TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

Shobit Jain 182458IVSM

IMPLEMENTATION OF MODEL TRANSFORMATIONS BETWEEN EVENT-B AND UPPAAL TIMED AUTOMATA

Master's Thesis

Supervisor: Leonidas Tsiopoulos
Ph.D.

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Shobit Jain 182458IVSM

EVENT-B JA UPPAALI AJAGA AUTOMAATIDE VAHELISE TEISENDUSE REALISEERIMINE

Magistritöö

Juhendaja: Leonidas Tsiopoulos Ph.D.

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Shobit Jain

05.01.2020

Acknowledgements

I would like to express my gratitude for my thesis supervisor Leonidas Tsiopoulos (Ph.D.) for his guidance during this project and encouraging me on every step. I have enjoyed having a professional relationship with him. I extend gratitude to Professor Jüri Vain for his outstanding contribution in this field. I also appreciate all the support I receive from the University of Tartu, Tallinn University of Technology and the Government of Estonia.

Furthermore, to my brother and family for all the unconditional support during the compilation of this dissertation. I also wish to thank Naveed Ahmed who has been a source of support. Many thanks to all friends that took part in the study and enabled this research to be possible.

I would like to thank Abhilash Yadav for pulling me up in difficult moments where I felt stumped and for goading me on to follow my dream of getting this degree. I sincerely appreciate God for everything from the bottom of my heart.

Abstract

This thesis work implements the tool support for the automatic mapping of Event-B models to UPPAAL models based on the mapping suggestions of [1], [2]. Event-B is a state-based formalism for the specification and verification of complex and critical systems with focus on the behavioural and safety aspects. Event-B follows the correct-by-construction paradigm wherein the system model is developed in a refinement-based stepwise manner. Theorem proving is the underlining tool for the verification of the system model. On the other hand, UPPAAL Timed Automata (UTA) is a state-based formalism with clear focus on the modelling and verification of real-time systems. UTA is well-supported by the UPPAAL toolbox for the modelling, simulation and verification of real-time systems. Verification is performed with the efficient built-in model-checker. Several attempts have been made to extend Event-B to support modelling and verification of timing properties [3]–[5]. Though, all of these works resulted in high complexity solutions. This motivated the integration of Event-B and UTA to mutually complement each other in complex system development without the need to extend any of the two formalisms.

The aim of this thesis is to implement the tool support for the established mapping between Event-B and UTA by processing the underlying Extensible Markup Language (XML) base of the mapped models.

Keywords: Event-B, iUML-B, UPPAAL, Timed Automata, real-time systems, model transformation, verification and validation.

This thesis is written in English and is 44 pages long, including 8 chapters, 18 figures, 2 tables and 1 appendix.

Annotatsioon

Käesolevas magistritöös on realiseeritud automaatne mudelteisendus Event-B formalismist UPPAALi ajaga automaatide formalismi. Teisendus põhineb publikatsioonides [1], [2]. toodud kirjeldusel. Event-B on oleku-põhine formalism, mis on loodud keerukate ja kriitiliste süsteemide spetsifitseerimiseks ning verifitseerimiseks fookusega süsteemide käitumis- ja ohutusaspektidel. Event-B toetab nn korrektsus-läbi konstruktsiooni paradigmat, mille puhul süsteemi mudel luuakse lähtudes abstraktsest spetsifikatsioonist läbi nõuete suhtes korrektsust säilitavate täpsustamissammude. Süsteemi mudeli korrektsuse verifitseerimine toimub Event-B teoreemitõestaja abil. Erinevalt Event-B-st on UPPAALi ajaga automaadid (UTA) oleku-põhine formalism, mille fookus on reaalaja süsteemide modelleerimisel ja verifitseerimisel. UTA-l on hästi väljaarendatud tööriistatugi reaalaja süsteemide modelleerimiseks, simulatsiooniks ja verifitseerimiseks. Mudelite verifitseerimiseks on UTA tööriistadesse sisseehitatud efektiivne mudelkontrolli motor. Kuigi Event-B puhul on tehtud mitmeid katseid laiendada formalismi adresseerimaks ka ajaga seotud omaduste modelleerimist ja verifitseerimist (vt viited [3]–[5]), on senised katsed viinud väga keerukate ja raskesti kasutatavate lahendusteni. See on andnud motivatsiooni Event-B ja UTA integreerimiseks, et luua keerukate süsteemide arenduseks vastastikku üksteist täiendav formalismide komplekt ilma vajaduseta laiendada neid neile mitteomaste omadustega.

Magistritöö raames loodud tööriist teisendab Event-B mudelid UTA mudeliteks parsides Event-B XML (Extensible Markup Language) failivormingut ning teisendades selle vastavasse UTA XML-failide vormingusse.

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 44 leheküljel, 8 peatükki, 18 joonist, 2 tabelit ja 1 apendiks.

List of abbreviations and terms

TA	Timed Automata
UTA	UPPAAL Timed Automata
iUML	Integrated Unified Modelling Language
PN	Petri Net
XML	Extensible Markup Language
UL	UPPAAL Language
MA	Movement authority
VSS	Virtual Sub Sections
VBD	Virtual Block Detector
TTD	Trackside Train Detection
RBC	Radio Block Centre

Contents

List of figures

List of tables

1 Introduction

Formal methods are techniques to model and reason about complex systems as mathematical objects. They are concerned with the application of a reasonably wide variety of theoretical computer science fundamentals. Formal methods are supported by formal languages (e.g., mathematical logic) and allow applying formal system verification which is essentially a complement to usual system/software testing for making sure the behaviour of the system conforms to its requirements. While systems grow more complex and safety becomes a more serious issue, the formal approach to system design allows for an extra level of insurance.

Formal methods differ from other system design techniques through the utilization of formal verification systems, such as theorem provers and model-checkers. Formal methods are necessary to avoid building/developing incomplete and/or incorrect systems due to unfinished, incompatible or ambiguous requirements. Formal methods can be applied to all steps of the software life-cycle and on various levels of abstraction.

Event-B [6] is a formal modelling method for developing a system and the formalism is widely accepted for the modelling and verification of complex systems. Event-B features the use of set theory, the use of the refinement paradigm and the use of mathematical theorem proving for developing and verifying systems at different levels of abstraction and in a correct-by-construction manner.

The UPPAAL toolbox is an environment developed as a collaborative effort by the Uppsala University and Aalborg University and it is used for modelling, validation and verification of real-time systems. The models are implemented as networks of timed automata which can be extended with data types (bounded integers, arrays, etc.).

1.1 Research goal

This thesis work targets a missing-link in model transformations between Event-B and UTA. The tool support for the mapping between Event-B and UTA has been missing. The integrated framework has been established but without tool support. The research goal is to implement tool support for the model transformations between Event-B and UTA.

1.2 Motivation

When developing large and complex systems, refinement alone is often insufficient. Refinement in the context of system modelling refers to the arrangement of models across various abstraction levels. Also, adding and verifying timing properties to Event-B models introduces complexity to the development process [3]–[5]. Furthermore, prior research works emphasize the need for integration of Event-B and UTA in order to address complexity of development of real time systems [1], [2] and [7]. By following the refinement-based and correct-by-correction approach, Event-B can handle efficiently the data and functional refinement of the model and UTA can handle efficiently the timing refinement of the same model focusing only on the verification of timing properties. As a result, the framework has been established [1], [2] and [7], but the tool support for automated mapping is still missing.

2 Related Work and Background

In this section, we shall explore some related works that inspire this thesis. We identify two important ones that serve as precedence for our work and also several others that are related to this project.

2.1 Refinement-Based Development of Timed Systems [7]

This paper was the first work regarding the integration of Event-B and UTA providing the theoretical basis for the work in this thesis. The authors showed that Event-B and UTA share the same functional semantics and proposed a model-level mapping via Extended Finite State Machines for each refinement level. The result of a data refinement step within Event-B served as an input to timing refinement step. Extended refinement proof obligations were also presented for both Event-B and UTA in order to guarantee correctness of the proposed system development. The approach was demonstrated on a safety controller design case study.

2.2 Integrating Refinement-Based Methods for Developing Timed Systems [2]

The model-level mapping proposed in [7] results in potentially too large UTA models. In [2] an event-level mapping was proposed in order to address the aforementioned issue. Each event from an Event-B model was mapped to a distinct UTA model eventually forming a full model through parallel composition. Also, to further facilitate the approach the mapping rules were defined on syntactic level. Since we reuse the suggestions of this paper, details on the mapping will appear in Section4.

2.3 Integration of iUML-B and UPPAAL Timed Automata for Development of Real-Time Systems with Concurrent Processes [1]

In this paper the mapping is still based on the events, as in [2], but the UTA model structure is partly extracted from the iUML-B state machine diagram which facilitates the development of the Event-B model by visualizing it. This way the potentially computational heavy approach proposed in [2] can be mitigated.

In this thesis we reuse and implement in a tool the mappings suggested in [1], [2] and details appear in Section 4.

2.4 Other related work

The following papers facilitate us in better understanding the mapping of Event-B and UTA as well as in observing various alternative options for mapping.

2.4.1 The EventB2PN Tool: From Event-B specification to Petri Nets through Model Transformation [8]

This paper proposes transformation rules from Event-B specifications to Petri Nets (PN) structures. Since PN can be mapped to UTA [9], this work could be beneficial for the Event-B to UTA mapping by providing fully the skeleton/structure of the UTA models, compared to the mapping proposed in [1]. At the time of working for this thesis the tool presented in [8] was under re-building and this did not allow us to consider further this work.

2.4.2 Code generation for Event-B [10]

This paper presents the syntactic translation from Event-B to Java programs. Every Event-B model are formed from *contexts* and *machines*. Java classes of Event-B represents that constants and their properties are defined under contexts. On the other hand, variables and their properties are defined under machine class. Furthermore, it explains initialization, guard, action, state transition and their relationship with each other. This paper acted as the guideline to understand the data structure and creating objects with an object-oriented programming language for Event-B model.

3 Preliminaries

3.1 Preliminaries of Event-B and iUML-B

Event-B [6] is a state-based formalism for the development of complex systems. Event-B adheres to the refinement approach wherein the system is created in a stepwise manner progressively adding details proving that each refinement step preserves the correctness of the previous steps. A model in Event-B, a *machine,* can be interpreted as a transition system where the variable valuations constitute the states and the events represent the transitions updating the variables. Event-B is supported by the Rodin Platform [11], which can be extended with plugins facilitating in various ways the modelling and verification process.

iUML-B is a graphical front-end for Event-B [12] implemented as plugin for the Rodin Platform allowing to build a model through a diagrammatic way in the form of state-machines and class diagrams. The embedded generator then generates Event-B automatically. Class diagrams model data relationships, while state-machines visualize the states and transitions of an Event-B machine. In a state-machine with a transition $e1$ between states $S1$ and $S2$, transition $e1$ can be fired if the state is $S1$ and the guard of the transition $G\ (t, v)$ evaluates to *true*. When $e1$ is fired it changes the state to $S2$ and may also modify other variables of the state-machine via actions $S\ (t, v)$. This corresponds to event $e1$ in Event-B [1].

$$e1 = \textbf{any}\ t\ \textbf{where}\ state = S1 \wedge G\ (t, v)\ \textbf{then}\ state := S2 \parallel S(t, v)\ \textbf{end}$$

An Event-B model holds two sections: a dynamic part (called *machine*) modelled by a transition system and a static part (called *context*) specifying the model's parameters, types and assumptions about them [13].

In Event-B dependency relationships are used to better structure the model. The SEES relationship formulates the relationship between a machine and its context for inquiring the static structure of the discrete system (constants, carrier sets, partitions of the sets, axioms, theorems, etc). In a refinement step a context may be extended by a new context and this is modelled with the EXTENDS relationship. A machine refines another machine with the REFINES relationship.

3.2 Preliminaries of UPPAAL Timed Automata

UTA [14] are defined as a closed network of extended timed automata that are called processes. The processes are combined into a single system by synchronous parallel composition like that in process algebra CCS. The nodes of the automata graph are called locations and directed lines between locations are called edges. For each edge, which is a transition between two locations, conditions or guards can be defined. Whenever the guard holds, the edge can be fired, which leads to a new location. Communication and synchronisation between different automata is taken care of by *send* and *receive* actions. An action *send* over a channel *h* is denoted by *h*! and its co-action, *receive* is denoted by *h*?. Formally, an UTA is defined as the tuple (L, E, V, CL, $Init$, Inv, T_L), where [14]:

– L is a finite set of locations,

– E is the set of edges defined by $E \subseteq L \times G\ (CL, V) \times Sync \times Act \times L$, where

- $G\ (CL, V)$ is the set of constraints in guards,
- *Sync* is a set of synchronization actions over channels and
- *an Act* is a set of sequences of assignment actions with integer and Boolean expressions as well as with clock resets.

– V denotes the set of integer and Boolean variables,

– CL denotes the set of real-valued clocks ($CL \cap V = \emptyset$),

– $Init \subseteq Act$ is a set of assignments that assigns the initial values to variables and clocks,

– $Inv : L \rightarrow I(CL, V)$ is a function that assigns an invariant to each location, $I(CL, V)$ being the set of invariants over clocks CL and variables V and

– T_L: $L \rightarrow$ {*ordinary, urgent, committed*} is the function that assigns the type to each location of the automaton.

In *urgent* locations an outgoing edge will be executed immediately when its guard holds. *Committed* locations are useful for creating atomic sequences of process actions since an outgoing edge must be executed immediately without time passing.

4 Mapping Rules and Implementation of Model Transformation[1]

In this section we first re-introduce the mapping rules from [1] and then we proceed with several detailed suggestions for model transformations based on observations at XML model representation level as well as on observations of Event-B events.

Mapping of Functions and Predicates [1]**.** Variables of integer and enumerated types in Event-B become integers in UTA, while finite sets and relations in Event-B are mapped to (multidimensional) arrays in UTA. The complex set and relational operators of Event-B can then be implemented as C-functions in UTA. More elaboration on this appears later on in section 4.2.9.

Mapping of Events [1]**.** Transitions in iUML-B state machines are generally translated to edges in UTA. In Figure 1 we exemplify the translation with an iUML-B state machine and Event-B code to the left and a corresponding UTA model to the right. Let

e = **any** p **where** G (p,v) **then** $S(p,v)$ **end**

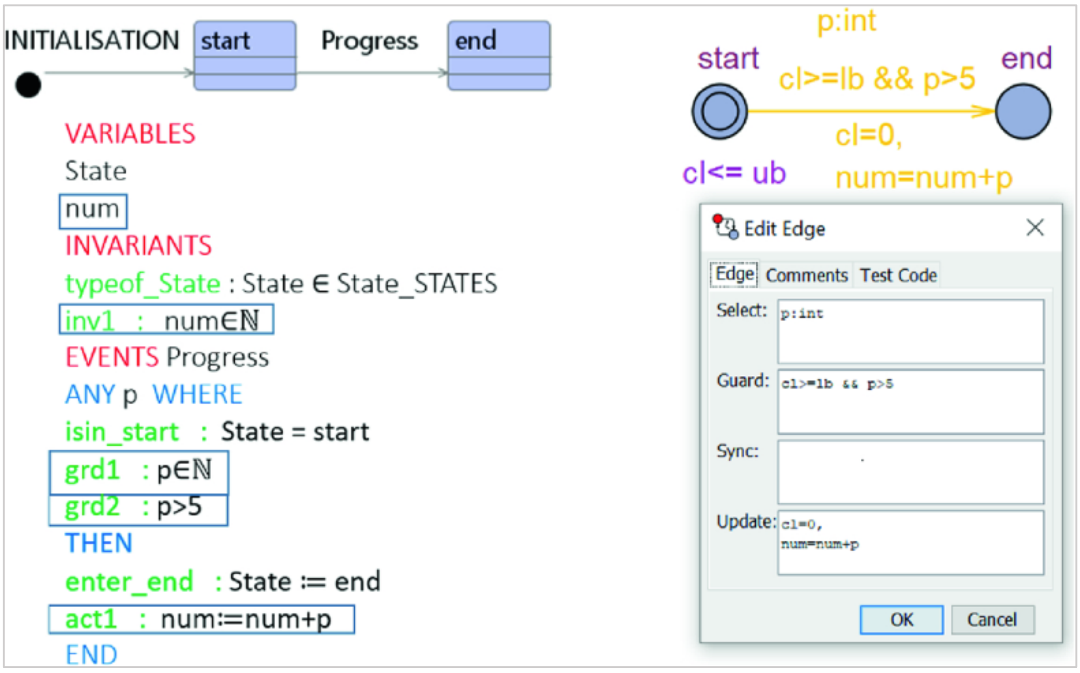
be an event of Event-B, then

(i) the parameter p will appear in the select label of the UTA edge, which contains a comma separated list of **p : int** expressions where p is a variable name and *int* is a defined type (see Figure 1).

(ii) the event guard $G(p,v)$ is mapped to the guard $G(V)$ of an edge where V denotes UTA variables corresponding to variables v (**p> 5** in Figure 1).

(iii) the event action $S(p,v)$ corresponds to assignment statements (updates) $V'= S(V)$ of the UTA edge (**num:=num+p** in Figure 1).

[1] https://bitbucket.org/shobitjain/thesis-eventb-to-uppaal



```
VARIABLES
State
num
INVARIANTS
typeof_State : State ∈ State_STATES
inv1  :  num∈ℕ
EVENTS Progress
ANY p  WHERE
isin_start  :  State = start
grd1  : p∈ℕ
grd2  : p>5
THEN
enter_end  : State ≔ end
act1  :  num≔num+p
END
```

Select: p:int

Guard: cl>=lb && p>5

Sync:

Update: cl=0,
num=num+p

Figure 1. Transition structure in iUML-B (left) and UTA (right) [1]

4.1 Model Transformation at XML level

In order to build a tool for transforming a model from Event-B to UTA, a predefined set of rules/pre-requisites is needed for model processing. For instance, variables must contain pre/post keywords to identify their type. Thus, some assumptions are taken into consideration. Two main such assumptions are listed below:

1. Every XML tag has few properties in Event-B model. The tag “carrier set” mainly contains three properties; *name*, *source* and *type* and also holds sub-tag “*constant*”. The type of *constant* consists of the name of *carrier set*. The assumption is if name property of *carrier set* contains “*_STATES*” keyword then all *constants* with the same type as the name of *carrier set* will be considered to become locations in the UTA model.
2. When the value in predicate holds ‘×’ as an argument for a guard, it is considered as the number of instances defined at runtime by the user. It is transformed as ‘id’ in UPPAAL for process instantiation.

4.2 Pattern Composition Elements for Mapping Algorithm

In this subsection we elaborate at XML model representation level with several detailed suggestions for model transformations exemplified with XML code snippets from the airport control system case study of [1] regarding the landing process of airplanes.

4.2.1 Constants

1. Whenever label property of carrier sets and axioms contains "states" as the post keyword. It would ignore all those constants that belong to this particular carrier set because these certain constants are variable states of the system.

2. In *context*, when the name of carrier set is similar to type of constant then constant belongs to that carrier set as shown in the snippet below, taken from the airplane landing case study of [1], regarding the carrier set for airplane's fuel level and its elements. Otherwise the type of constant would be a predefined datatype.

```
<org.eventb.core.scCarrierSet name="Fuel_Level" org.eventb.core.type="ℙ(Fuel_Level)"/>
<org.eventb.core.scConstant name="High" org.eventb.core.type="Fuel_Level"/>
<org.eventb.core.scConstant name="Low" org.eventb.core.type="Fuel_Level"/>
<org.eventb.core.scConstant name="Medium" org.eventb.core.type="Fuel_Level"/>
```

3. When the predicate property of axioms contains "partition" as shown in the snippet below, regarding the same case study of [1], then the first word after "partition" would be the type of constants (Fuel_Level) with the possible values in curly brackets ({High}, {Medium} and {Low}) being the names of constants.

```
<org.eventb.core.scAxiom name="Airport_C3" org.eventb.core.label="axm1" org.eventb.core.predi
cate="partition(Fuel_Level,{High},{Medium},{Low})"/>
```

The type and name of constants can be validated by observing the name and type from constant tag as shown in the snippet below.

```
<org.eventb.core.scConstant name="High" org.eventb.core.type="Fuel_Level"/>
```

4. The constant values of above type of constants are assigned from the predicate property of axioms as shown below.

```
<org.eventb.core.scAxiom name="Airport_C6" org.eventb.core.label="axm4" org.eventb.core.predi
cate="Level_Number={High ↦ 2,Medium ↦ 1,Low ↦ 0}"/>
```

The assignments and declarations of constants in UPPAAL on behalf of current and above two statements:

const int High = 2;   const int Medium = 1;   const int Low = 0;

5. Integer constants are directly identified by their type. Type (ℤ) and the constant value assigned from the predicate property of axiom.

4.2.2 Variables

1. **BOOL** is the enumerated set for the FALSE and TRUE Boolean values. It is defined in a predicate. The set of **integer** numbers are defined as ℤ and set of **natural** numbers are defined as N.

2. When integer type variable contains the number of instances for example ℙ(*SET_SIZE*×ℤ) then name of variable is a variable name. It would be transformed in two declarations like: 1) "typedef int [0, *SET_SIZE* - 1] id_t;", 2) "id_t *SET_SIZE* [id_t];" in UPPAAL. The value of *SET_SIZE* is given by user at runtime.

3. A power set is a set of all possible subsets of a set. When the power set is declared in Event-B like ℙ(*SET*) and it does not have the Cartesian product sign (×) as shown in the snippet below, then it would be transformed as const int variable[*SET*]; where *SET* is declared as const int *SET* = 2; and typedef int [0, *SET* - 1] id_t.

```
<org.eventb.core.scVariable name="variable" org.eventb.core.type="ℙ(SET)"/>
```

4.2.3 Invariants

Invariants define the types of variables including predefined types (**bool**, **int**, etc) in the context of a machine. They also specify system properties that should hold in every state of the system.

The invariants that contain "belongs to" (∈) in predicate property are declarations of the variables providing information about the type of the variables. This information can also be extracted from the type property of the variables. So we can ignore these invariants.

For instance the snippet below from the case study model of [1] show the declaration of a variable with an invariant.

```
<org.eventb.core.scInvariant name="Airport_M1_implicitContext" org.eventb.core.label="inv3" org
.eventb.core.predicate="queueH_in∈ℕ"/>
```

An example of an invariant that defines a system property regarding the relation of two variables is shown below.

```
<org.eventb.core.scInvariant name="Airport_M1_implicitContex}" org.eventb.core.label="inv7" org
.eventb.core.predicate="queueH_out≤queueH_in"/>
```

4.2.4 Axioms

Axioms are presumed properties of carrier sets and constants. In *context*, an axiom exists for every carrier set which contains a statement with partition property in its predicate. This property contains all the constants that belong to this particular carrier set. For instance the two snippets below show the declaration of constants, carrier set and their relation with axioms.

```
<org.eventb.core.scAxiom name="Airport_C3" org.eventb.core.label="axm1" org.eventb.core.predi
cate="partition(Fuel_Level,{High},{Medium},{Low})"/>
<org.eventb.core.scCarrierSet name="Fuel_Level" org.eventb.core.type="ℙ(Fuel_Level)"/>
<org.eventb.core.scConstant name="High" org.eventb.core.type="Fuel_Level"/>
<org.eventb.core.scConstant name="Level_Number" org.eventb.core.type="ℙ(Fuel_Level×ℤ)"/>
<org.eventb.core.scConstant name="Low" org.eventb.core.type="Fuel_Level"/>
<org.eventb.core.scConstant name="Medium" org.eventb.core.type="Fuel_Level"/>
```

The range of carrier set would be available also in a predicate property.

```
<org.eventb.core.scAxiom name="Airport_C5" org.eventb.core.label="axm3" org.eventb.core.predi
cate="Level_Number∈Fuel_Level ⤖ 0 .. 2"/>
```

The above range from the axiom would be transformed in UPPAAL like:

typedef int[0,2] Fuel_Level;

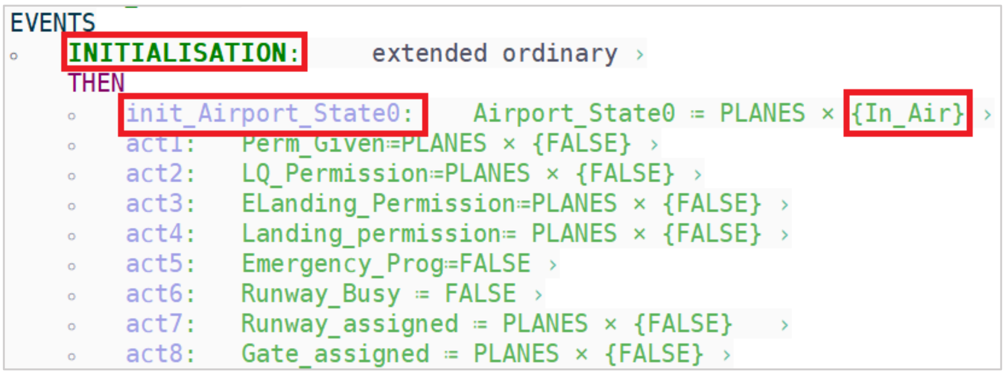
4.2.5 Events

1. In airport control model case study [1], the model is constructed in the form of a state machine by considering each event as an edge/transition from one state (given as part of the guards with “isin_” keyword) to another (given as part of the actions with “enter_” keyword). A variable “Airport_State0” is used to keep track of the state machine state. When an event is triggered, the initial and final states are assigned to “Application_State0” along with the guards and actions of that event. This variable is also responsible to hold all states of the system.
2. If an event does not contain an action with label property having the “enter_” keyword, it means that it is a self-loop transition on the source state of the event.

4.2.6 States

1. The initial state of the state machine is detected from the initialisation at the beginning of the events section. The action with “init_” keyword is assigning the initial state to the variable holding the state machine state information. To exemplify, as shown in the snippet below and in Figure 2, the initial state machine state for all airplanes in the airport control case study is “In_Air”. The rest of the initialisations concern the rest of the variables of the system.

```
<org.eventb.core.scAction name="''' org.eventb.core.assignment="Airport_State0 ≔ PLANES ×
{In_Air}" org.eventb.core.label="init_Airport_State0"/>
```



```
EVENTS
○   INITIALISATION:      extended ordinary ›
    THEN
      ○  init_Airport_State0:     Airport_State0 ≔ PLANES × {In_Air} ›
      ○  act1:    Perm_Given≔PLANES × {FALSE} ›
      ○  act2:    LQ_Permission≔PLANES × {FALSE} ›
      ○  act3:    ELanding_Permission≔PLANES × {FALSE} ›
      ○  act4:    Landing_permission≔ PLANES × {FALSE} ›
      ○  act5:    Emergency_Prog≔FALSE ›
      ○  act6:    Runway_Busy ≔ FALSE ›
      ○  act7:    Runway_assigned ≔ PLANES × {FALSE}   ›
      ○  act8:    Gate_assigned ≔ PLANES × {FALSE} ›
```

Figure 2. Initialization of States

2. In case the initial location is not detected from the initialization section then the tool searches for “isnotin_” keyword from the label property in the guards if it detects then

it initiate find location as initial location for instance if label property contains “isnotin_sm” then “sm” is an initial location.

3. When events do not contain “isin_”, “enter_” or “isnotin_” keywords they are considered as extra information for transformation to UPPAAL. It means that these events are not part of the transitions of the state machine of the Event-B system. Such events are handled as events that form separate UPPAAL automata themselves which are then composed in parallel to form a complete automaton, as it is proposed in [1]. We elaborate more on this in the Case Study section.
4. When the label property of an action in an event contains the “leave_” keyword, it would be considered also as the source state of the transition as with the “isin_” case. The only difference is that “isin_” appears in guards and “leave_” appears in actions. The target state of the transition would remain as described in statement 1 of Events subsection 4.2.5 above, i.e., indicated with the “enter_” keyword in one of the actions. Figure 3 shows the UPPAAL location extraction process in the form of a flow chart.

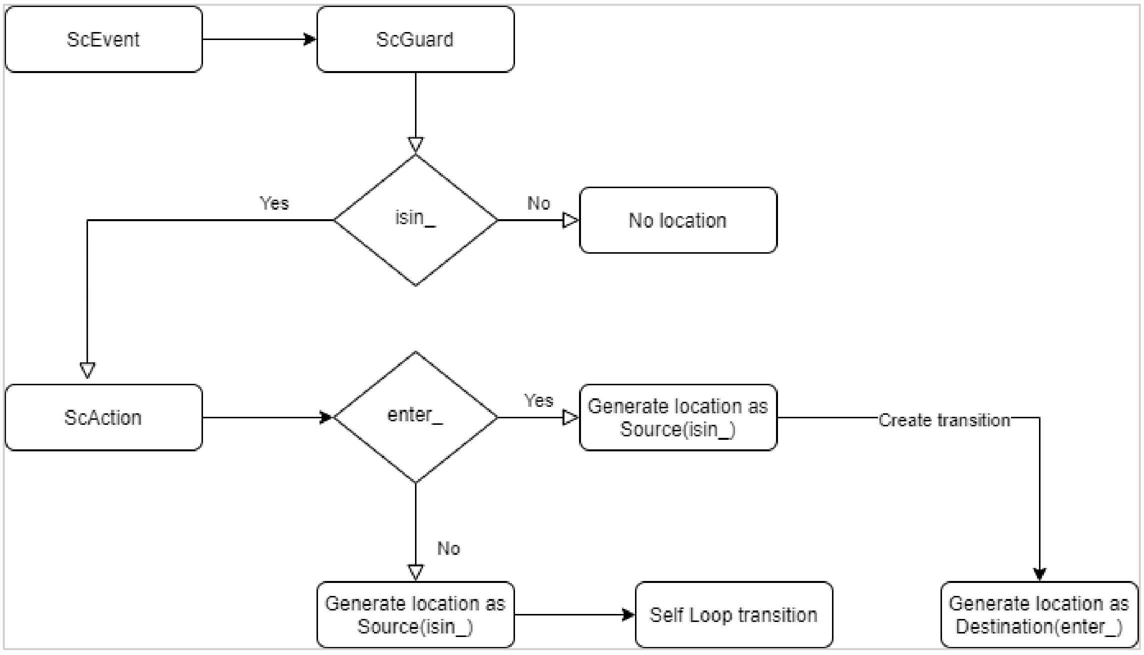

ScEvent
ScGuard
Yes
isin_
No
No location
ScAction
enter_
Yes
Generate location as Source(isin_)
Create transition
No
Generate location as Source(isin_)
Self Loop transition
Generate location as Destination(enter_)

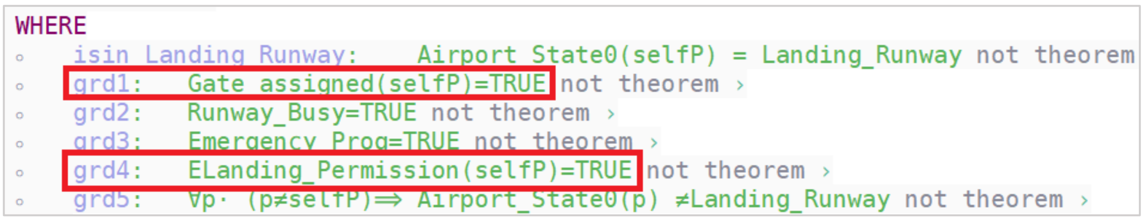
Figure 3. Step-by-step process to extract transition

4.2.7 Guards

1. The predicate property of guards in Event-B events holds the guard information for UPPAAL edges. All guard would be considered except the ones containing the "isin_" keyword in label properties. The reason is such guards represent the source locations

as explained earlier. Therefore, all guard tags with labels “grd(X)” are instantiated guards for the transitions based on the process instance and the values are in predicate properties.

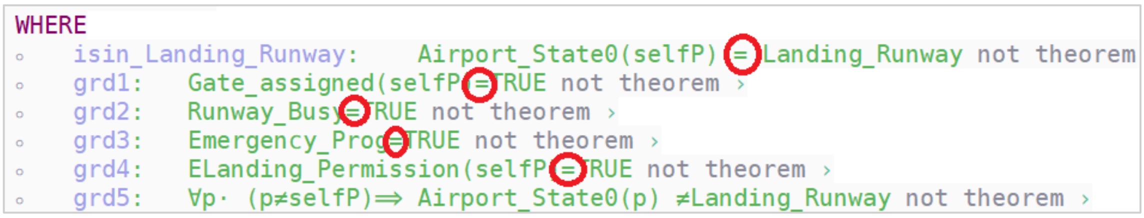
2. In order to generalize the transformation of parameters of every guard from Event-B on the basis of assumption 2 of section 4.1, the algorithm transforms the parameter by using regular expression for UL for instance (X) => [id]. In Figure 4, X is equal to “selfP” indicating the variables instantiation for each airplane.



```
WHERE
◦    isin_Landing_Runway:    Airport_State0(selfP) = Landing_Runway not theorem
◦    grd1:    Gate_assigned(selfP)=TRUE not theorem ›
◦    grd2:    Runway_Busy=TRUE not theorem ›
◦    grd3:    Emergency_Prog=TRUE not theorem ›
◦    grd4:    ELanding_Permission(selfP)=TRUE not theorem ›
◦    grd5:    ∀p· (p≠selfP)⇒ Airport_State0(p) ≠Landing_Runway not theorem ›
```

Figure 4. Parameter declarations in Guards (Event-B)

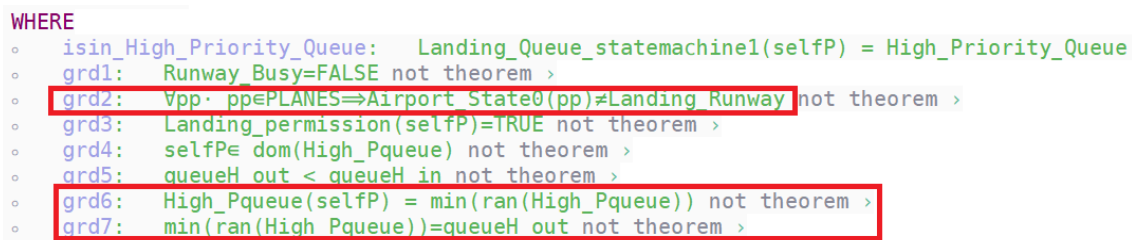
3. In Event-B, “=” is used for machine guards (see Figure 5) but UPPAAL supports equity operators as in C-like statements. Therefore, “=” is transformed to “==” for UL.



```
WHERE
◦    isin_Landing_Runway:    Airport_State0(selfP) = Landing_Runway not theorem
◦    grd1:    Gate_assigned(selfP)=TRUE not theorem ›
◦    grd2:    Runway_Busy=TRUE not theorem ›
◦    grd3:    Emergency_Prog=TRUE not theorem ›
◦    grd4:    ELanding_Permission(selfP)=TRUE not theorem ›
◦    grd5:    ∀p· (p≠selfP)⇒ Airport_State0(p) ≠Landing_Runway not theorem ›
```

Figure 5. Replaceable equity signs in guards

4. In Figure 6, grd2 states that for all airplane instances their state is not equal to Landing_Runway state from the Airport_State0 set. Since this is not supported directly in UPPAAL guards, it has to be implemented in a C-like function in local declarations of the UPPAAL template to be called from the guard. A similar solution has to be applied for guards grd6 and grd7.

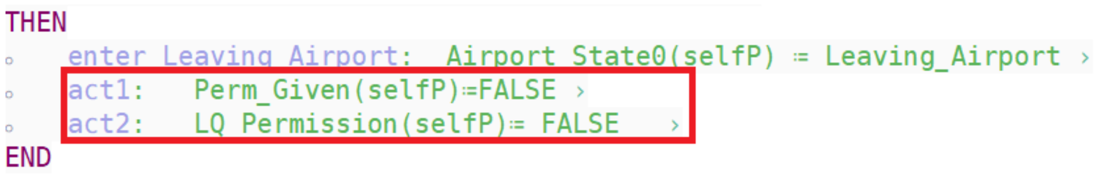


```
WHERE
○   isin_High_Priority_Queue:   Landing_Queue_statemachine1(selfP) = High_Priority_Queue
○   grd1:   Runway_Busy=FALSE not theorem ›
○   grd2:   ∀pp· pp∈PLANES⇒Airport_State0(pp)≠Landing_Runway not theorem ›
○   grd3:   Landing_permission(selfP)=TRUE not theorem ›
○   grd4:   selfP∈ dom(High_Pqueue) not theorem ›
○   grd5:   queueH_out < queueH_in not theorem ›
○   grd6:   High_Pqueue(selfP) = min(ran(High_Pqueue)) not theorem ›
○   grd7:   min(ran(High_Pqueue))=queueH_out not theorem ›
```

Figure 6. Complex guards in Event-B

4.2.8 Actions

1. The assignment properties of actions in events contain the value of assignment for the corresponding UPPAAL updates on edges. All actions would be considered except the ones containing "enter_" in label properties. The reason is this action represents the destination location as stated earlier. Therefor all the other action tags with labels would be defined as updates (assignments) in UPPAAL and their values are assignment properties, as shown in Figure 7.

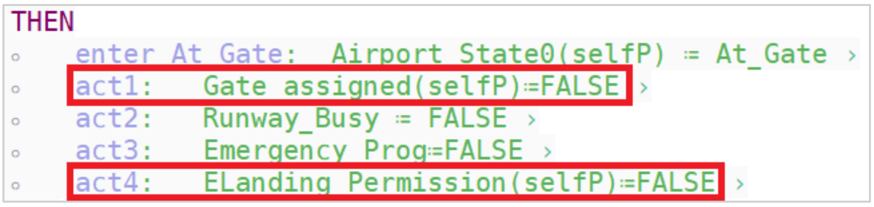


```
THEN
○   enter_Leaving_Airport:   Airport_State0(selfP) ≔ Leaving_Airport ›
○   act1:   Perm_Given(selfP)≔FALSE ›
○   act2:   LQ_Permission(selfP)≔ FALSE   ›
END
```

Figure 7. Identification of assignment

2. Functional override f(x): = E updates with value E the function f at place x[1]. When function overrides exist in assignments of actions in Event-B for instance (ELanding_Permission ≔ ELanding_Permission {selfP ↦ FALSE}) is transformed like (ELanding_Permission [id] := false). Where "id" is defined on the basis of assumption 2. For instance, in Figure 8 except first action with "enter_" keyword which corresponds to the destination state, all actions are assignments where the parameterized variables are of type bool and a value FALSE is assigned to them and selfP corresponds to the id of the selected instance.

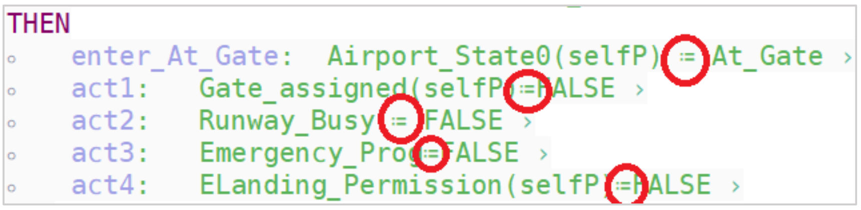
[1] https://wiki.event-b.org/images/EventB-Summary.pdf



```
THEN
∘    enter_At_Gate:   Airport_State0(selfP) ≔ At_Gate ›
∘    act1:    Gate_assigned(selfP)≔FALSE ›
∘    act2:    Runway_Busy ≔ FALSE ›
∘    act3:    Emergency_Prog≔FALSE ›
∘    act4:    ELanding_Permission(selfP)≔FALSE ›
```

Figure 8. Functional override (Event-B)

3. Event-B uses Unicode character (U+2254) for assignment which visualizes similar to colon equal in UPPAAL but they are not the same. Each platform supports different characters and UL does not support Event-B ones, generally. Hence, "≔" is replaced with ": =" or "=".



```
THEN
∘    enter_At_Gate:   Airport_State0(selfP) ≔ At_Gate ›
∘    act1:    Gate_assigned(selfP)≔FALSE ›
∘    act2:    Runway_Busy ≔ FALSE ›
∘    act3:    Emergency_Prog≔FALSE ›
∘    act4:    ELanding_Permission(selfP)≔FALSE ›
```

Figure 9. Replaceable keyword in Events

4. Relation overriding in an Event-B action (X <+ Y or X $\Leftarrow$ Y) would be transformed directly to X = Y in assignment of UPPAAL.

4.2.9 Event-B Mathematical Notation and Transformation to UTA

The complex mathematical operations in Event-B models can be implemented as C-functions in UTA. Some of the most prominent complex operations are the following below. Assume a relation regarding the favorite cities of some persons:

favoriteCities = {Khushboo↦Paris, Naveed↦Toronto, Shobit↦Venice, Jubril↦Makkah, Nadeem ↦Sydney}

1. The domain of the relation is the set of the first of all pairs in the relation and written as dom(favoriteCities) = {Khushboo, Naveed, Shobit, Jubril, Nadeem}
2. The range is quite similar to the domain, being the set of the second of all pairs in the relation and written as ran(favoriteCities) = {Paris, Toronto, Venice, Makkah, Sydney}

3. Ordered pairs ($A \mapsto B$) is a relation set which represents relation between elements of sets and written as (Naveed↦Toronto). One of the closest transformation to UTA would be two-dimensional arrays.
4. Subset ($A \subseteq B$) A is a subset of a set B which means all elements of A are also elements of B. Both sets can be equal as well. For instance, set {Toronto, Venice} is subset of set {Paris, Toronto, Venice, Makkah, Sydney}
5. Minimum/Maximum bounds of a set X are denoted with Min(X)/Max(X).
6. The inverse ($r^{\sim}$) relation operation reverses the function and the mathematical representation is (r^{-1}). For instance, f(y) = x, after inverse (f^{-1}), f(x) = y for instance favoriteCities = {Khushboo↦Paris} inverse relation would be favoriteCities = {Paris ↦ Khushboo}

The above-mentioned set theory operations are used to manipulate the sets in Event-B in one way or another. The equivalent to sets in UTA are arrays and there may be different ways to achieve the same functionality as in Event-B. The ideal way would be to use arrays in UTA and to perform array operations, functions must be defined with a combination of multiple steps which is not possible in single statements. For instance, in order to perform Min/Max operation in UTA there is no such direct operation in UTA as Event-B has. A loop within a C-like function needs to be created and a sorting algorithm to be applied to find Min/Max. Hence, these Event-B complex operations will be considered as C-like functions in UTA. Below are some more set theory operations with practical examples for Domain/Range Restriction and Subtraction.

7. Domain Restriction (S◁r) is a subset that contains all pairs of r where the first element is in S. For instance, regarding the example relation favoriteCities:
   {Shobit, Jubril}◁ favoriteCities = {Shobit↦Venice, Jubril↦Makkah }
   In UTA a C-like function is needed to implement it.
8. Domain Subtraction (S⩤r) is a subset that contains all pairs of r where the first element is not in S. For instance, regarding the example relation favoriteCities: {Shobit, Jubril} ⩤ favoriteCities = {Khushboo↦Paris, Nadeem↦Sydney}
   To achieve this in UPPAAL, a function has to be defined and used in the guard or assignment respectively.
   For example, there is an event “Landing_HighPQ” in airport control system of [1] in Event-B which contains an action “High_Pqueue ≔{selfP}⩤ High_Pqueue” for domain subtraction of finite set of planes “{selfP}” from “High_Pqueue”, i.e., removing an airplane from the high priority queue when finally it can land. An equivalent functionality has been accomplished in UTA by implementing a C function as shown below.

```
void deHPqueue()   // normal dequeue happens from High priority Queue
{
    int i = 0;
    --lenH;
    while (i < lenH)
    {
        HPqueue[i] = HPqueue[i + 1];
        i++;
    }
    HPqueue[i] = 0;
}
```

9. Range Restriction (r▷S) is a subset that contains all pairs of r where the second element is in S. For instance, regarding the example relation favoriteCities: favoriteCities ▷ {Venice, Paris} = {Shobit↦Venice, Khushboo↦Paris}
10. Range Subtraction (r⩥S) is a subset that contains all pairs of r where the second element is not in S. For instance, regarding the example relation favoriteCities:
    favoriteCities ⩥ {Shobit, Jubril} = {Khushboo↦Paris, Naveed↦Toronto,
    Nadeem↦Sydney}.
    Direct transformations for range restriction/subtraction are not possible from Event-B to UTA. Therefore, this complex operation is implemented with a C-like function in UTA.
11. Union of two sets X and Y, denoted by (X∪Y) is the set containing all elements of both sets that are either in X or in Y or in both X and Y. In UTA, to perform union operation a merge algorithm is needed to merge the two arrays corresponding to the two sets.
12. Intersection of two sets X and Y, denoted by (X∩Y) is the set containing all elements of A that also belong to B. In UTA, in order to find intersection of two sets, we initialize a set to hold a common element and use a search algorithm for matching the elements from both sets.

In general, set theory is represented in semantical form and to represent the same in a programming language, some form of enumeration must be used. For example, the supported enumeration in UTA is array which can contain multiple items and these items can be looped through using custom logic to perform some set theory concept. Statement numbers 13 to 23 below show additional mathematical operation of Event-B.

13. Partial functions: (⇸ Rightwards Arrow with Vertical Stroke) f ∈ X ⇸ Y, here f is a function with many to one relation. It is defined for some values in its domain. As Figure 10 shows below one and more than one are elements of domain X are mapped with one element of Y.

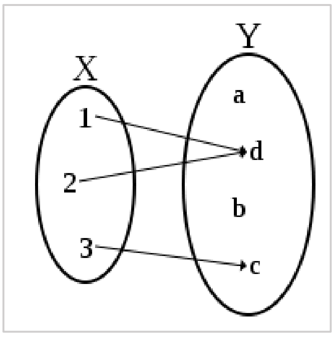


Figure 10. Partial function

14. Relational image: r[S] is the set of values related to all elements of s under the relation r.
15. Partial injections: (⤔ Rightwards Arrow with Tail with Vertical Stroke X ⤔ Y) One-to-one relations.

Suppose A= {1,2,3} and B= {r, s, t, u, v} and

f(1)=s	g(1)=r
f(2)=t	g(2)=t
f(3)=r	g(3)=r

Here f is injective, and g is not injective.

16. Total injections: Rightwards Arrow with Tail: (X ↣ Y). It is similar to partial injection but all elements are covered.
17. Partial surjections: (S ⤀ T) Rightwards Two-Headed Arrow with Vertical Stroke. Every element of T has some element of S. It can also hold one to many mapping that why it also called onto function.

    Suppose f: A⤀ B and g: B⤀ C are surjective functions.

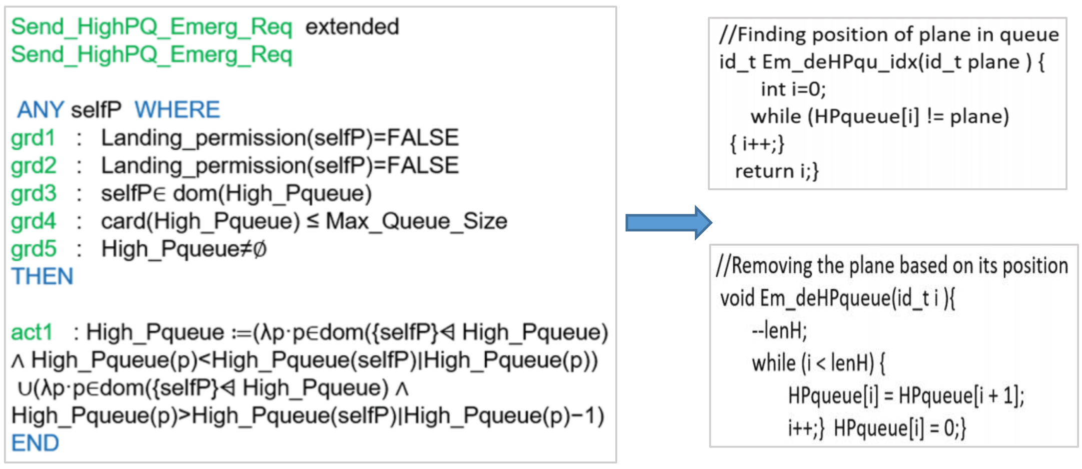
    Then g∘f: A⤀C is surjective also.
18. Total surjections: S ↠ T It is similar to partial injection but all elements from domain are covered also.
19. Bijections: (S ⤖ T) Rightwards Two-Headed Arrow with Tail. A function can be bijection if and only if it is both an injection and a surjection. A bijection is also called a one-to-one correspondence. Every element of S perfectly mapped to every element of T.

    If A= {1, 2, 3, 4} and B= {r, s, t, u}, then

f(1)=u,	f(2)=r,	f(3)=t,	f(4)=s,

Here f is a bijection.

20. Direct product: Circled Times (p ⊗ q) it is an operation that takes two sets and constructs a new set.
21. Parallel product: (Parallel to) (p ‖ q) it represents the reciprocal value of elements.
22. Lambda (λ) abstraction is an anonymous function and it gets the name from usual notation.
23. λp·P | E - P must constrain the variables in p. Below is an example of transformation of Lambda abstraction from Event-B to UTA from the airport control system of [1].



```
Send_HighPQ_Emerg_Req  extended
Send_HighPQ_Emerg_Req

 ANY selfP  WHERE
grd1   :   Landing_permission(selfP)=FALSE
grd2   :   Landing_permission(selfP)=FALSE
grd3   :   selfP∈ dom(High_Pqueue)
grd4   :   card(High_Pqueue) ≤ Max_Queue_Size
grd5   :   High_Pqueue≠∅
THEN

act1   : High_Pqueue :=(λp·p∈dom({selfP}⩤ High_Pqueue)
∧ High_Pqueue(p)<High_Pqueue(selfP)|High_Pqueue(p))
 ∪(λp·p∈dom({selfP}⩤ High_Pqueue) ∧
High_Pqueue(p)>High_Pqueue(selfP)|High_Pqueue(p)−1)
END
```

```
//Finding position of plane in queue
id_t Em_deHPqu_idx(id_t plane ) {
     int i=0;
    while (HPqueue[i] != plane)
 { i++;}
  return i;}
```

```
//Removing the plane based on its position
void Em_deHPqueue(id_t i ){
     --lenH;
     while (i < lenH) {
          HPqueue[i] = HPqueue[i + 1];
          i++;}  HPqueue[i] = 0;}
```

Figure 11. The transformation of Event-B Lambda abstraction (left) to UTA C-like functions (right) with FIFO queues [1]

In left side, action "act1" of event name "Send_HighPQ_Emerg_Req" contains lambda abstraction and in right side, it transforms into C-like function in UTA, upper function

“Em_deHP_qu_idx” is declared in guard and lower function with name “Em_deHP_queue” is declared in update of the edge.

Table 1 below shows the Event-B keywords we have discussed in this section and their correspondence in UTA terms.

Table 1. Event-B keywords and their relation to UTA

Event-B	**UTA**
isin_(X)	X is Transition Source
enter_(Y)	Y is Transition Target
States	Locations
Transition Label	Transition comment
File name	Template name/ File name
File name	File name
Event	Transition
Guard	Guard
Action	Update
$\mathbb{Z}$ / ∅	Integer / Empty set or array
ScVariable	Variable
≔ / (Assignment)	:= / (Update)
= / (Guard) / ≠	== / (Guard) / !=
⊥ / ¬ / ⇒	False / Negation / Imply
CAPITAL keywords (TRUE, FALSE)	Small keywords (true, false)

4.3 Tools and algorithm

When Event-B machine is created under Event-B core plugin in Rodin platform and it requires at least Java 1.6 and manipulates this machine into these file extensions.

Table 2. Rodin File Types

Files extension	**Root Element Type**	**Content**
.bum	IMachineRoot	Event-B Machine
.buc	IContextRoot	Event-B Context
.bcm	ISCMachineRoot	Event-B Statically Checked Machine
.bcc	ISCContextRoot	Event-B Statically Checked Context
.bpo	IPORoot	Event-B Proof Obligations
.bpr	IPRRoot	Event-B Proofs
.bps	IPSRoot	Event-B Proof Statuses

In order to create State-machine Diagram Layout, iUML-B State machines plug-in is used which save State-machine Diagram as (.sdm) file extension. Event-B tool use XML file for Event-B specification as (.bcm) file.

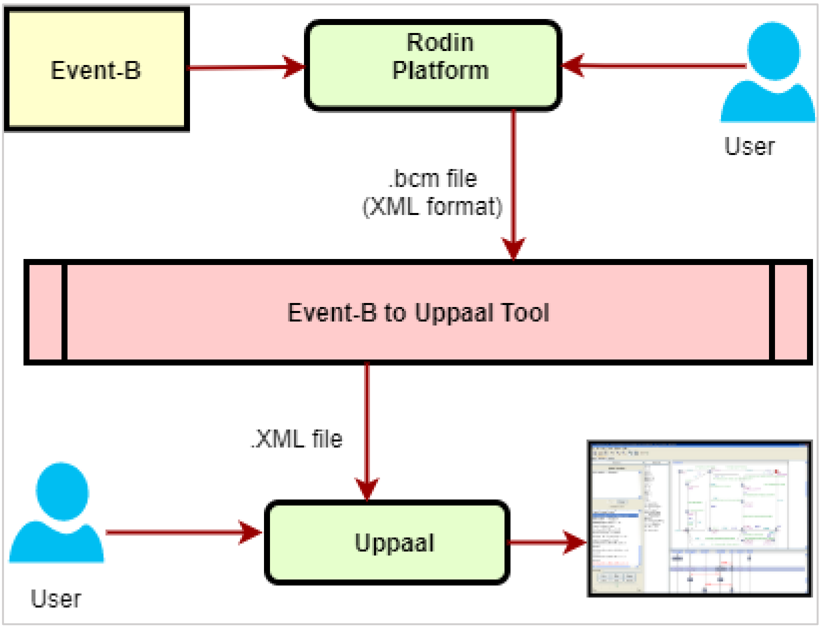


Figure 12. Overview of the transformation

This algorithm is written in C# programming language in form of window form application.

5 Case Study

During the initial stages of the work for this thesis the airport control system case study from [1] was used in order to understand the mapping and test the early model transformations implemented for the mapping tool. For the main development phase of the tool implementation the European Rail Traffic Management System (ERTMS) case study of [15] was used.

The authors of [15] demonstrated a systematic modelling of the safety-critical ERTMS system with iUML-B state machines and class diagrams following the refinement-based approach. ERTMS controls the train movement on a linear track. The Virtual Block Detector (VBD) provides locations to trains and tracks them virtually. All the movements of trains are managed by the Radio Block Centre (RBC). VBD provides information of free track section to RBC. Movement Authority (MA) issued by RBC permits the movement of trains.

The model for this case study consists of two parts: 1) The system under test which is composed of RBC and VBD, 2) the environment consisting of trackside equipment and the trains. The trackside equipment consists of Virtual Sub Sections (VSS) and the Trackside Train Detection (TDD). In other words, VSS is the track sections and TDD is the trackside train detection for the group of VSS.

Trains and tracksides report locations to VBD and VBD informs RBC about the free track sections. MAs by RBC are then allocated to the trains for the sections of the tracks that they can move into. Trains that are connected to VBD, send the following information: i) train current position, ii) length of the train, iii) and the train confirmation as integral or not. Trains are separated into three categories: *controlled* (with guaranteed free track sections), *trusted* (possible free track sections) and the ghost trains that are not communicating for any reason. If the train is not communicating with VBD, then VBD will consider that train is still respecting the MA authorisation and is still in mission.

In total eight refinement steps were performed. To exemplify the tool-based mapping we focus on the fifth refinement step. Figure 13. shows the iUML-B state machine for the train component. Together with train's local events, additional interface events are shown which are the events for receiving and sending information from/to VBD and RBC. These interface events together with other events for the controlling by the VBD and its communication with the other components of the overall system, they are not part of any other state machine diagram, rather, they are only depicted in a class diagram to visualise component dependencies (see Figure 14). The specification of all these events (including train's events) is written in the usual Event-B way in Rodin platform.

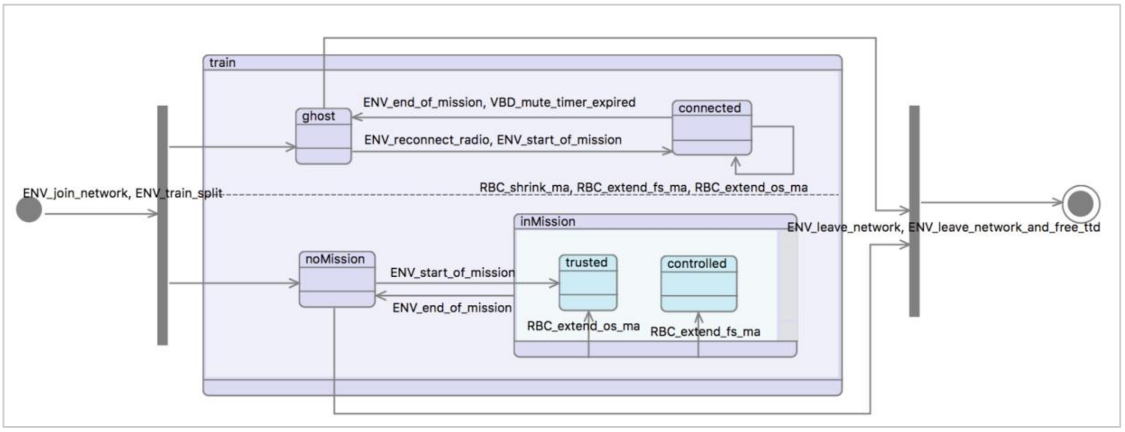

train
ghost
connected
ENV_end_of_mission, VBD_mute_timer_expired
ENV_reconnect_radio, ENV_start_of_mission
RBC_shrink_ma, RBC_extend_fs_ma, RBC_extend_os_ma
ENV_join_network, ENV_train_split
inMission
noMission
ENV_start_of_mission
ENV_end_of_mission
trusted
controlled
RBC_extend_os_ma
RBC_extend_fs_ma
ENV_leave_network, ENV_leave_network_and_free_ttd

Figure 13. The state machine of train component of 5th refinement step of ERTMS [15]

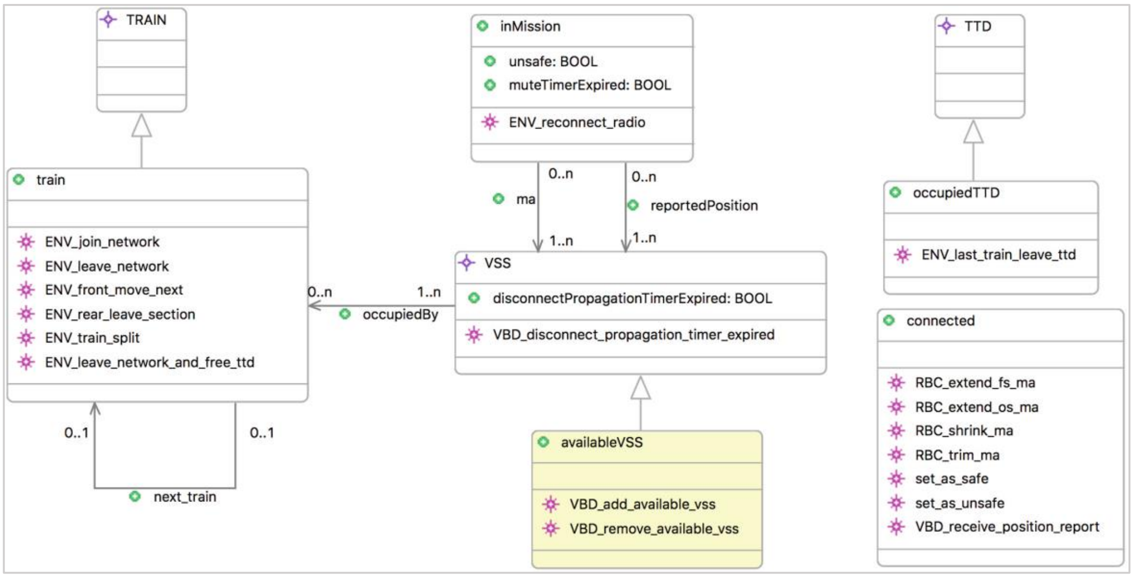


Figure 14. Class diagram representing dynamic aspects of the ERTMS environment [15]

When applying the developed mapping tool to generate the UTA model from the iUML-B/Event-B model, if a component is modelled with a iUML-B state machine then this state machine is transformed to an explicit UTA template with similar model skeleton adhering to the mapping propositions in Section 4. The Train UTA template is presented in Figure 15. The close visual resemblance to the original state machine is clear. State names from the state machine are the corresponding location names in UTA template. Transition/event names are translated to edge names in comments in the corresponding edges. Since this 5th refinement step of ERTMS already includes a lot of details modelled with various complex mathematical operations in guards and actions, most of the guards and updates include C function placeholders indicating where the modeller needs to implement the C functions corresponding to the Event-B mathematical operations. The function placeholder name is composed of a prefix “fn_” followed by the variable name of the complex guard or assignment. Moreover, the tool adds in comment lines the relevant original Event-B specification for assisting in what has to be implemented in the function bodies.

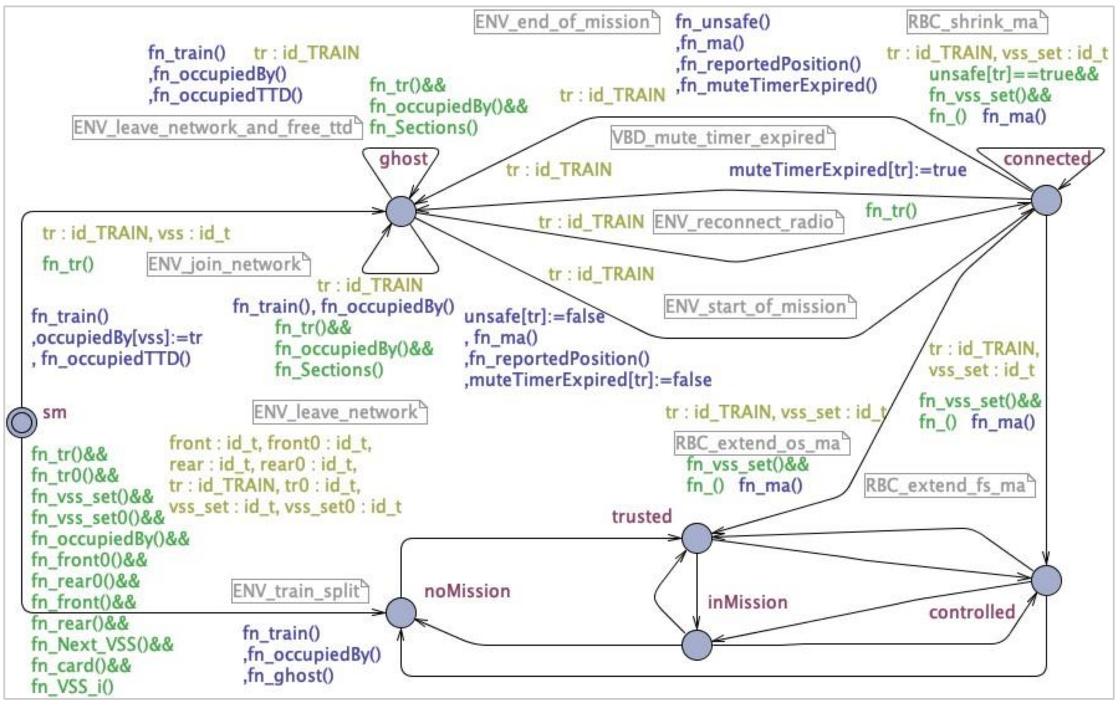
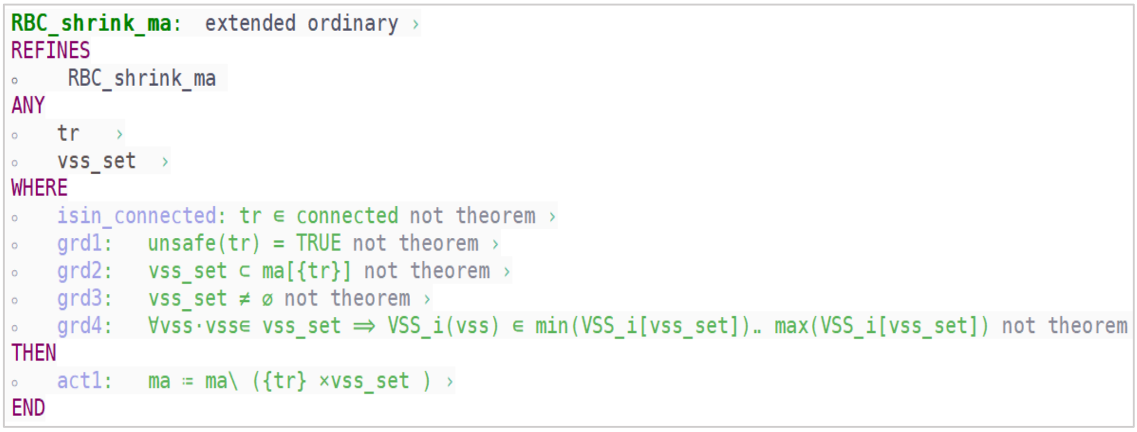

ENV_end_of_mission
fn_unsafe() ,fn_ma() ,fn_reportedPosition() ,fn_muteTimerExpired()
RBC_shrink_ma
tr : id_TRAIN, vss_set : id_t
unsafe[tr]==true&& fn_vss_set()&& fn_() fn_ma()
fn_train() ,fn_occupiedBy() ,fn_occupiedTTD()
tr : id_TRAIN
ENV_leave_network_and_free_ttd
fn_tr()&& fn_occupiedBy()&& fn_Sections()
tr : id_TRAIN
VBD_mute_timer_expired
ghost
connected
tr : id_TRAIN
muteTimerExpired[tr]:=true
fn_tr()
tr : id_TRAIN
ENV_reconnect_radio
tr : id_TRAIN, vss : id_t
fn_tr()
ENV_join_network
tr : id_TRAIN
tr : id_TRAIN
fn_train(), fn_occupiedBy()
fn_tr()&& fn_occupiedBy()&& fn_Sections()
ENV_start_of_mission
fn_train() ,occupiedBy[vss]:=tr , fn_occupiedTTD()
unsafe[tr]:=false , fn_ma() ,fn_reportedPosition() ,muteTimerExpired[tr]:=false
tr : id_TRAIN, vss_set : id_t
fn_vss_set()&& fn_() fn_ma()
sm
ENV_leave_network
tr : id_TRAIN, vss_set : id_t
RBC_extend_os_ma
fn_vss_set()&& fn_() fn_ma()
RBC_extend_fs_ma
fn_tr()&& fn_tr0()&& fn_vss_set()&& fn_vss_set0()&& fn_occupiedBy()&& fn_front0()&& fn_rear0()&& fn_front()&& fn_rear()&& fn_Next_VSS()&& fn_card()&& fn_VSS_i()
front : id_t, front0 : id_t, rear : id_t, rear0 : id_t, tr : id_TRAIN, tr0 : id_t, vss_set : id_t, vss_set0 : id_t
trusted
ENV_train_split
noMission
inMission
controlled
fn_train() ,fn_occupiedBy() ,fn_ghost()

Figure 15. Mapped ERTMS Train state machine to UTA[1]

Let us exemplify with the mapping of a single full event. Figure 16. shows the Event-B specification for event “RBC_shrink_ma”. The **ANY** clause in the specification contains two parameters “tr” and “vss_set” which are transformed into SELECT non-deterministic choice parameters on the transition in UTA. As per statement 2 of Section 4.2.5 above,

[1] https://bitbucket.org/shobitjain/thesis-eventb-to-uppaal/src/master/Generated_ERTMS_UPPAAL_Model.xml

because this event contains only “isin_” in one of the expressions in the **WHERE** clause, the transition is applied as a self-loop transition and the rest of the expressions are added as guards on this transition. Simple guards are directly translated to UTA and for the complex expressions, a function placeholder is added instead which will contain the logic for it, as stated above. Similarly, In **THEN** clause, due to the complex nature expression of the action, it is translated in UPPAAL using a placeholder for the C-like function to be finally implemented.



```
RBC_shrink_ma:  extended ordinary ›
REFINES
◦    RBC_shrink_ma
ANY
◦   tr   ›
◦   vss_set  ›
WHERE
◦   isin_connected: tr ∈ connected not theorem ›
◦   grd1:   unsafe(tr) = TRUE not theorem ›
◦   grd2:   vss_set ⊂ ma[{tr}] not theorem ›
◦   grd3:   vss_set ≠ ∅ not theorem ›
◦   grd4:   ∀vss·vss∈ vss_set ⇒ VSS_i(vss) ∈ min(VSS_i[vss_set])‥ max(VSS_i[vss_set]) not theorem
THEN
◦   act1:   ma ≔ ma\ ({tr} ×vss_set ) ›
END
```

Figure 16. Event RBC_shrink_ma in Event-B

Figure 17. shows corresponding representation of the above mentioned event in iUML-B and UPPAAL. iUML-B model contains the event and the state “connected” on which the event “RBC_shrink_ma” is applied. The same state (location) is generated in UPPAAL with the same transition and the properties applied on the transition are based on the specification mentioned for Figure 16.

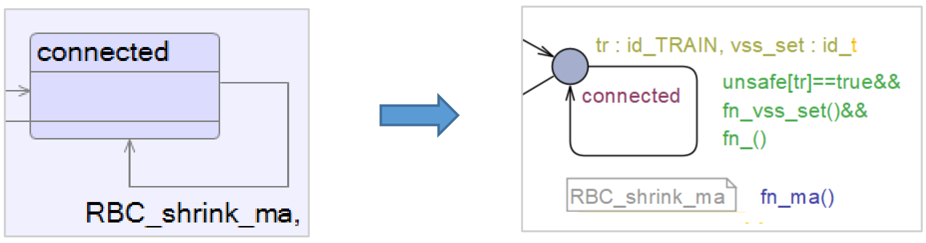


Figure 17. iUML-B (left) and UPPAAL (right) model for event RBC_shrink_ma

Regarding the rest of the ERTMS events that they do not belong to an iUML-B state machine, the mapping tool groups them in a separated UTA template as shown in Figure 18. Since the enabling conditions of these events are mutual exclusive, the events can be modelled as different local self-loop transitions on the same location. This mapping decision conforms also what was proposed in [2] regarding mapping of pure Event-B events not supported by any iUML-B diagram.

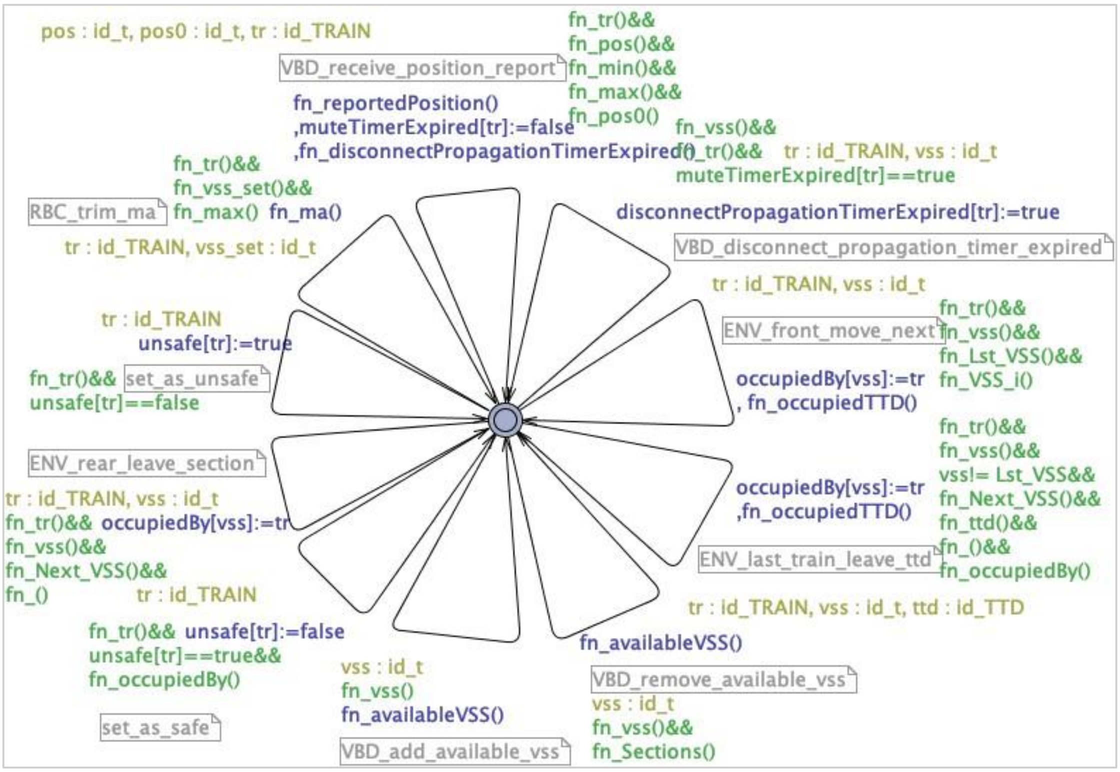


Figure 18. Tool generated ERTMS controller[1]

6 Validation of results

The first stage for validation of the results is based on visual inspection comparing the original Event-B/iUML-B model against the generated UTA model. Regarding the ERTMS case study, this stage confirms that the generated UTA model is for the biggest part as expected. For full validation some manual steps need to be still performed concerning the completion of the UTA model. The C-like functions have to be implemented, the separate templates need to be synchronized fully with any common global variables and additionally with synchronization channels and any real-time information needs to be added to the model to allow the verification of the timing properties of the system. After these manual steps the UTA model can be simulated (and system timing properties can be model-checked) and its traces can be compared with the simulation traces generated by the ProB animator [16] for the Event-B model allowing for complete validation of the correctness of the mapping.

7 Conclusion

In this thesis we demonstrated the implementation decisions for a tool for transforming Event-B/iUMLB models to UTA. Our goal was to establish an automated mapping process enabling the combination of the two formalisms to mutually complement each other for the formal development of complex systems. Some steps need to be still manually performed by the modeller to finalize the UTA model but, nevertheless, a complete and structured skeleton for the UTA model is automatically generated together with all necessary information to finalize the model.

8 Future work

Although the provided assumptions, findings and implementation of the tool are generic enough for transforming Event-B models to UTA, there are several improvements that can still be made. One direction is to take into account also the iUML-B class diagrams to structure in even better way the generated UTA templates. For example, by investigating closer the class diagram in Figure 14 a better decomposition of the system components could be applied instead of having just two UTA templates. Another future work is about automating the implementation of at least some of the C-like functions to model in UTA the Event-B complex guards and assignments. This would reduce the amount of manual work required to complete the UTA model. Furthermore, the tool functionality can be extended to be able to infer refined variable declarations and other gluing information between refinement levels in Event-B side of the modelling process. Currently the tool transforms individual refinement levels and the modeller is then responsible to add missing information from previous refinement steps.

References

[1] F. Shokri-Manninen, L. Tsiopoulos, J. Vain, and M. Waldén, “Integration of iUML-B and UPPAAL Timed Automata for Development of Real-Time Systems with Concurrent Processes,” in *Rigorous State-Based Methods*, Cham, 2020, pp. 186–202, doi: 10.1007/978-3-030-48077-6_13.

[2] J. Vain, L. Tsiopoulos, and P. Boström, “Integrating Refinement-Based Methods for Developing Timed Systems,” 2016, doi: 10.1201/b20053-17.

[3] D. Cansell, D. Méry, and J. Rehm, “Time Constraint Patterns for Event B Development,” in *B 2007: Formal Specification and Development in B,* Berlin, Heidelberg, 2006, pp. 140–154, doi: 10.1007/11955757_13.

[4] C. Zhu, M. Butler, and C. Cirstea, “Refinement of timing constraints for concurrent tasks with scheduling,” in *Abstract State Machines, Alloy, B, TLA, VDM, and Z: ABZ 2018,* May 2018, vol. 10817, pp. 219–233, Accessed: Jan. 02, 2021. [Online]. Available: https://eprints.soton.ac.uk/419024/.

[5] C. Zhu, M. Butler, and C. Cirstea, “Semantics of real-time trigger-response properties in Event-B,” in *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE),* Aug. 2018, pp. 150–156, doi: 10.1109/TASE.2018.00028.

[6] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge: Cambridge University Press, 2010.

[7] J. Berthing, P. Boström, K. Sere, L. Tsiopoulos, and J. Vain, “Refinement-Based Development of Timed Systems,” in *Integrated Formal Methods,* Berlin, Heidelberg, 2012, pp. 69–83, doi: 10.1007/978-3-642-30729-4_6.

[8] M. Garoui, B. Mazigh, and A. Koukam, “The EventB2PN Tool: From Event-B specification to Petri Nets through model transformation,” in *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD),* Jun. 2015, pp. 1–7, doi: 10.1109/SNPD.2015.7176278.

[9] A. Furfaro and L. Nigro, “Modelling and Schedulability Analysis of Real-time Sequence Patterns using Time Petri Nets and Uppaal,” 2007.

[10] A. Fürst, T. S. Hoang, D. Basin, K. Desai, N. Sato, and K. Miyazaki, “Code Generation for Event-B,” Cham, 2014, pp. 323–338.

[11] M. Jastram and P. M. Butler, *Rodin User's Handbook: Covers Rodin v.2.8*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2014.

[12] M. Y. Said, M. Butler, and C. Snook, "A method of refinement in UML-B," *Softw. Syst. Model.*, vol. 14, no. 4, pp. 1557–1580, Oct. 2015, doi: 10.1007/s10270-013-0391-z.

[13] T. S. Hoang, "An Introduction to the Event-B Modelling Method," in *Industrial deployment of system engineering methods*, Springer, 2013, pp. 211–236.

[14] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on Uppaal," in *Formal Methods for the Design of Real-Time Systems*, vol. 3185, M. Bernardo and F. Corradini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 200–236.

[15] D. Dghaym, M. Poppleton, and C. Snook, "Diagram-Led Formal Modelling Using iUML-B for Hybrid ERTMS Level 3," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Cham, 2018, pp. 338–352, doi: 10.1007/978-3-319-91271-4_23.

[16] J. Bendiposto, M. Lescheul, O. Ligot, and M. Samia, "La validation de modèles Event-B avec le plug-in ProB pour RODIN," *Tech. Sci. Inform.*, vol. 27, no. 8, pp. 1065–1084, Oct. 2008, doi: 10.3166/tsi.27.1065-1084.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Shobit jain

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Implementation of model transformations between Event-B and UPPAAL Timed Automata”, supervised by Leonidas Tsiopoulos.
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

05.01.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.