + z$.

In the second case, the non-deterministic action is made up of a list of variable names, followed by : |, then followed by a before-after predicate. The *before-after predicate* may contain all the variables of the machine, which denote the corresponding values *just before* the action takes place. It may also contain some variable identifiers which are primed; they correspond to the values *just after* the action has taken place. For instance, given three variables x, y and z, here is a non-deterministic action:

$$\textbf{act1}: \quad x, y \quad : | \quad x' > y \ \wedge \ y' > x' + z.$$

- Besides, each machine must contain a special **initialization** event.

2.2.4 Refinement

Refinement uses a top-down strategy to analyze and design a system. It first constructs a very abstract model of the whole system and then gradually adds more details and information to the next layer concrete model, which refines the abstract model [1]. This process is like looking through a microscope. The reality does not change. We are just looking at it from a closer perspective, and therefore we can see more accurately. Previously invisible details are now revealed by the microscope. The more powerful the microscope is (the more concrete the model is), the more accurately we can see. The reason we use refinement in system design and modelling is that it may be impossible for us to manage and understand the whole system represented in one large model, due the size of the state and the number of transitions. Put another way, refinement can help us to manage the complexity of large systems.

The above refinement is called *spatial extension*. Correlatively, there is a corresponding *temporal extension* [1]. The reason is that the new variables can be modified by some transitions, which could not have been present in the previous abstractions just because the variables concerned had not yet been introduced to them. In practice, *temporal extension* is implemented by means

of *new events* which involve the new variables only. From another perspective, this can be treated as if these new events refine some implicit events doing nothing on the abstractions.

There is a second usage of refinement which is called *data-refinement*. This refinement is used to modify the state so that it can be implemented on a computer by some programming language. It is used as a second technique, once all the important properties have been modelled.

2.2.5 Decomposition

The mastering of model complexity cannot be solved completely by using refinement only [1]. As a model is more and more refined, the number of its state variables and transitions may increase in such a way that it becomes impossible to manage the whole. At this point, it becomes necessary to split the single refined model into several loosely coupled pieces.

By definition, decomposition is precisely the process by which a single large complex model can be partitioned into various smaller simple component models (sub-models) in a systematic fashion. Using partition, we can manage the complexity of the whole by studying and refining each sub-model independently of the others. The definition implies that independent refinements of the sub-models can be composed again to form a single model that is guaranteed to be a refinement of the original one. The decomposition process can be further applied on the sub-models, and so forth. Because the decomposed model could already exist and developed, it is possible to combine a top-down approach with a bottom-up one in the decomposition process [1].

2.2.6 Proof Obligation Rules

The proof obligations (POs) define what needs to be proved by the Rodin Platform for a given Event-B model [2, 38]. They are automatically generated by a proof obligation generator in the Rodin Platform. Key tools for discharging the POs are the static checkers, proof obligation generator and the provers.

This section defines some of the most common PO rules. In order to define rules dealing with an event, we shall first give the general definition of an event [2].

Event $\triangleq$
  any
  x
  where
    $< G(s, c, v, x) >$
  then
    $< v : \mid BA(s, c, v, x, v') >$
  end

where x stands for the abstract parameters, s the seen sets, c the seen constants, v the variables of the Machine and v' the substituted variables. G denotes the event guards, $BA(s, c, v, x, v')$ denotes the before after predicate and $< v : \mid BA(s, c, v, x, v') >$ the event actions. Seen axioms and theorems are collectively denoted by $A(s, c)$, whereas invariants and local theorems are denoted by $I(s, c, v)$. In a refining Machine, the local invariants (where w denotes concrete variables) and theorems are denoted by $J(s, c, v, w)$.

Invariant preservation rule: INV

INV stands for the invariant of a Machine, which is always preserved by all the events. This PO is named "event name/invariant name/INV". The INV proof obligation rule is as follows.

$$
\begin{array}{l}
A(s,c) \\
I(s,c,v) \\
G(s,c,v,x) \\
BA(s,c,v,x,v') \\
\vdash \\
\mathrm{inv}(s,c,v')
\end{array}
$$

which means:

Axioms and theorems
Invariants and theorems
Guards of the event
Before-after predicate of the event
$\vdash$
Modified specific invariant

Feasibility rule: FIS

FIS ensures that a non-deterministic action is always feasible, which means there exists a value for the variable v that satisfies the before after predicate $BA(s,c,v,x,v')$. This PO is named "event name/invariant name/FIS". This proof obligation rule is as follows.

$$
\begin{array}{l}
A(s,c) \\
I(s,c,v) \\
G(s,c,v,x) \\
\vdash \\
\exists v' \ \cdot \ BA(s,c,v,x,v')
\end{array}
$$

which means:

Axioms and theorems
Invariants and theorems
Guards of the event
$\vdash$
$\exists v' \cdot$ Before-after predicate

Simulation rule: SIM

This proof obligation rule is designed for refinement. It makes sure that when a *concrete event* is executed, its behavior should be consistent with the corresponding *abstract event* behavior. That is, the *concrete event actions* simulate the *abstract event actions*. This PO is named "event name/action name/SIM". Given two events $event0$ and $event1$, suppose $event1$ refines $event0$. $event0$ has the abstract parameters x and the abstract before-after predicate $BA1(s, c, v, x, v')$. $event1$ has the concrete parameters y, the concrete event guards $H(y, s, c, w)$, the parameter witness predicates $W1(x, s, c, w, y, w')$, the variable witness predicates $W2(v', s, c, w, y, w')$ and the concrete before-after predicate $BA2(s, c, w, y, w')$. Then the SIM proof obligation rule is as follows:

$A(s, c)$
$I(s, c, v)$
$J(s, c, v, w)$
$H(y, s, c, w)$
$W1(x, s, c, w, y, w')$
$W2(v', s, c, w, y, w')$
$BA2(s, c, w, y, w')$
$\vdash$
$BA1(s, c, v, x, v')$

which means:

Axioms and theorems
Abstract invariants and theorems
Concrete invariants and theorems
Concrete event guards
Witness predicates for parameters
Witness predicates for variables
Concrete before-after predicate
Witness predicates for parameters
$\vdash$
Abstract before-after predicate

Guard strengthing rule: GRD

The GRD proof obligation rule ensures that the concrete guards in a refined event should be stronger than their corresponding abstract event guards, which means whenever a concrete event is enabled, its corresponding abstract event should also be enabled. This PO is named "event name/guard name/GRD". Given that $J(s, c, v, w)$ are the concrete invariants and theorems, $H(y, s, c, w)$ the concrete event guards, $W(x, s, c, w, y)$ the witness for parameters and $g(s, c, v, x)$ the abstract event specific guard, then the GRD proof obligation rule is the following:

$$
\begin{array}{l}
A(s,c) \\
I(s,c,v) \\
J(s,c,v,w) \\
H(y,s,c,w) \\
W(x,s,c,w,y) \\
\vdash \\
g(s,c,v,x)
\end{array}
$$

which means:

Axioms and theorems
Abstract invariants and theorems
Concrete invariants and theorems
Concrete event guards
witness predicates for parameters
witness predicates for parameters
$\vdash$
Abstract event specific guard

Numeric variant rule: NAT

This proof obligation rule ensures that under the guards of each convergent or anticipated event, a proposed numeric variant is indeed a natural number. This PO is named "event name/NAT". If $G(s,c,v,x)$ denotes the event guards and $n(s,c,v)$ is the numeric variant, then the NAT proof obligation rule is the following:

$$
\begin{array}{l}
A(s,c) \\
I(s,c,v) \\
G(s,c,v,x) \\
\vdash \\
n(s,c,v) \in \mathbb{N}
\end{array}
$$

which means:

Axioms and theorems
Invariants and theorems
Event guards
$\vdash$
A numeric variant is a natural number

Finite set variant rule: FIN

This proof obligation rule ensures that under the guards of each convergent or anticipated event, the proposed set variant is indeed a finite set. This PO is named "event name/FIN". If $G(s,c,v,x)$ denotes the event guards and $t(s,c,v)$ is the set variant, then the FIN proof obligation rule is the following:

$$
\begin{array}{l}
A(s,c) \\
I(s,c,v) \\
G(s,c,v,x) \\
\vdash \\
finite(t(s,c,v))
\end{array}
$$

which means:

Axioms and theorems
Invariants and theorems
Event guards
$\vdash$
Finiteness of set variant

The variant rule: VAR

This proof obligation rule makes sure that under the guards of each convergent or anticipated event, the proposed set variant or finite set variant is decreased. It is named "event name/VAR" If $G(s, c, v, x)$ denotes the event guards, $BA(s, c, v, x, v')$ the before-after predicate, $n(s, c, v)$ the variant and $n(s, c, v')$ the modified variant, then the VAR proof obligation rule is the following:

$A(s, c)$
$I(s, c, v)$
$G(s, c, v, x)$
$BA(s, c, v, x, v')$
$\vdash$
$n(s, c, v') < n(s, c, v)$

which means:

Axioms and theorems
Invariants and theorems
Event guards
Before-after predicate of the event
Event guards
$\vdash$
Modified variant smaller than variant

Other rules

There are some other proof obligation rules in Event-B in addition to the rules presented above. The well-definedness rule WD requires that a potentially ill-defined Event-B component is indeed well defined. The theorem rule THM makes sure that a theorem is provable. The non-deterministic witness rule WFIS ensures that a concrete event witness indeed exists. The guard merging rule MRG makes sure that a concrete event guard is indeed stronger than the disjunction of the merged abstract event guards.

2.2.7 Rodin Platform

The Rodin Platform was originally developed by the European Commission funded Rodin project (IST-511599), where Rodin is an acronym for "Rigorous Open Development Environment for Complex Systems" [13].

Basically, the Rodin Platform is an Eclipse-based IDE for formal modelling in Event-B. It provides effective support for refinement and mathematical proof. As open source software, the Rodin Platform supports extension points for a range of additional plugins which enrich the core functionality. ProB, UML-B and Camille are examples of the key plugins.

Rodin comes with the New PP prover installed; it can generate proof obligations that can be discharged either automatically or interactively. There are also some third-party provers available as plugins.

In our work, we used the Rodin Version 2.8. All the generated Event-B Models were eventually imported into this tool to verify and validate their correctness. It turned out that all the proof obligations of our case study were automatically discharged.

2.3 Matlab / Simulink

Matlab was first released in 1984 by the MathWorks company. It is a combination of two words: matrix and laboratory. As indicated by the name, Matlab was originally designed for matrix analysis and manipulation; but it evolved into a large platform for algorithm development, data visualization, data analysis, and numerical computing. There are also many tools for different applications, from engineering calculation to image processing and Financial Modelling. Matlab has integrated so many powerful functionalities into such a user-friendly graphical environment that it provides a comprehensive solution for scientific research, engineering design, and other fields that require effective numerical calculation. Since Matlab is so powerful, so easy to use, and has many toolboxes for various applications, it is widely used not only in academic and research institutions but also in industrial enterprises.

Built on top of the Matlab environment, Simulink has become one of the most important Matlab components. It provides an integration environment for dynamic system modelling, simulation and comprehensive analysis using a graphical block diagramming tool and a customizable set of block libraries. In this environment, a complex system can be constructed by simple mouse point and click actions without requiring a large amount of coding. Because of its advantages of clear structure and process, precise simulation, flexibility and high efficiency, Simulink is widely used in fields like control theory [35, 33, 3], digital signal processing [19, 20, 15] and finance [37, 6, 34].

In our tool, Matlab is used to provide a user GUI for editing tabular expressions requirements and checking the disjointness and completeness of tabular expressions. It is also responsible for calling the packaged Java code to gener-

ate the Even-B model.

Chapter 3

TX2EB: An Automatic Event-B Model Generator

3.1 Introduction

This chapter describes a tool which can generate Event-B models from tabular expressions. There is no tool that can automatically translate tabular expressions into Event-B specification currently. The tool developed by us can not only check the disjointness and completeness of tabular expressions but it can also generate the Event-B models directly from tabular expressions, which can be used to do formal reasoning and verification of the systems requirements in a Rodin Platform.

Our main objectives for developing this tool are as follows:

- Formal reasoning of tabular expressions.
- To integrate the completeness and disjointness checks with formal verification in one tool.
- To detect inconsistencies and errors in the documented system requirements.
- To liberate researchers from tedious and time-consuming manual transformation process.

3.1.1 Structure of This Chapter

This chapter is organized as follows. Section 3.2 presents the tool overview. Section 3.3 lists the requirements of the TX2EB tool. Section 3.4 gives the translation principles and rules of the tool. Section 3.5 talks about the test driven development (TDD) mode that we used. Section 3.6 depicts the tool architecture. Section 3.7 describes the four types of tool modes. Section 3.8 discusses the tool implementation and rationale. Section 3.9 lists the limitations of our tool. Finally, section 3.10 gives the user manual.

3.2 Tool Overview

Our TX2EB tool is developed on the basis of the Tabular Expression Tool (TET) [8], which only supports the most common and useful Horizontal Condition Tables (HCTs). In HCTs, only horizontal grids are allowed to have sub-grids. The structure and semantics of HCTs are provided in section 2.1.

Fig. 3.1 illustrates the transformation process of the tool in the form of a workflow. Our tool uses the Matlab GUI to create the tabular expression version requirements from existing informal requirements. We can check the completeness and disjointness properties of the tabular expression using a PVS or a CVC3 prover. Then we can generate an Event-B model in the form of a Rodin project that contains Machine, Context, and project files. Finally, the Event-B model can be imported into a Rodin Platform to do the formal verification. For example, some Proof Obligations(POs) can be automatically generated and discharged so as to verify the consistency and correctness of the system requirements.

3.3 TX2EB Requirements

A list of requirements of the TX2EB tool are given as follows:

1) The tool should generate a Rodin dependent project from tabular expressions.

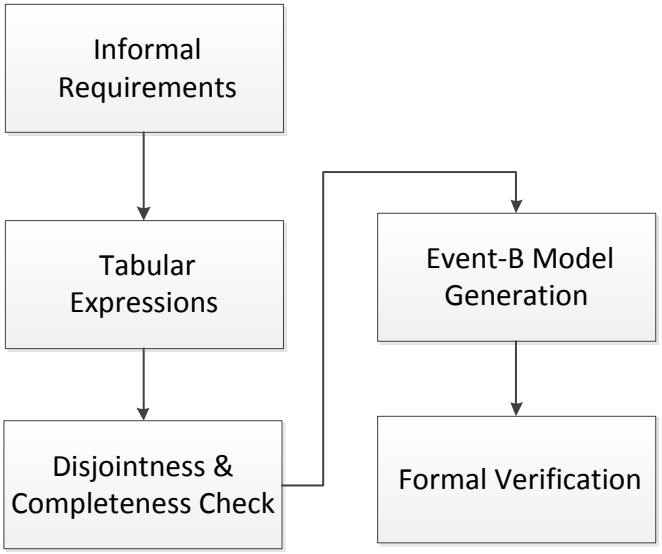


Figure 3.1: TX2EB Workflow

2) The tool should allow to input single table / multiple tables for model generation.

3) The tool should support: single output / multiple outputs.

4) The tool should produce an Event-B model with refinement / without refinement.

5) The tool should support Real, Enumeration and Bool data types.

6) The tool should support pre and post states (e.g., x , x_{-1}) during model generation.

3.3.1 General Thoughts on Requirements

According to the requirements given above, we have the following design and implementation thoughts on them.

1) The tool should generate a Rodin dependent project from tabular expressions.

   A Rodin project should generate three kinds of files: project file, Context file, and Machine file. The project file only contains a project name and some fixed format XML information that are easy to generate. The generated Context file contains static information of system in form of carrier sets, constants and axioms. The generated Machine file contains dynamic information of system in form of variables, invariants, events (guards and actions), file refinement relation, and Machine "sees" Context relation. According to the separation of concerns principle, the project file generation, Machine file generation and Context file generation should be implemented in separate modules. Then a project generation module can call the three modules to generate the final Rodin project.

   When a user manually creates a Rodin file for developing an EventB specification in the Rodin Platform, the Rodin Platform implicitly creates a XML file in the current workspace for saving the specification. Our tool is designed to simulate the user operations and inputs by directly editing into the XML file. Therefore, there is no essential difference between the manual process and the automatic process.

2) The tool should allow to input single table / multiple tables for model generation.

   The tool can save all the tables belong to one project to the same folder first. Then the tool can generate the Event-B model from all the tables in that folder.

   From the implementation perspective, we can first develop a tool prototype which allows to input single table for model generation. In order to support multiple tables, based on the single table tool prototype, we just need to add one more loop in the frontend Matlab to gather all the data information and put them in a tablesList (List<Table> type in Java) and pass them all to the backend Java part. In the Java part, the Context generation is created

using variable types in all the tables. Similarly, for the Machine generation module, the Variables & Invariants can be generated by iterating all the variables in every table in the tablesList. The events can be generated by traversing each table in the tablesList and then create the current table events. But the events number should be increased globally. Which means if the last table finishes at event N, the next table should start from N+1 rather than 1.

3) The tool should support two output modes: single output & multiple outputs.

   Since our tool is based on the TET tool [8], and the single output and multiple outputs GUI are already implemented for the PVS and CVC3 generator in their work. What we have to do is acquiring all the table information from the frontend Matlab GUI and doing appropriate processing in the Java backend according to the output mode. The main difference between these two modes lies in that, in the single output mode, every ordinary (non-initialisation) event has only one action and the action name is just $act1$. While in the multiple outputs mode, an ordinary event has N actions. Namely, $act1, act2, ...actN$. Where N is the number of outputs in the Matlab GUI.

4) The tool should produce an Event-B model with refinement / without refinement.

   This is a completely new feature. We have to add a menu for the user to choose the current refinement mode in the Matlab GUI. Once the current mode is selected, the information is saved in the GUI object for reference in the backend. In the backend, if it is in the without refinement mode, the whole table information acquired from the frontend will be transformed into one large model file (Machine file). Otherwise, in the with refinement mode, the first m columns and the outputs are used to generate the $(m-1)$th refinement model file. Where $1 \leq m \leq N, N$ is the total columns of the table. Which means, in this mode, N Machine model files are generated.

5) The tool should support Real, Enumeration and Bool data types.

   Since Event-B language only supports Integers. Therefore, the tool can just support the subset of Real: Integers.

   For the Enumeration type, a Matlab file defining the enumeration type is needed before the user can create the Enumeration type in the frontend GUI. In the backend, the Enumeration type is generated into a Context file. Specifically, to generate the Enumeration declaration XML file, we have to generate the Sets, Constants, and Axioms using the information of Enumeration type and values acquired from the Matlab file.

   For the bool type, the lower case bool, true and false in the Matlab GUI are converted to upper case BOOL, TRUE and FALSE for Event-B Model in the Rodin Platform.

6) The tool should support pre and post states (e.g., x , x_{-1}) during model generation.

   In Event-B, the last state can be just expressed by the current output variable name, like $var = var$. But TET cannot support this format. So we design the last state of the output variables to be expressed by the reserved pattern var_prev. For example, the last state value of variable x is expressed by x_prev. This is implemented by adding the postfix $\_prev$ to the all variables that have last state values in the Matlab GUI during the requirement creating process, and then removing the $\_prev$ in all variables during events generation process to accommodate the EventB syntax.

3.4 TX2EB Translation Principles and Rules

This section mainly describes the principles and rules that we use to translate the type checked tabular expressions to the standard formal specification of Event-B. Table 3.1 lists the general principles that the tool used to generate an Event-B model from tabular expressions.

Tabular Expression	**Event-B Model**
Enumeration Type	Enumerated Set
Input and output variables	Variables, Invariants
Expression Name (table name)	Event-B project name
Grid number & Sub-Grid number	Event name
Table Conditions	Event Guards
Table Results	Event Actions
Table Conditions & Results	Events

Table 3.1: Tabular Expressions to Event-B General Principles

The first column of the table contains a list of elements of tabular expressions. The second column of the table contains corresponding elements in the Event-B modeling language. For example, an enumeration type of the tabular expression is translated into an equivalent Event-B enumerated set, which is made up of a carrier set, constants, and axioms. All the input and output variables of tabular expressions are translated into the equivalent Event-B variables and invariants. The table name is used to generate the Event-B project name. The grid number and sub-grid number are used to generate the event name. The tabular expression conditions are translated into the equivalent event guards, and the table results are transformed to the equivalent event actions. Finally, the tabular expression conditions and results are translated into events of Event-B models.

3.4.1 Data Type and Expression Mapping Rules

A set of data types, symbols, and expressions supported by TX2EB is given in Table 3.2, which clearly illustrates the mapping rules that we have used to translate the tabular expressions into Event-B. In the following Table 3.2, VAR denotes VARIABLES, INV denotes INVARIANTS, INTI means the initialization event (the action part) in an Event-B Machine, and *Color* is an user defined Enumeration type.

Tabular Expression	**Event-B Language**	**Comments**
$x \;\; OR \;\; x : real$	x (VAR) $x \in \mathbb{Z}$ (INV) $x :\in \mathbb{Z}$ $(INTI)$	Real Type
$x : bool$	x (VAR) $x \in BOOL$ (INV) $x :\in BOOL$ $(INTI)$	Bool Type
$x : Color$	x (VAR) $x \in Color$ (INV) $x :\in Color$ (INTI)	Enumeration Type
$x : \{t : real \mid t > 0\}$	x (VAR) $x \in \{x \mid x \in \mathbb{Z} \wedge (x > 0)\}$ (INV) $x :\in \{x \mid x \in \mathbb{Z} \wedge (x > 0)\}$ (INTI)	Sub Type
x_prev	x	Pre Post State
$x + y$	$x + y$	Addition
$x - y$	$x - y$	Subtraction
$x * y$	$x * y$	Multiplication
x/y	$x \div y$	Division
$x \sim= y$	$x \neq y$	Not Equal
$x == y$	$x = y$	Equal
$x > y$	$x > y$	Great Than
$x >= y$	$x \geq y$	Great or Equal Than
$x < y$	$x < y$	Less Than
$x <= y$	x$\leq y$	Less or Equal Than
$\sim x$	$\neg(x = TRUE)$	NOT
$x\&\&y$	$(x = TRUE) \wedge (y = TRUE)$	AND
$x \| \| y$	$(x = TRUE) \vee (y = TRUE)$	OR

Table 3.2: Data Type and Expression Mapping Rules

3.4.2 Tabular expressions to Event-B Translation Rules

In this section, we describe the translation rules for generating Event-B models from tabular expressions. Fig. 3.2 shows an editor of our tool that is used for editing a tabular expression.

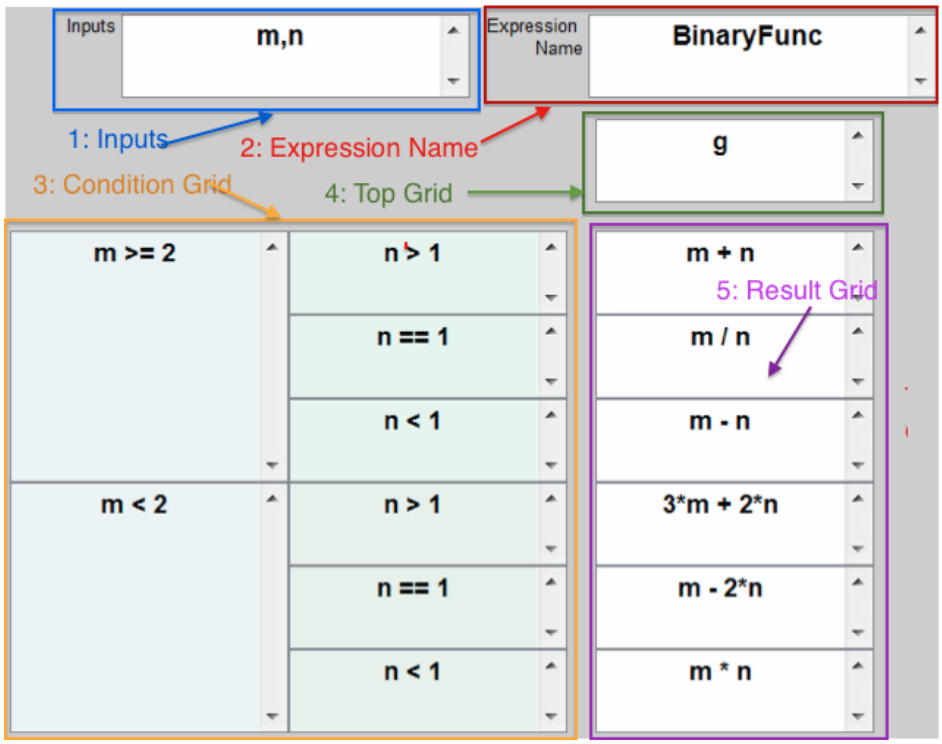


Figure 3.2: The Tabular Expression Used to Explain Translation Rules

As shown in the Fig. 3.2, the editor has five main components to describe tabular expressions. Component 1 (the blue color section) is the *Inputs* on the top left, which is used to define all the input variables used in the table. Component 2 (the red color section) is the *Expression Name*, which is used to input the name of the table. Component 3 (the orange color section), the big box on the left bottom of the table, is called the *condition grid*. It is used to input all the conditions of the tabular expression. We call the text box (Component 4, the green section) under the Expression Name text box, the *top grid*, which is used to define all the output variables. Then Component 5

(the purple section), the one to the right of the condition grid and under the top grid, is called the *result grid*. It is used to input all the output variable values. To create a single output table or multiple outputs table using our tool, a user can choose an option as shown in Fig. 3.3:

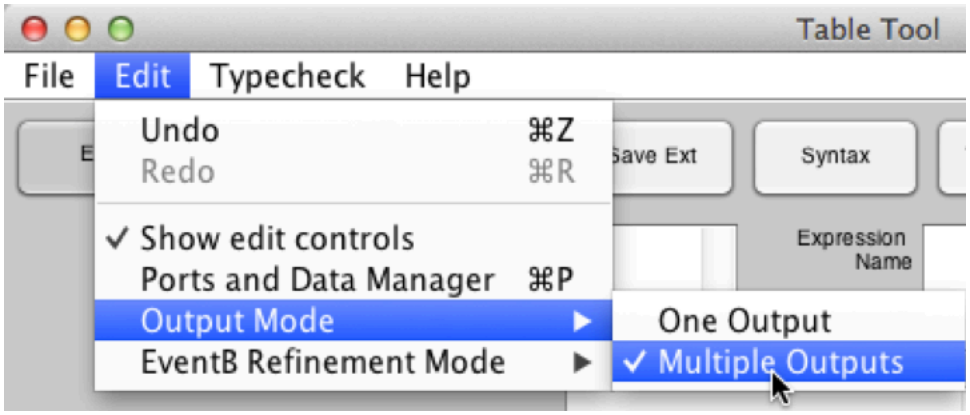


Figure 3.3: Output Mode Selection Menu

Before generating an Event-B model, the user must choose an option to generate an Event-B model with refinement or without refinement (no refinement) as shown in Fig. 3.4:

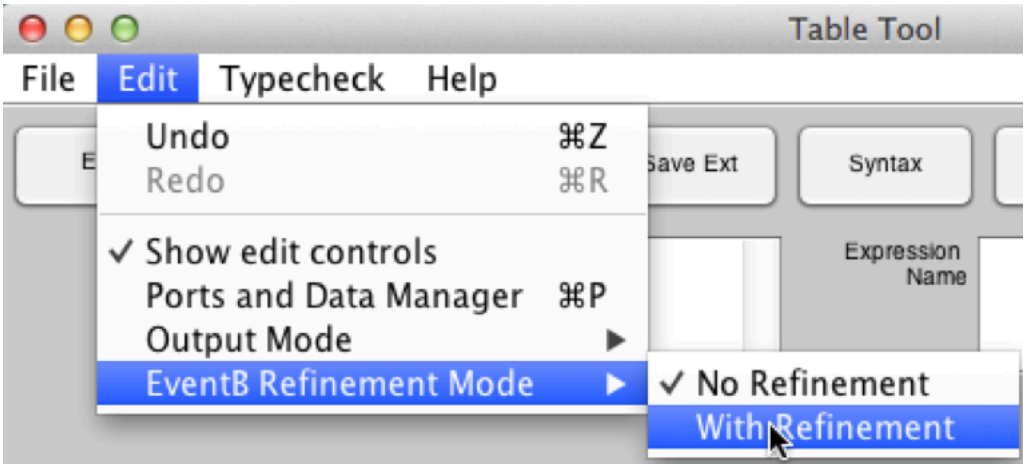


Figure 3.4: Refinement Mode Selection Menu

To Process Tabular Expression Variables

All the variables defined in the inputs text box (see Fig. 3.2) and the output variables in the top grid (see Fig. 3.2) are translated into *variables*, *invariants* and a list of *actions of the initialization event* of an Event-B model.

For example, the variables of Fig. 3.2 are translated into the following:

```
VARIABLES
  m
  n
  g
INVARIANTS
  inv1   :   m ∈ ℤ
  inv2   :   n ∈ ℤ
  inv3   :   g ∈ ℤ
EVENTS
  INITIALISATION   ≙
  STATUS
   ordinary
  BEGIN
   act1   :   m :∈ ℤ
   act2   :   n :∈ ℤ
   act3   :   g :∈ ℤ
  END
```

To Process Tabular Expression Name

The table name which is given in the Expression Name part of the tabular expression (see Fig. 3.2), is translated into an Event-B project name for single input table. Later, to support multiple input tables, all the input tables are saved into the same source folder. The source folder name is translated into the Event-B project name.

To Process Tabular Expression Arithmetic / Logic Expressions

All the arithmetic and logical expressions of the table are translated into equivalent expressions of the Event-B language. For example, the expression $m \,/\, n$ in Fig. 3.2 is translated into $(m \div n)$ in the Event-B model. All the expression mapping rules from tabular expression to Event-B language are given in the Table 3.2.

To Process Tabular Expression Rows

Each row of the tabular expression is transformed into an ordinary event (we use ordinary events to denote all the non-initialization events) in a model of Event-B. To extract information from tabular expressions for producing a formal model, we need to consider both rows and columns together to traverse the tables. In Fig. 3.2, if we consider the first column, then the two grid conditions in the first column and their corresponding values in the result grid are converted to two abstract non-deterministic events in an Event-B model (with refinement mode). If we consider the second column, then 6 conditions in the second column and the first two conditions in the first column as well as their corresponding values in the result grid are translated into 6 concrete events an the Event-B model (no refinement mode / with refinement concrete model).

The grid and sub-grid cell numbers in each row of the table are used to produce an event name; for example, the conditions $(m < 2)$ and $(n > 1)$ together with the output value $(3*m+2*n)$ in the Fig. 3.2 are converted into one event in the Event-B model. The event name is $evt2\_1$ since the grid cell number and sub-grid cell number are 2 and 1 respectively. If the users choose to generate Event-B model with refinement mode, for the abstract model, the second grid condition $(m < 2)$ and the values $\{(3*m+2*n), (m-2*n), m*n\}$ together are converted into one event in the Event-B model. The event name will be $evt2$. Fig. 3.5 illustrates $evt2$ in the abstract model and $evt2\_1$ in the concrete model. More detailed translation rules from tabular expression conditions to guards, from output variable and result values to actions are given in the following section.

To Process Tabular Expression Condition Grid

- If a user selects a mode to generate an Event-B model **without refinement**, all the conditions on the path from grid root (cell in the first column) to grid leaf (cell in the last column) are translated into event

```
evt2   ≙
STATUS
 ordinary
WHEN
 grd1   :   (m < 2)
THEN
 act1   :   g :∈ {((3 * m) + (2 * n)),(m - 2),(m * n)}
END

evt2_1   ≙
STATUS
 ordinary
REFINES
 evt2
WHEN
 grd1   :   (m < 2)
 grd2   :   (n > 1)
THEN
 act1   :   g ≔ ((3 * m) + (2 * n))
END
```

Figure 3.5: Generated Event-B Event Examples

guards directly. For instance the guards of $evt2\_1$ (see Fig. 3.2) can be translated as follows:

```
WHEN
 grd1   :   (m ≥ 2)
 grd2   :   (n = 1)
```

- If a a user chooses to generate the Event-B model **with refinement**, all the conditions on the path from condition grid root to condition grid leaf are also translated into event guards, but the guards are gradually added from the abstract model to the concrete model. For instance the guards of $evt2$ of the abstract model in the Fig. 3.2 will be $grd1 : (m < 2)$. It means that the abstract model is constructed from the first column which has one input variable m. In the concrete model, the guards will be exactly the same as the above without refinement mode (with both variables m and n). The following figure illustrates the $evt2$ guard in the abstract model and the $evt2\_1$ guards in the concrete model of the

tabular expression in Fig. 3.2.

Abstract Model Guard

```
WHEN
 grd1   :   (m < 2)
```

Concrete Model Guard

```
WHEN
 grd1   :   (m < 2)
 grd2   :   (n > 1)
```

To Process Tabular Expression Result Grid & Top Grid

- If a user chooses to generate an Event-B model **without refinement**, the output variable in the top grid and values in the result grid are translated into event actions (see examples of top grid and result grid in Fig. 3.2). In the multiple outputs mode, if a table has X output variables, then each variable and its corresponding value is translated into one event action. All the actions are named as act1, act2...actX. For instance the event action of $evt2\_1$ in the Fig. 3.2 is:

```
THEN
 act1   :   g ≔ ((3 ∗ m) + (2 ∗ n))
```

- If a user chooses to generate an Event-B model **with refinement**, the output variable in the top grid and values in the result grid are also translated into event actions. But the actions are refined from nondeterministic actions to deterministic actions. In the abstract model, the output variable in the top grid and the values in the result grid are translated into nondeterministic actions. But in the concrete model, all the actions become deterministic, just the same as the no refinement mode. For instance, in Fig. 3.2, the event actions of the output variable g and the three values $\{(3*m+2*n),(m-2*n),m*n\}$ that corresponds to the condition $m<2$ are defined in the abstract and concrete models as shown in Fig. 3.6.

The above generated Variables, Invariants, and Events (Guards, Actions) are all written into an Event-B model Machine file.

Abstract Model Action

```
THEN
 act1   :   g :∈ {((3 * m) + (2 * n)),(m - 2),(m * n)}
```

Concrete Model Action

```
THEN
 act1   :   g = ((3 * m) + (2 * n))

THEN
 act1   :   g := (m * n)

THEN
 act1   :   g := (m - (2 * n))
```

Figure 3.6: Event-B Actions in the Abstract and Concrete Models

To Process Tabular Expression Enumeration

In order to use a user defined enumeration type in the tabular expression, We have to define the enumeration type using Matlab script first. For example, if we want to define an enumeration type named *Color*, which has three elements: *Red*, *Green* and *Blue*. We should create the following definition of *Color* using Matlab script:

```
classdef Color < Simulink.IntEnumType
  enumeration
    Red(0)
    Green(1)
    Blue(2)
  end
end
```

After defining the above *Color* enumeration type using the Matlab script, we also need to register the enumeration information to TET in the command line using the command TET.getInstance.registerEnumeration('Color'). Then we can use the *Color* as a known data type in the inputs box of the GUI. For example, we can define an enumeration variable c of the *Color* enumeration type in the form of *c:Color*. This Matlab Enumerated type can be translated into the equivalent Event-B Enumerated set as follows:

- Each user defined enumeration type is defined as a Carrier Set. In this example, the type *Color* is converted to a Carrier Set as follows:

```
SETS
  Colors
```

- Each enumeration value is declared as a Constant. In the *Color* example, the three enumeration values *Red, Green* and *Blue* are translated into three Event-B Constants as follows:

```
CONSTANTS
  Red
  Green
  Blue
```

- The enumeration type and enumeration values all together are converted to an Axiom, which describes the partition relation between the enumeration type and its values. For example, we can define an axiom for the *Color* as follows:

```
AXIOMS
  axm1   :   partition(Colors, {Red}, {Green}, {Blue})
```

The generated enumerated Carrier Set, Constants, and Axiom are all written into an Event-B model Context file.

3.5 Test Driven Development (TDD)

The frontend (the GUI) and the backend (the generation logic) of our tool are developed separately. The former is written in Matlab script based on TET (Tabular Expression Tool) [9] while the later is developed in Java language based on jTET (java TET) [5]. If we want to test the tool, we have to use Maven (a software project management tool) to package the Java code into a

jTET.jar file and then put it to the TET/jTET folder. Then we can launch (or restart) the Matlab application and create the tabular expression in the GUI to test whether the generation process of EventB model is right or not. This manual process is both inefficient and time-consuming.

Thanks to the TDD (Test Driven Development) method. We can simulate the Matlab input data by creating Java test cases in JUnit for different tool features and scenarios. Whenever a new feature is introduced for the tool, we would first create a new test case (or modify one current test case) for that feature. Then we add code to support the new feature. After that, we first run all the test cases to make sure the new feature does not break any existing test cases, and it also satisfies the new test case. If there is any test case failure, we have to go back to modify the code. This process is repeated until all the test cases pass. Then we can start to run the tool in Matlab to check whether everything works properly. It is possible that the tool still has problems because the test cases cannot be complete. In that case, we go back to add a new test case or modify the existing test case written in Java to cover the scenario under which the tool fails and then rerun the test case. The test case will fail. Now we can fix the problem to make the test case pass and check it again in Matlab. Without TDD test cases, we have to package the Java code and import it into Matlab every time to test whether it works, which requires more time to fix all the found problems.

Moreover, TDD can also help us quickly verify whether the code refactoring is correct or not simply by running the all the test cases.

3.6 Tool Architecture

As mentioned before, rather than creating everything from scratch, our work inherits from TET [8], which uses a three-layer architecture: GUI layer, application layer, and data representation layer. We mainly extend the existing toolbox by adding one more Event-B Generator in the application layer. However, we also modify the GUI of the TET tool to support the Event-B model

generation. The data representation layer is untouched. The new architecture of the tool is illustrated in Fig. 3.7 (modified from Fig. 3.1 in [9]).

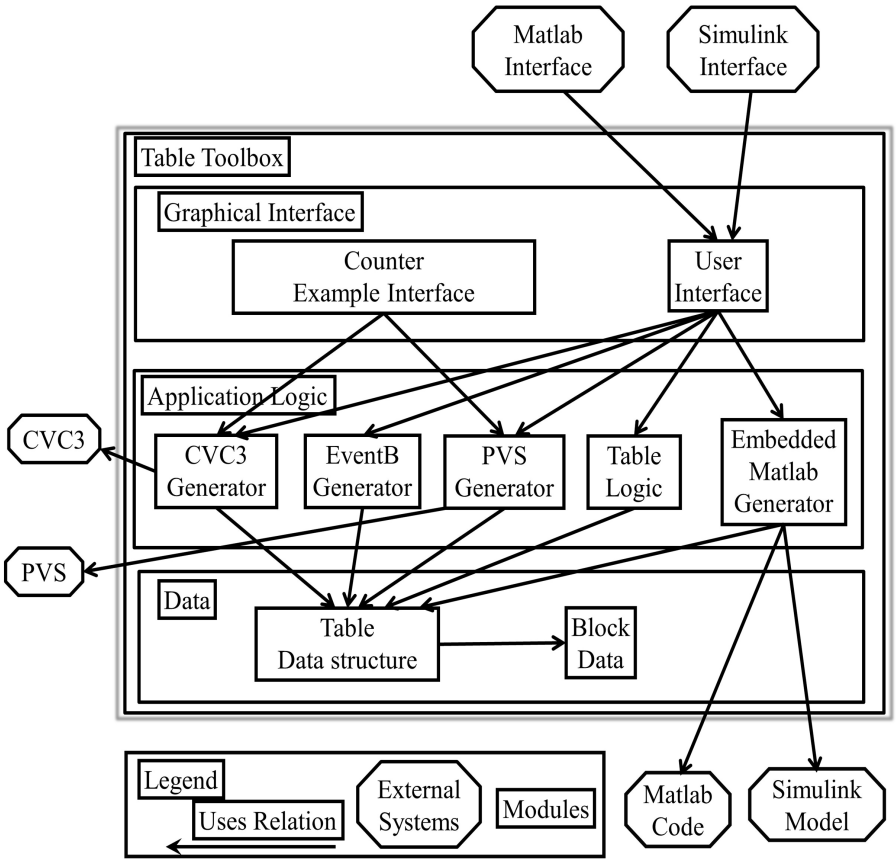


Figure 3.7: TX2EB Tool Architecture

3.7 Types of Tool Modes

Our tool supports four different types of modes which are: single output no refinement (SONR), multiple outputs no refinement (MONR), single output with refinement (SOWR) and multiple outputs with refinement (MOWR).

SONR allows single output only, and the whole table is transformed into one Event-B Machine file. MONR differs with SONR in the outputs, MONR

can support multiple outputs with the same condition. The difference between SOWR, MOWR and SONR, MONR is that the former two modes introduce refinement in their transformation process from tabular expressions to Event-B model. For example, if we have a two-column table. In the refinement mode, the first column conditions and their values in the result grid are transformed into an abstract Machine M0 and then the first and the second column conditions together with the values in the result grid are converted into a concrete Machine M1. And the concrete Machine M1 refines the abstract Machine M0. In the no refinement mode, the whole two-column table is directly transformed into one concrete Machine M. The differences and similarities among the four modes of the TX2EB tool are depicted in table 3.3.

Table 3.3: Comparison among tool modes

Tool Modes / Whether Support	$SONR$	$MONR$	$SOWR$	$MOWR$
$Multiple\ outputs?$	No	Yes	No	Yes
$Refinement?$	No	No	Yes	Yes

The reason for introducing table refinement lies in the observation that it is impossible or very difficult to manage all the requirements in one single Machine file for large and complex systems. For example, in the Insulin Infusion Pump Project [30], the Event-B model was so big that we needed to break it into 43 layers (we also call each column in the table a layer). Therefore, we need a mechanism that supports creating tables step by step. The Event-B refinement property is suitable for this purpose. Also, the gradual refinement is a good way to manage complexity for large systems and it can help us focus on different aspect of the system each time. Finally, the Event-B Rodin Platform can automatically generate Proof Obligations (POs) to check the consistency among different layers of refinement, especially for projects that are composed of multiple tables.

3.8 Implementation and Rationale

3.8.1 Implementation Decision

At the beginning of the development phase, we found two possible methods to implement the tool:

- Method 1: Call Rodin APIs.
- Method 2: Generate the Rodin XML file directly.

However, there are some difficulties for Method 1:

- There are too many dependencies.
- Rodin is an eclipse plugin, but existing work was developed in IntelliJ.
- The required ANTLR [1] version is not available in Eclipse.

On the other hand, there are some benefits for Method 2:

- Rodin APIs are not required.
- Easy to set up the develop environment.
- Easy to reuse the existing code.

Therefore, the Method 2 was chosen to implement the tool in our work.

3.8.2 Classes and Interfaces

The whole project is mainly implemented in Java together with some Matlab code to support the GUI. For the current version of the tool, we have nine classes and four interfaces for the transformation process. The nine classes are EventBProject, EventBTableGenerator, EventBTypeDeclarationGenerator, EventBVariablesDeclarationGenerator, HierarchicalGridEventBGenerator,

[1] Another Tool For Language Recognition, a parser generator that uses top-down parsing. ANTLR is used to generate the abstract syntax tree in our tool.

EventBExpressionGenerator, EventBFileWriter, TableGroup, and RefinementMode. The four interfaces are TypeDeclarationGenerator, VariablesDeclarationGenerator, ExpressionGenerator, and HierarchicalGridDFCheckerGenerator.

The relations between the classes and interfaces of our tool project are illustrated in Fig. 3.8. The software architecture is designed for the following reasons. First, the system is decomposed according to functionality. There are four modules, namely, the project (generate the project), file writer (save the generated file), the table generator (generate the Machine and Context files), and the hierarchical grid generator (walk tables to collect information). Each module is responsible for one functionality, and all the modules are loosely coupled. It's very easy to change one module without affecting the other modules. Second, the RefinementMode enumeration class is pulled out as a single class so that we can define it once and use it everywhere. Finally, the generator interfaces of type, variables, and expression are created so that not only Event-B but some other (e.g., CVC3 or SMTLIB) generators can also implement these interfaces. We can use runtime polymorphism to reduce code duplication between all the classes that implement these interfaces.

Specifically, the EventBProject class is used to create the Rodin dependent Event-B Model project, which can be directly imported into the Rodin Platform to generate the Event-B model. The project folder and the project file are both created inside this class. Also the *m_tablesGroup* parameter is configured using the setTablesGroup method of this class and it is passed to the EventBTableGenerator object created inside the EventBProject class to generate the whole Event-B model XML file of all the tables in a project. Ever since the single table version of the tool, we have expected that we are going to extend the code later to support multiple tables. Therefore, we designed the project to contain a single table originally and when we decided to support multiple tables, we just used a List of tables (List<Table>) to substitute the single Table object. The whole architecture stayed the same as before. Put another way, we use a more general design pattern, and we design for change and extensibility.

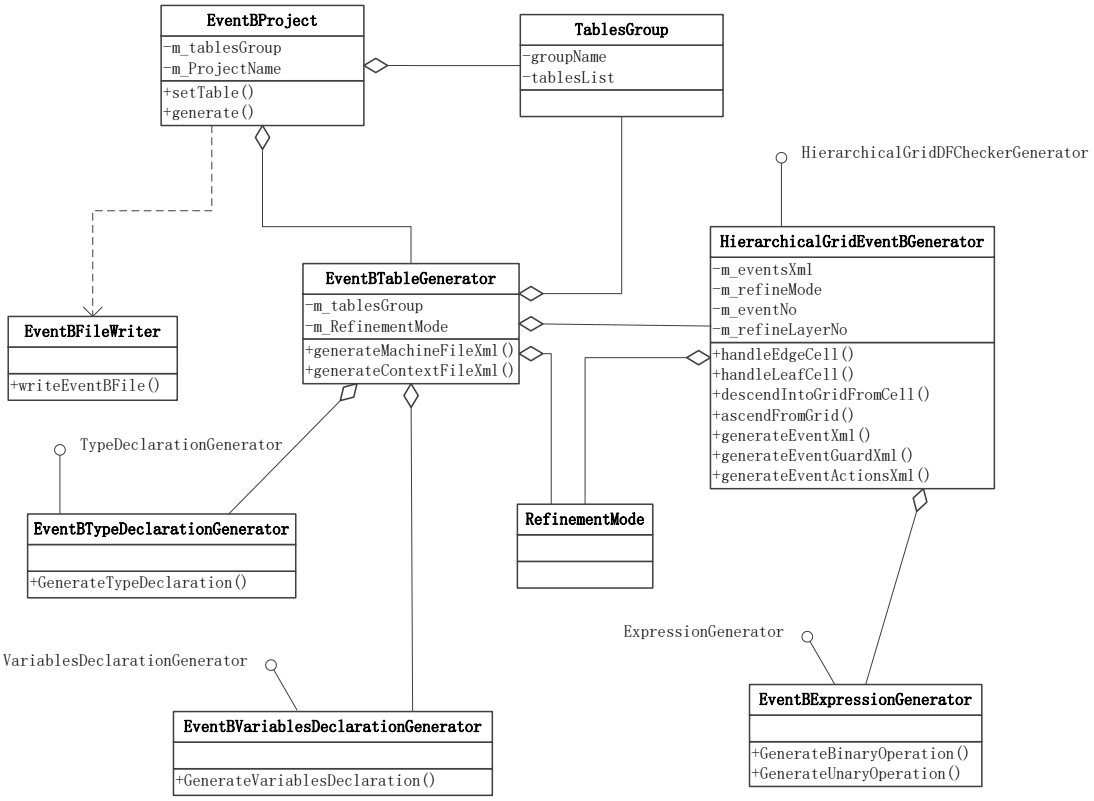


Figure 3.8: Classes and Interfaces UML diagram

As mentioned above, the EventBTableGenerator acquires parameters from the project class and controls the whole process of the Event-B Model (both the Machine and the Context) XML file generation. There are seven steps to generate the Machine XML file. The Machine file generation process is implemented in the method generateMachineFileXml() as illustrated in Fig. 3.9.

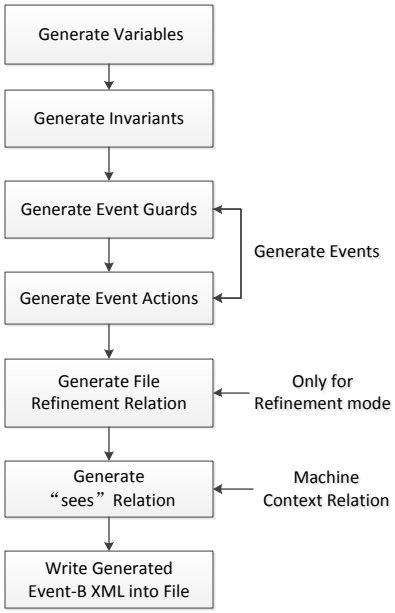


Figure 3.9: Process of Machine File Generation

Similarly, the method generateContextFileXml(), which generates the Context XML file, contains four steps as shown in Fig. 3.10.

The class EventBVariablesDeclarationGenerator implements the VariablesDeclarationGenerator interface. It generates the Machine Variables & Invariants part of the XML file (corresponds to step 1-2 in Fig. 3.9).

The class EventBTypeDeclarationGenerator implements the TypeDeclarationGenerator interface. It generates the types XML that is defined in the Model Context, like Enumerations. This is realized by generating the Sets, Constants and Axioms respectively (corresponds to step 1-3 in Fig. 3.10).

The HierarchicalGridEventBGenerator class is the most essential one of

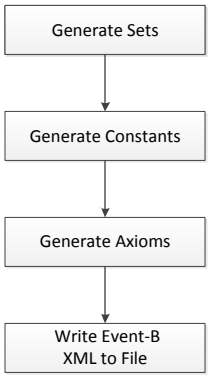


Figure 3.10: Process of Context File Generation

the whole project. It implements the HierarchicalGridDFCheckerGenerator interface to walk the hierarchical grid in depth-first (DF) order so as to gather the input information from the table grid to generate the events XML. This class implements the rules mentioned in Section 3.4.2. Specifically, it first generates the event names, and then it transforms all the conditions from the root cell (first column cell) to the leaf cell (last column cell) to the guards of the Event-B model. Finally, the output variables and their values are converted to the actions part of the XML file in this class (corresponds to step 3-4 in Fig. 3.9).

The EventBFileWriter class is created to write the generated XML into either a Machine or a Context file (corresponds to step 7 in Fig. 3.9 and step 4 in 3.10).

The TablesGroup class does not exist before introducing multiple tables. Instead, a Table class is used in the single table version tool. It is created to facilitate the parameters passing between different classes after introducing multiple tables.

The RefinementMode class defines the enum data type that is used in the whole project. It contains two items: *withRefinement* and *withoutRefinement*, which denote whether we want to generate the Event-B model with refinement or without refinement, respectively.

Furthermore, one parameterized test class named ParameterizedEventBTable-GeneratorTest which contains six tables covering all the different modes and important features of the tool is created in JUnit to validate the correctness of the transformation process. After verifying the test cases in JUnit, the source code is packaged into a jTET.jar file using Maven and put into a Matlab folder to do the integration test together with the Matlab GUI. We create many different tables for all the four modes in the Matlab GUI and import the generated project into the Rodin Platform. They all succeed in generating the right Event-B models and POs.

3.8.3 Context File Generation Process

The Context file generation process is illustrated in Fig. 3.10. The Sets, Constants and Axioms are generated separately using different java calling methods and then these generated codes are written in a Context file in XML format. Please note that only enumeration type is supported in the Context file now. Therefore, the following Sets, Constants, and Axioms generation steps are part of the enumeration type generation process.

Sets Generation

This is implemented by the method generateSetsXml(Set<Type> types): String. It iterates every input type in the Set<Type> parameter to judge whether the current type is an instance of EnumerationType. If this is true, the enumeration name is used to set the *carrierSet name* and the *identifier* in the Set XML (by appending the enumeration name to the *carrierSet name* and the *identifier* in the XML, as shown in the output XML). Finally the generated XML string of Carrier Sets is returned. The pseudo code for the Sets Generation algorithm is given as follows.

```
generateSetsXml(types)
    initialize setsXml to null
    for(type : types)
        initialize oneSetXml to null
        if(type instanceof EnumerationType)
```

```
            append carrierSet name XML to oneSetXml
            append type.name() to oneSetXml
            append carrierSet identifier XML to oneSetXml
            append type.name() to oneSetXml
        append oneSetXml to setsXml
    return setsXml
```

To describe the Sets Generation process for enumeration type, we choose the example *Color* enumeration from Section 3.4.2. A required input from tabular expression, an output in XML format to support the Rodin project, and finally output to appear on the Rodin editor after importing the generated code are given as follows:

Input from tabular expression:
Color Enumeration Type.

Output in XML format:

```
<org.eventb.core.carrierSet name="Colors" org.eventb.core.identifier="Colors"/>
```

Output in the Rodin editor:

```
SETS
  Colors
```

Constants Generation

This is implemented by the method generateConstantsXml(Set<Type> types): String. It iterates every input type in the Set<Type> parameter to judge whether the current type is an instance of EnumerationType. If this is true, another inner for loop is used to iterate each enumeration value to generate one Constant XML by setting the enumeration value to the *Constant name* and *Constant identifier* (i.e., by appending the enumeration value to the *name* and the *identifier* in the XML, as shown in the output XML). The generated

Constant XML is then concatenated to the Constants XML. Finally, the generated XML string of Constants is returned. The pseudo code for the Constants Generation algorithm is as follows.

```
generateConstantsXml(types)
    initialize constantsXml to null
    for(type : types)
        if(type instanceof EnumerationType)
            for(enumConstantValue : type.enumerationValues)
                initialize oneConstantXml to null
                append Constant name XML to oneConstantXml
                append enumConstantValue to oneConstantXml
                append Constant identifier XML to oneConstantXml
                append enumConstantValue to oneConstantXml
                append oneConstantXml to constantsXml
    return constantsXml
```

To describe the Constants Generation process for enumeration type, we choose the example *Color* enumeration from Section 3.4.2. A required input from tabular expression, an output in XML format to support the Rodin project, and finally output to appear on the Rodin editor after importing the generated code are given as follows:

Input from tabular expression:
Color Enumeration Values.

Output in XML format:

```
<org.eventb.core.constant de.prob.symbolic.symbolicAttribute="false" de.prob.units.inferredUnitPragma
Attribute="" de.prob.units.unitPragmaAttribute="" name="Red" org.eventb.core.identifier="Red"/>
<org.eventb.core.constant de.prob.symbolic.symbolicAttribute="false" de.prob.units.inferredUnitPragma
Attribute="" de.prob.units.unitPragmaAttribute="" name="Green" org.eventb.core.identifier="Green"/>
<org.eventb.core.constant de.prob.symbolic.symbolicAttribute="false" de.prob.units.inferredUnitPragma
Attribute="" de.prob.units.unitPragmaAttribute="" name="Blue" org.eventb.core.identifier="Blue"/>
```

Output in the Rodin editor:

```
CONSTANTS
  Red
  Green
  Blue
```

Axioms Generation

This is implemented by the method generateAxiomsXml(Set<Type> types): String. It iterates every input type in the Set<Type> parameter to judge whether the current type is an instance of EnumerationType. If this is true, the enumeration name is used to configure the *axiom name* and the Axiom *partition predicate* (see the output XML) in the Axiom XML. Then another inner for loop is used to pull out all the enumeration values, which are translated into the *partition predicate* components in the Axiom XML. And finally the generated XML string of Axioms is returned. The pseudo code for the Axioms Generation algorithm is as follows.

```
generateAxiomsXml(types)
    initialize axiomsXml to null
    initialize axiomNo to 0

    for(type : types)
        initialize oneAxiomXml to null
        axiomNo <- axiomNo + 1
        if(type instanceof EnumerationType)
            append Axiom name XML to oneAxiomXml
            append type.name to oneAxiomXml
            append Axiom label XML to oneAxiomXml
            append axiomNo to oneAxiomXml
            append Axiom partition XML to oneAxiomXml
            append type.name to oneAxiomXml

            initialize enumConstantsXml to null
            for(enumConstantValue : type.enumerationValues)
                append enumConstantValue to enumConstantsXml
                append enumConstantsXml to oneAxiomXml
                append oneAxiomXml to axiomsXml

    return axiomsXml
```

To describe the Axioms Generation process for enumeration type, we choose the example *Color* enumeration from Section 3.4.2. A required input from tabular expression, an output in XML format to support the Rodin project, and

finally output to appear on the Rodin editor after importing the generated code are given as follows:

Input from tabular expression:
Color Enumeration Type & Enumeration Values.

Output in XML format:

```
<org.eventb.core.axiom name="Colors" org.eventb.core.label="axm1"
org.eventb.core.predicate="partition(Colors, {Red}, {Green}, {Blue})"/>
```

Output in the Rodin editor:

```
AXIOMS
  axm1   :   partition(Colors, {Red}, {Green}, {Blue})
```

Write XML to Context File

All the generated XML strings of Sets, Constants and Axiom are written into a Context file in the project folder.

3.8.4 Machine File Generation Process

The Machine file generation process is illustrated in Fig. 3.9. The Variables, Invariants, Events (Guards, Actions), File Refinement Relation and "Sees" Relation are generated separately using different java calling methods, and then these generated codes are written in a Machine file in XML format.

Variables Generation

This is implemented by the method generateVariablesXml(StringBuilder ret, Collection<Variable> variables): void. It iterates every input variable in the Collection<Variable> parameter to judge whether the current variable is a reserved variable name (i.e., variable end with *_prev*). If this is true, it is

unnecessary to generate the Variable XML for the current variable. Otherwise, the variable name is used to set the *name* and *identifier* (see the output XML) in the Variable XML. And finally the generated XML string of Variables is returned by the update parameter *ret*. The pseudo code for the Variables Generation algorithm is as follows.

```
generateVariablesXml(ret, types)
    for(var : variables)
        initialize oneSetXml to null
        if(var is a reserved key variable)
            append Variable name XML to ret
            append var.name to ret
            append Variable identifier XML to ret
            append var.name to ret
```

To illustrate the Variables Generation process, we choose the table in Fig 3.2 as an example. A required input from tabular expression, an output in XML format to support the Rodin project, and finally output to appear on the Rodin editor after importing the generated code are given as follows:

Input from tabular expression:

three variables m, n, g.

Output in XML format:

```
<org.eventb.core.variable de.prob.units.inferredUnitPragmaAttribute=""
de.prob.units.unitPragmaAttribute="" name="m" org.eventb.core.identifier="m"/>
<org.eventb.core.variable de.prob.units.inferredUnitPragmaAttribute=""
de.prob.units.unitPragmaAttribute="" name="n" org.eventb.core.identifier="n"/>
<org.eventb.core.variable de.prob.units.inferredUnitPragmaAttribute=""
de.prob.units.unitPragmaAttribute="" name="g" org.eventb.core.identifier="g"/>
```

Output in the Rodin editor:

```
VARIABLES
  m
  n
  g
```

Invariants Generation

This is implemented by the method generateInvariantsXml(StringBuilder ret, Collection<Variable> variables): void. It iterates every input variable in the Collection<Variable> parameter to judge whether the current variable is a reserved variable name (i.e., variable end with *_prev*). if this is true, it is unnecessary to generate the Invariant XML for the current variable. Otherwise, the *invariant name* is set to InvariantX and the Invariant *label* (see the output XML) is set to invX, where X is the Invariant number (start from 1). Finally, depending on whether the current variable has subtype or not, the Invariant *predicate* (see the output XML) is set in the form of Variable $\in$ SubType (like $m \in \{m|m \in \mathbb{Z} \wedge (m > 0)\}$) or Variable $\in$ Type (like m $\in \mathbb{Z}$). And the generated XML string of Invariants is returned by the update parameter *ret*. The pseudo code for the Invariants Generation algorithm is as follows.

```
generateInvariantsXml(ret, types)
    invariantsNum <- 1
    for(var : variables)
        initialize oneSetXml to null
        if(var is a reserved key variable)
            append Invariant name XML to ret
            append invariantsNum to ret
            append Invariant label XML to ret
            append invariantsNum to ret
            append Invariant predicate XML to ret
            if(subtypePredicate of var is not null)
                append generated Subtype Predicate to ret
            else
                append generated Type to ret
            invariantsNum <- invariantsNum + 1
```

To illustrate the Invariants Generation process, we choose the table in Fig 3.2 as an example. A required input from tabular expression, an output in XML format to support the Rodin project, and finally output to appear on the Rodin editor after importing the generated code are given as follows:

Input from tabular expression:

three variables m, n, g.

Output in XML format:

<org.eventb.core.invariant name="Invariant1" org.eventb.

core.label="inv1" org.eventb.core.predicate="$m \in \mathbb{Z}$"/>

<org.eventb.core.invariant name="Invariant2" org.eventb.

core.label="inv2" org.eventb.core.predicate="$n \in \mathbb{Z}$"/>

<org.eventb.core.invariant name="Invariant3" org.eventb.

core.label="inv3" org.eventb.core.predicate="$g \in \mathbb{Z}$"/>

Output in the Rodin editor:

```
INVARIANTS
  inv1   :   m ∈ ℤ
  inv2   :   n ∈ ℤ
  inv3   :   g ∈ ℤ
```

Guards Generation

This is implemented by the method generateEventGuardXml(int guardNo, String guard): String. It is called in a for loop of the generateEventXml():String method which iterates each guard condition expression to generate the guards XML. Specifically, for every guard condition, the method removes all the *_prev* postfixes (used to denote the previous states of the variables) in the variables used in it first. Then the *guard name* (see the output XML) is set to GuardX and the *guard label* (see the output XML) is set to grdX, where X is the guard number (start from 1). Next, the *guard predicate* (see the output XML) is set to the preprocessed guard expression in the first step. Finally, the generated Guards XML string is returned by the update parameter *ret*. The pseudo code for the Guards Generation algorithm is as follows.

```
generateEventGuardsXml(ret,outputVars,guards)
    guardNo <- 1
    for(guard : guards)
        for(var : outputVars)
            replace all var_prev in guard with var
            append Guard name XML to ret
```

```
        append guardNo to ret
        append Guard label XML to ret
        append guardNo to ret
        append Guard predicate XML to ret
        append guard to ret
    guardNo <- guardNo + 1
```

To illustrate the Guards Generation process, we choose the Guards of $evt1\_2$ (the event that is generated from the conditions $m >= 2; n == 1$ and the result $g := m/n$) in Fig 3.2 as an example. A required input from tabular expression, an output in XML format to support the Rodin project, and finally output to appear on the Rodin editor after importing the generated code are given as follows:

Input from tabular expression:
conditions $m >= 2, n == 1$.

Output in XML format:

<org.eventb.core.guard name="Guard1" org.eventb.core.

label="grd1"org.eventb.core.predicate="(m ≥ 2)"/>

<org.eventb.core.guard name="Guard2" org.eventb.core.

label="grd2"org.eventb.core.predicate="(n = 1)"/>

Output in the Rodin editor:

```
WHEN
 grd1   :   (m ≥ 2)
 grd2   :   (n = 1)
```

Generate Actions

This is implemented by the method generateEventActionsXml(): String. Specifically, the method iterates every output variable to generate the actions XML. For each action, we have to judge whether it is a deterministic action or non-deterministic action. For deterministic action, a deterministic assignment ex-

pression is generated. A deterministic action can be defined as: outputVariable := value. For non-deterministic action, a non-deterministic assignment expression is generated. A non-deterministic action can be defined as: outputVariable :∈ {value_1, value_2,...value_k}. Then the *action name* (see the output XML) is set to ActionX, and the *action label* (see the output XML) is set to actX, where X is the action number (each action corresponds to one output variable). Finally, the *action assignment* (see the output XML) is set to the generated result in the first step, and the actions XML string is returned. The pseudo code for the Actions Generation algorithm is as follows.

```
generateEventActionsXml(ret,outputVars)
    actionNo <- 1
    operator <- null
    outputValue <- null
    for(var : outputVars)
        if (var has only one value)
            operator <- ':='
            outputValue <- value
        else (var has several values)
            operator <- ':∈'
            outputValue <- {value_1, ...value_k}
        append Action name XML to ret
        append actionNo to ret
        append Action assignment XML to ret
        append output Variable var.name to ret
        append operator to ret
        append outputValue to ret
        append Action label XML to ret
        append actionNo to ret
        actionNo <- actionNo + 1
```

To illustrate the Actions Generation process, we choose the Actions of evt1_2 (the event that is generated from the conditions $m >= 2; n == 1$ and

the result $g := m/n$) in Fig 3.2 as an example. A required input from tabular expression, an output in XML format to support the Rodin project, and finally output to appear on the Rodin editor after importing the generated code are given as follows:

Input from tabular expression:
result $g := m/n$.

Output in XML format:

<org.eventb.core.action name="Action1" org.eventb.core.

assignment="g := (m ÷ n)" org.eventb.core.label="act1"/>

Output in the Rodin editor:

```
THEN
 act1   :   g ≔ (m ÷ n)
```

Generate Events

This is implemented by the method generateEventXml(): String. This method is called while the hierarchicalGridEventBGenerator walks through the table using a depth-first search strategy. Specifically, in the no refinement mode, the generateEventXml() method will be immediately called to generate an event when the Generator reaches a leaf cell (the last column cell). The conditions on the path from the root cell (the first column cell) to the node cell are all converted to the guards of the events, and the only value corresponds to this path is converted to the action. In the with refinement mode, the generateEventXml() method is not necessarily called even a leaf cell is reached. Instead, if the current refinement is the last refinement, the event is directly generated just like the no refinement mode. But if the current refinement is not the last refinement, the Generator first collects all the values when it descends to the leaf cells. And it only generates an event when it ascends

back to the current refinement layer (i.e., when the cell number is equal to the current refinement number). The events guards are generated from the conditions from the root cell to current cell only and the actions are generated non-deterministically using all the collected leaves values. And finally the generated event XML string is returned. The pseudo code for the Events Generation algorithm is as follows.

```
generateEventXml()
    eventXml <- null
    eventXml += generateEventGuardsXml()
    eventXml += generateEventActionsXml()
    return eventXml
```

The Event Generation algorithm is called by the handleLeafCell() and the ascendFromGrid() methods in the table walk algorithm (implemented by the HandleLeftGrid method) as shown in the following:

```
HandleLeftGrid(grid, generator)
    cells = grid.getSubHiearchy()

        for (i = 0 to cells.size())
            cell = cells.get(i)
            if (cell.getSubHiearchy().isEmpty())
                outputCells <- null
                for (var : outputVariables)
                    add var outputCell info to outputCells
                generator.handleLeafCell(cell, outputCells)
            else
                generator.handleEdgeCell(cell)
                generator.descendIntoGridFromCell(cell)
                HandleLeftGrid(cell, generator)
                generator.ascendFromGrid()

/*************************************/
//m_eventsXml stores all the events XML
handleLeafCell(inputCell, outputCells)
    save inputCell
    save outputCells

    if (inputCell num <= refine layer num)
            m_eventsXml += generateEventXml()//generate leaf cell event XML

/*************************************/
ascendFromGrid()
    if (inputCell num == refine layer num)
        m_eventsXml += generateEventXml()//generate edge cell event XML
```

To illustrate the Events Generation process, we choose the event $evt1\_2$ (the event that is generated from the conditions $m >= 2; n == 1$ and the result $g := m/n$) in Fig 3.2 as an example. A required input from tabular expression, an output in XML format to support the Rodin project, and finally output to appear on the Rodin editor after importing the generated code are given as follows:

Input from tabular expression:

conditions $m >= 2, n == 1$ and the result $g := m/n$

Output in XML format:


```
<org.eventb.core.event name="Event1_2" org.eventb.core.convergence
="0" org.eventb.core.extended="false" org.eventb.core.label="evt1_2">
<org.eventb.core.guard name="Guard1" org.eventb.core.label="grd1"
org.eventb.core.predicate="(m ≥ 2)"/>
<org.eventb.core.guard name="Guard2" org.eventb.core.label="grd2"
org.eventb.core.predicate="(n = 1)"/>
<org.eventb.core.action name="Action1" org.eventb.core.assignment=
"g := (m ÷ n)" org.eventb.core.label="act1"/>
<org.eventb.core.refinesEvent name="evt1" org.eventb.core.target="evt1"/>
</org.eventb.core.event>
```


Output in the Rodin editor:

```
evt1_2   ≙
STATUS
 ordinary
REFINES
 evt1
WHEN
 grd1   :   (m ≥ 2)
 grd2   :   (n = 1)
THEN
 act1   :   g ≔ (m ÷ n)
END
```

Generate File Refinement Relation

This is implemented by the method generateFileRefineRelation(int refinerFileNo): String. But it is only called in the with refinement mode. The input parameter $refinerFileNo$ (count from 0) means the current Machine is generated from the first (refinerFileNo + 1) columns of the table. It refines the previous abstract Machine which is generated from the first refinerFileNo columns of the table. Specifically, the *refinesMachine name* (i.e., refiner Machine, see the output XML) in the refinement XML is set to the current Machine name, which is generated using the project name appending _M and appending the input refinerFileNo. Similarly, the *target* Machine (i.e., refined Machine) is set to the previous abstract Machine name, which is generated using the project name appending _M and appending the input refinerFileNo - 1. And finally the generated refinement relation XML string is returned.

To illustrate the File Refinement Relation process, we choose the table in Fig 3.2 as an example. The abstract Machine *g_M0* is generated from the conditions in the first column and all the results values of the table. The concrete Machine *g_M1* is generated from the conditions in the first column and the second column as well as all the results values of the table. The Concrete Machine *g_M1* refines the abstract Machine *g_M0*. A required input from tabular expression, an output in XML format to support the Rodin project, and finally output to appear on the Rodin editor after importing the generated code are given as follows:

Input from tabular expression:
refinerFileNo, projectName

Output in XML format:

```
<org.eventb.core.refinesMachine name="g_M1" org.eventb.core.target="g_M0"/>
```

Output in the Rodin editor:

```
MACHINE
  g_M1
REFINES
  g_M0
```

Generate "Sees" Relation

The "Sees" relation is a relation between a Machine and a Context. As mentioned in the background chapter. A Machine can "Sees" one or several other Contexts. This is implemented by the method generateMachineSeeContextRelationXml(): String. Specifically, both the *seesContext name* and the *target* (see the output XML) are set to the Context name which is generated using the project name appending _C. The generated XML for the Machine Context relation is returned in the end.

To illustrate the "Sees" Relation Generation process between Machine and Context, we choose the table in Fig 3.2 as an example. The Machine *g_ M0* "Sees" the Context *g_ C*. A required input from tabular expression, an output in XML format to support the Rodin project, and finally output to appear on the Rodin editor after importing the generated code are given as follows:

Input from tabular expression:
projectName

Output in XML format:

```
<org.eventb.core.seesContext name="g_C" org.eventb.core.target="g_C"/>
```

Output in the Rodin editor:

```
SEES
  g_C
```

Write XML to Machine File

All the generated XML strings of the Machine components are written into a Machine file in the project folder.

3.8.5 Supported Operators

A set of symbols supported by TX2EB is given in Table 3.4, which clearly illustrates the mapping relation that we used to translate the Tabular Expression symbols into Event-B symbols.

All those operators are all already supported in Matlab GUI for PVS and CVC3 in the TET tool. Also there is an ExpressionGenerator interface containing two methods GenerateUnaryOperation and GenerateBinaryOperation in the jTET tool [5]. EventB Generator only has to implement this interface and override those two methods so that all the operators acquired from the Matlab GUI can be transformed into the desired EventB operators.

Tabular Expression Operators	**Event-B Operatiors**
$+$	$+$
$-$	$-$
$*$	$*$
$/$	$\div$
$\sim=$	$\neq$
$==$	$=$
$>$	$>$
$>=$	$\geq$
$<$	$<$
$<=$	$\leq$
$\sim$	$\neg$
$\&\&$	$\wedge$
$\|\|$	$\vee$

Table 3.4: Tabular Expression and Event-B Operators Mapping

3.8.6 SubType Predicate

Our tool can also support the subtype predicate. For instance, $z : \{y : real \mid y > 0\}$ is used to express z is a positive real.

Subtype predicate can be described by its type declarations and subtype declarations. This feature is implemented by adding condition judgments in two process: the Invariant generation process and the initialization event action generation process. If a variable has no subtype predicate, we just generate the variable type as usual. Otherwise, we have to generate both the variable type (like the $y : real$ in the example) and its subtype (like the $y > 0$ in the example).

3.8.7 Complicated Expressions

The tool supports complicated expressions, like (((x * 5) = (1 ÷ (- 4))) ∨ (x > (8 + 3))), both in the condition grid and the result grid.

The Matlab parser (internally uses ANTLR grammar) to generate the abstract syntax tree (AST) for CVC3 Generator are already implemented in jTET [5] . In the AST, the unary and binary expression can nest as much as a user wants and the tree is eventually terminated by some leaf elements (variables or literals). Then the CVC3 Generator uses the generated AST to make the CVC3 expression. This also applies to our Event-B Generator. Therefore, we just reused the AST to generate the Event-B expression in our EventBExpressionGenerator.

3.8.8 Complex & Simplified Bool Expression

Our Tool supports both the complex bool expression (explicitly compare a bool variable with $TRUE$ or $FALSE$) and simplified bool expression (do not need to explicitly compare it with $TRUE$ or $FALSE$, bool variables stand alone).

This feature is necessary because the Rodin Platform can only support complex bool expression form like ($P = TRUE$, $P = FALSE$, $P \neq TRUE$,

and $P \neq FALSE$) as a predicate, and it does not support the simplified single bool variable form like P. But Matlab can support both two forms. Therefore, no matter what kind of bool expression is used by the user in Matlab GUI, the tool must recognize it automatically and translate it into a proper Event-B expression. It is implemented by substituting the variable P with ($P = TRUE$) when P is a bool variable acquired from a condition cell expression, which does not contain equal or not equal operators. Please note that we don't support combined usage of the two forms.

3.9 Limitations

Our developed tool has some limitations, which are listed as follows:

- The tool only supports the Horizontal Condition Table (HCT).
- The quantifiers ($\forall$ and $\exists$) are not supported in the current tool.
- The TET tool does *not* support states, we have to use reserved pattern `x_prev` to mimic/simulate the previous state of variable `x`. Therefore, variables ending with `_prev` cannot be used as regular variables.
- The TET tool was designed for the single table. In order to generate an Event-B model, we need to save all the tables into a folder that simulates the idea for handling multiple tables. But doing so, the relation among tables may be lost.
- User defined enumeration type has to be defined in Matlab code before using it in the tool GUI.

3.10 User Manual

In order to generate an Event-B model from tabular expressions using the TX2EB tool, the user has to follow the steps below.

- Copy the compiled TX2EB tool back-end Java code package jtet-1.0-SNAPSHOT-jar-with-dependencies.jar to the jTET folder of the TET front-end as shown in Fig. 3.11.

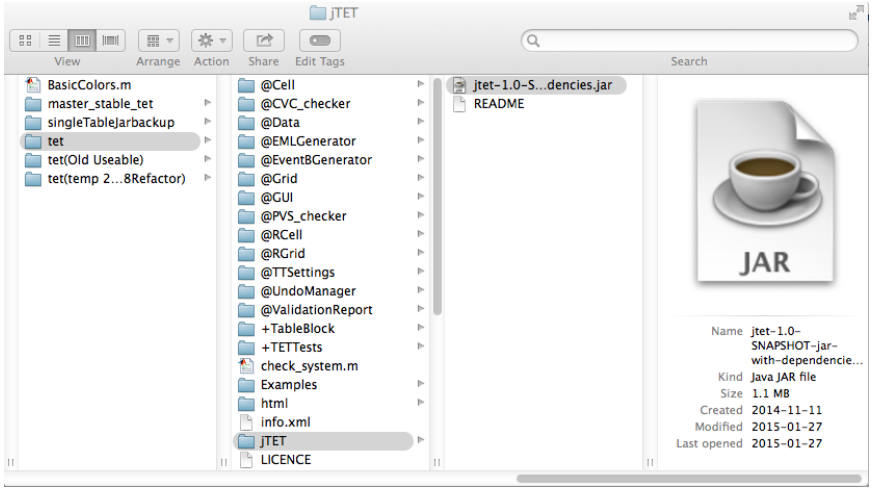


Figure 3.11: Copy Java jar to TET folder

- Start the Matlab application.

- Double click on the TableToolMatlab.m file under the TET root directory to open it in Matlab as shown in Fig. 3.12.

- Click on the *Run* button in the Matlab GUI to start our tool. If Matlab pops up a window displaying a message that the file TableToolMatlab.m is not found in the current folder or on the Matlab path, just click the "Add to path" button to add the file to the current Matlab path as shown in Fig. 3.13.

- Fig. 3.14 shows a started editor window of the tool. A user can input / modify their requirements in this GUI.

  The input text box accepts the input variables of the requirements. The default variable type is real. If the user wants to create some other variable type, then it can be entered as variable name : variable type. For example, b:bool, color:Color (enumeration type). An expression name of

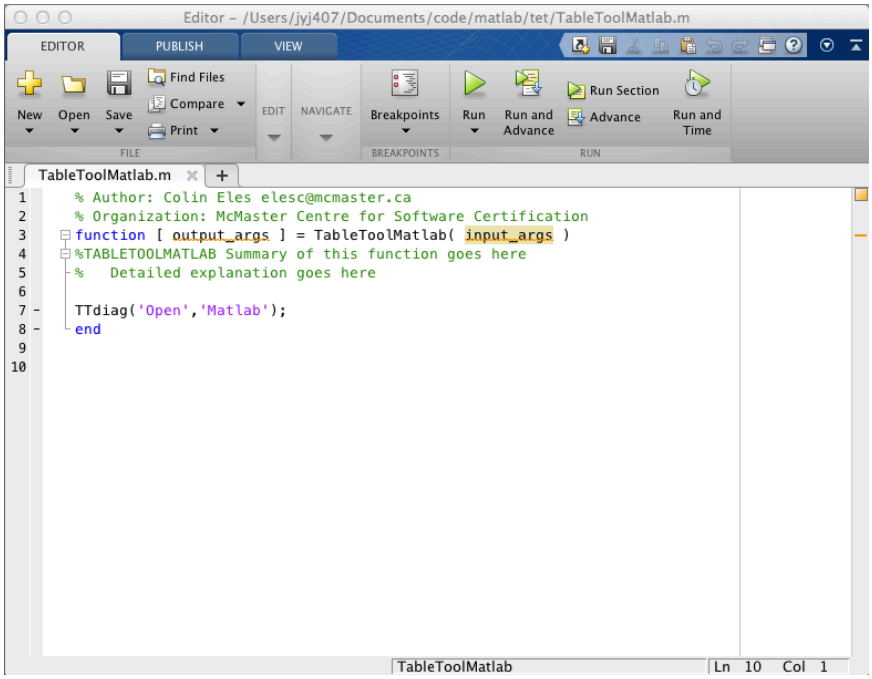


Figure 3.12: Open the TableToolMatlab.m file

Figure 3.13: Add TableToolMatlab.m to Matlab Path

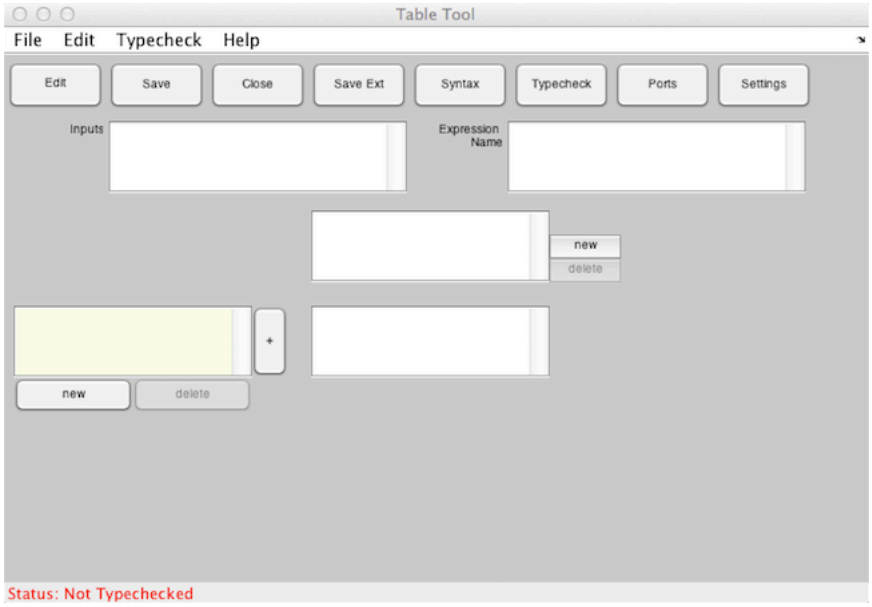


Figure 3.14: Start an Editor Window of the Tool

the table (i.e., table name) can be defined in the Expression Name text box. The leftmost grid is called conditional grid. The user can use the new button under it to expand it and use the delete button to shrink it. The addition symbol button at the end of the conditional grid is used to create sub grids of the current grid. The text box immediately under the Inputs and Expression Name text box is called top grid. The output variable name is given here. The user can also use the new and delete button after the top grid to add and remove output variables. The grid to the right of conditional grid and under the bottom of the top grid is called output grid. The output variables value is given there. If the user wants to use multiple outputs, he or she should set the output mode to multiple outputs at menu "Edit —> Output Mode —> Multiple Outputs" as shown in Fig. 3.15.

- A user can check the disjointness and completeness of the requirement by following the menu "Typecheck —> Typecheck". But make sure you have installed a CVC3 or PVS prover before type checking. The default

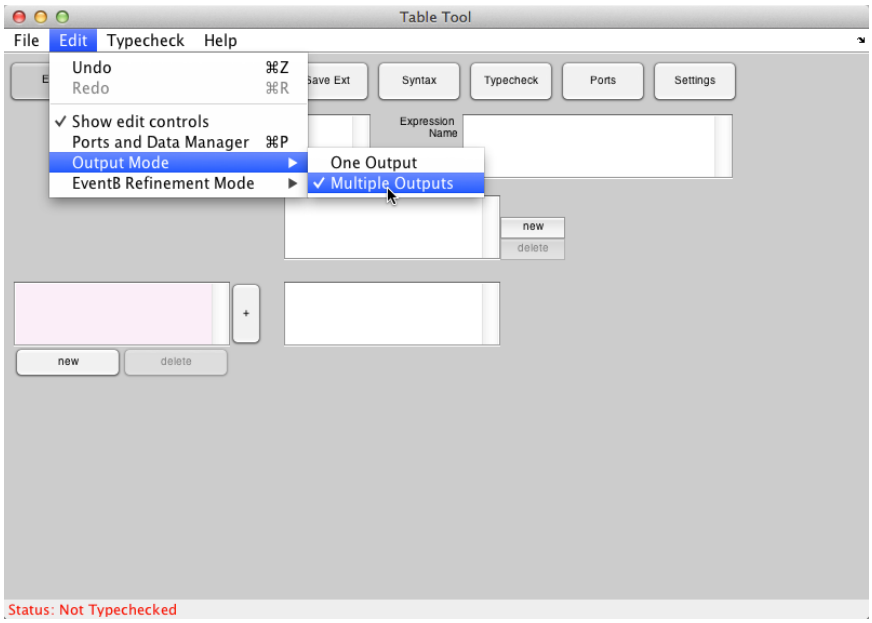


Figure 3.15: Set the Output Mode

Typecheck prover is the CVC3 prover. You can change the prover using menu command "Typecheck —> Default Prover". If the table is not checked, a counterexample will be generated which gives the reason why the table is not disjoint or complete. If the created table is both disjoint and complete, then the tool will pop up a window to tell the user the table is typechecked. For example, in Fig. 3.16, the tool generates a counterexample x = 0 since this situation is not covered in the table. After fixing this problem, the table is checked as shown in Fig. 3.17.

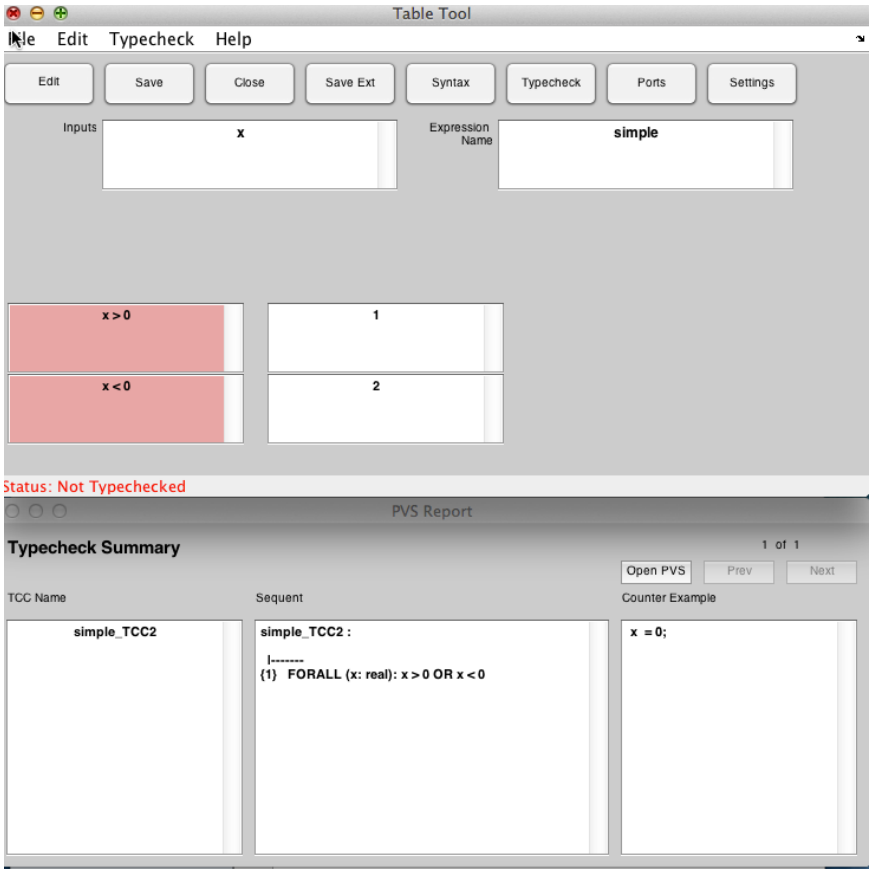


Figure 3.16: Not Checked Table Example

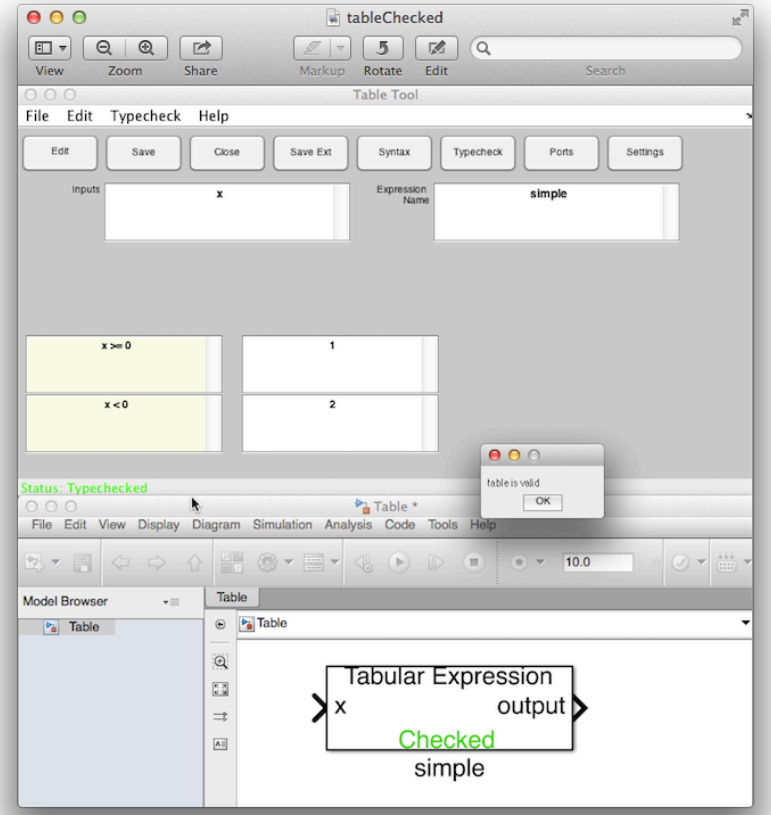


Figure 3.17: Checked Table Example

- The next step is to generate the Event-B model. Before generating the model, the user has to set the refinement mode. As described in former section of this chapter, there are two refinement modes, with refinement and no refinement. They can be set at menu "Edit —> EventB Refinement Mode —> No Refinement / With Refinement" as shown in Fig. 3.18.

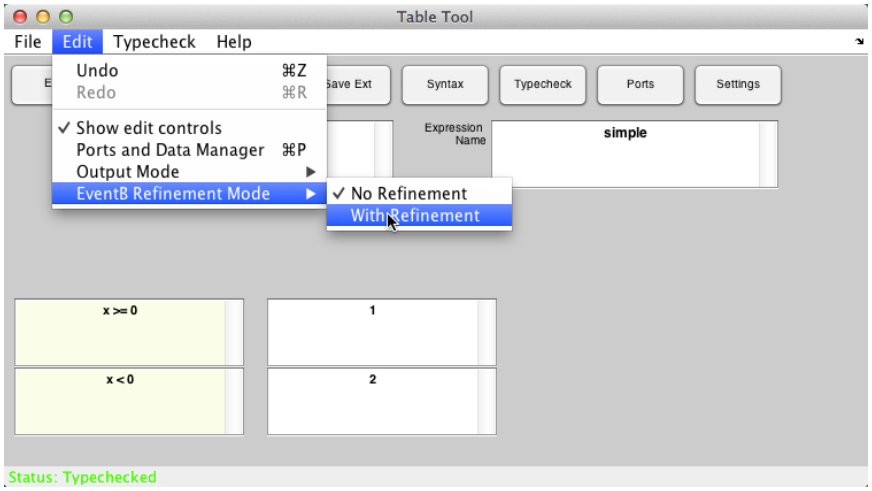


Figure 3.18: Set Refinement Mode

After setting the refinement mode, the user has to save the current table to a folder at menu "File —> Save to Table-File". The user can choose a folder to save the table file. If the requirements are made of more than one tables, the user just needs to create the tables one by one and then saves them to the same folder. Please note that the user can not save other non table files (*.table files) to the designated folder. The next step is to generate the Event-B Model. Click "Typecheck —> EventB Generate Project" on the GUI. And choose the folder that contains the tables saved in the previous steps using popped up window "Select Directory to Open" to generate the Event-B Model as shown in Fig. 3.19.

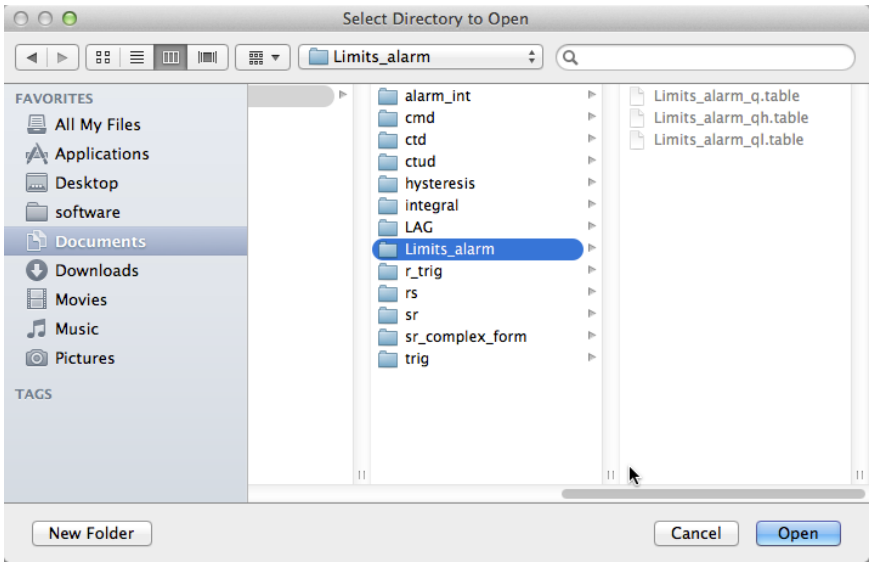


Figure 3.19: Generate EventB Model

- The generated Event-B model in the form of a Rodin project is saved in the current Matlab folder. For the no refinement model, it is saved in the NR (No Refinement) subfolder. For the with refinement model, it is saved in the WR (With Refinement) subfolder.

  To import the generated project into the Rodin platform, launch the Rodin Platform first. Then choose a menu "File —> Import" command as shown in Fig. 3.20.

  By default, the Rodin Platform generates some POs. They are usually automatically discharged if the system is correctly designed. If the POs cannot be discharged because the system is not properly designed, the user has to go back to edit the requirement and redo the above steps until all the POs are discharged. Sometimes, even the system is correctly designed and specified, the Rodin Platform still can not discharge it. In that case, the user has to try different prover to prove it or check the failure reason and try to discharge it manually.

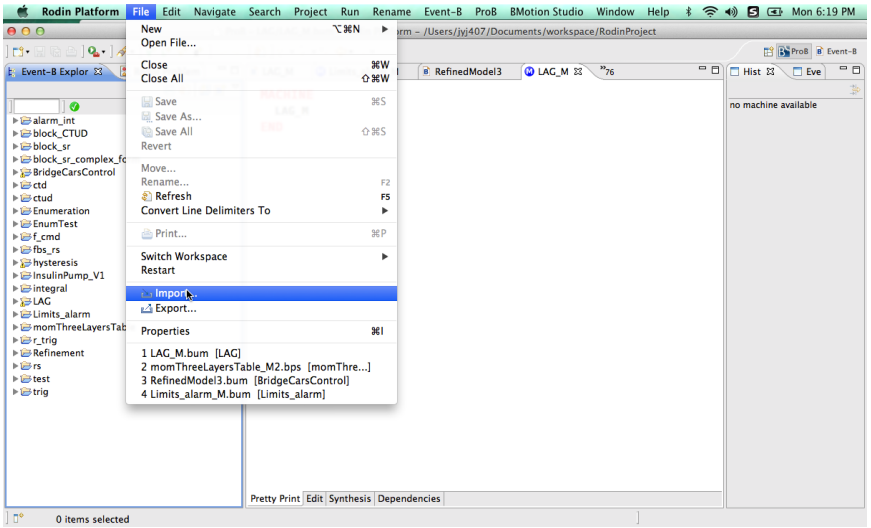


Figure 3.20: Import Generated Rodin Project into the Rodin Platform

Chapter 4

Case Study - IEC61131-3 Function Blocks

This chapter main talks about the function blocks of the IEC61131-3 standard that we have implemented using our TX2EB tool. We also prove the refinement relation between the implementation of a function block (*limit_ alarm*) and its requirement in section 4.5.

4.1 IEC61131-3 Function Blocks

IEC stands for the International Electrotechnical Commission which enacts and releases International Standards for all electrical, electronic and related technologies [11, 22]. The IEC61131-3 is one of the standards of IEC. It was enacted to unify the syntax and semantics of programming languages for programmable logic controller (PLC), which is a digital computer widely used for automation in real-time and embedded control systems.

Pang et al. [22] proposed a method to check the correctness of IEC61131-3 function blocks (FBs) and composition of FBs using PVS theorem prover. In our work, we have also used some FBs from IEC61131-3 standard to check the correctness of FBs using tabular expression and Event-B prover. Specifically, we first formalize the FBs using the tabular expressions tool. And then we

verify the disjointness and completeness of the FBs using CVC3 and PVS provers. Finally, we generate Event-B models from the tabular expressions and use Rodin provers for mathematical reasoning and consistency checking. We have selected only simple FBs, which do not require functional composition and quantifiers for defining functional behavior. Moreover, we also try to prove the refinement relation between system requirements and its implementation, in which both the system requirements and implementation are described using tabular expressions.

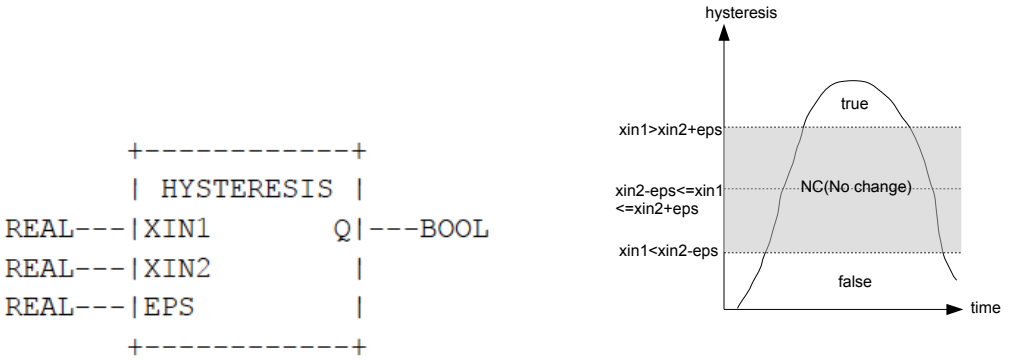
We have implemented twelve function blocks of the IEC61131-3 standard. They are *alrm_int, block_sr, ctd, ctud, f_cmd, trig, r_trig, fbs_rs, hysteresis, integral, LAG* and *limits_alarm*. All these function blocks are successfully translated to Event-B models using our tool. And the generated POs in Rodin Platform are also discharged by the Rodin prover.

The rest of this chapter is organized as follows. In Section 4.2, we describe a *hysteresis* FB for generating the Event-B model without (no) refinement. In Section 4.3, we use another FB for generating the Event-B models with refinement. In Section 4.4, we give an example of tables composition: the *alrm_int* Function Block. Finally, we present a case study: *limit_alarm* in Section 4.5. In this example, we generate models from both the abstract requirements of the FB and the concrete implementation of it. Then we import both the concrete and the abstract Event-B models into a single Rodin project. And then we provide a refinement relation between abstract and concrete models manually. Rodin prover automatically discharged all the generated proof obligations (POs). That proves the concrete model is a refinement of the abstract model. This demonstrates the real power of our tool.

4.2 No refinement Example: Hysteresis Function Block

A *hysteresis* function block is taken to illustrate the transformation process of the no refinement mode. This function block has only one block: *hysteresis*.

Its declaration, functional behavior, and tabular expression are depicted in Fig. 4.1.



Condition	*Result* q
xin1 $<$ (xin2 $-$ eps)	False
(xin2 $-$ eps) $\leq$ xin1 $\leq$ (xin2 $+$ eps)	NC
xin1 $>$ (xin2 $+$ eps)	True

assume: $eps > 0$

Figure 4.1: *Hysteresis* Functional Block

We create an equivalent tabular expression as above using our tool (see Fig. 4.2). And then we use our tool to check the properties of disjointness and completeness. The *status:TypeChecked* on the left bottom of Fig. 4.2 indicates that the table has passed the typechecking. Therefore, the table is both disjoint and complete. Finally, we generate the corresponding EventB model (see Fig. 4.3) from the TypeChecked tabular expression.

Figure 4.2: *Hysteresis* Tabular Expression

```
MACHINE
  hysteresis_M
SEES
  hysteresis_C
VARIABLES
  xin1
  xin2
  eps
  q
INVARIANTS
  inv1   :   xin1 ∈ ℤ
  inv2   :   xin2 ∈ ℤ
  inv3   :   eps ∈ {eps|eps ∈ ℤ ∧ (eps > 0)}
  inv4   :   q ∈ BOOL
EVENTS
  INITIALISATION   ≙
  STATUS
   ordinary
  BEGIN
   act1   :   xin1 :∈ ℤ
   act2   :   xin2 :∈ ℤ
   act3   :   eps :∈ {eps|eps ∈ ℤ ∧ (eps > 0)}
   act4   :   q :∈ BOOL
  END

  evt1   ≙
  STATUS
   ordinary
  WHEN
   grd1   :   (xin1 < (xin2 − eps))
  THEN
   act1   :   q ≔ FALSE
  END

  evt2   ≙
  STATUS
   ordinary
  WHEN
   grd1   :   (((xin2 − eps) ≤ xin1) ∧ (xin1 ≤ (xin2 + eps)))
  THEN
   act1   :   q ≔ q
  END

  evt3   ≙
  STATUS
   ordinary
  WHEN
   grd1   :   (xin1 > (xin2 + eps))
  THEN
   act1   :   q ≔ TRUE
  END

END
```


Figure 4.3: hysteresis EventB Model

As we can see from Fig. 4.3, only one model can be generated in the *no refinement* mode. Because there are three rows in the table, so three corresponding events are generated in the model. For example, $evt3$ is generated from the third row in the table. Because the condition in the third row is $xin1 > xin2 + eps$, so the guard for $evt3$ is $grd1 : (xin1 > (xin2 + eps))$. And similarly, because the result in the third row is $true$ and the output variable is q, so the action for $evt3$ is assigning the value $true$ to output variable q, which is $q := TRUE$.

4.3 With Refinement Example: ctd_cv Function Block

A *ctd_cv* function block is taken to illustrate the transformation process of the with refinement mode. Basically, this function block takes two Bool input variables ld and cd and two Int input variables pv and $PVmin$ and the value of output variable cv is determined based on the four input variables. Detailed functional behavior is described in the tabular expression of Fig. 4.4. We can see that the *ctd_cv* FB has three columns. Therefore, the tool generates three different models for it. The first abstract model is *ctd_cv_M0*, the second abstract model is *ctd_cv_M1*, and the concrete model is *ctd_cv_M2*. The first abstract model *ctd_cv_M0* is generated from the conditions in the first column and the results in the last column. The second abstract model *ctd_cv_M1* is generated from conditions in the first two columns and the results in the last column. The concrete model *ctd_cv_M2* is generated from all the conditions in the three columns and the results in the last column. The concrete model *ctd_cv_M2* refines the second abstract model *ctd_cv_M1*, and *ctd_cv_M1* refines the first abstract model *ctd_cv_M0*. For the first abstract model, the tool generates two events: $evt1$ and $evt2$. For example, the $evt2$ is generated from the left red box of Fig. 4.4 and its corresponding results. The first column condition ld is converted to the only one guard in the $evt2$. But there are three values: $\{cv\_prev,\ cv\_prev - 1,\ cv\_prev\}$ in

the action corresponding to this guard, which means the value of the output variable *cv* can be any of them, this is called the non deterministic value assignment. For the second abstract model, the tool generates 3 events ($evt1$, $evt2\_1$, $evt2\_2$) from the table. For example, the $evt2\_1$ is generated from the blue box of Fig. 4.4. It is named $evt2\_1$ because it is generated from the 2nd grid and the 1st sub grid of the 2nd grid. This event refines the $evt2$ in the first abstract model. Actually, it refines $evt2$ by introducing one more guard condition: $\sim cd$ to $evt2$ so that the output variable value for the action can be set deterministically. The concrete model *ctd_cv_M2* refines the second abstract model *ctd_cv_M1*. For instance, the the abstract event $evt2\_2$ in the second abstract model is further split into two concrete events: $evt2\_2\_1$ and $evt2\_2\_2$. They both refine the abstract event $evt2\_2$.

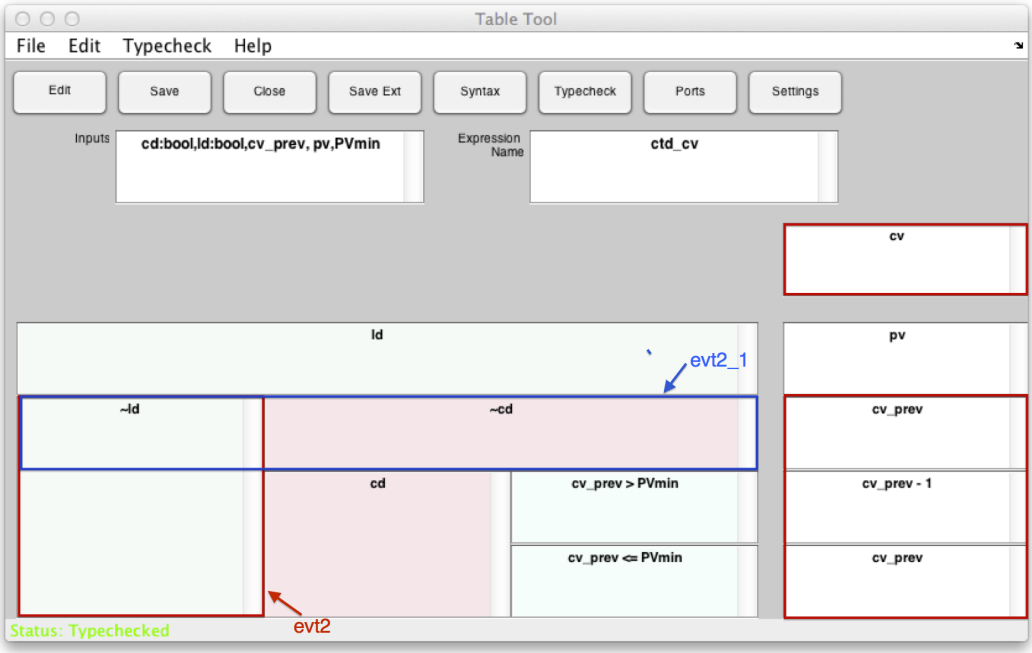


Figure 4.4: ctd_cv Tabular Expression

The generated three models: *ctd_cv_M0*, *ctd_cv_M1*, and *ctd_cv_M2* are illustrated in the following Fig. 4.5, Fig. 4.6, and Fig. 4.7.

```
MACHINE
  ctd_cv_M0
SEES
  ctd_cv_C
VARIABLES
  cd
  ld
  pv
  PVmin
  cv
INVARIANTS
  inv1   :   cd ∈ BOOL
  inv2   :   ld ∈ BOOL
  inv3   :   pv ∈ ℤ
  inv4   :   PVmin ∈ ℤ
  inv5   :   cv ∈ ℤ
EVENTS
  INITIALISATION   ≙
  STATUS
   ordinary
  BEGIN
   act1   :   cd :∈ BOOL
   act2   :   ld :∈ BOOL
   act3   :   pv :∈ ℤ
   act4   :   PVmin :∈ ℤ
   act5   :   cv :∈ ℤ
  END

  evt1   ≙
  STATUS
   ordinary
  WHEN
   grd1   :   (ld = TRUE)
  THEN
   act1   :   cv ≔ pv
  END

  evt2   ≙
  STATUS
   ordinary
  WHEN
   grd1   :   (¬ (ld = TRUE))
  THEN
   act1   :   cv :∈ {cv,(cv − 1),cv}
  END

END
```

Figure 4.5: Event-B Model of ctd_cv_M0

```
MACHINE
  ctd_cv_M1
REFINES
  ctd_cv_M0
SEES
  ctd_cv_C
VARIABLES
  cd
  ld
  pv
  PVmin
  cv
INVARIANTS
  inv1   :   cd ∈ BOOL
  inv2   :   ld ∈ BOOL
  inv3   :   pv ∈ ℤ
  inv4   :   PVmin ∈ ℤ
  inv5   :   cv ∈ ℤ
EVENTS
  INITIALISATION   ≙
  STATUS
   ordinary
  BEGIN
   act1   :   cd :∈ BOOL
   act2   :   ld :∈ BOOL
   act3   :   pv :∈ ℤ
   act4   :   PVmin :∈ ℤ
   act5   :   cv :∈ ℤ
  END

  evt1   ≙
  STATUS
   ordinary
  REFINES
   evt1
  WHEN
   grd1   :   (ld = TRUE)
  THEN
   act1   :   cv ≔ pv
  END

  evt2_1   ≙
  STATUS
   ordinary
  REFINES
   evt2
  WHEN
   grd1   :   (¬ (ld = TRUE))
   grd2   :   (¬ (cd = TRUE))
  THEN
   act1   :   cv ≔ cv
  END

  evt2_2   ≙
  STATUS
   ordinary
  REFINES
   evt2
  WHEN
   grd1   :   (¬ (ld = TRUE))
   grd2   :   (cd = TRUE)
  THEN
   act1   :   cv :∈ {(cv − 1),cv}
  END

END
```

Figure 4.6: Event-B Model of ctd_cv_M1

```
MACHINE
  ctd_cv_M2
REFINES
  ctd_cv_M1
SEES
  ctd_cv_C
VARIABLES
  cd
  ld
  pv
  PVmin
  cv
INVARIANTS
  inv1   :   cd ∈ BOOL
  inv2   :   ld ∈ BOOL
  inv3   :   pv ∈ ℤ
  inv4   :   PVmin ∈ ℤ
  inv5   :   cv ∈ ℤ
EVENTS
  INITIALISATION   ≙
  STATUS
   ordinary
  BEGIN
   act1   :   cd :∈ BOOL
   act2   :   ld :∈ BOOL
   act3   :   pv :∈ ℤ
   act4   :   PVmin :∈ ℤ
   act5   :   cv :∈ ℤ
  END

  evt1   ≙
  STATUS
   ordinary
  REFINES
   evt1
  WHEN
   grd1   :   (ld = TRUE)
  THEN
   act1   :   cv ≔ pv
  END

  evt2_1   ≙
  STATUS
   ordinary
  REFINES
   evt2_1
  WHEN
   grd1   :   (¬ (ld = TRUE))
   grd2   :   (¬ (cd = TRUE))
  THEN
   act1   :   cv ≔ cv
  END

  evt2_2_1   ≙
  STATUS
   ordinary
  REFINES
   evt2_2
  WHEN
   grd1   :   (¬ (ld = TRUE))
   grd2   :   (cd = TRUE)
   grd3   :   (cv > PVmin)
  THEN
   act1   :   cv ≔ (cv − 1)
  END

  evt2_2_2   ≙
  STATUS
   ordinary
  REFINES
   evt2_2
  WHEN
   grd1   :   (¬ (ld = TRUE))
   grd2   :   (cd = TRUE)
   grd3   :   (cv ≤ PVmin)
  THEN
   act1   :   cv ≔ cv
  END

END
```

Figure 4.7: Event-B Model of ctd_cv_M2

4.4 Tables Composition Example: *alrm_int* Function Block

This function block has three parts (sub function blocks): *fbs_hi*, *fbs_lo* and *fbs_alrm_int*. The tabular expressions of the three blocks are given in the following Fig. 4.8, Fig. 4.9, and Fig. 4.10. The output of the first two sub blocks *fbs_hi* and *fbs_lo* is the input of last *fbs_alrm_int* sub block. This function block is taken as an example for generating the Event-B model from multiple tables. Or put another way, we are dealing with tables composition. The Event-B model is generated using all the information from the three tables. Specifically, the Event-B model variables and invariants are generated from all the variables in the three tables (if one variable is used in several tables, we only define it once in the Event-B model). The Machine events are generated by sequentially generating the events in one table and then another table. This process is repeated until the last table events are generated. All these events are put into the final composed Event-B model and the event number is encoded globally. For example, in the *alrm_int* function block, the tool generates two events from the *fbs_hi* function block, two events from the *fbs_lo* function block, and four events from the *fbs_alrm_int* function block. Therefore, the eventually generated Event-B model contains eight events: $evt1, evt2, ..., evt8$. Where $evt1, evt2$ are from the *fbs_hi* function block, $evt3, evt4, evt5, evt6$ are from the *fbs_alrm_int* function block, and $evt7, evt8$ are from the *fbs_lo* function block (see Fig. 4.11).

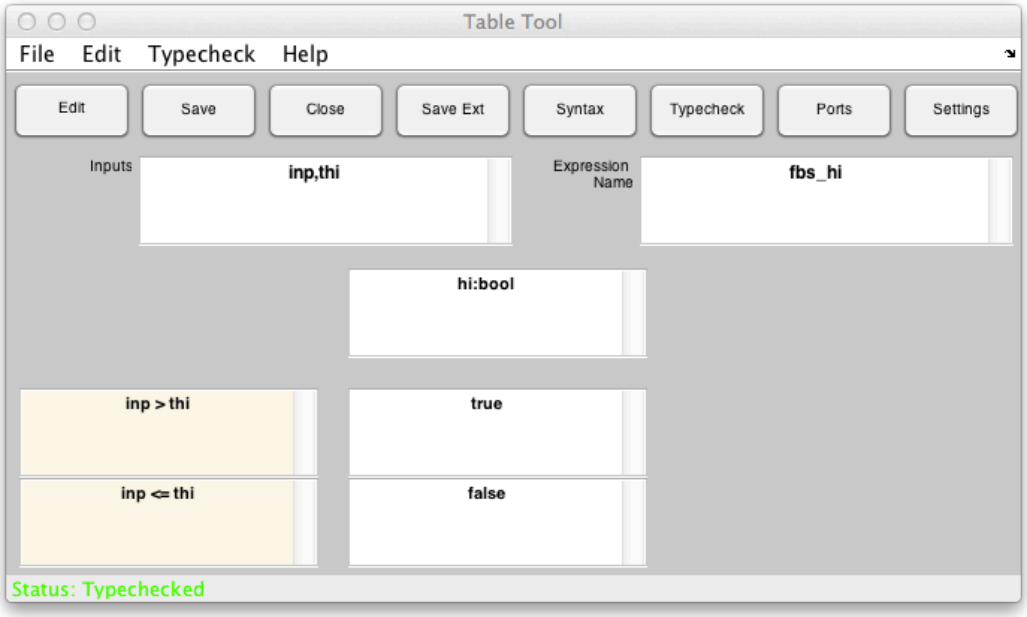


Figure 4.8: Tabular Expression of *fbs_hi*

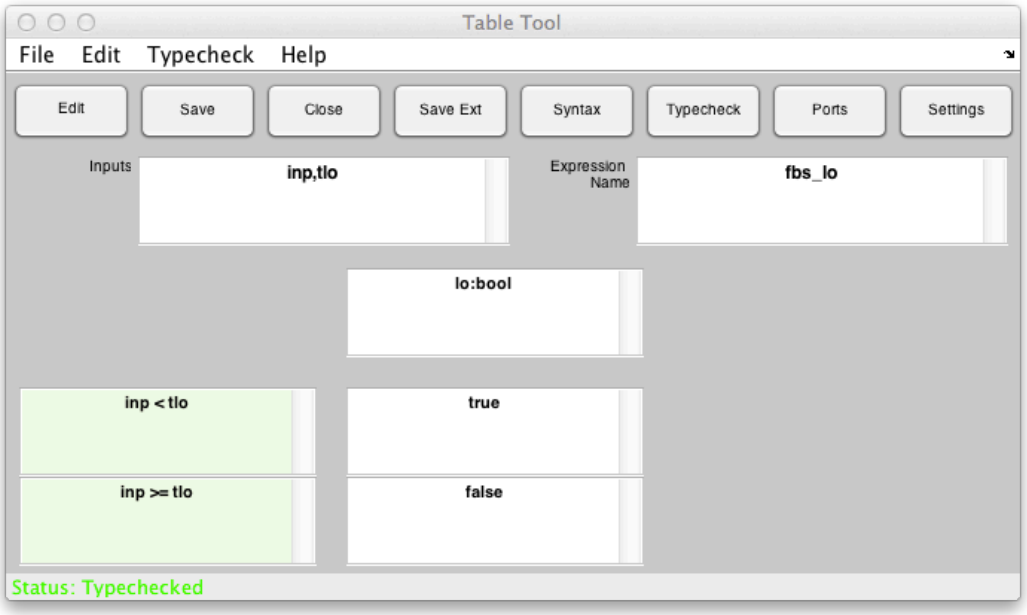


Figure 4.9: Tabular Expression of *fbs_lo*

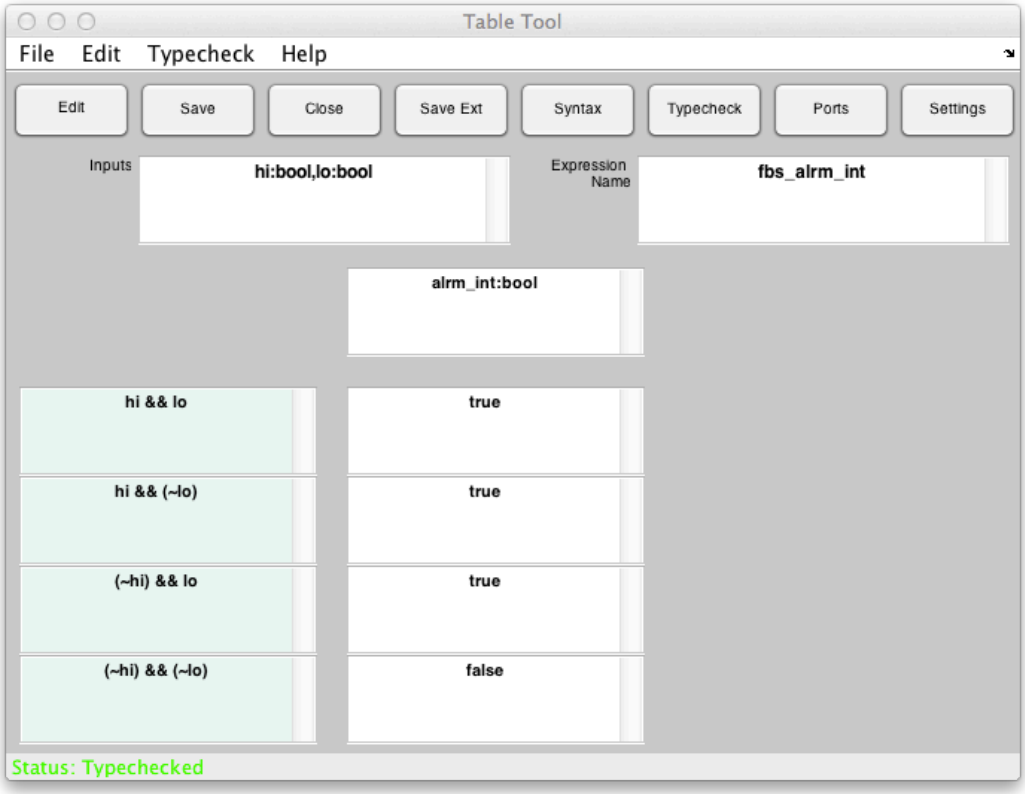


Figure 4.10: Tabular Expression of *fbs_alrm_int*

```
MACHINE
  alarm_int_M
SEES
  alarm_int_C
VARIABLES
  hi
  lo
  alrm_int
  inp
  thi
  tlo
INVARIANTS
  inv1   :   hi ∈ BOOL
  inv2   :   lo ∈ BOOL
  inv3   :   alrm_int ∈ BOOL
  inv4   :   inp ∈ ℤ
  inv5   :   thi ∈ ℤ
  inv6   :   tlo ∈ ℤ
EVENTS
  INITIALISATION   ≙
  STATUS
   ordinary
  BEGIN
   act1   :   hi :∈ BOOL
   act2   :   lo :∈ BOOL
   act3   :   alrm_int :∈ BOOL
   act4   :   inp :∈ ℤ
   act5   :   thi :∈ ℤ
   act6   :   tlo :∈ ℤ
  END

  evt1   ≙
  STATUS
   ordinary
  WHEN
   grd1   :   ((hi = TRUE) ∧ (lo = TRUE))
  THEN
   act1   :   alrm_int ≔ TRUE
  END

  evt2   ≙
  STATUS
   ordinary
  WHEN
   grd1   :   ((hi = TRUE) ∧ (¬ (lo = TRUE)))
  THEN
   act1   :   alrm_int ≔ TRUE
  END

  evt3   ≙
  STATUS
   ordinary
  WHEN
   grd1   :   ((¬ (hi = TRUE)) ∧ (lo = TRUE))
  THEN
   act1   :   alrm_int ≔ TRUE
  END

  evt4   ≙
  STATUS
   ordinary
  WHEN
   grd1   :   ((¬ (hi = TRUE)) ∧ (¬ (lo = TRUE)))
  THEN
   act1   :   alrm_int ≔ FALSE
  END

  evt5   ≙
  STATUS
   ordinary
  WHEN
   grd1   :   (inp > thi)
  THEN
   act1   :   hi ≔ TRUE
  END

  evt6   ≙
  STATUS
   ordinary
  WHEN
   grd1   :   (inp ≤ thi)
  THEN
   act1   :   hi ≔ FALSE
  END

  evt7   ≙
  STATUS
   ordinary
  WHEN
   grd1   :   (inp < tlo)
  THEN
   act1   :   lo ≔ TRUE
  END

  evt8   ≙
  STATUS
   ordinary
  WHEN
   grd1   :   (inp ≥ tlo)
  THEN
   act1   :   lo ≔ FALSE
  END

END
```

Figure 4.11: alrm_int EventB Model

4.5 Prove Refinement Relation between Implementation and Requirements

In this section, we use the *limits_alarm* to prove the refinement relation between implementation of *limits_alarm* and its requirements.

4.5.1 Requirements of the *limits_alarm* (Abstract Model)

The requirements of the function block (FB) *limits_alarm* consists of three parts: *limits_alarm_qh*, *limits_alarm_ql* and *limits_alarm_q* [21, 22, 11]. The output of the first two blocks *limits_alarm_qh* and *limits_alarm_ql* is the input of last block *limits_alarm_q*. Its functional behavior is illustrated in Fig. 4.12 [22].

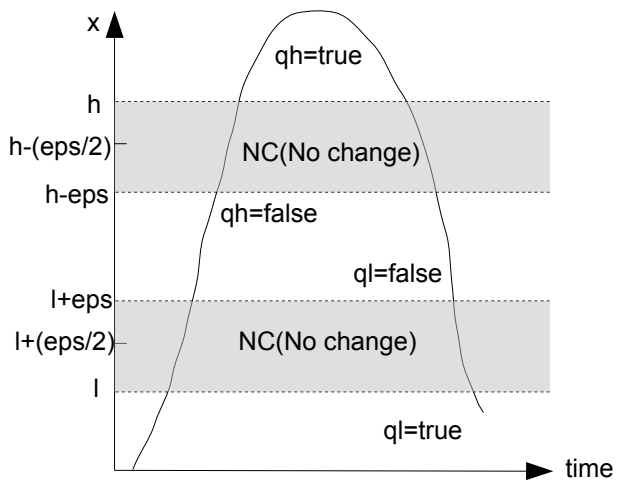


Figure 4.12: *limits_alarm* Functional Behavior

The function behavior of *limits_alarm* can also be described in the following tabular expressions as shown in Fig. 4.13 [22].

In order to generate the Event-B models from tabular expressions, we create the equivalent tabular expressions using our tool (see Fig. 4.14, Fig. 4.15, and

Condition	*Result* q
qh ∨ ql	True
¬(qh ∨ ql)	False

Condition	*Result* qh
x > h	True
h - eps ≤ x ≤ h	NC
x < h − eps	False

assume: $eps \geq 0$

Condition	*Result* ql
x < l	True
l ≤ x ≤ l + eps	NC
x > l + eps	False

assume: $eps \geq 0$

Figure 4.13: *Limits_ alarm* Requirement in Tabular Expressions

Fig. 4.16).

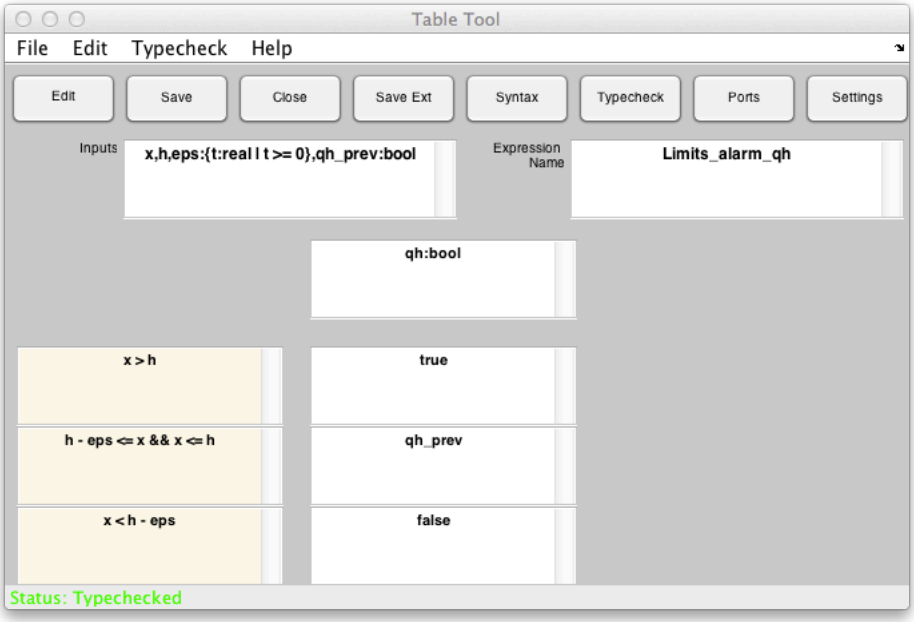


Figure 4.14: *Limits_ alarm_ qh* Tabular Expression

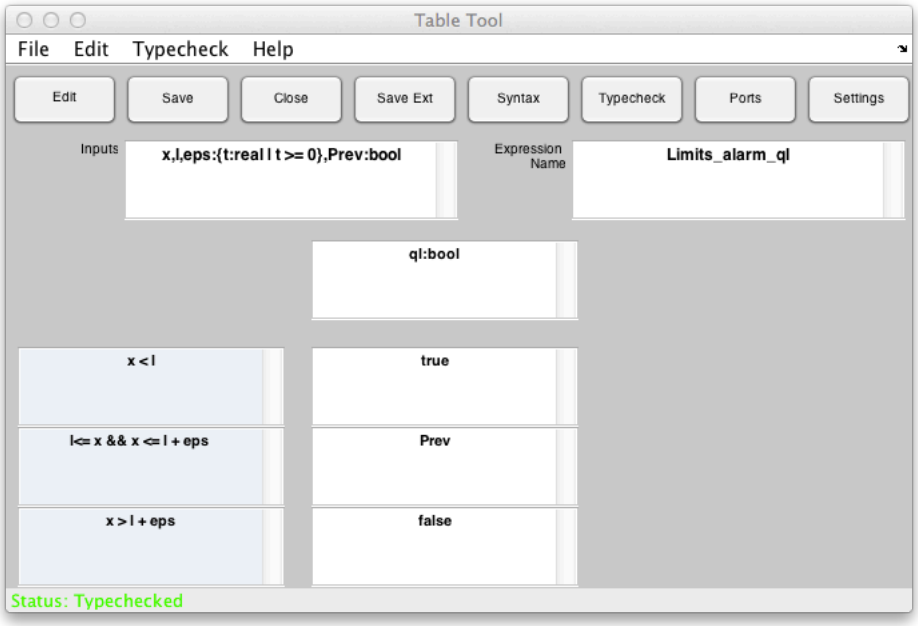


Figure 4.15: *Limits_ alarm_ ql* Tabular Expression

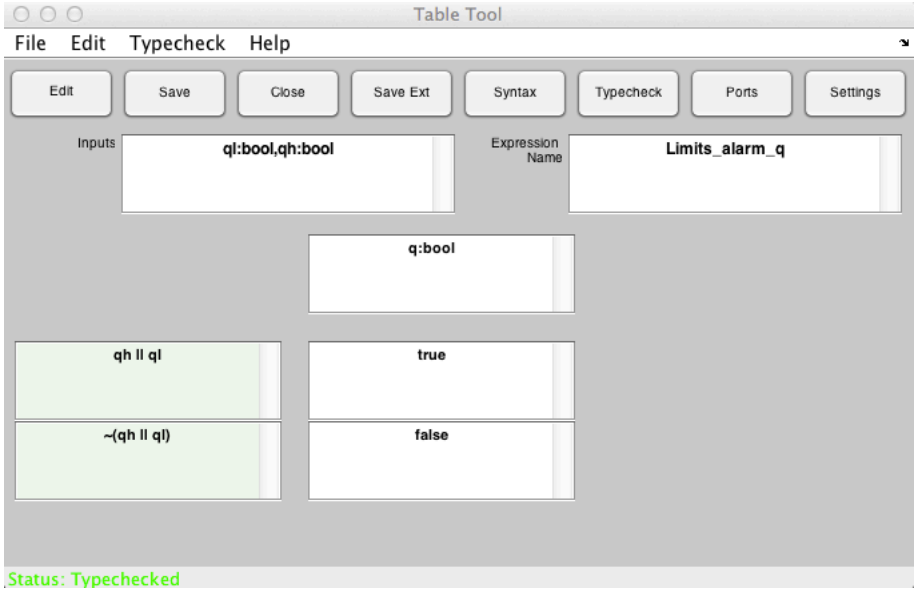


Figure 4.16: *Limits_ alarm_ q* Tabular Expression

After creating the three tables using our tool, we can check the disjointness and completeness properties (see the table status in the left bottom corner of each table that all the three tables are typechecked). Then we save the tables into the same folder, and generate the Event-B composition model for the *limits_alarm*. The final *limits_alarm* Event-B model is shown below.

An assumption $eps > 0$ is used in [22] for the qh and ql tabular expressions. But we found that the assumption $eps > 0$ can be relaxed to $eps >= 0$ (see Fig. 4.13). And another assumption $l + eps < h - eps$ for the q tabular expression is also used in [22]. But we found it is optional for proving the refinement relation between the concrete model and the abstract model (see Fig. 4.13).

An Event-B Specification of Limits_alarm_M
Creation Date: 15Apr2015 @ 05:35:35 PM

MACHINE Limits_alarm_M

SEES Limits_alarm_C

VARIABLES

ql

qh

q

x

h

eps

l

INVARIANTS

inv1 : $ql \in BOOL$

inv2 : $qh \in BOOL$

inv3 : $q \in BOOL$

inv4 : $x \in \mathbb{Z}$

inv5 : $h \in \mathbb{Z}$

inv6 : $eps \in \{eps | eps \in \mathbb{Z} \wedge (eps \geq 0)\}$

inv7 : $l \in \mathbb{Z}$

EVENTS

Initialisation

begin

act1 : $ql :\in BOOL$

act2 : $qh :\in BOOL$

act3 : $q :\in BOOL$

act4 : $x :\in \mathbb{Z}$

act5 : $h :\in \mathbb{Z}$

act6 : $eps :\in \{eps | eps \in \mathbb{Z} \wedge (eps \geq 0)\}$

act7 : $l :\in \mathbb{Z}$

end

Event *evt1* $\widehat{=}$

when

grd1 : $((qh = TRUE) \vee (ql = TRUE))$

then

act1 : $q := TRUE$

end

Event *evt2* $\widehat{=}$

when

grd1 : $(\neg((qh = TRUE) \vee (ql = TRUE)))$

then

act1 : $q := FALSE$

end

Event *evt3* $\widehat{=}$

when

$\texttt{grd1} : (x > h)$

then

$\texttt{act1} : qh := TRUE$

end

Event *evt4* $\widehat{=}$

when

$\texttt{grd1} : (((h - eps) \leq x) \wedge (x \leq h))$

then

$\texttt{act1} : qh := qh$

end

Event *evt5* $\widehat{=}$

when

$\texttt{grd1} : (x < (h - eps))$

then

$\texttt{act1} : qh := FALSE$

end

Event *evt6* $\widehat{=}$

when

$\texttt{grd1} : (x < l)$

then

$\texttt{act1} : ql := TRUE$

end

Event *evt7* $\widehat{=}$

when

$\texttt{grd1} : ((l \leq x) \wedge (x \leq (l + eps)))$

then

$\texttt{act1} : ql := ql$

end

Event *evt8* $\widehat{=}$

when

grd1 : $(x > (l + eps))$

then

act1 : $ql := FALSE$

end

END

4.5.2 Implementation of the *limits_ alarm* (Concrete Model)

The implementation of the *limits_ alarm* consists of six parts: one SUB (subtraction) block, one DIV (division) block, one ADD (addition) block, two *hysteresis* blocks, and one DISJ (disjunction) block, which are illustrated in the Fig. 4.17 [21]. The SUB, DIV, and ADD are regular operators so that we use arithmetic expression (predicate) of input variables to denote them directly. We only create two *hysteresis* blocks (see 4.2) and one disjunction blocks for the implementation of *limits_ alarm* in our tool. The tabular expressions are shown below (see Fig. 4.18, Fig. 4.19, and Fig. 4.20).

In order to prove that the implementation of *limits_ alarm* in Fig. 4.17 is a refinement of the *limits_ alarm* requirement, we generate the abstract model of the requirement first. And then we generate the concrete model of the implementation of *limits_ alarm*. We create a new Rodin project and manually import the two generated models into the new project. And then we manually add the refinement relation between the concrete model (the implementation of *limits_ alarm*, named *implementation_ M*) and the abstract model (the requirement of *limits_ alarm*, named *Limits_ alarm_ M*), like the *implementation_ M* model refines the *Limits_ alarm_ M* and every event in *implementation_ M* (see its Event-B model below) refines one event in the *Limits_ alarm_ M*. The Rodin prover generates some POs like the grd/GRD, grd/WD, inv/INV, act/WD, act/FIS and act/SIM. All of the POs are automatically discharged except two POs. That are evt5/grd1/GRD and evt8/grd1/GRD. These two POs are very similar, so we take the evt5/grd1/GRD as an example. This PO means that the concrete condition $x < h - eps \div 2 - eps \div 2$

should be stronger than its corresponding abstract event guard $x < h - eps$. We believe that this PO cannot be discharged because of the integer division problem in Rodin. Since Rodin can only support integer data type, if *eps* is an odd number, the concrete guard condition is not necessarily stronger than the abstract event that it refines. For example, if *eps* is 3, the concrete guard condition is $x < h - 3 \div 2 - 3 \div 2$, which is equivalent to $x < h - 2$ (integer division). While the abstract guard is $x < h - 3$. Therefore, the concrete guard is even weaker than the abstract guard which, of course, cannot be discharged. We solve this problem, by adding an extra assumption for *eps* in the concrete model manually, we assume *eps* is an even number ($eps\ mod\ 2\ =\ 0$). With this condition, there will be no accuracy loss, the concrete guard is at least as strong as the abstract event guard. So the prover should discharge the two undischarged POs. In fact, later we found that the predicate prover P1 of the Rodin Platform is enough to prove these undischarged POs with the above extra assumption ($eps\ mod\ 2\ =\ 0$).

Actually, the integer division issue in the Rodin Platform is an open problem. There are some other potential solutions. First, a lemma that "$eps \div 2 + eps \div 2 = eps$" can be added to the concrete model to fix this problem. Second, we can remove the assumption that $eps\ mod\ 2 = 0$, but use ($eps \div 2\ +\ eps\ mod\ 2$) instead of $eps \div 2$ to cope with the accuracy loss problem. Finally, using the latest library of the Rodin Platform to support the real number division is perhaps the best solution.

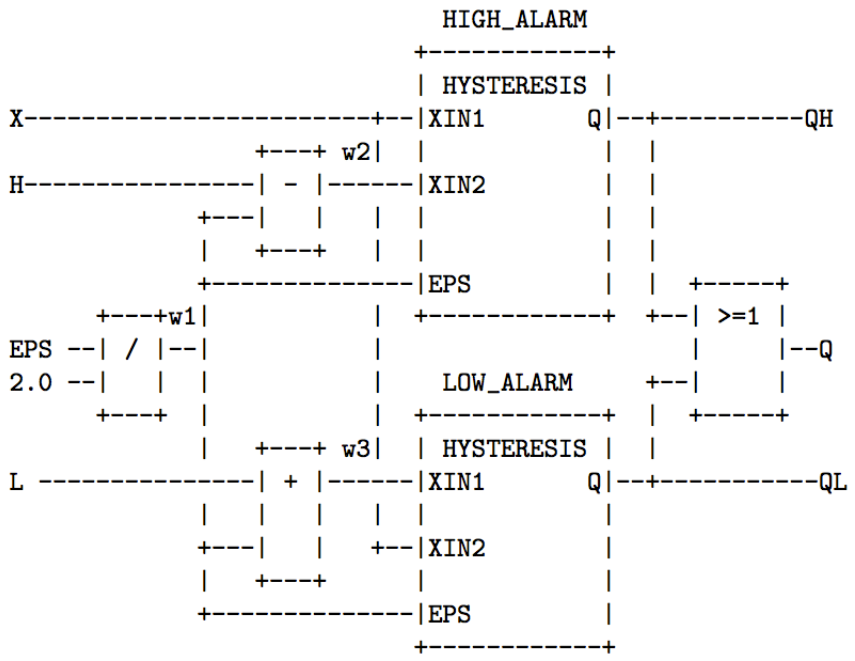


Figure 4.17: Implementation of Block *limits_alarm*

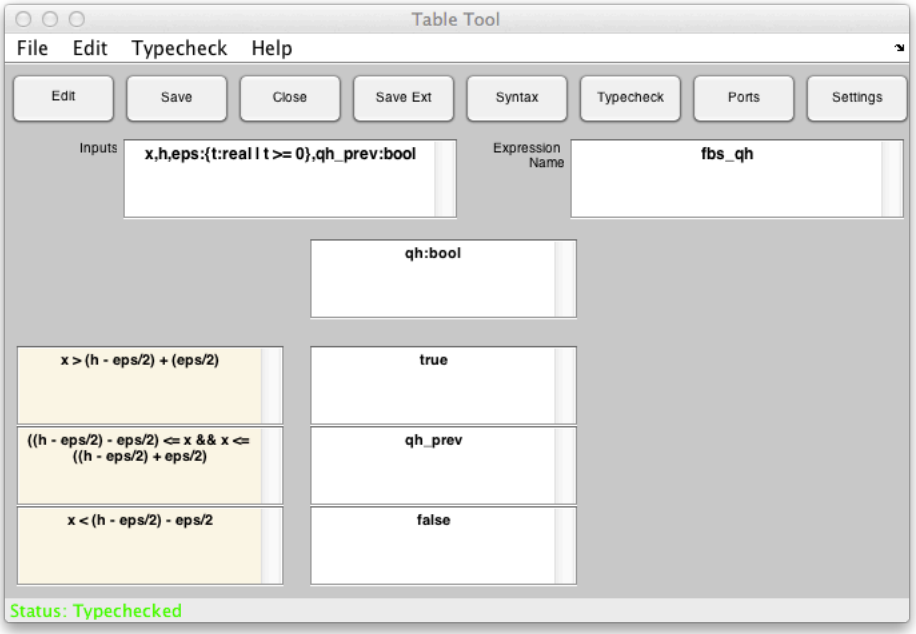


Figure 4.18: *Limits_alarm_qh* implementation Tabular Expression

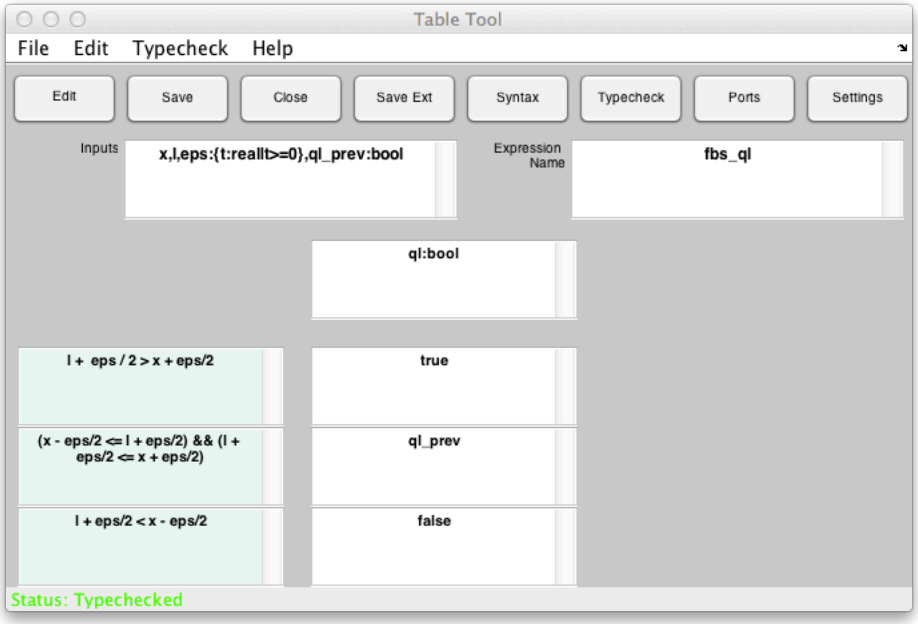


Figure 4.19: *Limits_ alarm_ ql* implementation Tabular Expression

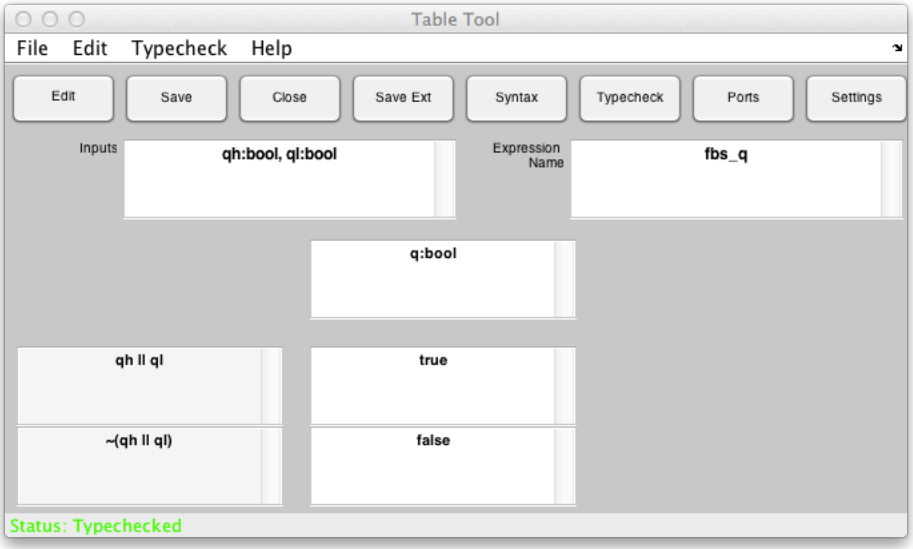


Figure 4.20: *Limits_ alarm_ q* implementation Tabular Expression

An Event-B Specification of implementation_M
Creation Date: 15Apr2015 @ 05:35:35 PM

MACHINE implementation_M

REFINES Limits_alarm_M

SEES implementation_C

VARIABLES

qh

ql

q

x

h

eps

l

INVARIANTS

inv1 : $qh \in BOOL$

inv2 : $ql \in BOOL$

inv3 : $q \in BOOL$

inv4 : $x \in \mathbb{Z}$

inv5 : $h \in \mathbb{Z}$

inv6 : $eps \in \{eps | eps \in \mathbb{Z} \wedge (eps \geq 0) \wedge (eps\ mod\ 2 = 0)\}$

inv7 : $l \in \mathbb{Z}$

EVENTS

Initialisation

begin

act1 : $qh :\in BOOL$

act2 : $ql :\in BOOL$

act3 : $q :\in BOOL$

act4 : $x :\in \mathbb{Z}$

act5 : $h :\in \mathbb{Z}$

act6 : $eps :\in \{eps | eps \in \mathbb{Z} \wedge (eps \geq 0) \wedge (eps\ mod\ 2 = 0)\}$

act7 : $l :\in \mathbb{Z}$

end

Event *evt1* $\widehat{=}$

refines *evt1*

when

grd1 : $((qh = TRUE) \vee (ql = TRUE))$

then

act1 : $q := TRUE$

end

Event *evt2* $\widehat{=}$

refines *evt2*

when

grd1 : $(\neg((qh = TRUE) \vee (ql = TRUE)))$

then

act1 : $q := FALSE$

end

Event *evt3* $\widehat{=}$

refines *evt3*

when

grd1 : $(x > ((h - (eps/2)) + (eps/2)))$

then

act1 : $qh := TRUE$

end

Event *evt4* $\widehat{=}$

refines *evt4*

when

$\texttt{grd1} : ((((h - (eps/2)) - (eps/2)) \leq x) \wedge (x \leq ((h - (eps/2)) + (eps/2))))$

then

$\texttt{act1} : qh := qh$

end

Event *evt5* $\widehat{=}$

refines *evt5*

when

$\texttt{grd1} : (x < ((h - (eps/2)) - (eps/2)))$

then

$\texttt{act1} : qh := FALSE$

end

Event *evt6* $\widehat{=}$

refines *evt6*

when

$\texttt{grd1} : ((l + (eps/2)) > (x + (eps/2)))$

then

$\texttt{act1} : ql := TRUE$

end

Event *evt7* $\widehat{=}$

refines *evt7*

when

$\texttt{grd1} : (((x - (eps/2)) \leq (l + (eps/2))) \wedge ((l + (eps/2)) \leq (x + (eps/2))))$

then

$\texttt{act1} : ql := ql$

end

Event *evt8* $\widehat{=}$

refines *evt8*

when

```
        grd1 : ((l + (eps/2)) < (x − (eps/2)))
    then

        act1 : ql := FALSE
    end
END
```

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis describes a tool TX2EB to generate Event-B models from tabular expressions. Our work was motivated by the fact that there is no existing tool that combines the tabular expressions (to check the disjointness and completeness of the requirements) and the Event-B language (good at modelling refinement, decomposition and consistency checking). By developing such a transformation tool on the basis of existing tools (TET [8], jTET [5]), we can check all the properties mentioned above in one tool to verify the correctness of the documented requirements. Our main contribution is automating this transformation process to facilitate software verification.

We have listed all the detailed requirements of the TX2EB tool. And then, based on the requirements, we discuss some design and implementation ideas motivated by each requirement. We have described the tool architecture and rational design decision for implementing the tool. Moreover, we elaborate on the functionality and relation of all the classes and interfaces in our tool, and the process of generating the Event-B Model, especially the Machine and Context file.

To assess the effectiveness of our developed tool, we use some case studies to generate Event-B models from tabular expressions. We have used 12

function blocks (FBs) of the IEC61131-3 standard for checking the correctness of of the functional behavior. These FBs are *alrm_int, block_sr, ctd, ctud, f_cmd, trig, r_trig, fbs_rs, hysteresis, integral, LAG* and *limits_alarm*. All the tabular expressions of the twelve FBs are successfully transformed into Event-B Models. We have used two function blocks (*hysteresis* and *ctd*) to illustrate the transformation process of the *no refinement* mode and the *with refinement* mode, respectively. Then we have used *alrm_int* function block to exemplify the process of table composition, in which functional behavior is described in multiple tables. Finally, the *limits_alarm* FB is used to prove the refinement relation between the system requirements and the corresponding implementation.

In summary, the TX2EB tool successfully transforms tabular expressions to corresponding Event-B Models. This tool has several benefits. First, the tool is very useful for detecting and fixing the errors and inconsistencies in the requirements at the beginning of the software development cycle. Because "An ounce of prevention is worth a pound of cure", it can reduce the total effort and expense spent on the whole software project. Second, the graphical user interface of the tool is very easy for the software engineer to use to check the proper properties of the requirements without requiring the user to know every formal detail of CVC3 and PVS. Third, the refinement based incremental development method is very useful for large and complex systems. Furthermore, the tool can be used to verify the refinement relation between an abstract model of the requirement and the concrete model of its implementation. Finally, our automated transformation from the tabular expressions to the Event-B models saves users the trouble of creating the Event-B model by hand. Therefore, it can increase the efficiency of a software engineer and reduce the time spent on V&V of the requirements, thus shortening the software development life cycle and cutting down on the total project cost.

5.2 Future Work

Future work can be explored in the following directions.

- The TX2EB tool is intended to be used as a verification tool to verify the correctness of the software and system requirements. But even the verification tool itself should also be verified. Therefore, the most urgent future work is to verify and validate the transformation process from Tabular Expressions to Event-B models.

- Because of the limitation of the TET tool, the quantifiers ($\forall$ and $\exists$) are not supported by the current version tool. To figure out a feasible and practical way to implement the quantifiers is another important future work.

- Until now, our case studies for the TX2EB tool are mainly on the IEC61131-3 standard function blocks. Even for this standard, we have not implemented all its function blocks. In the future, all the function blocks must be implemented using the TX2EB tool. Moreover, we must use this tool on some large and complex case studies. One possible case study is the Insulin Infusion Pump (IIP). We have manually transformed the tabular expressions of the Insulin Infusion Pump project to Event-B models. The next step is to use the TX2EB tool to generate the Event-B models automatically.

- Another future work is to combine the TX2EB tool with the EB2ALL tool [28] to generate the source code (C/C++/C#/Java) directly from tabular expressions.

- Currently, we have implemented only one refinement strategy to generate Event-B models from tabular expressions. However, we want to extend the current tool with the second refinement strategy [29] that also supports for formal model generation from tabular expressions when the system requirements are defined in hundreds of tabular expressions.

This new refinement strategy has very low memory and table traversing complexities compare to the first refinement strategy. To implement this refinement strategy is another future work.

- Currently, only the Horizontal Condition Table (HCT) is supported. Although HCT is the most common and useful kind of tabular expression. It is also useful to support some different types of tabular expression forms in the future.

- It is usually very boring and time-consuming for users to download, install and set up a tool. Therefore, in the future, we are planning to provide a service link on the website so that the users can access the service directly through the Internet, which will definitely increase the popularity of the tool.

- Some modern IDE can support auto code completion. Similarly, we are also planning to extend our tool to support auto condition completion for user inputs (i.e., when a user inputs a condition $x > 0$, the tool will give optional conditions automatically, like $x = 0$ and $x < 0$ to make the table disjoint and complete).

Bibliography

[1] Jean-Raymond Abrial. *Rigorous Open Development Environment for Complex Systems*. Information Society Technologies, 2005. 15, 16

[2] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010. 2, 10, 11, 16, 17

[3] R Alur, Dimitrios Hristu-Varsakelis, K-E Arzen, William Levine, John Baillieul, and TA Henzinger. *Handbook of networked and embedded control systems*. Springer Science & Business Media, 2007. 25

[4] Jonathan Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209, 1993. 1

[5] Matthew Dawson. Improvements to TET. Master's thesis, McMaster University, Hamilton, ON, Canada, 2015. 3, 42, 66, 67, 105

[6] Petr Dostál. Simulink and production-financial systems. *System*, 2:2. 25

[7] EB2ALL. eb2all.loria.fr. *in Formal Methods in System Design*, 2004. 4

[8] Colin Eles and Mark Lawford. A tabular expression toolbox for Matlab/Simulink. In *NASA Formal Methods*, pages 494–499. Springer, 2011. 1, 2, 8, 9, 28, 31, 43, 105

[9] Colin J. Eles. A tabular expression toolbox for Matlab/Simulink. Master's thesis, McMaster University, Hamilton, ON, Canada, 2011. 42, 44

[10] Software Engineering Research Group et al. Table tool system developer's guide. *CRL Report*, 339:38, 1997. 7

[11] IEC. *61131-3 Ed. 2.0 en:2003: Programmable Controllers — Part 3: Programming Languages*. International Electrotechnical Commission, 2003. 78, 91

[12] Ryszard Janicki, David Lorge Parnas, and Jeffery Zucker. *Tabular representations in relational documents*. Springer, 1997. 7

[13] Michael Jastram. *Rodin Users' HandBook Covers Rodin v.2.8*. Unknown, 2013. 10, 24

[14] Ying Jin and David Lorge Parnas. Defining the meaning of tabular mathematical expressions. *Sci. Comput. Program.*, 75(11):980–1000, November 2010. 7, 8

[15] Nasser Kehtarnavaz and Chandan Gope. Dsp system design using labview and Simulink: a comparative evaluation. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 2, pages II–II. IEEE, 2006. 25

[16] John C Knight. Safety critical systems: challenges and directions. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24th International Conference on*, pages 547–550. IEEE, 2002. 1

[17] M Lawford, P Froebel, and G Moum. Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. *in Formal Methods in System Design*, 2004. 1, 9

[18] David L Lempia and Steven P Miller. Requirements engineering management handbook. *National Technical Information Service (NTIS)*, 2009. 1

[19] Uwe Meyer-Baese and U Meyer-Baese. *Digital signal processing with field programmable gate arrays*, volume 65. Springer, 2007. 25

[20] Rashmi Nanda, Chia-Hsiang Yang, and Dejan Markovic. Dsp architecture optimization in Matlab/Simulink environment. In *VLSI Circuits, 2008 IEEE Symposium on*, pages 192–193. IEEE, 2008. 25

[21] Linna Pang. *An Engineering Methodology for the Formal Verification of Function Block Based Systems*. PhD thesis, McMaster University, Hamilton, ON, Canada, 2015. 91, 97

[22] Linna Pang, Chen-Wei Wang, Mark Lawford, and Alan Wassyng. Formalizing and verifying function blocks using tabular expressions and pvs. In *Formal Techniques for Safety-Critical Systems*, pages 125–141. Springer, 2014. 8, 78, 91, 94

[23] David L Parnas, GJK Asmis, and JD Kendall. Reviewable development of safety critical software. In *International Conference on Control and Instrumentation in Nuclear Installations, Glasgow, UK*, number 4.3, 1990. 7

[24] David Lorge Parnas. *Tabular representation of relations*. Citeseer, 1992. 7

[25] David Lorge Parnas, J Madey, and GJK Asmis. Assessment of safety-critical software in nuclear power plants. *Nuclear safety*, 32(2), 1991. 7

[26] David Lorge Parnas and Jan Madey. *Functional Documentation for Computer Systems Engineering: Version 2*. McMaster University, Faculty of Engineering, Communications Research Laboratory, 1991. 7

[27] David Lorge Parnas, Jan Madey, and Michal Iglewski. Precise documentation of well-structured programs. *Software Engineering, IEEE Transactions on*, 20(12):948–976, 1994. 7

[28] Neeraj Kumar Singh. *Using Event-B for Critical Device Software Systems*. Springer Science & Business, 2013. 4, 5, 107

[29] Neeraj Kumar Singh, Thomas S. E. Maibaum, Mark Lawford, and Alan Wassyng. Refinement automation and formal reasoning of tabular expression using event-b. *Unpublished*, 2015. 107

[30] Neeraj Kumar Singh, Hao Wang, Mark Lawford, Thomas S. E. Maibaum, and Alan Wassyng. Report 18: Formalizing Insulin Pump using Event-B. https://www.mcscert.ca/index.php/documents/mcscert-reports Technical Report 18, McSCert, McMaster University, October 2014. 45

[31] Colin Snook and Michael Butler. Uml-b: A plug-in for the event-b tool set. 2008. 3

[32] Colin Snook, Michael Poppleton, and Ian Johnson. Rigorous engineering of product-line requirements: a case study in failure management. *Information and Software Technology*, 50(1):112–129, 2008. 3

[33] Juing-Huei Su, Jiann-Jong Chen, and Dong-Shiuh Wu. Learning feedback controller design of switching converters via MATLAB/SIMULINK. *Education, IEEE Transactions on*, 45(4):307–315, 2002. 25

[34] Jerzy Tchorzewski. Systemic method for searching regularity of electricity market using MATLAB and Simulink environment. In *Electricity Market, 2008. EEM 2008. 5th International Conference on European*, pages 1–6. IEEE, 2008. 25

[35] A. Tewari. *Modern Control Design: With MATLAB and SIMULINK*. Wiley, 2002. 25

[36] Alan Wassyng and Ryszard Janicki. Tabular expressions in software engineering. In *Proceedings of ICSSEA*, volume 3, pages 1–46, 2003. 1, 7, 8

[37] Yong-qiang WU and Xi LUO. Simulation and analysis of the chaotic financial system based on Simulink. *Science and Technology of West China*, 22:025, 2009. 25

[38] Hao Xu. Model based system consistency checking using event-b. Master's thesis, McMaster Univeristy, Hamilton, ON, Canada, 2011. 16