TIMING DIAGRAMS REQUIREMENTS MODELING USING EVENT-B FORMAL METHODS

Tossaporn Joochim, Colin F. Snook, Michael R. Poppleton and Andy M. Gravell
School of Electronic and Computer Science,
University of Southampton,
United Kingdom
{tj04r, cfs, mrp, amg}@ecs.soton.ac.uk

ABSTRACT
Timing diagrams provide an intuitive graphical specification for time constraints and causal dependencies between a system's objects. Such a view can provide useful insight during Requirements Engineering (RE). Formal Modeling techniques allow abstract system level models to be explored in revealing detail and provide feedback via verification and validation methods such as proofs of consistency, model checking and animation. Here, we bring these two modelling approaches together. In particular we present techniques to extend a graphical modeling capability for formal modeling into the real-time domain by developing a Timing diagram view for the Event-B formal method and its graphical front-end, UML-B. Translation schemes to Event-B and UML-B are proposed and presented. A case study of a lift system is used to demonstrate the translation in practice.

KEY WORDS
Visual and Formal modeling, Timing diagram, Event-B, UML-B

1. Introduction

Event-B [1] is a formal language for state-based modeling and verification of systems. An extensible, open-source platform for Event-B modelling and verification has been developed in the context of RODIN [2], a European IST project. Event-B enables a precise system requirements specification to be developed and verified using set-theoretic notation. However, Event-B has no explicit support for specifying timing constraints and causal dependencies within system requirements. Moreover, using FMs, such as Event-B, can involve a costly learning curve as effective modelling requires training of engineers [3]. Research [4] shows that this learning curve is easier to overcome if more intuitive graphical modelling interfaces are used. The UML-B [5] is a UML-like graphical front-end for Event-B. The UML-B uses Class diagrams and Statemachines to generate system specification models. Additional textual guards, actions and invariants can be added in the Event-B notation. The UML-B models are then automatically translated into Event-B using the U2B translator where the Rodin verification tools perform static (type and syntax) checking and any problems are annotated back on the UML-B diagrams

To some extent, the requirements of timing constraints and causal dependencies among system's objects can be expressed using Statemachines in UML-B. However, there are limitations to this approach. For instance, in general, causal interactions will exist between the transitions of several statemachines and cannot be seen on one diagram or view making them hard to visualise. It is not helpful for the users in term of modeling. In a Timing diagram (TD), we can describe the causality explicitly in the arrows between events and have them all in the same screen. The TD notations include graphically described timing constraints. It is very natural to form expressions as timing constraints.

Our contribution focuses on how the parts of a system's requirements concerned with timing constraints and causal dependencies between a system's objects, are generated into FM models. That is, adding to Event-B and UML-B the ability to express timing constraints and event dependency requirements. A lift system based on Jackson's work [6] (in which there is one lift in the system) is used as a primitive case study. The case study has been modified by adding timing constraints to event dependency requirements to demonstrate the issue of this paper. The TD we use is a variation of the OMG's TD [7]. It has been amended to include features that simplify translation into Event-B.

There exist TD editors such as TimeGen [8], TimingTool [9], and SynaptiCAD [10]. In general, those editors are engineering computer-aided design (CAD) software tool that helps draw Timing diagrams. The output Timing diagrams are then exported to other applications, such as PDF, HTML and GIF for use in writing design specifications. Although these editors can identify clock and use arrows to show a cause and effect, they do not fit with our research since they are not suitable to define simultaneous events nor complex combinations of causal relationships as in our TD. Moreover, they are not written

on the Eclipse framework making tooling more difficult to achieve. Thus, they would not easily fit as a plugin extension of RODIN and UML-B.

The rest of the paper is structured as follows. Section 2 gives an overview of the research. Section 3 presents transformation rules for TD to Event-B and UML-B. Section 4 compares and evaluates the technique for the two translations techniques. Finally, section 5 concludes our work and defines future research directions.

2. Overview and Background

2.1 Contribution Overview

We show how to generate Event-B models from TD as shown in Fig. 1. Requirements are partitioned into other requirements (non-timing), and timing and causal dependency requirements. The causal dependency and timing constraints requirements are modeled by TD. As a precursor to tool development, rules were specified for mapping TD to Event-B in order to gain experience and to formalise the definition of TD. The rules were based on TD Backus-Nuar Form (BNF) definitions that we have generated. Moving towards practical tool development we transform TD to UML-B using a model-to-model transformation written in Atlas Transformation Language (ATL) [11]. Other requirements are used to manually add the remainder of the Event-B and UML-B models for completion. Next, Event-B and UML-B models are analysed/verified by the RODIN Toolkit. Any errors may lead to revisions of the system requirements, the TD models or the manually added parts of the models depending on the source of the problem.

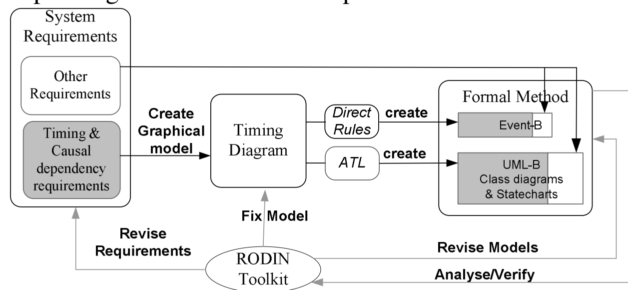


Fig. 1. Contribution Overview

This step is repeated until the models are correct by means of proof. This process has a beneficial effect on system requirements as it increases the degree of confidence that the output system has few errors, is unambiguous and consistent.

2.2 Timing Diagram

Timing diagrams (TDs) model changes to an object's state through time. Our TD notations are based on the UML Robust TD notations [7] which shows the state of each object on the Y-axis while timing constraints are on the X-axis. A subset of the UML TD notation is selected and some notations are adapted so that they are more suitable for generating expressions in Event-B.

To illustrate our TD notation, we use the lift system example described below and its corresponding TD shown in Fig. 2.
"The relation between lift movement and the floor sensors is: whenever a user presses a button to request a lift, the lift starts moving departing up (a) or departing down (b) from the current floor. Within between 2-5 seconds after the lift starts moving departing up/down, the current floor sensor will turn off. At the same point of time, if the lift starts moving departing up say, the up lamp changes its status to activate (d) while the down lamp changes its status to deactivate (c)"

In our TD for this example, we show four `Timelines` which represent the state changes in time for the objects: *floorsensor*, *lift*, *uplamp*, and *downlamp*, belonging to classes *FLOORSENSOR*, *LIFT*, *UPLAMP* and *DOWNLAMP* respectively. The solid arrowed line (called `CauseAffectArrow`) represents the cause-effect relationships among objects. A compound cause-effect can be specified using ○ and ● symbols which represent disjunction and conjunction respectively.

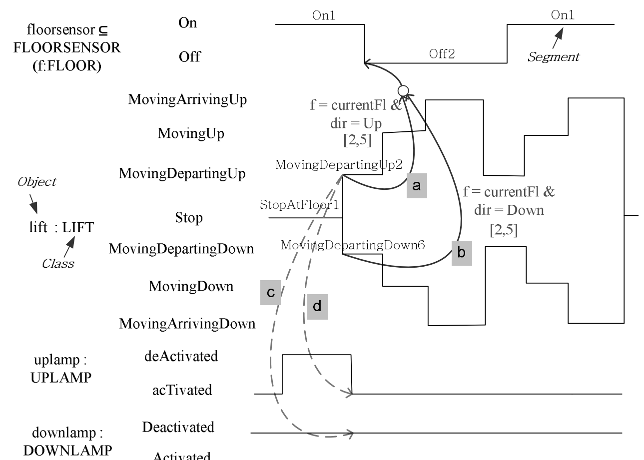


Fig. 2. A lift Timing diagram

In Fig. 2, the *floorsensor Off2* segment has a `CauseEffectArrow` that consists of a disjunctive combination of two `CauseEffectArrow`. This means the *floorsensor* is set to *Off* if the *lift* is in either the state *MovingDepartingUp* or *MovingDepartingDown*. Predicates such as *f = currentFl & dir = Up* are additional conditions on the `CauseEffectArrow` where *f* represents a floor and is a dynamic state parameter that can change in time. Here, *f* is also the object index for class *FLOORSENSOR*.The *currentFl* represents the present floor for the lift, while *dir* represents the direction of the lift. The curved, dashed-lines (c and d) represent `SimultaneityArrow`. They are used to synchronize the

liftMovingDepartingUp segment with the *uplamp* and *downlamp* objects to indicate that the occurrences of these events happen arbitrarily close to each other with no particular constraint (at this level of abstraction). A timing constraint is defined using symbols, i.e. [t1, t2] indicates the time constraint > t1 and < t2.

2.3 Event-B

An Event-B model comprises static and dynamic parts, which are called CONTEXT and MACHINE respectively. A machine SEES at least one context. The CONTEXT may contain carrier sets, constants, axioms and theorems. The MACHINE defines the behavior of the Event-B model. Machine variables are used to maintain state information while performing events. Invariants define properties over the state of the variables that must be satisfied by all events. Events define spontaneously occurring atomic units of behavior that make state changes. An event has a name and is composed of guards *G* and actions *S*. Guards identify lists of conditions for the event to occur, while actions identify how the state variables evolve when the event occurs. The general form of an event with event local variables *l* is shown in Fig. 3 (as highlighted). Note that this paper presents only an example of generating parts of an event's guard from the TD as described in section 3.1.

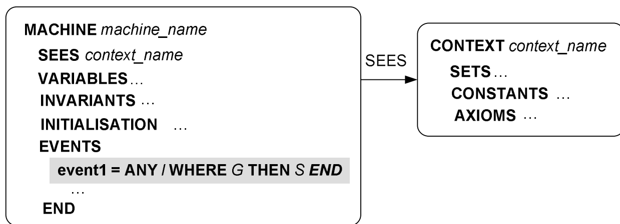
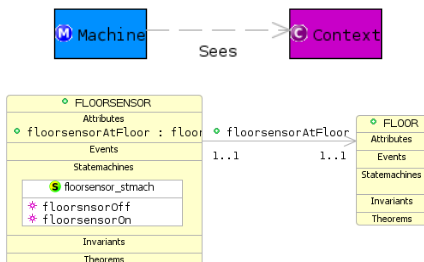


Fig. 3. The structure of an Event-B model

2.4 UML-B

UML-B is a tool that supports the construction of an Event-B graphical model. An UML-B project is contained in a package diagrams. A package diagram is used to describe the association between component, CONTEXTS and MACHINES, in an UML-B project (see Fig. 4. top for an example of a package diagram). The CONTEXTS are used to define constant data while the MACHINES are described by Class diagrams and Statemachines to represent variables and events.



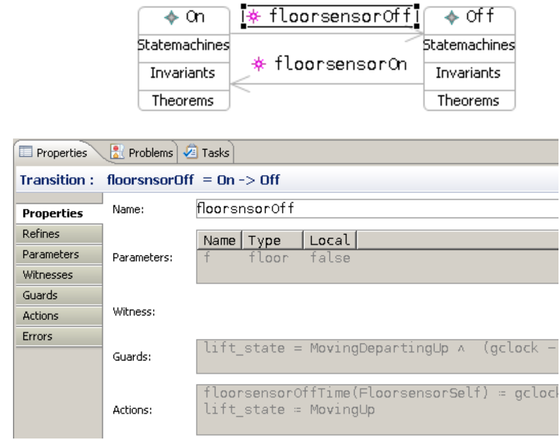


Fig. 4. UML-B floor sensor Class and Statemachine

For example, Fig. 4., there are two classes: FLOORSENSOR and FLOOR. The association floorsensorAtFloor represents the link from each floor sensor to its corresponding floor. The association's multiplicities show that each floor sensor belongs to exactly one floor and each floor has one floor sensor. The Statemachine attached to classes (as shown in Fig. 4., floorsensor_stmch) is used to describe behavior of an object in that class, in this case is the floor sensor. The Statemachine's transitions (as shown in Fig. 4., floorsensorOff) represent events in which the additional behavior associated with the change of states can be identified by using properties windows (as show in Fig. 4. bottom). UML-B models can then be automatically translated to Event-B using the U2B translator. The Event-B model is then verified by the RODIN toolkit for static checking and proof obligations (POs) are automatically generated by the RODIN tools.

2.5 ATL

ATL [11] is a model to model transformation language that was developed for the Eclipse Modelling Framework (EMF) [12]. We select ATL since UML-B is built on the same EMF and ATL provides a declarative notation to define the transformation and good tool support for generating usable tools. An ATL transformation module is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target model. An ATL module is consists of 2 parts: header and transformation rules. The header declares the source and target models used for the transformation. For example, in our work, TD is used as a source model to generate a target model, UML-B. The transformation rules express the transformation logic and provide the means for ATL developers to specify the target model elements to be generated from the source model elements. The full ATL declaration can be found at [11]. Here, we provide examples of ATL transformation rules, that are used in this research, as described in section 3.2.

3. Transformation of Timing Diagrams

We present two approaches to linking timing diagrams with Event-B. In the first approach we manually translated TD into Event-B in one step. This approach was useful as a learning stage and to provide a formal definition of TD (leveraged from the corresponding Event-B). However, for tool supported translation and verification a more pragmatic approach is to provide TDs as an extension to UML-B. UML-B already provides support for the underlying features such as classes, objects, states and transitions. In the second approach described in this section, we describe a translation from TD into UML-B.

3.1 Direct translation into Event-B

We firstly provide a definition of TD in Backus-Nuar form (BNF) and then identify translation rules using the TD BNF definition. Parts of the TD BNF are illustrated below:

Machine ::= name Class$^+$
Class ::= ClassName Obj$^+$ Obj_Def
Obj :: = ObjName ObjSt$^+$ Timeline
Timeline ::= name Segment$^+$
Segment ::= ObjSt number Simul* [CauseEffectArrow]
CauseEffectArrow ::= Constraint
Constraint ::= NodeType
NodeType ::= Simple | OR_node | AND_node
OR_node ::= NodeType ○ NodeType
AND_node ::= NodeType ● NodeType
Simple ::= CauseSegmt [Timing] [Predicate*]
Timing ::= "[" lowerlimit "," upperlimit "]"
lowerlimit ::= $\mathbb{Z}^+$, upperlimit ::= $\mathbb{Z}^+$

A TD machine has a (unique) name and one or more classes. A class consists of a name (ClassName), at least one object (Obj) and an object definition (Obj_Def). An object has a name (ObjName), at least one state (ObjSt) and a Timeline. A Timeline consists of at least one segment. A segment represents one of its owner object's states in a particular sequence of state changes and is therefore always associated with a unique state transition by which it can be reached. Hence an state value could be represented by several different segments.. For example in Fig. 2., MovingDepartingUp2 which is reached from StopAtFloor1, is a different segment from MovingDepartingUp5 which is reached from MovingArrivingUp4. A segment may own a CauseEffectArrow which defines a constraint (Constraint) on the transition to reach that segment. The constraint is defined by a type (NodeType) which can be a simple (Simple) or a grouping of either OR nodes (OR_node) or AND nodes (AND_node). Those grouping nodes allow one to create compound cause conditions. A Simple constraint consists of a cause segment (CauseSegmt), which is the source of the CauseEffectArrow, an optional timing constraint (Timing) and an optional string condition (Predicate). A timing constraint provides bounds to the time interval between the cause segment being entered and the target segment being reached. To simplify the expression of rules, we provide a set of basic 'accessor' translation rules that extract a sub-clause from a compound one. For example:

TTiming(Simple) $\longrightarrow$ Timing; this rule produces the Timing value for an input Simple.

Generally, a translation rule is composed of a sequence of basic rules which may have sub-rules. For example, in Fig. 5, we show the top-level translation rule **TEvent** that is used to iteratively create all the Event-B events in a Machine.

The rule **TEvent** uses a TD Machine for an input parameter. Segments that are the target of a SimultaneityArrow do not generate a new event. Instead they contribute actions to the event generated from the segment that is the source of the SimultaneityArrow. A top-level rule, such as **Tevent**, constructs the structure of an Event-B element using mid-level rules. The mid-level rules produce text fields by concatenating strings that are obtained from parts of the TD model using the basic accessor rules.
An Event-B event is created from 4 groups of the rules as shown in Fig. 5. The first group creates the event name from the Segment definition. The second group generates the event's local variables (parameters) and their constraints (including type information). The third group is for creating an event's guards which includes imposing the timeline sequence of the segments as well as the additional constraints of the CauseEffectArrow. The fourth group generates the event's actions which include the effect of the transition to the new segment and any simultaneous transitions.

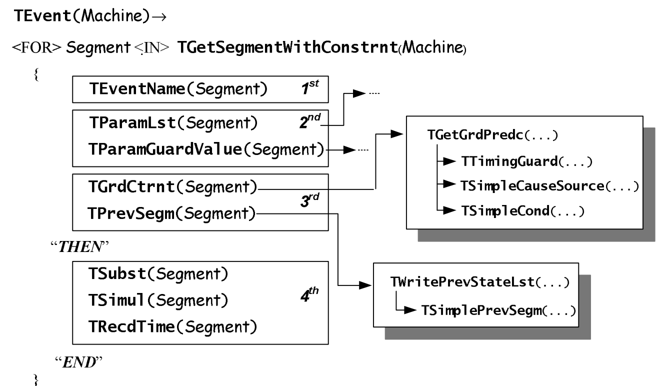


Fig. 5. Top-level translation rule to create Event-B event

Here, we show only how a rule **TTimingGuard** is used to generate an event's guard from timing constraints. Note that we do not explain how to pass parameters from the earlier processes to the rule **TTimingGuard**. This can be found in [13]. The rule **TTimingGuard** uses Simple as an

input parameter. The detail of the rule is shown below. Note that string literals are shown in italics and string concatenation is denoted by '+'. The literal *gclock* is an Event-B variable used to model the passing of time.

TTimingGuard(**TSegment**(Simple), **TTiming**(Simple)) =

TTimingGuard(Segment, Timing) ⟶

"*(gclock -* " + **TObj**(**TObjSt**(Segment))
+ **TObjSt**(Segment)) + "*Time* ≥"
+ **TLowerLmt**(Timing) +"*)*" + "*& (gclock -* "
+ **TObj**(**TObjSt**(Segment))
+ **TObjSt**(Segment))
+ "*Time* ≤" + **TUpperLmt**(Timing) + "*)*"

For example, from Fig. 2, the segment `Off2` is defined with a `CauseEffectArrow`. Thus, there is an event, floorsensorOff, generated by this segment. This `CauseEffectArrow` is a disjunction (OR_node) of two Simple constraints. Using the BNF definition for Simple (Simple ::= CauseSegmt [Timing] [Predicate*]) these are:

Simple1: MovingDepartingUp2 [2,5] f=currentFl & dir = Up,
Simple2: MovingDepartingDown6 [2,5] f=currentFl & dir = Down.

Timing guards are recursively generated by the rule **TTimingGuard**. The recursion expands compound constraint nodes (OR_node and AND_node) until a Simple constraint is reached. The simple constraint is expanded into a text string representation which is used as the guard. Thus, for the event floorsensorOff the first disjunct, Simple1, is generated as shown below.

TTimingGuard(MovingDepartingUp2, [2, 5]) ⟶
"*(gclock -* " + **TObj**(**TObjSt**(MovingDepartingUp2))
+ **TObjSt**(MovingDepartingUp2)) + "*Time* ≥"
+ **TLowerLmt**([2,5]) +"*)*" + "*& (gclock -* "
+ **TObj**(**TObjSt**(MovingDepartingUp2))
+ **TObjSt**(MovingDepartingUp2))
+ "*Time* ≤" + **TUpperLmt**([2, 5]) + "*)*"

The output is *(gclock - liftMovingDepartingUpTime ≥ 2) & (gclock - liftMovingDepartingUpTime ≤ 5).*

Fig. 6 shows the position of the timing constraints guard which is created for the event floorsensorOff.

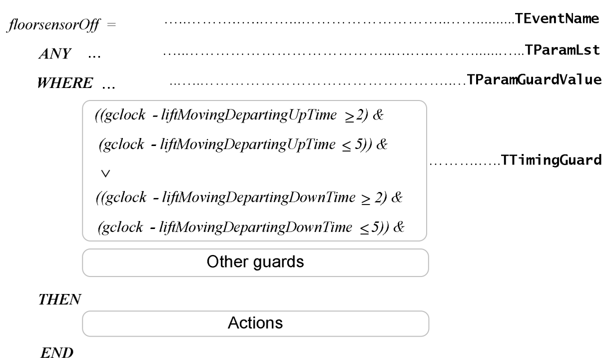


Fig. 6. A floorsensorOff event

There are some events which cannot be created by translation rules, for example, an event that changes the direction of the lift i.e. from up to down and vice versa. That is because this information cannot be represented by TD notations but it needs to be generated by hand.

3.2 Translation into UML-B

UML-B is implemented using the EMF which requires a metamodel to define the abstract syntax. Fig. 7 shows a small part of the UML-B metamodel for classes and state-machines.

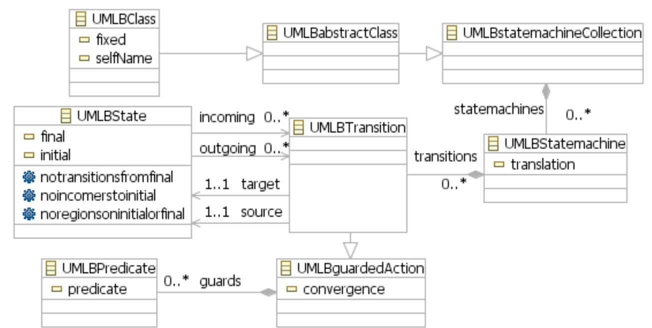


Fig. 7. Parts of UML-B Metamodel

A UML-B class may own statemachines which may own zero or more transitions and states (the latter is not shown in Fig. 7). Each transition references a source state and a target state. Each state references its incoming and outgoing transitions. Transitions are implicitly guarded by their source state and act to alter the statemachine to their target state. They may also have additional guards which are defined in a predicate string.

In order to define a model-to-model transformation we also need to express the abstract syntax of TD in a similar EMF metamodel. Some parts of the TD EMF metamodel are illustrated in Fig. 8. There are some classes in the UML-B metamodel and TD metamodel that are similar and have an obvious correspondance. For example, `UMLBClass` with `TDClass`, `UMLBStatemachine` with `TDTimeline`, and `UMLBTransition` with `TDTimelineTransition`. Other parts such as `CauseEffectArrow`, `SimultaneityArrow`, and timing constraints exist only in the TD metamodel and have no corresponding modelling concept in UML-B. These are new modelling concepts that TD provides. During translation these concepts will be translated down into textual guards in a similar way to that described for the direct translation to Event-B.

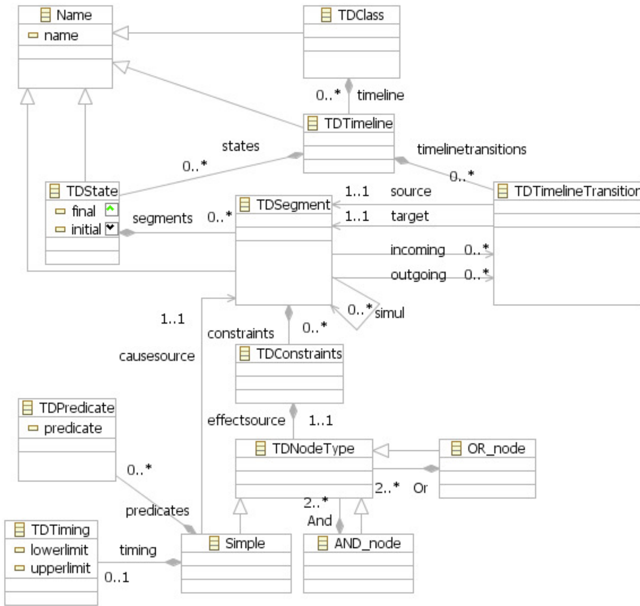


Fig. 8. Parts of Timing diagram Metamodel

Those TD meta-classes that have a correspondence with UML-B meta-classes are directly mapped using ATL rules to generate UML-B elements as shown below:

```
rule Class {
from t : TDMetamodel!TDClass
to u : umlbMetamodel!UMLBClass
   (name <- t.name,
    selfName <- t.name + 'Self',
    statemachines <- t.timeline), …
rule StateMachine {
from t : TDMetamodel!TDTimeline
to u : umlbMetamodel!UMLBStatemachine
   (name <- t.name + '_state',
    states <- t.states,
    transitions <- t.timelinetransitions)}
rule Transition {
from t : TDMetamodel!TDTimelineTransition
to u : umlbMetamodel!UMLBTransition
   (name <- t.target.getTransitionName(),
    …
```

Fig. 9. ATL rules: Class, Statemachine and Transition

From the rule, `Class`, the keyword **`rule`** is used to identify a rule name while the **`from`** and the **`to`** are used to define local names for the source element (`t : TDMetamodel!TDClass`) and for the generated target element (`u : umlbMetamodel!UMLBClass`). Properties of the target element are then defined using a syntax `target_property_name <- expression`. For example, `selfName <- t.name + 'Self'` generates a name for the local variable used to represent contextual instances of a UML-B class (represented by `selfName` in the UML-B metamodel, see Fig. 7.) The contextual instance is similar to 'this' in Java, but since UML-B has less encapsulation it is often necessary to choose unique identifiers to be used in each UML-B class. The `selfName` is constructed by concatenating the TD class name with a literal string `'Self'`. The clause, `statemachines <- t.timeline` maps the containment of TD `timeline` to the conatinament of UMLB `statemachines`. Note that this only maps the containment (ownership) features. It is also necessary to define another rule that gives the details of mapping a TD `timeline` to a UMLB `statemachine`. Similarly, the clause, `transitions <- t.timelinetransitions` links the TD `timelinetransitions` containment to the UMLB `transitions` containment.

When the TD as shown in Fig. 2 is used as a source model, the ATL rules above (with the hidden sub-rules not illustrated here) automatically generate UML-B classes, statemachines and transitions as shown in Fig. 4 (but not the associations between classes).

There are parts of the UML-B metamodel and TD metamodel that cannot be mapped so simply. For example, the UML-B Statemachine transitions' guards are generated from several TD meta-classes (i.e. `TDConstraint`, `TDNodeType`, `Simple`, `OR_node`, `AND_node`, and `TDTiming`) by the rule `Constraint`. This rule calls other sub-rules as shown below.

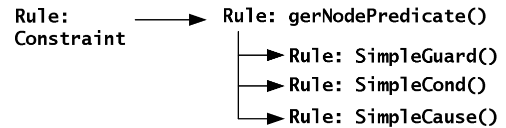
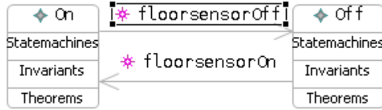


Fig. 10. Rule Constraint and sub-rules

We do not show the detail of these rules in this paper but an example of the transition guards created by the rule `Constraint` from the transition floorsensorOff is shown in Fig. 11. Notice that the figure is similar to Fig. 4. but the guard is incomplete (xAssociationx is a placeholder for additional information). TD is a partial view of a system and not designed to model the relationships between variables. Thus, there are some UML-B model features, such as associations among classes, that cannot be created from the TD model. For example, the marker xAssociationx is used as a placeholder for users to add the corresponding associations (if any) to complete the model. In this example the missing relationship might be needed to select the correct instance of Lift that is associated with the current contextual instance of FloorSensor. In the previous example of Fig. 4, we chose to model an implicit singleton Lift and therefore did not need any association.

As with the direct translation to Event-B, in UML-B, some information may need to be added. Apart from the associations among classes, the same events that are added in the direct translation Event-B, i.e. lift changes directions of movement, are also added by hand here.



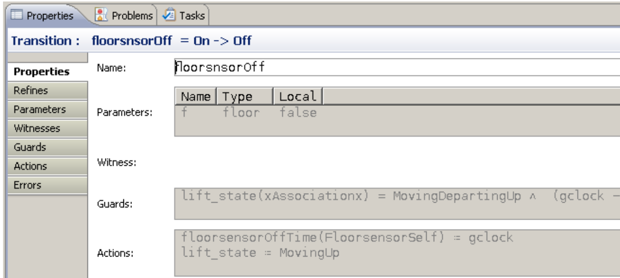


Fig. 11. Guards and actions for the floorsensorOff transition

4. Evaluation and Related Work

The output of our translation can be automatically validated by the RODIN verification tools platform [2]. This includes an attempted automatic proof of consistency of implementation relative to high level specifications. The ProB model checker and animator [14] performs consistency checking (finding deadlocks and invariant violations) and animation for validation purposes. The generated Event-B models for the case study Lift system were proven with the following results:

- For an Event-B model from the direct translation: Total POs: 135, Automatically discharged: 122, Manually discharged: 11, Reviewed: 2 and Undischarged: 0.
- For an Event-B model generated via a UML-B model: Total POs: 142, Automatically discharged: 54, Manually discharged: 84, Reviewed: 4 and Undischarged: 0.

The number of POs automatically discharged in the UML-B model is fewer than in the Event-B model because the UML-B model contains more indirection in the modelling of relationships between objects. This may be a side effect of the current style of modelling used in the example rather than an inherent feature of the notation and requires further investigation.

Since TDs express only a part of the whole system specifications, some events and variables must be added manually to the generated model. For example, in the lift case study, we added events: ChangeDirUp and ChangeDirDown while variables currentFl and dir were added to represent the current position and direction of the lift respectively.

There are related works which are trying to bridge the gap among causal dependencies relationships, timing constraints and B-method. For example, Abrial [15] introduces patterns for state-based specifications in Event-B and uses informal graphical notations similar to TD to illustrate the patterns. This demonstrates the need for such a visualisation. The patterns are useful for our research but only support cause/effect relationships, not timing constraints. Thus, we need to define more notations for this. Cansell et al. [16] introduce a time constraints pattern based on an Event-B model for distributed applications. This work uses global time which interacts with a number of active times as do our patterns. The difference is that they introduce time in refinement steps while we focus on time at the abstract stages. The work is restricted to message passing between two devices in the system while our work can handle many objects in the same time. Moreover, this work uses informal graphical notations which are not similar to TD for expressing timing constraints. Bicarregui [17] extends Event-B notations to three linear temporal logic (LTL) operators: *Next* (○), *Eventually* (◊) and *Bounded eventually* (≤ n) where n denotes time units. The work proposes using three new constructs that are to replace the standard Event-B structure, WHEN…THEN…END, to represent the three LTL operators. However, this work presents models in textual form. Our work is unique in providing techniques to create timing constraints from a TD to Event-B and UML-B models by using the standard Event-B notations provided.

KAOS [18] is a goal-oriented modelling technique for requirements specification, in which a goal defines an objective of the composite system. KAOS uses a *Goal model* to declare the system requirements. The *Goal model* is composed of a goal name, definition, and formal definition, where the latter is written as a temporal logic statement using LTL. Since the LTL can explain the specification of some properties - for example, next (○) and eventually (◊) - those properties are similar to what can be expressed by TD. Apart from our earlier work [19] that investigates how to generate KAOS goals from TD, there are a number of investigations that explore possible techniques for translating KAOS framework to other models. An attempt to combine KAOS with B is introduced by Ponsard and Dieul [20]. However, this work only focuses on traceability links. Other work has been done by Hassan [21] to transform KAOS *Operation model* to B specification language in security requirements. There is no work try to generate KAOS from TD.

5. Conclusion and Future directions

We provide a formal visual notation, based on the Robust TD notations [7], for specifying causal dependencies with timing constraints and link this to Event-B (via its UML-B graphical front-end) for verification and validation using the Rodin toolset. We propose systematic translation rules to transform TD into Event-B and UML-B models and demonstrate them using a real time case study: a lift system. A subset of TD notations was selected and some notations were adapted to make it easier to generate Event-B and UML-B. Thus, instead of manually generating these parts of the model in a textual

form, users can use the TD as a graphical front-end, and these details are created automatically in UML-B and Event-B. We provide multiple views of one system's requirements by expressing them in TD, Event-B and UML-B models.

Some future directions are suggested as follows.

1. Currently no graphical editor exists to support the drawing of our TD. They are entered using the EMF model editors. We intend to develop a graphical TD editor tool that integrates with the UML-B tools
2. For large systems scalability may be an issue. Future work will investigate techniques to compose TD subsystems.
3. Identify refinement steps in the Event-B model. For example, in the lift case study, the abstract model has basic lift behavior while the timing constraints are introduced in the refinement steps
4. Integrate KAOS framework with TDs.

The goal to generate translation techniques to transform a TD to Event-B and UML-B was accomplished. The work provides a more intuitive approach to specifying timing constraints and causal dependencies of a system than Event-B and/or UML-B by using the graphical visualisation of TD.

Acknowledgements

Colin Snook's contribution to this work was funded by the Deploy Project which is an EU, FP7 Integrated Project (IP 214158).

References

[1] J.-R. Abrial & S. Hallerstede, Refinement Decomposition and Instantiation of Discrete Models: Application to Event-B, *Fundamenta Informaticae*, ***77***(1-2), 2006, 1-28.

[2] RODIN. *Development Environment for Complex Systems (Rodin)*. 2009. http://rodin.cs.ncl.ac.uk/.

[3] N. Bashar & S. Easterbrook. Requirement Engineering: A Roadmap. *Proc.the Future of Software Engineering*, Limerick, Ireland, 2000, 35-46.

[4] M.A.Yoder & B.A. Black. A Study of Graphical vs. Textual Programming for Teaching DSP. *Proc. 36th Annual Frontiers in Education Conf.,* 2006, San Diego, CA, 17-18.

[5] C. Snook & M. Butler, UML-B and Event-B: an integration of languages and tools. *Proc. IASTED International Conf. on Software Engineering (SE2008)*, Innsbruck, Austria, 2008.

[6] M. Jackson, *Problem frames analysis and structuring software development problems* (Addison-Wesley, 2001).

[7] OMG. *UML 2.0*. 2008. http://www.uml.org/#UML2.0.

[8] Intel. TimeGen. 2009. http://www.xfusionsoftware.com/.

[9] MOHC. *TimingTool*. 2009. http://www.timingtool.com/.

[10] SynaptiCAD. 2009. http://www.syncad.com/.

[11] ATL. *ATLAS Transformation Language*. 2008. http://www.eclipse.org/m2m/atl/.

[12] Eclipse. *Eclipse Modeling Framework Project (EMF)*. 2008. http://www.eclipse.org/modeling/emf/.

[13] T. Joochim, Bringing Requirements Engineering to Formal Methods: Timing diagrams for Event-B and KAOS, *PhD. thesis*, School of Electronics and Computer Science, Southampton University, Southampton, United Kingdom, 2009.

[14] ProB. *ProB 1.2*. 2009. http://www.stups.uniduesseldorf.de/ProB/overview.php.

[15] J.-R. Abrial, Tutorial - Case study of a complete reactive system in Event-B: A mechanical press controller. *Proc. 5th International Symposium on Formal Methods (FM'2008)*, Turku, Finland, 2008.

[16] D. Cansell, D. Méry & J. Rehm, Time Constraint Patterns for Event B Development. *Proc. Formal Specification and Development in B, 7th International Conf. of B (B 2007)*, Besancon, France, 2007. 140-154.

[17] J. Bicarregui, et al, Towards Modelling Obligations in Event-B. *Proc. International Conf. of ASM, B and Z Users*, London, UK, 2008, 181-194.

[18] E. Letier & A.V. Lamsweerde, Agent-Based Tactics for Goal-Oriented Requirements Elaboration. *Proc. 24th International Conf. on Software Engineering (ICSE'02)*, Orlando, Florida, USA, 2002, 83-93.

[19] T. Joochim & M.R. Poppleton, Transforming Timing Diagrams into Knowledge Acquisition in Automated Specification. *Proc. 2nd International Conf. on Advance in Information Technology (IAIT2007)*, King Mongkut's University of Technology, Bangkok, Thailand 2007, 103-110.

[20] C. Ponsard & E. Dieul, From Requirements Models to Formal Specifications in B. *Proc. International Workshop on Regulations Modelling and their Validation and Verification (REMO2V'06)*, Universitaires de Namur, Luxemburg , 2006, 249-260.

[21] R. Hassan, et al, Integrating formal analysis and design to preserve security properties. *Proc. 42nd Hawaii International Conf. on System Sciences (HICSS-42)*, 2009. Waikoloa, Hawaii, USA, 1-10.