UNIVERSITY OF SOUTHAMPTON
FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
School of Electronics and Computer Science

Bringing Requirements Engineering to Formal Methods: Timing diagrams for Event-B and KAOS

by
Tossaporn Joochim

Thesis for the degree of Doctor of Philosophy

February, 2010

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING

SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

BRINGING REQUIREMENTS ENGINEERING TO FORMAL METHODS: TIMING DIAGRAMS FOR EVENT-B AND KAOS

by Tossaporn Joochim

Event-B is a language for the formal development of reactive systems. At present the RODIN toolkit (RODIN, 2009) for Event-B is used for modelling requirements, specifying refinements and verification. In order to extend the ability to model graphically requirements for the real-time domain, where timing constraints are essential, we use Timing diagrams for Event-B, UML-B and Knowledge Acquisition in autOmated Specification (KAOS). The Timing diagrams, based on UML 2.0 Timing diagram notation (OMG, 2007), provide an intuitive graphical specification capability for timing constraints and causal dependencies between system events. Translation schemes to Event-B, UML-B and KAOS are proposed and presented.

The benefit of our contribution is providing a graphical option to generate timing constraints and causal dependencies of a reactive system to Event-B, UML-B and KAOS Goals. Thus, instead of manually generating these Event-B, UML-B and KAOS Goal models in a textual form, users can use the TD as a graphical front-end, and these target models are created automatically.

We compare the three applications of the Timing diagrams in terms of their contribution to formal requirements engineering. A partial case study of a Lift System is used to demonstrate the translation in practice.

Contents

List of Figures

List of Tables

Declaration of Authorships

I, Tossaporn Joochim, declare that the thesis entitled "Bringing requirements engineering to formal methods: timing diagrams for Event-B and KAOS" and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this university:
- where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted for the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.
- parts of this work have been published as :

Joochim, T. and Poppleton, M. (2007) Transforming Timing Diagrams into Knowledge Acquisition in Automated Specification. In: *IAIT2007: The 2nd International Conference on Advances in Information Technology 2007*, Bangkok, Thailand.

Joochim, T. et. al. (2010) Timing Diagrams Requirements Modeling using Event-B Formal Methods. In: *SE 2010: Software Engineering 2010*, Innsbruck, Austria: Actapress.

Signed:

Date:

Acknowledgements

I would like to thank and extend my heartfelt gratitude to the following persons who have made the completion of this thesis possible:

My supervisors, Dr. Mike R. Poppleton and Dr. Andrew M. Gravell, for providing assistance in numerous ways. For supervision and continuous guidance enabled me to complete my work. Thank you for their expertise, patience, and kindness.

My parents, grandparents, my brothers and their family, for their vital encouragement; for their ultimate supporting and loving, for they have never hesitate to help if I needed and for all their sacrificing;

Antonin Hrdlicka for always being here to provide physical and mental support, very concerning of my safety and giving a great help in all matters;

Dr. Colin Snook who had done a wonderful job and I really appreciate his incredible advices on UML-B, and the willingness to help;

Dr. Emmanuel Letier for his expertise and great advice on KAOS;

Prof. Peter Henderson who gave me an opportunity to join the group; for his kindliness and his grateful advices during my exams;

Dr. Quintin Gee, Stuart McIntosh, Andy Edmund, and Alain J. Alherbe for proofreading my thesis;

My Thai friends, Mar Y. Said, Nurlida Basir, and DSSE secretaries;

The Royal Thai government, for the full funding during my PhD;

I would like to give very special thanks to Dr. Andrew M. Gravell. This thesis would not have been possible without him.

To my beloved grandfather, Mr. Boonchom Prasertsri

Definitions and Abbreviations Used

API	Application programming interface
ATL	Atlas Transformation Language
BNF	Backus-Naur Form
EMF	Eclipse Modelling Framework
FM	Formal Method
GMF	Graphical Modelling Framework
GORE	Goal-Oriented requirements engineering
IDE	Integrated Development Environment
KAOS	Knowledge Acquisition in automated Specification
MOF	Meta-Object Facility
OMG	Object Management Group
POs	Proof Obligations
RE	Requirements engineering
TD	Timing Diagram
UML	Unified Modelling Language

Chapter 1 Introduction

1.1 Overview

A requirement is “a feature of the system or a description of something the system is capable of doing in order to fulfil the system’s purpose” (Pfleeger, 1998). Requirements engineering (RE) is a part of the software development life cycle that is important for acquiring explicit system requirements. The RE is used to explore problems and potential solutions. It is also used for comparing alternative solutions and deciding which solution should be adopted for that system (Jureta, 2006). To specify requirements, one can use many different techniques, such as rich text, dataflow diagram, prototyping, Unified Modelling Language (UML) (OMG, 2008), Goal-Oriented Requirements Engineering (GORE), Knowledge Acquisition in autOmated Specification (KAOS) (Lamsweerde, Dardenne et al., 1991), and Formal Methods (FMs).

Critical systems are systems whose failure may have serious consequences to human beings, systems or businesses. Examples are: fire alarms, medical systems, traffic control, chemical plant control, and automotive control systems. Thus, to develop critical systems, one has to ensure that, as far as possible, the processes used are rigorous. Using mathematical notations – which describe the system in terms of predicates, booleans, sets, relations, and functions, as in Formal Methods (FMs) – is a way to improve the conformance of design to specifications, and to help eliminate errors early in the design process (Abrial, 1996; Bowen and Hinchey, 2006).

Since FMs has the concept of proving correctness, which supports the accuracy of software development, FMs have a major benefit in defining the precise specification and processing its verification (Abrial, 2005; Hall, 2007). The benefits of FMs can be summarized as follows:

- Developers are forced to consider more error behaviours arising from requirements, which can be eliminated by well-defined mathematical notations (Abrial, 2007; Langari and Pidduck, 2005). Developers are guided towards creating reliable and secure software systems. This aspect is always omitted from informal descriptions (Hall, 2007).
- Formal modelling is a way of improving the system analysis phase (Agerholm and Larsen, 1998). It can help developers achieve a better understanding of requirements and discover errors early in the lifecycle (Langari and Pidduck, 2005). This reduces the overall cost of the project (Agerholm and Larsen, 1998; Hall, 2007). King (King, Hammond et al., 2000) has shown that performing proof of correctness in FMs can detect more errors early in the development lifecycle; which is expedient from the economic point of view.
- Formal specifications of design and refinements can be proved consistent by model checking and by proof (Abrial, 2008a). It is also possible to use animation to help validate. Examples of tool support are in RODIN (RODIN, 2009), Atelier B (Requet, 2008; ClearSy, 2009) and ProB (ProB, 2009).
- Reasoning about derived system properties by stating theorems and other properties about the system makes the models more precise (George and Vaughn, 2003; Lamsweerde, 2009).
- In the formal development, the first model is called the abstract model. The abstract model is transformed through a formal sequence to obtain the refinement/concrete model. The concept of refinement in formal methods allows more detail, and the expression of some design decisions, to be added, in a stepwise manner, into the model. The advantage of refinement is allowing the model to be analysed at an abstract level, resulting in

reduced complexity/ambiguities (Abrial and Hallerstede, 2006). Absence of ambiguities is a benefit brought about by using formal specifications.

- FMs have been shown to provide more valuable documentation (Bowen and Hinchey, 2006).

The Event-B (Abrial and Hallerstede, 2006) method is an FM developed by Jean-Raymond Abrial. It is a formal language for state-based modelling and verification for reactive systems, developed in the context of RODIN (RODIN, 2009), a European IST project. Event-B itself is composed of static and dynamic parts. The static part is called a CONTEXT and is used to declare constants, carrier sets and axioms. The dynamic part is called a MACHINE, which contains state variables, variable properties described by invariants and units of behaviour, which are called EVENTS. Event-B is good for identifying precise system requirements (due to its use of mathematical notation, and well-defined semantics), but it is not yet clear how best to model various complex requirements patterns in Event-B, such as timing constraints and causal dependencies on system events. Moreover, Event-B can be difficult to uses and it requires trained professionals (Bashar and Easterbrook, 2000; Lamsweerde, 2000; Bowen and Hinchey, 2006).

UML (OMG, 2008) is a language for specifying, visualizing, and documenting the artifacts of software systems using graphical diagrams. UML is suitable for using in object-oriented analysis and design (Popandreeva, 2007) and is best used to describe functional requirements (defining what the system has to do in its environment). For example, the lift must stop at the requested floors, and the lift's door must be opened only when the lift is stopped are functional requirements for the lift system. Other examples of systems that can use UML to identify their specfications are handling control of technical equipment (e.g. uses *Sequence diagram* and *Statechart*), embedded systems such as mobile phones (e.g. uses *Component diagram*), and giving a clear description of what the system should do (e.g. uses *Use-case diagram*). Currently, the official version is UML 2.0 (OMG, 2008).

Even though UML is a popular object-oriented modelling approach and has been using widely, it lacks mappings to formal models. Presently, many groups of people are trying to bridge the gap between B-Method and UML diagrams (Ledang

and Souquierès, 2002a; Ledang and Souquierès, 2002b; Jiufu, 2007; Younes and Ayed, 2007); the U2B and UML-B (Snook and Butler, 2008a) projects.

UML-B tool is a graphical front end for Event-B; UML-B language defines abstract syntax with the Eclipse Meta-Object Facility (MOF) (OMG-MOF, 2007). MOF is one of the OMG standards and a meta-metamodel. It is a mechanism for building metamodels, which is used to define types of model structures and architecture. MOF is designed as a four-layered structure:

M0: this level is used to describe real-world objects.

M1: this level is used to define models such as UML and UML-B diagrams.

M2: this level is used to define metamodel description –syntax and semantic- of elements in the M1 layer. For instance, the UML-B metamodel and our Timing Diagram (TDs) metamodel are defined at this level.

M3: this level is a meta-metamodel; it is used to define MOF itself.

UML-B uses UML-like diagrams, i.e. Class diagrams and Statecharts, to generate system specifications models. UML-B models can then be translated into Event-B by using a U2B translator. Users can update/add/modify information directly using the tool.

Timing constraints and causal dependencies among objects play an essential role in the different varieties of systems. Timing constraints are one of the control issues in reactive and critical systems that are particularly critical to systems and must be controlled (Liu, Chou et al., 2001; Ng and Patel, 1994). A system which fails to meet the timing constraints deadline may not only be able to make an emergency control but can have also other inconvenient consequences (Groom, Maciejewski et al., 1999). Some failures may cost a great deal of money and even human lives (LeMieux, 2003). Thus, it is important to correctly model timing and causal constraints system.

Timing diagrams (TDs) (OMG, 2008) are one of the new artefacts introduced to UML 2.0 and are used to explain the behaviours of objects throughout a given period of time (Ambler, 2004; Khan, Geihs et al., 2006). TDs are best used to depict functional requirements with causal dependencies between objects and timing constraints (Gavras, 2003; Brisolara, Kreutz et al., 2009). For example, parts of a lift system: "…The lift will be stopped at the current floor between 1-5 seconds

after the current floor sensor is set on. A lift door does not open until the lift stops…"

Even though the information on the TD, such as the lift system, can be expressed in other diagrams, for example using Statecharts in UML-B, it is not a helpful way for the users to operate. For instance, one can put timing or state constraints into Statecharts, but, in general, one Statemachine refers to other Statecharts for the dependency.

If we have three different classes, and each object of these classes has state changes, then we need three Statecharts. Each such Statechart may have guards that refer to other Statecharts which means, in using UML-B, we have guards on the state transitions here which refer to some activities going on somewhere else. For (Sommerville, 2004) example, Figure 1-1, there are three different Statecharts: *Door*, *Lift* and *Floorsensor*. There are guards from the *Lift* to the *Floorsensor*, from the *Door* to the *Lift*, and from the *Floorsensor* to the *Lift*.

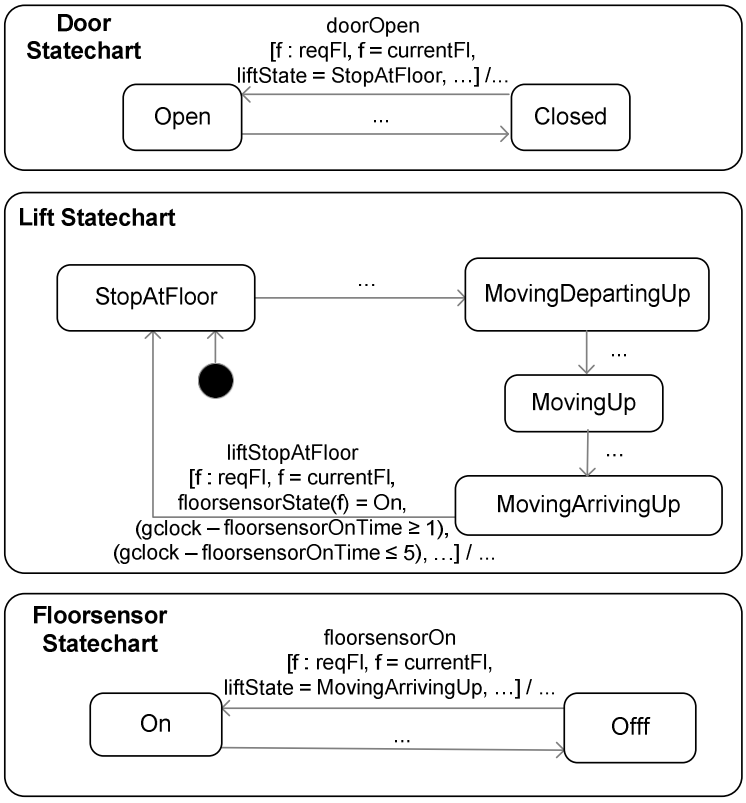


Figure 1-1 Example of Statecharts for Door, Lift and Floorsensor

If a guard is concerned with timing constraints, it must be declared with a long condition on the state transition (as shown in Figure 1-1 by the guard between *MovingArringUp* and *StopAtFloor* states of the *Lift Statechart*). In UML-B, the causal interaction between these objects cannot be contained in a single diagram. Thus, we have many charts to display at the same time which makes it difficult to read on a computer screen, and is not helpful for the users in terms of modelling.

In TDs, as shown in Figure 1-2, we can describe the causality explicitly with arrows between the *Door*, the *Lift* and the *Floor sensor*, and have them all on the same screen.

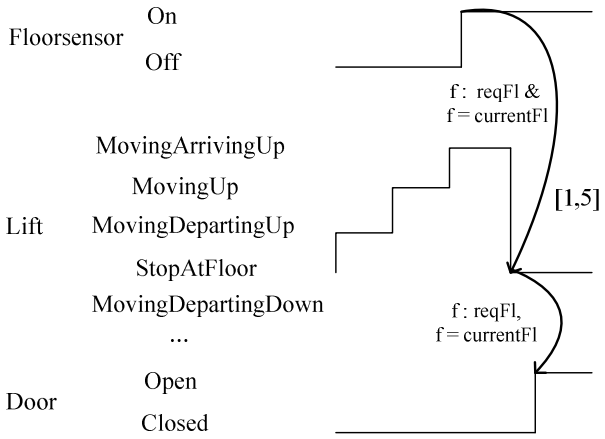


Figure 1-2 Example of Timing diagram for Door, Lift and Floorsensor

The TD notations include graphically described extra conditions (as shown by *f : reqFl & f = currentFl*) and timing constraints (as shown by [1,5]). It is very natural to form expressions in timing constraints using a TD timing constraints notation. Therefore, combining TD and UML-B would be beneficial for the user.

There are other two mathematical modelling languages concerned with time: Timed Petri Nets (Berthomieu and Diaz, 1991; Ramchandani, 1974) and Time Petri Nets (TPNs) (Cerone and Maggiolo-Schettini, 1999). Both are graphic representation for concurrent formalisms approaches for specifiying real-time formal systems and extend Petri Net (Reisig, 1985). Timed Petri Nets and TPNs consist of *places*, *transitons*, *time*, and *directed arcs* which represent conditions, events, timing constraints of the *transtitions*, and relationships between *places* and *transtions* in the system respectively. For Timed Petri Net, a *transition* can fire as

soon as possible whilst for TPNs it fires within a time interval (Cassez and Roux, 2005).

In this research we selected TDs over Timed Petri Nets/TPNS since adding new notations (with the purpose of generating expressions to interface with Event-B and KAOS, as described in Chapter 4) is more flexible with TDs than Timed Petri Nets/TPNs. Moreover, TDs use simple graphical notations and are not difficult to understand.

Requirements are often unclear when first elicited from stakeholders. Goal-Oriented requirements engineering (GORE) allows the requirements to be clarified throughout an incremental process. It concerns the use of goals for eliciting, elaborating and refining, specifying and modelling of requirements (Lamsweerde 2004; Anwer and Ikram 2006). Examples of the goal-oriented approach are Non-Functional Requirements (NFRs) (Chung, 1993), i* diagrams (Yu, 1993), Goal-Oriented Idea Generation Method (GOIG) (Oshiro, Watahiki et al., 2003) and Knowledge Acquisition in autOmated Specification (KAOS) (Dardenne, Lamsweerde et al., 1993) frameworks. NFRs are used to represent and analyze non-functional requirements and guides the design processes. i* diagrams show how actors in a system depend on each others for a specific goal in a system. GOIG is focused on idea-generation, that is, stakeholders' ideas are elicited as sub-goals. The ideas are grouped, and associations between those ideas are used to generate a goal graph.

KAOS is a goal-oriented modelling requirements specification technique, in which a goal defines an objective of the composite system. KAOS has concepts of refining goals, identifying agents, and exploring alternative responsibilities (Letier and Lamsweerde 2002a); it uses the *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 linear temporal logic (LTL). Since the LTL can explain the specification of some properties - for example, next (○) and eventually (◊) - those properties are similar with what can be expressed by TD. This is the reason KAOS is selected over the other GOREs. KAOS is a semi-FM and does not have the capability of generating and discharging proof obligations as in full FMs. Thus, an attempt to generate a FM model from a KAOS model is founded in (Nakagawa, Taguchi et al., 2007) to transform KAOS

into VDM++ (Fitzgerald, Larsen et al. 2004), a formal object-oriented specification language. There are a number of tools supports that generation of KAOS models such as Objectiver (Delor, farimont et al., 2003; Ponsard, Balych et al., 2006) and FAUST tool (Rifaut, Massonet et al., 2003). The Objectiver is a tool for generating KAOS models and documents while the FAUST tool is used to verify KAOS models. Heaven and Finkelstein attempted to combine UML and KAOS; the researchers created a tool to allow KAOS to be represented in UML by using a profile (Heaven and Finkelstein, 2004).

Problem Frames (Jackson, 1995) is a technique to demonstrate problem requirements in a diagrammatic form, which the diagram is called Problem diagrams. As shown in Figure 1-3, a Problem diagram comprises a software *Machine*, real world which is called *Problem World*, and the system requirements are represented by a dotted oval (Jackson, 2001). The Problem diagrams identify how these system components relevant with each others. The machine interacts with the *Problem World* by shared control phenomena (e.g. shared events and/or shared states), called *specification phenomena*. The links between the *Problem World* and the requirements are called *requirement phenomena* which are "the phenomena that the customer for the system would observe to determine whether the requirement is satisfied" (Jackson, 2005).

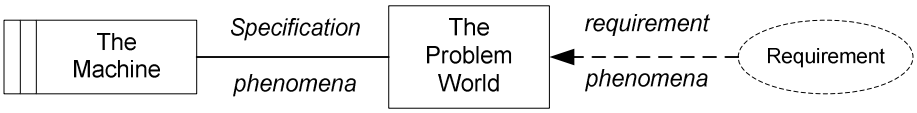


Figure 1-3 Problem diagram

Problem Frames has a concept of decomposition in which a large problem can be separated into subproblems. Each subproblem is a complete system which has its own Problem diagram, a *Machine*, a requirement and *Problem World* (Jackson, 2005; Cox, Hall et al., 2005).

Even though the concept of Problem Frames to refine a large problem into subproblem is similar with KAOS, the Problem Frames is not aiming at using formal descriptions such as temporal logics nor mathematic notations. Thus, the

Problem Frames is not selected in this research as we aim at generating tool supported formal method models.

1.2 Motivation

The key contributions of this work are indicated by three motivating assumptions we make:

1. When FMs are used early in the system development process, they help to remove ambiguity, incompleteness, and inconsistencies in system specifications (Sommerville, 2004; Wing, 1990). This decreases requirements' errors because it forces the developers to do a detailed analysis of the requirements (Abrial, 2005; Hall, 2007). Thus, implementation and validation costs should be reduced, as there are fewer errors in the specifications; that is useful in term of requirement engineering. However, FMs demand costly trainings of engineers because of their mathematical and logical basis (Bashar and Easterbrook, 2000; Lamsweerde, 2000; Bowen and Hinchey, 2006). This leads to the second assumption.
2. It is useful to enable more requirements to be expressed graphically when working with FMs. That is, we wish to enhance the graphical aspects of FMs with graphical elements (such as is done in UML-B and KAOS). Using graphical methods has some benefits over FMs as in the following:
    - Presenting requirements in graphical form is an easier way and more readable for software developers/students to define their requirement specifications than by difficult using of formal notations (Yoder and Black, 2006). As (Razili, Snook et al., 2007) has suggested that model comprehensibility can be improved by using UML-based graphical specifications rather than the formal notation alone.
    - It reduces the training for the formalism if developers are able to model graphically rather than using FMs (Becker-Kornstaedt, Neu et al., 2001). Modelling may become

accessible to more staff and it does not require a high level of professional training.

- Using simple symbols helps teaching FM courses (Snook and Butler., 2001; Razili, Snook et al., 2007).

There are other papers to support those ideas, such as Zimmerman who states that tabular and diagrammatic notations are more readable than textual ones in a complex system (Zimmerman, Lundqvist et al., 2002). This is confirmed by a number of related studies in (Petre, 1995).

3. The integration of different specification modelling frameworks for specifying and reasoning about requirements is beneficial (Allemand, Attiogbé et al., 2002; Attiogbé, Poizat et al., 2003). Moreover, it is also useful to describe one system in multiple views.

1.3 Goal

The goals for the research are identified as in the following

1. To provide an option to help users/developers generate timing constraints and casual dependecies requirements in a reactive system in forms of Event-B and UML-B formal models.
2. To generate a translation technique to transform a TD into KAOS Goal models. The TD graphical front-end is beneficial in an engineering context since the original KAOS Goals' formal definitions is defined by linear temporal logics (LTLs) textual declarations. It is inconvenient for a user who is unfamiliar with using temporal logics. Thus, a TD is used as graphical front-end to represent a KAOS Goal model. With the translation rules, KAOS goals are automatically generated from TD.
3. To confirm that using graphical TD to specify timing constraints and casual dependencies requirements in Event-B is easier than using textual methods.

According to the goals above, we select some modelling frameworks as the basis for our contribution:

1. Extended Timing diagrams (TDs): OMG UML2.0 TD notations are clearly defined and widely used to describe behaviours of objects in many critical systems and even within electronics engineering for a long time (Fowler and Scott 2004). Thus, we select to extend UML TD notations for timing constraints graphical modelling and causal event dependencies. The reason to extend UML TDs is they do not support adequate notations to explain certain kinds of specification. For example, identifying combination of causes that make something to happen, and showing synchronisation of objects that change their states simultaneously. Thus, AND and OR node notations are created as well as simultaneity arrows (more detail in Chapter 4). Here, a TD is used as a source model in generating target models: Event-B, UML-B and KAOS models.
2. Event-B is selected as it is well used in Electronics and Computer Science school, University of Southampton. There are many partners through the RODIN project and is used in industries (Europe). Moreover, it is integrated well with Eclipse and has good tools support such as Event-B RODIN toolkits, B prover and animators. Although, Event-B is good for identifying precise system requirements by using set-theoretic notation, it is not yet clear how to model timing constraints and causal dependencies on system requirements in Event-B. Thus, we selected to add TDs as the front-end for Event-B. Event-B then can be described by visualisation graphics for the time.
3. UML-B is selected as it is plug-in for RODIN and is developed on Eclipse. UML-B is graphical Event-B modelling in which Class diagrams and Statecharts are used to express formal specifications. An Event-B model is generated automatically when the model is saved. Thus, it is suitable as an alternative way to generate an Event-B model.
4. KAOS has been widely applied in many critical systems, according to (Lamsweerde 2004), it is used in Air Traffic Control (conflict handling between ground and on board collision avoidance systems) and Aerospace (design of test suites for rocket launch). KAOS explains timing constraints by LTLs, and cause-effect relationships in pre- and post-conditons which are in textual form. In contrast, TDs timing constraints can be clearly

explained using notations that are time bounds and causal dependencies arrows. TDs use more natural visualisation graphical to declare the time than in the KAOS LTL operators. Thus, TD is selected as a front-end for KAOS. Moreover, both KAOS and Event-B use first-order predicate logics to describe system behaviour and have a concept of refinement to explain more system detail in the further steps. Thus, it is interesting to integrate TD to KAOS which aims to generate Event-B models later.

5. Atlas Transformation Language (ATL) (ATL 2008) is developed on Eclipse and a language to generate a target model from source models based on metamodel. We select ATL as a language for generation an UML-B model from a TD since UML-B is also developed on Eclipse. Moreover, there are many ATL examples on-lines.
6. Backus-Nuar Form (BNF): BNF is used to describe TD notations which are used to generate an Event-B model and KAOS *Goal model*. BNF is widely used for explaining syntax of a language and provides standard symbols to do that. Thus, it is suitable to use BNF for creating formally systematic translation rules in our work.
7. A lift case study is used in the transformations. Even though, the lift is a single example, it is appropriate to validate my work as follows. It has real time properties; represents causal dependencies among objects in the system; and some parts of the specification cannot be modelled by timing diagram (see section 5.2, 6.5.1 and 8.1.6 for detail) which is useful as an example of fulfilling models by hand. Moreover, a specification of lift is well-known, not hard to understand and is widely used in many works, as details describe in section 8.2.

1.4 Contribution Overview

Our contribution focuses on how parts of system's requirements, concerned with timing constraints and causal dependencies between a system's objects, are transformed into FM models. The aim of this contribution is to enable users to easily model critical system requirements using graphical notations e.g. TD; by

adding UML-B and KAOS graphical capability to express timing constraints and event dependencies requirements.

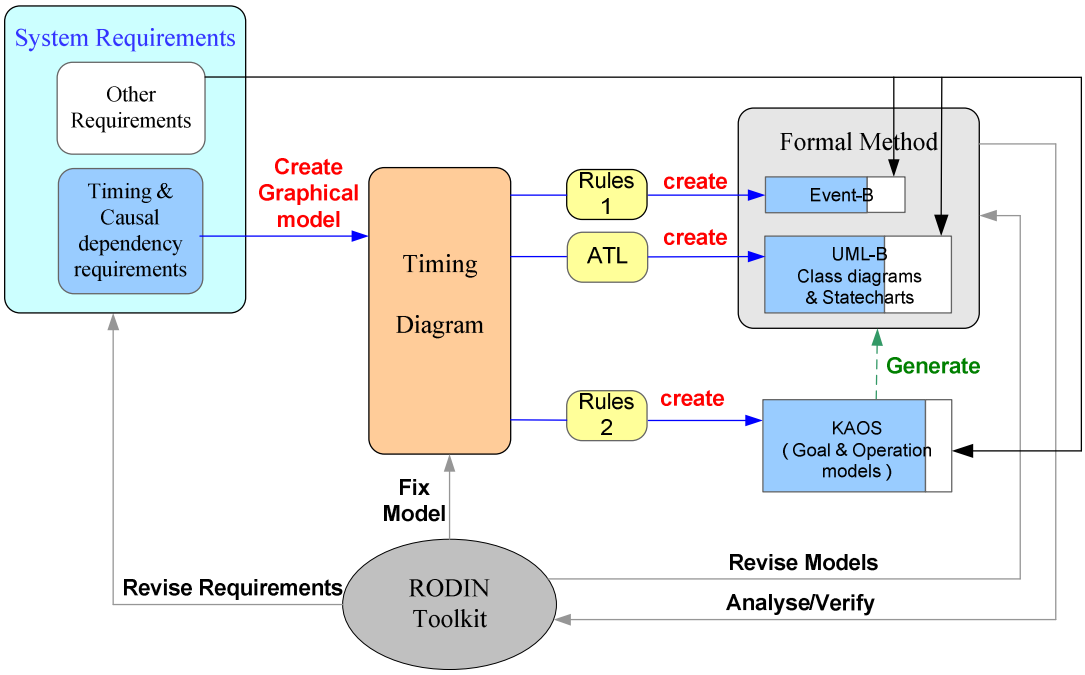


Figure 1-4 Research aim

Figure 1-4 presents the whole thesis scenario. Requirements are partitioned into other requirements (non-timing), and timing and causal dependency requirements. The requirements which can be described by causal dependency and timing constraints are modelled by TD. Formal translation *rules 1* and *rule 2* are built based on TD BNF definitions to create Event-B and KAOS models from TD respectively. Other requirements are used to generate the remainder of the Event-B and KAOS models for completion. Atlas Transformation Language (ATL) (ATL 2008) transformation rules are generated to create UML-B Class diagrams and Statecharts from TD. The remainder of the UML-B models are also generated from the rest of the requirements. Next, Event-B and UML-B models are analysed/verified by the RODIN Toolkit. If there are any errors, ambiguities or incompleteness, which are indicated by the RODIN (model checking and proof obligations), the Event-B and UML-B models are revised; the TD can be fixed as well as system requirements may be revised. 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. It enables the gaining of a clear understanding of the task at an early stage.

1.5 Document Structure

The remainder of this thesis is structured as follows:

Chapter 2 reviews the literature on the technical approaches that are directly involved in the research. The chapter starts with describing the general idea of RE techniques and FMs. The Event-B notations and methods used to develop Event-B are described. The use of refinement in Event-B, that takes a model of abstract level to one with more concrete detail, is explained. The RODIN tool set that can be used in Event-B development is also explained. The UML-B toolkit, a graphical front-end for Event-B, and its implementation, are demonstrated. The KAOS framework, that is a technique for goal-oriented modelling of requirements specification, and its notation, are described. There is an explanation of MOF and Eclipse metamodelling, which are used to generate UML-B and TD metamodel. The chapter finishes with an explanation of the ATL language, which is used to generate formal rules to transform TD to UML-B model, illustrated by examples of ATL rules.

Chapter 3 describes other relevant techniques which are elaborated in this thesis. The chapter starts with giving explanation of OMG System Modelling Language (SysML), which is a graphical modelling language for specifying, analyzing, and designing systems. Requirements diagram, which is a new diagram for SysML is discussed, illustrated with an example of modelling a lift system. An Action/Reaction pattern, which is used as a guideline for translating TD to Event-B, is described. Relevant researches on combining KAOS, B, UML and CSP, is discussed; likewise works on transforming TD to LTL formulas. The chapter finished with an explanation of properties that are significant for maintaining the correctness of doing RE, i.e. traceability, safety, liveness and fairness.

Chapter 4 describes a case study, lift System, which provides examples of requirements focusing on timing constraints and causal dependencies among objects. It is used for exploring translations from TD to Event-B, UML-B and KAOS. UML TD 2.0 is described; this is the standard notation used for defining the behaviour of different objects within a time-scale. Selected and amended TD notations are explained. A preliminary TD editor is introduced at the end of the chapter, but it was based on outdated TD notations, and so was not used for creating TD here. Instead, we created TD from Microsoft Visio for the representation/visualisation. For translating TD into UML-B, the TD description is generated by EMF.

Chapter 5 describes how to generate direct translation rules that are used to transform TD into Event-B model. TD BNF definitions are provided and used as input parameters for formal translation rules. The rule definitions are explained, followed by illustrations of generating Event-B models from the rules. The chapter finishes with a description of how non-timing requirements are added to complete the Event-B model.

Chapter 6 describes the translation rules for generating UML-B models from TD. The chapter starts with explanation of TD Metamodel created by EMF. TD used for the translation is introduced. ATL translation rules which, are used to create UML-B components, are described through examples. The chapter finishes with an explanation of how additional information are added to the model.

Chapter 7 describes the translation techniques that are used for generating KAOS *Goal models* from TD. The chapter starts by explaining a scope of TD and LTL operators which can be used for the translation. Next, explanation of TD BNF, and formal translation rules are provided, together with examples. Steps of goal trees creation and manual information addition are illustrated. The chapter finishes with a description of how *Operation models* are created.

Chapter 8 gives a comparative evaluation of the three direct translation methodologies; that is from TD to Event-B, UML-B and KAOS models. The comparisons explain what the differences and similarities in techniques and notations used to generate those models, as well as what additional information and where it is needed for each. We address how straightforward or complicated it is to generate and alter the models. This chapter provides the comparison with other related works. The comparison of a number of proof obligations in Event-B and UML-B models is provided. Finally, an example of proof obligations is explained.

Chapter 9 explains the contributions of this research. Limitations of the work are examined. Possible directions for future work are described.

Chapter 2 Technical Background

This chapter aims at giving background to the knowledge used in the thesis. Many fields of knowledge are used vary from FMs: specifically Event-B and UML-B techniques, Goal-oriented requirement engineering, Eclipse modelling framework and metamodel. The knowledge explanations are provided along with examples.

The structure of this chapter is as follows. Section 2.1 introduces background knowledge of RE methodologies, along with FMs. Section 2.2 explains FM methodologies and their categories. Section 2.3 gives the detail of Event-B modelling by describing the philosophy, followed by an introduction to the constructs used for modelling systems in Event-B. More detail is given on the refinement method used to develop Event-B models and proof obligations. Section 2.4 explains the RODIN tools used for creating and verifying the models generated in the thesis. Section 2.5 explains an UML-B tool that is used to develop a UML-B model. Section 2.6 discusses Linear Temporal Logic (LTL) operators that are used to describe KAOS *Goal* and *Operation models*. Section 2.7 explains KAOS frameworks, with corresponding examples. Section 2.8 introduces metamodelling: creating types and model structures for the models. Section 2.9 describes ATL and its components.

2.1 Requirements Engineering

Requirements engineering (RE) is the first step of the system development process. It is concerned with activities for eliciting, evaluating, specifying, analysing, documenting, and revising, the objectives, functionalities, and constraints to be obtained for a proposed system within a particular environment. Requirements can be grouped into two categories: functional and non-functional. Functional requirements associate with specific functions, tasks or behaviours the proposed system must support. For example, "lift doors must be closed when the lift is moving", and "the lift must be eventually stop at requested floors" are functional requirements. Non-functional requirements provide constraints that are not explicitly functional but do satisfy functional requirements. They include availability, reliability, performance, convenience, installation, and maintainability requirements. For example, "the lift should move smoothly between floors", "the lift position must be clearly seen at any time by users", and "the lift has to be tested every year", are non-functional requirements. This thesis focuses on functional requirements.

Requirements are elicited (by using techniques such as data collection, questionnaires, prototyping, knowledge reuse) and evaluated (e.g. by inconsistency management and risk analysis). More detail of elicitation and evaluation can be found in (Lamsweerde, 2009). Later, the results of elicitation and evaluation need to be specified and documented. There are many techniques for identifying requirements specification. For example, describing in natural language, using decision tables, entity-relationship diagrams (ERD) (Chen, 1976), dataflow diagrams (DFD), UML diagrams (OMG, 2008), e.g. TD, UML-like diagrams, e.g. UML-B (Snook and Butler, 2008a), semi-formal specifications, e.g. KAOS (Lamsweerde*, et al.*, 1991), and formal specifications, e.g. Z (Spivey, 1992) and Event-B (RODIN, 2009).

This thesis examines a combination of requirements specification techniques: TDs, Event-B, UML-B, and KAOS (as described in Chapter 1). TD was selected because we emphasise modelling a system's timing requirements where there are causal dependencies between system objects. Moreover, OMG UML provides TD standard notations, some of which are appropriate for our translation. Event-B and

UML-B were chosen as they are techniques for FM modelling and have effective tools support. KAOS is a semi-formal method, which uses timing constraints by discrete time points. Thus, it was selected to combine with TD.

2.2 Formal Methods

FMs are a set of techniques used to create a formal specification, develop a new specification (for example: refinements), and verify a specification by using mathematical notations for software engineering. The benefit expected from formalization is a higher degree of precision in specification, as it forces one to write an unambiguous detailed description and consider all the cases that may cause erroneous behaviour. As a result, the specification gains a high-level of correctness of requirements and benefits the design process. Using a FM helps reduce defect rates in software development and saves money in fixing errors in requirements, as shown by (Praxis High Integrity Systems, 2008) and (Hall, 2005). FMs can be broadly classified into two categories.

- State-based notations: this kind of FM supports creating system specifications by construction of a set-theoretic model. The model is described by invariants, state variables, and operations over the states. Invariants define condition constraints that the system's states must be always hold. Variables are used to indicate system state information. An operation is defined by pre- and post-conditions over system variables. A pre-condition contains necessary input variables that are constraints for an operation to be applied. A post-condition contains output variables after an operation is applied; it updates the system states. Examples of this kind of FM are VDM (Jones, 1986), Z (Spivey, 1992) and B (Abrial, 1996).
- Process algebras notations: this kind of FM supports creating system specifications by using methods derived from algebraic operators. It specifies a system as collections of concurrent and communication processes. These processes can be executed by many abstract machines according to specific rules of interaction. In particular, this FM requires interactions between components of software architectures and protocols.

Examples of this kind of FM are LOTOS (Bolognesi and Brinksma, 1987) and Communicating Sequential Process (CSP) (Hoare, 1985).

There are some other FM methodologies, whose features are defined in between those categories above. Two examples are: Petri Nets (Peterson, 1981) which is state-based, defined as a graphical language, and suitable for modelling concurrent behaviour of distributed systems; and Larch (Guttag*, et al.*, 1993) which is a state-based and algebraic specification method, specialized in the specification of abstract data types and their properties.

2.3 Event-B Modelling

The classical B-Method (Abrial, 1996; Schneider, 2001) is a mathematical method for formal system specification, design and implementation of software based on refinement. The classical B-Method defines a machine with variables, invariants, and operations. It has a concept of refinement that allows one to gradually build a model more and more precise in detail. The benefit of refinement helps to reduce degree of model's complexity. Moreover, if the model is massive, it is impossible to represent everything. To verify the correctness of a B model, proof obligations and model checking are used. Examples of tools supporting verification in B are Atelier B (ClearSy, 2009), B-toolkit (Sørensen, 1994) and ProB (ProB, 2009).

Event-B is derived from classical B. It keeps the concepts of classical B-Method but adds the concept of event. Event-B has simplified language syntax, stronger refinement notion and more powerful tool support (RODIN, 2009). Since Event-B models have well-defined syntax and semantics, it is possible to test them by proving that transitions made during the software process are correct. The Event-B provides proof obligations (POs) to ensure the correctness of a model. The POs are generated according to the correctness criteria, which are required within the models. Those POs have to be discharged by users and can be supported by automated proof tools, the RODIN tool (Butler and Hallerstede, 2007). Other plug-ins for RODIN are UML-B (Snook and Butler, 2008b) for adding class-oriented and Statemachine Event-B modelling capabilities, ProB (Leuschel, 2007) for

animating, systematically checking and assisting proving a B model, and BRAMA (Requet, 2007) for animating B models.

The B-Method has been successfully employed in the development of safety-critical systems such as signalling on Line 14 of the Paris Metro (Dehbonei and Mejia, 1995), and the Roissy Airport Shuttle (Abrial, 2006; Abrial, 2007). Bicarregui reports using B in six case studies, such as a short-term conflict alert air traffic control application, and clinical biochemistry (Bicarregui*, et al.*, 1997). The B-Method also contributed to the development of IBM's CICS product (Hoare*, et al.*, 1996).

2.3.1 *Contexts and Machines*

Event-B's kernel mathematical language is defined and explained in (Métayer and Voisin, 2007). 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. Carrier **SETS** (*s*) define sets and are represented by their name. Different carrier sets are independent. **CONSTANTS** (*c*) are defined by a number of **AXIOMS** *A*(*s*,*c*). **AXIOMS** gives properties about constraints and are dependent on the carrier sets *s* (Abrial and Hallerstede, 2006). **THEOREMS** are required assertions for proving. They are derived properties that should be provable from axioms (Hallerstede, 2006). The structure of an Event-B context is illustrated in the following:

CONTEXT *context_name*
SETS *s*
CONSTANTS *c*
AXIOMS *A*(*s*,*c*)
THEOREMS

Figure 2-1 Event-B Static structure: Context

The **MACHINE** defines the behaviour of the Event-B model. It includes **VARIABLES** *v*, **INVARIANTS** *I*(*s*, *c*, *v*), **INITIALISATION** *T* and **EVENTS** *E*. **VARIABLES**

define machine variables, which are used to maintain state information while performing events. **INVARIANTS** are used to define a property over the states and context of the system that must be satisfied by all events. **INITIALISATION** is used to specify the initial values of variables, while **EVENTS** define the units of behaviour that include possible state changes. The structure of an Event-B machine is illustrated in the following:

MACHINE *machine_name*
SEES *context_name*
VARIABLES *v*
INVARIANTS *I*(*s*, *c*, *v*)
INITIALISATION *T*
EVENTS
E1 = **WHEN** $G_1(s, c, v)$ **THEN** $S_1(s, c, v)$ **END**
E2 = **ANY** *l* **WHERE** $G_1(l, s, c, v)$ **THEN** $S_2(l, s, c, v)$ **END**
…
END

Figure 2-2 Event-B Dynamic structure: Machine

An event has a name and is composed of guards *G*(*s*, *c*, *v*) and actions *S*(*s*, *c*, *v*). Guards identify lists of conditions for the event to occur, while actions identify how the state variables evolve when the event occurs. Alternatively, an event can be defined without a guard or possibly with a non-deterministic clause, as shown in Figure 2-3. From this figure, three possible structure types of an event are shown: *Simple*, *Guarded* and *Non deterministic*.

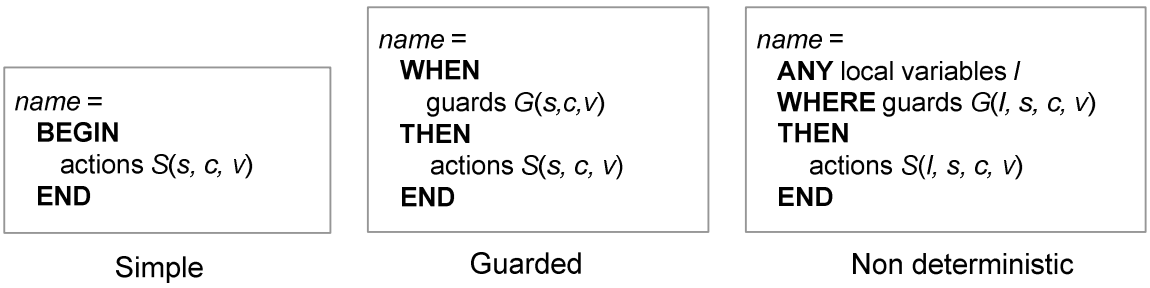


Figure 2-3 Event-B Structure

A *Simple* structure declares an event that does not have a guard but actions *S(s, c, v)*. A *Guarded* structure is used to identify an event with guards *G(s, c, v)* and actions *S(s, c, v)* but omitting local variables *l*. A *Non deterministic* structure is the general form of an event and used when the event has local variables *l* with guards *G(l, s, c, v)* and actions *G(l, s, c, v)*. Examples of each Event-B structure are given below:

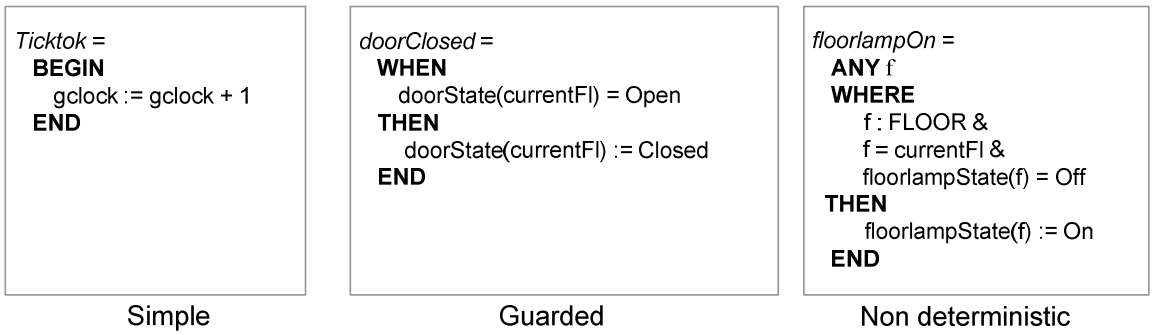


Figure 2-4 Examples of each Event-B Structure

From Figure 2-4, an event *Ticktok* is defined as a *Simple* structure and gclock as a machine variable. An event *doorClosed* is defined as a *Guard* structure, where doorState and currentFl are machine variables. Note that currentFl will be defined as an element of a class FLOOR, while doorState is defined as a surjective function from a class FLOOR to a set of door's states in **INVARIANT**. An event *floorlampOn* is defined as a *Non deterministic* structure with a non-deterministic local variable f under **ANY** clause. The guards, f : FLOOR & f = currentFl & floorlampState(f) = Off, are defined in a **WHERE** clause, where currentFl and floorlampState are machine variables. The action clause is defined by floorlampState(f) := On.

2.3.2 *Before-After predicates associated with an assignment*

A before-after predicate (BA) is used to express a relationship between the machine's state variable before an assignment takes place (denoted by *v*), and after an assignment takes place (denoted by *v'*). The before-after predicates are defined within three kinds of assignment: Deterministic, Non-deterministic and Empty.

Deterministic: a deterministic assignment is in a form <*variable identifier list*> := <*expression list*>. That is, if v is a list of variables and E a list of expressions, an action is declared by $v := E(v)$ in which its before-after predicate is defined by $v' = E(v)$. For example, an action $v := v + 1$ is written in the form of a before-after as $v' = v + 1$.

Non-deterministic: a non-deterministic assignment is in a form of
<*variable identifier list*> :| <*before-after predicate*>.
For example, $v,y :| v' = v + 1 \wedge y' = y + 1$ which is equivalent to $v,y := v + 1, y + 1$.

Empty: the substitution does nothing and is assigned to *skip*. The before-after state for this kind of substitution is $v' = v$.

Consistency Proofs

An Event-B model has to perform consistency proofs to ensure the correctness of the model. In the RODIN tool, the POs are automatically generated by the *Proof obligation generator* and the outcomes are transmitted to the *Prover* (Abrial, 2008b). The *Prover* performs automatic or interactive proofs and provides the outcomes. The detail of the tool is described in section 2.4 below. There are a number of POs that have to be generated, as described in (Métayer, *et al.*, 2005; Abrial and Hallerstede, 2006; Abrial, *et al.*, 2007). Here, we give examples of two proof obligations: *Invariance Preservation* and *Feasibility*.

The *invariant preservation statement* (**INV**) is the PO that each invariant is maintained whenever variables' values are changed by each event. The formal definition of **INV** is illustrated below (Abrial, 2008b).

Axioms	$A(s,c)$
Invariants	$I(s, c, v)$
Guards of the event	$G(s, c, v)$
Before-after predicate of the event	$BA(s, c, v, v')$
$\Rightarrow$	$\Rightarrow$
Modified Specific Invariant	$Im(s, c, v')$

(1)

A *feasibility statement* (**FIS**) is the PO that under the axiom $A(s,c)$, the invariants $I(s, c, v)$, and the guard $G(s, c, v)$, the action gives at least an after value v'. The formal definition of **FIS** is illustrated below (Abrial, 2008a).

Axioms	$A(s,c)$
Invariants	$I(s, c, v)$
Guards of the event	$G(s, c, v)$
$\Rightarrow$	$\Rightarrow$
$\exists$ v' Before-after predicate	$\exists v' BA(s, c, v, v')$

(2)

2.3.3 *Refinement*

The concept of refinement in Event-B allows more detail, and the expression of some design decisions, to be added, in a stepwise manner, into the model. The advantage of refinement is allowing the model to be analysed at an abstract level, resulting in reduced complexity (Abrial and Hallerstede, 2006). In the formal development, the first Event-B model is called the abstract model. The abstract model is transformed through a formal sequence to obtain the refinement/concrete model. Performing refinements can be done in many ways, such as adding new variables and constants, introducing new events, decomposition of events, changing/adding algorithms detail, and replacing existing variables.

Refinement is sub-categorized into *feature augmentation* and *structural refinement* (Butler*, et al.*, 2008).

- *Feature augmentation*: a feature augmentation is a refinement in which existing model features are maintained and additional features are added, such as variables, invariant, events, additional guards and actions. This kind of refinement defines new properties for a model. It can be called a *superposition* or a *horizontal refinement*.
- *Structural refinement*: this refinement is adding detailed design to the implementation. Examples of structural refinements are refining the algorithm of an event's operators, event decomposition, and replacing an existing event's variable with new variables. This refinement can be called a *procedural refinement* or a *vertical refinement*.

When a *Structural refinement* is applied to a model, *gluing invariants* must be introduced. A *gluing invariant* links the state of the concrete model to the states of its abstract model. For example, one performs a refinement when a variable v in the abstract model is replaced by a variable w in the concrete model. In this case, a *gluing invariant* $J(v, w)$ is used to glue variable v to the variable w mathematically. Thus, the states of abstract machines are related to the states of refinement machines. An example of defining a *gluing invariant* is now given.

Model $A(v)$ has a variable v defined by $v \subseteq T$, where T is a set of integers; model $B(w)$ has a variable w that represents a sequence of integers and is defined by $w \in seq(T)$. A possible refinement of model A by model B has gluing invariant $J(v, w) \cong v = ran(w)$. This gluing invariant includes the abstract variable v and is called a gluing invariant because it glues the two models together. It is used to relate new variables to those in the abstract models.

The general form of a refinement model is shown in Figure 2-5 where w represents concrete variables, $J(s, c, v, w)$ gluing invariants, and N concrete initialisation. $H(s, c, w)$ and $R(s, c, w)$ are guards and actions for concrete event $Er1$ respectively.

MACHINE *refinement_model_name*
  REFINES *abstract_model_name*
    SEES *context_name*
    VARIABLES *w*
    INVARIANT *J*(*s*, *c*, *v*, *w*)
    INITIALISATION *N*
    EVENTS
      Er1 **REFINES** *E1* = **WHEN** *H(s, c, w)* **THEN** *R(s, c, w)* **END**
      Er2 **REFINES** *E2* = **ANY** … **WHERE**…**THEN** … **END**
      …
END

Figure 2-5 Refinement model structure

Consistency Proofs for Refinement

Since new events can be introduced in the refinement, the new events also have to be proved. For example, it is necessary to prove that the new events will not run forever, or, when a concrete event in the new event is enabled, the corresponding abstract one is enabled. The latter is called *Guard strengthening* (**GRD**) and is an example of a PO illustrated in the following formula (Abrial, 2008b). Other numbers of POs can be found in (Métayer*, et al.*, 2005).

Axioms	$A(s,c)$
Abstract invariants and theorems	$I(s, c, v)$
Concrete invariants and	$J(s, c, v, w)$
Concrete event guards	$H(s, c, w)$
$\Rightarrow$	$\Rightarrow$
Abstract event specific guard	$g(s, c, v)$

(3)

2.4 RODIN Tools

The RODIN toolkit version 0.9.1 (Event-B.org, 2009), used in this thesis, is an Eclipse environment for modelling and proof in Event-B. RODIN is built on the Eclipse platform and comprises many features, for example, refinement, PO generation and some plug-in tools. Some of the latter are: Atelier B (ClearSy, 2009), ProB (ProB, 2009), UML-B (Snook and Butler, 2008b), and B2Latex (Event-B.org, 2008). The RODIN tool has two default perspectives as shown in Figure 2-6 and Figure 2-7.

In RODIN, Event-B **CONTEXTS**, **MACHINES** and their refinements, are created within the same project as shown in the *Project Explorer* tab in Figure 2-6. The *Editor* tab (in the centre) is for editing a model whose elements' properties are shown in the *Properties* tab beneath. The *Outline* tab displays the list of model elements.

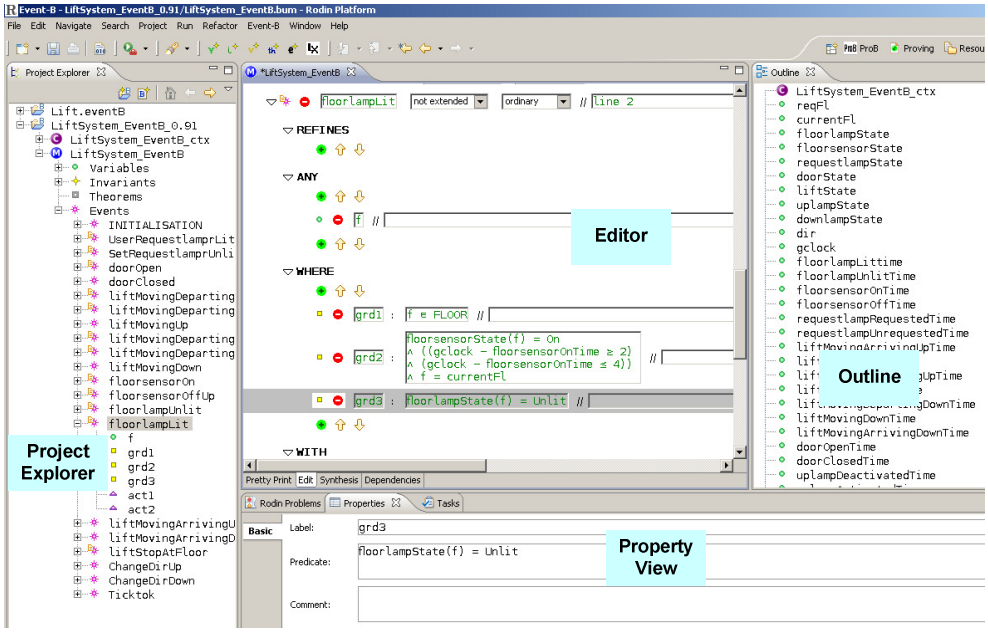


Figure 2-6 RODIN Modelling Perspective

The RODIN tool contains a proof obligation generator, automated and interactive provers (Abrial, *et al.*, 2008). The automated and interactive proof is shown in the *Obligations Explorer* tab, Figure 2-7. To perform interactive proof, one can select hypotheses from the *Selected Hypotheses* tab (in the upper centre). The *Proof Tree* and *Goal* tabs display the sequence of proving, and the goal of proving, respectively. The proved result and a number of provers (provided by the tool) are in the *Proof Control* panel.

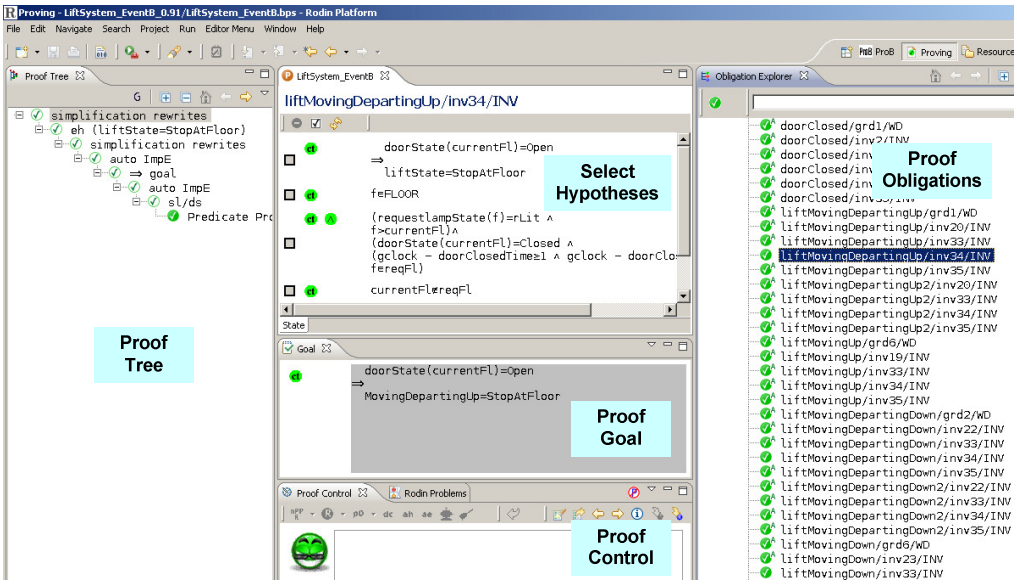


Figure 2-7 RODIN Proving Perspective

2.5 UML-B

UML-B (Snook and Butler, 2008b) has been developed as a plug-in for RODIN toolkits and implemented by the Eclipse Modelling Framework (EMF). It is a graphical formal modelling notation based on UML (Booch, *et al.*, 2003), and relies on Event-B (Abrial, *et al.*, 2007) and its verification tool (Abrial, *et al.*, 2005). UML-B is a tool that supports the construction of a graphical model, using UML-like diagrams, i.e. Class diagram and Statemachines, and an Event-B like annotation language. UML-B models can then be automatically translated to Event-B using the U2B translator for further analysis. In this thesis, the UML-B version 0.4.3 is used.

2.5.1 *Package diagram*

The UML-B top-level *Package diagram* is first opened with an empty canvas. This is the default perspective for representing a UML-B project.

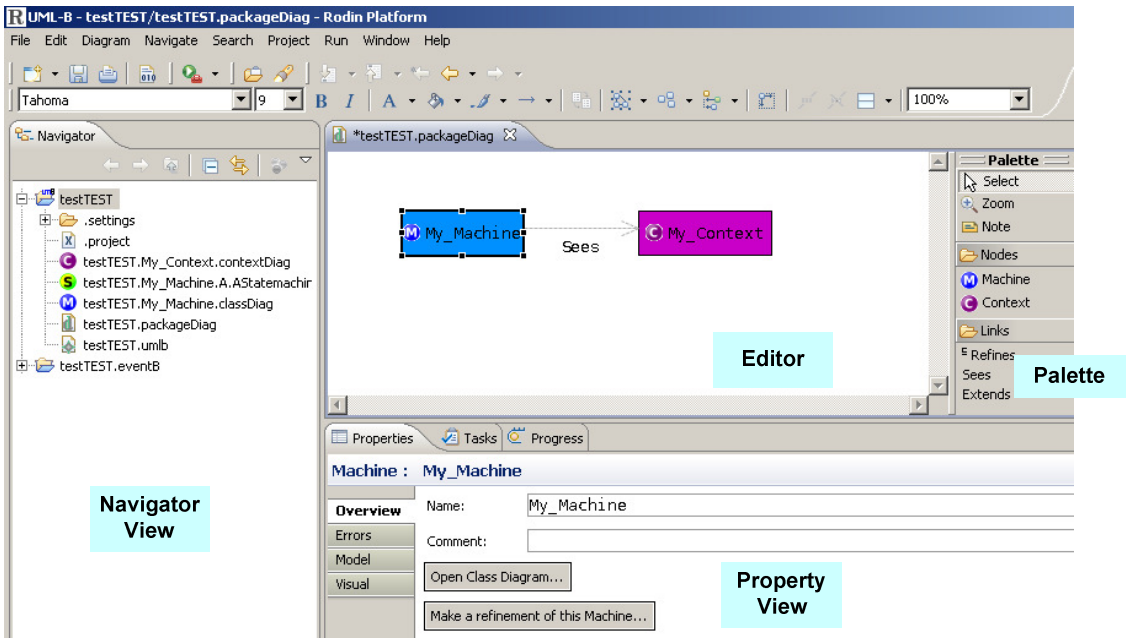


Figure 2-8 UML-B Package diagram perspective

A *Package diagram* is used to describe the association between machines and contexts in a UML-B project. UML-B provides drawing tools as illustrated in the *Palette* panel, on the right. This is used to create machines and contexts with a graphical representation as shown in the *Editor* panel in the centre. For example, in Figure 2-8, `My_Machine` is a machine while `My_Context` is a context. A machine sees a context via the relationship `Sees`. The *Properties* tab represents properties of the selected component in the *Editor* view, while the *Navigator* tab is for displaying the list of diagrams within a project.

2.5.2 *Context diagram*

Static data in Event-B, such as sets, constants, axioms and theorems, are modelled in the **CONTEXT** part. UML-B provides this in a separate package called a *Context diagram*. The *Context diagram* is drawn as a Class diagram but has constant data represented by *ClassType*, *Attributes, Constants* and *Association*.

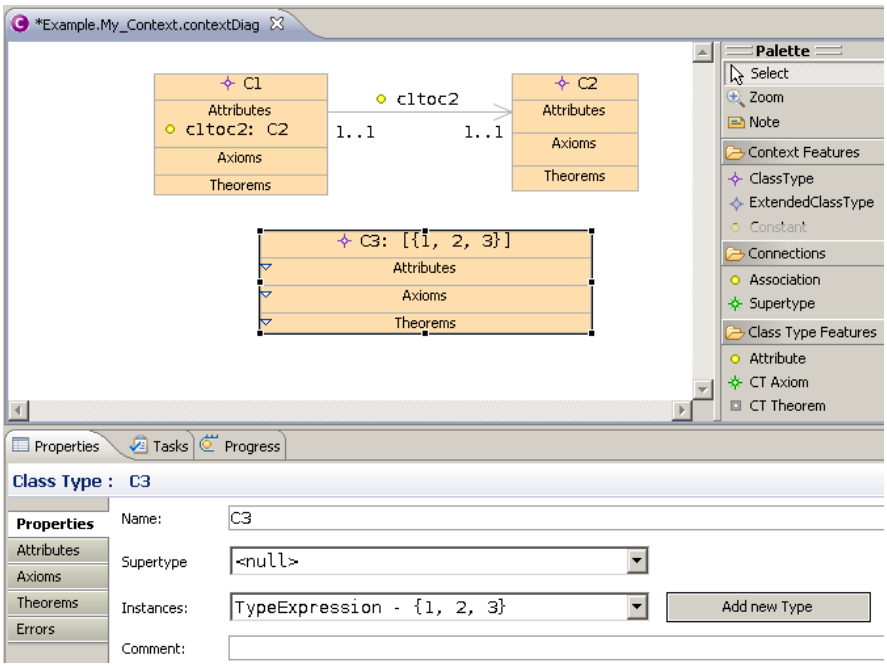


Figure 2-9 UML-B Context diagram perspective

Whenever a UML-B model generates an Event-B model, *ClassTypes* are defined as carrier sets or constants. In Figure 2-9, *ClassType* `C1` and `C2` are defined

as sets, while *ClassType* `C3` is defined as a constant as shown in Figure 2-10. `C3` is generated as a constant since it is assigned to constant values `{1, 2, 3}`. An association between *ClassType*, for example `c1toc2`, is also generated as a constant with a corresponding axiom as shown below.

```
CONTEXT
  My_Context
SETS
  C1      //   ClassType
  C2      //   ClassType
CONSTANTS
  C3      //   classType instances
  c1toc2      //   attribute of C1
AXIOMS
  C3.value    :   C3 = {1, 2, 3}
  c1toc2.type    :   c1toc2 ∈ C1 ⇸ C2
END
```

Figure 2-10 Event-B

2.5.3 *Class diagrams*

The dynamic part is generated in a Class diagram and used to describe a machine. In a machine, one can define classes, variables, events, Statemachines and invariants.

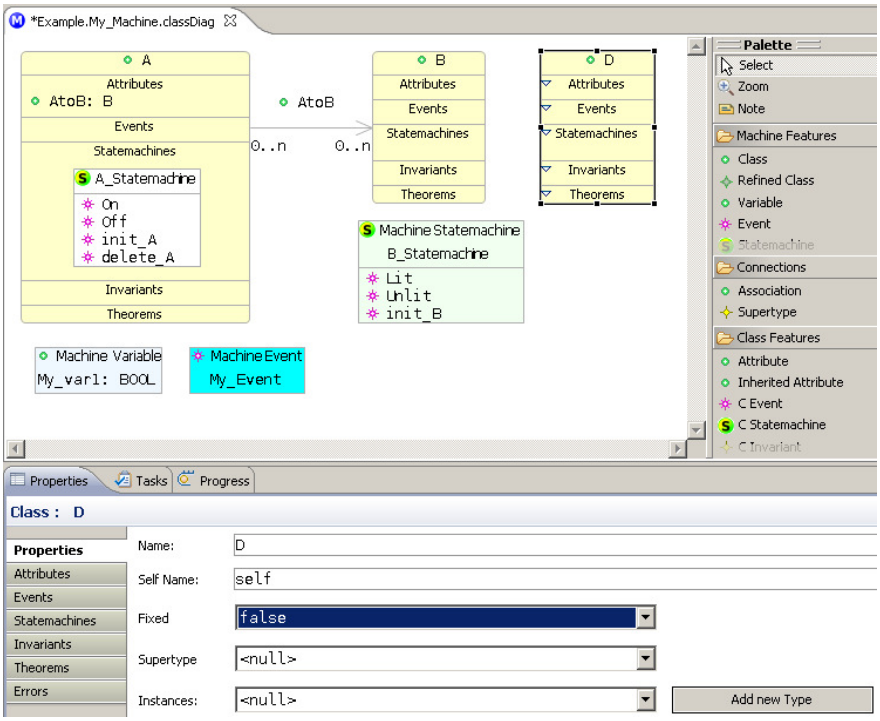


Figure 2-11 UML-B Context diagram perspective

"Classes represent subsets (variable or fixed) of the *ClassType* that were introduced in the context" (Snook and Butler, 2008b). That means a class's *Fixed* property can be set to `false` (default value) or `true`. If it is set to `false`, that class is generated as a variable unless it is a SET. For example, in Figure 2-11, class `D` is generated as a variable for a machine with its invariants `D ∈ ℙ (D_SET)` as shown in Figure 2-12.

```
VARIABLES
  A      //   class instances
  B      //   class instances
  D      //   class instances
  My_var1      //   utility variable
  AtoB      //   attribute of A
  A_Statemachine      //   statemachine belonging to class, A
  B_Statemachine      //   statemachine belonging to the machine
INVARIANTS
  A.type    :    A ∈ ℙ (A_SET)
  B.type    :    B ∈ ℙ (B_SET)
  D.type    :    D ∈ ℙ (D_SET)
  My_var1.type    :    My_var1 ∈ BOOL
  AtoB.type    :    AtoB ∈ A ⟷ B
  A_Statemachine.type    :    A_Statemachine ∈ A ⟶ A_Statemachine_STATES
  B_Statemachine.type    :    B_Statemachine ∈ B_Statemachine_STATES
```

Figure 2-12 An Event-B variable is generated from an UML-B non-fixed property class

If the *Fixed* property for class `D` is set to `true`, the Event-B generated from class `D` is shown in Figure 2-13.

```
SETS
  A_SET      //   Class
  B_SET      //   Class
  D      //   Class
  A_Statemachine_STATES      //   A-statemachine
  B_Statemachine_STATES      //   statemachine
CONSTANTS
  a1      //   A_Statemachine-state
  a2      //   A_Statemachine-state
  b1      //   B_Statemachine-state
  b2      //   B_Statemachine-state
AXIOMS
  A_Statemachine_STATES.value    :    A_Statemachine_STATES = {a1,a2}
  B_Statemachine_STATES.value    :    B_Statemachine_STATES = {b1,b2}
  a1.type    :    a1 ∈ A_Statemachine_STATES
  a2.type    :    a2 ∈ A_Statemachine_STATES
  b1.type    :    b1 ∈ B_Statemachine_STATES
  b2.type    :    b2 ∈ B_Statemachine_STATES
  distinctStates a2,a1    :    a2 ≠ a1
  distinctStates b2,b1    :    b2 ≠ b1
```

Figure 2-13 An Event-B class is generated from an UML-B fixed property class

Associations between classes, for example an association `AtoB` in Figure 2-11 and Figure 2-12, define machine variables (global variables). Attributes and events that are attached to a class are generated as events' local variables and machine events respectively.

2.5.4 *Statemachines*

A Statemachine is used to model the behaviours of a system. It can be identified in two ways: within a corresponding class, and as a *Machine Statemachine*. A Statemachine is defined within a class in order to explain the behaviour of a class's states changing and modifying a class's variables. In contrast, if an object has to be represented by a Statemachine, a *Machine Statemachine* is utilized. For example, from Figure 2-11, the `A_Statemachine` is defined within class `A` while `B_Statemachine` is a *Machine Statemachine*. Below is an example of the `A_Statemachine`.

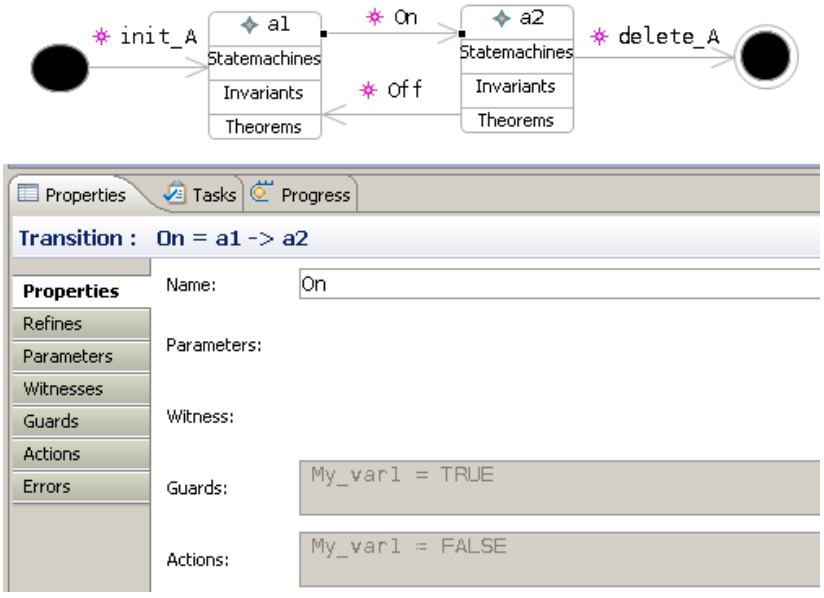


Figure 2-14 An example Statemachine

A Statemachine transition represents an event with behaviour associated with the change of states, from a source state to a target state. Each transition is generated as an event. Figure 2-14 shows two events are created: `On` and `Off`. Additional guards and actions can be attached to the transition in the *Properties* tab

to describe the events' behaviours. Note that, `My_var1` is a `Machine Variable` defined in Figure 2-11.

Each event uses a keyword <*ClassNameSelf*>, a class name in which a transition is followed by *Self*, to identify the non-deterministic selection of an instance of the class. For example, consider the event `On` created from the transition `On` in Figure 2-14, as illustrated below.

```
On   ≙
STATUS
  ordinary
ANY
  ASelf      //   contextual instance of class A
WHERE
  Aself.type   :   ASelf ∈ A
  A_Statemachine_isin_a1   :   A_Statemachine(ASelf) = a1
  On.Guard1   :   My_var1 = TRUE
THEN
  A_Statemachine_enterState_a2   :   A_Statemachine(ASelf) ≔ a2
  On.Action1   :   My_var1 ≔ FALSE
END
```

Figure 2-15 An event On created from a transition

The word `ASelf` is automatically created as a non-deterministic variable with a guard `ASelf ∈ A`, where `A` represents a class in which this transition takes place. A source state (`A_Statemachine(ASelf) = a1`) and a target state (`A_Statemachine(ASelf) ≔ a2`) are automatically generated as a guard and an action respectively.

2.5.5 *Implementation of UML-B*

UML-B is implemented with the EMF, which is an Eclipse project providing code generation, model editor, and efficient Application Programming Interface (API) utilities based on a metamodel (Snook and Butler, 2006). Graphical Modelling Framework (GMF) is an Eclipse project used to automatically generate code for the UML-B graphical modelling tool, based on the EMF model (detail of EMF is given in section 2.8.2). UML-B provides drawing tools and a translator to generate Event-B models, i.e. whenever an UML-B drawing model is saved, the

U2B translator automatically generates the corresponding Event-B model. RODIN automatically verifies the Event-B model and reports any errors.

Even though UML-B is similar to UML, it is designed on a separate metamodel (Snook and Butler, 2008a). Figure 2-16 shows parts of the UML-B metamodel in which classes represent abstract meta-classes. Class **UMLBProject** defines the name of a project via **UMLBname** where *name* is defined as a string. **UMLBProject** is composed of **UMLBconstruct** in which is subtyped into **UMLBMachine** and **UMLBContext**. **UMLBMachine** contains **UMLBEvent** and **UMLBVariable**, which are used to define machine events and machine variables respectively. The class **UMLBMachine** contains a *contexts* association. This is the way that machines are linked to contexts in a model.

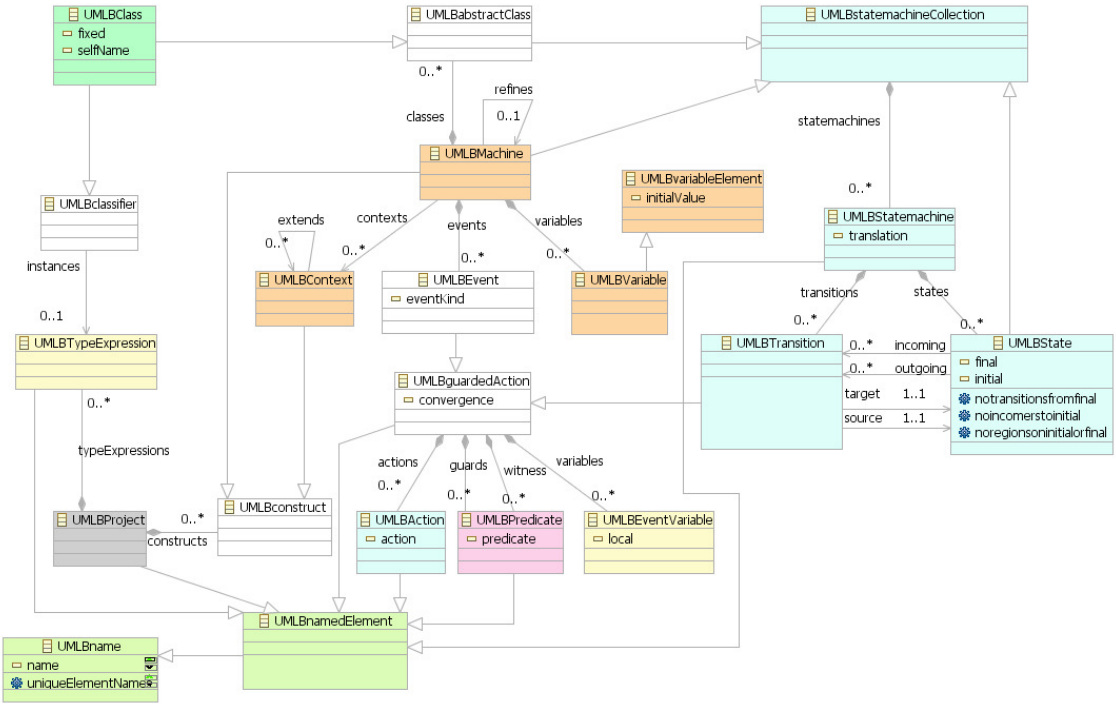


Figure 2-16 Parts of UML-B Metamodel

UMLBClass is a subtype of **UMLBabstractClass**. As shown in Figure 2-17, the **UMLBabstractClass** contains **UMLBEvent** and **UMLBabstractAttribute**, which are used to define classes' events and classes' attributes respectively.

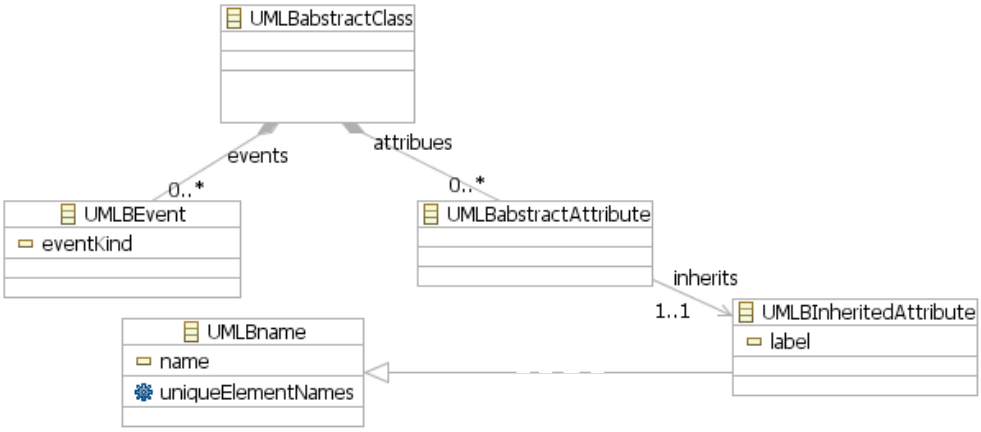


Figure 2-17 UMLBabstractClass, UMLBEvent and UMLBabstractAttribute Metamodel

In Figure 2-16, **UMLBstatemachineCollection** contains **UMLBStatemachine** that is used to define Statemachines. The **UMLBStatemachine** contains **UMLBTransition** and **UMLBState**. The **UMLBTransition** represents Statemachines' transitions, in which each transition links a couple of states by *target* and *source* associations to **UMLBState**. The **UMLBTransition** is a subtype of **UMLBguardedAction**. The **UMLBguardedAction** contains **UMLBAction**, **UMLBPredicate** and **UMLBEventVariable**, which are used to define actions, guards and events' variables (local variables for an event) respectively for a transition.

2.6 Linear Temporal Logic (LTL)

LTL is used to describe a sequence of events referring to time. It is defined over discrete time points and has proved convenient for specification requirements (Letier, 2001). LTL provides the temporal operators as follows.

◊	some time in the future	♦	some time in the past
□	always in the future	■	always in the past
U	always in the future *until*	***S***	always in the past *since*
W	always in the future *unless*	***B***	always in the past *back to*
○	in the next state	●	in the previous state

In LTL (Lamsweerde, 2009), time is declared as the set *Nat* of natural numbers, and a history H is defined as a function, H: $Nat \rightarrow State(x)$, where x represents the set of system variables and $State(x)$ stands for the set of all possible states for the corresponding variables in x. This function operates for every time point i in H. To define the LTL semantics more precisely, the notion $(H, i) \models P$ is used to express the LTL assertion that P is satisfied by history H at time position i, where $i \in Nat$. The semantic rules for the LTL temporal operators are divided into two categories: future operators and past operators, as follows (taken from (Lamsweerde, 2009)).

Future operators

$(H, i) \models \Diamond P$	**iff** for some $j \geq i$: $(H, j) \models P$	
$(H, i) \models \Box P$	**iff** for every $j \geq i$: $(H, j) \models P$	
$(H, i) \models P \, \mathbf{U} \, Q$	**iff** there exists a $j \geq i$ such that $(H, j) \models Q$ and for every k, $i \leq k < j$: $(H, k) \models P$	
$(H, i) \models P \, \mathbf{W} \, Q$	**iff** $(H, i) \models P \, \mathbf{U} \, Q$ or $(H, i) \models \Box P$	
$(H, i) \models \circ P$	**iff** $(H, i+1) \models P$	
$P \Rightarrow Q$	"entails"	Equivalent to $\Box (P \rightarrow Q)$
$P \Leftrightarrow Q$	"congruent"	Equivalent to $\Box (P \leftrightarrow Q)$

Past operators

$(H, i) \models \blacklozenge P$	**iff** for some $j \leq i$: $(H, j) \models P$	
$(H, i) \models \blacksquare P$	**iff** for every $j \leq i$: $(H, j) \models P$	
$(H, i) \models P \, \mathbf{S} \, Q$	**iff** there exists a $j \leq i$ such that $(H, j) \models Q$ and for every k, $j < k \leq i$: $(H, k) \models P$	
$(H, i) \models P \, \mathbf{B} \, Q$	**iff** $(H, i) \models P \, \mathbf{S} \, Q$ or $(H, i) \models \blacksquare P$	
$(H, i) \models \bullet P$	**iff** $(H, i-1) \models P$ with $i > 0$	
$@ P$	Equivalent to $(\bullet \neg P) \wedge P$	

Relative Real-time Properties

In RE, some properties are need to be defined over real-time constraints. Examples of such properties are:

"*All borrowed books must be returned within a week*"

"*Lift door must be opened between 1 and 5 minutes after the lift stops at that floor*"

Relative real-time properties are properties referring to real-time delays between system states. In order to specify such properties, bounded versions of the above temporal operators are used. Examples of those operators are

$\Diamond_{\leq d}$ (some time in the future within deadline d)
$\Box_{\leq d}$ (always in the future up to deadline d)

To define those operators, a *temporal distance* function is used, as defined in the following:

$dist$: $Nat \times Nat \rightarrow D$ where $D = \{d \mid$ there exists a natural n such that $d = n \times u\}$ where u denotes a chosen time unit such as second, minute and hour.

$dist(i, j) = |j - i| \times u$

For example, the semantics of the real-time operators is then defined below (the rest of the semantics is declared in (Lamsweerde, 2009)).

$(H, i) \models \Diamond_{\leq d} P$ **iff** for some $j \geq i$ with $dist(i, j) \leq d$: : $(H, j) \models P$
$(H, i) \models \Box_{\leq d} P$ **iff** for every $j \geq i$ such that $dist(i, j) \leq d$: : $(H, j) \models P$

2.7 Knowledge Acquisition in autOmated Specification (KAOS)

A system requirement is a statement of what the system has to perform to accomplish the system's goal. A requirement for a computer system specifies a statement to be implemented by the proposed system. It always involves other system components and is described in terms of environmental phenomena (e.g. agents and system's constraints). Examples of system requirements are

- All lift doors shall always remain closed while the lift is moving
- A book must be returned within a deadline

A goal is a prescriptive statement and defines an objective the composite system should meet through the cooperation of its agents in the environment. A requirement is a goal under the responsibility of a single software agent. An agent is an active object and performs a specific role/operation in a requirement. Agents can be human, devices, software, etc. For example, the first requirement in the above list is assigned to a *DoorController* agent, while the second is assigned to a *Library software* agent.

KAOS stands for Knowledge Acquisition in autOmated Specification (Dardenne*, et al.*, 1993) or Keep All Objects Satisfied (Letier and Lamsweerde, 2002b). It is a goal-oriented RE that uses a *Goal model* to generate system requirements. A *Goal model* is then used to generate one or more *Operation model*s. Each *Operation model* defines the state transitions in the application domain by using pre- and post-conditions. The detail of *Goal* and *Operation models* are described in the following sections.

2.7.1 *Goal model*

The first model generated in KAOS is a *Goal model*. A *Goal model* is created by focusing on a part of the goal and then proceeding to the next part until completing the whole *Goal model* (El-Maddah and Maibaum, 2003). This process is called goal refinement. A *Goal model* is represented as a tree structure, which can be called a *Goal tree*, as shown in Figure 2-20. Each goal is graphically represented by a parallelogram labelled by the goal's name and prefixed by its type, as shown in Figure 2-18.

There are four types of goal (Darimont and Lamsweerde, 1996; Rubio-Loyola*, et al.*, 2005) :

- *Achieve* and *Cease* goals require some target properties to be eventually satisfied or denied, respectively, in some future state. This goal category is used for specification of liveness properties.
- *Maintain* and *Avoid* goals require some target properties to be permanently satisfied or denied, respectively, in every future state.

Maintain and *Avoid* goal categories are used to specify safety properties.

To illustrate how a KAOS model is created, an example of a meeting scheduling problem is used as an example from this point forward (Letier, 2001). Parts of the meeting scheduling problem specification are provided below:

"*… Each requested meeting will eventually be held with the presence of all intended participants. Participants' date constraints are eventually accurately known by the scheduler …*"

From the problem statement above, supposes a "*participants' constraints known*" is selected to be generated as a goal. This goal is created as shown in Figure 2-18. The goal has identified **Achieve** as a goal type with a name PrtcptsCstrKnow.

Figure 2-18 An example of a goal

2.7.2 *Goal formal definition*

Each goal is declared by a type (*Achieve*, *Maintain*, *Cease* and *Avoid*), definition (**Definition**) and formal definition (**FormalDef**). A goal definition is described by text. A formal definition is composed of optional inputs/outputs, pre-conditions and post-conditions. Inputs/outputs declare objects' attributes of an operation. Pre- and post-conditions describe current conditions and target conditions of an operation, respectively. A goal formal definition uses LTL to define a goal description. Thus, a goal formal definition is written as a temporal logic statement. An example of the definition of the Goal **Achieve**[PrtcptsCstrKnown] from Figure 2-18 is illustrated below:

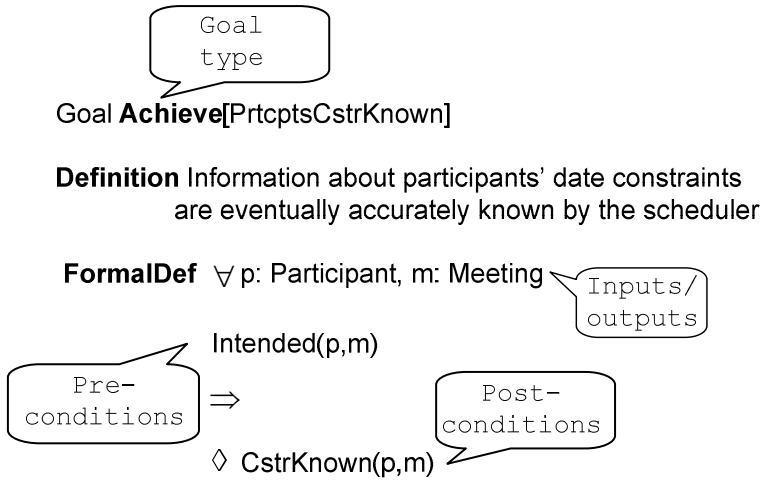


Figure 2-19 A definition of the goal Achieve[PrtcptsCstrKnown]

Goal types are keywords that allow one to specify a goal formal definition pattern at the declaration level (Lamsweerde and Willemet, 1998). Each of these goal patterns represents a particular shape of temporal logic formula. Examples of those patterns are illustrated in

Table 2-1.

Goal Type	Temporal logic formula	Pattern
Achieve	$P \Rightarrow \Diamond Q$	Unbounded Achieve
	$P \Rightarrow \Diamond_{\leq d} Q$	Bounded Achieve
	$P \Rightarrow \circ Q$	Immediate Achieve
Cease	$P \Rightarrow \Diamond \neg Q$	Unbounded Cease
	$P \Rightarrow \Diamond_{\leq d} \neg Q$	Bounded Cease
	$P \Rightarrow \circ \neg Q$	Immediate Cease
Maintain	$P \Rightarrow Q$	Permanent Maintain/ Immediately response
	$P \Rightarrow \Box Q$	After Maintain
Avoid	$P \Rightarrow \neg Q$	Permanent Avoid
	…	…

Table 2-1 Goal types with temporal logic formulas

For example from Figure 2-19, the Goal **Achieve**[PrtcptsCstrKnown] is specified as a *Unbounded Achieve* $P \Rightarrow \Diamond Q$ where P is Intended(p,m) and Q is CstrKnown(p,m).

Another example is a *Bounded Achieve* $P \Rightarrow \Diamond_{\leq d} Q$, it means whenever the current condition P occurred, the target condition Q will eventually occur within deadline d. The *Permanent Maintain/Immediately response* $P \Rightarrow Q$ means whenever the current condition P occurs, the target condition Q must be occurred at the same time point. More goal patterns can be found in (Letier, 2001).

2.7.3 *Goal refinement*

A *Goal model* is created by an AND/OR graph called a *goal refinement graph*. A *goal refinement graph* shows how a parent goal (at a higher-level) is refined into subgoals, and how subgoals are grouped into the higher-level one (Lamsweerde, 2001); this is called goal refinement. Asking WHY and HOW questions are techniques used to generate a *goal refinement graph*. By asking HOW questions, subgoals are identified from an already identified parent goal (top-down processes). By asking WHY questions, a parent goal is generated from already identified subgoals (bottom-up processes). The goal refinement is stopped when every subgoal can be assigned to a single agent. Leaf node goals in a *goal refinement graph* represent software requirements.

To explain how a *goal refinement graph* is created, consider the goal ***Achieve****[PrtcptsCstrKnown]* as shown below (the same goal within Figure 2-18). The goal ***Achieve****[PrtcptsCstrKnown]* is refined into two subgoals ***Achieve****[PrtcptsCstrRequested]* and ***Achieve****[RequestedCstrProvided]* by asking a HOW question. Similarly, other parts of the goal refinement graph are generated by asking HOW and WHY questions. A process of goal refinement is brought about by application of formal goal refinement patterns to expand the parent goal is described later in section 2.7.4.

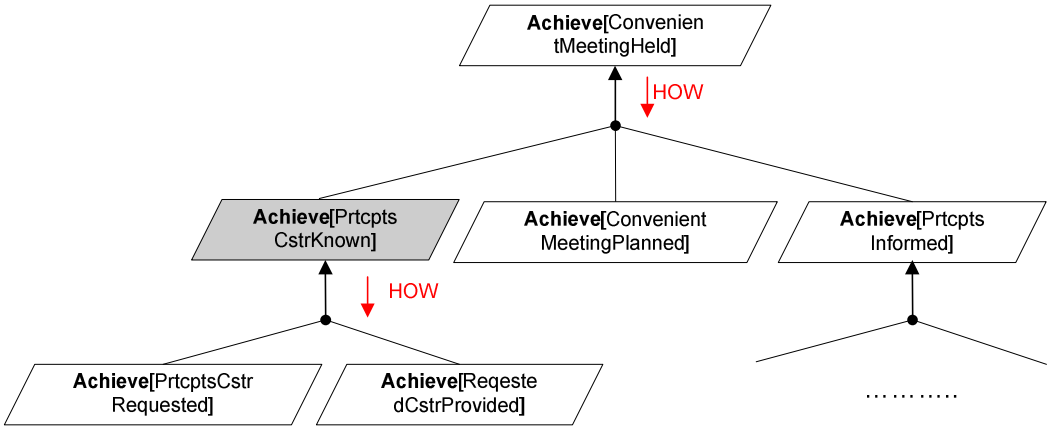


Figure 2-20 KAOS goal refinement graph

AND and OR refinement combinations

A goal refinement process uses logic to decompose a parent goal into subgoals, or compose subgoals to generate a parent goal. Decomposing and composing use two kinds of goal refinements in combination: AND and OR. An AND-refinement is represented by a black circle symbol while an OR-refinement is represented by a white circle symbol, as shown below.

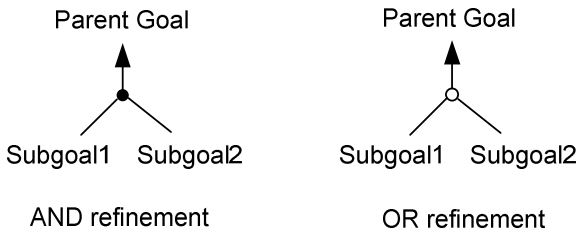


Figure 2-21 Symbols for AND and OR refinement

Using AND-refinement means a parent goal can be refined into subgoals that are more detailed; for example, Subgoal1 and Subgoal2. This means that to achieve a parent goal, all subgoals must be selected. OR-refinement is an alternative goal refinement. In this case, more than one alternative subgoal can be selected.

2.7.4 *Formal goal refinement patterns*

"Goal decompositions made by hand are usually incomplete and sometimes inconsistent" (Lamsweerde and Massonet, 1995). Thus, Darimont provides formal patterns for building goal refinement graphs that are proved correct (Darimont and Lamsweerde, 1996). A formal refinement pattern is a one-level AND-tree of a parent goal. That means there is no pattern for OR-refinement. There are a number of goal refinement pattern defined in (Darimont, 1995). Here, we explain those are used in the thesis.

A Milestone-driven goal refinement pattern

The Milestone-driven goal refinement pattern refines an Unbounded Achieve goal of the form $P \Rightarrow \Diamond Q$ by introducing an intermediate state M (milestone), see Figure 2-22. To reach a state satisfying the target condition Q from a state satisfying the condition P, it must act via the intermediate state satisfying condition M.

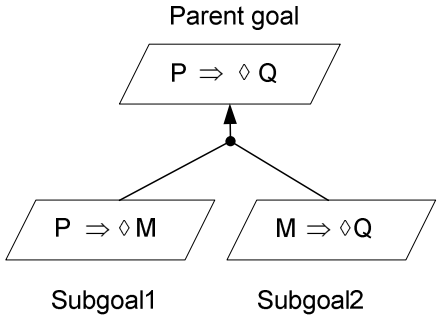


Figure 2-22 A Milestone-driven goal refinement pattern
(Darimont and Lamsweerde, 1996; Letier, 2001)

For example from Figure 2-19, the goal **Achieve**[PrtcptsCstrKnown] is refined into two subgoals ***Achieve****[PrtcptsCstrRequested]* and ***Achieve****[RequestedCstr Provided]* by using the Milestone-driven goal refinement pattern where

P : Intended(p,m)
Q: CstrKnown(p,m)
M: CstrRequested(p,m)

The following subgoals are thereby obtained:

Goal ***Achieve****[PrtcptsCstrRequested]*
FormalDef $\forall$ *p: Participant, m: Meeting*
$Intended(p,m) \Rightarrow \Diamond CstrRequested(p,m)$

Goal ***Achieve****[RequestedCstrProvided]*
FormalDef $\forall$ *p: Participant, m: Meeting*
$CstrRequested(p,m) \Rightarrow \Diamond CstrKnown(p,m)$

A case-driven goal refinement pattern: s**plit antecedent**

The Case-driven: split antecedent goal refinement tactic refines a goal by splitting it into cases as shown in Figure 2-23. This technique is used when different cases can be identified in a goal.

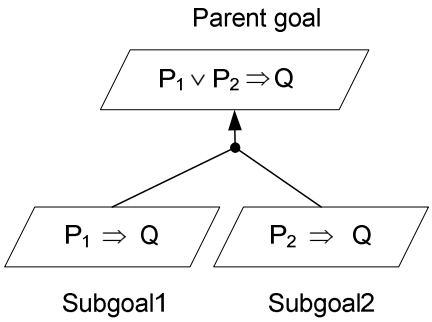


Figure 2-23 A case-driven goal refinement pattern: split antecedent

For example, a fire-safety monitoring problem is provided as "… If the room temperature is overheated or the room is very humid, a room window will be eventually opened …"

This specification above can be generate as a Goal **Achieve**[TheRoom WindowOpenAfterTheRoomIsOverHeatedOrTheRoomIsHumid] as in the following:

Goal ***Achieve****[TheRoomWindowOpenAfterTheRoomIsOverHeatedOrTheRoomIsHumid]*
FormalDefinition: *When the room temperature is overheated or the room is very humid, a room window will be eventually opened.*
FormalDef: $\forall$ *r : Room*

$r.temperatureLevel = \text{'Overheated'} \vee r.humidityLevel = \text{'High'}$

$\Rightarrow$

$\Diamond\ r.windowState = \text{'Open'}$

The *Goal* ***Achieve****[TheRoomWindowOpenAfterTheRoomIsOverHeatedOrTheRoomIsHumid]* is refined into two subgoals ***Achieve****[TheRoomWindowOpenAfterTheRoomIsOverHeated]* and ***Achieve****[TheRoomWindowOpenAfterTheRoomHumidityIsHigh]* by using the Case-driven: split antecedent goal refinement where:

P_1 : *r.temperatureLevel = 'Overheated'*

P_2 : *r.humidityLevel = 'High'*

Q : ◊ *r.windowState = 'Open'*

The following subgoals are thereby obtained:

Goal ***Achieve****[TheRoomWindowOpenAfterTheRoomIsOverHeated]*
FormalDef ∀ *r :Room*
r.temperatureLevel = 'Overheated' ⇒ ◊ *r.windowState = 'Open'*

Goal ***Achieve****[TheRoomWindowOpenAfterTheRoomHumidityIsHigh]*
FormalDef ∀ *r :Room*
r.humidityLevel = 'High'⇒ ◊ *r.windowState = 'Open'*

2.7.5 *Operation model*

Once subgoal-agent allocation is complete, each leaf node goal is assigned to an operation. The operations are defined by the following conditions (this section is taken from (Lamsweerde, 2009)):

- A domain pre-condition (*DomPre*) characterizing the input states when the operation is applied.
- A domain post-conditon (*DomPost*) characterizing the output states when the operation has been applied.
- Required pre-condition (*ReqPre*) is a condition on the operation's input states for satisfaction of the goal. It captures a *permission*; under this condition the operation *may* be applied when the domain pre-condition holds.
- Required trigger condition (*ReqTrig*) is a condition on the operation's input states for satisfaction of the goal. It captures an *obligation*; under this condition, the operation *must* be applied when the domain pre-condition holds.

- Required post-condition (*ReqPost*) is a condition on the operations' output states for satisfaction of the goal. It captures an additional effect that the operation must have specifically to ensure the goal.

Note that the operation is not applied if a trigger condition becomes true in a state where the operation's domain pre-condition is not true. If the domain pre-condition becomes subsequently true and the trigger condition is still true, the operation must be applied.

The operation is not applied if a required pre-condition becomes true in a state where the operation's domain pre-condition is not true. If the domain pre-condition becomes subsequently true and the required pre-condition is still true, then the operation may be applied – but not necessarily.

There are a number of operation model patterns as defined in (Letier, 2001). Here, we explain those that are used in this thesis.

Operation model: Global Invariant

The goal Permanent Maintain/Immediately response of the form $P \Rightarrow Q$ has an operation model, which is called Global invariant, as illustrated in the following:

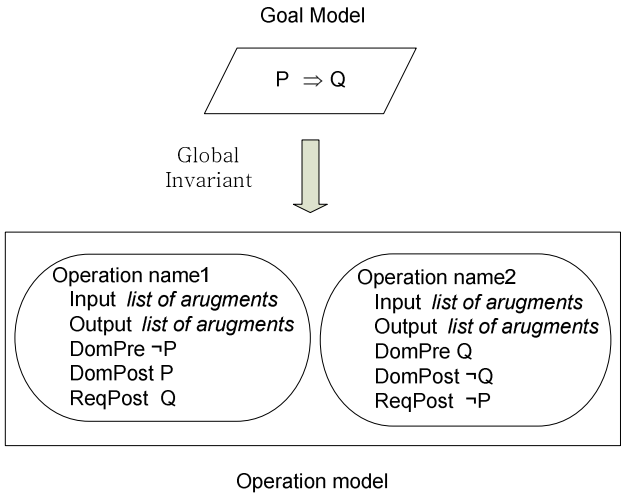


Figure 2-24 Operation model: Global invariant

For example, suppose a simple fire alarm problem is identified as "… fire alarm is set to switch on immediately after the carbon monoxide level inside that room is critical …".

The formal definition of the Goal **Maintain**[FireAlarmsOn] which corresponds to this problem is shown below.

Goal ***Maintain****[FireAlarmsOn]*
FormalDef $\forall$ *r :Room, f : FireAlarm*
$r.CO_2Level$ = *'Critical'* $\Rightarrow$ *f.State = 'On'*

Thus, the two corresponding operations: FireAlarmOn and FireAlarmOff are defined as in the following, where

P : r.CO_2Level = 'Critical'
Q : f.State = 'On'

Operation FireAlarmOn
Input r : Room
Output r : Room
DomPre r.CO_2Level $\neq$ 'Critical'
DomPost r.CO_2Level = 'Critical'
ReqPost f.State = 'On'

Operation FireAlarmOff
Input f : FireAlarm
Output f : FireAlarm
DomPre f.State = 'On'
DomPost f.State = 'Off'
ReqPost r.CO_2Level $\neq$ 'Critical'

Operation model: Bounded Achieve

The goal Bounded Achieve of the form $P \Rightarrow \Diamond_{\leq d} Q$ has an operation model as illustrated in the following:

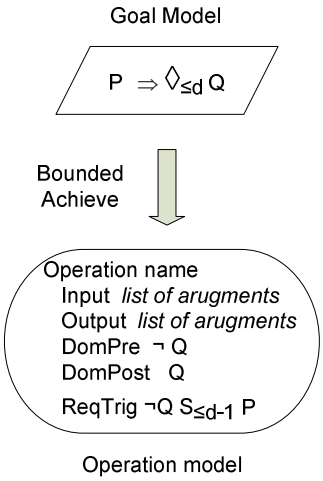


Figure 2-25 Operation model: Bounded achieve

For example, suppose the fire alarm problem is modified to "… fire alarm is set to switch on within time interval of 2-3 seconds after carbon monoxide level inside that room is critical …". The formal definition of the Goal **Achieve**[FireAlarmsOn] is shown below:

Goal ***Achieve****[FireAlarmsOn]*
FormalDef $\forall$ *r :Room, f : FireAlarm*
$r.CO_2Level = \text{'Critical'} \Rightarrow \Diamond_{[2,3]} f.State = \text{'On'}$

A corresponding goal model is generated by this goal is illustrated in the following, where

P : $r.CO_2Level = \text{'Critical'}$
Q: f.State = 'On'

Operation *FireAlarmOn*
Input *f : FireAlarm*
Output *r: Room, f : FireAlarm*
DomPre *f.State = 'Off'*
DomPost *f.State = 'On'*
ReqTrig *f.State = 'Off'* $\mathbf{S}_{[1,2]}$ $r.CO_2Level = \text{'Critical'}$

2.8 Metamodelling

In software engineering, metamodelling comprises a means of construction, identification rules, frames, and constraints that are useful for modelling software problems. Similarly, it can be said that metamodelling provides a particular model's properties concept. Creating a model always conforms to its metamodel. Metamodels can be defined in many ways. For example, the most well-known are using Meta-Object Facility (OMG-MOF, 2007) and Ecore (EMFT-Eclipse, 2009). The following sections discuss the literature on these examples.

2.8.1 *Meta-Object Facility (MOF)*

MOF (OMG-MOF, 2007) is one of standard technologies developed by the Object Management Group (OMG). It is a language for describing other languages (meta-metamodel). MOF 2.0 is the current standard and has been used as meta-

metamodel for UML2.0 (OMG, 2008), and SysML (SysML, 2008). MOF and UML share core modelling concepts, while MOF reuses UML notation for visualizing metamodels. MOF is a four-layered architecture (numbered M0-M3). Examples of models are defined in each layer shown in Figure 2-26.

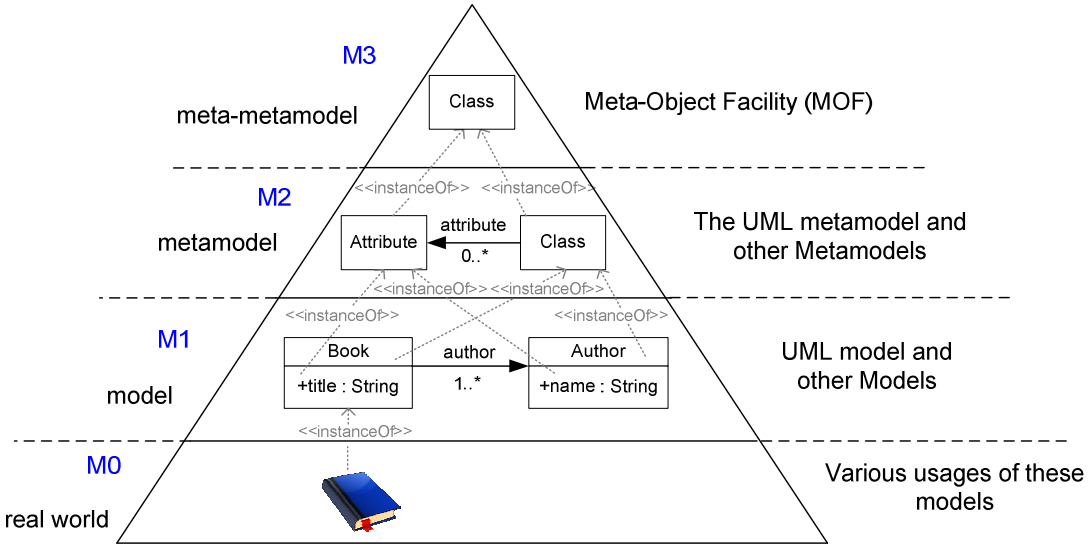


Figure 2-26 Four-layer MOF Architecture

The M3 layer is the meta-metamodel. The meta-metamodel is a mechanism for building metamodels. The well-known models defined in the M3 layer are MOF itself and Ecore (Budinsky, *et al.*, 2003b).

The M2 layer consists of metamodel descriptions. These metamodels are used to define syntax and semantic of M1 elements. Examples of languages described in the M2 layer are UML, XML, JAVA, Event-B languages, and our TDs.

The M1 layer consists of model instances conforming to the M2 metamodel layer. Examples of models in the M1 layer are model written using UML diagrams, i.e. specific Class diagrams and state machines.

The M0 layer comprises real world objects. These might be actual data values and model instances, e.g. object diagrams.

2.8.2 *Eclipse Modelling Framework*

Eclipse Modelling Framework (EMF) (Budinsky, *et al.*, 2003a; Eclipse, 2008) is one implementation of Meta Object Facility (MOF). EMF was started as a MOF of the OMG implementation and is an enhancement of MOF2.0. It is open source and is used for "modelling frameworks and a code generation facility for building tools and other applications based on a structured data model" (Eclipse, 2008). EMF specifies a model by identifying its objects, attributes, relationships between objects, object operations and object constraints, such as multiplicity.

Ecore, which is an EMF model and metamodel itself, is a model used to represent models in EMF. An Ecore model can be generated in any of these forms: Java interfaces, XML Schema or UML diagrams. That is, one can write a Java program to declare a model, or define a model as an XML file. The last option is using UML diagram to create and edit a model. These forms give the same information, just different representations. In summary, one can choose any of them that matches this perspective and EMF can generate the others.

EMF includes a graphical Ecore editor (EMFT-Eclipse, 2009) based on UML notations. For example, Figure 2-27 shows part of a TD metamodel represented by a UML diagram. This UML diagram defines relationships between classes **Name**, **TDClass**, **TDParameter**, and **TDTimeline** for TD metamodel. A corresponding generated Ecore model is shown in Figure 2-28.

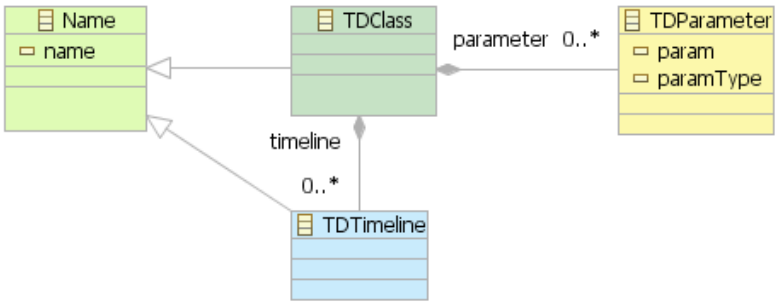


Figure 2-27 Example of UML diagram of interfaces:

TD metamodel (parts of)

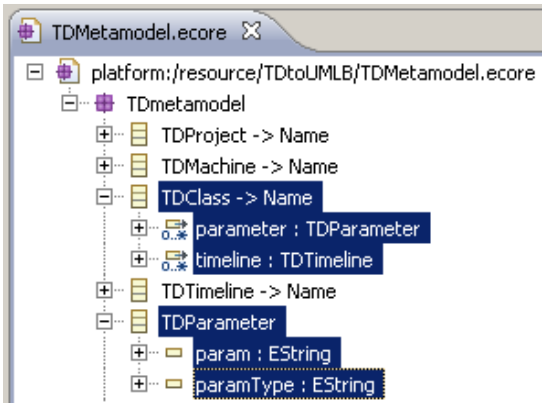


Figure 2-28 Ecore model is generated from a UML diagram

An example of how the UML diagram above is re-represented in a Java interface and an XML file shown below.

- A Java interface is created by EMF

```
public interface TDClass extends Name {

      EList<TDParameter> getParameter();

      EList<TDTimeline> getTimeline();

} // TDClass


public interface TDParameter extends EObject {
      String getParam();

      void setParam(String value);

      String getParamType();

      void setParamType(String value);

} // TDParameter
```

- An XML Schema is generated by EMF

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
  name="TDmetamodel"
  nsURI="ecs.soton.ac.uk" nsPrefix="TDmetamodel">
  <eClassifiers xsi:type="ecore:EClass" name="TDClass"
    eSuperTypes="#//Name">
    <eStructuralFeatures xsi:type="ecore:EReference"
```

```
      name="parameter" upperBound="-1"
      eType="#//TDParameter" containment="true"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="TDParameter">
      <eStructuralFeatures xsi:type="ecore:EAttribute"
       name="param"
      eType="ecore:EDataType"
      http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute"
      name="paramType"
      eType="ecore:EDataType
      http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    </eClassifiers>
</ecore:EPackage>
```

The benefit of EMF is automatic Java code generation and plug-ins. In doing that, an EMF generator generates a *generator model* from an Ecore model (in any of the three forms above). This *generator model* is used to generate code and a plug-in. This is the same process that we used to create a TD plug-in, as shown in Figure 2-29, named **TDmetamodel Model**. The TD plug-in is then used to define a TD instance for transforming TD into UML-B by the Atlas Transformation Language (ATL), which is described in Chapter 6.

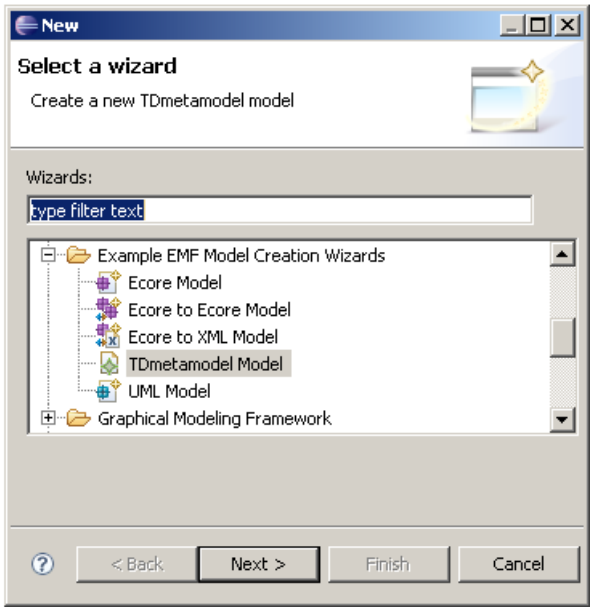


Figure 2-29 TDmetamodel Model Plug-in

2.9 Atlas Transformation Language (ATL)

UML-B is implemented by EMF, which is a metamodel based on an Eclipse project. Similarly, ATL is developed on Eclipse platform and is used to generate a

target model from a source model. Since UML-B and ATL are built on the same platform, it is appropriate to use ATL to transform a TD model into a UML-B model. In order to do that, a TD metamodel is provided and an existing UML-B metamodel is used (detailed in Chapter 6). The explanation of ATL now follows.

ATL is a model transformation language and was developed by the ATLAS INRIA & LINA research group (ATL, 2008). It was developed within the Eclipse platform in which the ATL Integrated Development Environment (IDE) provides a number of development tools such as syntax highlighting and debugging (Allilaire and Idrissi, 2004). 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. The ATL transformation approach is summarised in Figure 2-30 (ATLAS Group, 2008).

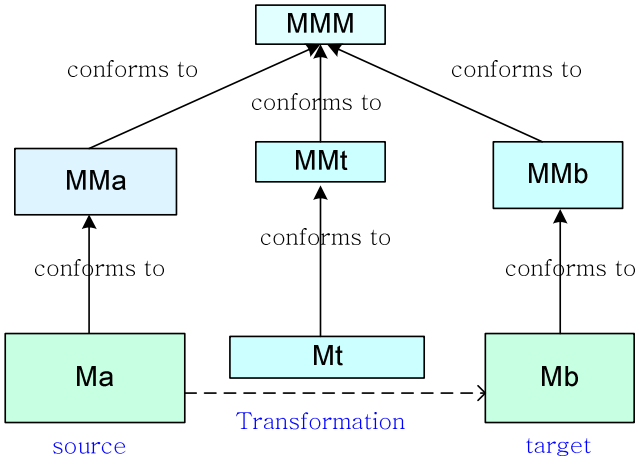


Figure 2-30 ATL transformation approach

A source model *Ma* conforms to a metamodel *MMa* and is transformed into a target model *Mb*, which conforms to a metamodel *MMb*. The transformation definition *Mt* is written in the ATL language. This transformation definition is a model and conforms to a metamodel *MMt*. These metamodels conform to the metamodel *MMM* (such as Meta-Object Facility (MOF) defined by OMG or Ecore, within the Eclipse Modelling Framework (EMF)).

ATL is a mixture of declarative and imperative constructs. Note that there are two common approaches to programming: imperative programming and

declarative programming. Imperative programming provides a list of instructions, or algorithm, to be executed in a particular order. An example of the imperative approach is a Java program that counts the number of words in a sentence beginning with a capital letter. Declarative programming describes a set of conditions without giving its control flow, and lets the program figure out how to accomplish them. The SQL statement `SELECT * FROM Book WHERE Author = 'Tony'` is an example for the declarative approach "In other words, 'specifying how' describes imperative programming and 'specifying what is to be done, not how' describes declarative programming." (Jayaratchagan, 2004).

ATL transformations are unidirectional; they operate on read-only source models and produce a output target model. That is, during the execution of the transformations, the source model is navigated but is not allowed to change; the target model cannot be navigated. An ATL module is composed of a header, imports, helpers and transformation rules. The detail of each component is now described.

2.9.1 *Header*

A header names the transformation model and declares the source and target models. A scheme of a header section is shown below.

```
module module_name;
  create OUT : target_metamodel_name from
         IN : source_metamodel_name;
```

The header section starts with the keyword **`module`** followed by the name of the module (`module_name`). The keyword **`create`** defines the target model while the keyword **`from`** indicate the source model. The target and source models are bound to variables `OUT` and `IN` to indicate the target metamodel's name and the source metamodel's name respectively. Generally, more than one source and target models can be declared in the header section.

2.9.2 *Transformation rules*

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 transformation rules syntax definition is described below.

```
rule rule_name {
 from in_var : in_type [(condition)]?
 [using { var1 : var_type1= init_exp1;
 ...
 varn : var_typen= init_expn; } ]?
 to out_var1 : out_type1
 (binding1),
 ...
 out_varn : out_typen
 (bindingn)
 [do { statements } ]? }
```

Each rule is identified by a rule name (`rule_name`) which must be unique within an ATL transformation model. An ATL rule is composed of two mandatory parts (the **`from`** and the **`to`**) and two optional parts (the **`using`** and the **`do`**).

The **`from`** part is used to indicate the source model. It comprises a source variable declaration (`in_var`) and its type (`in_type`). The `in_type` is declared in a form of `metamodel_name!metamodel_element`. This is the way to identify with which elements the rule is involved. For example

```
from c : GeometricElement!Circle
```

where `c` is a source variable used in the rule, the `GeometricElement` is a source metamodel's name and the `Circle` is a source model element. The variable may contain an optional boolean expression (`condition`) to state a subset of the source model elements.

The **`using`** part defines a number of local variables which are used in the **`to`** and the **`do`** parts. An example of a **`using`** part is shown in Figure 2-31 (ATLAS Group, 2008); it defines a `pi` and an `area` values as variables to use in the rule.

```
from c : GeometricElement!Circle
using { pi : Real = 3.14;
area : Real = pi * c.radius.square(); }
to ...
```

Figure 2-31 An example of the **using** section

The **to** part contains a number of target pattern elements. It is a mandatory section and has to contain at least one target pattern element. Each target pattern element is declared by a name (`out_var`) and its type (`out_type`) in which each element is separated by a comma.

A target element is identified by a set of bindings (`binding`) which is used to define the way a source element is generated to be a target element. Each binding has to be identifid by the syntax definition below. The name of a target element (`target_element_name`) must be matched with the element's name defined in the target model.

```
target_element_name <- expression
```

The **do** part is optional and is used to specify some imperative codes that will be executed after the initialization of the target elements generated by the rules.

An example of defining the **to** and the **do** parts are illustrated by a rule *Machine*, Figure 2-32. This rule aims to create an UML-B machine (`umlbMetamodel!UMLBMachine`) and a context (`umlbMetamodel!UMLBContext`) from a source model element (`TDMetamodel!TDMachine`) where variable `t` is used to represent a source model element, while variables `m` and `ctx` represent target model elements.

```
rule Machine {
   from t : TDMetamodel!TDMachine
   to m : umlbMetamodel!UMLBMachine
      (name <- t.name,
      classes <- t.class),
      ...
      ctx : umlbMetamodel!UMLBContext
      (name <- t.name + '_ctx' )
   do { m.contexts <- m.contexts.append(ctx);
        ...} }
```

Figure 2-32 An example of the **do** section

For creating the UML-B machine, a machine's name and a machine's class are created by source elements `t.name` and `t.class` respectively. For creating the UML-B context, the context's name is created from a source element `t.name` appended by the string `_ctx`. The **do** section expresses the way to add the variable `ctx` into the UML-B machine by using the keyword `append`, where `contexts` is an association in the target model `umlbMetamodel!UMLBMachine` used to link contexts to a machine.

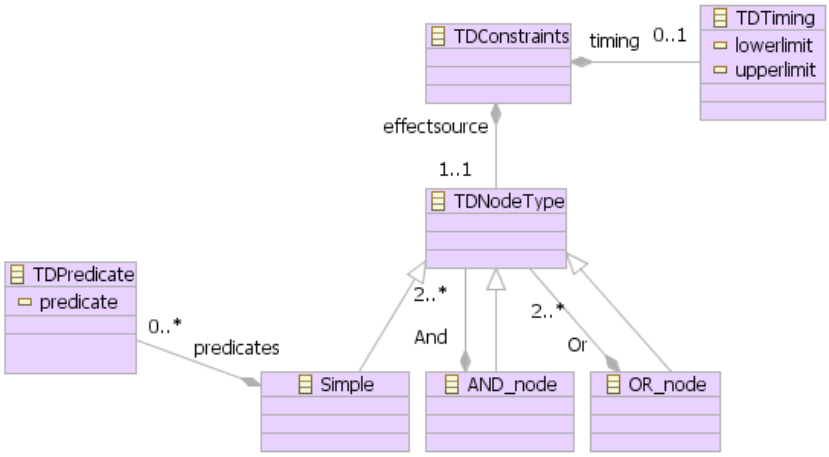


Figure 2-33 Example of TDMetamodel (parts of)

Another example is shown in Figure 2-34. This figure shows the rule `Constraint` which aims to generate a guard for a UML-B transition. This rule uses a source model element `TDMetamodel!TDConstraints`, as shown in Figure 2-33, to generate a target model element `umlbMetamodel!UMLBPredicate`. The rule calls a helper `getNodePredicate(t.timing)`, as detailed in Figure 2-35, to generate a predicate string and then assign to a target model element `predicate`.

```
rule Constraint{
 from t : TDMetamodel!TDConstraints
 to u : umlbMetamodel!UMLBPredicate (
 name <- 'TimingCnstrntGuard',
 predicate <- t.effectsource.getNodePredicate(t.timing))
 }
```

Figure 2-34 Example of a rule: `Constraint`

2.9.3 *Helpers*

A helper is a technique to define ATL translation rules with specific behaviours. An ATL helper makes it possible to define ATL code that can be called from different points of an ATL transformation. Helpers can be defined only on source models, since target models are not allowed to navigate. An ATL helper is defined by the following:

- an optional context type : defines kind of element the helper applies to
- a helper name : each helper must have a name defined as a string
- an optional set of parameters; a parameter is identified by `parameter_name : parameter_type`
- a return value type : each helper must have a return value
- an ATL expression that represents the ATL helper's code

There are two kinds of helpers: *Operation helpers* and *Attribute helpers* as follows.

***Operation helpers*:** an operation helper can have input parameters, and a result of the *Operation helpers* is created each time the helper is called. *Operation helper* syntax is defined below.

```
helper [context context_type]? def : helper_name (parameter_name :
parameter_type) : return_type = expression;
```

An example of an *Operation helper* is illustrated in Figure 2-35. This helper is named `getNodePredicate` and aims to generate a guard – a return value – which is a string for an UML-B transition. The helper uses an input parameter `t` whose type is defined by a source model element, `TDMetamodel!TDTiming`, as shown in Figure 2-33.

```
helper context TDMetamodel!TDNodeType
def : getNodePredicate(t:TDMetamodel!TDTiming) : String =
      if self.oclIsKindOf(TDMetamodel!Simple)
      then self.SimpleGuard(t) -> concat(self.SimpleCond())
      end if self.oclIsKindOf(TDMetamodel!AND_node)
      then ... ;
```

Figure 2-35 Example of an Operation helper: `getNodePredicate`

This helper checks whether the *node type* is `Simple`, `AND_node`, or `OR_node`. In order to do that, a condition **`if`** `self.oclIsKindOf(t:TDMetamodel!TDNodeType)` is used. The `self` is a keyword and used to define a context of an instance of a specific type. Thus, in this helper, `self` is used to indicate an instance of `TDNodeType`. The keyword `oclIsKindOf()` is an operation that returns a Boolean value stating whether `self` is either an instance of what defined inside the parentheses "(…)" or of one of its subtypes (ATL, 2008). This helper returns a string which is generated from concatenation (`concat`) of strings created from the other two helpers: `SimpleGuard(t)` and `SimpleCond()`. The helper `SimpleGuard(t)` is also an *Operation helper* and uses `t` as an input parameter, while the helper `SimpleCond()` is an *Attribute helper*, the detail of which is explained in the following paragraph. Note that the `TDMetamodel` and the helper `getNodePredicate` described in this section are different from that explained in Chapter 6.

***Attribute helpers*:** an attribute helper is used to associate read-only named values to source model elements. An *Attribute helper* cannot have input parameters and its return value is calculated only once when the value is required for the first time. *Attribute helper* syntax is defined below.

```
helper [context context_type]? def : helper_name :
                    return_type = expression;
```

An example of an *Attribute helper* is illustrated in Figure 2-36. This helper is named `SimpleCond()` and is called from the helper `getNodePredicate` as shown in Figure 2-35.

```
helper context TDMetamodel!TDNodeType
def : SimpleCond() : String =
self.predicates -> iterate(e; ret : String = '' |
      if not e.oclIsUndefined() then
            ret -> concat(' & ' + e.predicate)
      else
            ret -> concat('')
      endif
);
```

Figure 2-36 Example of an Attribute helper: `SimpleCond()`

The helper aims to generate a string value as a part of a condition for an UML-B transition, if there is any. From Figure 2-33, a string is created by an iterative process to concatenate predicate values (`predicate`) defined in a `TDPredicate`. In order to do that, we have to do iteration with an association `predicates` attached to the `Simple` node type, which is represent by `self.predicates` in ATL. The keyword `self` represents a source element `Simple` since this helper is called by the helper `getNodePredicate` whose `Simple` is inherited. The recursion is defined by the keyword `iterate`. The iterative syntax is defined below.

```
source -> iterate(iterator; variable_declaration = init_exp | body)
```

This iterative expression comprises four parts: `iterator`, an accumulator variable declaration (`variable_declaration`), a variable initial value, and a `body`. The `iterator` is used to refer an instance of a source collection. In the case of `SimpleCond()` helper, `e` is defined as an `iterator` representing a *predicate* value. The accumulator variable declaration is used to define an accumulator variable and its initial values (`init_exp`) are used inside the `body`, which is `ret` in this case. The body expresses the use of the `iterator` and variable. The `iterate()` operation returns a value in the accumulator variable once the last iteration has been performed.

From Figure 2-36, the body of the `SimpleCond()` helper checks whether the *predicate* value is empty by the keyword `oclIsUndefined()`. The `oclIsUndefined()` returns a boolean value true if *predicate* is undefined. If there

are *predicate* values, each of them is concatenated with each other with a symbol "&", if not the helper returns an empty string.

2.10 Summary

This chapter provides background knowledge used in this thesis. It starts from broad RE techniques and then focuses on using FMs. Event-B modelling is introduced; the detail of performing refinement and POs is given. The way RODIN toolkits are used to create and verify a model is presented. We explain features of an UML-B tool that is a graphical front-end for Event-B and used to create a model from TD. KAOS framework descriptions are discussed as goal-oriented modelling. The knowledge of metamodelling and Eclipse EMF is explained since they are used to generate ATL translation rules for mapping a TD model to a UML-B model.

Chapter 3 Other Relevant Work

This chapter aims at giving background of the knowledge other related work used in this research. These works are relevant to our research since one of them is used as a part of our translation patterns. Some provide tools that may useful for future work. Some show how their work is trying to expand KAOS, TD and Event-B in other ways. This Chapter begins with describing SysML background; section 3.2 explains an Action/Reaction pattern; section 3.3 gives an explanation of relevant research in KAOS and B. The next section describes research in KAOS and UML, while section 3.5 explains work on CSP and B; section 3.6 gives an explanation of other related research concerning TD, while section 3.7 describes LTL properties, which are useful for RE.

3.1 SysML

UML has been used broadly but it does not have a digram to identify some special needs such as modelling requirements and defining functions. Thus, Systems Modelling Language (SysML, 2008), which is as an extension of UML 2.0, was developed. SysML is a “general-purpose graphical modelling language for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities” (OMG, 2008;SysML,2008).

SysML 1.1 (SysML, 2008) reuses a subset of UML 2.0 and defines additional extensions by using UML's profile mechanism (Hause*, et al.*, 2005; Vanderperren and Dehaene, 2005). Figure 3-1 illustrates the reuse and extension of UML 2.0 by SysML.

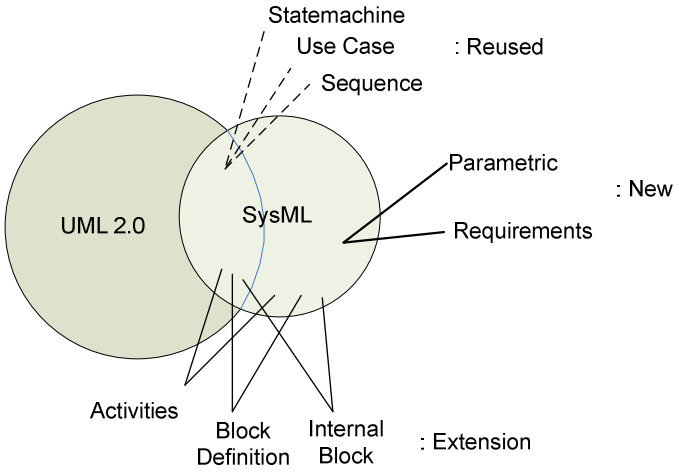


Figure 3-1 UML 2.0 and SysML 1.0

UML 2.0 Statemachine, Use Case, and Sequence diagrams are reused while some existing UML diagrams are extended as follows:

- Block Definition diagram: the Block Definition diagram is based on the UML Class diagram. It uses blocks, which are modular units of system description, to describe the structure of a system or element of interest in broad view.
- Internal Block diagram: the Internal Block diagram is based on the UML Composite Structure diagram. It is used to show how the defined blocks are used in detail.
- Activity diagram: the Activity diagram is based on the UML activity diagram. It is used to show the control flow, flow of inputs and outputs between actions.

SysML introduces two new diagrams, the Parametrics diagram and the Requirements diagram. The Parametrics diagram is used to show relations between parameters. The Requirements diagram provides a modelling construction for text-based requirements, and the relationship between requirements and other model

elements that satisfy or verify them in a graphical manner. An example of a Requirement diagram for a simple lift system is shown in Figure 3-2.

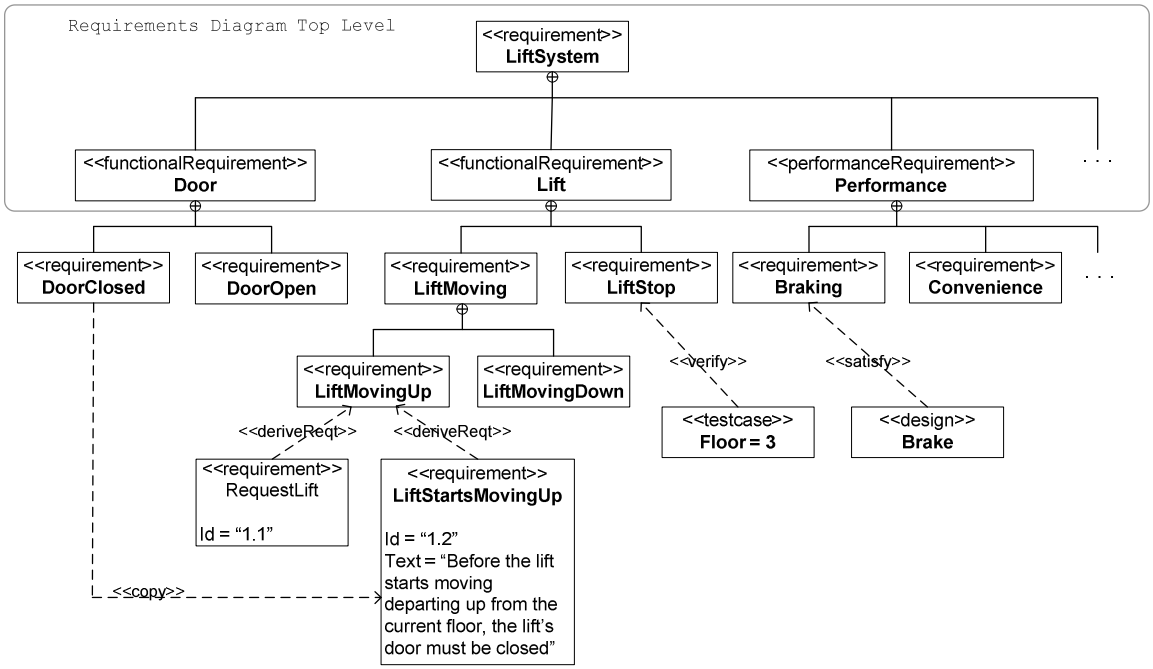


Figure 3-2 An example of Requirements diagram for a lift system

A *Requirements diagram* uses <<requirement>> stereotype to identify the requirements in which, for example, there can be subcategories of <<functionalRequirement>> and <<PerformanceRequirement>>. A <<functionalRequirement>> is used for specifying an operation that a system must perform while a <<PerformanceRequirement>> is used for identifying satisfaction constraints of the system. Relationships between requirements are shown by using stereotypes such as <<deriveReqt>>, <<satisfy>>, <<verify>> and <<copy>>. The <<deriveReqt>> describes the derivation of multiple requirements that support a source requirement while <<satisfy>> describes the satisfaction of requirements by designing and implementation (Moore, 2006; SysML Partners, 2006). The <<verify>> is used to specify the relationship between a requirement and a test case. The <<copy>> is for reusing requirements; that is, the slave requirements property is a read-only copy of the master requirements property. For example in Figure 3-2, a part of a slave requirement **LiftStartsMovingUp**'s text property is copy from text property of a master **DoorClosed** requirement.

The *Requirements diagram* has the idea of breaking a compound requirement into multiple subrequirements as shown in the figure above. That is, a top-level Requirement diagram illustrates whole requirements in general while the bottom-level shows detailed requirements and relationships between them. The SysML decomposition concept of requirement is similar to KAOS goal refinement.

3.2 Action/Reaction Pattern and B

An *Action/Reaction pattern* was introduced by Abrial to describe causes and effects in reactive systems (Abrial, 2005b; Abrial, 2008b; Abrial and Hoang, 2008). The actions are the causes which make the effects take place. As shown in Figure 3-3, the continuous line, dashed line and curved arrow represent action, reaction and cause/effect between action and reaction respectively. The *Action/Reaction pattern* is used to model a B machine while refinements are gradually created corresponding to additional information in the *Action/Reaction* models (Abrial and Hoang, 2008).

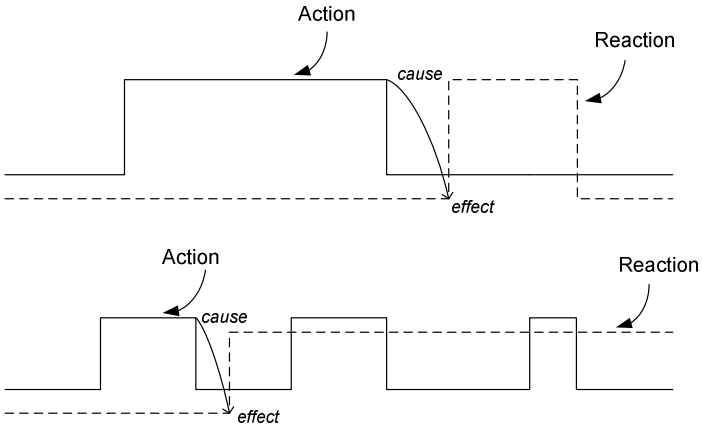


Figure 3-3 Examples of action and reaction pattern

Figure 3-4 illustrates an example of *Action/Reaction patterns* for the lift system corresponding to <<requirement>> **DoorClosed** and <<requirement>> **liftStartsMovingUp** in Figure 3-2. Note that this is only a straightforward example; extra information is added to this Event-B for completeness later.

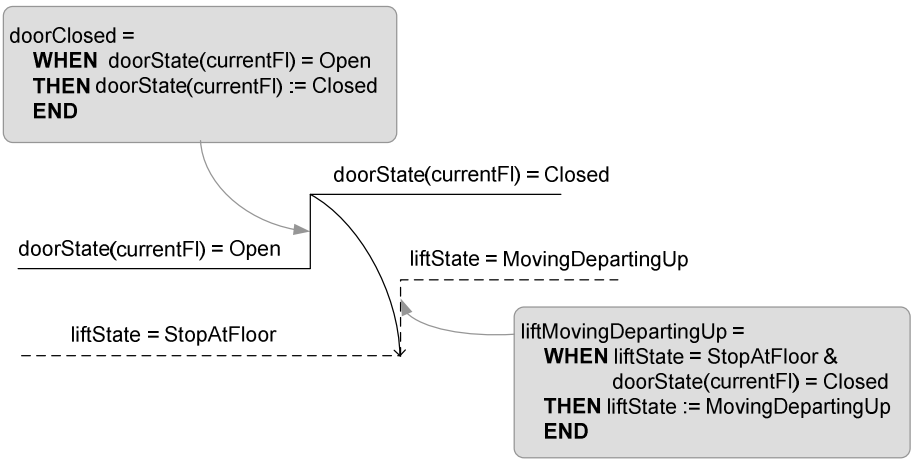


Figure 3-4 Action/Reaction patterns and corresponding B machines

The *Action/Reaction pattern* describes changing of states, which is similar to changing states in TD. Translating TD to Event-B in this work uses this pattern as a part of translation patterns and has some extra structures and information.

3.3 KAOS and B

One approach for diminishing the gap between KAOS requirement and formal method specification is introduced by (Ponsard and Dieul, 2006). The idea is to generate a B machine from a KAOS model and to create the connection between FAUST toolbox (FAUST, 2008) and RODIN platform. The FAUST toolset aims at achieving formal assurance, verification and validation (V & V), for the KAOS model at an early stage (Ponsard, *et al.*, 2007). The FAUST toolbox composes tools such as `Refinement checker`, `Compiler` and `Animator`. The `Refinement checker` can automatically verify and validate goals, and operations on a given domain. The `Compiler` is used to generate a finite state machine from a KAOS *Operation model* and represents it in a graphical domain-based visualisation using `Animator`.

Matoussi has been investigating a technique how to create Event-B models incrementally from KAOS goal models (Matoussi, *et al.*, 2008). Currently, the technique can generate Event-B models from two KAOS refinement patterns: milestone-driving tactic and case-driven refinement tactic, in which the latter needs

to have additional constraints to complete an Event-B model. Those two KAOS patterns are the same as we found in mapping TD to KAOS.

3.4 KAOS and UML

Heaven and Finkelstein introduced a technique to create a profile to allow the KAOS model to be represented in UML (Heaven and Finkelstein, 2004). The UML is extended by introducing new stereotypes and tags which allow one to model the KAOS in UML. Since UML editors do not support temporal logic notation, the formal definitions in KAOS have to be rewritten in ASCII. The following is an example of how to represent a KAOS goal model by the UML stereotype. (Example below has been taken from (Heaven and Finkelstein, 2004))

Goal **Achieve**[AmbulanceIntervention]
InformalDef For every urgent call reporting and incident, there should be an ambulance at the scene of the incident within 14 mins
FormalDef $\forall$ c: UrgentCall, inc : Incident (@ Reporting (c, inc)) $\Rightarrow$
$\Diamond_{\leq 14\ min}$ $\exists$ amb: Ambulance (Intervention (amb,inc)))

UML which represents the same goal is:

<<goal>>
AmbulanceIntervention

{**form** = Achieve
informalDef = For every urgent call reporting and incident, there should be an ambulance at the scene of the incident within 14 mins
formalDef = forall c: UrgentCall, inc : Incident (just Reporting(c,inc) -->
eventually [<= 14 min] exists amb: Ambulance (Intervention(amb, inc)))}

Though this technique explains how to combine KAOS with UML, there is no clear use for this contribution of KAOS in UML. The users have to learn and understand how to use KAOS-UML apart from only modelling. The benefit is unclear. This approach merely describes how to model the KAOS by using UML notation.

3.5 CSP and B

A B machine is good for modelling a reactive system, since the operations thereby enabled can run in parallel. Thus, parallel activities are easily modelled in B. However, B machines "can be less convenient at modelling sequential activity" (Butler*, et al.*, 2005a). It needs to have a program counter to order the actions' execution. In contrast, Communication Sequential Process (CSP) – a process algebra defined by (Hoare, 1985) – provides operators such as sequential composition, choice and parallel composition of processes, as well as synchronous communication between parallel processes (Butler*, et al.*, 2005b). CSP was designed for describing systems of interacting components, where each component is called a process. The process communicates with others and its environment using an alphabet of events. "An event describes a particular kind of atomic indivisible action that can be performed or suffered by the process" (Schneider, 2000).

Butler introduces csp2B, which allows specifications to be written in a combination of CSP and B (Butler, 2000). Then, the CSP can be compiled to a pure B representation which can be analyzed by a standard B tool such as ProB. (Butler*, et al.*, 2005a) proposes a technique to represent an extension of ProB which supports checking of specifications written in a combination of CSP and B. The technique is to define events in the CSP specification to have the same name as B operations. The combination of CSP and B enables ProB to do automated consistency checking and refinement checking of specifications written in a combination of CSP and B.

A case below provides an example of how to identify a lift is moving up specification in CSP. The lift is moving up specification is composed of 4 states: `StopAtFloor`, `MovingDepartingUp`, `MovingUp` and `MovingArrivingUp`. After the lift is in a state of `MovingDepartingUp`, the corresponding `floorsensor` at that floor is set to `Off` and then the `lift` changes to the state `MovingUp`. Whenever the `lift` is in a state of `MovingArrivingUp`, the `floorsensor` for the upper floor is set to `On` and then the lift can be in a state of `MovingDepartingUp` or `StopAtFloor`. The symbols $\rightarrow$, ?, $\Box$ and ; are used for prefix operator, input, deterministic choice and sequential composition respectively.

```
LIFT(f) = (StopAtFloor → MVDU(f)) □ (StopAtFloor → MVDD(f))

MVDU(f) = MovingDepartingUp → FloorsensorOff?(f) → MovingUp
            → MVAU(f)

MVAU(f) = (MovingArrivingUp → FloorsensorOn?(f+1) → MVDU(f+1))
           □
          (MovingArrivingUp → FloorsensorOn?(f+1) → StopAtFloor)
MVDD(f) = …
```

3.6 Other concepts

PLS/Sugar 2.0 (IBM, 2008) is a formal specification language used to describe hardware's behaviour over time. PSL/Sugar 2.0 uses *Sugar Extended Regular Expressions* (SERE) to describe a set of state sequences (Fisman and Eisner, 2009) in which the sequence can be represented by a TD. An example of SERE is `{req;busy[*4];gnt}` which can be illustrated in TD as shown in Figure 3-5 (taken from (Fisman and Eisner, 2009)).

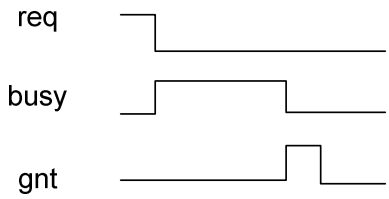


Figure 3-5 Timing diagram representing `{req;busy[*4];gnt}`

Figure 3-5 shows that, first, the `req` is set true for 1 unit of time. Then, whenever `req` is false, the busy is held true for 4 units of time. Finally, `gnt` is set true after the busy is set false. PLS/Sugar 2.0 provides another way that is easy for the user to understand and to read a sequence of system behaviour. However, PLS/Sugar 2.0 does not identify notations that are used for sending message between objects as in UML TD. The PLS/Sugar 2.0 diagram is used to describe the sequence of events and does not describe causality.

Fisler proposes an event-sequence language for capturing TD's transitions into an event of a sequence and a temporal constraint (Fisler, 2006). An example of

TD is illustrated in figure 3-6. A transition in TD is indicated by a state value, such as a, followed by an arrow direction such as a↑ and a↓ to denote falling and rising transitions of a respectively. An event e is a conjunction of transitions.

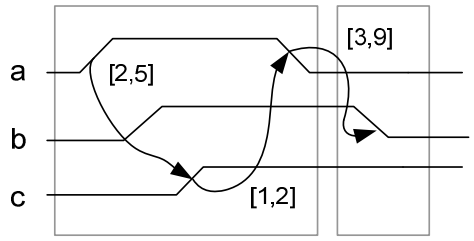


Figure 3-6 An example of a Timing diagram

In the figure above, two outlined areas indicate regions of the TD that occur in sequence. A cluster `C` is used to specify shade regions in the TD. Timing constraints `T` are specified by a set of tuples `<e, l, u, Boolean value>`, where `e` are events covered, `l` and `u` are lower and upper bound timing constraints, and the `Boolean value` is whether the timer is enabled. Below, we show the event-sequence language which corresponds with the TD above:

```
C = {{a↑, b↑, c↑, a↓}; b↓}
T = {<a↑, c↑, 2, 5, true>
    <c↑, a↓, 1, 2, true>
    <a↓, b↓, 3, 9, true>}
```

This technique is easily understood and offers notations that are readable for users.

Barland describes the meaning of temporal logic notations in a timeline (Barland, *et al.*, 2006). An example of a timeline which represents $\Box(q \rightarrow \Box\neg p)$ is illustrated in Figure 3-7.

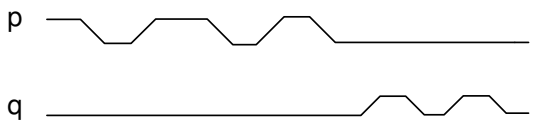


Figure 3-7 Timeline
after (Barland, *et al.*, 2006)

Even though transferring from LTL notations to TD is easy to understand, the researchers do not propose a technique to express timing constraints. Moreover, translating from a LTL formula to TD is implicit. That is because one LTL formula can be translated to one or more TD. As shown in Figure 3-8, the notation $p \rightarrow \Diamond q$ can be illustrated by more than one TD.

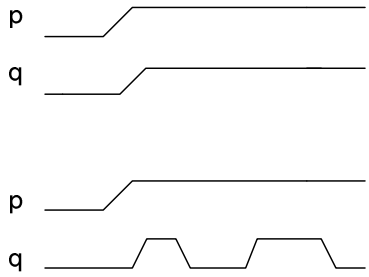


Figure 3-8 Timing diagram for $p \rightarrow \Diamond q$ notation

3.7 LTL properties and Requirements Engineering

This topic focuses on some LTL properties, i.e. traceability, safety, liveness (progress) and fairness. These properties are importance and used for maintaining the correctness of doing RE.

Traceability: in the RE context, traceability is understanding how high-level requirements – objective, goals, aims, expectations, and needs – are transformed into low-level requirements (Hull*, et al.*, 2004). SysML Requirements diagram (SysML Partners, 2006), which was described in section 3.1, provides requirements traceability.

Safety: a safety property one that guarantees something bad never happens. A temporal logic formula for the safety property can be written as $\Box\neg unsafe$ where unsafe is a propositional formula. A system has the safety property whenever all states of the system can be reached. The safety property can be declared as the `Avoid` goal pattern in KAOS model and is the main objective of using FM in RE processes. In Event-B, the safety properties are identified as invariants. For example, "the lift door must be closed all the time while the lift is moving" is a safety property and is defined as an Event-B invariant.

Liveness progress: a liveness progress property asserts that something good eventually happens. A temporal logic formula for the liveness property can be written as ◊G for some propositional formula G. In the same way, we can say that it guarantees an action will eventually be executed (Friedental and Steiner, 2004). The progress property is the opposite of starvation (deadlock) and can be declared as the `Achieve` goal pattern in KAOS model.

Fairness: a fairness property indicates that, from time to time, a system must pass through a state which satisfies some properties. A temporal logic formula for the fairness property can be written as □ ◊ G which means G holds definitely often. In this thesis, we do not model the fairness properties. However in the lift case study, one can identify a fairness property as a performance requirement. For example, a lift must be shut down for its annual check.

An example of a tool which can check the states' correctness of a model is *ProB* (Leuschel and Butler, 2005). *ProB* is a graphical animator and model checker for B method. It provides a feature to verify the safety and progress property of the system states. The model checker in *ProB* does this by automatically detecting invariant violations and deadlocks in traceable state spaces. Apart from ProB, there is a model checker which can verify program requirements such as deadlock freedom and livelock freedom which is called "*Timeline Editor*" (Smith*, et al.*, 2001). The *Timeline Editor* is used to verify requirements which are implemented in the form of events along a timeline. The timeline looks similar to UML TD. However, it is represented in new notations and extra definitions such as events and lines. To obtain the requirement to be checked, "the timeline specification is mechanically converted into an equivalent test automaton for using in a logic model checking process such as Spin" (Hozmann, 1997). The tool has an interface that is easy for users and can fully verify requirement properties such as deadlock freedom and liveness issues. However, the notations used in the timing-like diagram for the identification of events along a timeline needs training, because they are different from UML 2.0 TD.

3.8 Summary

The literature review in this chapter describes work related to this thesis. SysML introduces some new diagrams to those defined in UML; one of them is a Requirement diagram. The Requirement diagram represents system requirements in a graphical way. The diagram has a concept of requirements decomposition, which is similar to KAOS goal refinement. This is beneficial to software developers for presenting sub-requirements and tracing them back to corresponding documentation, test cases and design modules. Action/Reaction patterns provide a method of creating an Event-B model from causal dependency relationships between objects. This pattern is used as part of our techniques to generate Event-B and KAOS models from a TD. Some relevant work that concerns the combination of KAOS, B, CSP and UML, is described in this chapter. Some work has been trying to generate formal languages from TD, such as PLS/Sugar and the event-sequence language. LTL properties such as traceability, safety, liveness and fairness, that are important for requirements engineering, are explained. Those properties should be concerned whenever modelling explicit system requirements.

Chapter 4 Timing Diagrams and Lift Specification

Recently, TD has been added to the UML 2.0 specification, but it has been used in electronic engineering for a long time (Fowler and Scott, 2004). The TD is a particular type of interaction diagram and is used for exploration and monitoring of the behaviour of objects over any given period of time. However, using TD is suitable for some kinds of specification behaviours. We clarify what kinds of system specification are appropriately and inappropriately described by the TD.

- Appropriate requirements are those that can be declared as changing states of hardware with time, or there are causal dependencies between the system's objects, or both; for example, embedded software components for a microwave controller, vendor machine controllers, and ATM transaction processing.

- Inappropriate requirements are those concerns with human actions such as modelling a person pressing a button, business requirements such as budget controlling, and improving response time to customer inquiries.

The rest of this chapter starts by presenting lift specifications that are used in this work. Section 4.2 explains UML 2.0 TD (OMG, 2008). Section 4.3 provides the amended TD notations that are obtained by selecting UML 2.0 TD's notations and adding some new notations to make it suitable for translation; section 4.4 illustrates TD for the lift specifications. Section 4.5 provides a brief glossary for TD; section 4.6 gives an example of a preliminary TD editor.

4.1 Lift Specification

The original lift position display specification is taken from (Jackson, 2001) where it is described as the following:

> "A somewhat primitive lift in a small hotel has been installed and successfully operated for many years. Now it is to be fitted with an information panel in the lobby, to show waiting guests where the lift is at any time, so that they will know how long they can expect to wait until it arrives.
>
> The panel has two lamps for each floor. There is a floor lamp (square lamp) to show that the lift is at the floor, and a round lamp to show that there is a request outstanding for the lift to visit the floor. In addition, there are two arrow-shaped lamps to indicate the direction of travel. There is a lobby, and there are eight other floors, so the panel looks like this."

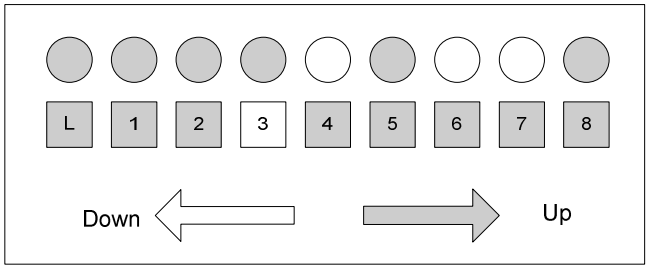


Figure 4-1 Lift Position Display

> "The job is to drive the panel display from a very minimal interface with the existing request buttons and floor sensors of the lift. A floor sensor is on when the lift is within 6 inches of the rest position of the floor. Pressing a button is detected as a pulse. There is one button at each floor to summon the lift, and a set of buttons inside the lift car – one button to direct the lift to each floor."

The lift specification used in this thesis

The specification above shows causal dependencies between system objects that can be specified in TD. To make it more suitable for modelling with TD, Event-B and KAOS, the specifications are expanded to include timing constraints and hardware, i.e. *Door*. In addition, it is assumed that there is one floor sensor for each floor.

The amended lift system specifications are described in two parts: the lift moves from the current floor to service a request at floor f, and the lift general servicing.

1. The lift moves from the current floor to service a request at floor f
   - 1.1 The request lamp for floor f must be lit.
   - 1.2 Before the lift starts moving departing up/down from the current floor, the lift's door must be closed.
   - 1.3 If the lift door is open at the current floor and there is a request to service some floor f, then the lift door at the current floor must be closed. Next, within between 1-5 seconds after the door closed, the lift starts moving departing up or moving departing down.
   - 1.4 The current floor sensor must be off within 2-5 seconds after lift starts moving departing up or moving departing down.
   - 1.5 The floor lamp for floor f will be unlit within 2-4 seconds after the current floor sensor is set off.
   - 1.6 Whenever the floor sensor status is off, it means the lift is moving (possibly moving up or moving down, cannot be both).
   - 1.7 The floor sensor for floor f must be on within between 2-5 seconds when the lift is moving nearly arriving up/down at the rest position of the floor f.
   - 1.8 The lift will be stopped at floor f within between 1-5 seconds after floor sensor at floor f is set on.
   - 1.9 The floor lamp for floor f will be lit within 2-4 seconds after the current floor sensor is set on.

1.10 Whenever the lift stops at the requested floor f, the lift door will be opened within between 1-5 seconds.

1.11 Request lamp for floor f will be unlit within 2-4 seconds after the lift stops at floor f.

2. Lift general servicing

2.1 While the lift starts moving departing up, the up lamp must be activated and the down lamp must be deactivated.

2.2 While the lift starts moving departing down, the up lamp must be deactivated and the down lamp must be activated.

2.3 If the lift is stationary, both up and down lamps must be deactivated.

2.4 If there is no request, the lift will stop at the last floor serviced.

The simple example below indicates the kind of requirements we believe can be specified in TD. This example shows how a *floorlamp* and a *floorsensor* objects – requirements 1.5 and 1.9 – are associated in TD.

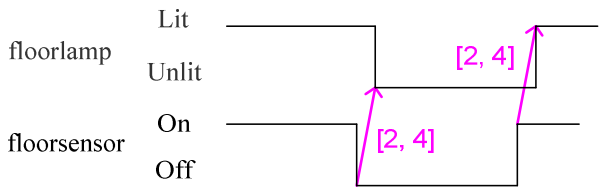


Figure 4-2 A simple TD shows relationship between *floorlamp* and *floorsensor*

4.2 UML 2.0 Timing Diagram

There are two forms of TD: a compact notation and a robust/full notation. The details of these notations are described below.

The compact TD uses a *Lifeline* to represent individual object in the diagram. An object is identified on the left-hand side while its states are listed along the right-hand side. A state is denoted by text and a state change represented

by a crossing (OMG, 2007; Visual Paradigm, 2007). A *DurationConstraint* is used to specify the period of time for each state. The compact TD is suitable for exploring the general behaviour of one or more objects during a period of time, while the robust TD is used whenever one would like to identify more detailed information. An example of the compact TD is illustrated in Figure 4-3.

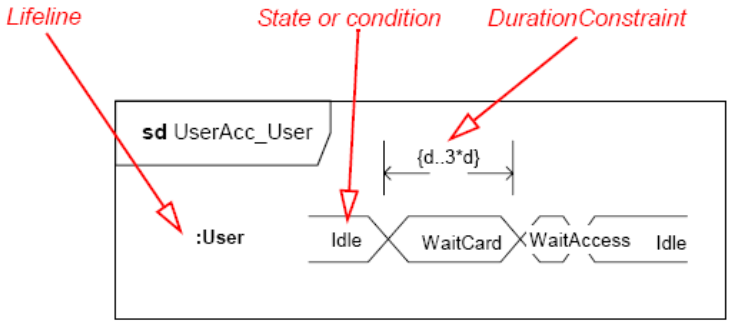


Figure 4-3 Compact Timing diagram (OMG, 2007)

The robust TD shows the states of each object on the left-hand side of the diagram (Y-axis) while timing constraints are on the X-axis. A timeline is used to display the change in state or value of one or more elements over time (Sparx Systems, 2006).

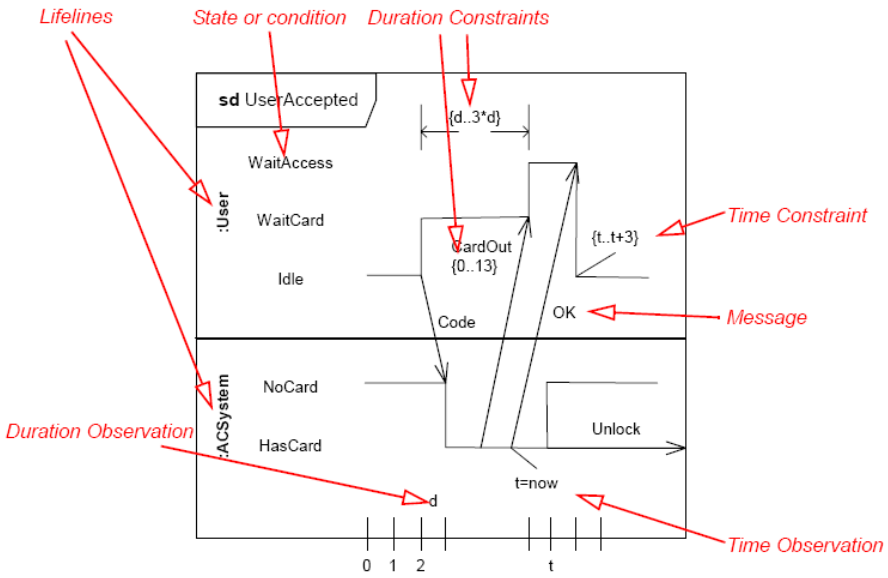


Figure 4-4 Robust Timing diagram (Ambler, 2004)

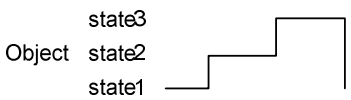
Event/Stimuli are optionally labelled at transition points to indicate the reason for the change (Ambler, 2004). An example of a robust TD is illustrated in Figure 4-4.

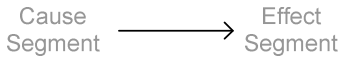
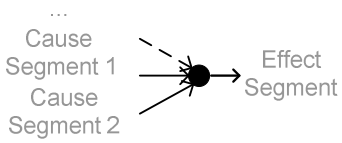
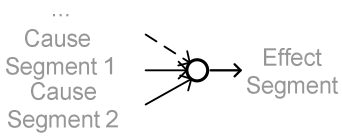
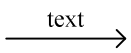
According to Figure 4-4, `Code` and `OK` are messages sent between objects. `Cardout` is an event which makes an object `user` change its state from `Waitcard` to `WaitAccess`. *Time Constraint* indicates when an event must occur, while *Duration Constraints* indicate how long a state or value must be in effect; where `d` and `t` represent a unit of duration and time respectively. A *Time Observation* indicates the point of time a *Lifeline*'s state is observed.

4.3 UML Timing Diagram Amended

Though UML 2.0 TD uses simple notations to explain the changing of object's states through time, it is composed of many notations specifying properties that are not dealt with in this work. Thus, a subset of notations is selected and some notations are justified, which are easier for generating expressions to interface with Event-B and KAOS. The TD notations used in this research are based on the (OMG, 2007) Robust TD notations. The notations for graphic nodes and paths to be included in the TD are described in

Table 4-1.

Node Type and Notation	Reference
Object and State 	A state notation on the horizontal axis indicates the state of an object.
Timeline 	A Timeline is used to illustrate an object changing states, where an object can have a Timeline. A Timeline is composed of a chain of segments in which segments represents an object's state and the position it appears on the Timeline. A segment is connected with another by a Transition. Time is

Node Type and Notation	Reference
	indicated on the horizontal axis.
CauseEffectArrow 	An arrowed line indicates a cause and an effect between objects' segments. The beginning of line represents a cause segment while the end of the line (with arrow) represents an effect segment. A simple form of a CauseEffectArrow is to link a cause segment to an effect segment.
AND  OR 	"AND" and "OR" notations are used for specifying combinations of cause segments within a CauseEffectArrow. Currently, they are not used to contribute one cause to many effect segments. Using "AND" notation means the causes that make an object changing its state are derived from a combination of those cause segments, while "OR" indicates or-inclusive relationship. Each "AND/OR" notation comprises the minimum of two cause segments (as represented by bold-lines, while dashed-lines represent other specified segments if there are any). Nested "AND" and "OR" relationships for a CauseEffectArrow are allowed.
Condition 	Conditions are optional additional constraints that cause a state change. A condition is represented by plain text presented above the CauseEffectArrow.
Duration constraint [t1, t2]	Duration indicates time constraints and is used to describe how long a state or value must be in effect. Time unit in the duration constraint can be second or minute. The duration constraints can be identified by using symbols, i.e. [t1, t2] indicates the time constraint starts from t1 and ends at t2.

Node Type and Notation	Reference
SimultaneityArrow 	Simultaneity is represented by an arc dashed-line and is used to synchronize objects that change their states simultaneously. When the application is eventually developed, one does not expect things to be exactly simultaneous. It means one expects them to happen very close to each other and no particular constraint; that is two things are very close in time. It is used in terms of "the level of abstraction."

Table 4-1 Timing diagram notations

To be practical, we define a `CauseEffectArrow` to be drawn from the start point of a cause segment to the start point of an effect segment as shown in Figure 4-5 (A). However, if an object has no state change, it can be drawn as shown in Figure 4-5 (B).

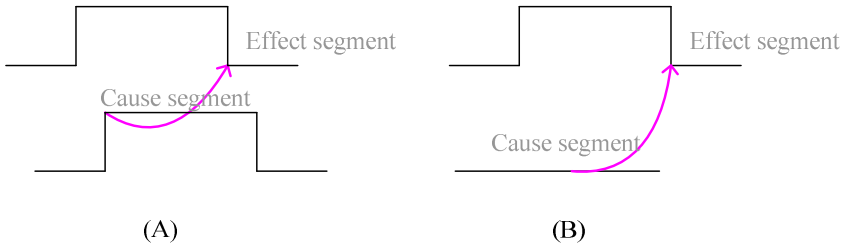


Figure 4-5 Robust Timing

The amended TD is generally designed to fit with other systems that concern timing constraints, changing an object's state through time and within an object itself. It provides sufficient notation to identify discrete timing constraints in the system specification. Here, we clarify some points of similarity and difference between amended TD and UML 2.0 TD.

Similarity:

1. Timelines
2. States
3. Duration constraints
4. Conditions can be seen as messages in standard UML TD notations

Difference:

- An arrowed line is used to indicate cause and effect between objects' states rather than sending messages between the objects as in standard UML TD.
- `SimultaneityArrows` are a new notation
- `AND` and `OR` nodes are new notations

So far, we have not found any cases in the lift system that need to be modelled by *Time Observation* (defined in standard UML TD). Thus, we do not deal with this symbol at this time.

4.4 Timing Diagram for the Lift specifications

To provide a simple example, we select requirements 1.4 and 2.1 that are concerned with four objects: *lift*, *floor sensor*, *up lamp* and *down lamp*. TD which is created from these objects represents specification number 1.4 (lines a and b) and 2.1 (lines c and d) is shown in Figure 4-6. Note that the symbols a and b are not TD notations but used only for explanation in this section.

Figure 4-6 shows that the lift comprises seven states: *MovingArrivingUp*, *MovingUp*, *MovingDepartingUp*, *StopAtFloor*, *MovingDepartingDown*, *MovingDown* and *MovingArrivingDown*. A *floor sensor* has two states: *On* and *Off*. *Uplamp* has two states: *deActivated* and *acTivated*, while *downlamp* has two states: *Deactivated* and *Activated*. We have to use different names for *Uplamp* and *Downlamp* states since Event-B and UML-B models do not allow duplicate names.

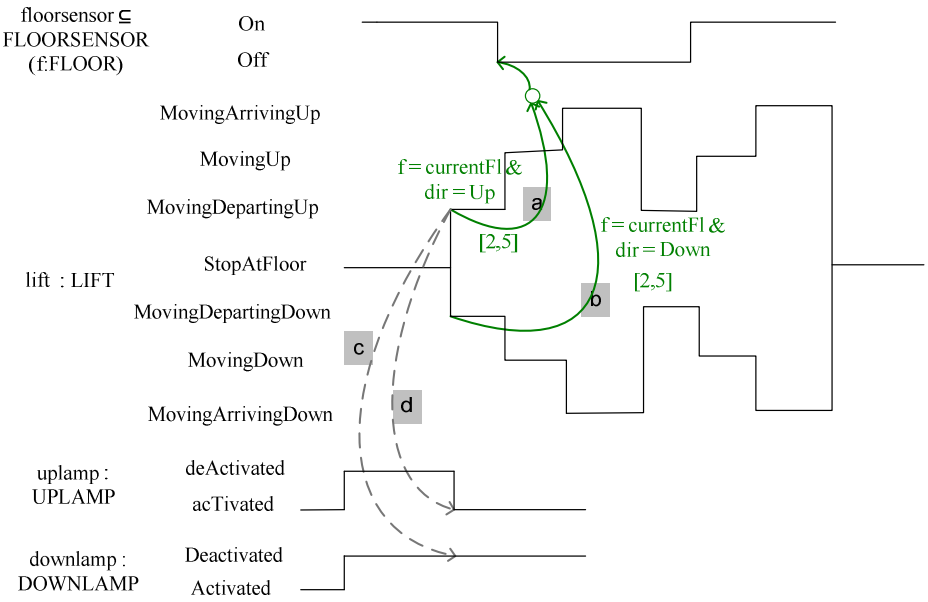


Figure 4-6 Timing diagram from Floorsensor, Lift, Uplamp and Downlamp

In terms of RE, we can describe the relation between lift movement and the floor sensors as: whenever a user presses a button to request a lift, the lift starts moving departing up (a)/ 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, requirement 1.4. 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), requirement 2.1.

In term of TD notations, we say that there are four `Timelines` which represent the state changes in time for the corresponding objects: *floorsensor*, *lift*, *uplamp*, and *downlamp*, belonging to classes *FLOORSENSOR*, *LIFT*, *UPLAMP* and *DOWNLAMP* respectively. The lines a and b show the combination of the `CauseEffectArrow` by using "*OR*" notation; it means the *floorsensor* is set to *Off* according to whether the *lift* is in the state of *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 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 determine the occurrences happen very close to each other with no particular constraint. The whole TD for the lift specification is illustrated in Figure 4-7.

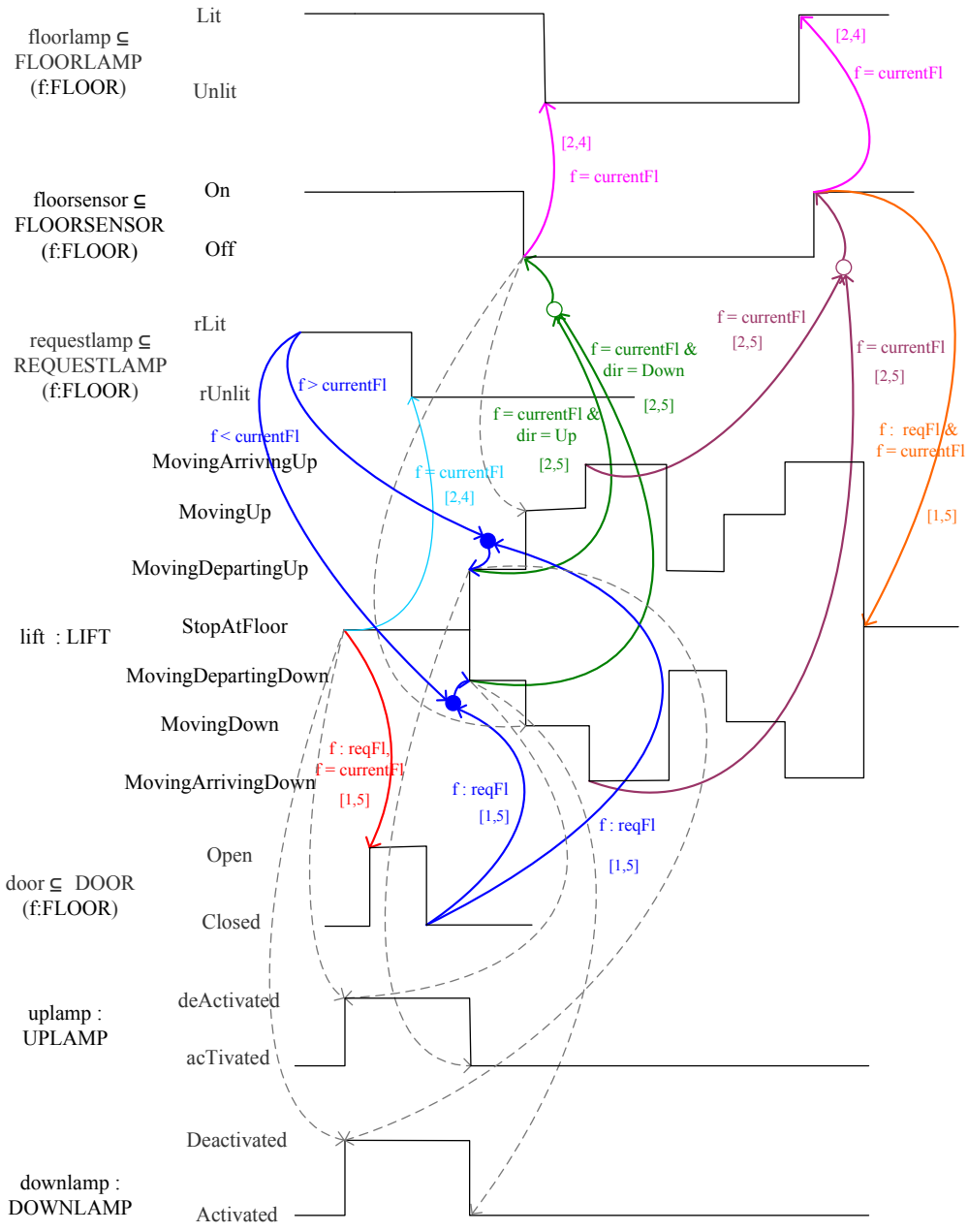


Figure 4-7 Timing diagram for the lift specification

4.5 A brief glossary for Timing Diagrams

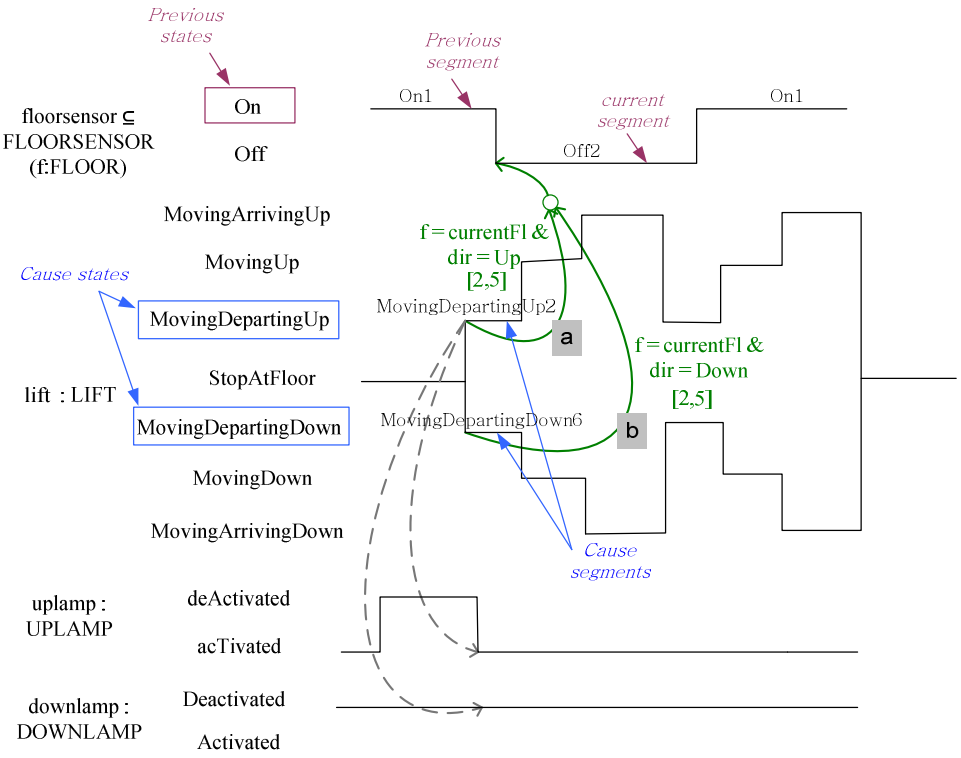


Figure 4-8 Timing diagram and named parts

For translation rules we describe in the later chapters, we repeatedly refer to parts of TD by using specific terms. We would like to describe those terms by using Figure 4-8: the `CauseEffectArrow` a and b.

- **Cause states**: *MovingDepartingUp* and *MovingDepartingDown* are causes that make a *floor sensor* change its state from *On* to *Off*. Thus, we say MovingDepartingUp and MovingDepartingDown are cause states of this `CauseEffectArrow`.
- **Cause segments**: a segment represents an object's state and the positon it apperars on the Timeline. Thus, *MovingDepartingUp2* and *MovingDepartingDown6* are segments that make a *floor sensor* change its state from *On* to *Off*. Thus, we say MovingDepartingUp2 and MovingDepartingDown6 are cause segments of this `CauseEffectArrow`.

- **Previous states**: A previous state is a state before the current state of interest. A state before *floor sensor* changes to be *Off* is *On*. Thus, we say *On* is a previous state.
- **Previous segments**: A previous segment is a segment before the current segment of interest. A segment before *floor sensor* changes to be *Off* is *On*. Thus, we say `On1` is a previous segment.

4.6 Preliminary Timing diagram editor

Working with a group design project from the School of Electronics and Computer Science, University of Southampton (Cobden, *et al.*, 2007), a preliminary TD editor plug-in was created. The interactive editor was created based on our TD notations (at that time), and used the Eclipse EMF and GMF frameworks. Figure 4-9 provides parts of a screenshot for the lift system from the TD editor window.

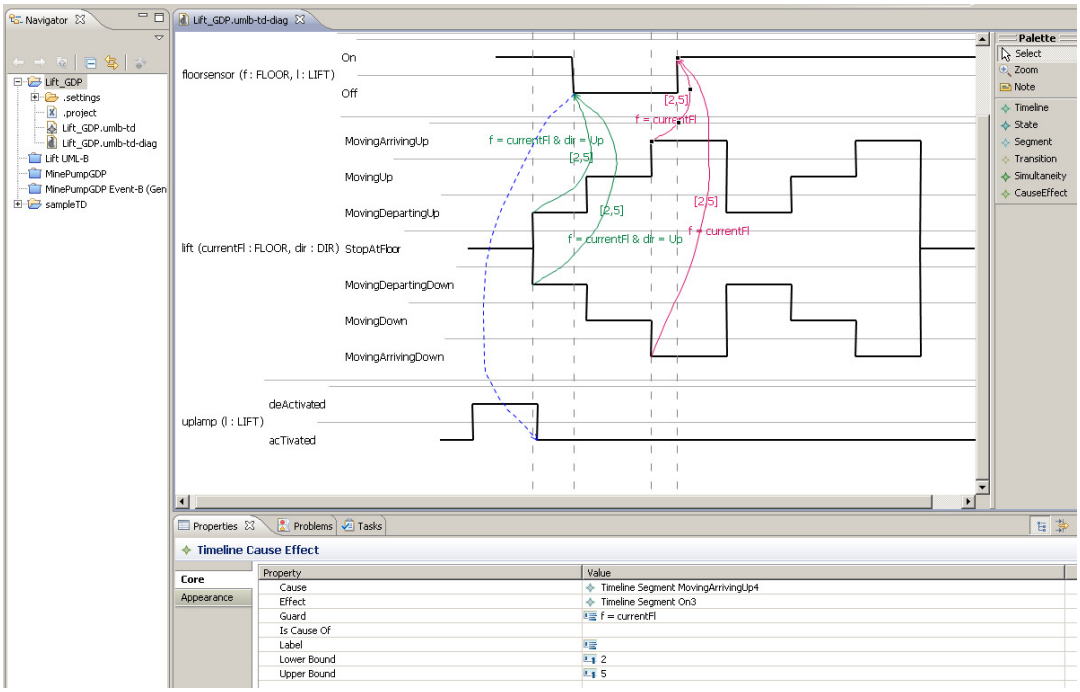


Figure 4-9 Timing diagram editor window

In its default configuration, the TD editor window displays the Navigator tab on the left side of the window. The Editor's tab is located in the top middle part of the window, and the palette is positioned on the top right part of the window. The three remaining tabs (Problems, Properties and Tasks) are located at the bottom of the window.

A TD can be created by selecting elements in the palette that are `Timeline`, `State`, `Segment`, `Transition`, `SimultaneityArrow` and `CauseEffectArrow`. A `Timeline` is used to represent an object in which one `Timeline` can be identified by many states. A `Timeline` comprises a number of `segments` that represent an object's state. A `Transition` is used to link individual `segments` in the same `Timeline`. A `CauseEffectArrow` is used to connect different objects' segments to identify causal dependency between `Timelines`. Time constraints are identified by `Lower Bound` and `Upper Bound` and are attributes of the `CauseEffectArrow`. A `SimultaneityArrow`, shown as a blue dashed-line in Figure 4-9, links a `CauseEffectArrow` and a `segment`. That is, the beginning of the `SimultaneityArrow` is the `CauseEffectArrow` and the end (with arrow) is the `segment`. This is different from the current TD in which a `SimultaneityArrow` links segments.

In Figure 4-9, a `Time synchronisation line` is represented by a vertical dashed-line and is used to synchronize duration constraints between objects. However, this notation is no longer used in the current TD notations. That is because the lines are not used for the translation. Moreover, it makes the diagram rather untidy, especially whenever there are many objects in it.

Although, the editor can generate most TD notations, it was created on an earlier TD metamodel version. Thus, it is not used to generate the TD as shown in this work. Moreover, the editor cannot specify the combination of "AND/OR" relationships for `CauseEffectArrow`, nor identify parameters for a model. Parameters of each `Timeline`, for example *l : LIFT* as shown in Figure 4-10, are simply created as textual descriptions.

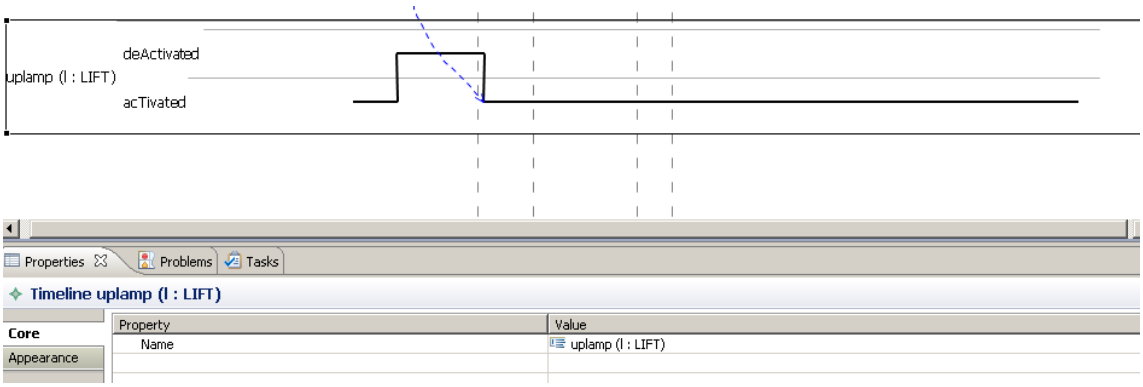


Figure 4-10 Timing diagram editor: Parameter

In this thesis, a TD is created from Microsoft Visio™ for the representation/visualisation. For translating TD into UML-B, the TD description was generated by EMF, whose detail is explained in Chapter 6.

4.7 Summary

This chapter shows the lift system specifications and TD notations used for translation. Some TD notations are obtained from UML 2.0 TD while others are introduced for the proper translation. The full detail of TD is generated from the case study represented in this chapter. A glossary for TD normally used in the later chapters is identified. A preliminary TD editor is discussed.

Chapter 5 Translating Timing Diagrams into Event-B models (direct translation)

This chapter describes translation rules used to transform a TD into an Event-B model. The clarification for what kind of the systems' specifications are suitable for description by TD has been explained at the beginning of chapter 4. There are two steps to create translation rules to transform TD into Event-B: defining TD BNF and identifying translation rules. Research by (Essalmi and Ayed 2006) proposed transformation rules of BNF and Extended BNF (ISO/IEC 2008) grammars to UML Class diagrams, while we have approached this in a different way. We identify TD BNF that describes features and relationships among the TD's notations. Then, translation rules are created by using a TD element as an input parameter for the rules to generate an Event-B model.

Section 5.1 explains the TD BNF definitions. Section 5.2 shows the corresponding Event-B parts are created from the top-level translation rules. Section 5.3 gives the basic translation rules and gives detailed examples, which are used to generate Event-B elements. The details of extra information required to complete the model are discussed in section 5.4.

5.1 TD BNF definition

This section introduces the TD BNF definitions in which are used to create translations rules to transform TD into an Event-B model. The BNF symbols (Métayer and Voisin 2007) are as follows.

- The symbol ::= means "is defined as". The element on the left of the symbol is defined by the expression on the right.
- The symbol | denotes alternative.
- Constructs within square brackets [...] are optional.
- Terminals are surrounded by quotes " ".
- The symbol a* represents n concatenated instances of a, where $n \geq 0$. The symbol a+ represents n concatenated instances of a, where $n \geq 1$
- Parentheses (...) are used for grouping.
- The symbol /* comment is for additional explanation; this symbol is not a part of the translation rules.

Figure 5-1 shows an example of how the TD BNF definitions are represented. Note that strings such as On1, Off2, …, MovingDepartingDown6, represent segment names that are generated by BNF definitions and are described later.

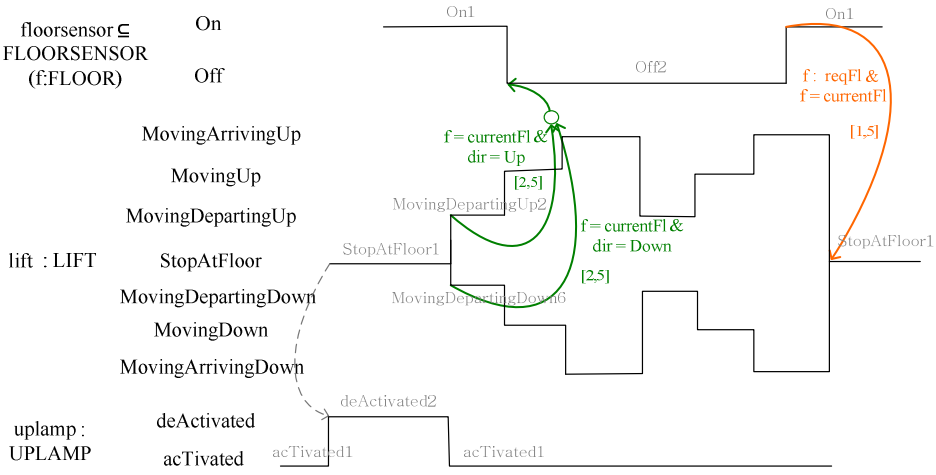


Figure 5-1 Timing diagram for floorsensor, lift and uplamp

(Parts of Figure 5-2)

A TD project (Project) is represented by a name and is composed of at least one TD machine (Machine). We decided to have many Machines in a Project to correspond with the UML-B metamodel (in chapter 6). A TD machine has a name (must be unique) and comprises one class as a minimum. A class is defined by a name (ClassName), at least one object and an object definition (Obj_Def).

```
Project ::= name Machine+
Machine ::= name Class+
name ::= String
Class ::= ClassName Obj+ Obj_Def
ClassName ::= String
```

In TD, we allow naming of an object and its class to indicate whether the object occurs singly (:) or multiply (⊆) in the system. This naming is defined by Obj_Def.

```
Obj_Def ::= ObjName " ⊆ " Class_Clause | ObjName " : " Class_Clause
ObjName ::= String
Class_Clause ::= ClassName |
            ClassName"(" Param ":" ParamType ( "," Param ":" ParamType)* ")"
Param ::= name   /* represents parameter's name
ParamType ::= name      /* represents parameter's type's name
```

For example in a lift case study (as shown in Figure 5-1), there is only one lift in the system. Thus, an Obj_Def for the lift is declared as *lift : LIFT*. In contrast, there is a floor sensor in every floor, *floorsensor* ⊆ *FLOORSENSOR* is defined.

A class may have parameters (Param) with parameter types (ParamType) in which both of them are defined by a string. A parameter is used to indicate the specific object of interest from the set. For example, in the case of an object *floorsensor*, a parameter *f : FLOOR* identifies which *floorsensor* it is where *f* is a parameter with a type *FLOOR*. Thus, the complete identification for an object

floorsensor is declared by *floorsensor* $\subseteq$ *FLOORSENSOR(f:FLOOR)*. The parameter tells which object one is using in that case and that information is required for the translation. This is the way one can introduce information for the translation.

Obj :: = ObjName ObjSt$^+$ Timeline

An object is defined by a name (ObjName). It is composed of at least one object's state (ObjSt) and a Timeline. A Timeline represents a chain of an object's states changing in a class. Since one object has one Timeline, a Timeline's name is defined by the same name as its corresponding object's name. A Timeline is composed of at least one segment. A segment is presented by a corresponding object's state's name followed by a positive integer. For example, On1, Off2 and On1, in Figure 5-1 represent segments for the object *floorsensor*.

Timeline ::= name Segment$^+$

Segment ::= ObjSt number Simul* [CauseEffectArrow]

ObjSt ::= name

number ::= Z^+

Simul ::= StartSegmt EndSegmt

StartSegmt ::= Segment

EndSegmt ::= Segment

One segment is composed of zero or more SimultaneityArrows (Simul). A SimultaneityArrow links a segment (StartSegmt) and another segment (EndSegmt). For example, in Figure 5-1, there is one SimultaneityArrow in which *StopAtFloor1* and *deActivated2* are StartSegmt and EndSegmt respectively. Presently, we do not allow a SimultaneityArrow in the same segment, nor combinations of SimultaneityArrow using "AND" or "OR" nodes. A segment can have a CauseEffectArrow which is optional.

CauseEffectArrow ::= Constraint

Constraint ::= NodeType

NodeType ::= Simple | OR_node | AND_node

Simple ::= CauseSegmt [Timing] [Predicate*]

CauseSegmt ::= Segment

Predicate ::= String

OR_node ::= NodeType ○ NodeType

AND_node ::= NodeType ● NodeType

Timing ::= "[" lowerlimit "," upperlimit "]"

lowerlimit ::= Z^+

upperlimit ::= Z^+

A CauseEffectArrow is actually used to define a constraint (Constraint) between segments. This 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 combinations of cause segments. A Simple consists of a cause segment (CauseSegmt), an optional timing constraint (Timing) and an optional string condition (Predicate). A timing constraint is declared as a pair of positive integer values: a lower bound (lowerlimit) and an upper bound (upperlimit).

For example, in Figure 5-1, a segment Off2 has a CauseEffectArrow which is declared by an OR_node. The BNF definitions of this CauseEffectArrow are identified as in the following.

First, from the BNF definition CauseEffectArrow ::= Constraint, it is applied to CauseEffectArrow ::= OR_node

Second, from the BNF definition OR_node ::= NodeType ○ NodeType,
each NodeType is replaced with Simple that are segments MovingDepartingUp2 and MovingDepartingDown6

Third, from the BNF definition Simple ::= CauseSegment [Timing] [Predicate*],
each Simple is given Timing and Predicate values thus:
Simple ::= MovingDepartingUp2 [2, 5] f = currentFl & dir = Up
Simple ::= MovingDepartingDown6 [2, 5] f = currentFl & dir = Down

Finally, the `CauseEffectArrow` for the segment `Off2` is defined as

MovingDepartingUp2 [2, 5] f = currentFl & dir = Up ○

MovingDepartingDown6 [2, 5] f = currentFl & dir = Down

A TD used for transforming into an Event-B model is illustrated in Figure 5-2.

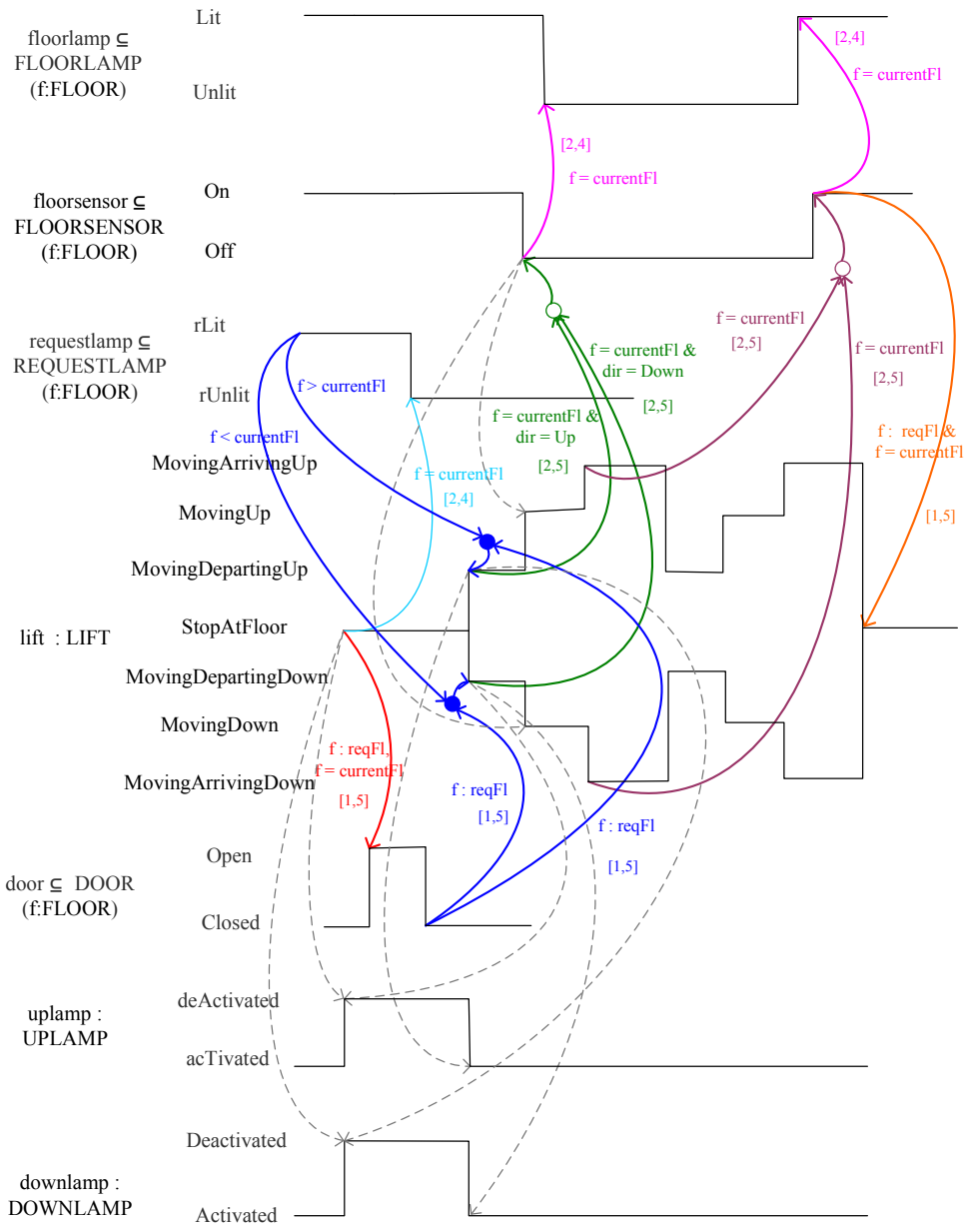


Figure 5-2 Timing diagram for an Event-B model direct translation

5.2 Event-B model parts vs. Top-level textual translation rules

This section gives the whole picture of how each Event-B model's part is generated from corresponding top-level textual rules as shown in Figure 5-3. In this figure, the blue coloured boxes represent parts generated from the rules, and the dotted boxes represent parts the extra information added for the model completion.

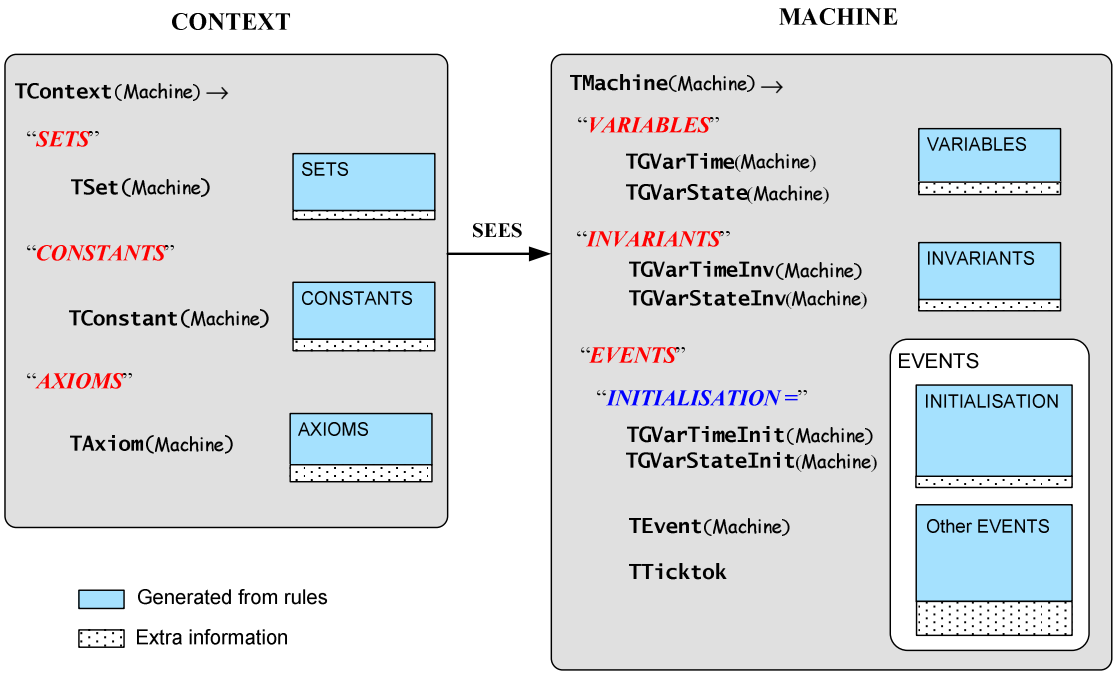


Figure 5-3 Event-B model's parts correspond with top-level textual rules

The translation rules cover generating **CONTEXT** and **MACHINE** parts for an Event-B model are now described.

For the *Context* part, the rules **`TSet`**, **`TConstant`** and **`TAxiom`** use `Machine` as an input parameter to create sets, constants, and axioms for the model respectively. The details of those rules are described in section 5.3.1 below. TD notations, that can be used to directly generate a **CONTEXT** part, are classes and objects' states. However, if one intended to identify extra information that cannot be identified by TD, such as a specific member of a class, e.g. there are three floors for the lift system, or extra sets provide supportive information for the system, e.g. the directions (*DIR*) of a lift movement can be only `Up` and `Down`, this information has

to be manually added. For example, identifying a set *DIR* in a **CONTEXT** is shown in Figure 5-4. The set's name is declared in ***SETS***, each element of a set is defined as a constant in ***CONSTANTS***, while a set's name assigned to its element values is identified in ***AXIOMS***.

```
SETS
  DIR
CONSTANTS
  Up
  Down
AXIOMS
  DIR = {Up, Down}
```

Figure 5-4 A set DIR

For a **MACHINE** part, rules **`TGVarTime`** and **`TGVarState`** are used to generate machine variables. Most of machine's variables are generated by the rules. However, it may have some variables that are manually added. Those variables are actually used in a `CauseEffectArrow` predicate. For example, in case of the lift system, *reqFl* and *currentFl* are variables that are added by hand and used to represent a list of requested floors and a current lift position respectively.

Variables that can be generated by the translation rules have to define their invariants in an ***INVARIANTS*** part. This can be done by using rules **`TGVarTimeInv`** and **`TGVarStateInv`**. Additional invariants may be appended in this step. For example, a condition that defines that an up lamp and a down lamp must not be activated at the same time, and the lift door must not open while the lift is moving.

Events in a machine comprise two kinds: an ***INITIALISATION*** event and other events. The ***INITIALISATION*** event is used to declare variables' initial states, which are created by rules **`TGVarTimeInit`** and **`TGVarStateInit`**. The other events are defined by a rule **`TEvent`**. There are some events which cannot be created by translation rules. For example, an event that changes the direction of the lift, and events that represent the lift continue moving for many floors before stopping. That is because in the first example, this information cannot be

represented by TD notations; in the second example, this information is not represented by Segments but states while the rules use Segments in generating an event (as described in section 5.3.3 below).

5.3 Translation rules

This section demonstrates the translation rules that are used for transforming TD into an Event-B model. In these translation rules, a component using bold typewriter font demonstrates a name of the translation rule, e.g. **TEvent**. A plain string inside angle brackets, e.g. <IF> and <THEN>, is a keyword in the macro translation language. TD language elements are defined in the same font as TD BNF definitions, e.g. Objst. The Event-B parts are shown using italic font written in quotations, e.g. "*Time*" and "***WHEN***".

The following table identifies the whole set of basic rules generally used for translation. Note that this table does not contain compound translation rules that appear in the following sections, but only those fundamental rules that are usually used. The details of the complex rules are given in Appendix A.

last((elem1, elem2, …, elemn)) → elemn; this rule produces the last element for an input sequence of elements.
TAllInstances(NodeType) → (NodeType1, NodeType2, …); this rule produces a sequence containing the instances which are sub-NodeType of an input NodeType.
TAllParam(*Class*) → (Param1, Param2, …); this rule produces the sequence of parameters for an input *Class*.
TAllParamType(*Class*) → (ParamType1, ParamType2, …); this rule produces the sequence of parameter types for an input *Class*.
TAllPrevSegm(Segment) → (Segment1, Segment2, …); this rule produces a sequence containing all the previous segments for an input Segment.
TAllState(Timeline) → (Objst1, Objst2, …); this rule produces the sequence of objects states for an input Timeline.

TAllTimeline(Machine) → (Timeline1, Timeline2, …); this rule produces the sequence of Timelines for an input Machine.

TClass(Obj) → Class; this rule produces the class for an input object.

TClassName(Timeline) → ClassName; this rule produces the class's name for an input Timeline.

TCond(Simple) → (Predicate1, Predicate2, …); this rule produces the sequence of Predicates for an input Simple.

TConstrnt(Segment) → Constraint; this rule produces the Constraint for an input Segment.

TEmpty(set) → BOOL; this rule checks whether an input set is empty. If so, the rule produces the Boolean value true.

TEndSegm(Simul) → EndSegmt; this rule produces the EndSegmt for an input SimultaneityArrow.

TGetSegmentWithConstrnt(Machine) → (Segment1, Segment2, …); this rule produces a sequence containing all the segments defined with Constraints for an input Machine.

THasParam(Class) → BOOL; this rule checks whether an input Class has parameters. If so, the rule produces the Boolean value true.

THasTiming(Simple) → BOOL; this rule checks whether an input Simple node has been defined with timing constraints. If so, the rule produces the Boolean value true.

TLowerLmt(Timing) → lowerlimit; this rule produces the lowerlimit value for an input Timing.

TName(Timeline) → name; this rule produces the Timeline's name for an input Timeline.

TNodeType(Constraint) → NodeType; this rule produces the NodeType for an input Constraint.

TObj(ObjSt) → Obj; this rule produces the object for an input object state.

TObjName(Obj) → ObjName; this rule produces the object name for an input object.

TObjSt(Segment) → ObjSt; this rule produces the object state for an input Segment.
TParamType(Param) → ParamType; this rule produces the parameter types for an input parameter.
TSegment(Simple) → Segment; this rule produces the Segment value for an input Simple.
TSimulSeq(Segment) → (Simul1, Simul2, …); this rule produces a sequence of SimultaneityArrow for an input Segment.
TTimelineInClass(Timeline) → Class; this rule produces the Class for an input Timeline.
TTiming(Simple) → Timing; this rule produces the Timing value for an input Simple.
TUpperLmt(Timing) → upperlimit; this rule produces the upperlimit value for an input Timing.

Table 5-1 Basic rules for TD to Event-B translation

5.3.1 *Translation rules for creating a set in the Context part*

The **CONTEXT** part is used to identify static values such as sets, constants and axioms in an Event-B model. Here, we describe how translation rules create the **CONTEXT** part. The rule **TSet** (Figure 5-3) is used to create a set's name in which each element in a set is defined as a constant with the rule **TConstant**. The rule **TAxiom** genereated axioms which are declaration of sets's names followed by their elements. Below is an explanation of the rules for **TAxiom**, while the detail of the rules **TSet** and **TConstant** can be found in Appendix A.

The rule **TAxiom**, Figure 5-5, uses a Machine as an input value and recursively generates a list of states as elements for a set. Each axiom is created by a Timeline which is represented by an iterator t. This rule creates a set name followed by the list of the set's elements. Those elements are generated by a sub-rule **TWriteAllStates** which uses Timeline as an input parameter.

TAxiom(Machine) →

<FOR> t <IN> **TAllTimeline**(Machine)

{ **TClassName**(t) + "*_STATES* = " + **TWriteAllStates**(t) }

TWriteAllStates(t) → "*{*" + **TAllStateLst**(**TAllState**(t)) + "*}*"

TAllStateLst(Head : SeqTail) → Head + "," + **TAllStateLst**(SeqTail)

TAllStateLst(Head : < >) → Head

Figure 5-5 Rule **TAxiom** : creating axioms in an Event-B Context

For example, the rule **TAxiom** generates an axiom for a `Timeline` *floorsensor* as *FLOORSENSOR_STATES = {On, Off}*.

5.3.2 *Translation rules for creating variables and their initial values*

Variables are dynamic parts of a machine and are used to maintain local state information. There are two kinds of variable that can be generated from a TD: variables used to record timing constraints and variables used to record state values.

Variables used to record timing constraints. Whenever a segment has a `CauseEffectArrow`, that means it may have timing constraints between objects. If so, this timing must be recorded and used as guards for synchronising corresponding events. Thus, each event must record a current time in its related machine variables whenever that event is performed. In doing that, the rules **TGVarTime**, **TGVarTimeInv** and **TGVarTimeInit** are used to identify variables, their invariants and initial values respectively. Below is the detail of the rule **TGVarTime**.

TGVarTime(Machine) →

<FOR> t <IN> **TAllTimeline**(Machine)

{<FOR> s <IN> **TAllState**(t)

{**TName**(t) + s + "*Time*" } }

Figure 5-6 Rule **TGVarTime**: creating machine variables to record time

This rule uses a Machine as an input value. It collects `Timeline` from the Machine and then uses it to generate each variable. A variable is generated from a `Timeline`'s name followed by each state of the `Timeline` and a string *Time*. For example in a lift system, there are seven `Timelines`: *floorlamp*, *floorsensor*, *requestlamp*, *lift*, *door*, *uplamp* and *downlamp*. The rule **`TGVarTime`** generates variables from each timeline. For the `Timeline` *floorsensor*, it creates two variables: *floorsensorOnTime* and *floorsensorOffTime*. The invariants of these variables are defined by the rule **`TGVarTimeInv`** as *floorsensorOnTime* ∈ ℕ and *floorsensorOffTime* ∈ ℕ. Initial variables' values are generated by the rule **`TGVarTimeInit`** as *floorsensorOnTime := 0* and *floorsensorOffTime := 0*. The details of the rules **`TGVarTimeInv`** and **`TGVarTimeInit`** are shown in Appendix A.

Variables used to record state values. Since an object changes its state based on the constraints it satisfies, it is necessary to have a variable to record the object's current state. These kinds of variable are used for synchronising events. As shown in Figure 5-3, these variables are generated by the rule **`TGVarState`**, while their invariants and initial values are created by rules **`TGVarStateInv`** and **`TGVarStateInit`** respectively. Below is the detail of the rule **`TGVarStateInv`**.

`TGVarStateInv`(Machine) →
<LET> exp = **`TClass`**(**`TTimelineInClass`**(t)) ..... (1)
<IN> <FOR> t <IN> **`TAllTimeline`**(Machine) ..... (2)
{**`TName`**(t) + "*State* ∈ " + ... (3)
<IF> **`THasParam`**(**`TClass`**(**`TName`**(t))) ..... (4)
<THEN> ..... (5)
"(" + **`TWriteParamForInv`**(**`TAllParamType`**(exp)) + ")" ..... (6)
" → " + **`TClass`**(**`TName`**(t)) + "*_STATE*" ..... (7)
<ELSE> **`TClass`**(**`TName`**(t)) + "*_STATE*" ..... (8)
<ENDIF> ..... (9)
} ..... (10)
`TWriteParamForInv`(Head : SeqTail) → ..... (11)

Head + "×" + **TWriteParamForInv**(SeqTail) ..... (12)

TWriteParamForInv(Head : < >) → Head ..... (13)

Figure 5-7 Rule **TGVarStateInv**: creating machine variables to record states

This rule uses a Machine as an input value. It collects `Timeline` from the Machine and then uses it to generate each variable as shown at line (2). A variable is generated from a `Timeline`'s name followed by a string "*State* ∈ " at line (3). If a corresponding class has parameter, the output string from line (3) is concatenated with parameter type at line (6) followed by a class name and the string "*_STATE*", at line (7). If the corresponding class has no parameter, then line (8) is performed instead.

Sub-rule **TWriteParamForInv** is called from line (6) whenever the corresponding class has a parameter. This sub-rule is defined recursively to give parameter types for that class. For example, an invariant is created from this `Timeline` *floorsensor* shown in the following:

TGVarStateInv(Machine) →
<LET> exp = **TClass**(**TTimelineInClass**(t))
<IN>
<FOR> t <IN> **TAllTimeline**(Machine)..*floorsensor, floorlamp, requestlamp,* etc.
{**TName**(t) + "*State* ∈ " + ...*floorsensorState* ∈
<IF> **THasParam**(**TClass**(**TName**(t))) ..... *TRUE*
<THEN>
"("+ **TWriteParamForInv**(**TAllParamType**(exp)) + ")" ..... *(FLOOR)*
+ " → " + **TClassName**(t) + "*_STATE*" ..... → *FLOORSENSOR_STATE*
<ELSE> **TClassName**(t) + "*_STATE*"
<ENDIF>
}
TWriteParamForInv(*FLOOR* : < >) → *FLOOR* ..... *FLOOR*

Output: *floorsensorState* ∈ *(FLOOR)* → *FLOORSENSOR_STATE*

Suppose a class *floorsensor* has two parameters, *f : FLOOR* and *a : AA*, the sub-rule **TWriteParamForInv** generates a relationship between those parameters as *(FLOOR × AA)*. Thus, an invariant in this case is:

$$floorsensorState \in (FLOOR \times AA) \rightarrow FLOORSENSOR\_STATE$$

From Figure 5-2, one may expect that an object state's initial value can be generated from the first segment in the `Timeline`. For example, the first segment of the object *door* is *Closed*, in which the corresponding variable generated by the rule **TGVarState** is *doorState*. Thus, by the rule **TGVarStateInv**, an invariant for this variable is created as $doorState \in FLOOR \rightarrow DOOR\_STATE$. This variable has its initial value defined as *doorState := FLOOR × {Closed}*. That means, at the initial state, the door for every floor is closed. However, it is incorrect to use the first segment as an initial state for every object. For example, an object *floorsensor* has a first segment *On*, but one cannot identify its initial state directly as *floorsensorState := FLOOR × {On}*. That is because the *floorsensorState* for that floor is set *On* if an only if the lift is at that floor. Thus, it is not true that at the initial state, the lift stations at every floor. In fact, in the beginning if the lift is stationed at the first floor, then only the *floorsensorState* at the first floor is set *On*. If there are three floors in a system, the initial value for the *floorsensorState* is *floorsensorState* := $\{1 \mapsto On, 2 \mapsto Off, 3 \mapsto Off\}$ where 1, 2 and 3 denotes the number of the floors.

TGVarStateInit(Machine) →
  <FOR> t <IN> **TAllTimeline**(Machine)
    {**TName**(t) + "*State := {xInitValuex}*" }

Figure 5-8 Rule **TGVarStateInit**: creating initial values for those variables used to record states

Thus, the rule **TGVarStateInit**, which is used to define the initial states of these variables need to be generated by hand, which is represented by a marking *xInitValuex*.

Other examples of variables that have to be generated by hand are *dir* and *currentFl*, which are used to indicate the lift direction and the current position for the lift. Actually, these variables are already shown as a string as the `CauseEffectArrow`'s predicates. However, one cannot generate variables from the predicates as it is not a notation but a string of conditions.

5.3.3 *Structure of Translation rules for creating an Event-B event*

Each Event-B event is created by the rule **TEvent**. This rule uses a Machine for an input parameter and is defined recursively. The rule **TEvent** is composed of sub-rules as shown below.

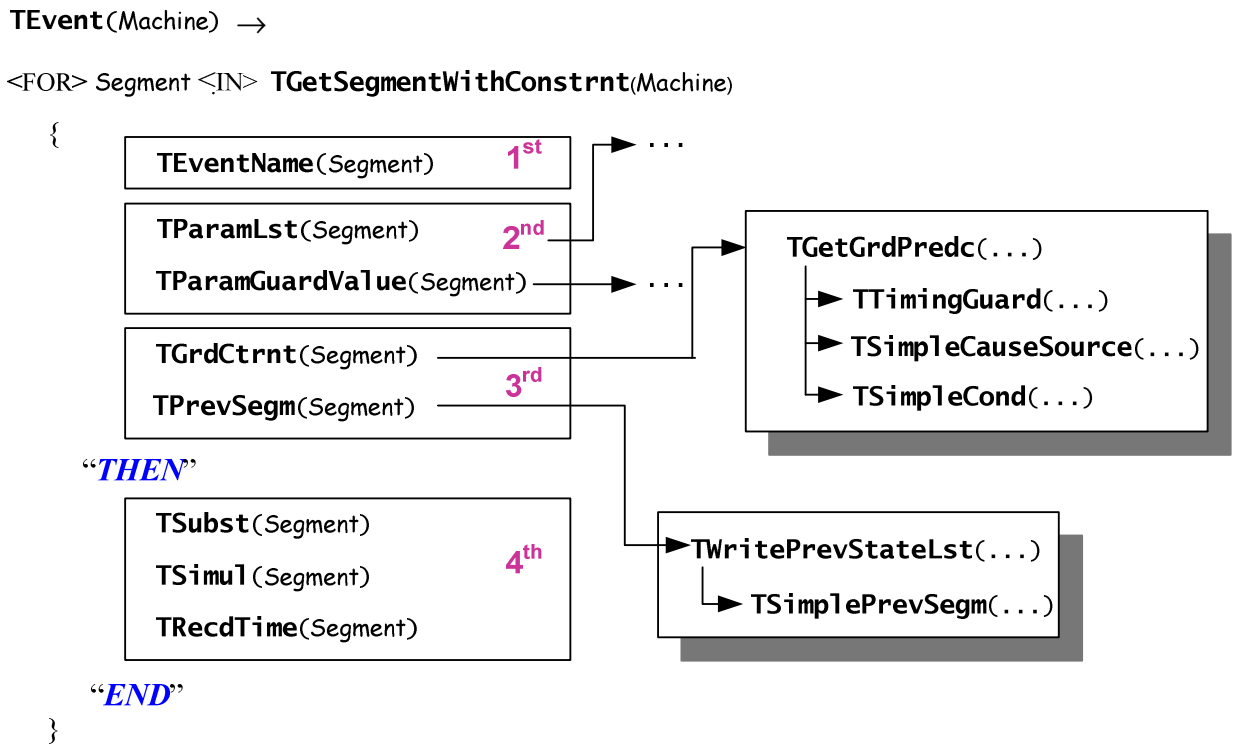


Figure 5-9 Structure of translation rules to create an Event-B event

To generate events, first, the rule **TGetSegmentWithConstrnt**(Machine) is used to collect only `Segments` defined with constraints – i.e. that segment has a `CauseEffectArrow` – from a machine. Without `CauseEffectArrow`, a `Segment` is an ordinary segment. It does not have a causal dependency between objects and will not be considered to generate an event. Next, each `Segment` from the collection is used to generate an individual event.

An Event-B event is basically composed of a name, guards and actions, thus the rule **TEvent** is designed to generate those parts. The rule **TEvent** is sub-divided into four groups.

1st group: this group has a rule **TEventName** (detailed in Figure 5-11) that is used to create an event's name.

2nd group: this group comprises translation rules that are used to create guards for an event. As described in chapter 2, an event can be defined into three types: *Simple*, *Guards* and *Non deterministic*. The rules in the 2nd group are used to define *Guards* and *Non deterministic* types, not the *Simple* type. Since the *Simple* type has only the action part but not guards, it is inappropriate to generate this type from the TDs. TDs are designed to explain the changing of state according to conditions, which are guards.

As shown in Figure 5-10, for the *Non deterministic* type, the rule **TParamLst** (detailed in Figure 5-13) is used to create a string ***ANY*** and a list of local variables; the rule **TParamGuardValue** (detailed in Appendix A) is used for identifying those local variables with their corresponding types. For the *Guard* type, the rule **TParamLst** is used to create a string ***WHEN***.

3rd group: this group comprises translation rules used to create event guards. Those guards are created from four features that are associated with that `Segment`: previous segments, cause segments, conditions, and timing constraints. A rule **TGrdCtrnt** (detailed in Figure 5-14) is used to create guards from cause segments, conditions, and timing constraints. A rule **TPrevSegm** (detailed in Appendix A) is used to create a guard from previous segments. Most of the guards are generated from those rules. However, some additional guards may be added. Most of them

are associated with extra variables generated manually as covered earlier in **CONTEXT**.

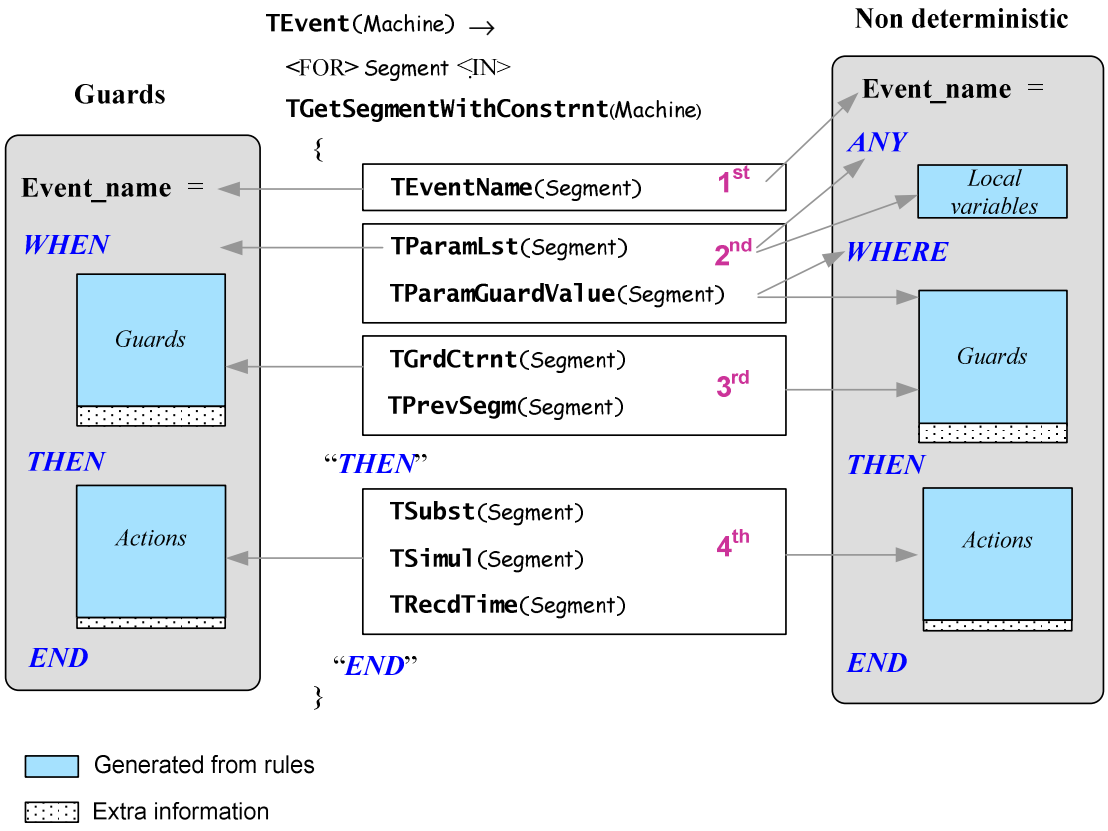


Figure 5-10 Structure of translation rules and Event-B model types

4th group: this group comprises translation rules used to create events' actions. There are three kinds of actions generated here. First, an action is generated from a segment with constrints, by a rule **`TSubst`** (detailed in Figure 5-20). Secondly, if a `Segment` has `SimultaneityArrows`, an action is created by the rule **`TSimul`** (detailed in Figure 5-22). Thirdly, actions are created to record the current time of a corresponding machine variable whenever the event is activated, by the rule **`TRecdTime`** (detailed in Figure 5-23). The rules generate mostly essential actions. However, in some events, actions may need to be added. For example, in the case of the lift system, it has to add actions to update current floor position whenever the lift is moving up or moving down.

5.3.4 *Creating an event's name*

To create an event's name, the **TEventName** rule is used. This rule gives an event's name for an input Segment and uses basic rules, i.e. **TObj** and **TObjSt**, as described in Table 5-1.

TEventName(Segment) →

<LET> exp = **TObjSt**(Segment)

<IN> **TObj**(exp) + exp + "="

Figure 5-11 Rule **TEventName**: creating an event's name

This rule creates an event's name by concatenating an object's name with an object state's name followed by the "=" symbol.

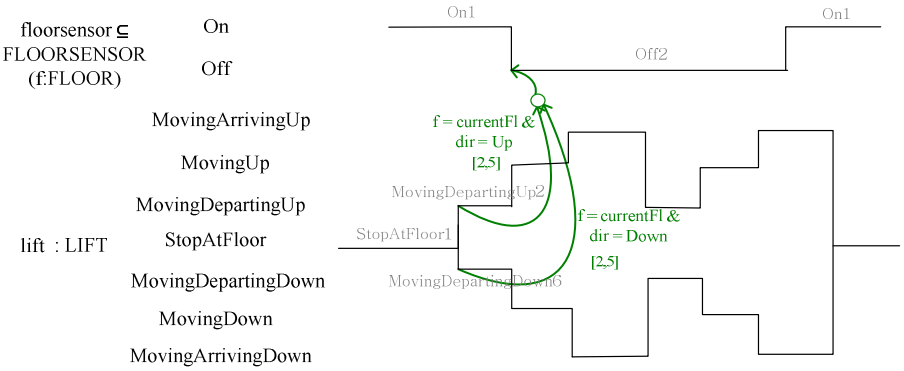


Figure 5-12 Timing diagram for floorsensor and lift (parts of Figure 5-2)

For example in Figure 5-12, Segment Off2 has a CauseAffectArrow in which MovingDepartingUp2 or MovingDepartingDown6 are cause segments that stimulate the object *floorsensor* to change its state from *On* to *Off*. Generating an event's name from the Segment Off2 is illustrated below:

TEventName(Off2) →

<LET> exp = **TObjSt**(Off2)

<IN> **TObj**(exp)..... *floorsensor*

+ exp + "=" ..... *Off =*

Output: *floorsensorOff* =

5.3.5 *Creating non-deterministic local variables and their values*

A rule **TParamLst** is used to check whether an event is defined by *Guard*s or *Non deterministic* type. Each of these types identify the beginning of the guards with a string ***WHEN*** or ***ANY*** corresponding to a type *Guards* or *Non deterministic* respectively. This rule uses a Segment as input parameter.

TParamLst(Segment) →

<LET> exp = **TObj**(**TObjSt**(Segment)) .....(1)

<IN>.....(2)

<IF> **THasParam**(**TClass** (exp)) .....(3)

<THEN>.....(4)

"***ANY***" + ... (5)

TWriteAllParams(**TAllParam**(**TClass**(exp))) .....(6)

<ELSE>.....(7)

"***WHEN*** " .....(8)

<ENDIF>.....(9)

TWriteAllParams(Head : ParamSeqTail) →

Head + "," + **TWriteAllParams**(ParamSeqTail)

TWriteAllParams(Head : < >) → Head

Figure 5-13 Rule **TParamLst**: creating a list of local variables for an event

The rule **TParamLst** checks whether a class corresponding to the input Segment has a parameter at line (3). If so, this rule generates a string ***ANY***, line (5), followed by a list of parameters as shown in line (6); those parts are for creating *Non-deterministic* type. Otherwise, it creates a string ***WHEN*** for *Guarded* type, as shown in line (8). The list of parameters is generated by a sub-rule **TWriteAllParams**. This rule iteratively generates parameters, each of them being separated by a "," symbol. For example, a segment Off2 in Figure 5-12, belongs to an object *floorsensor* which resides in a class *FLOORSENSOR* whose parameter is *f*. An example of creating a local variable from this segment is illustrated below.

TParamLst(Off2) →
<LET> exp = **TObj**(**TObjSt**(Off2))
<IN>
  <IF> **THasParam**(**TClass** (exp)) ..... *True*
  <THEN>
    "***ANY***" + ... *ANY*
    TWriteAllParams(**TAllParam**(**TClass**(exp)))
  <ENDIF>
TWriteAllParams(f : ‹ ›) → f

Output: ***ANY*** *f*

Each local variable needs to identify its types within ***WHERE*** clauses. Rule **TParamGuardValue** (Figure 5-10, detailed in Appendix A) is used to identify the variables' types. For example, within the same example above, this rule generates ***WHERE*** *f : FLOOR* for output.

Suppose a class *FLOORSENSOR* has parameters *f*, *a* and *b* with a type *FLOOR*, *AA* and *BB* respectively. Thus, the rule **TParamGuardValue** would generate ***WHERE*** *f : FLOOR & a : AA & b : BB*. The detail of this rule is shown in Appendix A.

5.3.6 *Creating an Event's guards*

As shown in Figure 5-10, event guards are created by the rule **TGrdCtrnt** and **TPrevSegm**. This section explains how to create guard from the rule **TGrdCtrnt**, while the detail of the rule **TPrevSegm** can be found in Appendix A.

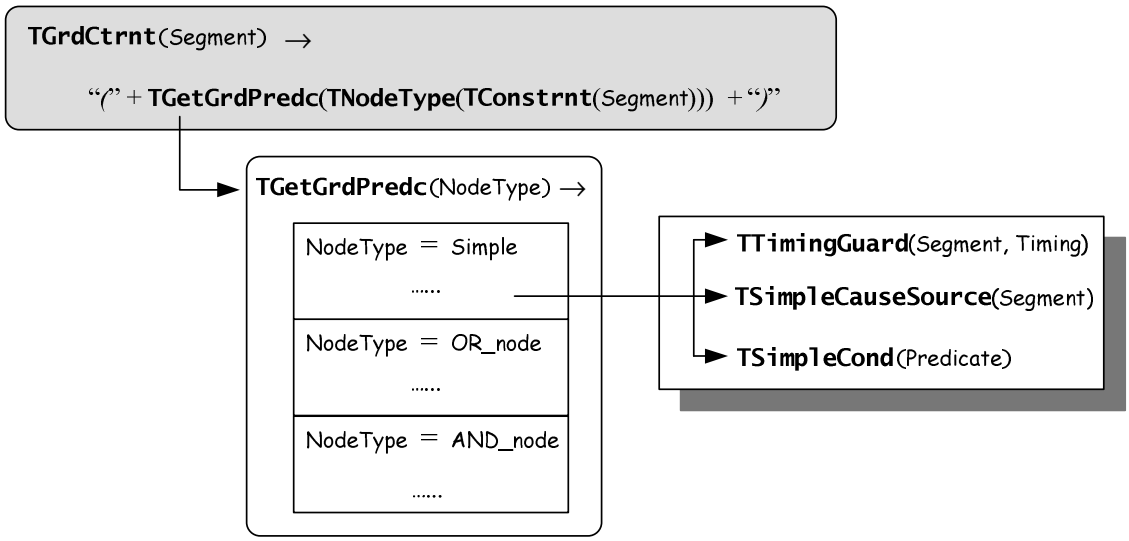


Figure 5-14 Rule **TGrdCtrnt** and sub-rules

The detail of the rule **TGrdCtrnt** is shown in Figure 5-14, a coloured box. This rule gives an output `NodeType` for an input `Segment`. The `NodeType` then is used as an input parameter for the sub-rule **TGetGrdPredc**.

The rule **TGetGrdPredc** checks whether the input `NodeType` is a Simple, OR_node or AND_node. If `NodeType` is Simple, three other sub-rules, **TTimingGuard**, **TSimpleCauseSource** and **TSimpleCond**, are called in order to generate guards from timing constraints, cause segments and conditions respectively. If the `NodeType` is OR_node or AND_node, the rule **TGetGrdPredc** is recursively called. The detail of the rules **TGetGrdPredc** is illustrated as follows.

TGetGrdPredc(NodeType) →

1st part: If NodeType is Simple

<IF> NodeType = Simple.....(1)

<THEN><IF> **THasTiming**(Simple) .....(2)

<THEN> **TTimingGuard**(**TSegment**(Simple), **TTiming**(Simple)).....(3)

+ "&" + **TSimpleCauseSource**(**TSegment**(Simple)) .....(4)

+ **TSimpleCond**(**TCond**(Simple)) .....(5)

<ELSE>.....(6)

TSimpleCauseSource(**TSegment**(Simple)) .....(7)

+ **TSimpleCond**(**TCond**(Simple)) .....(8)

<ENDIF> .....(9)

2nd part: If NodeType is Or_node

<ELSE><IF> NodeType = OR_node..... (10)

<THEN><LET> Nodes = **TAllInstances**(OR_node).....(11)

<IN> Nodes → <ITERATE> (n; ret : String = "(" | .....(12)

<IF> n = **last**(Nodes).....(13)

<THEN> ret = ret + **TGetGrdPredc**(n) + ")" .....(14)

<ELSE> ret = ret + **TGetGrdPredc**(n) + ")∨(" .....(15)

<ENDIF>).....(16)

<ENDIF>.....(17)

3rd part: If NodeType is And_node

<ELSE><IF> NodeType = AND_node

<THEN><LET> Nodes = **TAllInstances**(AND_node) .....(18)

<IN> Nodes → <ITERATE> (n; ret : String = "(" | .....(19)

<IF> n = **last**(Nodes).....(20)

<THEN> ret = ret + **TGetGrdPredc**(n) + ")".....(21)

<ELSE> ret = ret + **TGetGrdPredc**(n) + ")∧(" .....(22)

<ENDIF>).....(23)

<ENDIF>.....(24)

<ENDIF> .....(25)

Figure 5-15 Rule **TGetGrdPredc**: creating event guards from timing constraints, cause segments and conditions

For example if a NodeType is OR_node, a rule **TAllInstances**(OR_node) at line (11) collects the elements underneath the OR_node as a sequence within a variable Nodes. Line (12) is defined as iteration in which an iterative expression is defined by the ATL-like syntax as in the following:

$$source \rightarrow \langle ITERATE \rangle (iterator;\ return\_var\_declaration :\ return\_var\_type = init\_expression \mid body)$$

That is, at line (12), the variable Nodes is a source of iteration process when n is an iterator. This iteration returns a variable ret which is defined as a string provided with an initial value equal to "*(*". Line (13) checks whether n is the last element in the sequence. If so at line (14), the return value ret is concatenated with string value from calling itself, **TGetGrdPredc**(n), followed by the "*)*" symbol. If not, line (15), the return value ret is concatenated with string value from calling itself followed by the string "*)* $\vee$ *(*".

For example, from Figure 5-16, the Segment Off2 is used to generate guards for the event *floorsensorOff* by the rule **TGetGrdPredc**. The process of generating guards can be done step by step as shown in Figure 5-17. Note that, we present order numbers such as 1, 2 and 3 to show which parts of the CauseEffectArrow are used in the rule **TGetGrdPredc**; these numbers are not TD notations.

Step 1, the rule **TGrdCtrnt** (from Figure 5-14) gives a NodeType which is equal to OR_node as an output.

Step 2, the rule **TAllInstance**(OR_node), at line (11) in Figure 5-15, collects all NodeTypes beneath this OR_node and keeps them in a variable Nodes as a sequence. Remember that, since the Simple BNF definition is defined as Simple ::= CauseSegmt [Timing] [Predicate*], the variable Nodes has two Simple elements as shown in the following:

Nodes = (Simple1, Simple2)
where Simple1 = MovingDepartingUp2 [2,5] f= currentFl & dir = Up
Simple2 = MovingDepartingDown6 [2,5] f= currentFl & dir = Down

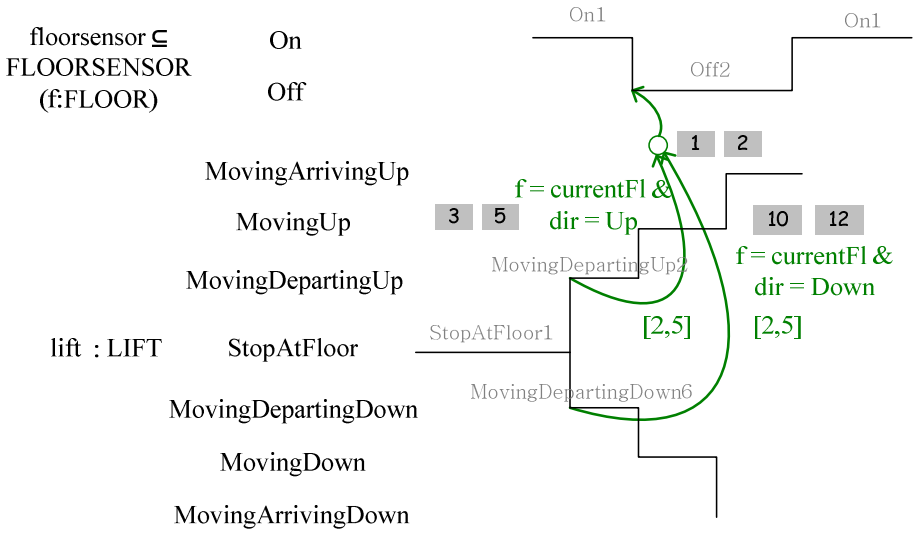


Figure 5-16 Timing diagram for floorsensor and lift (same as Figure 5-6)

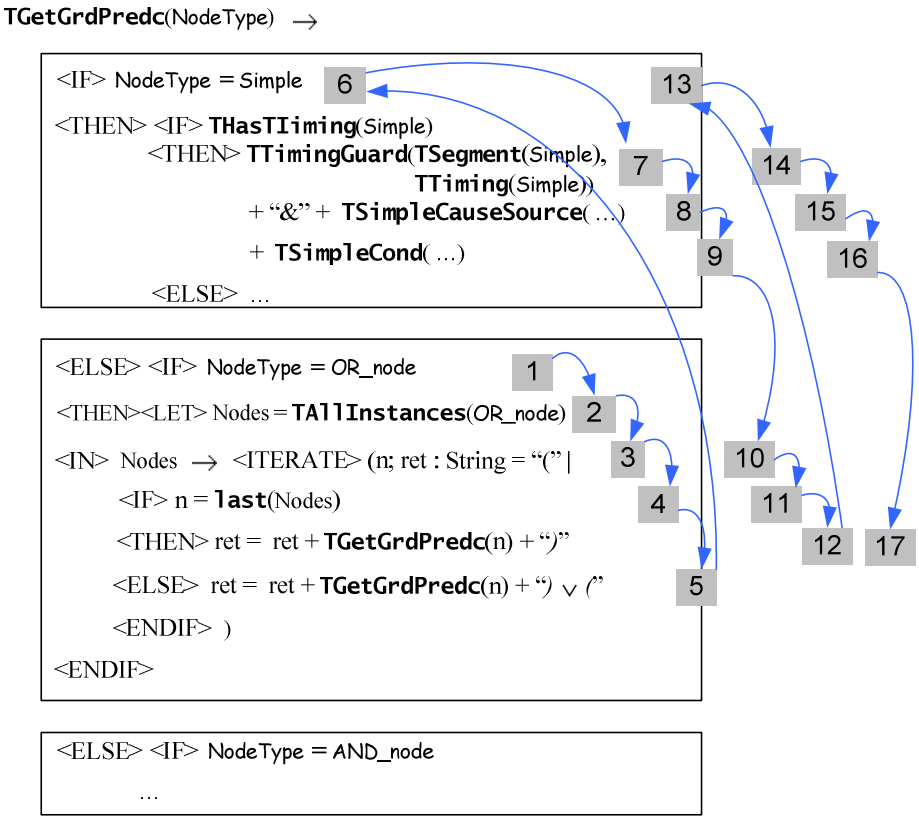


Figure 5-17 An example of a process for creating guards from Figure 5-16

Step 3, each NodeType is used to generate guards, where the initial return value is equal to "*(*". Thus, in this step, the Simple1 is used first

Step 4, the Simple1 is not the last node in the sequence.

Step 5, the Simple1 is used as input parameter for the rule **TGetGrdPredc** itself. The output from the rule is concatenated "*)* $\vee$ *(*"

Step 6-9, since Simple1 is a Simple NodeType, it is used to create guards by sub-rules in steps 7-9. At this point, suppose the steps 7-9 return a group of output guards called *guard_clauses1*.

Step 10, Simple2 is used.

Step 11, Simple2 is the last node in the sequence.

Step 12, Simple2 is used as input parameter for the rule **TGetGrdPredc** itself. The output from the rule is concatenated with "*)*".

Steps 13-16, Since Simple2 is a simple NodeType, it is used to create guards by sub-rules in step 14-16. At this point, suppose the steps 14-16 return a group of output guards called *guard_clauses2*.

Step 17, the return value is *(guard_clauses1)* $\vee$ *(guard_clauses2)*

Within the same process, if the NodeType is AND_node, the return value is in a form of (*guard_clauses1)* $\wedge$ *(guard_clauses2).*

5.3.7 *Creating an Event's guards from Timing constraints*

The rule **TTimingGuard** uses Segment and Timing as input parameters. The rule generates timing constraints as a guard by concatenating an object's name, an object's state, additional strings, and timing constraints.

TTimingGuard(Segment, Timing) →

"*(gclock -* " + **TObj**(**TObjSt**(Segment))

+ **TObjSt**(Segment)) + "*Time* ≥ "

+ **TLowerLmt**(Timing)+"*)*" + "*& (gclock -* "

+ **TObj**(**TObjSt**(Segment)) + **TObjSt**(Segment))

+ "*Time* ≤ " + **TUpperLmt**(Timing) + "*)*"

Figure 5-18 Rule **TTimingGuard**: creating a timing constraint guard

From Figure 5-16, and step 7 in Figure 5-17, when Simple1 is used as an input parameter for the rule **TTimingGuard**, the following output is generated.

TTimingGuard(**TSegment**(Simple1), **TTiming**(Simple1)) =..... step 7

TTimingGuard(MovingDepartingUp2, [2, 5]) →

"*(gclock -* "..... *(gclock -*

+ **TObj**(**TObjSt**(MovingDepartingUp2) ..... *lift*

+ **ObjSt**(MovingDepartingUp2) ..... *MovingDepartingUp*

+ "*Time* ≥ "..... *Time* ≥

+ **TLowerLmt**([2, 5]) +"*)*"..... *2)*

+ "*& (gclock -* " ..... *& (gclock -*

+ **TObj**(**TObjSt**(MovingDepartingUp2)) ..... *lift*

+ **TObjSt**(MovingDepartingUp2)) ..... *MovingDepartingUp*

+ "*Time* ≤ "..... *Time* ≤

+ **TUpperLmt**([2, 5]) + "*)*"..... *5)*

Output: *(gclock - liftMovingDepartingUpTime ≥ 2)*

& (gclock - liftMovingDepartingUpTime ≤ 5)

The output for a Simple2 is generated within the same way,

(gclock - liftMovingDepartingDownTime ≥ 2)

& (gclock - liftMovingDepartingDownTime ≤ 5)

The guards generated from timing constraints (by the rule **TTimingGuard**) are then concatenated with guards created from cause segments (by the rules **TSimpleCauseSource**) and conditions (by the rule **TSimpleCond**). The details of rules **TSimpleCauseSource** and **TSimpleCond** are shown in Appendix A.

Up to this point the `Segment Off2` in Figure 5-16 is used to generate parts of an event as illustrated below:

floorsensorOff = ……TEventName

ANY *f* ……TParamLst

WHERE *f : FLOOR* ……TParamGuardValue

((gclock - liftMovingDepartingUpTime ≥ 2) &
(gclock - liftMovingDepartingUpTime ≤ 5)) & ……TTimingGuard

liftState = MovingDepartingUp ……TSimpleCauseSource

& f = currentFl & dir = Up ……TSimpleCond

∨

((gclock - liftMovingDepartingDownTime ≥ 2) &
(gclock - liftMovingDepartingDownTime ≤ 5)) & ……TTimingGuard

liftState = MovingDepartingDown ……TSimpleCauseSource

& f = currentFl & dir = Down ……TSimpleCond

& (floorsensorState(f) = On) ……TPrevSegm

THEN …

Figure 5-19 Parts of an event *floorsensorOff*

5.3.8 *Creating an Event's actions from an effect segment*

As shown in Figure 5-9, actions for an event are generated from three rules: **TSubst**, **TSimul** and **TRecdTime** which are placed in between ***THEN*** …. ***END*** clause. The rules **TSubst** and **TSimul** are used to generate actions from that segment, and from `SimultaneityArrows` attached to that segment respectively. The rule **TRecdTime** generates an action to record the current time whenever that event is activated.

The detail of the rule **TSubst** is shown in Figure 5-20, where Segment is used as input parameter.

TSubst(Segment) →

<LET> exp = **TObj**(**TObjSt**(Segment)) ..... (1)

<IN> <IF> **THasParam**(**TClass**(exp)) ..... (2)

<THEN> **TObjName**(exp) ..... (3)

+ "*State(* " ..... (4)

+ **TWriteParamLst**(**TAllParam**(**TClass**(exp))) ..... (5)

+ "*) :=* " ..... (6)

+ **TObjSt**(Segment) ..... (7)

<ELSE> exp ..... (8)

+ "*State : =* " ..... (9)

+ **TObjSt**(Segment) ..... (10)

<ENDIF> ..... (11)

TWriteParamLst(Head : SeqTail) →

Head + "↦" + **TWriteParamLst**(SeqTail) ..... (12)

TWriteParamLst(Head : ‹ ›) → Head ..... (13)

Figure 5-20 Rule **TSubst**: creating an Event's action from a Segment

The rule checks whether a class has a parameter, in line (2). If so, lines (3)-(7) are used to generate an action by concatenating an object's name with the string "*State(*", at lines (3)-(4), then followed by a list of parameters which is generated by the sub-rule **TWriteParamLst**. The result is concatenated with the "*) :=* " symbol, at line (6), and object's state at line (7). Where the class has no parameters, lines (8)-(10) are used.

An example of generating an action where the Segment *Off2*, as in Figure 5-16, is used as an input parameter is illustrated below:

TSubst(*Off2*) →

<LET> exp = **TObj**(**TObjSt**(*Off2*))

<IN> <IF> **THasParam**(**TClass**(exp)) ..... *TRUE*

<THEN> **TObjName**(exp) ..... *floorsensor*

+ "*State(* " ..... *State(*

+ **TWriteParamLst**(**TAllParam**(**TClass**(exp))) ..... *f*

+ "*) :=* " ..... *) :=*

+ **TObjSt**(*Off2*) ..... *Off*

<ENDIF>

TWriteParamLst(f : ‹ ›) → f ..... *f*

Output: *floorsensorState(f) := Off*

Suppose a class *floorsensor* has two parameters, e.g. *f : FLOOR* and *a : AA*, the sub-rule **TWriteParamlst** generates *(f↦a).* Thus, an action in this case is defined as *floorsensorState (f ↦ a) := Off*.

5.3.9 *Creating an Event's action from a* `SimultaneityArrow`

This section explains how a `SimultaneityArrow` is used to create an action clause. That is, if a segment has `SimultaneityArrows`, each is used to create an action.

In Figure 5-21, since the segment StopAtFloor1 has a `CauseEffectArrow`, this segment is used to generate an event *liftStopAtFloor* (by the rules explained above). The segment StopAtFloor1 has two `SimultaneityArrow`s a and b. Remember that, the TD BNF definition for a `SimultaneityArrow` is Simul ::= StartSegmt EndSegmt. Thus, the StartSegmt of a and b is the same segment; that is StopAtFloor1, while the EndSegmt of a and b are Deactivated2 and deActivated2 respectively. With the translation rule **TSimul**, the event *liftStopAtFloor* has an action generated by these `SimultaneityArrows`.

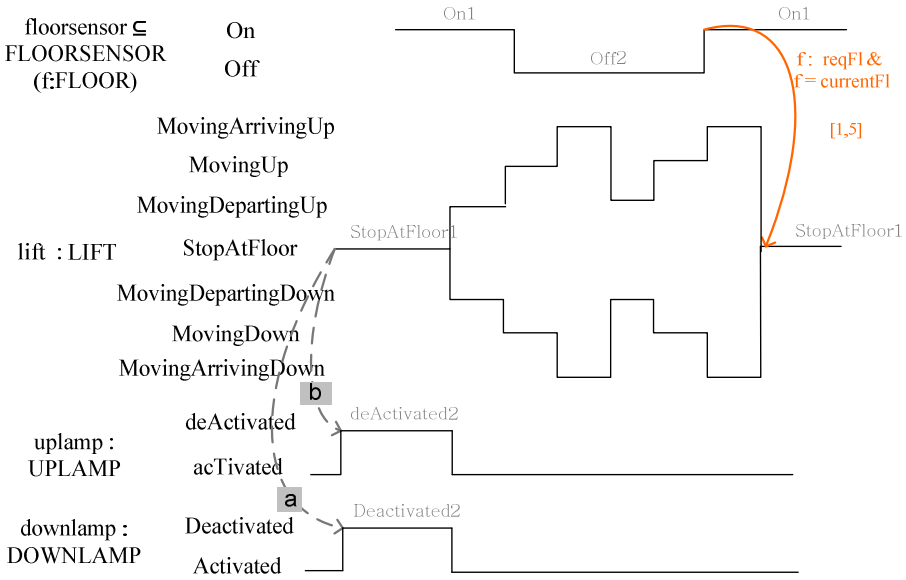


Figure 5-21 Timing diagram shows `Simultaneity` between lift, uplamp and downlamp (parts of Figure 5-2)

The rule **`TSimul`** creates an action from an input `Segment`. The detail of the rule is illustrated in Figure 5-22. Line (2), this rule checks whether there is `SimultaneityArrow` for the segment. If so, the rule iteratively generates an action as shown at line (4) – (19); otherwise it creates nothing as shown at line (21). The detail of the rule is illustrated in the following.

`TSimul`(`Segment`) →

<LET> exp = **`TClass`**(**`TObj`**(**`TObjSt`**(**`TEndSegm`**(`s`)))) ..... (1)

<IN> <IF> **`THasSimul`**(`Segment`) ..... (2)

<THEN> <FOR> `s` <IN> **`TSimulSeq`**(`Segment`) ..... (3)

{<IF> **`THasParam`**(exp) ..... (4)

<THEN> ..... (5)

`TObj`(**`TObjSt`**(**`TEndSegm`**(`s`))) ..... (6)

+ "*State(* " ..... (7)

+ **`TWriteParamLst`**(**`TAllParam`**(exp)) ..... (8)

+ "*) := "* ..... (9)

+ **TObjSt**(**TEndSegm**(*s*)) ..... (10)
<ELSE> ..... (11)
TObj(**TObjSt**(**TEndSegm**(*s*))) ..... (12)
+ "*State :=* " ..... (13)
+ **TObjSt**(**TEndSegm**(*s*)) ..... (14)
<ENDIF> ..... (15)
<IF><NOT> *s* = **last**(**TSimulSeq**(Segment)) ..... (16)
<THEN> "*&*" ..... (17)
<ELSE> <SKIP>. ..... (18)
<ENDIF> ..... (19)
} ..... (20)
<ELSE> <SKIP> ..... (21)
<ENDIF> ..... (22)

Figure 5-22 Rule **TSimul**: creating a substitution

Since there are two SimultaneityArrows a and b attached with the Segment StopAtFloor1 in Figure 5-21, an action is generated by two iteration processes as shown in the following.

TSimul(StopAtFloor1) →
<LET> exp = **TClass**(**TObj**(**TObjSt**(**TEndSegm**(*s*))))
<IN> <IF> **THasSimul**(StopAtFloor1) ..... TRUE
<THEN> <FOR> *s* <IN> **TSimulSeq**(StopAtFloor1) ..... (a, b)
{<IF> **THasParam**(exp) ..... FALSE, when s = a
<ELSE>

1st iteration

TObj(**TObjSt**(**TEndSegm**(a))) ..... *downlamp*
+ "*State :=* " ..... *State :=*
+ **TObjSt**(**TEndSegm**(a)) ..... *Deactivated*

<ENDIF>
<IF><NOT> *s* = **last**(**TSimulSeq**(StopAtFloor1))
<THEN> "*&*" ..... *&*

<ENDIF>

2^{nd} *iteration*

TObj(**TObjSt**(**TEndSegm**(b))) ..... *uplamp*
+ "*State :=* "..... *State :=*
+ **TObjSt**(**TEndSegm**(b))..... *deActivated*

}

Output : *downlampState := Deactivated & uplampState := deActivated*

5.3.10 *Creating an action for recording current time whenever that event is activated*

To record the current time whenever that event is activated, the rule **TRecdTime** is used. This time is used for synchronisation of events. The rule uses a Segment as an input. The detail of the rule is shown below:

TRecdTime(Segment) →
TObj(**TObjSt**(Segment))
+ **TObjSt**(Segment)
+ "*Time := gclock*"

Figure 5-23 Rule **TRecdTime**: creating an action

Thus, an action is generated from the Segment Off2 in Figure 5-16 by the rule **TRecdTime** is *floorsensorOffTime := gclock*.

5.3.11 *Creating an event Ticktok*

An event *Ticktok* is introduced in the model for generating time progression. This event presents ticking of the clock that occurs independently, and the clock is used for synchronisation of events. The *Ticktok* event uses a global variable *gclock* which represents the current time and is advanced by the event. The *gclock* is defined as an integer with initial value 0. We use a discrete time model rather than

a continuous one since it is suitable for ensuring the time is held within fixed limits. Using discrete time is similar to the approach of (Butler and Falampin 2002). The detail of the **TTicktok** rule is shown in the following.

TTicktok → "*Ticktok* = ***BEGIN*** *gclock := gclock + 1* ***END***"

Figure 5-24 Rule **TTicktok**: creating a *Ticktok* event

This rule generates an event *Ticktok* = ***BEGIN*** *gclock := gclock + 1* ***END***

The event *Ticktok* identifies a unit of time progress equal to 1. This unit can broadly be millisecond, second, minute, etc. The lift system case study identifies timing constraints in seconds. Thus, we use a second unit for our model.

To control the accuracy of system timing constraints, it is necessary to "ensure the timing constraints are satisfied by preventing the clock variable (in our case *gclock*) from progressing to a point at which the required properties would be violated" (Butler and Falampin 2002). However, in a real system, time cannot be prevented from progressing and we leave this for the implementation to ensure timing properties are always satisfied in time.

Addition information add into a Ticktok event

To prevent the time from progressing, it is necessary to add stronger guards for the *Ticktok* event. Those guards are derived from each timing constraint that is attached to the `CauseEffectArrows`. For example from Figure 5-2, the `CauseEffectArrows` in the TD involves ten timing constraints called *timing(1)*, *timing(2)*, … for explanation here.

floorlamp Unlit within [2, 4] seconds after *floorsensor Off* ..... *timing(1)*

floorlamp Lit within [2, 4] seconds after *floorsenseor On* ..... *timing(2)*

floorsensor Off within [2, 5] seconds after *lift MovingDepartingUp* ..... *timing(3)*

floorsensor Off within [2, 5] seconds after *lift MovingDepartingDown* *timing(4)*

lift StopAtFloor within [1, 5] seconds after *floorsensor On* ..... *timing(5)*

… etc.

Thus, there are ten guards to be added into the *Ticktok* event. Each guard comprises two parts: pre- and post-conditions in the form of <*pre-condition* ⟹ *post-condition*>.

Ticktok =

WHERE

grd1: . . .

grd2: . . .

grd3: *(liftState = MovingDepartingUp &* .....(1)

floorsensorState(currentFl) = On &.....(2)

((gclock − liftMovingDepartingUpTime) ≥ 2) & .....(3)

((gclock − liftMovingDepartingUpTime) ≤ 5) .....(4)

).....(5)

⟹.....(6)

gclock - liftMovingDepartingUpTime < 5.....(7)

grd4: *(liftState = MovingDepartingDown &*.....(8)

floorsensorState(currentFl) = On &.....(9)

((gclock − liftMovingDepartingDownTime) ≥ 2) & .....(10)

((gclock − liftMovingDepartingDownTime) ≤ 5).....(11)

).....(12)

⟹.....(13)

gclock - liftMovingDepartingDownTime < 5..... (14)

grd5: . . .

. . .

grd10: . . .

THEN *gclock := gclock + 1* ***END***

Figure 5-25 *Ticktok* event's guards (parts of)

Figure 5-25 gives an example to illustrate the detail of how ***grd3*** and ***grd4*** are generated from *timing(3)* and *timing(4)* respectively. The full detail of other guards can be found in Appendix B. The ***grd3*** has pre-conditions as shown in line (1) to

(4). The pre-conditions are similar to those defined by the rules **`TSimpleCauseSource`**, **`TPrevSegm`** and **`TTimingGuard`** in Figure 5-19 respectively. However, in line (2), the local variable *f* is replaced by the machine variable *currentFl*. Thus, there are no non-deterministic variables defined by the *Ticktok* event. In ***grd4***, lines (8)-(12), are also similar to those defined in Figure 5-19. Thus, identifying the Ticktok event's guards is a process of re-defining cause segments, previous segments, and timing constraints. Notice that in other events, those cause segments, previous segments, and timing constraints are combined within the same guard, as in the example shown in Figure 5-19, while in the *Ticktok* event they are separated, as seen in ***grd3*** and ***grd4***. This is the reason to simplify POs and make it easier to identify *Ticktok*'s guards' post-conditions.

A *Ticktok* guard post-condition is defined by the pattern below:

"*gclock* - " + Obj + CauseSegmt + "*Time*" + " < " + upperlimit

For example, the post-condition for **grd3** as defined at line (7) is

gclock - liftMovingDepartingUpTime < 5

This means the clock is allowed to progress between an upper and lower bound until time expires. For example, from the **grd3,** a *floorsensor* is being set to *Off* between 2 and 5 seconds after the *lift* is in the state of *MovingDepartingUp*.

5.4 User manual input on modelling

Since the translation rules create events from `Segments` that have constraints (have a `CauseEffectArrow`), there are some events that have to be manually added into the Event-B machine. That is because not every changing state in the TD is identified by the `CauseEffectArrows`. For example, the changing states of the door from *Open* to *Close*, and changing state of the lift from *MovingArrivingUp* to *MovingDepartingUp* or *MovingArrivingDown* to *MovingDepartingDown*.

Moreover, TD are not designed to keep the whole information of the system. Thus, there is missing information which may not be identified in the specification (and that is why it is not generated as a TD) from the beginning, or information that cannot be identified as a TD because it is not supported by TD notations. For example, the lift changing directions from up to down or from down to up needs to be created manually since it is not identified in the specification, but it needs to be included in the system.

Currently, a `SimultaneityArrow` is not designed to have a combination of OR nodes. Thus, if there is a `SimultaneityArrow` that is used to indicate this kind of relationship, the output model has to be altered manually. Thus, in Figure 5-26, a and b are `SimultaneityArrows` that demonstrate whenever a *floorsensor* is set *Off*, the *lift* is in a state of *MovingUp* or *MovingDown* instantly. The whole *floorsensorOff* event is generated by translation rules shown in Figure 5-27.

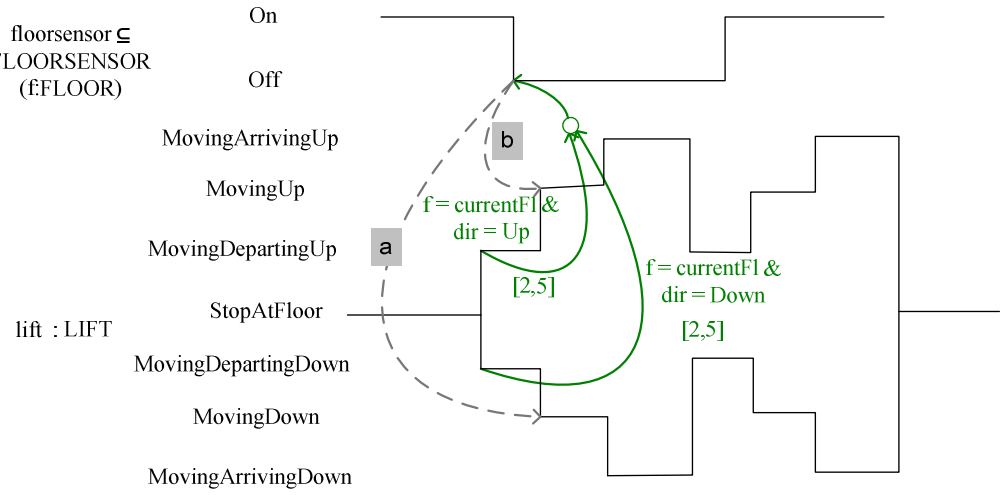


Figure 5-26 `SimultaneityArrow` for the lift object

(Parts of Figure 5-2)

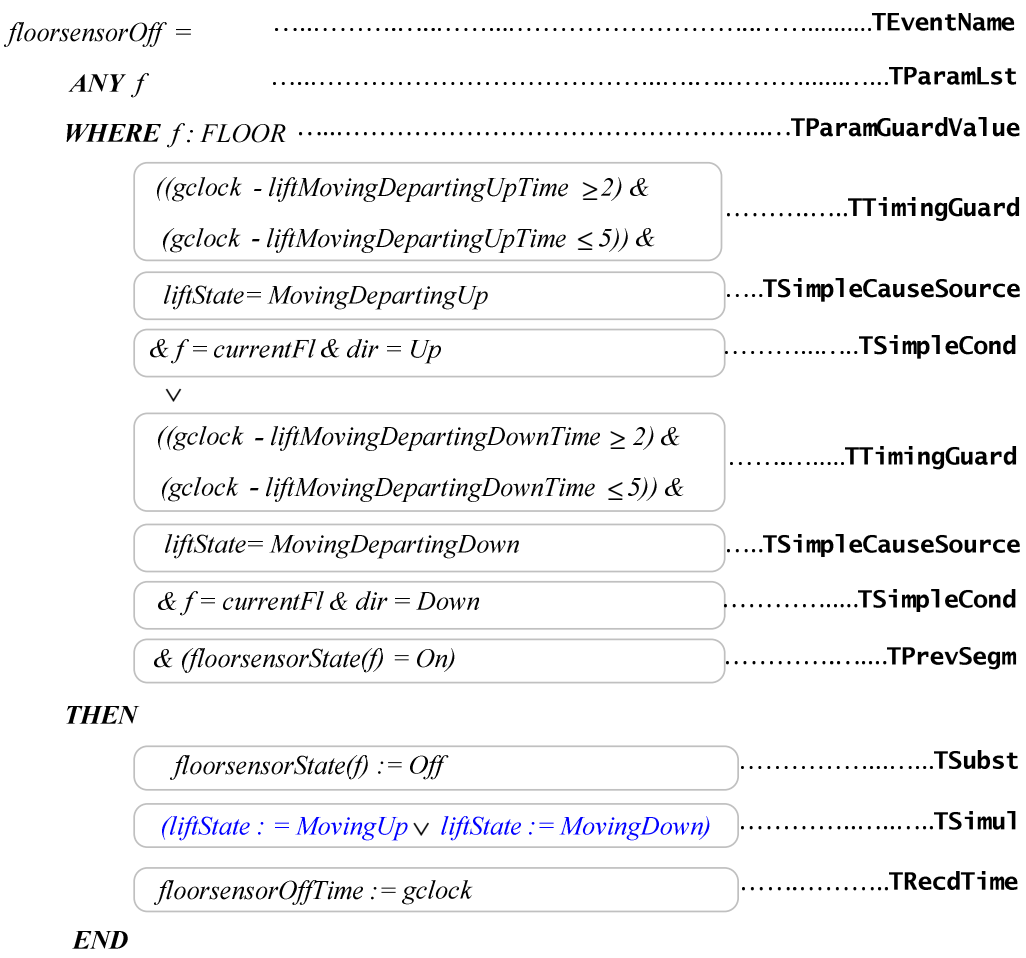


Figure 5-27 A *floorsensorOff* event before revision

In Figure 5-27, the action generated by the rule **TSimul** is not recognized by Event-B complier. That is because Event-B does not deal with OR relationships in an action part. Thus, we have to revise the *floorsensorOff* event by separating it into two events: *floorsensorOffUp* and *floorsensorOffDown* as shown in Figure 5-28. In order to do that, we also split the original *floorsensorOff* event's guards and actions into the corresponding events.

floorsensorOff Up =

ANY *f*

WHERE *f* : *FLOOR*

((*gclock* - *liftMovingDepartingUpTime* ≥ 2) &

(*gclock* - *liftMovingDepartingUpTime* ≤ 5)) &

liftState = *MovingDepartingUp*

& *f* = *currentFl* & *dir* = *Up*

& (*floorsensorState*(*f*) = *On*)

THEN

floorsensorState(*f*) := *Off*

liftState := *MovingUp*

floorsensorOffTime := *gclock*

END

floorsensorOff Down =

ANY *f*

WHERE *f* : *FLOOR*

((*gclock* - *liftMovingDepartingDownTime* ≥ 2) &

(*gclock* - *liftMovingDepartingDownTime* ≤ 5)) &

liftState = *MovingDepartingDown*

& *f* = *currentFl* & *dir* = *Down*

& (*floorsensorState*(*f*) = *On*)

THEN

floorsensorState(*f*) := *Off*

liftState := *MovingDown*

floorsensorOffTime := *gclock*

END

Figure 5-28 Two new events are regenerated from *floorsensorOff* event

5.5 Summary

This chapter explains how translation rules are used to transform a TD to an Event-B model. First, we generate BNF definitions for describing a TD. Next, translation rules are created in which TD BNF elements are used as input parameter for the rules. The rule covers generating the Event-B **CONTEXT** and the **MACHINE** parts.

For the **CONTEXT** part, we can generate sets, constants, and axioms. Additional sets that cannot be identified by TD need to be added by hand; for example, identifying a set of lift directions to up and down.

For the **MACHINE** part, the rules can generate machine variables, invariants, variables' initial values, and events. Normally, if an extra set is generated by hand in the context part, the additional machine variables, invariants and their initial values corresponding to that set are generated by hand in the **MACHINE** part. Some other machine variables may also be identified. For example, in the lift case study, the machine variable *currentFl* is manually added to represent the current floor of the lift. In the **MACHINE** part, each event is generated by a segment that has a `CauseEffectArrow`. The rules can generate an event's name and its body in one of two types: *Guard* or *Non deterministic*. The first type does not have local variables, while the latter is declared with non-deterministic local variables. An event's guards are generated from timing constraints, Cause segments, Previous segments, and conditions attached to the `CauseEffectArrow`. An event's actions are generated from a target state and `SimultaneityArrows`. Each event is provided with an action to record the time it is activated. This time is used to synchronise events. Currently, TD notation does not support identifying `SimultaneityArrow` with OR nodes, thus any action created by this kind of node needs to be split into corresponding events.

There are some events that need to be added by hand. That is because not every event can be identifyied by a `CauseEffectArrow`. For example, changing state in an object itself, such as an event to represent the state of a door changing

from open to close, needs to be generated by hand. This alteration depends on the characteristics of each system.

To control timing of events, we create the *Ticktok* event for time progression and for preventing the clock variable progressing to a point at which system properties will be violated.

Chapter 6 Translating Timing diagrams into UML-B

The use of TD is suitable for identifying timing constraints in an object itself and among other objects. However, a TD is not designed to add state-based information nor gather whole system information. Thus, to create a complete Event-B model from a TD, the missing information must be added, such as variables, constants and some events. In order for that process to be accomplished, either the information must be added by hand or an existing tool like UML-B must be used. A UML-B is a plug-in for RODIN toolkits and is implemented by Eclipse EMF. UML-B is an Event-B graphical front end, has a well-defined Metamodel of Classes, and Statemachines, and can be automatically translated into an Event-B model whenever the model is saved. The Event-B verification tools, i.e. syntax checker and Prover, then run and immediately display any problems which are shown in the relevant UML-B diagrams. Thus, we selected the UML-B plug-in as it provides Event-B integration and its features – using Class and Statemachine – are TD compatible. For example, it enables us to compare state changes in the TD along a `Timeline` using the UML-B Statemachine. ATL, which is also developed on the Eclipse platform and generates a target model from a source model, was selected for translation rules. ATL like UML-B also has a well-defined Metamodel. Section 6.1 identifies TD used for translation into UML-B. Section 6.2 gives an overview of how a TD source model is transformed into a target UML-B model, using ATL translation rules. The abstract syntax of a TD is identified by a TD

metamodel in section 6.3, and is used to generate a TD input model as described in section 6.4. ATL translation rules for creating each UML-B model component are explained in section 6.5. TD cannot be used to create a complete UML-B output model because a TD in itself only represents some parts of the whole specification. Thus, some additional information is required for the model, as explained in section 6.6.

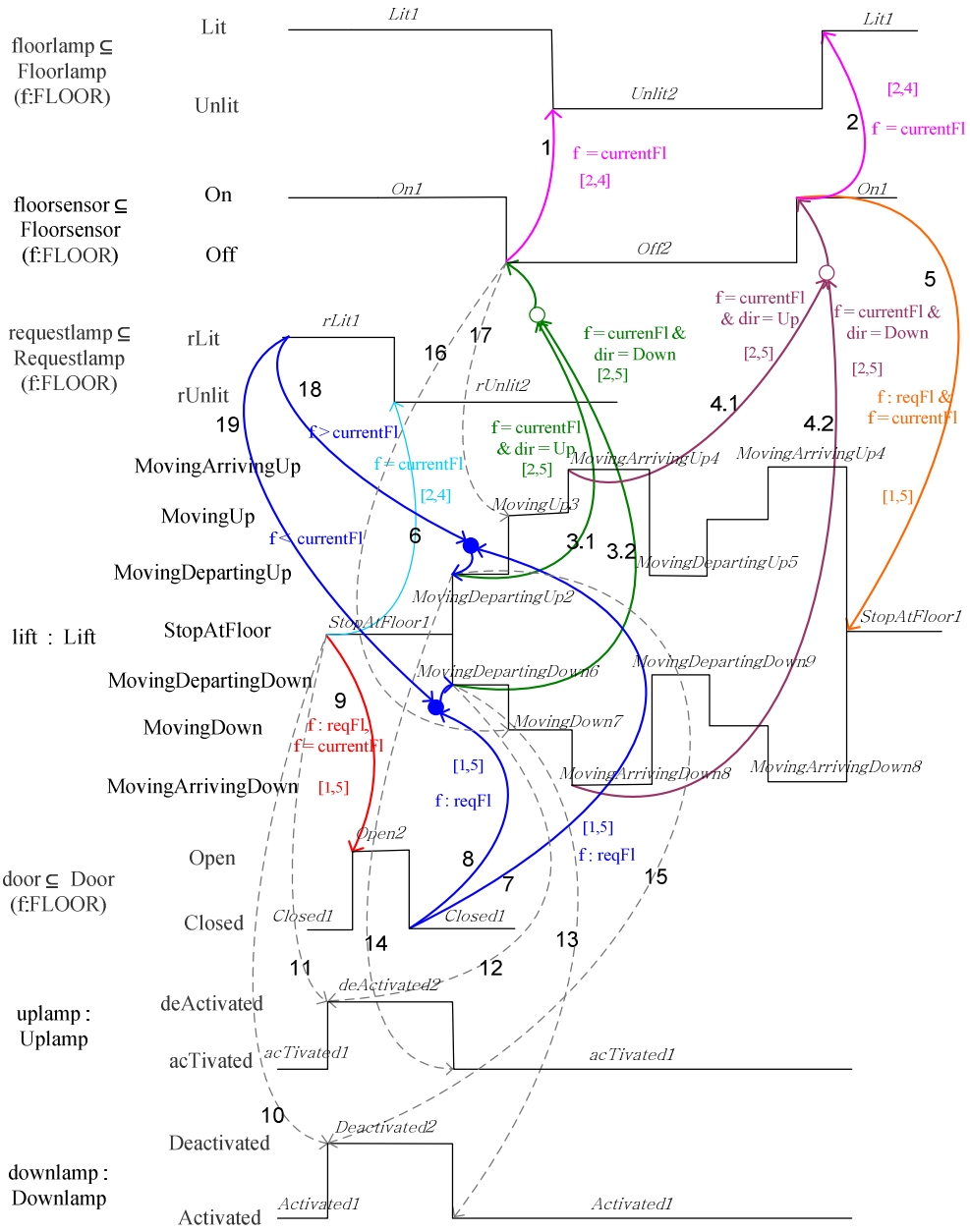


Figure 6-1 Timing Diagram used for transforming into a UML-B model

6.1 Timing Diagram used for translation into UML-B

A TD used for generating UML-B is slightly different from the TD used for the direct translation of an Event-B model shown in Chapter 5.

Unlike the TD for the direct translation, where a class name is represented by capital letters, in TD translation to a UML-B model, a class name must begin with a capital letter followed with small letters.

Thus, in chapter 5, floorsensor $\subseteq$ FLOORSENSOR(f:FLOOR)

in chapter 6, floorsensor $\subseteq$ Floorsensor(f:FLOOR)

For the direct translation, class names are generated as a set in a **CONTEXT** part. For the UML-B translation, class names are generated as a class in a **MACHINE** part.

6.2 Overview of the TD to UML-B ATL transformation

We use ATL as a language to transform a TD model into a UML-B model. Figure 6-2 shows a source model Timing diagram (TD), which conforms to a metamodel TDMetamodel, transformed into a target model UML-B which conforms to a metamodel umlbMetamodel. The transformation definition TDtoUMLB.atl is written in ATL language and conforms to a metamodel ATL. The whole metamodel conform to the Ecore metamodel.

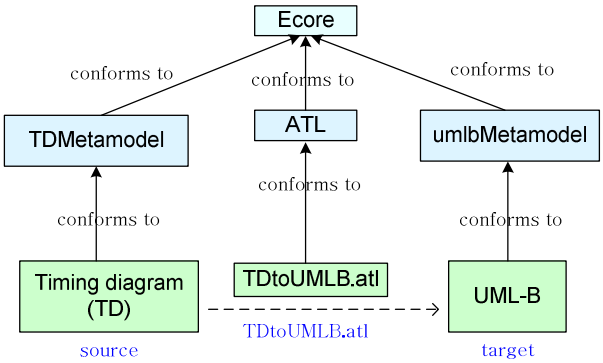


Figure 6-2 Overview of the TD to UML-B ATL transformation

6.3 Timing diagram Metamodel

The TD metamodel created by EMF to describe abstract syntax of TD is illustrated in Figure 6-3. The same colours within Figure 2-16 are used to identify which parts of the TD metamodel are generated into UML-B metamodel parts. A TD model is initially generated inside a project (**TDProject**) with a string name (**Name**) provided. A project is made up of one or more TD machines (**TDMachine**). A **TDMachine** contains at least a TD class (**TDClass**). Each machine and class is given a name. A class may or may not have parameters. If there is a parameter (**TDParameter**), the parameter is defined by a string name (**param**) and type (**paramType**). A class has zero or many `Timelines` (**TDTimeline**). Each `Timeline` has at least one state (**TDState**), and zero or many transitions (**TDTimelineTransition**).

Each TD state may have zero or many segments (**TDSegment**), in which each segment is identified by its incoming (**incoming**) and outgoing (**outgoing**) transitions. Each transition connects to a couple of segments: a source (**source**) and a target (**target**) segment. A segment may or may not have a `SimultaneityArrow` (**Simul**). If so, it connects two segments. At present, the TD metamodel allows developers to generate a `SimultaneityArrow` within the same segment. However, we must assume that to correctly generate a TD model, one has to know that a `SimultaneityArrow` links different object segments.

A segment has zero or more constraints (**TDConstraints**) in which each constraint has one node type (**TDNodeType**). Why do we need a class **TDConstraints** instead of having a direct association between **TDSegment** and **TDNodeType**? The reason is to maintain the **TDConstraints** class. Without this class ATL cannot generate a UML-B model correctly. We may need to explore the reason in future work; however, we think that it could be a problem with ATL itself or the ordering of translation rules.

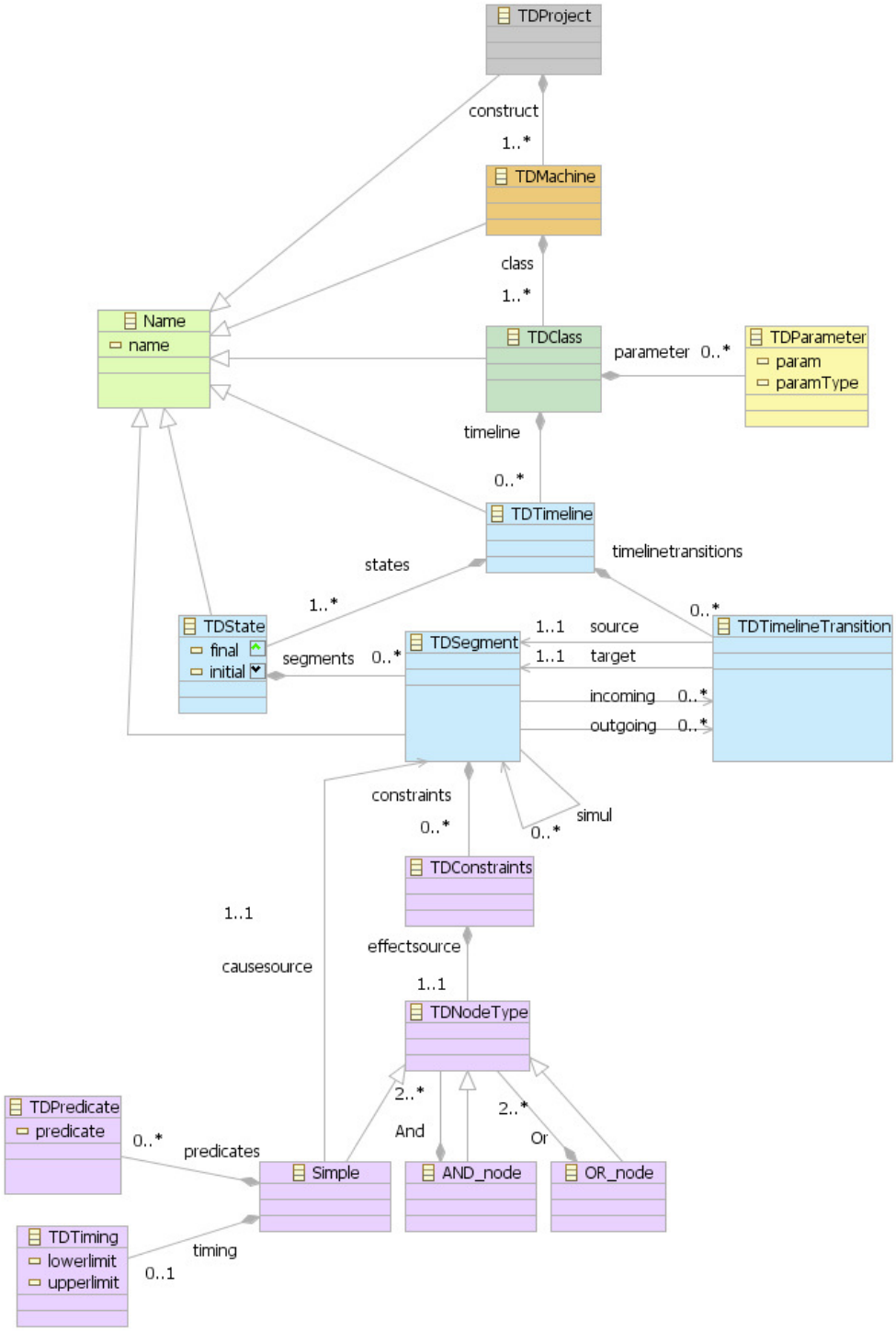


Figure 6-3 Timing diagram Metamodel

There are three kinds of node type: *Simple* node (**Simple**), *And* node (**AND_node**), and *Or* node (**OR_node**). *And* and *Or* nodes require at least two

node types; they can be *And* or *Or* nodes themselves or *Simple* node types. A *Simple* node type is actually a segment and is used to identify a cause segment (**causesource**) for a `CauseEffectArrow`. Each *Simple* node could have zero or more conditions (**TDPredicate**), with each condition identified by a string. In addition, a *Simple* node may have at most one timing constraint (**TDTiming**). A timing constraint is declared by lower bound **(lowerlimit**) and upper bound (**upperlimit**) whose values are integers.

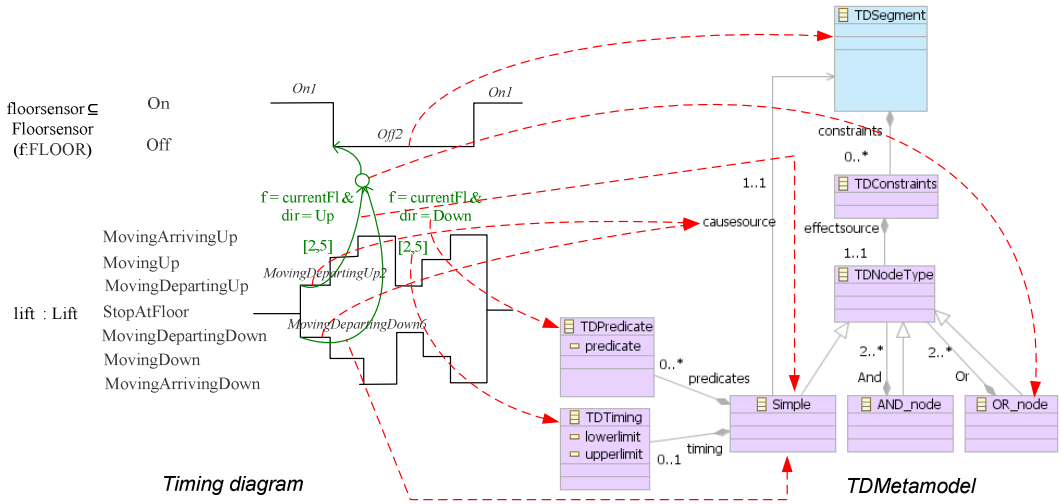


Figure 6-4 An example TD vs. TDMetamodel

For example in Figure 6-4, a segment *Off2* has a constraint defined by a node type *OR*. This node type comprises two simple node types pointing to segments *MovingDepartingUp2* and *MovingDepartingDown6*. The simple node type *MovingDepartingUp2* has predicates and a timing constraint defined by *f* = *currentFl* & *dir* = *up*, and [*2, 5*] respectively. In the same manner, the simple node type *MovingDepartingDown6* has predicates and a timing constraint defined by *f* = *currentFl* & *dir* = *down*, and [*2, 5*] respectively.

6.4 Generating a TD input model

A TD model is generated from TD metamodel using Eclipse EMF. Figure 6-5 shows parts of a screenshot of an Eclipse EMF editor view for a lift system.

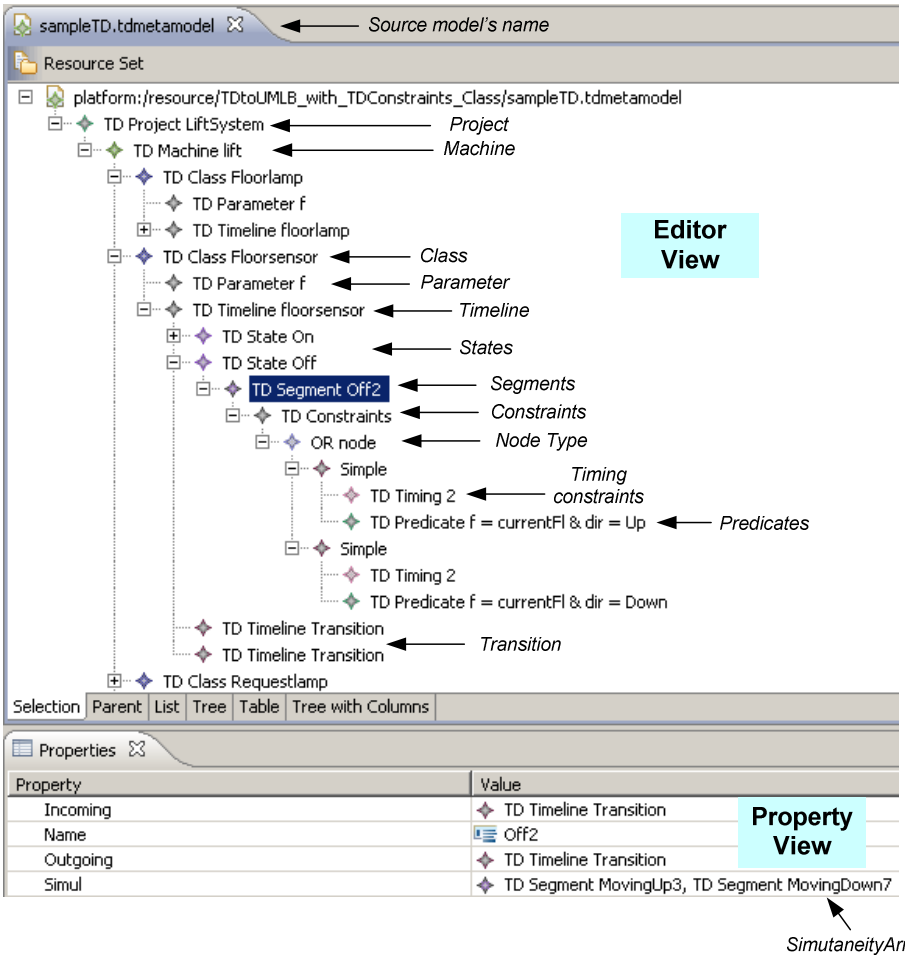


Figure 6-5 Timing diagram instance generated by Eclipse EMF

The editor view is located at the top of the window while the Properties tab is positioned at the bottom. The figure shows a TD machine named lift located inside a LiftSystem project. Each class is declared inside the machine, together with any existing Parameters, Timelines, States, Segments, Nodetypes, Timing constranints, Predicates and Transitions. For example the highlighted segment in Figure 6-5 indicates a segment Off2. This segment belongs to a class Floorsensor. This class has

a parameter f, a Timeline named floorsensor, and comprises two states: On and Off. Each state is defined by its segment, for example, a segment Off2 belongs to the state Off. This segment has a constraint defined by an OR node with a combination of two Simple NodeTypes represented by line 3.1 and 3.2 in Figure 6-1. Each Simple NodeType has Timing and Predicates as shown in Figure 6-5. Incoming, Outgoing and Simul are defined by the Properties tab as shown at the bottom of the figure. In Figure 6-1, the segment Off2 has two `SimultaneityArrows`: MovingUp3 and MovingDown7 as shown in the Properties tab. Since the TD Timeline transitions do not have name, we do not declare a name for `Timeline Transitions` in the metamodel. Thus, we have to carefully select the corresponding transitions. Giving `Timeline Transitions` names is considered as future work.

6.5 ATL Translation rules

This section describes details of ATL translation rules used to transform a TD into a UML-B diagram. Figure 6-6 shows an ATL header section named `TDtoUMLB` which use a target and a source model conforming to `umlbMetamodel` and `TDMetamodel` respectively. They are some helpers defined at the beginning of the ATL module such as `umlbproject` and `nat1Type` (the details of ATL helpers are described in section 2.9.3). These helpers will be used in the rule `Project` as shown in Figure 6-10 to append the corresponding values to a target UML-B model. For example, the helper `umlbproject` is used to add a project that is created from a TD to a UML-B Project. The helper `nat1Type` is used to add a positive number to a UML-B TypeExpression. The details of using `umlbproject` and `nat1Type` helpers are explained in section 6.5.2 below while the other helpers are detailed in Appendix C.

```
module TDtoUMLB;
create OUT : umlbMetamodel from IN : TDMetamodel;
helper def : umlbproject : umlbMetamodel!UMLBProject =
                        umlbMetamodel!UMLBProject;
helper def : nat1Type : umlbMetamodel!UMLBTypeExpression =
                        umlbMetamodel!UMLBTypeExpression;
```

Figure 6-6 Header section of TDtoUMLB.atl

Figure 6-7 illustrates parts of a UML-B metamodel in which the same colours used in Figure 6-3 are used to emphasize corresponding TD to UML-B parts used during the conversion.

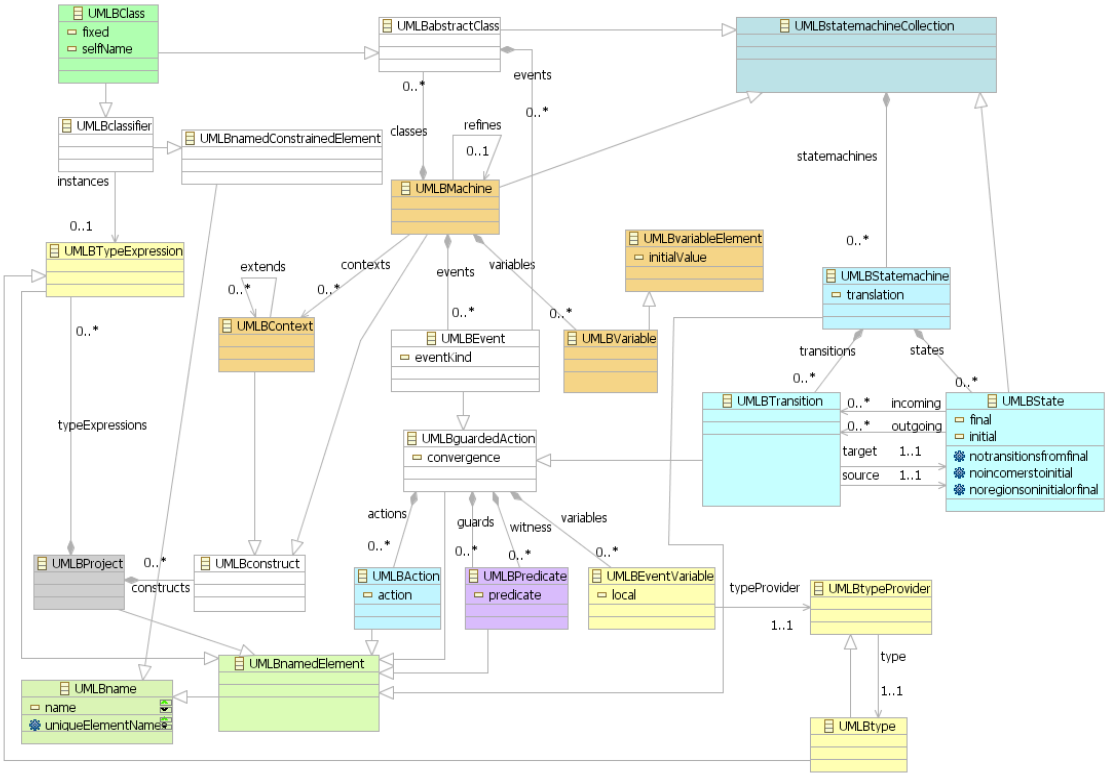


Figure 6-7 UML-B Metamodel (parts of)

There are a number of UML-B parts which can be directly generated from TD components, e.g. *Project*, *Machine* and *Class*. However, some of UML-B components cannot be directly created. For example, generating a guard for a

UML-B event, many of TD metamodel classes are involved, such as **TDConstraints**, **TDNodeType**, **Simple, AND_node, OR_node, TDPredicate** and **TDTiming**. The detail of creating a UML-B model is described next.

6.5.1 *Top-Level ATL translation rules*

This section explains the structure of the top-level ATL rules and the corresponding UML-B model components created. As shown in Figure 6-8, an UML-B project's name is created from the rule `Project`, while a machine is generated from the rule `Machine`. The rule `Machine` is also used for creating a machine event *Ticktok* and a machine variable *gclock*, which are used to generate time progress, and the global clock for the machine respectively. Extra machine variables are added such as *reqFl* to keep the list of requested floors (this is the same variable created by hand in Chapter 5). A *SEES* association and a context's name are created from the `Machine` rule. However, the context detail has to be declared manually. This is because ATL has a limitation and cannot re-use elements to generate other new elements across rules. ATL does not have the flexibility to generate an element that has to be created from the combination of used target elements. Thus, we cannot use TD class names to generate carrier sets in a *Context*, since they are already used to create classes by the rule `Class` (as described in section 6.5.4 below).

UML-B class names and attributes are created from the rule `Class`. Some attributes need to be redefined since part of an attribute's name is generated from its corresponding state's name. Statemachines belonging to corresponding classes are generated by the rule `Statemachine`.

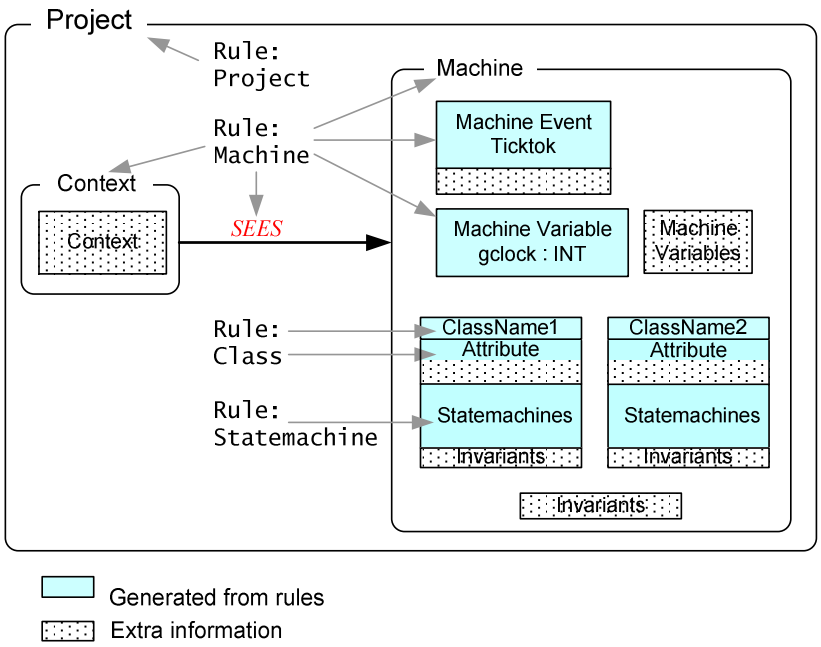


Figure 6-8 Top-level ATL rules

Our translation rules do not cover defining UML-B Machine Statemachines. This is because a TD Timeline, which can be seen as a UML-B Class Statemachine must belong to a class. According to our TD metamodel, one cannot generate a `Timeline` without a class. Invariants have to be manually created since they can not be declared by TD.

6.5.2 *Creating UML-B Project*

An UML-B project is generated by mapping a class **TDProject** to a class **UMLBProject** (Figure 6-9) by the rule `Project` (Figure 6-10) as detailed in line (2) and (3) where a variable `u` is used to represent a target model element, Project. From Figure 6-10, the rule `Project` maps the source model element `TDMetamodel!TDProject` represented by a variable `t` in line (2), to a target model element `umlbMetamodel!UMLBProject` represented by a variable `u` in line (3). UML-B project's name is created from a TD project's name as shown in line (4).

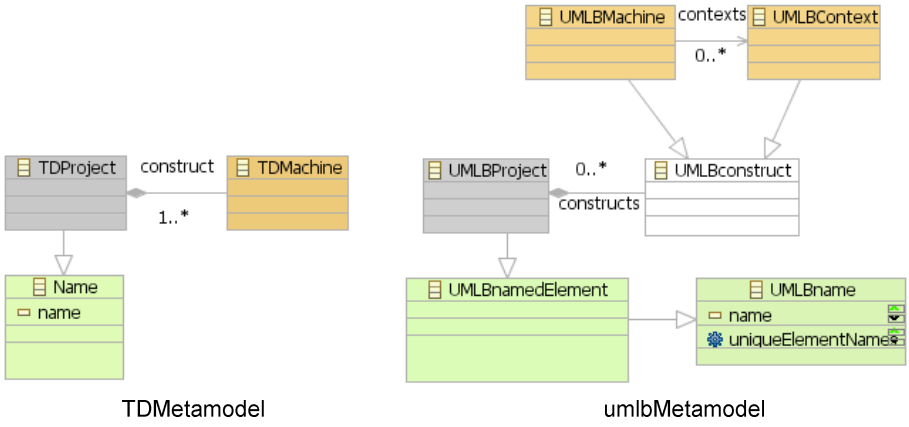


Figure 6-9 TDMetamodel and umlbMetamodel : Project and Machine

As shown in Figure 6-9 right, **UMLBProject** comprises **UMLBconstruct** which is sub typed into **UMLBMachine** and **UMLBContext**. Thus, line (5) maps an association `construct` of TDMetamodel to an association `constructs` of umlbMetamodel. This association maps **UMLBMachine** and **UMLBContext** (which are both created later by the rule `Machine`, Figure 6-12) into **UMLBProject** automatically.

```
rule Project {.....(1)
   from t : TDMetamodel!TDProject.....(2)
   to u : umlbMetamodel!UMLBProject.....(3)
            (name <- t.name,.....(4)
            constructs <- t.construct),.....(5)
        pt1 : umlbMetamodel!UMLBTypeExpression.....(6)
            (name <- 'BOOL'),.....(7)
        pt2 : umlbMetamodel!UMLBTypeExpression.....(8)
            (name <- 'NAT'),.....(9)
        pt3 : umlbMetamodel!UMLBTypeExpression.....(10)
            (name <- 'NAT1'),.....(11)
        pt4 : umlbMetamodel!UMLBTypeExpression.....(12)
            (name <- 'INT').....(13)
do { thisModule.umlbproject <- u; .....(14)
```

```
        thisModule.boolType <- pt1; ..... (15)
        thisModule.natType <- pt2; ..... (16)
        thisModule.nat1Type <- pt3; ..... (17)
        thisModule.intType <- pt4; ..... (18)
        u.typeExpressions <- u.typeExpressions.append(pt1); ..... (19)
        u.typeExpressions <- u.typeExpressions.append(pt2); ..... (20)
        u.typeExpressions <- u.typeExpressions.append(pt3); ..... (21)
        u.typeExpressions <- u.typeExpressions.append(pt4);} ... (22)
}
```

Figure 6-10 ATL rules for creating UML-B Project

The texts, such as `BOOL` and `NAT1`, inside the `' '` symbol as shown above are additional information. We use them to create variable types such as Boolean (`BOOL`), positive integer (`NAT1`), etc., for use in the model. If we do not create those types in advance, the user must define them manually later. Moreover, since our model defines a timing constraint as an integer, generating a type `INT` also supports this. This way one can introduce strings or variable types directly to the UML-B model.

Lines (6)-(13) show assigning `BOOL`, `NAT`, `NAT1` and `INT` to each target model element **UMLBTypeExpression** which is represented by variables `pt1`, `pt2`, `pt3` and `pt4` respectively. Those variables are assigned to a corresponding helper in lines (15)-(18), in a **do** part in which a command `<thisModule.helperName>` is used for inferring a helper. Note that using **to** and **do** is described in section 2.9.2. Lines (19)-(22) are used to append those variables to the project.

6.5.3 *Creating a UML-B Context's name and Machine*

A UML-B context's name and machine are created by the rule `Machine` as shown in Figure 6-12. This rule uses the source model element `TDMetamodel!TDMachine` as shown in line (2). As shown in lines (4)-(5), a context's name, represented by the variable `ctx`, is created by the TD machine name followed by the string `_ctx`. The texts such as `_ctx`, `Ticktok` and `gclock :=`

`gclock + 1`, as shown in Figure 6-12, are additional information which we use to directly generate UML-B parts that cannot be obtained by the TD. In this case, they are used to introduce a string, an event name, and an event action. Line (7) is the generation of the UML-B machine name by the TD machine name. Line (8), an association `class` in TDMetamodel is mapped to an association `classes` in umlbMetamodel; this is for adding classes (that are created later in rule `Class`) to the machine.

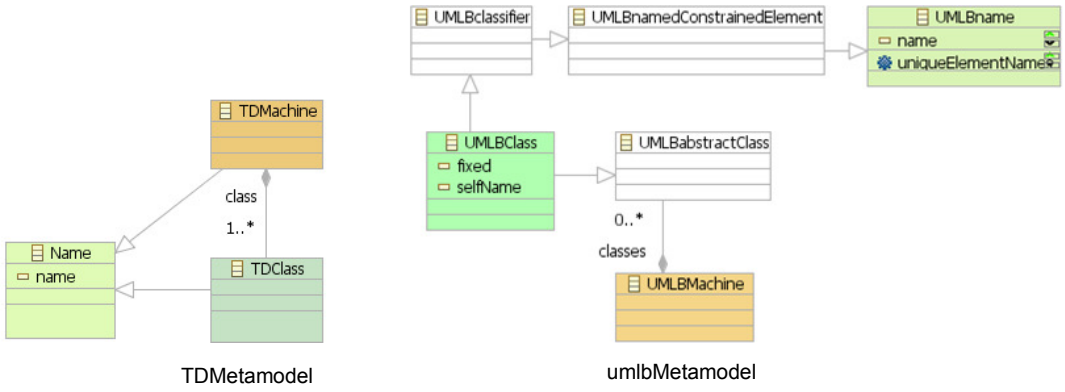


Figure 6-11 TDMetamodel and umlbMetamodel : Machine and Class

```
rule Machine {                                                    (1)
    from t : TDMetamodel!TDMachine                                (2)
    to                                                            (3)
      ctx : umlbMetamodel!UMLBContext                             (4)
          (name <- t.name + '_ctx'),                              (5)
      m : umlbMetamodel!UMLBMachine                               (6)
          (name <- t.name,                                        (7)
          classes <- t.class),                                    (8)
      e : umlbMetamodel!UMLBEvent                                 (9)
          (name <- 'Ticktok'),                                    (10)
      a : umlbMetamodel!UMLBAction                                (11)
          (name <- 'Action1',                                     (12)
          action <- 'gclock := gclock + 1'),                      (13)
      gclk : umlbMetamodel!UMLBVariable                           (14)
          (name <- 'gclock',                                      (15)
          typeProvider <- thisModule.intType,                     (16)
```

```
                initialValue <- '0')..........(17)
    do {..........(18)
        m.events <- m.events.append(e);..........(19)
        e.actions <- e.actions.append(a);..........(20)
        m.variables <- m.variables.append(gclk);..........(21)
        thisModule.umlbmachine <- m;..........(22)
        m.contexts <- m.contexts.append(ctx);..........(23)
        thisModule.umlbproject.constructs <-..........(24)
            thisModule.umlbproject.constructs.append(ctx);
        thisModule.umlbproject.constructs <-..........(25)
            thisModule.umlbproject.constructs.append(m); }
}
```

Figure 6-12 ATL rules for creating UML-B Machine

An event *Ticktok*, represented by the variable `e` of the target model element `umlbMetamodel!UMLBEvent`, is created in lines (9)-(10). A `Ticktok` action is assigned to `gclock := gclock + 1` as shown in lines (11)-(13) while the machine variable *gclock* whose type is assigned to an integer with an initial value of 0, is generated as shown in lines (14)-(17). The variables `ctx`, `m`, `e`, `a` and `gclk` are assigned to corresponding UML-B components by the **`do`** section.

In the **`do`** section, line (19) is used to add the event `Ticktok` to **UMLBMachine**, line (20) appends the action to the event `Ticktok`. Line (21) assigns the variable `gclk` as a machine variable, and then the machine is added to **UMLBMachine** by calling the helper `thisModule.umlbmachine`, shown in line (22). Line (23) links the context to the machine by adding this context to an association, `contexts`. Line (24) appends this context to a project by calling the helper `thisModule.umlbproject.constructs`. The helper `thisModule.umlbproject` is defined earlier (Figure 6-6) and `constructs` is an association name as illustrated in Figure 6-9. Line (25) appending the machine to the project. The rule `Machine` generates a package diagram and the event `Ticktok` in the machine part as illustrated below.

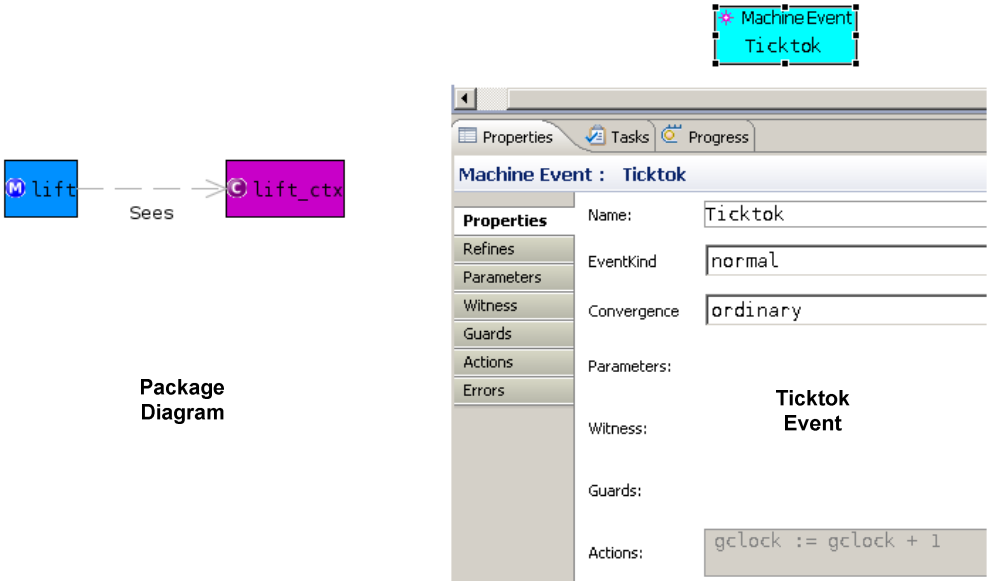


Figure 6-13 Package Diagrams and Event Ticktok in a Machine part

6.5.4 *Creating UML-B Class and local attributes*

Figure 6-14 shows part of the corresponding TD, **TDMetamodel** and **umlbMetamodel** used for generating UML-B classes and attributes. UML-B classes and attributes are created by the rule `Class` as shown in Figure 6-15. A class name is generated by a **TDClass** name followed with a string `Self` as shown in lines (4)-(5). The `Self` is used to identify a unique non-deterministic variable name for the class. For example, *FloorsensorSelf* is a non-deterministic variable used in the class *Floorsensor*. Line (6) shows the mapping of the TD `timeline` association to the UMLB `statemachines` association. This is how we link Statemachines to a class.

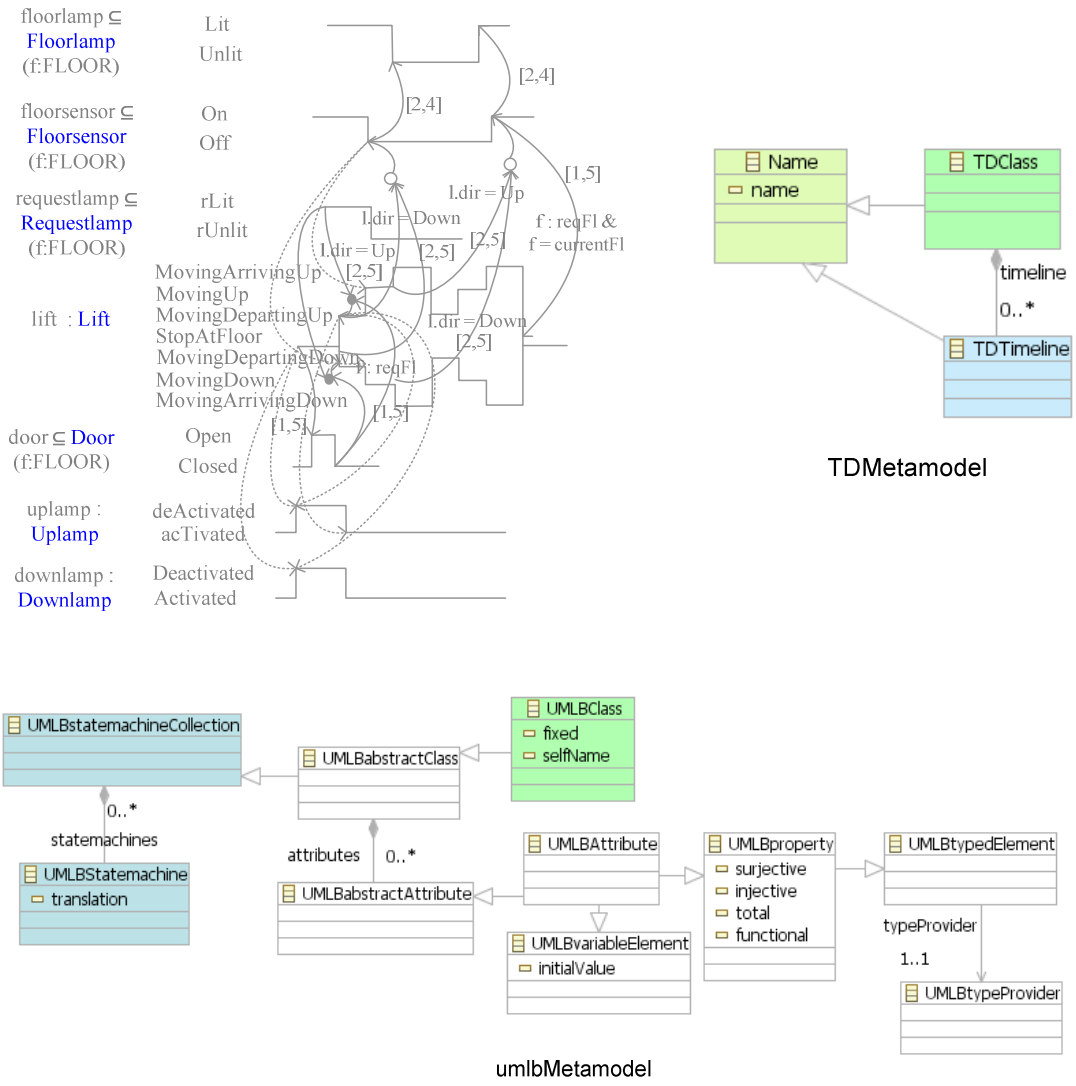


Figure 6-14 TDMetamodel and umlbMetamodel : Class and Attribute

A class attribute is generated by mapping the **TDClass** to the **UMLBAttribute**, where the result is kept in variable `att`, as shown in line (7). An attribute name is generated by the **TDClass** name which is changed to lower case letters by the function `toLower()`, followed by the string `xStatexTime`, line (8). We use a string `xStatexTime` to represent features that need to be completed by hand. In this case, it is a part of a class attribute name. Every class attribute name is generated from every corresponding state name of a class. For example, the class *Floorsensor* must have attributes *floorsensorOffTime* and *floorsensorOnTime*. We

cannot generate whole attributes because this rule is working with **UMLBClass**, line (2)**,** not a **UMLBState**. The **UMLBState** is used for generating a Statemachine in the rule `StateMachine` as shown in Figure 6-17 below. As ATL cannot reuse the same elements to generate other components across the rules, we cannot generate whole attributes for this class.

```
rule Class {.....(1)
        from t : TDMetamodel!TDClass .....(2)
        to u : umlbMetamodel!UMLBClass .....(3)
                (name <- t.name, .....(4)
                selfName <- t.name + 'Self', .....(5)
                statemachines <- t.timeline), .....(6)
        att : umlbMetamodel!UMLBAttribute .....(7)
                (name <- t.name.toLower()+ 'xStatexTime', .....(8)
                typeProvider <- thisModule.intType, .....(9)
                initialValue <- '0') .....(10)
do { u.attributes <- u.attributes.append(att); } .....(11)
}
```

Figure 6-15 ATL rules for creating UML-B Class

Lines (9)-(10) show how to assign an attribute type and initial value which are integer and 0 respectively. The attribute is appended to **UMLBClass** as shown in line (11). Those attributes are used to record the current time whenever corresponding events belonging to the class are activated. Figure 6-16 shows how classes and their attributes are generated from the rule `Class`.

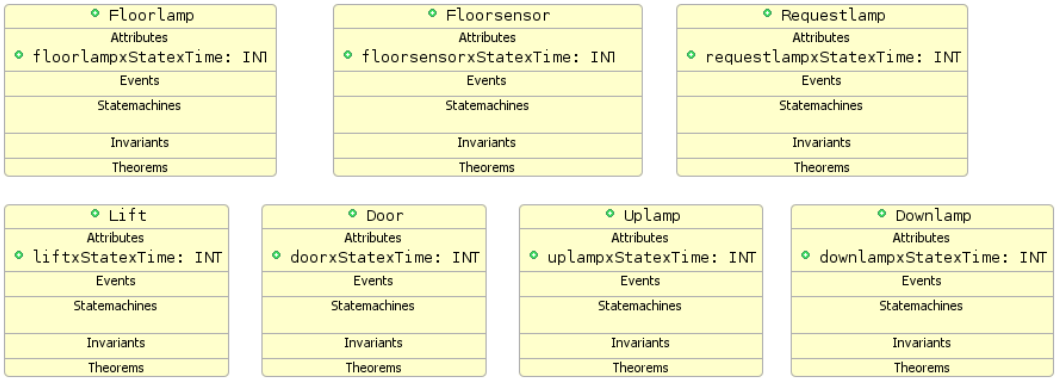


Figure 6-16 Lift system Class diagrams

Even though a TD has symbols ":" and "⊆" to indicate whether an object appears singly or multiple in a class, those symbols are not defined in a TD metamodel nor in a UML-B metamodel. That is because in UML-B, defining a class with many objects inside can be done by using a Machine Class; defining an object is done by a Machine Statemachine. This is not identified within a TD metamodel but depends on the user's choice. ATL translation rules create classes. Thus, if an object occurs singly in a system, such as in our lift system case study, the UML-B output model has to be modified as described in section 6.6.3 below.

6.5.5 *Creating UML-B Statemachines*

This section shows the `StateMachine` rule which is used to generate a UML-B Statemachine as shown in Figure 6-17. An example of a Statemachine that is generated by this rule is shown in Figure 6-18, and the corresponding parts of TD, **TDMetamodel** and **umlbMetamodel** are shown in Figure 6-21. In Figure 6-17, a **TDTimeline** is transformed into a **UMLBStatemachine** in which a Statemachine name is generated by **TDTimeline** name followed by the string `_state`. This rule also generates mappings of TD associations `states` and `timelinetransitions` to UML-B associations, `states`, and `transitions` respectively. This mapping is used to generate UML-B Statemachine states and transitions as shown later in the rules `State` and `Transition` respectively.

```
rule StateMachine {
      from t : TDMetamodel!TDTimeline
      to u : umlbMetamodel!UMLBStatemachine (
                  name <- t.name + '_state',
                  states <- t.states,
                  transitions <- t.timelinetransitions)
      }
```

Figure 6-17 ATL rule for creating a UML-B Statemachine

For example, the result from this rule generates a Statemachine named `floorsensor_state` for the class *Floorsensor* as shown in the following:

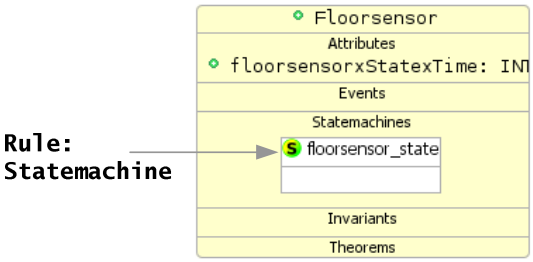


Figure 6-18 An example of a Statemachine generated from the rule Statemachine

6.5.6 *ATL translation rules for creating UML-B Statemachine states, transitions and actions*

Statemachine states and transitions are generated by the rules `State` and `Transition` as shown in Figure 6-19. Each transition is identified by a name which represents an Event-B name. A transition may have parameters, guards, and actions, which are created by rules `Parameter`, `Constraint`, and `Transition` respectively. Additional information may need to be identified to complete the model.

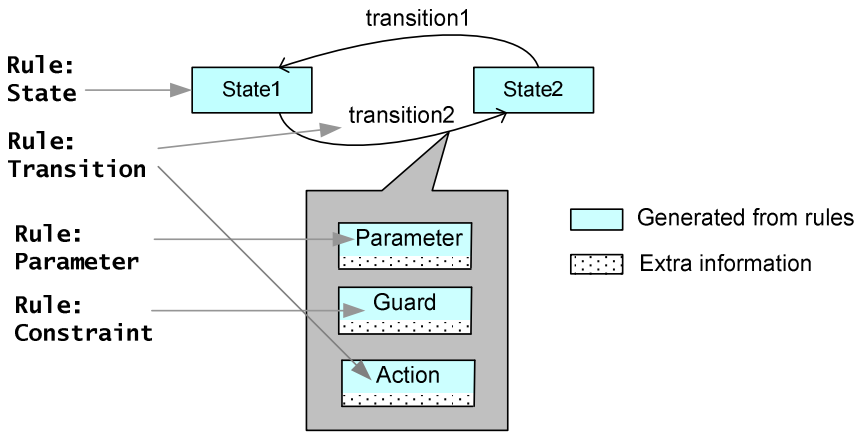


Figure 6-19 ATL rules for creating UML-B Statemachine State, Transition, Parameters and Actions

6.5.7 *Creating UML-B Statemachine states*

Figure 6-21 shows corresponding parts of TD, **TDMetamodel** and **umlbMetamodel** used to generate UML-B states and transitions.

Statemachine states are generated by the rule `State`. Each state has a name that is generated from **TDState** name as shown in Figure 6-20 line (4).

```
rule State {                                                    (1)
        from t : TDMetamodel!TDState                            (2)
        to u : umlbMetamodel!UMLBState                          (3)
             (name <- t.name,                                   (4)
             incoming <- t.segments -> collect(c|c.incoming),   (5)
             outgoing <- t.segments -> collect(c|c.outgoing))   (6)
}
```

Figure 6-20 ATL rule for creating UML-B State

Since UML-B does not have segments, TD `incoming` and `outgoing` associations cannot be directly mapped. Those associations are collected by a keyword `collect` and then assigned to `segments`. Next, those `segments` are

assigned to the corresponding **UMLBState** `incoming` and `outgoing` associations as shown in lines (5)-(6).

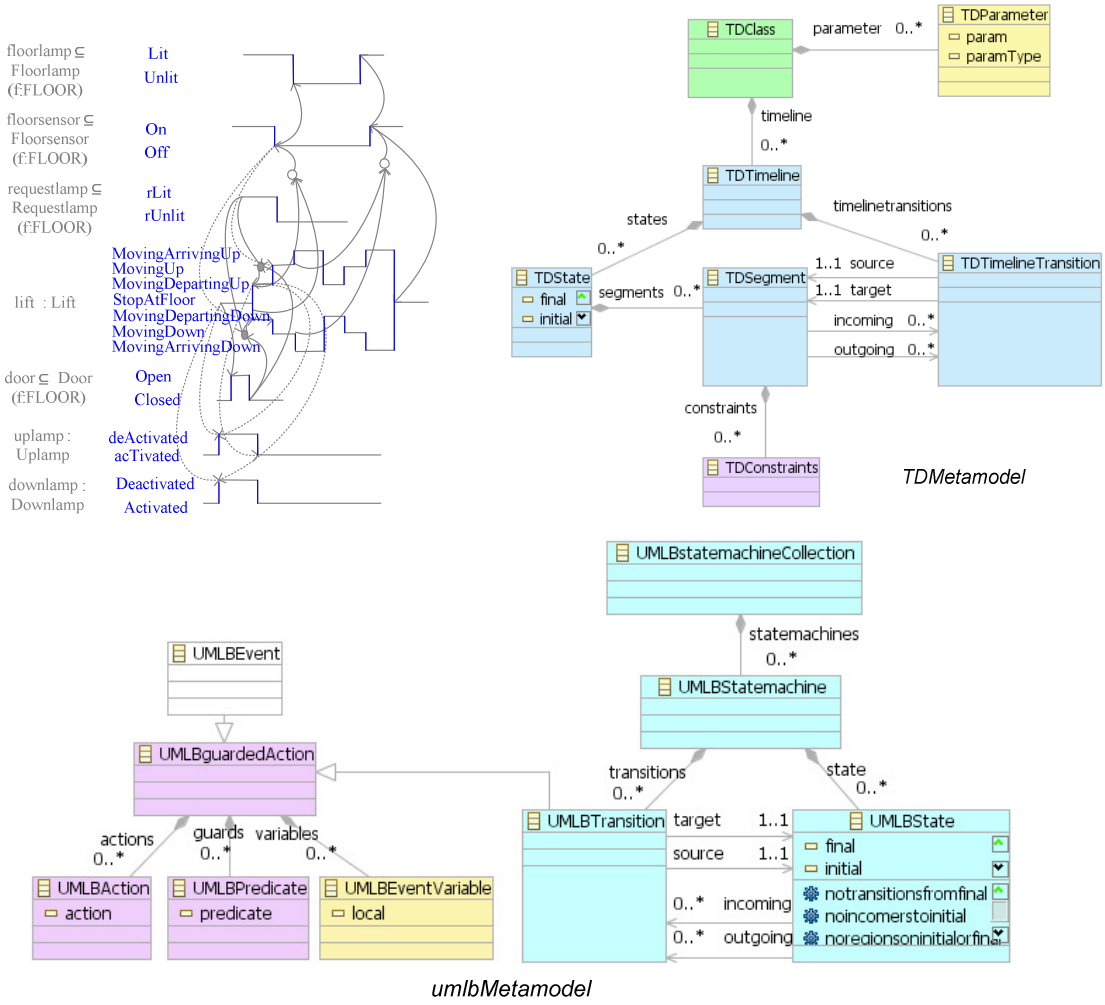


Figure 6-21 TDMetamodel and umlbMetamodel : Statemachine, State, Transition, Action, Guard and Parameter

6.5.8 *Creating UML-B Statemachine transitions and actions*

UMLBTransition can be generated from **TDTimelineTransition** by the rule `Transition` as shown in Figure 6-22. This rule is composed of two parts. The first part from lines (3)-(8), is for generating transitions, and the second part, in lines (9)-(15), is for creating actions.

```
rule Transition {                                                    (1)
from t : TDMetamodel!TDTimelineTransition                            (2)
to u : umlbMetamodel!UMLBTransition                                  (3)
     (name <- t.target.getTransitionName(),                          (4)
     target <- t.target.eContainer(),                                (5)
     source <- t.source.eContainer(),                                (6)
     guards <- t.target.constraints,                                 (7)
     variables <- t.eContainer().eContainer().parameter ),           (8)
actgclock : umlbMetamodel!UMLBAction                                 (9)
     (name <- t.eContainer().name + '.gClockAction',                 (10)
     action <- t.target.eContainer().eContainer().name               (11)
    + t.target.eContainer().name                                     (12)
    + 'Time('                                                        (13)
    + t.target.eContainer().eContainer().eContainer().name           (14)
    +'Self) := gclock' )                                             (15)
do {u.actions <- u.actions.append(actgclock); }                      (16)
}
```

Figure 6-22 ATL rule for creating UML-B Transition

First part, creating transitions: A transition has a name which represents an event's name and is created by the helper `getTransitionName` as shown in line (4). Lines (5)-(6) is maps TD associations `target` and `source` to UML-B associations `target` and `source`. Keyword `eContainer()` is used to refer to an upper class level in an aggregation association. For example, from **TDMetamodel** in Figure 6-21 and line (5) in Figure 6-22, the command `target <- t.target.eContainer()` means traversal from the class **TDTimelineTransition**, which is represented by `t` of the `target` association, to a class **TDSegment**. The `eContainer()` of the class **TDSegment** is the class **TDState**. Line (7) maps the TD association `t.target.constraints` to an UML-B association `guards`. This is for creating a UML-B transition guard. Line (8) shows an association creating transition parameters.

Second part, creating actions: Lines (9)-(15) generate an action for each transition. An action label, `.gClockAction`, is created in line (10), while the body of an action is created in lines (11)-(15). Line (16) appends the guard created earlier from, lines (9)-(15), to **UMLBTransition**. For example, in the following figure, `floorsensor.gClockAction` is a label while `floorsensorOnTime(FloorsensorSelf) := gclock` is a guard. This guard is used to record the current time whenever the corresponding event is activated.

Name	Action
floorsensor.gClockAction	floorsensorOnTime(FloorsensorSelf) := gclock

Figure 6-23 A *floorsensorOff* transition action

6.5.9 *Creating an Event name*

An event name is generated using the helper `getTransitionName()`, as illustrated in Figure 6-24.

```
helper context TDMetamodel!TDSegment .....(1)
def : getTransitionName() : String = .....(2)
let simuls : Set(TDMetamodel!TDSegment) = .....(3)
 TDMetamodel!TDSegment.allInstances()-> .....(4)
 select(c|c.simul ->includes(self)) .....(5)
 in .....(6)
 if simuls -> isEmpty() then .....(7)
      self.eContainer().eContainer().name .....(8)
      + self.eContainer().name .....(9)
 else .....(10)
      simuls.last().getTransitionName() .....(11)
 endif; .....(12)
```

Figure 6-24 ATL rule for creating an event name

The helper returns a string value which is a transition name and uses the keyword `self`, as shown in line (5), which represents an instance segment belonging to a `TDMetamodel!TDSegment`. At line (3), `simuls` is a variable defined for use only in this helper. This variable is initiated as a set of segments by the keyword `Set`. In line (4)-(5), the members of the `simuls` set are selected from `SimultaneityArrows` (`simul`) that is related to the segment indicated by the command `includes(self)`. For example in Figure 6-25, consider the segment `MovingUp3`, which is the `self` in this case. This segment has one `simul` a that is pointed from segment `Off2`. Thus, the `simuls` set for the segment `MovingUp3` is `simuls = {Off2}`. The segment `Off2` has no `SimultaneityArrow`. Thus, a set `simuls` for the segment `Off2` is defined as `simuls = {}`.

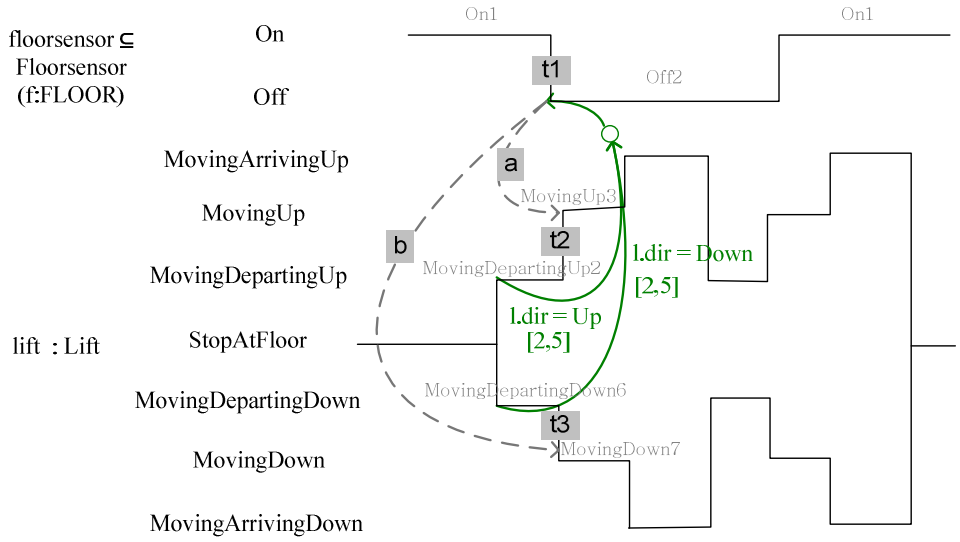


Figure 6-25 Timing diagram: floorsensor and Lift with `SimultaneityArrows`

Line (7) checks whether `simuls` is empty by keyword `isEmpty()`, if so an event name is generated from a Timeline name, in line (8), followed by a state name, in line (9). For example the segment `Off2`, which is a target segment of a transition t1, has `simuls = {}`. Thus, a transition name is generated from a Timeline name, *floorsensor*, followed with a state name, *Off*; a transition name for t1 is *floorsensorOff* as shown in Figure 6-26 top. If `simuls` is not empty, it returns the last elements in `simuls` to the helper `getTransitionName()` as shown in line

(11). For example in Figure 6-25, the segment `MovingUp3`, which is a target segment of a transition t2, has `simuls = {Off2}`, the segment `Off2` is sent to the helper `getTransitionName()`. The segment `Off2` itself has no `SimultaneityArrow`. We then follow the same process when generating a transition name for t1. Finally, the transition name for t2 is *floorsensorOff* which is a name for the transition link between the state *MovingDepartingUp* and *MovingUp* as shown in Figure 6-26 below. The transition name for t3 is also *floorsensorOff* which is generated following the same process of the transition t2 by `SimultaneityArrow` b. This transition links states *MovingDepartingDown* and *MovingDown*.

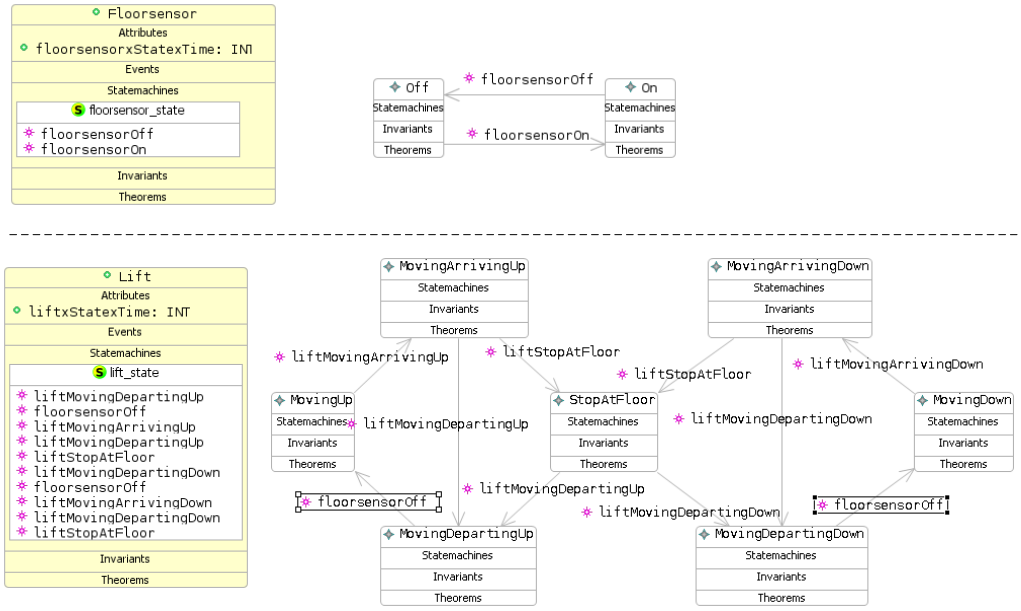


Figure 6-26 The *floorsensorOff* transitions are generated from `SimultaneityArrows`

Up to this point, the ATL translation rules generate Class, Statemachine inside that class, Statemachine states and transitions, and actions for the corresponding transitions, as shown by an example of class *Floorsensor* in Figure 6-27.

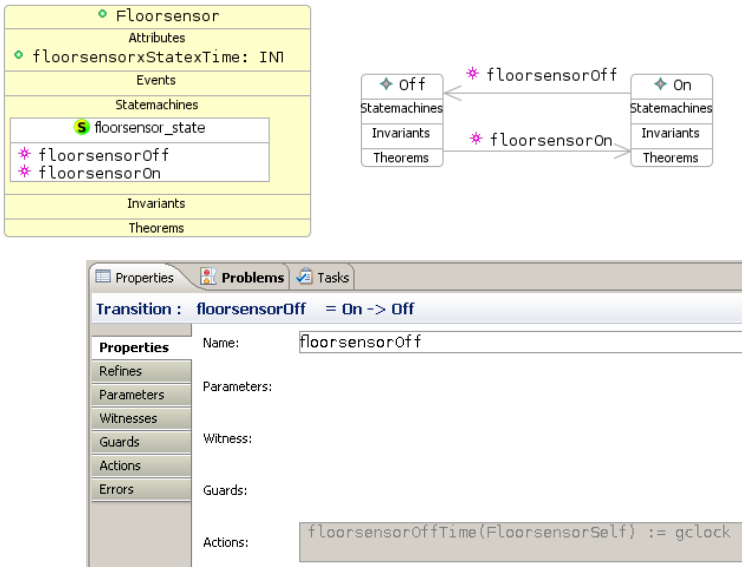
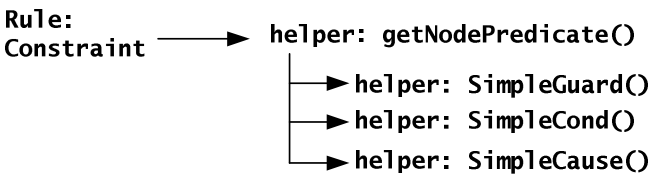


Figure 6-27 UML-B floorsensor Class diagram and its Statemachine

6.5.10 *Creating UML-B transition's guards*

The rule `Constraint` is used to generate guards of a transition. This rule uses the helper `GetNodePredicate()` which is made up of three sub-helpers: a helper for creating timing constraints (`SimpleGuard`), conditions (`SimpleCond`), and cause segments (`SimpleCause`), as shown below:



The details of the rule `Constraint` are shown in Figure 6-28. This rule creates a guard labelled `TimingCnstrntGuard` while the guard itself is generated by the helper `getNodePredicate()`.

```
rule Constraint{
 from t : TDMetamodel!TDConstraints
 to u : umlbMetamodel!UMLBPredicate (
            name <- 'TimingCnstrntGuard',
            predicate <- t.effectsource.getNodePredicate()) }
```

Figure 6-28 ATL main rule for creating UML-B Guards

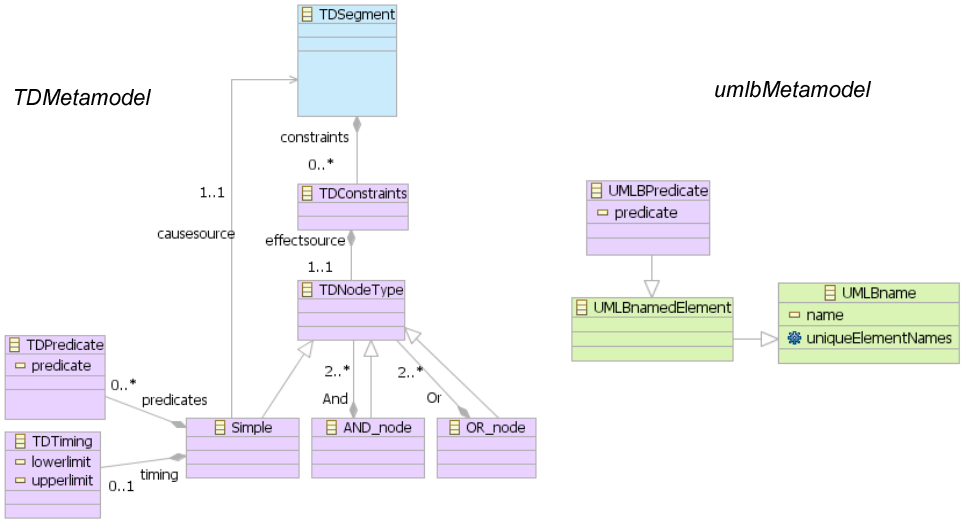


Figure 6-29 TDMetamodel and umlbMetamodel:

TDConstraints and **UMLBPredicate**

Checking Node types

Figure 6-29 shows the parts of **TDMetamodel** and **umlbMetamodel** used to generate the detail of a UML-B transition guard. The helper `getNodePredicate()`, as shown in Figure 6-30, is used for checking whether a node type is `Simple`, `OR_node` or `AND_node` as shown in lines (3), (13) and (20) respectively. If a node type is `Simple`, it further checks whether that `Simple` node type has timing constraints by an ATL function `oclIsUndefined()`, as shown in line (5). This function returns a Boolean value **true** if there is no timing. If `timing` is defined, a guard is generated by concatenating the output from the three helpers, i.e. `SimpleCause()`, `SimpleGuard()`, and `SimpleCond()`, as shown in lines (6)-(8). Otherwise, a guard is generated without timing constraints as shown in lines (10)-(11).

```
helper context TDMetamodel!TDNodeType .....(1)
def : getNodePredicate() : String = .....(2)
 if self.oclIsKindOf(TDMetamodel!Simple) .....(3)
 then .....(4)
```

```
        if not self.timing.oclIsUndefined() .....(5)
        then self.SimpleCause() .....(6)
        -> concat(' & '+ self.SimpleGuard()) .....(7)
        -> concat(self.SimpleCond()) .....(8)
        else .....(9)
        self.SimpleCause() .....(10)
        -> concat(self.SimpleCond()) .....(11)
        endif .....(12)
    else if self.oclIsKindOf(TDMetamodel!OR_node) .....(13)
        then self.Or -> iterate(e; ret : String = '('| .....(14)
            if e=self.Or.last() .....(15)
            then ret -> concat(e.getNodePredicate()+')') .....(16)
            else .....(17)
            ret -> concat(e.getNodePredicate()+') or (') .....(18)
            endif) .....(19)
        else if self.oclIsKindOf(TDMetamodel!AND_node) .....(20)
            then self.And -> iterate(e; ret : String ='('| .....(21)
                if e=self.And.last() .....(22)
                then ret -> concat(e.getNodePredicate()+')') .(23)
                else .....(24)
                ret -> concat(e.getNodePredicate()+ ') & (') (25)
                endif) .....(26)
            else 'unrecognised nodeType' .....(27)
            endif
        endif
    endif;
```

Figure 6-30 A helper for checking node types and event's guards

If a node type is `OR_node`, line (13), the sub-node type of the `OR_node` is collected by an expression `self.Or`, line (14). This collection is iterated by means of an `iterate` operation in which `e` represents an iterator, `ret` is a return value with an initial value equal to the string '`(`'. Each element in the collection is

checked whether it is the last node, as shown in line (15). If so, this node type is used in recursive call for the rule `getNodePredicate()`. The result generated from the rule is added with a symbol '`)`' at the end, line (16). Otherwise, this node type is used in recursive call for the rule `getNodePredicate()` and ending with a string '`) or (`' as shown in line (18). This is the way to generate guards with nested *OR* node types. A guard for *AND* node types also uses the same process as shown in lines (20)-(26). Note that whenever the string `&` and `or` are generated in a UML-B model, they are automatically changed to the ∧ and ∨ symbol.

Creating a guard from a Cause segment

The helper `SimpleCause()` is used to generate a guard from a cause segment, as illustrated in Figure 6-31. This helper works with the source model element `TDMetamodel!TDNodeType`. Thus, `self` in this case represents a node type. A guard is generated from a `Timeline` name of a cause segment, line (3), then concatenated with the string `_state(xAssociationx) =`, in line (4), followed by the state name in line (5).

```
helper context TDMetamodel!TDNodeType .....(1)
def : SimpleCause() : String = .....(2)
        self.causesource.eContainer().eContainer().name .....(3)
        + '_state(xAssociationx) = ' .....(4)
        + self.causesource.eContainer().name; .....(5)
```

Figure 6-31 A helper for creating a UML-B guard from a cause segment

The string `xAssociationx` is a mark for additional information added by hand. The reason is to have a complete UML-B model, one may have to declare associations among class or/and other classes' attributes, since TD notations do not support identifying that kind of information. Thus, the string `xAssociationx` is represented for the user to replace with the proper information later. Section 6.6.5 explains through examples the replacement of `xAssociationx`. Figure 6-32 shows

an example of a guard for the transition *floorsensorOff*. This example focuses on the part of guard generated by the helper `SimpleCause()`, while the parameter *f* with type *FLOOR* is generated by the rule `Parameter`, as shown in Appendix C. The whole guards for this transition are illustrated in the next section.

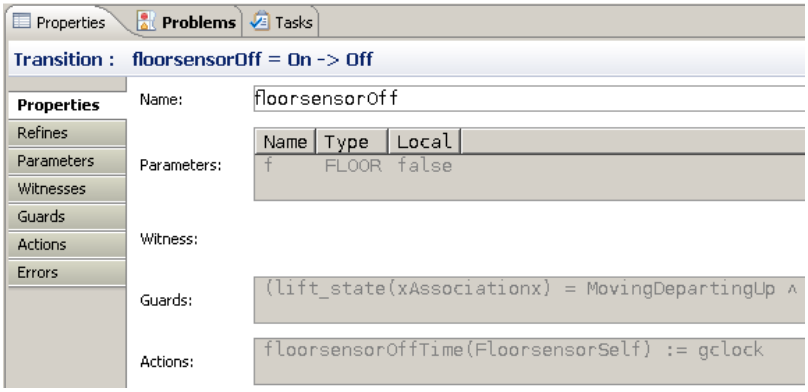


Figure 6-32 Guards generated from a cause segment for the *floorsensorOff* event

Creating a guard from Timing constraints

The helper `SimpleGuard()` is used for creating a UML-B guard from a timing constraint. The details of this helper are illustrated in Figure 6-33. This helper works with a source model element `TDMetamodel!TDNodeType`. Thus, `self` here represents a node type. The helper generates a guard by concatenating the string `(gclock - xAssociationx`, with other corresponding TD elements such as timing constraints.

```
helper context TDMetamodel!TDNodeType
def : SimpleGuard() : String =
      '(gclock - xAssociationx.'
      + self.causesource.eContainer().eContainer().name
      + self.causesource.eContainer().name
      + 'Time >= '
      + self.timing.lowerlimit.toString() + ') '
      + ' & (gclock - xAssociationx.'
      + self.causesource.eContainer().eContainer().name
      + self.causesource.eContainer().name
      + 'Time <= '
      + self.timing.upperlimit.toString() + ')';
```

Figure 6-33 The helper for creating a UML-B guard from a timing constraint

An illustration of how to generate a guard with the helper `SimpleCond()` is not show here, but its detailed explanation can be found in Appendix C. This helper simply takes predicates, if there are any defined within **TDPredicate**, see Figure 6-29, and concatenates with those guards generated by the helpers `SimpleCause()` and `SimpleGuard()`. An example of a guard for the transition *floorsensorOff* is shown below:

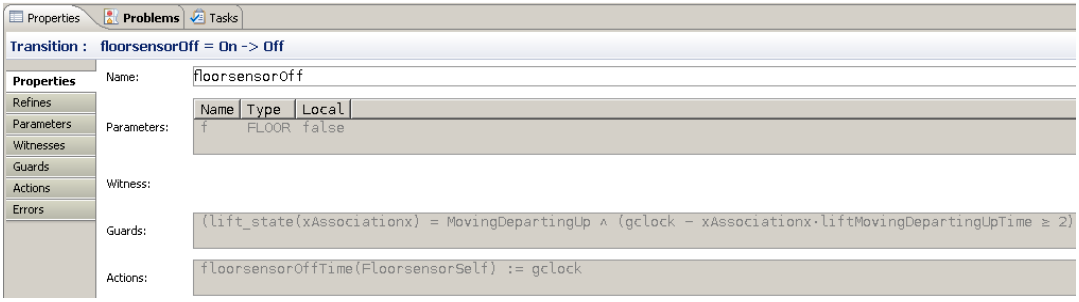


Figure 6-34 Timing constraint guard for *floorsensorOff* event

The UML-B tool does not allow adding a carriage return in the property view for a display arrangement. Thus, since the length of this guard is too long to be captured in one screen, we copy the whole guard from Figure 6-34 and represent it as the following:

```
(lift_state(xAssociationx) = MovingDepartingUp ∧ ◄── SimpleCause()
(gclock − xAssociationx.liftMovingDepartingUpTime ≥ 2) ∧
(gclock − xAssociationx.liftMovingDepartingUpTime ≤ 5)      } SimpleGuard()
∧ f = currentFl ∧ dir = Up)◄── SimpleCond()
∨
(lift_state(xAssociationx) = MovingDepartingDown ∧
(gclock − xAssociationx.liftMovingDepartingDownTime ≥ 2) ∧
(gclock − xAssociationx.liftMovingDepartingDownTime ≤ 5)
 ∧ f = currentFl ∧ dir = Down)
```

6.6 UML-B Model alteration

As mentioned above, TD is not designed to add state-based information nor gather whole system information. Thus, there are some UML-B model features that cannot be created by TD itself. In addition, ATL has a limitation and cannot generate multiple outputs from an input element if that element is used across the rules, as explained in section 6.5.4. This section identifies what features need to be added to an UML-B output model.

6.6.1 *Adding UML-B Context diagram body*

We can generate a UML-B context diagram name as shown in section 6.5.3. However, there are no details inside the context diagram such as *ClassTypes*, *Constants*, and *Axioms*. Thus, this part is generated by hand.

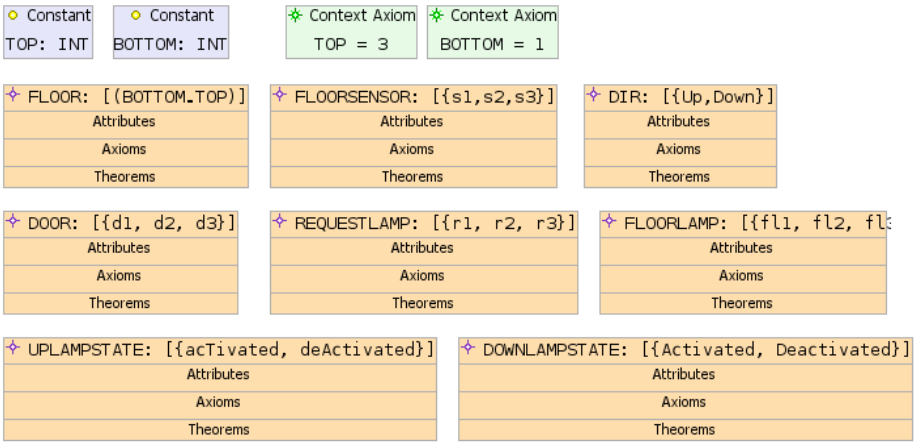


Figure 6-35 Context Diagram for the Lift system

In case of the lift system, *ClassTypes*, e.g. `FLOOR`, `FLOORSENSOR`, `DOOR`, `DIR`, etc., as shown in Figure 6-35, are generated as sets in Event-B unless it is assigned a constant value. For example, the *ClassType* `FLOOR` is defined as a set of integers `{1,2,3}`, representing a number of floors starting from 1. Thus, the *ClassType* `FLOOR` is generated as a constant, while its value is defined as *AXIOMS* in Event-B

as shown in Figure 6-36. `DIR` has its instances property set to `Up` and `Down` to identify the direction of the lift. `DIR` is created as a set while its instance properties are generated as *CONSTANTS* for an Event-B model, also shown in Figure 6-36.

```
CONTEXT
  L_ctx
SETS
  FLOORSENSOR      //   ClassType
  REQUESTLAMP      //   ClassType
  DOOR      //   ClassType
  DIR      //   ClassType
  UPLAMPSTATE      //   ClassType
  DOWNLAMPSTATE      //   ClassType
  FLOORLAMP      //   ClassType
CONSTANTS
  FLOOR      //   classType instances
  BOTTOM      //   utility constant
  TOP      //   utility constant
  Up      //   enumeration constant
  Down      //   enumeration constant
  s1      //   enumeration constant
  s2      //   enumeration constant
  s3      //   enumeration constant
  .....
AXIOMS
  FLOOR.value   :   FLOOR = (BOTTOM‥TOP)
  Axiom   :   TOP = 3
  Axiom1   :   BOTTOM = 1
  DIR.value   :   DIR = {Up,Down}
  FLOORSENSOR.value   :   FLOORSENSOR = {s1,s2,s3}
  REQUESTLAMP.value   :   REQUESTLAMP = {r1, r2, r3}
  DOOR.value   :   DOOR = {d1, d2, d3}
  UPLAMPSTATE.value   :   UPLAMPSTATE = {acTivated, deActivated}
  DOWNLAMPSTATE.value   :   DOWNLAMPSTATE = {Activated, Deactivated}
  FLOORLAMP.value   :   FLOORLAMP = {fl1, fl2, fl3}
  BOTTOM.type   :   BOTTOM ∈ ℤ
  TOP.type   :   TOP ∈ ℤ
  s1.type   :   s1 ∈ FLOORSENSOR
  s2.type   :   s2 ∈ FLOORSENSOR
  .....
```

Figure 6-36 Event-B Context part is generated from UML-B diagram for the Lift system

6.6.2 *Modifying UML-B Classes*

Modifying class attributes and defining classes to their corresponding sets

As described before, the string *xStatex* is used to illustrate missing information that cannot be created by TD itself, or from the limitations of ATL. For example, the class `Floorlamp` in Figure 6-37 left has an attribute defined by `floorlampxStatexTime : INT`.

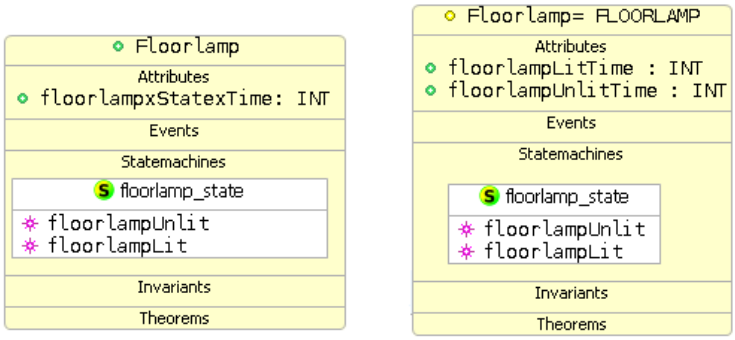


Figure 6-37 UML-B Class diagram for *Floorlamp* before and after modification

For it to be correct, attributes for this class are generated from every state corresponding to the class. The class `Floorlamp` has two states: `Unlit` and `Lit`. Thus, attributes for the class `Floorlamp` are `floorlampLitTime : INT` and `floorlampUnlitTime : INT` as shown in Figure 6-37 right.

To identify classes representing subsets of the corresponding *ClassTypes* that are introduced in the **CONTEXT**, an assignment <`Class = ClassType`> is used. For example, `Floorlamp = FLOORLAMP` (as shown in Figure 6-37 right) allows a `Floorlamp` class instance to get its values from *ClassType* `FLOORLAMP`.

Adding associations between classes and machine invariants

Associations between classes are information that is not declared by a TD. Which associations are added depend on each system specification. For example, in the lift case study, there are some associations added such as `doorAtfloor` and `floorsensorAtfloor` to declare a *door* and a *floorsensor* at a *floor* respectively, as shown in Figure 6-38 (other associations are shown in Appendix D). Those associations are created as variables with their invariants, as shown by an example for `floorsensorAtfloor` below:

```
Variables:- floorsensorAtfloor
Invariants:- floorsensorAtfloor ∈ Floor ⤖ Floorsensor
```

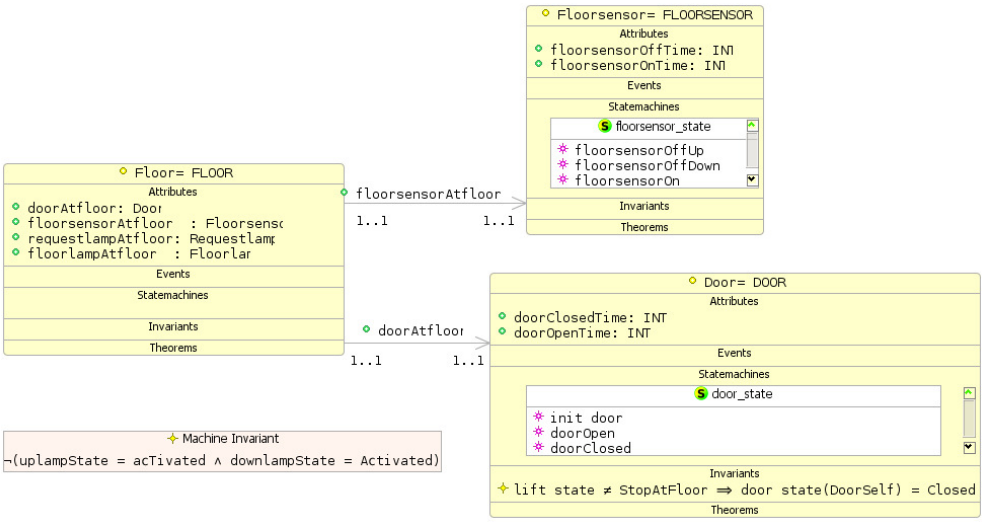


Figure 6-38 Association between classes

Figure 6-38 also shows how to declare invariants. Invariants can be defined manually inside a corresponding class as shown in the class `Door`, or defined as Machine Invariants. The invariant inside the class `Door` is used to indicate whenever the lift is not stationary at a floor, the lift door must be closed. The Machine Invariant indicates that *uplamp* and *downlamp* must not be activated at the same time. The rest of the invariants can be found in Appendix D.

6.6.3 *Modifying to create a lift in a system*

Since ATL translation rules generated a class `Lift`, to create a lift in a system, the class `Lift` is changed to a Statemachine `lift_state` as shown in Figure 6-39.

The class `Lift`'s attributes, such as `liftMovingArrivingUpTime` and `liftStopAtfloorTime`, must then change to machine variables. Other related variables like `currentFl` and `dir` are used to represent a current position and directions of the lift are added by hand. There are extra events: `ChangeDirUp` and `ChangeDirDown` are manually created for controlling the change in direction of the lift.

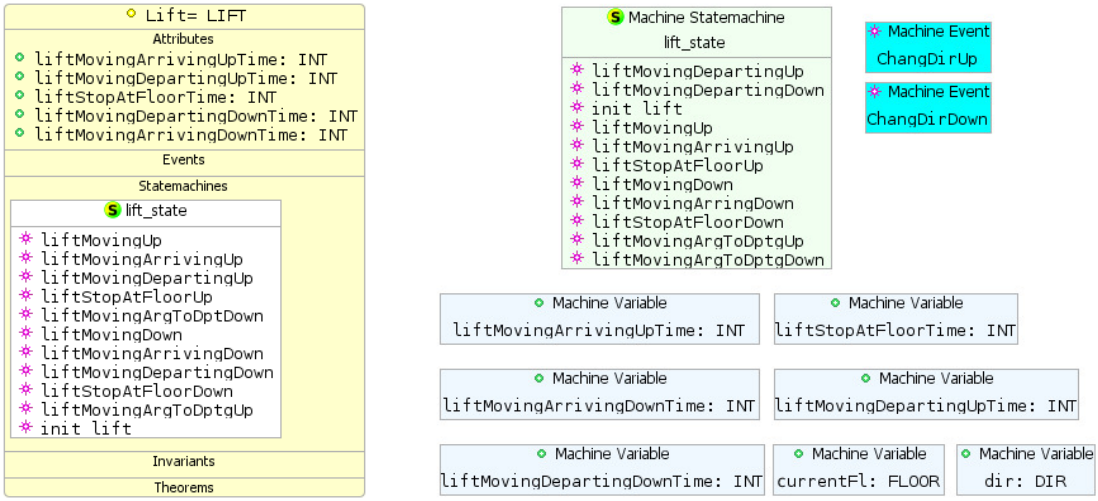


Figure 6-39 A class lift is changed to a Statemachine lift_state

6.6.4 *Modifying UML-B Statemachine*

Modifying Statemachine initial state

Our rule can generate the Statemachine for each class. However, one needs to identify an initial state for that Statemachine. For example, Figure 6-41 shows the door_`state` Statemachine before and after adding an initial state. This initial state generates an Event-B *INITIALISATION* as shown in Figure 6-40.

```
INITIALISATION   ≙
STATUS
  ordinary
BEGIN
  door_state.init   :   door_state ≔ Door × {Closed}
```

Figure 6-40 Parts of an Event-B model: generate *door* initialisation

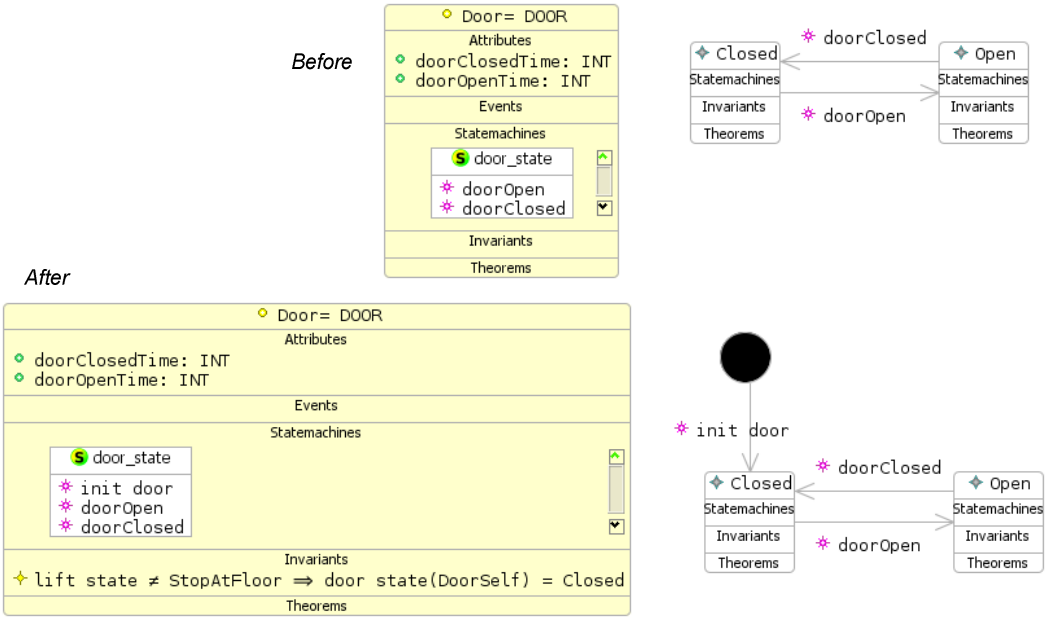


Figure 6-41 UML-B Statemachine for Door before and after modification

Modifying Statemachine Transitions

Each UML-B Statemachine transition generates an Event-B event with the corresponding transition name. Therefore, each transition name should to be unique, as well as its action should do a specific task and not be in conflict. Two problems occur with the UML-B Statemachine generated from ATL and U2B from the example shown in Figure 6-42. Its corresponding Machine Statemachine `lift_state` and Statemachine `Lift` are shown in Figure 6-39 and Figure 6-43 respectively.

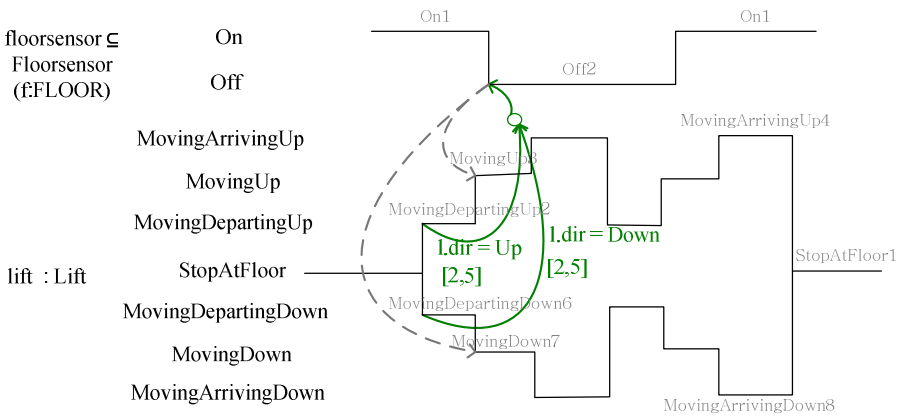


Figure 6-42 TD for the Lift and Floorsensor

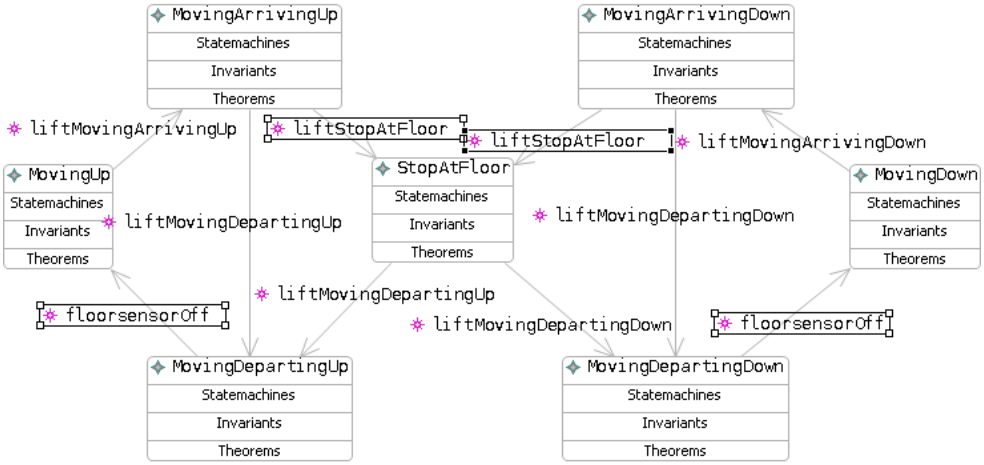


Figure 6-43 Statemachine for the Lift generated from ATL

The first problem concerns the condition that the lift can *StopAtFloor* whenever it is in a previous state of *MovingArrivingUp* or *MovingArrivingDown*, as shown in Figure 6-42. The Statemachine corresponding to the `Lift` is shown in Figure 6-43 in which there are two state transitions assigned with the same name `LiftStopAtFloor`. The U2B translator converts a UML-B model to an Event-B model as shown in Figure 6-44.

```
liftStopAtFloor   ≙
STATUS
  ordinary
ANY
  f
WHERE
  f.type   :   f ∈ FLOOR
  liftStopAtFloor.TimingCnstrntGuard   :   floorsensor_state
                                           (xAssociationx) = On ∧ (gclock − xAssociationx
  lift_state_isin_MovingArrivingUp   :   lift_state = MovingArrivingUp
  lift_state_isin_MovingArrivingDown   :   lift_state = MovingArrivingDown
THEN

    ...
```

Figure 6-44 An Event-B *liftStopAtFloor* event generated from UML-B *liftStopAtFloor* transition

Consider the two highlighted guards; these guards are previous states before the lift stops at the floor, and are automatically generated by U2B, not by ATL rules. These guards made the event `liftStopAtfloor` incorrect since the two guards are in conflict. That is, the lift cannot be in a state of *MovingArrivingUp* and *MovingArrivingDown* at the same time. This problem can be fixed by combining these two guards with a conjunction $\vee$ (or) by hand. This combination may be generated automatically if and only if the U2B translator is re-designed to do this. However, we selected to solve this problem another way. In the solution, those transitions `liftStopAtfloor` are assigned to different names as shown in Figure 6-45, in order to have them generated separately in Event-B as shown in Figure 6-46. This way, events are simpler and easier for proving than combining guards together within an event.

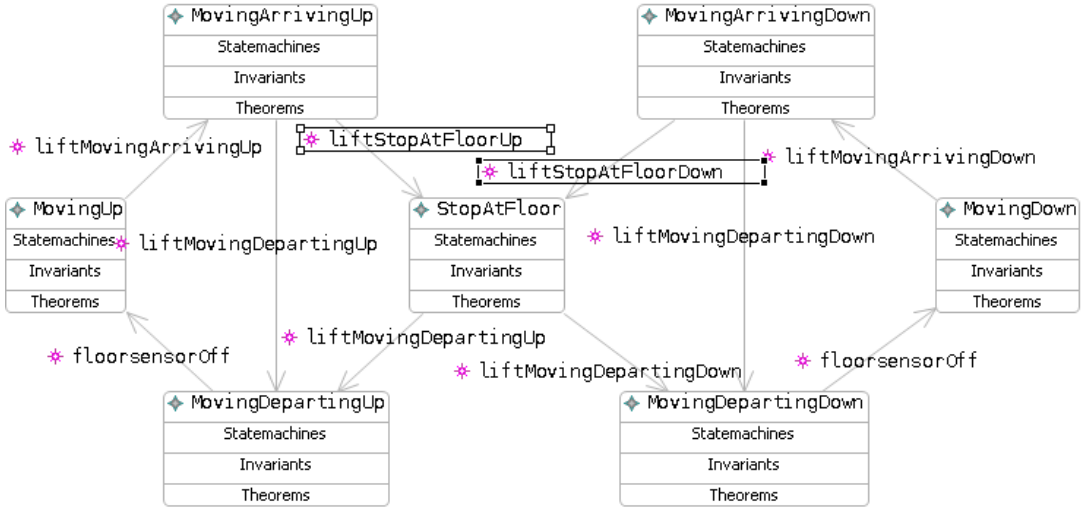


Figure 6-45 UML-B transitions *liftStopAtFloorUp* and *liftStopAtFloorDown* after modification

```
liftStopAtFloorUp   ≙
STATUS
  ordinary
ANY
  f
WHERE
  f.type   :   f ∈ FLOOR
  liftStopAtFloorUp.TimingCnstrntGuard   :   floorsensor_state
                                             (xAssociationx) = On ∧ (gclock − xAssociationx·floorsens
  lift_state_isin_MovingArrivingUp   :   lift_state = MovingArrivingUp
THEN

     ...
liftStopAtFloorDown   ≙
STATUS
  ordinary
ANY
  f
WHERE
  f.type   :   f ∈ FLOOR
  liftStopAtFloorDown.TimingCnstrntGuard   :   floorsensor_state
                                               (xAssociationx) = On ∧ (gclock − xAssociationx·floors
  lift_state_isin_MovingArrivingDown   :   lift_state = MovingArrivingDown
THEN

     ...
```

Figure 6-46 Event-B events: *liftStopAtFloorUp* and *liftStopAtFloorDown*

The second problem happens because `SimultaneityArrow`s. Figure 6-42 shows that there are two `SimultaneityArrow`s from the segment *Off2* to segments *MovingUp3* and *MovingDown7*. This causes UML-B to generate two state transitions with the same name `floorsensorOff`, as shown in Figure 6-43 (by the helper `getTransitionName()` as shown in Figure 6-24). The problem is U2B generates those two UML-B transitions to the same Event-B event, `floorsensorOff`, as shown in Figure 6-47. This event is incorrect since guards and actions themselves (highlighted) are in conflict. The lift cannot be in states of *MovingDepartingUp* and *MovingDepartingDown* at the same time, nor can it be in the states of *MovingUp* and *MovingDown* after performing the event. However, changing transition names alone raises another problem. This is because not only is the `floorsensorOff` transition generated in the *Lift* Statemachine, but also in the *Floorsensor* Statemachine, as shown in Figure 6-43 and Figure 6-48.

```
floorsensorOff   ≙
STATUS
  ordinary
ANY
  FloorsensorSelf      //   contextual instance of class Floorsensor
  f
WHERE
  lift_state_isin_MovingDepartingDown    :   lift_state = MovingDepartingDown
  lift_state_isin_MovingDepartingUp    :   lift_state = MovingDepartingUp

    ...

THEN
  lift_state_enterState_MovingDepartingDown    :   lift_state := MovingDown
  lift_state_enterState_MovingDepartingUp    :   lift_state := MovingUp

    ...
```

Figure 6-47 An Event-B *floorsensorOff*

Following this example, if we rename the transition `floorsensorOff` in the *Lift* Statemachine, we have to rename it with the same name in the *Floorsensor* Statemachine. Unfortunately, UML-B does not allow renaming elements already existing with that name, even though they are generated with the same transition name from the beginning.

Figure 6-48 A Statemachine for *floorsensor*

Thus, the solution to this problem is renaming `floorsensorOff` transitions in the *Lift* Statemachine to `liftMovingUp` and `liftMovingDown`, and splitting the `floorsensorOff` transition in the *Floorsensor* Statemachine to `floorsensorOffUp` and `floorsensorOffDown` as shown in Figure 6-49.

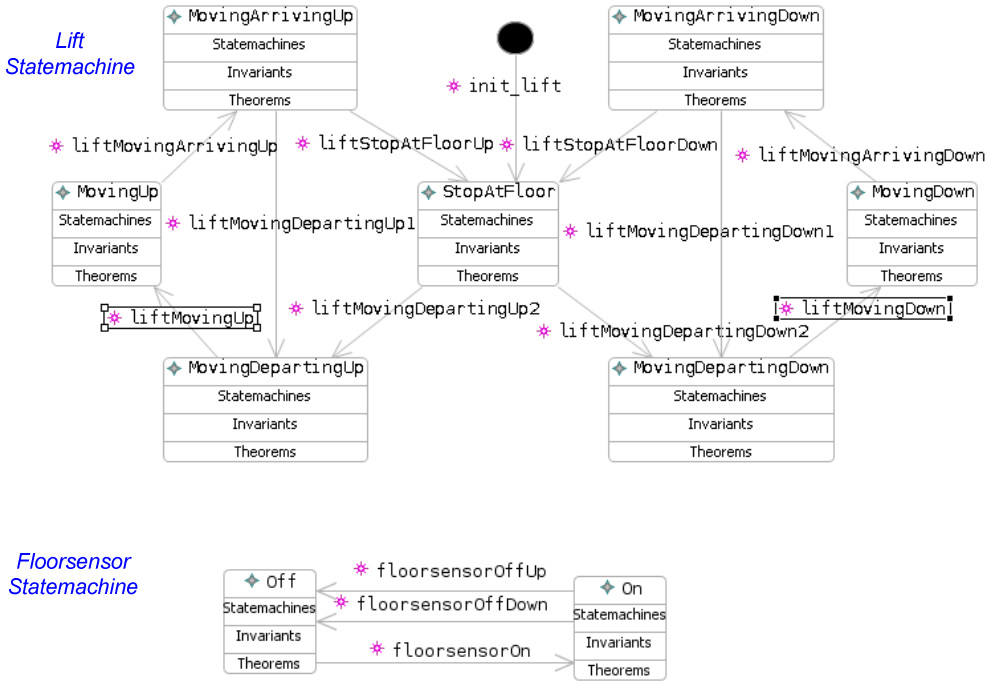


Figure 6-49 A Statemachine for *lift* and *floorsensor*

Guards and actions for events `floorsensorOffDown` and `floorsensorOffUp` are split from the former `floorsensorOff` transition.

6.6.5 *Modifying UML-B event guards*

As mentioned in section 6.5.10, a transition guard is generated with the marking *xAssociationx*, which needs to be deleted or replaced. To explain how to delete *xAssociationx*, consider the timing constraint guard for the transition *floorsensorOffUp* (the corresponding Statemachine is shown in Figure 6-49) with *xAssociationx* as illustrated below:

```
(gclock − xAssociationx.liftMovingDepartingUpTime ≥ 2) ∧
(gclock − xAssociationx.liftMovingDepartingUpTime ≤ 5)
```

Since the Statemahine `Lift` has no association to other classes, this guard is altered by deleting marking *xAssociationx*. Thus, the correct version of this transition's guard is shown below.

```
(gclock - liftMovingDepartingUpTime ≥ 2) ∧
(gclock - liftMovingDepartingUpTime ≤ 5)
```

In some cases, the marking has to be replaced by corresponding associations and attributes. Those associations and attributes are created earlier by hand in **CONTEXT** and/or class diagram. For example, Figure 6-50 top illustrates an association `floorsensorAtfloor` between classes `Floor` and `Floorsensor`, where a Statemachine `floorlamp_state` is shown at the bottom of the figure.

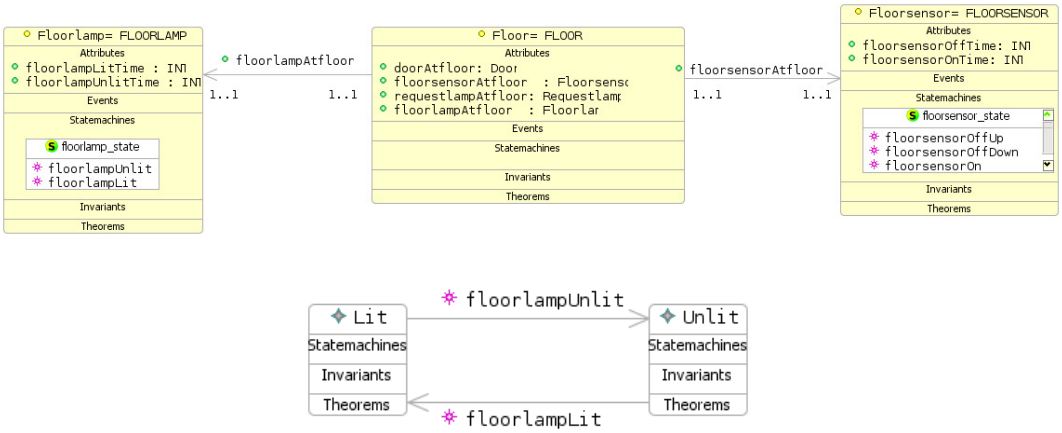


Figure 6-50 An association between classes *Floorlamp*, *Floor* and *Floorsensor*

The transition `floorlampUnlit` in Figure 6-50 has part of the guard generated by the ATL translation rules as shown below:

```
(gclock - xAssociationx.floorsensorOffTime ≥ 2) ∧
(gclock - xAssociationx.floorsensorOffTime ≤ 4)
```

The marking in this guard is replaced by an association `floorsensorAtfloor` and a variable `currentFl`, from Figure 6-39, as illustrated below:

```
(gclock - (floorsensorAtfloor(currentFl)).floorsensorOffTime ≥ 2)
∧
(gclock - (floorsensorAtfloor(currentFl)).floorsensorOffTime ≤ 4)
```

The symbol "." represents referring to an attribute for the corresponding class. This symbol is changed to "( )" automatically by the U2B translator whenever it is found in an expression. For the example above, it is changed to

```
(gclock − floorsensorOffTime((floorsensorAtfloor(currentFl))) ≥ 2)
∧
(gclock − floorsensorAtfloor((floorsensorAtfloor(currentFl))) ≤ 4)
```

This is the way one can correct the marking *xAssociationx*. Other *xAssociationx* are replaced with a similar technique. Figure 6-51 shows the full detail of the event `floorlampUnlit` in Event-B.

```
floorlampUnlit   ≙
STATUS
  ordinary
ANY
  FloorlampSelf     //   contextual instance of class Floorlamp
  f
WHERE
  f.type   :   f ∈ FLOOR
  FloorlampSelf.type   :   FloorlampSelf ∈ Floorlamp
  floorlamp_state_isin_Lit   :   floorlamp_state(floorlampAtfloor(currentFl)) = Lit
  floorlampUnlit.Guard3   :   floorsensor_state(floorsensorAtfloor(currentFl)) = Off
                                  f = currentFl ∧
  floorlampUnlit.TimingCnstrntGuard   :   (gclock − floorsensorOffTime((floorsensorAtfloor
                                  (currentFl))) ≥ 2) ∧ (gclock − floorsensorOffTime
                                  ((floorsensorAtfloor(currentFl))) ≤ 4)
  floorlampUnlit.Guard1   :   floorlampAtfloor~(FloorlampSelf) = currentFl
THEN
  floorlamp_state_enterState_Unlit   :   floorlamp_state(floorlampAtfloor(currentFl)) ≔ Unlit
END
```

Figure 6-51 An event `floorlampUnlit` is generated in Event-B

6.6.6 *Timing Constraints*

As described earlier, the event *Ticktok* is generated by the rule `Machine`, as shown in Figure 6-12. The rule also generates the event action, that is *gclock* := *gclock* + *1*. The variable *gclock*, whose type is assigned to an integer and initial value 0, is also created by this rule. The event guards are manually created, using the same technique with the Event-B direct translation as described in section

5.3.11. The *Ticktok* event provides time progress as an output value. Below is a part of the event *Ticktok*'s guards.

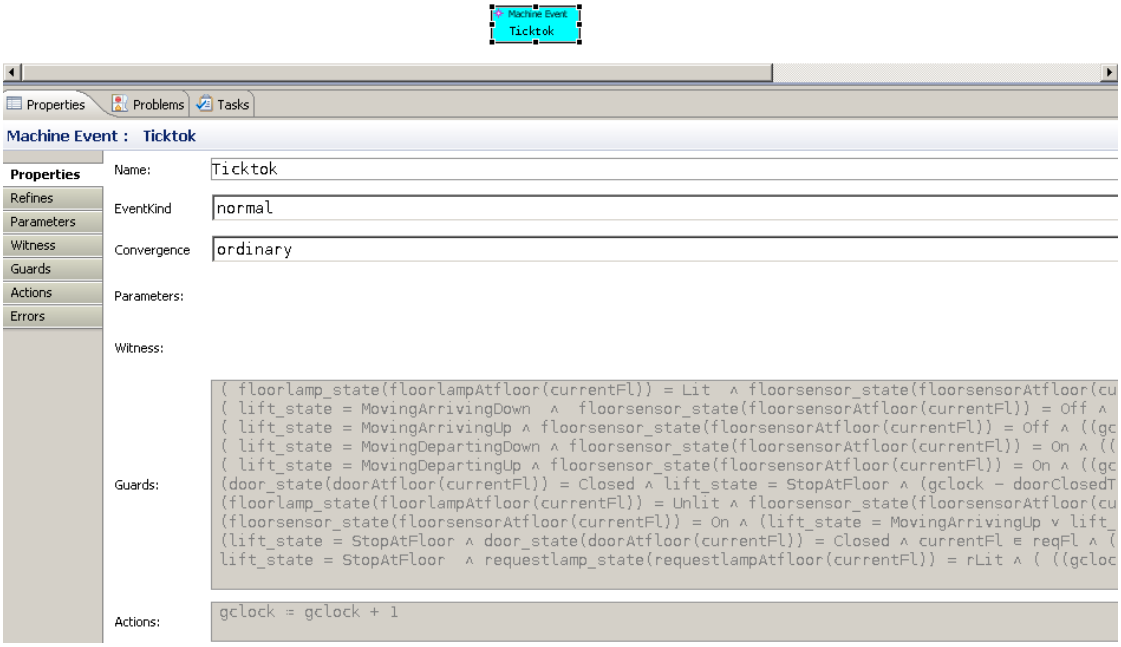


Figure 6-52 A *Ticktok event*

6.7 Summary

This chapter explains how to generate a UML-B model from a TD using ATL rules. TD used for this translation is slightly different from the direct translation (Chapter 5), as identifying a first letter for a class name by a capital letter (as described in section 6.1). This is because the class names here are used to generate class in Class diagram in the **MACHINE** part, while class names in Chapter 5 are used to generate sets in the **CONTEXT** part. Since the UML-B metamodel does not specify if there are single or multiple objects for a class, but leaves it to user choice, ATL translation rules generate only TD classes and objects in classes. Thus, one needs to alter the result model by hand to have it fit the system specification. Here is the summary of generating an UML-B from TD.

First, TD metamodel is created and used to describe abstract syntax of TD. It is an Ecore model itself.

Secondly, a TD model conforms to the TD metamodel generated by Eclipse EMF. This model is used as a source model for the ATL translation rules.

Next, the ATL translation rules for creating a UML-B model are identified. The rules can generate a UML-B Project, **CONTEXT** name, **MACHINES** and Class diagrams. For a Class diagram, the rules can generate attributes and its Statemachines. For a Statemachine, we create rules for generating states and state transitions. Each state transition comprises parameters, guards and actions that are created from our rules. We also have a rule for creating a *Ticktok* event for time progressing.

Finally, since TD illustrates some parts of the whole specifications, an output UML-B model generated from ATL rules has to be completed or modified. For example, associations among classes need to be added since they cannot be identified by TD notations. Some events, such as `ChangeDirUp` and `ChangeDirDown`, and invariants, are invented. Moreover, ATL does not allow generating UML-B components from TD elements already used in another rule. Thus in ATL rules, a symbol *xStatex* and *xAssociationx* are used as marks where the UML-B output model components require adjustment. The *xStatex* represents states needing replacement, while *xAssociationx* needs to be replaced by association among *Classes*. Additional parts of a UML-B model that have to be modified are: **CONTEXT** diagram, class attributes, initial state for a Statemachine and some Statemachine transition guards.

UML-B tool itself also does not fully support identifying multiple previous states of the same target state, see section 6.6.4. The same problem occurs with `SimultaneityArrow`s. For example, where there are two `SimultaneityArrow`s originally starting from a same source segment but ending at different target segments. Thus, the generated UML-B output model has some Statemachine transitions providing the same name. U2B translator gathers those same transition names to generate an event. As a result, this event comprises guards and actions from many conflicting transitions. We need to split these kinds of event into many events with corresponding guards and actions.

Chapter 7 Translating Timing diagrams into KAOS

This chapter investigates the techniques for generating KAOS *Goal* and *Operation models* from TD. KAOS is a semi-formal method in which each goal definition is identified by linear temporal logic (LTL). TDs demonstrate system specifications in some temporal logic shapes along a timeline, i.e. in the next state ($\circ$), some time in the future ($\Diamond$), and entails ($\Rightarrow$). Thus, it is possible to generate KAOS from TD. This transformation attempts to add a KAOS graphical capability for expressing timing constraints and event dependency requirements. Transforming TD into KAOS can help check the completeness of a system's goals. Additional information, that may need to be added to KAOS that is obtained from the generating process, could identify what information is missing from the TDs.

This chapter starts with defining the scope of TD and LTL operators used for the translation. Section 7.2 explains BNF TD definition used for transforming into KAOS. Section 7.3 gives steps for generating KAOS *Goal* and *Operation models* from TD. Section 7.4 provides textual translation rules. Sections 7.5 and 7.6 explain how to create a goal from a segment defined with `CauseEffectArrow`s and `SimultaneityArrow`s. Section 7.7 describes a technique for splitting a goal into subgoals whenever the goal's pre-condition is defined with the OR relationship. Section 7.8 explains techniques used to generate goal trees. Section 7.9 gives examples of user manual input to modelling. Section 7.10 shows examples of generating *Operation model*.

7.1 Scope of LTL operators and shape of Timing Diagrams

For KAOS, we are concerned with generating events that will occur in the future under the timing constraints provided. We are not dealing with timing constraints that have occurred in the past states. The example on the next page gives a case where it would be useful to support past operators. However, LTL past operators are not used for defining a KAOS goal model. In other words, we are not modelling a KAOS goal that includes timing constraints as pre-conditions (because it has to be defined as a past operator) but in a post-condition (see section 2.7.2 for the KAOS goal structure).

Currently, we have found in the case study that there are two LTL future operators which correspond to two KAOS *Goal models*: `Maintain Global invariant P` $\Rightarrow$ `Q` and `Bounded achieve P` $\Rightarrow$ $\Diamond_{\leq d}$ `Q` (see section 2.7.2 and 2.7.5). Our work is generating translation rules to create these kinds of KAOS goal models.

Aspects a timing constraint does allow

To clarify what TD is suitable for using KAOS translation, consider a room heating and humidity control problem as defined below.

"*…whenever the room temperature is overheated or the room is overhumid with a condition that there is electricity in the system, a room window will be eventually opened between 3 and 5 seconds…*"

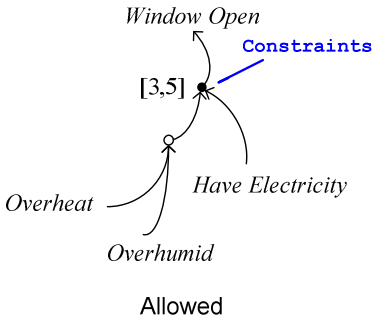


Figure 7-1 A timing diagram where KAOS translation is allowed

The specification above generates a TD as shown in Figure 7-1, where a corresponding goal formal definition is defined by `Bounded achieve P` $\Rightarrow \Diamond_{\leq d}$ `Q` as shown in the following:

pre-condition P: *(Overheat* $\vee$ *Overhumid))* $\wedge$ *Have Electricity*

$\Rightarrow$

post-condition Q: $\Diamond_{[3,5]}$ *Window Open*

This kind of TD is allowed for the KAOS transformation since there are no past operators in the pre-conditions.

Aspects a timing constraint does not allow

If the room heating and humidity control problem specification above is modified to "*...whenever the room temperature is overheated or the room is very humid, for between 1 and 2 seconds with a condition there is electricity in the system, a room window will be eventually opened between 3 and 5 seconds…*".

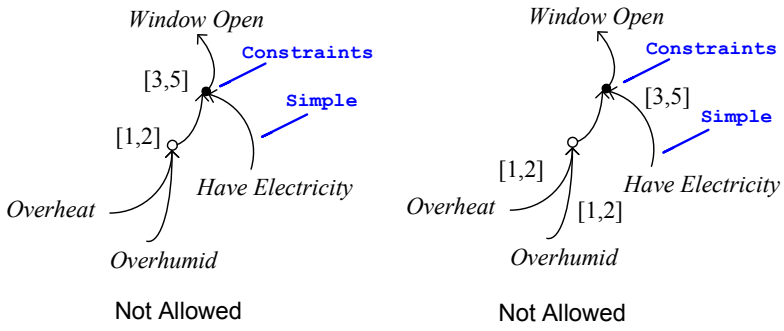


Figure 7-2 Timing diagrams where KAOS translation is not allowed

A TD generated with this new specification above is illustrated in Figure 7-2 left. This kind of TD with nested timing constraints is not allowed for the KAOS transformation. That is because nested timing constraints cause a pre-condition to be included with a LTL past operator ♦ (some time in the past), which we are not dealing with at this moment as shown in the following.

pre-condition : $\blacklozenge_{[1,2]}$ *(Overheat* $\vee$ *Overhumid))* $\wedge$ *Have Electricity*

$\Rightarrow$

post-condition : $\Diamond_{[3,5]}$ *Window Open*

Another example supposes timing constraints are defined by a Simple node (see section 5.1 for the original TD BNF definitions) as shown in Figure 7-2 right. This kind of TD is not allowed for the KAOS transformation. That is because, not only nested timing constraints alone force one to define LTL past operators, but also having timing constraints by a Simple node allows identifying multiple timing constraints in a `CauseEffectArrow`. It is complicated and unclear how to generate a KAOS goal formal definition from this kind of TD. This is the reason the TD BNF definition for KAOS is slightly different from that defined in the direct translation in chapter 5. The detail of TD BNF definitions for KAOS is described in the following section.

7.2 BNF Timing Diagram for KAOS

Most TD BNF definitions used for KAOS translation is the same as that defined by the Event-B direct translation (Chapter 5). However, for KAOS, there is a difference in defining timing constraints. That is, a timing constraint (Timing) is directly connected with Constraints instead of a Simple segment. The rest of TD BNF definitions are the same. The TD BNF definitions for KAOS shown below highlight the definitions for Constraints and Simple concerned with the differentiation.

Project ::= name Machine$^+$

Machine ::= name Class$^+$

name ::= String

Class ::= ClassName Obj$^+$ Obj_Def

ClassName ::= String

Obj_Def ::= ObjName " $\subseteq$ " Class_Clause | ObjName " : " Class_Clause

ObjName ::= String

```
Class_Clause ::= ClassName |
ClassName"(" Param ":" ParamType ( "," Param ":" ParamType)* ")"
Param ::= name
ParamType ::= name
Obj :: = ObjName ObjSt+ Timeline
Timeline ::= name Segment+
Segment ::= ObjSt number Simul* [CauseEffectArrow]
ObjSt ::= name
number ::= Z+
Simul ::= StartSegmt EndSegmt
StartSegmt ::= Segment
EndSegmt ::= Segment
CauseEffectArrow ::= Constraint
Constraint ::= NodeType [Timing]                    /* different from earlier */
NodeType ::= Simple | OR_node | AND_node
Simple ::= CauseSegmt [Predicate*]                  /* different from earlier */
CauseSegmt ::= Segment
Predicate ::= String
OR_node ::= NodeType ○ NodeType
AND_node ::= NodeType • NodeType
Timing ::= "[" lowerlimit "," upperlimit "]"
lowerlimit ::= Z+
upperlimit ::= Z+
```

Figure 7-3 illustrates TD used for transforming to KAOS *Goal* and *Operation models*. Notice that there is a timing constraint for each `CauseEffectArrow`. Numbers such as 1, 2, and 3 are not TD notations but are used for explanation in this chapter.

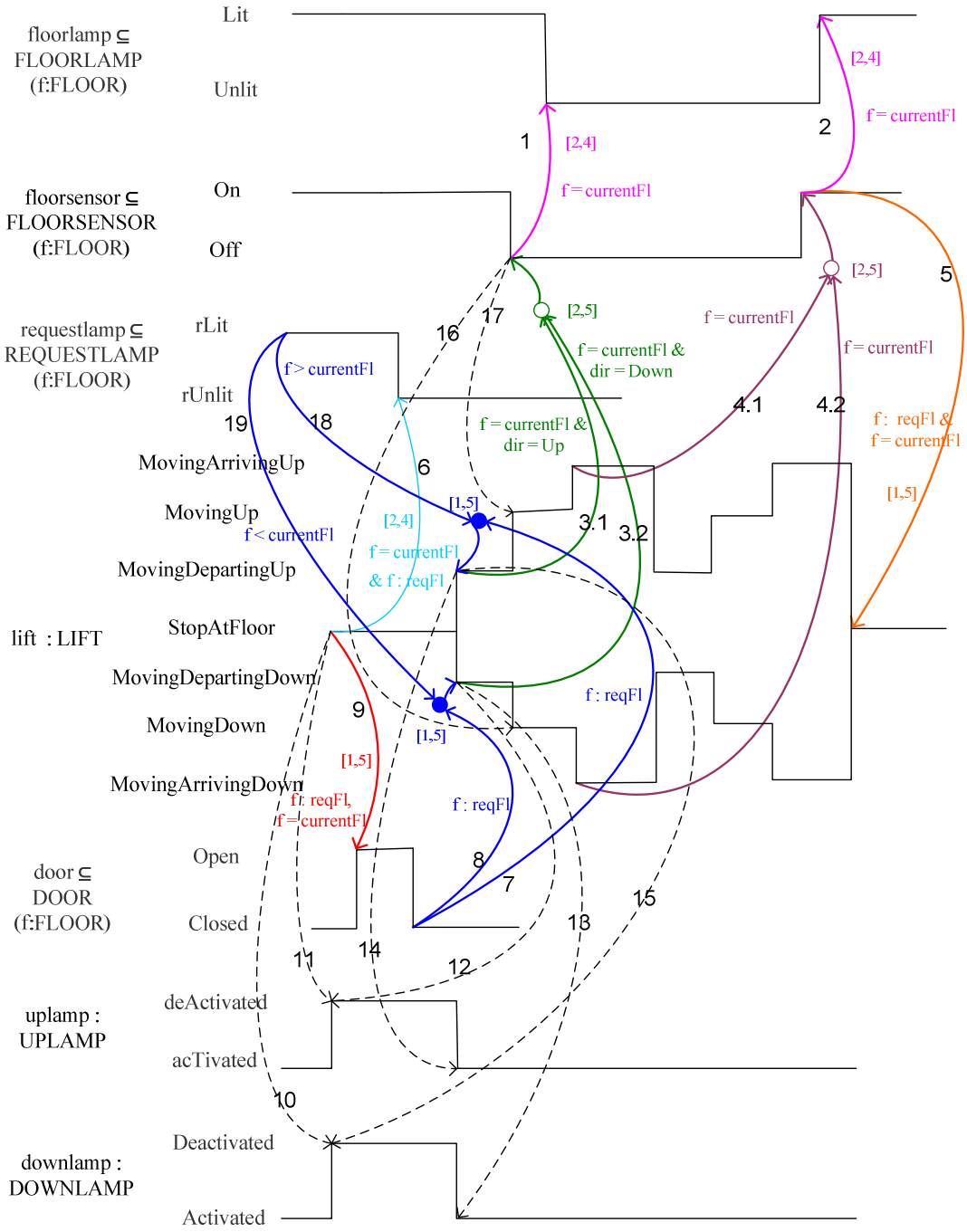


Figure 7-3 Timing diagram used for KAOS Models

7.3 Steps in generating KAOS Goal and Operation models

Generating a KAOS *Goal* and *Operation models* comprises four steps.

1. A KAOS goal is created by two TD notations: segments which are declared with `CauseEffectArrow`s and `SimultaneityArrow`s. This step uses our textual transformation rules, as explained in sections 7.5 and 7.6.

2. Consider the goals obtained from step 1:

- If the goal pre-condition is declared with OR relationships, that goal is split into sub-goals by a pattern below:

  Parent goal: $P1 \vee P2 \Rightarrow Q$

  Subgoal1: $P1 \Rightarrow Q$

  Subgoal2: $P2 \Rightarrow Q$

- The goal reamins the same if its pre-condition is declared with AND relationships.

This step breaks a complex goal into simple goals. Each simple goal is then used as a leaf node goal for a goal tree in step 3. Explanations of this process are described in section 7.7.

3. Generate goal trees from goals obtained from steps 1 and 2. Goal trees generated correspond to KAOS goal refinement patterns, as described in detail in section 7.8.

4. An operation is generated from individual leaf node of goal trees by KAOS operation patterns as described in detail in section 7.10.

7.4 Textual translation rules for generating a goal

This section explains formal translation rules used to transform a TD into a KAOS *Goal model*. There are extra basic translation rules apart from those defined in the direct translation TD to Event-B, chapter 5. Top-level textual translation rules for creating a goal from a segment that has a `CauseEffectArrow` is described in section 7.5, while section 7.6 explains how to generate a goal from a `SimultaneityArrow`.

Basic translation rules

To generate KAOS goal from a TD, some rules are reused from the direct translation (Table 5-1), while others are introduced in this chapter as shown in the table below.

TKEmpty(Timing) → BOOL; this rule checks whether an input Timing exists. If so, the rule gives the Boolean value true.
TKTiming(Constraint) → Timing; this rule gives a Timing for an input Constraint.
TKAllSimul(Machine) → (Simul1, Simul2, …); this rule gives a sequence of SimultaneityArrows for an input Machine.
TKSStartSegm(Simul) → StartSegm; this rule gives the SimultaneityArrow start segment for an input SimultaneityArrow.

Table 7-1 Additional basic rules for TD to KAOS transformation

7.5 Textual translation rules for creating a goal from segments

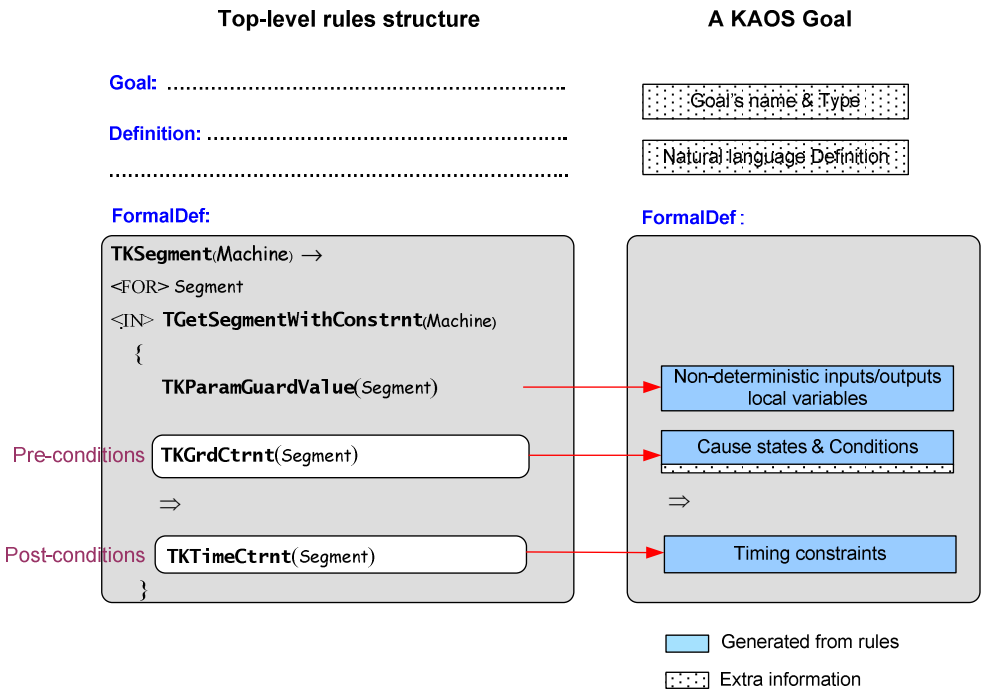


Figure 7-4 Top-level rules structure for creating a goal from a segment

An overview of top-level textual rules used to generate goal formal definitions from segments that have `CauseEffectArrows` is shown in Figure 7-4. In this figure, the coloured boxes represent parts generated from the rules, and hatched boxes represent parts the extra information added for the model completion. See section 7.9 for a discussion of manually added information.

A goal's name and its type have to be generated by hand. A goal formal definition is created by the rule **`TKSegment`** in which a `Machine` is used as a parameter for the rule. This rule generates each goal by the sub-rule **`TGetSegmentWithConstrnt`**, which is reused from chapter 5, Table 5-1. This rule collects only segments that are defined with `CauseEffectArrows` as a sequence. Next, each `Segment` is used to generate other parts of the goal formal definition by other sub-rules. A goal formal definition is composed of three parts: non-deterministic inputs/outputs local variables, pre-conditions, and post-conditions. Each is generated by the sub-rules as explained in the following:

- Non-deterministic inputs/outputs local variables are generated by a goal **`TKParamGuardValue`**. These local variables are used inside a goal. The detail of this rule is shown in Appendix E.
- Pre-conditions that are cause states and conditions are generated from the sub-rule **`TKGrdCtrnt`**. This rule uses a `Segment` as an input parameter.
- A post-condition is generated by the sub-rule **`TKTimeCtrnt`**. This rule uses a `Segment` as an input parameter to generate a post-condition and defines LTL future operator ◊.

The detail of sub-rules and examples are explained in the following section.

7.5.1 *Creating pre-conditions from cause states and conditions*

This section describes the rule **`TKGrdCtrnt`** that is used to generate parts of KAOS pre-condition: cause states and conditions. This rule calls a sub-rule **`TKGetGrdPredc`** as shown in Figure 7-5. The rule **`TKGetGrdPredc`** creates pre-conditions of a goal formal definition for an input `NodeType`.

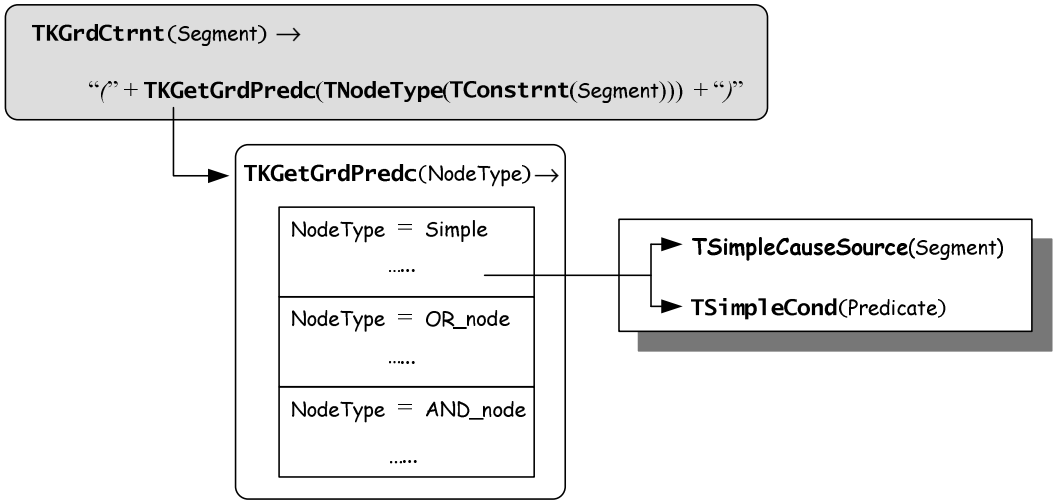


Figure 7-5 Rule: **TKGrdCtrnt** and sub-rules

The rule **TKGetGrdPredc** is similar to the rule **TGetGrdPredc** in section 5.3.6. It checks whether the input NodeType is Simple, OR_node or AND_node. The difference here is, if the NodeType is Simple, two other sub-rules **TSimpleCauseSource** and **TSimpleCond** are called in order to generate guards from cause states and conditions respectively. The rest of this rule is the same as the rule **TGetGrdPredc** in section 5.3.6. That is, if a NodeType is OR_node or AND_node, the rule **TKGetGrdPredc** recursively calls itself. The detail of **TKGetGrdPredc** is illustrated in the following, which shows only part of a Simple node that is different from section 5.3.6.

TKGetGrdPredc(NodeType) .....(1)
<IF> NodeType = Simple .....(2)
<THEN>**TSimpleCauseSource**(**TSegment**(Simple)) .....(3)
+ **TSimpleCond**(**TCond**(Simple)) .....(4)
<ELSE><IF> NodeType = OR_node ..... (5)
…
<ELSE> <IF> NodeType = AND_node ..... (6)
…
<ENDIF>

Figure 7-6 Rule: **TKGetGrdPredc**

The rules **TSimpleCauseSource** and **TSimpleCond** are also reused from the TD direct translation rules in chapter 5. For example, Figure 7-7 shows how the segment *Off2* is used to create a pre-condition by the rule **TKGrdCtrnt**.

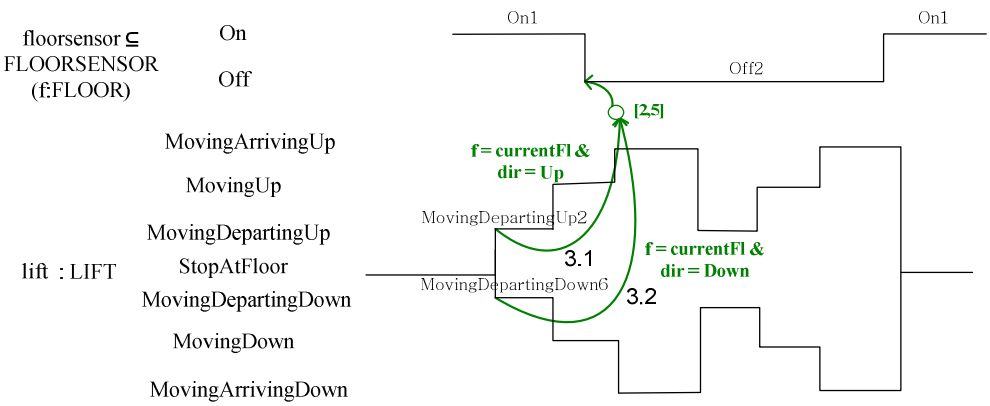


Figure 7-7 Timing diagram for floorsensor and lift (parts of Figure 7.3)

Since the rule **TKGetGrdPredc** is similar to the rule **TGetGrdPredc** in section 5.3.6, we do not repeat how to generate it step by step. Instead, we explain how to generate cause segments and conditions from the rule **TSimpleCauseSource** and **TSimpleCond** (see Figure 7-8).

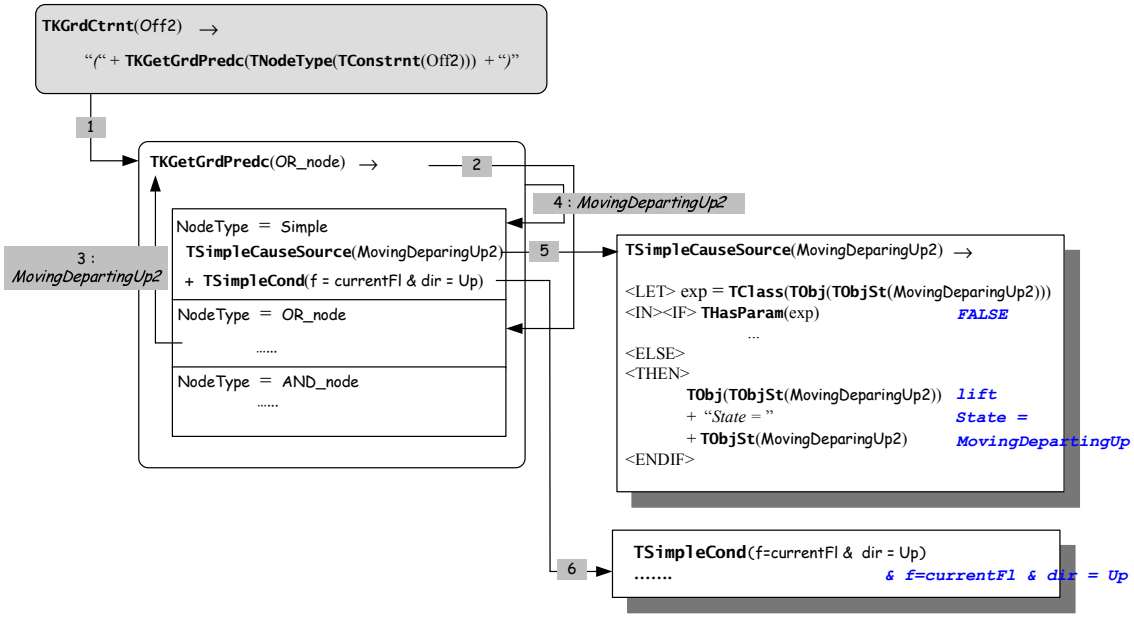


Figure 7-8 Steps for generating pre-conditions for lines 3.1 and 3.2 in Figure 7-7

Note that Figure 7-8 shows only relevant parts of the rule **TSimpleCauseSource** used for creating pre-condition for a segment *Off2* in Figure 7-7. The detailed rules are given in Appendix A.

Steps for generating pre-conditions from cause states and conditions are shown in Figure 7-8. First, the rule **TKGrdCtrnt** is used with the segment *Off2* as the input parameter. At this step, the basic rule **TNodeType** gives the NodeType of the segment *Off2*, which is an OR_node. Next, in step 1, the sub-rule **TKGetGrdPredc** is called, where OR_node is used as an input parameter. Since the NodeType is OR_node, then step 2 is actioned. Note that the details of steps 2-4 were explained in detail in section 5.3.6.

In the OR_node section, each sub-NodeType underneath the OR_node is collected as a sequence. In this case, there are two Simple nodes: MovingDepartingUp2 and MovingDepartingDown6 (see Figure 7-7). The MovingDepartingUp2 is first sent back to the rule **TKGetGrdPredc** as the input parameter as shown in step 3, and then it is sent as Simple node to the Simple node section in step 4.

Step 5 shows the generation of pre-conditions from cause states by the rule **TSimpleCauseSource**, where a Simple node is used as the input parameter. The detail of the cause state generated by the rules is shown in this figure. Step 6 shows a condition, which is attached to the MovingDepartingUp2, is generated to be a part of pre-condition by the rule **TSimpleCond**. The rule **TSimpleCond** simply concatenates each condition if there are many of them. The details of these rules are given in Appendix A.

Next, steps 3-6 are repeated to generate a cause state and conditions for the Simple node, MovingDepartingDown6. The pre-conditions generated from the segment *Off2* are shown below.

Goal ***Achieve****[FIsensorForTheCurrentFloorIsEventuallySetOffW/N2-5secsAfterLift StartsMvgDptUpOrStartsMvgDptDwn]*

Definition: .....

FormalDef:

.....

	From Line
(liftState = 'MovingDepartingUp' & f = currentFl & dir = Up) ∨	3.1
(liftState = 'MovingDepartingDown' & f = currentFl & dir = Down)	3.2

.....

7.5.2 *Creating post-conditions*

As shown in Figure 7-4, the rule **TKTimeCtrnt** is used to create a goal post-condition. The detail of the rule, in which Segment is used as an input parameter, is illustrated in Figure 7-9. This rule calls a sub-rule **TKGetTimingPredc**, where Segment and Timing are input parameters for the rule.

TKTimeCtrnt(Segment) →

{ **TKGetTimingPredc**(Segment, **TKTiming**(**TConstrnt**(Segment))) }

TKGetTimingPredc (Segment, Timing) → ..... (1)

<IF> !**TKEmpty**(Timing) .....(2)

<THEN> "◊" + Timing .....(3)

<ELSE> <SKIP> ..... (4)

<ENDIF> ..... (5)

<LET> exp = **TClass**(**TObj**(**TObjSt**(Segment))) ..... (6)

<IN> ..... (7)

<IF> **THasParam**(exp) ..... (8)

<THEN> ..... (9)

TObj(**TObjSt**(Segment)) ..... (10)

+ "*State(* " ..... (11)

+ **TWriteParamLst**(**TAllParam**(exp)) ..... (12)

+ "*)* = " ..... (13)

+ " ' " + **TObjSt**(Segment) + " ' " ..... (14)

<ELSE> ..... (15)

TObj(**TObjSt**(Segment)) ..... (16)
+ "*State* = " ..... (17)
+ " ' " + **TObjSt**(Segment) + " ' " ..... (18)
<ENDIF> ..... (19)

Figure 7-9 Rules: **TKTimeCtrnt** and **TKGetTimingPredc**

The rule **TKGetTimingPredc** checks whether there is a Timing parameter value, at line (2). If so, it generates a timing constraint in the form of $\Diamond_{[lowerlimit, upperlimit]}$ at line (3). At line (8), the rule checks whether a class corresponding to that segment has a parameter. If so, the rest of a goal post-condition (which is an effect of a `CauseEffectArrow`) is generated at lines (10)-(14); otherwise lines (16)-(18) are exceuted. The sub-rule **TWriteParamLst** is reused, as detailed in section 5.3.8. This rule is used to identify the whole parameter for a class. An example of a goal's post-condition, generated from segment *Off2* by the rule **TKTimeCtrnt**, is shown below.

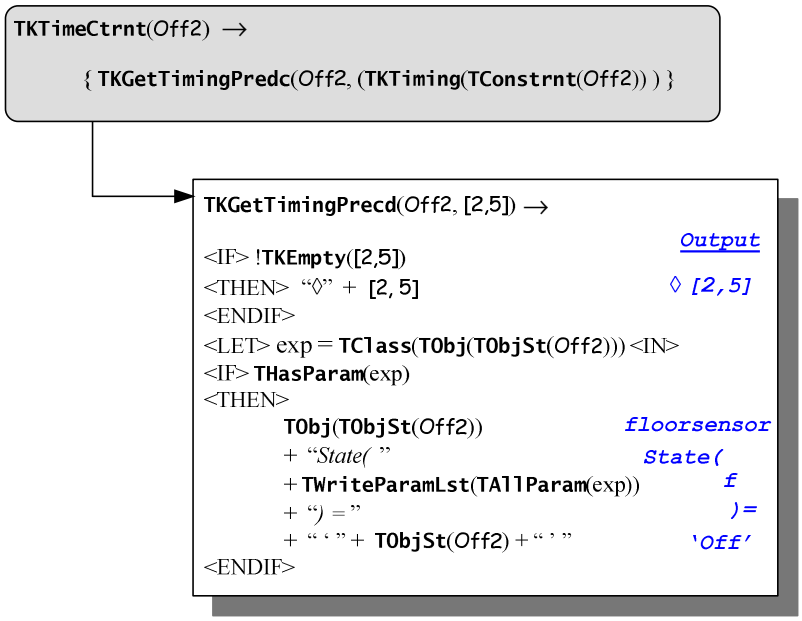


Figure 7-10 Example steps of generating post-conditions for a segment *Off2*

Figure 7-11 illustrates a summary of the segment *Off2* used to generate a *Goal* ***Achieve****[FlsensorForTheCurrentFloorIsEventuallySetOffW/N2-5secsAfterLiftStartsMvgDpt-UpOrStartsMvgDptDwn]*.

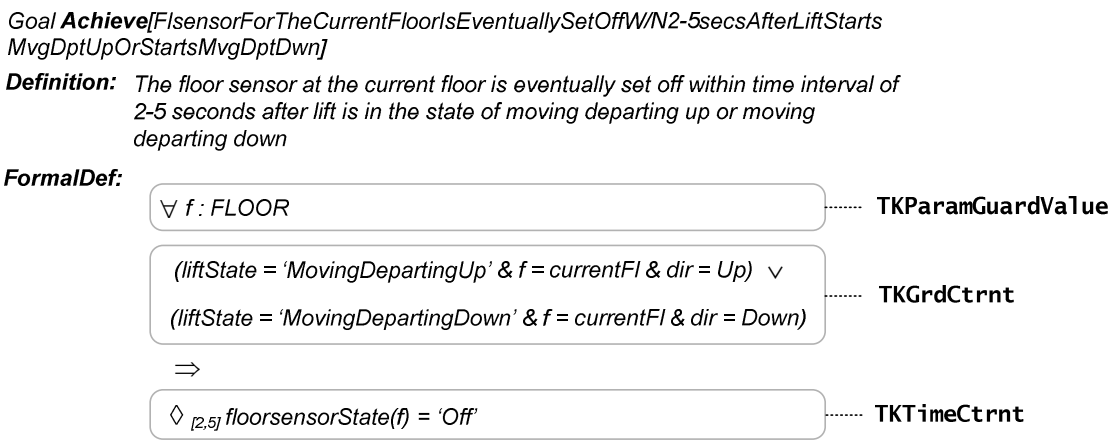


Figure 7-11 A goal 3.1 & 3.2 description

7.6 Top-level textual translation rules for creating a goal from a `SimultaneityArrow`

As mentioned by (Letier, Kramer et al. 2008), in KAOS, a temporal logic pattern `Immediate Response` property can be specified as a goal with the temporal logic $\mathbf{P} \Rightarrow \mathbf{Q}$ where the response **Q** occurs within the same time as the **P** triggering condition. Since `SimultaneityArrow`s are used to show two things happening very close in time (at the level of abstraction), the `SimultaneityArrow` has the same property as the `Immediate Response` property. Thus, each `SimultaneityArrow` is created as an individual goal with this temporal logic pattern. The top-level rules structure for creating a goal from a `SimultaneityArrow` is shown in Figure 7-12. The structure is similar to the top-level rules for creating a goal from a segment. The only difference is that the post-condition is defined without a timing constraint.

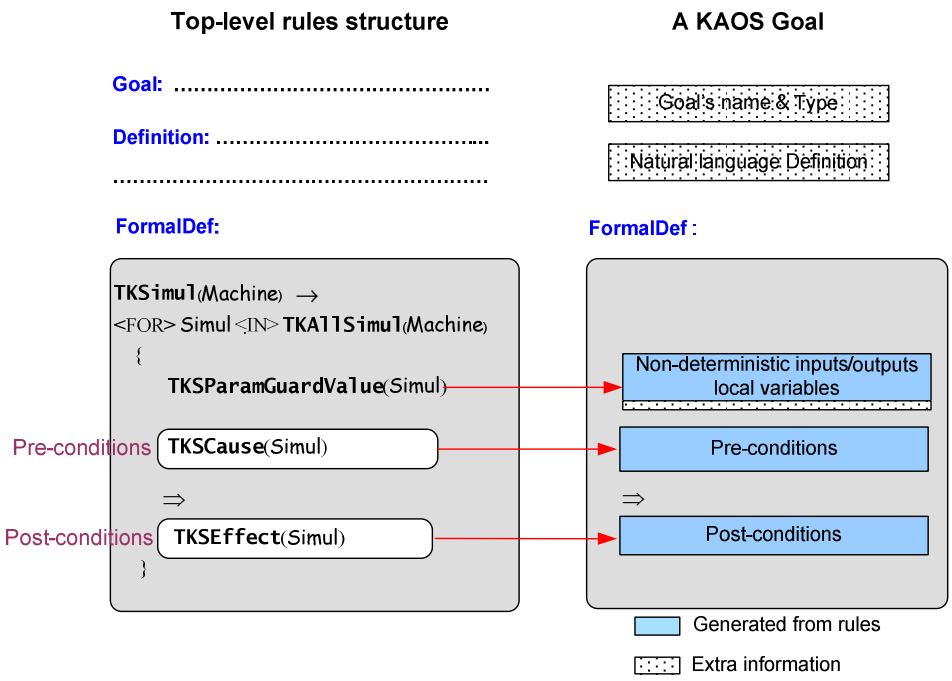


Figure 7-12 Top-level rules structure for creating a goal from `SimultaneityArrows`

A goal formal definition is created by the rule **`TKSimul`** in which *Machine* is used as an input parameter. This rule is defined as an iteration process for generating each `SimultaneityArrow` as a KAOS goal. The sub-rule **`TKAllSimul`** uses *Machine* as an input parameter. The **`TKAllSimul`** rule comprises sub-rules **`TKSParamGuardValue`**, **`TKSCause`** and **`TKSEffect`** for generating a list of non-deterministic parameter using in a goal, goal pre-conditions, and goal post-conditions respectively. There are some goals that are needed to add extra non-deterministic local variables. See section 7.9 for discussion of manually added information. The rule **`TKSParamGuardValue`** is the same as the rule **`TKParamGuardValue`** (Appendix E), the only difference being their input parameters; the **`TKSParamGuardValue`** uses *Simul* while the **`TKParamGuardValue`** uses *Segment*. The rule **`TKSCause`** is the same as the rule **TSimpleCauseSource** (Chapter 5, and detailed in Appendix A); only the input parameters are different.

The post-condition of a goal is generated by the rule **`TKSEffect`** as shown below. This rule does not create timing constraints for a post-condition since a `SimultaneityArrow` does not have timing constraints.

TKSEffect(Simul) →

<LET> exp = **TObj**(**TObjSt**(**TEndSegm**(Simul)))

<IN><IF> **THasParam**(**TClass**(exp))

<THEN> exp

+ "*State(* "

+ **TWriteParamLst**(**TAllParam**(**TClass**(exp)))

+ "*)* = "

+ " ' " + **TObjSt**(**TEndSegm**(Simul)) + " ' "

<ELSE> exp

+ "*State* = "

+ " ' " + **TObjSt**(**TEndSegm**(Simul)) + " ' "

<ENDIF>

Figure 7-13 Rules for creating a KAOS goal from a `SimultaneityArrow`

For example, the post-condition generated from the `SimultaneityArrow` line 16 in Figure 7-2 is shown in the following:

TKSEffect(line 16) →	*Output*
<LET> exp = **TObj**(**TObjSt**(**TEndSegm**(line 16))) …….	*exp = lift*
<IN><IF> **THasParam**(**TClass**(exp)) ……..……..	*FALSE*
…	
<ELSE> exp ……..……	*lift*
+ "*State* = " ……...	*State =*
+ " ' " + **TObjSt**(**TEndSegm**(line 16)) + " ' " ….	*'MovingDown'*
<ENDIF>	

The whole goal formal definition is created from the `SimultaneityArrow` line 16 in Figure 7-2, is illustrated below

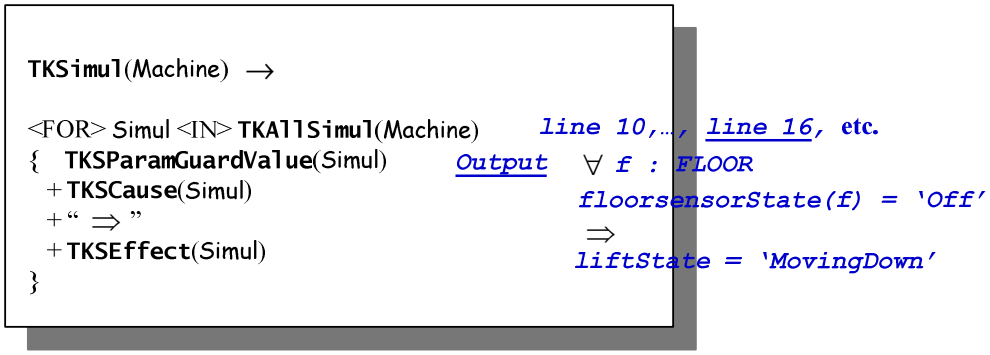


Figure 7-14 The goal formal definition for the `SimultaneityArrow` line 16

7.7 Splitting OR relationships in a goal pre-condition into subgoals

As mentioned in section 7.3, if a goal pre-condition is declared with OR relationships, that goal is split into subgoals. This is an attempt to generate simple goals from a complex goal, in which each of them is used as a leaf node for a goal tree that will be generated later. For example, from Figure 7-11, a *Goal* ***Achieve****[FlsensorForTheCurrentFloorIsEventuallySetOffW/N2-5secsAfterLiftStartsMvgDpt UpOrStartsMvgDptDwn]* has pre-condition defined by an OR relationship. Thus, this goal is split into two subgoals: *Line3.1 Goal* ***Achieve****[FlsensorForTheCurrentFloorIs EventuallySetOffW/N2-5secsAfterLiftStartsMvgDptUp]* and *Line3.2 Goal* ***Achieve****[Flsensor ForTheCurrentFloorIsEventuallySetOffW/N2-5secsAfterLiftStartsMvgDptDwn]* by a pattern: parent goal: **P1** $\vee$ **P2** $\Rightarrow$ **Q**, subgoal1: **P1** $\Rightarrow$ **Q**, and subgoal2: **P2** $\Rightarrow$ **Q** as shown in Figure 7-15.

where: **P1** *: liftState = 'MovingDepartingUp' & f = currentFl & dir = Up*

P2 *: liftState = 'MovingDepartingDown' & f = currentFl & dir = Down*

Q *:* $\Diamond_{[2,5]}$ *floorsensorState(f) = 'Off'*

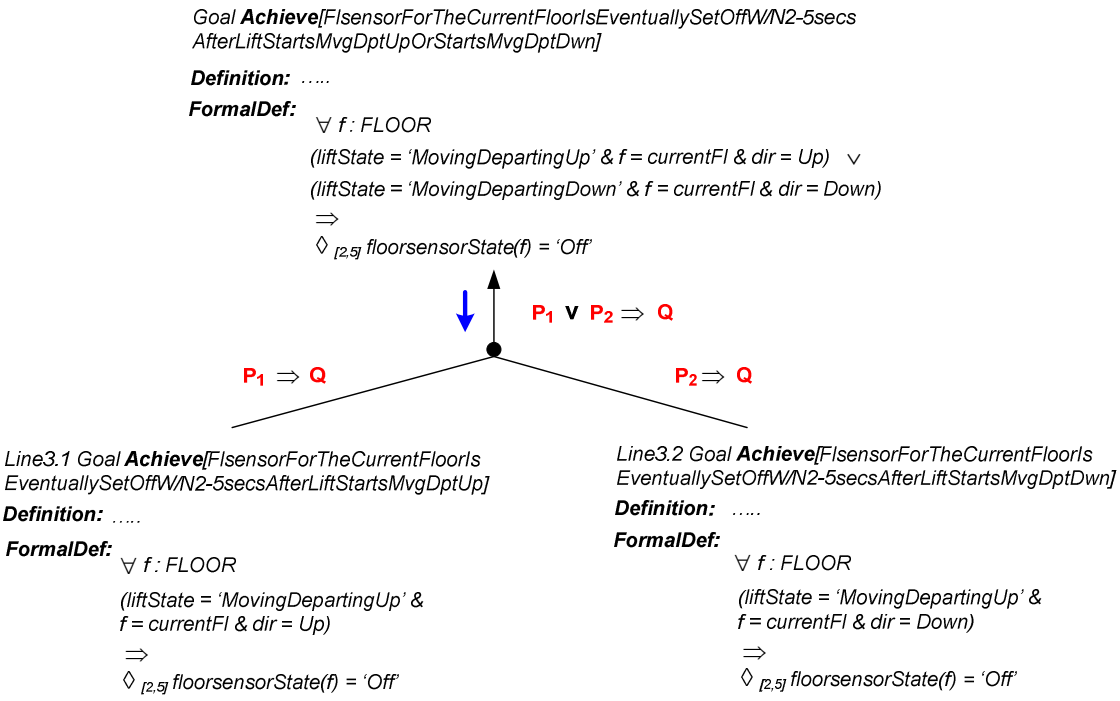


Figure 7-15 Splitting an OR relationship in a goal pre-condition into subgoals

By contrast, any goal that has AND relationships defined in its pre-condition remains the same. An example of the goal generated from line 18 and line 7 in Figure 7-3 by the textual translation rules is illustrated in the following.

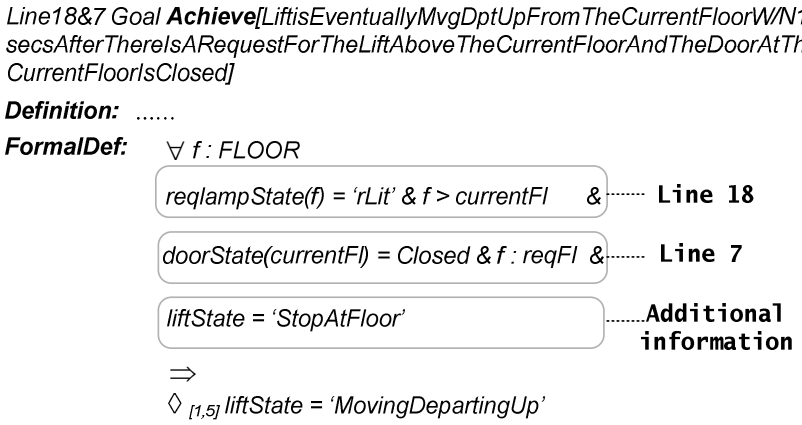


Figure 7-16 An example of AND relationship in a goal pre-condition

In the example in Figure 7-16, where the pre-condition of the goal is combined with an AND relationship, this goal is left the same. However, this goal shows two examples of adding extra information manually.

- First, in some cases, it is necessary to identify the previous states of the post-condition as shown in Figure 7-16, *liftState = 'StopAtFloor'*. That is, the lift must be in a state of stop at floor before it can start moving departing up. In the lift case study, there are four goals: *Line 18&7 Goal*, *Line 19&8 Goal*, *Line5(a) Goal*, and *Line5(b) Goal* that need to be added with this kind of extra information (as detailed in Appendix F).
- Secondly, the original pre-condition generated from line 7 by the translation rules is *doorState(f) = Closed & f : reqFl*. The non-deterministic variable *f* in this pre-condition has to be changed to *currentFl*. This is because we would like to identify the current floor door's state that must be closed, not any other doors. Only two actions need to be altered in the lift case study which are *Line18&7 Goal* and *Line19&8 Goal* (as detailed in Appendix F).

7.8 Generating goal trees

Goals obtained from the steps in sections 7.5 to 7.7 are used to generate goal trees. A goal tree comprises a parent goal and its subgoals. Each sub-goal specifies explicit tasks in which the combination of subgoals explains what to do in general in the upper level, the parent goal. In this thesis, we propose two techniques that are "guidelines" of goal trees generation. In the first technique, a goal tree is created whose subgoals illustrate how changing an object's state causes another object's state to be changed in the system (as detailed in section 7.8.1 below). In the second technique, a goal tree is generated from a group of `CauseEffectArrow`s and `SimultaneityArrow`s that share the same cause segment (as detailed in section 7.8.2 below).

7.8.1 *A goal tree illustrates an object's state change causes another object's state to be changed*

To generate this kind of goal tree, the technique looks for continuity of `CauseEffectArrows` and `SimultaneityArrows` that occurs from the left hand side of the TD to the right hand side. For example, in Figure 7-17, the `CauseEffectArrow` lines 18 and 7 have a MovingDepartingUp2 as an effect segment. This MovingDepartingUp2 becomes a cause segment of the `CauseEffectArrow` line 3.1. The `CauseEffectArrow` line 3.1 has a segment Off2 as an effect segment. This segment Off2 becomes a `StartSegmt` of the `SimultaneityArrow` line 17. Note that at this point, the goal generated originally by lines 3.1 and 3.2 is separated to individual *Line3.1 Goal* and *Line3.2 Goal* already since there is an OR relationship in the pre-condition (as described in section 7.7, Figure 7-15). Thus, *Line3.1 Goal* can be used for this goal tree while the *Line3.2 Goal* is used in another goal tree.

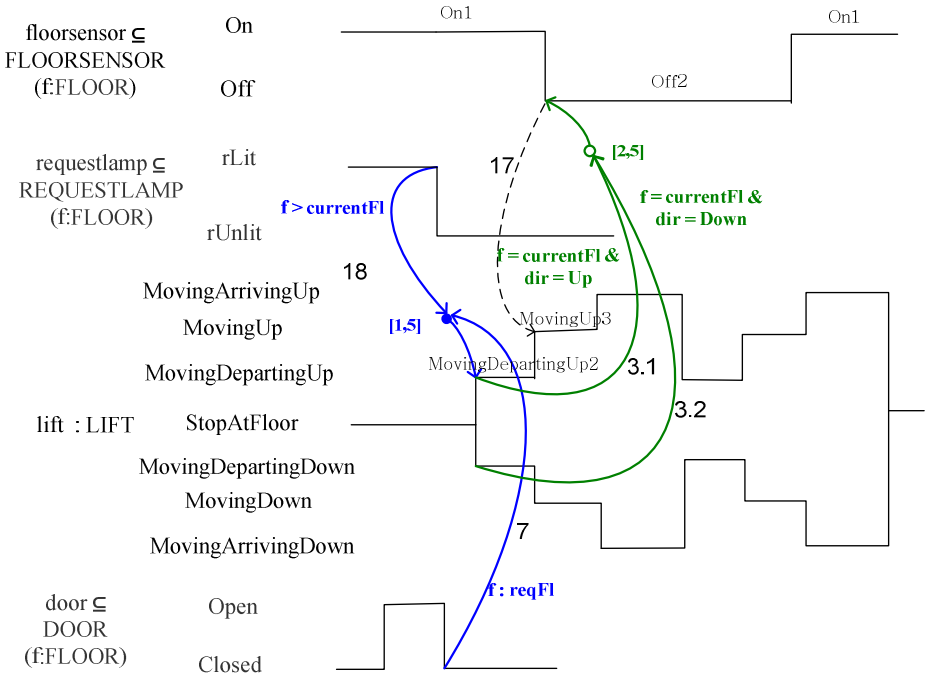


Figure 7-17 The lift timing diagram (parts of Figure 7-3)

The occurrence of the continuity of `CauseEffectArrows` and `Simultaneity` above generates a goal tree by the Milestone-driven goal refinement pattern (detail in section 2.7.4) as shown below.

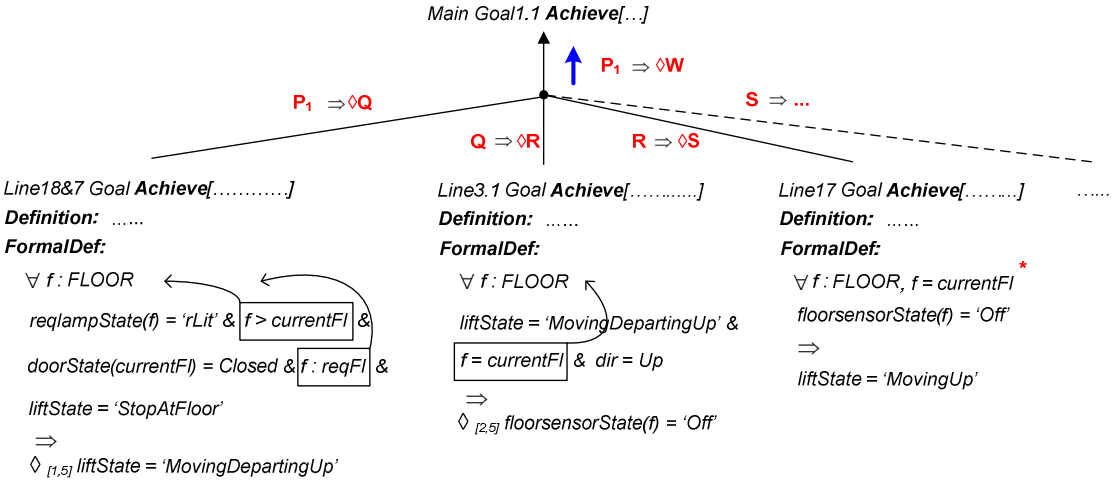


Figure 7-18 Parts of a goal tree

From Figure 7-18, few alterations need to be made in order to have correct goal trees, as described below.

1. Non-deterministic variables' definitions are moved on the top of the tree after the symbol "∀". The move does not change the meaning of a goal, but the rearrangement. For example, moving *f > currentFl* and *f : reqFl* of the *Line18&7 Goal*.
2. Extra information is added. This information is needed only in some line such as *Line17 Goal* and *Line18 Goal*. For example, in Figure 7-18, the *Line17 Goal*, which is generated by the `SimultaneityArrow`, has the added condition *f = currentFl*. Because the notation of `SimultaneityArrow` itself does not allow identifying predicates, additional information is needed in this case.

The goal model after doing these alterations is shown in the following.

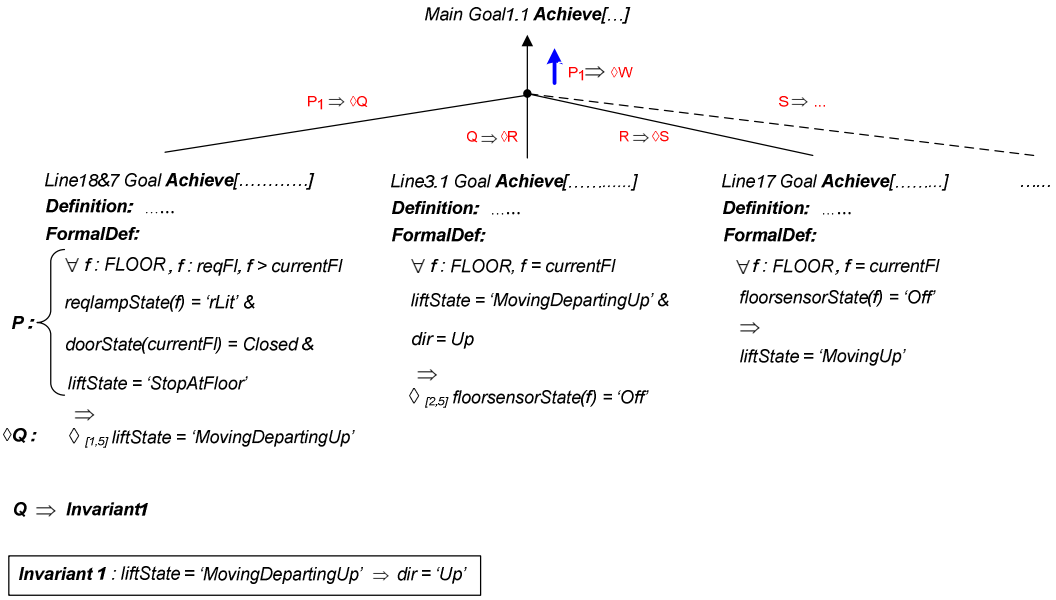


Figure 7-19 Parts of a goal tree after alteration

Line18&7 Goal is linked to the *Line3.1 Goal* by the Milestone-driven goal refinement pattern where *Q* is used as intermediate state. However, if we use *Q* alone, generating this goal tree is not correct. That is, a condition *dir = Up* does not exist for *Q* in the *Line18&7 Goal* but it does exist for *Q* in the *Line3.1 Goal* as a pre-condition. To resolve this problem, an invariant is introduced. Invariants are properties that remain true for a specific sequence of operations in the system. In this case, the *Invariant 1: liftState = 'MovingDepartingUp' ⇒ dir = 'Up'* is used to identify that whenever the lift is in the state of *MovingDepartingUp*, the lift direction must be always *Up*. With this invariant, the goal tree is correctly generated.

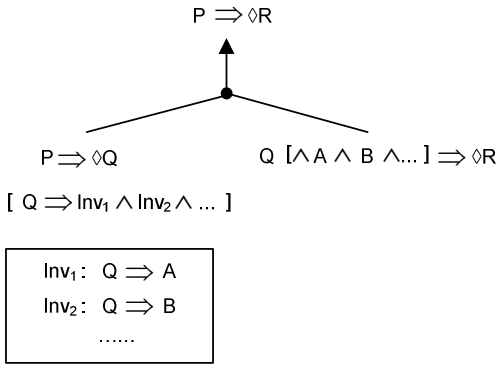


Figure 7-20 A pattern for generating KAOS goal tree

A summary of the Milestone-driven goal refinement pattern used for creating a goal tree, where Inv denotes Invariant is shown in Figure 7-20. An invariant for a goal is an option, which is defined inside the "[...]" symbol. Q is an intermediate state. Q may have invariants defined by Inv1, and Inv2 (and others if there are any) which provides predicates A and B respectively. A and B then are used as a part of pre-condition for the corresponding goal. Not every goal requires an invariant. The invariant is used only when the next goal's pre-condition(s) has extra information that is not identified earlier in the previous goal's post-condition, as in the example shown in Figure 7-19.

7.8.2 *A goal tree is generated from a group of `CauseEffectArrow`s and `SimultaneityArrow`s that share the same cause segment*

This kind of goal tree is generated by looking for a common segment which is used as a cause segment for relevant `CauseEffectArrow`s and `SimultaneityArrow`s. A common segment is generated as a parent goal while the relevant `CauseEffectArrow`s and `SimultaneityArrow`s are created as sub-goals.

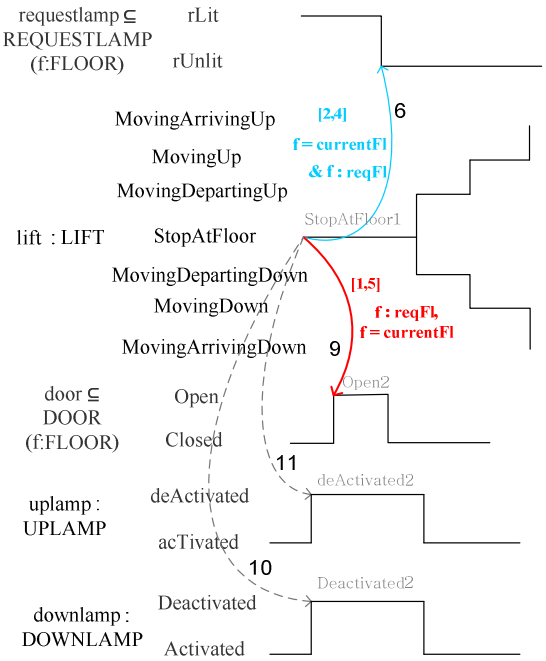


Figure 7-21 Parts of a goal tree representing requestlamp, lift, door, uplamp and downlamp

For example in Figure 7-21, the segment StopAtFloor1 is a common cause segment for the lines 6, 9, 10, and 11. The segment StopAtFloor1 is generated as a parent goal, while each of those lines becomes a sub-goal of the parent goal. Remember, each sub-goal is actually generated earlier by textual translations, as described in sections 7.5 and 7.6. Thus, only the parent goal has to be generated in this step.

This kind of goal tree is generated by the Case-driven : Split consequent pattern (Letier 2001).

Parent goal: $P \Rightarrow Q \wedge R \wedge S$

Subgoal1: $P \Rightarrow Q$

Subgoal2: $P \Rightarrow R$

Subgoal3: $P \Rightarrow S$

Figure 7-22 shows the goal tree generated by those lines and the Split consequent pattern, where

P: $\forall$ *f : FLOOR, f : reqFl, f = currentFl, liftState = 'StopAtFloor'*

Q: $\Diamond_{[2, 4]}$ *requestlampState(f) = 'rUnlit'*

R: $\Diamond_{[1, 5]}$ *doorState(f) = 'Open'*

S: *uplampState = 'deActivated'*

T: *downlampState = 'Deactivated'*

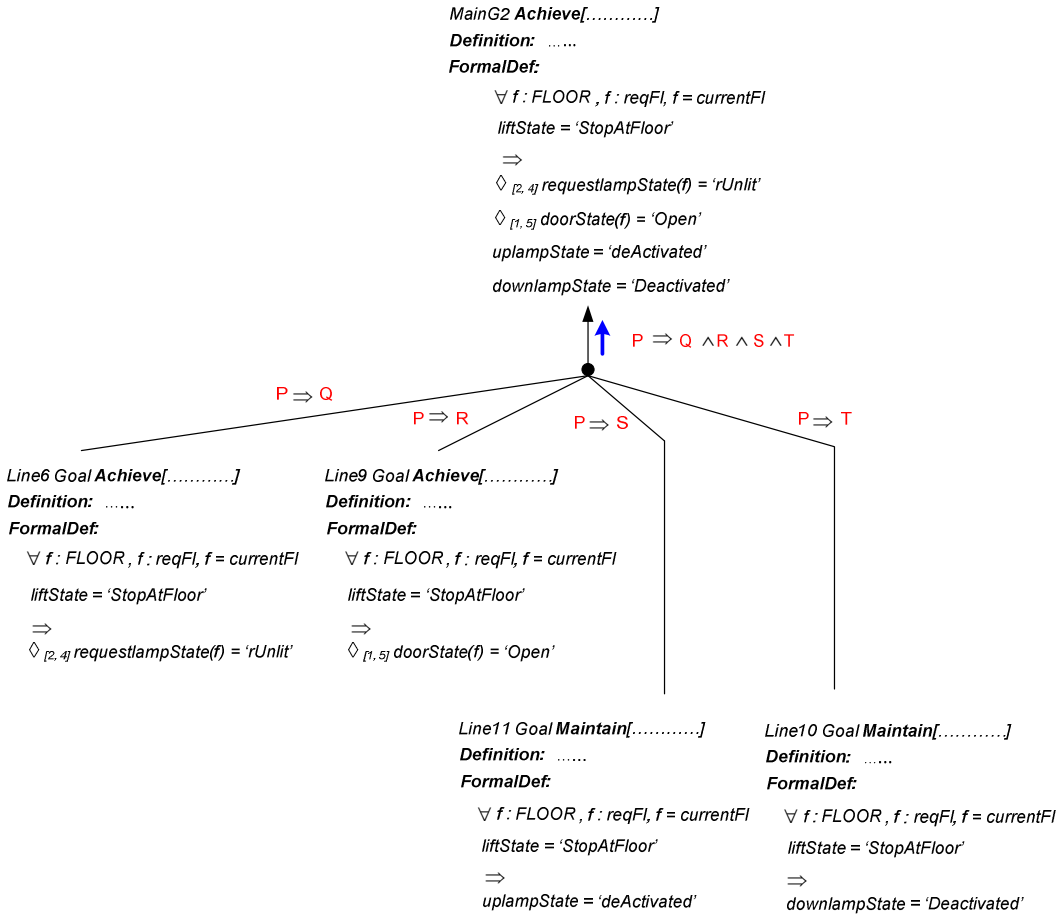


Figure 7-22 A goal tree representing lines 6, 9, 10 and 11 in Figure 7-21

Currently, we have not found it necessary to use invariants in these kinds of the goal tree. The *Goal trees* generated by TD in Figure 7-3 are shown in Appendix F.

7.9 Manual input to modelling

For each goal, the translation rules can automatically generate a goal's formal definition that is composed of non-deterministic local variables, pre-conditions, and post-conditions, where the latter is defined by an LTL operator and timing constraints. The goal's name, goal's type, and goal's textual definition need to be created by hand, because how these parts are described depends on a user's choice. Extra information needs to be added to some goals for two reasons.

1. **To complete the information on the individual goal**

   In a few goals, it is necessary to declare the previous state(s) to the pre-condition of the goal. The previous states make some goals explicit and more correct. However, most goals do not need previous state(s). That is because it is unnecessary to declare previous states, which raises problems when generating *Goal trees* that require more invariants, and creates duplicate information in the *Operation model*, making the *Operation model* complicated. Thus, our design does not create the previous state automatically using the translation rules. An example of adding the previous state is described in Figure 7-16.

2. **To complete goal trees**

   There are two reasons to add extra information. First, `SimultaneityArrow`s will not explain the conditions on the line like `CauseEffectArrow`s. When a *Goal tree* includes any goals that are created from `SimultaneityArrow`s, some extra information may need to be added to the goal to make the *Goal tree* correct. An example of adding extra information for this kind of problem is shown by *Goal Line17* in Figure 7-18. Secondly, to have a complete *Goal tree*, some goals are manually generated. These goals are actually obtained from changing states (transitions) in the TDs, such as the lift is changing state from moving up to moving arriving up. However, these goals are not created by the translation rules, since the rules do not generate a goal from a transition but segments and `SimultaneityArrow`s. These goals are needed since they are used to bridge the gap between the goals inside a *Goal tree*, and make a *Goal tree* complete. An example of introducing a new goal into a *Goal tree* is described below.

Figure 7-23 shows the bigger figure of the goal tree from Figure 7-18. This figure illustrates a *GoalA1* ***Achieve****[LiftStateIsEventuallyMvgArgUpAfterMvgUp]* that is generated by hand. This goal is necessary since it is used to bridge the gap between the *Line17 Goal* and the *Line4.1 Goal*. Note that we used "*A*" after the word "*Goal*" as an abbreviation for the additional goal; for example, *GoalA1* is the additional goal no. 1.

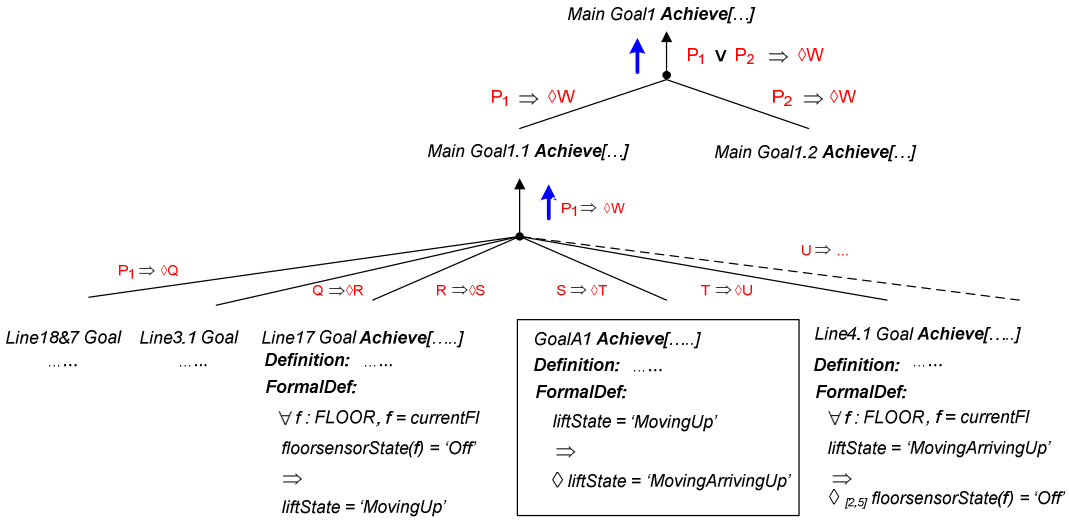


Figure 7-23 The MainG1

In the lift system case study, only two goals are newly generated, *GoalA1* and *GoalA2,* in which the latter describes the lift state as eventually changing its state from *MovingDown* to *MovingArrivingDown*. The detail of this goal can be found in Appendix F.

7.10 Operation model

An *Operation model* defines state transitions of a goal by using `DomPre` and `DomPost` conditions. The `DomPre` is used to describe the state before an operation, while `DomPost` defines a relation between states before and after application of the operation. In addition, further requirements of operations can be defined by using `ReqPre`, `ReqPost`, and `ReqTrig`, as mentioned in section 2.7.5.

An operation is created from a leaf node of goal trees. Thus, an *Operation model* is a collection of operations created from whole leaf nodes. Each goal pattern has a unique operation pattern. For example, goals with a pattern `Bounded Achieve` $P \Rightarrow \Diamond_{\leq d} Q$ and a pattern `Global Invariant` $P \Rightarrow Q$ have operation patterns defined as shown in Figure 7-24. Those operation patterns are well defined

by (Lamsweerde, Dardenne et al. 1991; Letier 2001), here we generate the *Operation model* that follows these patterns.

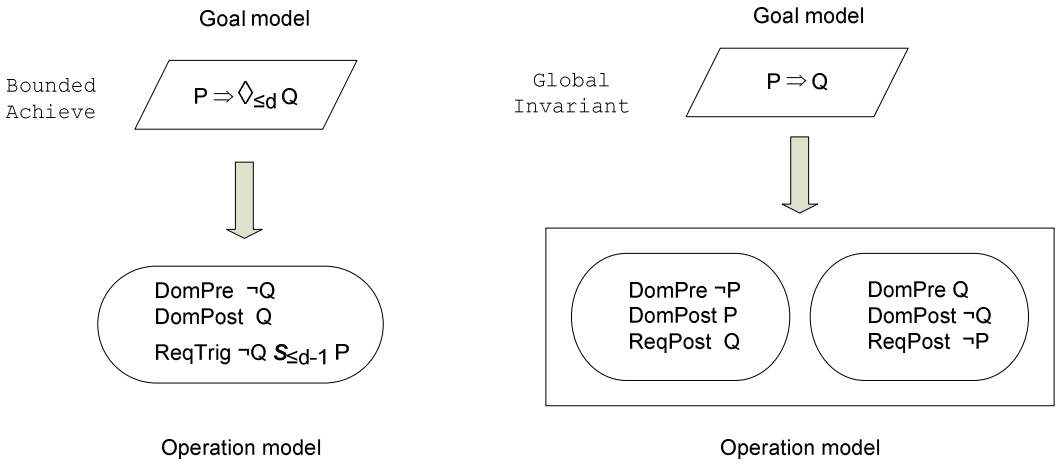


Figure 7-24 Operation patterns: `Bounded Achieve` and `Global Invariant`

For example, consider the *Line9 Goal* below:

Line9 Goal ***Achieve*** *[TheDoorAtTheCurrentFloorIsEventuallyOpenBetween1-5secsAfter LiftStopsAtThatFloor]*

Definition*: The door at the current floor is eventually open between 1 and 5 seconds after the lift is stopped at that floor.*

FormalDef $\forall$ *f : FLOOR, f : reqFl, f = currentFl*
liftState = 'StopAtFloor'
$\Rightarrow$
$\Diamond_{[1, 5]}$ *doorState(f) = 'Open'*

The *Line9 Goal* is declared by the `Bounded Achieve pattern` where **P**: *liftState = 'StopAtFloor'* and **Q**: *doorState(f) = 'Open'*. Thus, its operation is defined as

Operation *DoorOpen*
Input *door{arg f : FLOOR, f : reqFl, f = currentFl }state*
Output *door{arg f : FLOOR, f : reqFl, f = currentFl}state*
DomPre *doorState(f) = 'Close'*
DomPost *doorState(f) = 'Open'*
ReqTrig *doorState(f) = 'Close'* $\mathbf{S}_{[0.99, 4]}$ *(liftState = 'StopAtFloor')*

Another example is the *Line10 Goal*. This goal is defined by the `Global Invariant pattern`. The operation model generated from this goal is illustrated below.

Line10 Goal ***Maintain****[DownlampIsDeactivatedSimultaneouslyWhenLiftStopsAtFloor]*
Definition*: The downlamp is set to deactivate at once whenever the lift stops at that floor*
FormalDef*:*

liftState = 'StopAtFloor'
$\Rightarrow$
downlampState = 'Deactivated'

The *Line10 Goal* is defined as a `Maintain` and corresponds to the `Global Invariant pattern` as shown in Figure 7-24, where **P**: *liftState = 'StopAtFloor' and* **Q**: *downlampState = 'Deactivated'*. The operations for the *Line10 Goal* are defined as follows.

Operation *downlampDeactivated*	***Operation*** *downlampActivated*
Input *liftState*	***Input*** *downlampState*
Output *liftState*	***Output*** *downlampState*
DomPre *liftState ≠ 'StopAtFloor'*	***DomPre*** *downlampState = 'Deactivated'*
DomPost *liftState = 'StopAtFloor'*	***DomPost*** *downlampState = 'Activated'*
ReqPost *downlampState = 'Deactivated'*	***ReqPost*** *liftState ≠ 'StopAtFloor'*

Other operation models can be found in Appendix F.

7.11 Summary

This chapter explains the textual translation rules used to generate KAOS goals from segments defined with `CauseEffectArrow`s and from `SimultaneityArrow`s. The translation rules use TD BNF definitions as input parameters to generate individual goals. The TD BNF definitions for KAOS transformation differs from what was declared in Event-B translation. Here, one timing constraint for each `CauseEffectArrow` is allowed. Creating a goal from nested timing constraints is not supported. Currently, an individual goal is created

by the rules either in a pattern of **Achieve:** `Bounded Achieve P` $\Rightarrow$ $\Diamond_{\leq d}$ `Q` or **Maintain:** `Global Invariant P` $\Rightarrow$ `Q`. Next, those goals are used to create *Goal trees*.

A *Goal tree* can be generated by two techniques. First, the *Goal tree* is generated in which its subgoals illustrate how changing of an object's state causes another object's state to be changed in the system. Secondly, the *Goal tree* is generated from a group of `CauseEffectArrow`s and `SimultaneityArrow`s that share the same cause segment.

For the first technique, some goals need to be declared with invariants. Using invariants, which is an option, enable the creation of a correct goal tree (as described in section 7.8.1). That is because invariants give supportive information that is not directly shown by the goals. The invariants are not used in the second technique.

Some additional goals are added by hand. These goals are introduced into corresponding goal trees in order to complete the goal model (as described in section 7.9). *Operation models* are generated from the leaf node of the goal trees which use well-defined operation patterns, provided by (Lamsweerde, Dardenne et al. 1991; Letier 2001).

Chapter 8 Comparison and Evaluation

This chapter explains the differences and similarities of each technique used to transform TD into Event-B, UML-B and KAOS models. Section 8.1 describes the comparison between Event-B, UML-B and KAOS models. Section 8.2 gives the comparison for the other related works. Section 8.3 provides the evaluation of our model. Section 8.4 gives quantification manual editing while an example of PO is explained in section 8.5.

8.1 Comparison between Event-B, UML-B and KAOS models

Transforming TD into Event-B, UML-B and KAOS models have some things in common and differences in detail.

8.1.1 *Timing diagram notations*

- The same TD notations can be used both for creating an Event-B model from the direct translation rules, and for generating KAOS *Goal* and *Operation models.* That is, the whole of a class's name are defined as uppercase letters. For example, `FLOORSENSOR`.

- The TD used for transforming into UML-B model is a bit different. That is, the first character of a class's name is an uppercase letter and the rest are lowercase letters. For example, `Floorsensor`.
- Each TD class for translating an Event-B model is created as a set in the Event-B **CONTEXT** part, while each TD class for translating an UML-B model is generated as a class in the Event-B **MACHINE** part.

8.1.2 *Identify TD Timing constraints*

- In transforming TD into Event-B and UML-B, defining TD timing constraints is the same. That is, a timing constraint is attached with the Simple NodeType. Then, one can define nested timing constraints for a `CauseEffectArrow`.
- In transforming TD into KAOS, defining TD timing constraints is different. That is, at most one timing constraint for a `CauseEffectArrow` (section 7-1 and 7-2) is allowed. That is because we are not identifying past LTL operators as goal pre-conditions. Using nested timing constraints has to use past operators.

8.1.3 *How models are generated*

- **Metamodel**:
  - In Event-B and KAOS: TD metamodel is defined by BNF definitions
  - In UML-B: TD metamodel is created by EMF
- **Defining timing constraint in a model**
  - In Event-B and UML-B: timing constraints are defining in a pre-condition (guard)
  - In KAOS: timing constraints are defining in a post-condition (action).

8.1.4 *TD components used for the translation*

- In transforming TD into an Event-B model: each segment that is declared with constraints is used to create an Event-B event. If that segment has `SimultaneityArrows` defined, the `SimultaneityArrows` are also generated as a part of that event.
- In transforming TD into UML-B: each TD state transition is used to generate an Event-B event.
- In transforming TD into KAOS: each segment that is declared with constraints and `SimultaneityArrows` are used separately to create a goal.

8.1.5 *Ease of production and amendment*

- To generate an Event-B model: the difficult part is generating TD BNF definitions that should represent TD correctly and can be used as closely as possible for the rest of the translation techniques. Textual translation rules use BNF elements as input parameters. Most Event-B components can be generated from the rules and altering a model is easy to do.
- To generate a UML-B model: it takes a lot of effort to generate a model starting from creating the TD metamodel and source model using Eclipse, and using the UML-B toolkit since it needs a high specification computer. Using ATL has problems as it does not support creating an output element by combination of source elements across the rules. Moreover, the UML-B itself does not fully support generating `SimultaneityArrows` nor identifying multiple previous states to the same target state. The output model needs to be altered such as adding associations to classes since TD notation does not support this.
- To generate KAOS *Goal* and *Operation models*: the TD BNF from the direct translation of an Event-B model can be reused with some modifications, as well as the textual translation rules. The hardest part

for the KAOS translation is generating goal trees since they need to be created with the KAOS refinement patterns. Generating a KAOS *Operation model* uses the pattern provided at the leaf nodes of goal trees.

8.1.6 *Manual additional information*

- **Context**:
  - In Event-B: most of the context elements are generated from TD by the textual translation rules, only a few have to be created manually. Those manual creation elements are actually defined as predicates on the `CauseEffectArrows` but they cannot be used to generate context since TD notations do not support this.
  - In UML-B: since the limitations of ATL, the ATL rules can generate the context's name while the body of the context must be generated by hand.
  - In KAOS: there is no concept of context.

- **Events/Goal**
  - In Event-B: some events are necessary added manually. That is because TD expresses only a part of the whole system specifications. Moreover, each event is generated by two TD notations: segments with constraints (`CauseEffectArrows`) and `SimultaneityArrows` attached to the segment. However, not every event can be represented by `CauseEffectArrows` and `SimultaneityArrows`. Thus, some events need to be added. For example, in the lift case study, we have to add events: *ChangeDirUp*, *ChangeDirDown* and *doorClosed*.
  - In UML-B: there are fewer events manually appended since every transition is generated to be an event. However, there is more alteration in the UML-B model than in the Event-B model generated

by the direct translation. That is for two reasons: first, the limitation of ATL itself. Secondly, to generate some variables used in the model, associations among classes need to be generated, which cannot be done directly from TD notations, but must be by hand.

- In KAOS: a number of goals need to be added manually. That is because each goal is generated by two TD notations: segment with constraints and `SimultaneityArrows`. However, not every system specification can be represented by these notations. Thus, some goals need to be appended. We find what goal is missing and needs to be added, while generating a goal tree. For example, the goal that describes changing the state of the lift from moving up to moving arriving up.

- **Variables**
  - In Event-B and UML-B model: variables are added manually for the same reasons described above. Some of these variables are actually defined as a part of predicate, some are not. However, since none of the TD notations can be used to identify these kinds of variables, they have to be defined by hand. For example, in the lift case study, we have to add variables *currentFl* and *dir* to represent the current position of the lift and lift direction respectively. These variables are defined as machine variables.
  - In KAOS: there are no variables to be added.

8.1.7 *Invariants*

- In Event-B and UML-B: invariants are used to maintain some properties that remain true for a specific sequence of operations of the system.
  - In Event-B: invariants are defined by hand within the **MACHINE** part **INVARIANTS**.

  - In UML-B: invariants are defined within the **MACHINE** part by hand. They can be defined as machine invariants or class invariants.
- In KAOS: invariants are used with the same propose and identified at some points of a goal tree by hand. Using invariants in a goal tree is useful because they provide the supportive information that is needed for generating a correct goal tree.

8.1.8 *Controlling time progress: Ticktok event*

- In Event-B and UML-B: a *Ticktok* event is generated for the purpose of controlling time progress.
- There is no *Ticktok* event created in KAOS.

8.1.9 *Easy to Understand*

- For an Event-B model: the Event-B model output is simple to understand for someone who has knowledge of Event-B.
- For UML-B model: UML-B has specific keywords such as *Self* and uses special symbols such as "." to refer to attributes of a class. Thus, time may be needed for developers/users at the beginning to understand these symbols before generating a model. The advantage of using UML-B is its graphical user interface; thus it is easy for users to figure out where to add the missing information to the model.
- For KAOS model: since defining KAOS looks similar to declaring an event in Event-B, creating a KAOS goal is adapted from what is done in Event-B. The KAOS output goals are not difficult to understand since there is a textual definition for each goal to explain what the goal aims for. The formal definition for the goal elaborates the goal by using temporal logic operators, which currently is only the operator ◊ (eventually).

8.1.10 *Capturing all requirements*

TDs are best used to describe the behaviour of functional requirements with causal dependencies between objects and timing constraints. However, TDs are not suitable for use with some kinds of requirements, for example, non-functional requirements. Even though TDs can capture the functional requirements as described above, in generating Event-B, UML-B and KAOS models there needs to be some extra information added, as described in section 8.1.6.

8.2 Comparison with other related works

Some groups have investigated cause/effect relationships and timing constraints. For example, (Abrial, 2008b) introduces patterns for state-based specifications in Event-B. The patterns are useful for our research. They can, however, illustrate only cause/effect relationships, not timing constraints. (Cansell, *et al.*, 2007) introduces timing constraints pattern for distributed applications. A number of groups combined UML and B such as (Ledang and Souquierès, 2002a), who investigated a combination of B-Method with Class diagram and State diagram, while (Jiufu, 2007) has proposed translating statechart diagrams into B; (Younes and Ayed, 2007) focuses on the translation of Activity diagrams into Event-B; (Idani and Ledru, 2007) propose systematic transformation rules to generate a Class diagram from a B specification. Our work is unique in providing techniques to create timing constraints from a TD to an Event-B model.

There is a work by (Bicarregui, *et al.*, 2008) to extend Event-B notations to three LTL operators: *Next* (○), *Eventually* (◊) and *Bounded eventually* ($\leq_n$) where n denotes time units. The work proposes using three new constructs that are to replace the standard Event-B structure, **WHEN**…**THEN**…**END**, that are **WHEN**…**NEXT**…**END**, **WHEN**…**EVENTUALLY**…**END** and **WHEN**…**WITHIN**...**NEXT**…**END** to represent the three LTL operators *Next*, *Eventually* and *Bounded eventually* respectively. We have approached this in a

different way, as we are generating timing constraints in Event-B model by using the standard Event-B notations provided.

(Aziz*, et al.*, 2009) captures three KAOS *Goal model* patterns: *Immediate achieve*, *Eventually/Unbound achieve*, and *Bounded achieve* to represent three new constructs as proposed by (Bicarregui*, et al.*, 2008) above.

Apart from our early work in (Joochim and Poppleton, 2007) that investigates how to generate KAOS goal trees from TD, there are a number of investigations that explore possible techniques for translating KAOS framework to other models. For example, (Letier*, et al.*, 2008) proposes a technique to translate KAOS Operation models to Labelled Transition Systems (LTS). The LTS is Statemachine-like diagram; it is a group of components in which each component is defined by a set of states and transitions, where each transition is labelled by an event. (Landtsheer*, et al.*, 2004) investigates translating KAOS *Operation models* into event-based tabular specifications, which describe system requirements through a set of tables. Some attempts to combine KAOS with B are introduced by (Ponsard and Dieul, 2006) who try to generate B operations from KAOS operations. However, this work only focuses on traceability links. Other work has been done by (Hassan*, et al.*, 2009) to transform KAOS *Operation model* to B specification language in security requirements, unlike our work, which attempts to generate KAOS *Goal model* and *Operation model* from TD.

A variety of versions of the lift case study are used in many papers such as (Dardenne*, et al.*, 1991), who explain how to generate KAOS goals, agents and operations for a simple lift case study. Some of those lift specifications are functional requirements, as ours is, but no timing constraints are involved. A number of the specifications identify human activities such as “passenger out of elevator when at destination floor”, which we do not deal with in our research. Research by (Choppy and Reggio, 2005) represents a combination of Problem frames and UML diagrams (Use case, Class, and State diagrams) by using a lift system case study. This paper shows how to define a lift system in a class diagram and a state diagram with a fewer number of components than our work, and with no timing constraints involved. The classical B machine which represents a lift

control system found in (Abrial, 1996) is the most similar model to ours that shows how to represent the lift specifications by B method. However, this case study also has a fewer number of objects than within our case study and has no timing constraints.

There exist TD editors such as TimeGen (Intel), TimingTool (MOHC, 2009), and SynaptiCAD (SynaptiCAD, 2009). However, these editors do not fit with our research since they are defined with different types from our TD, and are not written on the Eclipse framework. Thus, they could not easily fit with RODIN and UML-B.

8.3 Evaluation

8.3.1 *Tool validation*

The output of our translation can be automatically validated by the RODIN tools. B prover is an automatic proof of correctness of implementation relative to high level specifications. It also does syntax checking for a model. ProB performs consistency checking (finding deadlocks and invariant violations) and animation. The validation detail for each model is shown below:

For an Event-B model from the direct translation: We used RODIN Platform 0.9.1 for creating an Event-B model obtained from the direct translation rule. The Event-B model is verified by RODIN toolkit for proof obligations (POs) and syntax checking while a RODIN plugin, ProB 1.1.0, is used for consistency checking (find deadlocks and invariant violations) and animation. We also used ProB 1.2.6 (which is a separated tool from the RODIN toolkit) for model re-checking and verifying deadlock freeness. The result of validation is: Total POs: 135, Auto discharged: 122, Manual discharged: 11, Reviewed: 2 and Undischarged: 0.

For an Event-B model generated from a UML-B model: The UML-B 0.4.3 is used for generating a model obtained from ATL, RODIN Platform 0.9.1 is used for POs. A RODIN plugin, ProB 1.1.0, is used for consistency checking and animation. The result of validation is: Total POs: 142, Auto discharged: 54, Manual discharged: 84, Reviewed: 4 and Undischarged: 0. The number of POs auto discharged in the UML-B model is fewer than in the Event-B model and manual discharged is more because the UML-B model comprises a large number of transitions and classes. Moreover, the way to define guards and invariants by combining many associations among classes makes it harder to prove than in the direct translation.

During the process of improving the translation tools, we have had to rework proofs many times. As the work progressed, the number of automatically proved obligations slightly increases while the number of manually proved obligations increases a lot.

KAOS: there is a tool for *Goal model* verification (Rifaut*, et al.*, 2003). However, to use it one needs to be trained abroad.

8.3.2 *Validation of the correctness of the transformations defined*

Currently, we use a lift as only one case study. The lift case study has many objects and shows various kinds of timing constraints, and simultaneous and causal dependecies in a reactive requirements system. However, it is needed to have other case studies to ensure the correctness of the transformation defined. The purpose is to check whether our TD notaions cover other kinds of requirements. The other case studies should have different kinds of casual dependencies and timing constraints from the lift system. Morover, it is necessary to validate the transformation rules are correct and complete. To do so, we should to provide incorrect/incomplete input models to inspect whether the translation rules generate an incorrect output model. This task is considered as further work.

8.4 Quantification manual editing

The Event-B, UML-B and KAOS output models are needed to be manually modified in order to make the models complete. The quantification of how much manual editing is needed for each model shown in the following.

Event-B : 108 modified to 450 lines generated (24%)

UML-B : 162 modified to 557 lines generated (29.08%)

KAOS: 8 modified to 32 leaf node goals generated (2.50%)

How to make the tools fully automated is explained in the following.

8.4.1 *Event-B*

For the additional information that cannot be identified by the TD notations itself (e.g. identifying the number of floors), we have nothing to do with the rules in such this case.

For the information that already have in the model -e.g. variables *currentFl* and *dir*- but we cannot generate to Event-B, we may create a new TD notation to support identifying variables at the `CauseEffectArrow`s' conditions. Thus, model variables can be directly generated from those `CauseEffectArrow`s' conditions. Moreover, the `SimultaneityArrow`s should be identified by a combination of OR nodes (see the example problem in section 5.4).

For some extra events added, we may alter the rules to generate an Event-B model from the TD state transitions instead of using TD segments as what we have done.

Currently, the Event-B output model is generated as text. Users have to copy the text to RODIN tool again. Thus, to make the tool more efficiently, the Event-B output model should automatically be generated in the RODIN tool.

8.4.2 *UML-B*

The ways to correct the UML-B model is the same as those described for the Event-B model above. However, the limitations of ATL and UML-B cause some

parts of UML-B output models have to be manually generated. The further step of fulfilling the TD to UML-B translation rules is to revise UML-B tool to support identifying TD multiple previous states of the same target state and `SimultaneityArrow`s.

8.4.3 *KAOS*

The same ways used in the Event-B model are also used to have complete KAOS Goal models. The problem only found in KAOS is, in some goals, it is needed to declare conditions on the `SimultaneityArrow`s. Thus, a new notation for the `SimultaneityArrow`s to identify conditions is introduced. The conditions are optional and used as guards for the goals.

8.5 Example of proof obligations

This section shows an example of how the *invariant preservation statement* (**INV**), as described in section 2.3.2, is used for the PO. Consider an event `floorsensorOffUp` which is obtained from the UML-B model as shown in the following:

```
MACHINE L
…
INVARIANTS
      Invariant1: ∀d·((d∈Door)⇒(lift_state ≠ StopAtFloor ⇒
                  door_state(d) = Closed))
…
EVENTS floorsensorOffUp
    ANY FloorsensorSelf
        f
    WHERE
      Guard1: f ∈ FLOOR
      Guard2: FloorsensorSelf ∈ Floorsensor
      Guard3: floorsensor_state(FloorsensorSelf) = On
      Guard4: lift_state = MovingDepartingUp
```

```
    Guard5: (gclock – liftMovingDepartingUpTime ≥ 2)
            ∧(gclock – liftMovingDepartingUpTime ≤ 5)
            ∧ f = currentFl ∧ dir = Up
THEN
  Action1: lift_state ≔ MovingUp
   …
END
```

According to the consistency proofs as described in section 2.3.2, the corresponding parts of the machine are used in the **INV** proof obligation for the event `floorsensorOffUp` as shown in the following. This PO is named automatically by the RODIN prover as `floorsensorOffUp/Inv1/INV`. Notice that `Guard5` is separated into individual guards for the proof as shown below:

Hypothesis	**Axioms**	-
	Invariant1	∀d·((d∈Door)⇒(lift_state ≠ StopAtFloor ⇒ door_state(d) = Closed))
	Guard1	f ∈ FLOOR
	Guard2	FloorsensorSelf ∈ Floorsensor
	Guard3	floorsensor_state(FloorsensorSelf) = On
	Guard4	lift_state = MovingDepartingUp
	Guard5	gclock – liftMovingDepartingUpTime ≥ 2
	Guard6	gclock – liftMovingDepartingUpTime ≤ 5
	Guard7	f = currentFl
	Guard8	dir = Up
	Before-after predicate of the event (BA)	lift_state = MovingUp
	⇒	
Goal	**Modified Specific Invariant (Im)**	∀d·((d∈Door)⇒(MovingUp ≠ StopAtFloor ⇒ door_state(d) = Closed))

As shown above, a proof obligation comprises two parts: a hypothesis, and a goal; shown by the elements before and after the $\Rightarrow$ symbol respectively. In this case, this goal is to prove that after the action: `lift_state = MovingUp` (which is represented by **BA**) is performed, the **Invariant1** is still preserved. That is, a goal is generated by assigning a state `MovingUp` in **BA** to the `lift_state` in **Invariant1** (as highlighted). This goal is proved interactively by the *Predicate Prover (PP)* in the *Proof Control* panel as shown in Figure 2-7.

Chapter 9 Contribution and Limitations

The contribution of the thesis is showing how to formalise specification of systems that contain causal dependencies with timing constraints, in Event-B and KAOS by using TDs. As a result, we propose systematic translation rules to transform TD into Event-B, UML-B and KAOS *Goal models*.

This chapter declares benefits and contributions to research we have done in section 9.1 and section 9.2 respectively. The limitation of the work is demonstrated in section 9.3, and future directions are stated in section 9.4.

9.1 Benefits

According to the research goals in section 1.3, the first two goals to generate translation techniques to transform a TD to Event-B, UML-B and KAOS were accomplished. The benefit of our contribution is providing another option to generate timing constraints and causal dependencies requirements of a reactive system to Event-B, UML-B and KAOS Goals by using graphical visualisation, TD. Thus, instead of manually generating these targets model (Event-B, UML-B and KAOS Goal model) in a textual form, users can use the TD as a graphical front-end, and these target models are created automatically. Moreover, in Event-B and UML-B, we provided a pattern to generate events' pre- and post-conditions

that concern with timing constraints, and an event (Ticktok) to control time progression. Having the timing constraints guard and the time progress patterns decrease the time required in considering how to model the time from the beginning.

For KAOS, apart from having the translation rules to automatically generate KAOS goals from TD, we also provided guidelines to generate KAOS Goal trees from TD. These guidelines assist users to generate KAOS parent goals from sub-goals. Along the parent goal/sub-goal creating process, some goals may be introduced. Thus, it helps users to find incomplete information that may be left since from the requirements elicitation processes.

The third goal in section 1.3, evaluating the use of TD to specify timing constraints and casual dependencies requirements in Event-B compare with using textual one has not been done due to limited time.

9.2 Contribution

We produced a model - in four different forms - on a real time case study: a lift system.

1. TD based TD UML 2.0 diagram notations
2. Event-B
3. UML-B
4. KAOS *Goal* and *Operation models*

Our contribution can be identified as the following:

1. We propose bridging the gap between graphical requirements notations (TD) and declarative FM (Event-B). We provide a technique to generate Event-B from an existing tool UML-B from TD. This is another contribution of generating Event-B models from graphical notation TD. Both model generated from 1 and 2 can be proved correct by RODIN tools.
2. Since KAOS Goal models explain timing constraints by linear temporal logics (LTLs) which are in textual forms, we present a technique to generate

KAOS goals' formal definition by TD which it represents as graphical requirements.

3. We provide multiple views of one system's requirements by expressing them in TD, Event-B, UML-B and KAOS models.

The detail of each contribution is described in the following sections.

9.2.1 *Requirements to TD*

We used TD which is based on the (OMG, 2007) Robust TD notations for capturing the requirements of a system. A subset of TD notations was selected and some notations were justified to make it easy to generate Event-B, UML-B and KAOS *Goal models*. Those TD notations are essential to identify causal dependencies between objects and their combinations. TD classes were generated from objects in requirements that have causal dependency between them. One can define timing constraints, conditions that make states of objects change, and simultaneous events, by TD notations. The selected TD notations have abilities to model other systems that can be described with time constraints.

9.2.2 *TD to Event-B Translation*

We produced rules for translating systematically. We created an Event-B model from TD. In doing this, first, we identified TD BNF definitions to describe individual TD notations. Next, we created formal translation rules to transform TD into a textual Event-B model, where the TD BNF definitions are used as input parameters for the translation.

- The translation rules create sets, constants and axioms in a **CONTEXT** part. For a **MACHINE** part, the rules can create variables and their initial values, invariants, events and a `Ticktok` event, of which the latter is used for time progress. For each event, the translation rules can create an event's name, non-deterministic local variables (if there are any), events' guards and actions.

- Other parts that cannot be identified by TD such as additional variables, events and invariants, need to be created by hand. For example, lift changing direction and guards for ticking the clock. The detail of generating Event-B to TD is explained in Chapter 5.

9.2.3 *TD to UML-B Translation*

Since TD represents partial system requirements, to generate a complete Event-B model, one needs to use other non-timing requirements. In doing that, one may add those requirements directly to an Event-B model as in the previous translation or use another model, e.g. Class diagram and Statecharts. To make it convenient for users and to integrate TD with an existing tool, we have implemented systematic translation to provide part of an automatic translation system from TD using UML-B.

We generated transition rules to transform TD to UML-B by using ATL. In doing that, the TD metamodel is created on Eclipse and used to create a case study as example for a source model; an existing UML-B metamodel is used as a target model.

- The rules can generate a **CONTEXT** (without detail inside due to the limitation of ATL) and a **MACHINE** part. In the **MACHINE** part, the rules generate classes, class attributes and their types, Statemachines, some machine variables and a `Ticktok` event. In a Statemachines, the rules generate states, state transitions with names, parameters with their types, guards and actions.
- Other parts, such as detail inside the **CONTEXT**, invariants, additional variables and events, are created by hand since they cannot be identified by TD notations.

9.2.4 *TD to KAOS Translation*

The third approach was adopted because other relevant research tries to combine KAOS and B, but does not deal with integrating requirements in which

there are timing constraints and causal dependencies between objects to KAOS. Our research has been done in a different way, in which we use TD information to generate KAOS *Goal model*.

- We generate translation rules that use TD BNF definitions as input parameters for creating an individual KAOS goal, focusing on goal's formal definition. The rules create each goal's formal definition, a goal's name, and type, while its textual definition is created manually. Next, those goals are formalised and grouped by KAOS goal refinement patterns to generate a goal tree by our proposed techniques. Invariants are used in some points of the goal tree in order to fulfil the goal tree refinement pattern and additional goals are added by hand in this step.
- An operation is generated from each leaf node goal of goal trees by KAOS goal refinement patterns provided by (Letier, 2001).

9.3 Limitations

9.3.1 *General limitations*

At the moment, the TD can generate partial Event-B machines both from the direct translation rules and UML-B as well as partial KAOS *Goal* and *Operation models*. However, the TD has not been designed to collect whole system requirements. Therefore, some information needs to be added in these models.

Another constraint is the original UML TD 2.0 and our TD notations still cannot be used to demonstrate human actions. There are many requirements concerned with human activities, for example in the lift system that needs human intervention to request the lift by pressing buttons. In this case, we can demonstrate the pressing activity by representing it as an event in Event-B, but cannot control human pressing activity time. For a clearer example, there is the case study of the Ambulance Service system in (Letier, 2001), which is used to generate a KAOS model. The Ambulance Service system has many timing constraints; one of them is responding to emergency calls requiring the rapid intervention of an ambulance.

That requirement has to deal with calling by operators. It is a good case study for KAOS, but not for TD, since we cannot guarantee the correctness of a model depending on human activities.

9.3.2 *Timing diagram notations and tool limitations*

For the limitation of the TD itself, one cannot identify a `SimultaneityArrow` with a combination OR node. For example, Figure 7-3, lines 16 and 17, are used to identity whenever a *floor sensor* is set *off*, once the *lift* is in a state of *moving up* or *moving down*. Those lines are represented by `SimultaneityArrows` since there is no timing constraint concerned. Not having a combination node causes a problem whenever generated by an Event-B model (as describe in section 5.4). That is, an event action is generated in which there are two conflicting actions within the same event. This has to be resolved by separating them into different events manually. The UML-B tool also has the limitation that cannot fully support generating `SimultaneityArrows` (as shown in Figure 6-43). Another weakness is, currently, there is no TD editor. Thus, sometime it takes a lot effort to create and to alter TD manually while using EMF.

9.3.3 *KAOS translation limitation*

At present, KAOS translation has a limitation of not dealing with timing constraints that have occurred in the past states. This issue is considered to be a future work.

9.4 Future directions

Some future directions are suggested as follows.

1. We found that from the lift case study, sometimes, it is necessary to identify combination of OR nodes and constraints for the

`SimultaneityArrows`. Thus, TD `SimultaneityArrows` should be appended by these properties.

2. In UML-B translation, `Timeline Transitions` names may be identified. Defining `Timeline Transitions` names would help creating events' name easier. Instead of the events' name being generated by a combination of many elements, it is defined directly from the `Timeline Transitions`.
3. Eliminate the manual addition of information which can be generated from the TD.
4. Include past operators in the KAOS goal models to cover other applications that may have to use them.
5. At the moment TD is created by using EMF. Thus, having a graphical front end for TD is a way for creating and modifying a TD model easily.
6. In a case where enormous system requirements with timing constraints are concerned, it is better to generate a TD for each subsystem and integrate the TDs to form a whole system. The future work is to find techniques to combine those TD subsystems.
7. Identify refinement steps in the Event-B model. For example, in the lift case study, the abstract model has basic lift behaviour while the timing constraints are introduced in the refinement steps.
8. Investigate a technique to transfer KAOS *Goal* and *Operational models* to an Event-B model.
9. More case studies to ensure the toolset techniques are sufficiently general and robust.

References

Abrial, J.-R. (1996). *The B-book : Assigning Programs to Meanings*, Cambridge University Press.

Abrial, J.-R. (2005a). *Formal Method Course*. Retrieved 26 April 2005.

Abrial, J.-R. (2005b). *Using Design Patterns in Formal Developments*. In Proceedings of the Refinement Workshop (REFINE 2005), University of Manchester, UK, Elsevier.

Abrial, J.-R. (2006). *Formal Methods in Industry: Achievements, Problems, Future*. In Proceedings of the 28th International Conference on Software Engineering (ICSE'06), Shanghai, China, ACM.

Abrial, J.-R. (2007). Formal Methods : Theory Becoming Practice, *Journal of Universal Computer Science* 13(5): 619-628.

Abrial, J.-R. (2008a). *Summary of Event-B Proof Obligations*. Retrieved 29 April 2009, Available from http://www.cs.man.ac.uk/~banach/COMP60110.Info/CourseSlides/Slides.6up.0903ProofObs.pdf.

Abrial, J.-R. (2008b). *Tutorial - Case study of a complete reactive system in Event-B: A mechanical press controller*. In Proceedings of the 5th International Symposium on Formal Methods (FM'2008), Turku, Finland, Springer, LNCS 5014.

Abrial, J.-R., Arief, B., Butler, M., Coleman, J., Iliasov, A., Johnson, I., Jones, C., Khomenko, V., Koutny, M., Laibinis, L., Leppanen, S., Lecomte, T., Leuschel, M., Oliver, I., Razali, R., Rezazadesh, A., Romanaovsky, A., Snook, C., Troubitsyna, E., Voisin, L., and Warwick, J. (2007). *RODIN Assessment Report 3*, Deliverable D34 (D7.4), RODIN.

Abrial, J.-R., Butler, M., Hallerstede, S., and Voisin, L. (2008). *A Roadmap for the Rodin Toolset*. In Proceedings of the 1st International Conference on Abstract State Machines, B and Z, London, UK, Springer-Verlag, LNCS 5238.

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

Abrial, J.-R., Hallerstede, S., Metha, F., Metayer, C., and Voisin, L. (2005). Specification of Basic Tools and Platform. *RODIN Deliverable D10*.

Abrial, J.-R., and Hoang, T. S. (2008). *Using Design Patterns in Formal Methods: an Event-B Approach*. In Proceedings of the 5th International Colloquium : Theoretical Aspects of Computing (ICTAC 2008), Istanbul, Turkey, Springer-Verlag.

Agerholm, S., and Larsen, P. G. (1998). *A Lightweight Approach to Formal Methods*. In Proceedings of the International Workshop on Current Trends in Applied Formal Methods, Boppard, Germany, Springer-Verlag.

Allemand, M., C. Attiogbé, et al. (2002). *SHE'S Project*. A report of join workshops on the 2nd International Workshop on Integration of Specification Techniques for Applications in Engineering (INT'02), Grenoble, France.

Allilaire, F., and Idrissi, T. (2004). *ADT : Eclipse development tools for ATL*. In Proceedings of the 2nd European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA-2), Canterbury, UK, Computing Laboratory, University of Kent.

Ambler, S. W. (2004). *The Object Primer: Agile Model Driven Development with UML 2*, Cambridge University Press.

Anwer, S., and Ikram, N. (2006). *Goal Oriented Requirement Engineering: A Critical Study of Techniques*. In Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC'06), Bangalore, India, IEEE Xplore.

Attiogbé, C., P. Poizat, et al. (2003). *Integration of Formal Datatypes within State Diagrams*. In Proceeding of the European Joint Conferences on Theory and Practice of Software, Warsaw, Poland, LNCS.

ATL (2008). *ATLAS Transformation Language*. Retrieved 20 April 2008, Available from http://www.eclipse.org/m2m/atl/.

ATLAS Group, L. a. I. (2008). *ATL : Atlas Transformation Language ATL User Manual - Version 0.7*. Retrieved 11 Febuary 2008, Available from http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf.

Aziz, B., Arenas, A. E., Bicarregui, J., Ponsard, C., and Massonet, P. (2009). *From Goal-Oriented Requirements to Event-B Specifications*. In Proceedings of the 1st NASA Formal Methods Symposium, Moffett Field, California, USA, Deploy-Project ePrint.

Barland, I., Greiner, J., and Vardi, M. (2006). *Using Temporal Logic to Specify Properties*. Retrieved 3 July 2006, Available from http://cnx.org/content/m1231/latest.

Bashar, N., and Easterbrook, S. (2000). *Requirement Engineering: A Roadmap*. In Proceedings of the Conference on the The Future of Software Engineering, Limerick, Ireland, ACM.

Becker-Kornstaedt, U., H. Neu, et al. (2001). *Software Process Technology Transfer: Using a Formal Process Notation to Capture a Software Process*

in Industry. In Proceeding of the 8th European Workshop:software Process Technology, Germany, Springer Berlin.

Berthomieu, B. and M. Diaz (1991). "Modeling and Verification of Time Dependent Systems Using Timed Petri Nets." *IEEE Transactions on Software Engineering* 17(3): 259-273.

Bicarregui, J., Arenas, A., Aziz, B., Massonet, P., and Ponsard, C. (2008). *Towards Modelling Obligations in Event-B*. In Proceedings of the International Conference of ASM, B and Z Users, London, UK, Springer, LNCS 5238.

Bicarregui, J. C., Clutterbuck, D. L., Finnie, G., Haughton, H., Lano, K., Lesan, H., Marsh, D. W. R. M., Matthews, B. M., Moulding, M. R., Newton, A. R., Ritchie, B., Rushton, T. G. A., and Scharbach, P. N. (1997). Formal methods into practice: case studies in the application of the B method, *Software Engineering* 144(2): 119-133.

Bolognesi, T., and Brinksma, E. (1987). Introduction to the ISO specification language LOTOS, *Computer Networks and ISDN Systems* 14(1): 25-59.

Booch, G., Rumbaugh, J., and Jacobson, I. (2003). *The Unified Modeling Language User Guide*, Pearson Education.

Bowen, J. P., and Hinchey, M. G. (2006). Ten Commandments of Formal Methods …Ten Years Later, *Computer* 39(1): 40-48.

Brisolara, L. B. d., M. E. Kreutz, et al. (2009). *UML as Front-End Language for Embedded Systems Design*, IGI Global.

Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., and Grose, T. J. (2003a). *Eclipse Modeling Framework*, Addison-Wesley Professional.

Budinsky, F., Steinberg, D., Merks, E., Raymond, Ellersick, and Grose., T. (2003b). *EclipseModeling Framework*, Addison Wesley Professional.

Butler, M. (2000). csp2B : A Practical Approach to Combining CSP and B, *Formal Aspects of Computing* 12(3): 182-196.

Butler, M., Abrial, J.-R., Damchoom, K., and Edmunds, A. (2008). *Applying Event-B and Rodin to the filestore (Invited paper)*. In Proceedings of the ABZ 2008, London, UK, ASRNet.

Butler, M., and Hallerstede, S. (2007). *The Rodin Formal Modelling Tool*. In Proceedings of the BCS-FACS Christmas 2007 Workshop - Formal Methods In Industry, London, United Kingdom, BCS.

Butler, M., Leuschel, M., and Snook, C. (2005a). *Combining CSP and B for Specificaiton and Property Verification*. In Proceedings of the Formal Methods 2005, Newcastle Upon Tyne, England, Springer, LNCS 3582.

Butler, M., Leuschel, M., and Snook, C. (2005b). *Tools for system validation with B abstract machines (Invited papers)*. In Proceedings of the 12th International Workshop on Abstract State Machines (ASM 2005), Paris, France, Laboratory of Algorithm, Complexity and Logic.

Cansell, D., Méry, D., and Rehm, J. (2007). *Time Constraint Patterns for Event B Development*. In Proceedings of the Formal Specification and Development in B, 7th International Conference of B (B 2007), Besancon, France, Springer, LCNS 4355.

Cassez, F. and O.-H. Roux (2005). "Structural Translation from Time Petri Nets to Timed Automata." *Electronic Notes in Theoretical Computer Science* 128(6): 145-160.

Cerone, A. and A. Maggiolo-Schettini (1999). "Time-based expressivity of time Petri nets for system specification." *Theoretical Computer Science* 216(1-2): 1-53.

Chen, P. P.-S. (1976). The Entity-Relationship Model - Toward a Unified View of Data, *ACM Transactions on Database Systems (TODS)* 1(1): 9-36.

Choppy, C., and Reggio, G. (2005). A UML-based approach for problem frame oriented software development, *Information and Software Technology* 47(14): 929-954.

Chung, L. (1993). Representing and Using Non-Functional Requirements: A Process-Oriented Approach. PhD from *Department of Computer Science*, University of Toronto.

ClearSy (2009). *Atelier-B*. Retrieved 19 October 2008, Available from http://www.atelierb.eu/index_en.html.

Cobden, M., Humphreys, B., Macarthur, K., and O'Neill, B. (2007). *Timing Diagram Plugin Support for RODIN/UML-B*, A group design project report, Department of Electronics and Computer Science, University of Southampton.

Cox, K., J. G. Hall, et al. (2005). "Editorial: A roadmap of problem frames research." *Information and Software Technology* 47(14): 891-902.

Dardenne, A., Fickas, S., and Lamsweerde, A. v. (1991). *Goal-directed Concept Acquisition in Requirements Elicitation*. In Proceedings of the 6th International Workshop on Software Specification and Design, Como, Italy, IEEE.

Dardenne, A., Lamsweerde, A. v., and Fickas, S. (1993). Goal-Directed Requirements Acquisition, *Science of Computer Programming* 20(1-2): 3-50.

Darimont, R. (1995). Process Support for Requirments Elaboration. PhD from *Dépt. Ingénierie Informatique*, Université Catholique de Louvain.

Darimont, R., and Lamsweerde, A. v. (1996). Formal Refinement Patterns for Goal-Driven Requirements Elaboration, *ACM SIGSOFT Software Engineering Notes* 21(6): 179-190.

Dehbonei, B., and Mejia, F. (1995). Formal development of safety-critical software systems in railway signalling. *Applications of Formal Methods*. M. G. Hinchey and J. P. Bowen, Prentice-Hall**:** 227–252.

Eclipse (2008). *Eclipse Modeling Framework Project (EMF)*. Retrieved 2 September 2008, Available from http://www.eclipse.org/modeling/emf/.

El-Maddah, I., and Maibaum, T. (2003). *Goal-Oriented Requirements Analysis for Process Control Systems Design*. In Proceedings of the Formal Methods and Models for Co-Design (MEMOCODE'03), Mont Saint-Michel, France, IEEE Computer Society.

EMFT-Eclipse (2009). *Eclipse Modeling Framework Technology (EMFT)*. Retrieved 9 March 2009, Available from http://www.eclipse.org/modeling/emft/?project=ecoretools.

Event-B.org (2008). *B2Latex*. Retrieved 17 November 2008, Available from http://www.event-b.org/plugins.html.

Event-B.org (2009). *Rodin Platform Installation*. Retrieved 15 Febuary 2009, Available from http://www.event-b.org/platform.html.

FAUST (2008). *An Overview of the FAUST Toolbox*. Retrieved 20 November 2008, Available from http://faust.cetic.be.

Fisler, K. (2006). Towards Diagrammability and Efficiency in Event Sequence Language, *International Journal on Software Tools for Technology (STTT)* 8(4): 431-447.

Fisman, D., and Eisner, C. (2009). *Sugar 2.0 Formal Specification Language*. Retrieved 17 April 2009, Available from www.haifa.ibm.com/projects/verification/sugar/images/sugar2_sv-ac.ppt.

Fitzgerald, J., Larsen, P. G., Mukherjee, P., Plat, N., and Verhoef, M. (2004). *Validated Designs for Object-oriented Systems*, Springer.

Fowler, M., and Scott, K. (2004). *UML Distilled: A Brief Guide to The Standard Object Modelling Language*, Addison-Wesley Professional.

Friedental, S., and Steiner, R. (2004). *System Modeling Language (SysML) Overview*. In Proceedings of the NDIA System Engineering.

Gavras, A. (2003). "Considerations on telecom modelling languages." Retrieved 7 October, 2009, Available from http://www.modatel.org/~Modatel/pub/deliverables/D3.add2-final.pdf.

George, V. and R. Vaughn (2003). "Application of Lightweight Formal Methods in Requirement Engineering1." CrossTalk-The Journal of Defense Software Engineering(Jan).

Guttag, J. V., Horning, J. J., Garland, S. J., Jones, K. D., Modet, A., and Wing, J. M. (1993). *Larch : Language and Tools for Formal Specification*, Springer-Verlag.

Hall, A. (2007). Realising the Benefits of Formal Methods, *Formal Methods and Software Engineering*: 1-4.

Hallerstede, S. (2006). *Justifications for the Event-B Modelling Notation*. In Proceedings of the Formal Specification and Development in B (B 2007), Besancon, France, Springer, LNCS 4533.

Hassan, R., Bohner, S., El-Kassas, S., and Hinchey, M. (2009). *Integrating formal analysis and design to preserve security properties*. In Proceedings of the 42nd Hawaii International Conference on System Sciences (HICSS-42), Waikoloa, Hawaii, USA, IEEE Computer Society.

Hause, M., Thom, F., and Moore, A. (2005). Inside SysML, *Computing & Control Engineering* 16(4): 10-15.

Heaven, W., and Finkelstein, A. (2004). A UML profile to support requirements engineering with KAOS, *Software Engineering* 151(1): 10-27.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*, Prentice-Hall International Series In Computer Science.

Hoare, J., Dick, J., Neilson, D., and Sørensen, I. (1996). *Applying the B technologies on CICS*. In Proceedings of the 3rd International Symposium of Formal Methods Europe (FME'96), Oxford, United Kingdom, Springer-Verlag.

Hozmann, G. J. (1997). The model checker SPIN, *IEEE Transactions on Software Engineering* 23(5): 275-295.

Hull, E., Jackson, K., and Dick, J. (2004). *Requirements Engineering*, Springer.

IBM (2008). *Sugar 2.0*, Available from http://www.eetimes.com/news/design/showArticle.jhtml?articleID=16504943.

Idani, A., and Ledru, Y. (2007). Object oriented concepts identification from formal B specifications *Formal Methods in System Design* 3: 233-247.

Intel. Retrieved 2 June 2009, Available from "NEW" http://www.xfusionsoftware.com/.

Jackson, M. (1995). *Software Requirements and Specifications : A Lexicon of Practice, Principles and Prejudices*, Addison-Wesley.

Jackson, M. (2001). *Problem Frames Analysis and Structuring Software Development Problems*, Addison-Wesley.

Jackson, M. (2005). "Problem Frames and Software Engineering." *Information & Software Technology* 47(14): 903-912.

Jayaratchagan, N. (2004). *Declarative Programming in Java*, Available from http://www.onjava.com/pub/a/onjava/2004/04/21/declarative.html.

Jiufu, L. (2007). Integration of statechart and B method based analysis and verification for flight control software of unmanned aerial vehicle, *ACM SIGSOFT Software Engineering Notes* 32(2): 1-4.

Jones, C. B. (1986). *Systematic Software Development Using VDM*, Prentice Hall.

Joochim, T., and Poppleton, M. R. (2007). *Transforming Timing Diagrams into Knowledge Acquisition in Automated Specification*. In Proceedings of the 2nd International Conference on Advance in Information Technology (IAIT2007), Bangkok, Thailand, King Mongkut's University of Technology.

Joochim, T. at. el. (2010). *Timing Diagrams Requirements Modeling using Event-B Formal Methods*. In Proceedings of the Software Engineering (SE 2010), Innsbruck, Austria, Actapress.

Jureta, I. (2006). *Engineering Requirement for Information Systems using KASO and Request frameworks*. Retrieved 22 JaNaury 2009, Available from http://www.isys.ucl.ac.be/staff/stephane/GETI2100Slide/KAOS.pdf.

Khan, M. U., Geihs, K., Gutbordt, F., Gohner, P., and Trauter, R. (2006). *Model-Driven Development of Real-Time Systems with UML 2.0 and C*. In Proceedings of the Joint Meeting of the Fourth on Model-Based Development Computer-Based Systems and The Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, Postdam, Germany, IEEE Computer Society.

King, S., Hammond, J., Chapman, R., and Pryor, A. (2000). Is Proof More Cost-Effective Than Testing?, *IEEE Transactions on Software Engineering* 26(8): 675-686.

Langari, Z. and A. B. Pidduck (2005). *Quality, Cleanroom and Formal Methods*. International Conference on Software Engineering, the third workshop on Software quality St Louis, Missouri, USA, ACM.

Lamsweerde, A. v. (2000). *Formal Specification : a Roadmap*. In Proceedings of the Future of Software Engineering Track (ICSE' 00), Limerick, Ireland, ACM.

Lamsweerde, A. v. (2001). *Goal-Oriented Requirements Engineering: A Guide Tour*. In Proceedings of the 5th IEEE International Symposium on

Requirements Engineering (RE'01), Toronto, Canada, IEEE Computer Society.

Lamsweerde, A. v. (2004). *Goal-Oriented Requirement Engineering : A Roundtrip from Research to Practice*. In Proceedings of the 12th IEEE Joint International Requirements Engineering Conference (RE'04), Kyoto, Japan, IEEE Xplore.

Lamsweerde, A. v. (2009). *Requirements Engineering : From System Goals to UML Models to Software Specifications*, John Wiley & Son.

Lamsweerde, A. v., Dardenne, A., Delcourt, B., and Dubisy, F. (1991). *The KAOS Project: Knowledge acquisition in automated specifications of software*. In Proceedings of the AAAI Spring Symposium series, Symposium: Design of Composite Systems, Stanford University, California, USA, AI Magazine.

Lamsweerde, A. v., and Massonet, R. D. P. (1995). *Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt*. In Proceedings of the 2nd IEEE International Symposium on Requirements Engineering, York, England, IEEE Computer Society.

Lamsweerde, A. v., and Willemet, L. (1998). Inferring Declarative Requirements Specifications from Operational Scenarios, *IEEE Transactions on Software Engineering* 24(12): 1089-1114.

Landtsheer, R. D., Letier, E., and Lamsweerde, A. v. (2004). Deriving tabular event-based specifications from goal-oriented requirements models, *Requirements Engineering* 9(2): 104-120.

Ledang, H., and Souquierès, J. (2002a). *Contributions for Modelling UML State-Charts in B*. In Proceedings of the 3rd International Conference on Integrated Formal Methods, Turku, Finland, Springer, LNCS 2335.

Ledang, H., and Souquierès, J. (2002b). *Integration of UML Views using B Notations*. In Proceedings of the Workshop on Integration and Transformation of UML models (WITUML'02), Malaga, Spain.

LeMieux, D. H. (2003). On-Line Termal Barrier coating Monitoring for Real-time Failure Protection and Life Maximization, U.S. Department of Energy, National Enery Technology Laboratory**:** 1-15.

Letier, E. (2001). Reasoning about Agents in Goal-Oriented Requirement Engineering. PhD Thesis from *Dépt. Ingénierie Informatique*, Universite Catholique de Louvain Belgium.

Letier, E., Kramer, J., Magee, J., and Uchitel, S. (2008). Deriving Event-Based Transition Systems from Goal-Oriented Requirements Models, *Automated Software Engineering* 15(2): 175-206.

Letier, E., and Lamsweerde, A. v. (2002a). *Agent-Based Tactics for Goal-Oriented Requirements Elaboration*. In Proceedings of the 24th International Conference on Software Engineering (ICSE'02), Orlando, Florida, USA, ACM.

Letier, E., and Lamsweerde, A. v. (2002b). *Deriving Operational Software Specifications from System Goals*. In Proceedings of the 10th International Symposium on the Foundation of Software Engineering (FSE 2002), USA, ACM, Vol. 27.

Leuschel, M. (2007). *ProB*. In Proceedings of the RODIN Industry Day, Paris, France, CLEARSY.

Leuschel, M., and Butler, M. (2005). *Automatic Refinement Checking for B*. In Proceedings of the 7th International Conference on Formal Engineering Methods (ICFEM'05), Manchester, UK, Springer, LNCS 3785.

Liu, J., P. H. Chou, et al. (2001). *Power-Aware Scheduling under Timing Constraints and Slack Analysis for Mission-Critical Embedded Systems*. 38th Design Automation Conference, Las Vegas, NV, USA.

Matoussi, A., Gervais, F., and Laleau, R. (2008). *A First Attempt to Express KAOS Refinement Patterns with Event B*. In Proceedings of the 1st International Conference on Abstract State MAchine, B and Z (ABZ 2008), London, UK, Springer-Verlag, LNCS 5238.

Métayer, C., Abrial, J.-R., and Voisin, L. (2005). *Event-B language*. Retrieved 15 March 2009, Available from http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf.

Métayer, C., and Voisin, L. (2007). *The Event-B Mathematical Language*. Retrieved 2 October 2008, Available from http://www.labri.fr/perso/casteran/FM/Rodin/mathLanguage-2007-10-26.pdf.

MOHC (2009). *TimingTool*. Retrieved 10 June 2009, Available from "NEW" http://www.timingtool.com/.

Moore, A. (2006, 1 May 2006). *SysML Effort About to Bear Fruit*. Retrieved 7 March 2009, Available from http://www.sdtimes.com/content/article.aspx?ArticleID=29301.

Nakagawa, H., Taguchi, K., and Honiden, S. (2007). *Formal specification generator for KAOS: model transformation approach to generate formal specifications from KAOS requirements models*. In Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering, Atlanta, Georgia, USA, ACM.

OMG-MOF (2007). *Meta Object Facility (MOF) specification*, 12 May 2009, Available from http://www.omg.org/mof/.

OMG (2007). *UML Superstructure Specification, v2.0*. Retrieved 22 Janury 2009, Available from http://www.omg.org/cgi-bin/doc?formal/05-07-04. .

OMG (2008). *UML 2.0*. Retrieved 5 August 2008, Available from http://www.uml.org/#UML2.0.

Oshiro, K., Watahiki, K., and Saeki, M. (2003). *Goal-Oriented Idea Generation Method for Requirements Elicitation*. In Proceedings of the 11th IEEE International Conference on Requirements Engineering, California, USA, IEEE Computer Society.

Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*, Prentice Hall.

Petre, M. (1995). Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming, *Communications of the ACM* 38(6): 33-44.

Pfleeger, S. L. (1998). *Software Engineering Theory and Practice*, POrentice Hall.

Ponsard, C., and Dieul, E. (2006). *From Requirements Models to Formal Specifications in B*. In Proceedings of the International Workshop on Regulations Modelling and their Validation and Verification (REMO2V'06), Luxemburg, Presses Universitaires de Namur.

Ponsard, C., Massonet, P., Molderez, J. F., Rifaut, A., Lamsweerde, A. v., and Van, H. T. (2007). Early Verification and Validation of Mission Critical Systems, *Formal Methods in System Design* 30(3): 133-247.

Popandreeva, A. (2007). Object-Oriented Analysis and Design Using UML of a Test "Rotation with Sample". International Conference on Computer Systems and Technologies (CompSysTech' 07), University of Rousse, Bulgaria, ACM.

Praxis High Integrity Systems (2008). *Correctness by Construction*. Retrieved 22 December 2008, Available from http://www.praxis-his.com.

ProB (2009). *ProB 1.2*. Retrieved 15 March 2009, Available from http://www.stups.uni-duesseldorf.de/ProB/overview.php.

Ramchandani, C. (1974). Analysis of asynchronous concurrent systems by timed Petri nets. Massachusetts Institute of Technology. MA, Cambridge. **P**hD Thesis.

Razili, R., Snook, C., Poppleton, M., Garratt, P., and Walters, R. (2007). *Experimental Comparison of the Comprehensibility of a UML-based Formal Specification versus a Textual One*. In Proceedings of the 11th International Conference on Evaluation and Assessment in Software Engineering (EASE'07), Keele University, UK, ACM.

Reisig, W. (1985). *Petri nets: an introduction*, Springer-Verlag New York, Inc.

Requet, A. (2007). *BRAMA*. In Proceedings of the RODIN Industry Day, Paris, France, CLEARSY.

Requet, A. (2008, 16 July 2008). *The B formal Method: from Research to Teaching*, 19 April 2009, Available from http://www.atelierb.eu/pdf/nantes_2008_atelier_b_v4.pdf.

Rifaut, A., Massonet, P., Molderez, J.-F., Ponsard, C., Stadnik, P., Lamsweerde, A. v., and Hung, T. V. (2003). *FAUST : Formal Analysis of Goal-Oriented Requirements Using Specification Tools*. In Proceedings of the 11th IEEE International Requirements Engineering Conference (RE'03), Monterey Bay, California, USA, IEEE.

RODIN (2009). *Development Environment for Complex Systems (Rodin)*. Retrieved 10 Febuary 2009, Available from http://rodin.cs.ncl.ac.uk/.

Rubio-Loyola, J., Serrat, J., Charalambides, M., Flegkas, P., Pavlou, G., and Lafuente, A. L. (2005). *Using linear temporal model checking for goal-oriented policy refinement frameworks*. In Proceedings of the 6th IEEE International Workshop on Policies for Distributed Systems and Networks, Stockholm, Sweden, IEEE Computer Society, Vol. 4347.

Schneider, S. (2000). *Concurrent and Real-time Systems: The CSP Approach*, John Wiley & Son, Ltd.

Schneider, S. (2001). *The B-method : An introduction*, Palgrave Macmillan.

Smith, M. H., Hozmann, G. J., and Etessami, K. (2001). *Event and Constraints: A Graphical Editor for Capturing Logic Requirement of Programs*. In Proceedings of the 5th IEEE International Symposium on Requirements Engineering, Toronto, Canada, IEEE Computer Society.

Snook, C., and Butler, M. (2006). UML-B: Formal modelling and design aided by UML, *ACM Transactions on Software Engineering and Methodology* 15(1): 92-122.

Snook, C., and Butler, M. (2008a). *UML-B and Event-B: an integration of languages and tools*. In Proceedings of the IASTED International Conference on Software Engineering (SE2008), Innsbruck, Austria, ACTA Press.

Snook, C., and Butler, M. (2008b). *UML-B: A plug-in for the Event-B tool set* In Proceedings of the 1st International Conference on Abstract State Machines, B and Z, London, UK, Springer-Verlag.

Snook, C., and Butler., M. (2001). *Using a Graphical Design Tool for Formal Specification*. In Proceedings of the 13th Workshop of the Psychology of Programming Interest Group, Bournemouth, United Kingdom, PPIG.

Sommerville, I. (2004). "Critical Systems Specifications 3 Formal Specification." Retrieved 5 October, 2009, Available from www.cs.st-andrews.ac.uk/~ifs/Books/SE8/Syllabuses/CRIT-SYS-SLIDES/CritSysSpec-3.ppt.

Sørensen, I. H. (1994). *The B-Toolkit demonstration*. In Proceedings of the 6th Nordic Workshop on Programming Theory, Aarhus, Denmark, Springer, LNCS 915.

Sparx Systems (2006). *UML 2 Timing Diagram*. Retrieved 26 May 2009, Available from http://sparxsystems.com.au/resources/uml2_tutorial/uml2_timingdiagram.html

Spivey, J. M. (1992). *The Z Notation. A Reference Manual*, Prentice Hall.

SynaptiCAD (2009). Retrieved 25 May 2009, Available from http://www.syncad.com/.

SysML (2008). *OMG System Modelling Language*. Retrieved 3 Febuary 2009, Available from http://www.omgsysml.org/.

SysML Partners (2006). *SysML v.1.0a Specification (revised OMG Submission)*. Retrieved 22 JaNaury 2009, 2006, Available from http://www.sysml.org.

Vanderperren, Y., and Dehaene, W. (2005). *UML 2 and SysML: an Approach to Deal with Complexity in SoC/NoC Design*. In Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05), Munich, Germany, IEEE Computer Society.

Visual Paradigm (2007). *UML 2 Diagrams : Timing Diagram*. Retrieved September, 2007, Available from http://www.visual-paradigm.com/VPGallery/diagrams/TimingDiagram.html.

Yoder, M. A. and B. A. Black (2006). *A Study of Graphical vs. Textual Programming for Teaching DSP*. In Prodeeding of the 36th annula Frontiers in Education Conference, San Diego, CA, IEEE Xploer.

Younes, A. B., and Ayed, L. J. B. (2007). *Using UML Activity Diagrams and Event B for Distributed and Parallel Applications*. In Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), Beijing, China, IEEE Computer Society, Vol. 1.

You, S. K E.(1993). *Towards Modeling and Reasoing Support for Early-Phase Requiremetns Engineering*. In Proceeding of the 1st International Symposium on Requirements Engineering (RE'93), Bonn, Germany, IEEE Xplore.

Wing, J. M. (1990). "A Specifier's Introduction to Formal Methods." I*EEE Computer* 23(9): 8-26.

Zimmerman, M. K., Lundqvist, K., and Leveson, N. (2002). *Investigating the Readability of State-Based Formal Requirements Specification Languages*. In Proceedings of the 22nd International Conference on Software Engineering (ICSE'02), Orlando, Florida, USA, ACM.

Appendix A. **Event-B Textual Translation rules**

A.1 Event-B systematic textual direct translation rules

1. Rule : **TContext**

TContext(Machine) →
“*SETS*”
TSet(Machine)
“*CONSTANTS*”
TConstant(Machine)
“*AXIOMS*”
TAxiom(Machine)

2. Rule : **TMachine**

TMachine(Machine) →
“*VARIABLES*”
TGVarTime(Machine)
TGVarState(Machine)
“*INVARIANTS*”
TGVarTimeInv(Machine)
TGVarStateInv(Machine)
“*EVENTS*”
TGVarTimeInit(Machine)
TGVarStateInit(Machine)
TEvent(Machine)

3. Rule : **TSet**

TSet(Machine) →

<FOR> t <IN> **TAllTimeline**(Machine)

{ **TClassName**(t) + "*_STATES* " }

4. Rule : **TConstant**

TConstant(Machine) →

<FOR> t <IN> **TAllTimeline**(Machine)

{ **TWriteAllCntstStates**(t) }

TWriteAllCntstStates(t) → **TWriteAllConsts**(**TAllState**(t))

TWriteAllConsts(Head : SeqTail) → Head + " "

+ **TWriteAllConsts**(SeqTail)

TWriteAllConsts(Head : < >) → Head

5. Rule : **TAxiom**

TAxiom(Machine) →

<FOR> t <IN> **TTimelineSet**(Machine)

{ **Tname**(t) + "*_STATES* = " + **TWriteAllStates**(t) }

TWriteAllStates(t) → "*{*" + **TAllStateLst**(**TAllState**(t)) + "*}*"

TAllStateLst(Head : SeqTail) → Head + "," + **TAllStateLst**(SeqTail)

TAllStateLst(Head : < >) → Head

6. Rule : **TGVarTime**

TGVarTime(Machine) →
<FOR> t <IN> **TAllTimeline**(Machine)
{ <FOR> s <IN> **TAllState**(t)
{ **TName**(t) + s + "*Time*" } }

7. Rule : **TGVarTimeInv**

TGVarTimeInv(Machine) →
<FOR> t <IN> **TAllTimeline**(Machine)
{ <FOR> s <IN> **TAllState**(t)
{ **TName**(t) + s + "*Time* ∈ ℕ" } }

8. Rule : **TGVarTimeInit**

TGVarTimeInit(Machine) →
<FOR> t <IN> **TAllTimeline**(Machine)
{ <FOR> s <IN> **TAllState**(t)
{ **TName**(t) + s + "*Time := 0*" } }

9. Rule : **TGVarState**

TGVarState(Machine) →
<FOR> t <IN> **TAllTimeline**(Machine)
{**TName**(t) + "*State*" }

10. Rule : **TGVarStateInv**

TGVarStateInv(Machine) →
<LET> exp = **TClass**(**TTimelineInClass**(t))
<IN>

<FOR> t <IN> **TAllTimeline**(Machine)
{ **TName**(t) + "*State* ∈ " +
<IF> **THasParam**(**TClass**(**TName**(t)))
<THEN>
"("+ **TWriteParamForInv**(**TAllParamType**(exp)) + ")"
+ " → " + **TClassName**(t) + "_STATE"
<ELSE> **TClassName**(t) + "_STATE"
<ENDIF>
}

TWriteParamForInv(Head : SeqTail) →
Head + "×" + **TWriteParamForInv**(SeqTail)
TWriteParamForInv(Head : < >) → Head

11. Rule : **TGVarStateInit**

TGVarStateInit(Machine) →
<FOR> t <IN> **TAllTimeline**(Machine)
{ **TName**(t) + "*State := {xInitValuex}*" }

12. Rule : **TTicktok**

TTicktok → "*Ticktok = **BEGIN** gclock := gclock + 1 **END***"

A.2 Translation rules for creating an event

13. Rule : **TEventName**

TEventName(Segment) →
<LET> exp = **TObjSt**(Segment)
<IN> **TObj**(exp) + exp + "="

14. Rule : **TParamLst**

TParamLst(Segment) →
<LET> exp = **TObj**(**TObjSt**(Segment))
<IN>
    <IF> **THasParam**(**TClass**(exp))
    <THEN>
        "***ANY***" +
        TWriteAllParams(**TAllParam**(**TClass**(exp)))
    <ELSE>
        "***WHEN*** "
    <ENDIF>

TWriteAllParams(Head : ParamSeqTail) →
        Head + ";" + **TWriteAllParams**(ParamSeqTail)
TWriteAllParams(Head : < >) → Head

15. Rule : **TParamGuardValue**

TParamGuardValue(Segment) →
<LET> exp = **TClass**(**TObj**(**TObjSt**(Segment)))
<IN>
    <IF>**THasParam**(exp)
    <THEN>
        "*WHERE*" +
        TWriteAllParamsLst(**TAllParam**(exp))
    <ELSE> <SKIP>
    <ENDIF>

TWriteAllParamsLst(Head : ParamSeqTail) →
        Head + ":" + **TParamType**(Head) + "&" +
        TWriteAllParamsLst(ParamSeqTail)

TWriteAllParamsLst(Head : < >) → Head + ":" + **TParamType**(Head)

16. Rule : **TGrdCtrnt**

TGrdCtrnt(Segment) →

{ "(" + **TGetGrdPredc**(**TNodeType**(**TConstrnt**(Segment))) + ")" }

17. Rule : **TGetGrdPredc**

TGetGrdPredc(NodeType) →

<IF> NodeType = Simple

<THEN><IF> **THasTiming**(Simple)

<THEN> **TTimingGuard**(**TSegment**(Simple), **TTiming**(Simple))

+ "&" + **TSimpleCauseSource**(**TSegment**(Simple))

+ **TSimpleCond**(**TCond**(Simple))

<ELSE>

TSimpleCauseSource(**TSegment**(Simple))

+ **TSimpleCond**(**TCond**(Simple))

<ENDIF>

<ELSE><IF> NodeType = OR_node

<THEN><LET> Nodes = **TAllInstances**(OR_node)

<IN> Nodes → <ITERATE>(n; ret : String = "(" |

<IF> n = **last**(Nodes)

<THEN> ret = ret + **TGetGrdPredc**(n) + ")"

<ELSE> ret = ret + **TGetGrdPredc**(n) + ")∨ ("

<ENDIF>)

<ENDIF>

<ELSE> <IF> NodeType = AND_node

<THEN><LET> Nodes = **TAllInstances**(AND_node)

<IN> Nodes → <ITERATE>(n; ret : String = "(" |

<IF> n = **last**(Nodes)

<THEN> ret = ret + **TGetGrdPredc**(n) + ")"

<ELSE> ret = ret + **TGetGrdPredc**(n) + "*)* ∧ *(*"

<ENDIF>)

<ENDIF>

<ENDIF>

18. Rule : **TTimingGuard**

TTimingGuard(Segment, Timing) → "*(gclock* - "

+ **Tobj**(**TobjSt**(Segment))

+ **TobjSt**(Segment))

+ "*Time* ≥ "

+ **TlowerLmt**(Timing) + "*)*"

+ "*& (gclock* – "

+ **Tobj**(**TobjSt**(Segment))

+ **TobjSt**(Segment))

+ "*Time* ≤ "

+ **TupperLmt**(Timing) + "*)*"

19. Rule : **TSimpleCauseSource**

TSimpleCauseSource(Segment) →

<LET> exp = **TClass**(**TObj**(**TObjSt**(Segment)))

<IN>

<IF> **THasParam**(exp)

<THEN>

TObj(**TObjSt**(Segment))

+ "*State(* "

TWriteParamLst(**TAllParam**(exp))

+ "*)* = "

+ **TObjSt**(Segment)

<ELSE>

TObj(**TObjSt**(Segment))

+ "*State* = "

+ **TObjSt**(Segment)

<ENDIF>

TWriteParamLst(Head : SeqTail) → Head +

"↦" + **TWriteParamLst**(SeqTail)

TWriteParamLst(Head : < >) → Head

20. Rule : **TSimpleCond**

TSimpleCond(Predicate) →

<IF> **TEmpty**(Predicate)

<THEN> <SKIP>

<ELSE> **TAllInstances** (Predicate) → <ITERATE>(p; ret : String = " " | →

ret ← "&" + p)

<ENDIF>

21. Rule : **TPrevSegm**

TPrevSegm(Segment) →

"&" + "(" + **TWritePrevStateLst**(Segment,**TAllPrevSegm**(Segment)) + ")"

TWritePrevStateLst(Segment, Head : SegmSeqTail) →

TSimplePrevSegm(Segment, Head)

+ " ∨ "

+ **TWritePreStateLst**(Segment, SegmSeqTail)

TWritePrevStateLst(Segment, Head : < >) →

TSimplePrevSegm(Segment, Head)

22. Rule : **TSimplePrevSegm**

TSimplePrevSegm(Segment, Head) →

<LET> exp = **TClass**(**TObj**(**TObjSt**(Segment)))

<IN>
<IF> **THasParam**(exp)
<THEN> **TObj**(**TObjSt**(Segment)) + "*State(*"
+ **TWriteParamLst**(**TAllParam**(exp))
+ "*) =* "
+ **TObjSt**(Head)
<ELSE>
TObj(**TObjSt**(Segment)) + "*State =*"
+ **TObjSt**(Head)
<ENDIF>

23. Rule : **TSubst**(Segment)

TSubst(Segment) →
<LET> exp = **TObj**(**TObjSt**(Segment)
<IN>
<IF> **THasParam**(**TClass**(exp))
<THEN> **TObjName**(exp)
+ "*State(* "
+ **TWriteParamLst**(**TAllParam**(**TClass**(exp)))
+ "*) :=* "
+ **TObjSt**(Segment)
<ELSE> exp
+ "*State : =* "
+ **TObjSt**(Segment)
<ENDIF>

24. Rule : **TSimul**

TSimul(Segment) →
<LET> exp = **TClass**(**TObj**(**TObjSt**(**TEndSegm**(s))))
<IN>
<IF> **THasSimul**(Segment)

<THEN> <FOR> s <IN> **TSimulSeq**(Segment)
{<IF> **THasParam**(exp)
<THEN>
TObj(**TObjSt**(**TEndSegm**(s)))
+ "*State(* "
+ **TWriteParamLst**(**TAllParam**(exp))
+ "*) :=* "
+ **TObjSt**(**TEndSegm**(s))
<ELSE>
TObj(**TObjSt**(**TEndSegm**(s)))
+ "*State :=* "
+ **TObjSt**(**TEndSegm**(s))
<ENDIF>
<IF> <NOT> s = **last**(**TSimulSeq**(Segment))
<THEN> "*&*"
<ELSE> <SKIP>
<ENDIF>
}
<ELSE> <SKIP>
<ENDIF>

25. Rule : **TRecdTime**

TRecdTime(Segment) →
TObj(**TObjSt**(Segment))
+ **TObjSt**(Segment)
+ "*Time := gclock*"

Appendix B. An Event-B model created from the Direct translation rules

An Event-B model is generated from systematic textual translation rules is illustrated below. This Event-B model composes of two parts: a context named `LiftSystem_EventB_ctx` and a machine named `LiftSystem`.

B.1 Context : LiftSystem_EventB_ctx

context LiftSystem_EventB_ctx

constants Lit Unlit On Off rLit rUnlit MovingArrivingUp MovingUp MovingDepartingUp StopAtFloor MovingDepartingDown MovingDown MovingArrivingDown Open Closed acTivated deActivated Activated Deactivated FLOOR TOP BOTTOM Up Down

sets FLOORLAMP_STATES FLOORSENSOR_STATES REQUESTLAMP_STATES LIFT_STATES DOOR_STATES UPLAMP_STATES DOWNLAMP_STATES DIR

axioms

@axm39 **DIR** = {**Up, Down**}

@axm1 **FLOORLAMP_STATES** = {**Lit, Unlit**}

@axm2 **Lit** ≠ **Unlit**

@axm3 **FLOORSENSOR_STATES** = {**On, Off**}

@axm4 **On** ≠ **Off**

@axm5 **REQUESTLAMP_STATES** = {**rLit, rUnlit**}

@axm6 **rLit** ≠ **rUnlit**

```
    @axm7 LIFT_STATES =
          {MovingArrivingUp, MovingUp, MovingDepartingUp,
           StopAtFloor, MovingDepartingDown, MovingDown,
           MovingArrivingDown}
    @axm8 MovingArrivingUp ≠ MovingUp
    @axm9 MovingArrivingUp ≠ MovingDepartingUp
    @axm10 MovingArrivingUp ≠ StopAtFloor
    @axm11 MovingArrivingUp ≠ MovingDepartingDown
    @axm12 MovingArrivingUp ≠ MovingDown
    @axm13 MovingArrivingUp ≠ MovingArrivingDown
    @axm14 MovingUp ≠ MovingDepartingUp
    @axm15 MovingUp ≠ StopAtFloor
    @axm16 MovingUp ≠ MovingDepartingDown
    @axm17 MovingUp ≠ MovingDown
    @axm18 MovingUp ≠ MovingArrivingDown
    @axm19 MovingDepartingUp ≠ StopAtFloor
    @axm20 MovingDepartingUp ≠ MovingDepartingDown
    @axm21 MovingDepartingUp ≠ MovingDown
    @axm22 MovingDepartingUp ≠ MovingArrivingDown
    @axm23 StopAtFloor ≠ MovingDepartingDown
    @axm24 StopAtFloor ≠ MovingDown
    @axm25 StopAtFloor ≠ MovingArrivingDown
    @axm26 MovingDepartingDown ≠ MovingDown
    @axm27 MovingDepartingDown ≠ MovingArrivingDown
    @axm28 MovingDown ≠ MovingArrivingDown
    @axm29 DOOR_STATES = {Open, Closed}
    @axm30 Open ≠ Closed
    @axm31 UPLAMP_STATES = {acTivated, deActivated}
    @axm32 acTivated ≠ deActivated
    @axm33 DOWNLAMP_STATES = {Activated, Deactivated}
    @axm34 Activated ≠ Deactivated
    @axm35 FLOOR = (BOTTOM‥TOP)
    @axm38 BOTTOM = 1
    @axm37 TOP = 3
    @axm36 Up ≠ Down
end
```

B.2 Machine : LiftSystem_EventB

machine LiftSystem_EventB **sees** LiftSystem_EventB_ctx

variables reqFl currentFl floorlampState floorsensorState requestlampState doorState liftState uplampState downlampState dir gclock floorlampLittime floorlampUnlitTime floorsensorOnTime floorsensorOffTime requestlampRequestedTime requestlampUnrequestedTime liftMovingArrivingUpTime liftMovingUpTime liftMovingDepartingUpTime liftStopAtFloorTime liftMovingDepartingDownTime liftMovingDownTime liftMovingArrivingDownTime doorOpenTime doorClosedTime uplampDeactivatedTime uplampActivatedTime downlampDeactivatedTime downlampActivatedTime floorlampLitTime

invariants

@inv1 requestlampState $\in$ **FLOOR** $\rightarrow$ **REQUESTLAMP_STATES**

@inv2 reqFl $\subseteq$ **FLOOR**

@inv3 floorlampState $\in$ **FLOOR** $\rightarrow$ **FLOORLAMP_STATES**

@inv4 floorsensorState $\in$ **FLOOR** $\rightarrow$ **FLOORSENSOR_STATES**

@inv5 doorState $\in$ **FLOOR** $\rightarrow$ **DOOR_STATES**

@inv6 liftState $\in$ **LIFT_STATES**

@inv7 uplampState $\in$ **UPLAMP_STATES**

@inv8 downlampState $\in$ **DOWNLAMP_STATES**

@inv9 currentFl $\in$ **FLOOR**

@inv10 dir $\in$ **DIR**

@inv11 gclock $\in \mathbb{N}$

@inv12 floorlampLittime $\in \mathbb{N}$

@inv13 floorlampUnlitTime $\in \mathbb{N}$

@inv14 floorsensorOnTime $\in \mathbb{N}$

@inv15 floorsensorOffTime $\in \mathbb{N}$

@inv16 requestlampRequestedTime $\in \mathbb{N}$

@inv17 requestlampUnrequestedTime $\in \mathbb{N}$

@inv18 liftMovingArrivingUpTime $\in \mathbb{N}$

```
  @inv19 liftMovingUpTime ∈ ℕ
  @inv20 liftMovingDepartingUpTime ∈ ℕ
  @inv21 liftStopAtFloorTime ∈ ℕ
  @inv22 liftMovingDepartingDownTime ∈ ℕ
  @inv23 liftMovingDownTime ∈ ℕ
  @inv24 liftMovingArrivingDownTime ∈ ℕ
  @inv25 doorOpenTime ∈ ℕ
  @inv26 doorClosedTime ∈ ℕ
  @inv27 uplampDeactivatedTime ∈ ℕ
  @inv28 uplampActivatedTime ∈ ℕ
  @inv29 downlampDeactivatedTime ∈ ℕ
  @inv30 downlampDeactivatedTime ∈ ℕ
  @inv31 downlampActivatedTime ∈ ℕ
  @inv32 floorlampLitTime ∈ ℕ
  @inv33 ¬(uplampState = acTivated
        ∧ downlampState = Activated)
  @inv34 doorState(currentFl) = Open ⇒
        liftState = StopAtFloor
  @inv35 liftState ≠ StopAtFloor ⇒
        doorState(currentFl) = Closed
  @inv36 currentFl ≠ (currentFl + 1) // For POs
  @inv37 currentFl ≠ (currentFl − 1) // For POs

events
  event INITIALISATION
    then
      @act1 requestlampState ≔ FLOOR × {rUnlit}
      @act2 reqFl ≔ ∅
      @act3 floorlampState ≔ {1↦Lit, 2↦Unlit, 3↦Unlit}
  // if changes to floorlampState ≔ {}, PO is discharged
      @act4 floorsensorState ≔ {1 ↦ On, 2 ↦ Off, 3 ↦ Off}
  // if changes to floorsensorState ≔ {}, PO is discharged
      @act5 doorState ≔ FLOOR × {Closed}
      @act6 liftState ≔ StopAtFloor
```

```
    @act7 uplampState ≔ deActivated
    @act8 downlampState ≔ Deactivated
    @act9 currentFl ≔ BOTTOM
    @act10 dir ≔ Up
    @act11 gclock ≔ 0
    @act12 floorlampLittime ≔ 0
    @act13 floorlampUnlitTime ≔ 0
    @act14 floorsensorOnTime ≔ 0
    @act15 floorsensorOffTime ≔ 0
    @act16 requestlampRequestedTime ≔ 0
    @act17 requestlampUnrequestedTime ≔ 0
    @act18 liftMovingArrivingUpTime ≔ 0
    @act19 liftMovingUpTime ≔ 0
    @act20 liftMovingDepartingUpTime ≔ 0
    @act21 liftStopAtFloorTime ≔ 0
    @act22 liftMovingDepartingDownTime ≔ 0
    @act23 liftMovingDownTime ≔ 0
    @act24 liftMovingArrivingDownTime ≔ 0
    @act25 doorOpenTime ≔ 0
    @act26 doorClosedTime ≔ 0
    @act27 uplampDeactivatedTime ≔ 0
    @act28 uplampActivatedTime ≔ 0
    @act30 downlampDeactivatedTime ≔ 0
    @act31 downlampActivatedTime ≔ 0
    @act29 floorlampLitTime ≔ 0
end

event UserRequestlamprLit
// The original name got from the rule is requestlamprLit
  any f
  where
    @grd1 f ∈ FLOOR
  then
    @act1 reqFl ≔ reqFl ∪ {f}
```

```
    @act2 requestlampState(f) ≔ rLit
    @act3 requestlampRequestedTime ≔ gclock
  end

  event SetRequestlamprUnlit
// line 6; the original name got from the rule
is requestlamprUnLit
    any f
    where
      @grd1 f ∈ FLOOR
      @grd2 liftState = StopAtFloor
        ∧ ((gclock − liftStopAtFloorTime ≥ 2)
        ∧ (gclock − liftStopAtFloorTime ≤ 4))
        ∧ f = currentFl
      @grd3 requestlampState(f) = rLit
    then
      @act1 requestlampState(f) ≔ rUnlit
      @act2 requestlampUnrequestedTime ≔ gclock
  end

  event doorOpen // line 9
    any f
    where
      @grd1 liftState = StopAtFloor
        ∧ ((gclock − liftStopAtFloorTime ≥ 1)
        ∧ (gclock − liftStopAtFloorTime ≤ 5))
        ∧ f ∈ reqFl ∧ f = currentFl
      @grd3 doorState(f) = Closed
    then
      @act1 doorState(f) ≔ Open
      @act2 doorOpenTime ≔ gclock
  end

  event doorClosed
    where
      @grd1 doorState(currentFl) = Open
```

```
        @grd2 liftState = StopAtFloor // from POs
      then
        @act1 doorState(currentFl) ≔ Closed
        @act2 reqFl ≔ reqFl ∖ {currentFl}
        @act3 doorClosedTime ≔ gclock
    end

    event liftMovingDepartingUp // line 18 and 7
      any f
      where
        @grd6 f ∈ FLOOR
        @grd1 (requestlampState(f) = rLit
              ∧ f > currentFl)
              ∧ (doorState(currentFl) = Closed
              ∧ ((gclock − doorClosedTime ≥ 1)
              ∧ (gclock − doorClosedTime ≤ 5))
              ∧ f ∈ reqFl)
        @grd5 currentFl ∉ reqFl
        // The lift much serve the currentFl first if there is
     a request   for the currentFl. Otherwise, the lift will
    move to other floors and in the same time service
    the currentFl.
        @grd3 liftState = StopAtFloor
        @grd7 dir = Up
      then
        @act1 liftState ≔ MovingDepartingUp
        @act2 uplampState ≔ acTivated
        @act3 downlampState ≔ Deactivated
        @act4 liftMovingDepartingUpTime ≔ gclock
    end

  event liftMovingDepartingUp2
    // Used whenever there is no continuously request between
connected floors; The lift will change its state from
MovingArrivingUp to MovingDepartingUp.
      any f
```

```
  where
    @grd1 f ∈ FLOOR
    @grd2 f ∈ reqFl ∧ f > currentFl
    @grd3 currentFl ∉ reqFl
    @grd4 liftState = MovingArrivingUp
    @grd5 dir = Up
  then
    @act1 liftState ≔ MovingDepartingUp
    @act2 uplampState ≔ acTivated
    @act3 downlampState ≔ Deactivated
    @act4 liftMovingDepartingUpTime ≔ gclock
end

event liftMovingUp
  any f
  where
    @grd1 f ∈ FLOOR
    @grd2 f ∈ reqFl ∧ f > currentFl
    @grd3 currentFl ∉ reqFl
    @grd4 liftState = MovingDepartingUp
    @grd5 dir = Up
    @grd6 floorsensorState(currentFl) = Off
     // addition guard from Simultaneity -- future work
  then
    @act1 liftState ≔ MovingUp
    @act2 uplampState ≔ acTivated
    @act3 downlampState ≔ Deactivated
    @act4 liftMovingUpTime ≔ gclock
end

event liftMovingDepartingDown // line 19 and 8
  any f
  where
    @grd1 f ∈ FLOOR
    @grd2 ( requestlampState(f) = rLit ∧ f < currentFl)
          ∧ (doorState(currentFl) = Closed
```

```
                ∧ ((gclock − doorClosedTime ≥ 1)
                ∧ (gclock − doorClosedTime ≤ 5))
                ∧ f ∈ reqFl)
        @grd3 currentFl ∉ reqFl
        @grd6 liftState = StopAtFloor
        @grd5 dir = Down
      then
        @act1 liftState ≔ MovingDepartingDown
        @act2 uplampState ≔ deActivated
        @act3 downlampState ≔ Activated
        @act4 liftMovingDepartingDownTime ≔ gclock
    end

  event liftMovingDepartingDown2
    // Used whenever there is no continuously request between
connected floors;The lift will change its state from
MovingArrivingDown to MovingDepartingDown.
      any f
      where
        @grd1 f ∈ FLOOR
        @grd2 f ∈ reqFl ∧ f < currentFl
        @grd3 currentFl ∉ reqFl
        @grd4 liftState = MovingArrivingDown
        @grd5 dir = Down
      then
        @act1 liftState ≔ MovingDepartingDown
        @act2 uplampState ≔ deActivated
        @act3 downlampState ≔ Activated
        @act4 liftMovingDepartingDownTime ≔ gclock
    end

    event liftMovingDown
      any f
      where
        @grd1 f ∈ FLOOR
        @grd2 f ∈ reqFl ∧ f < currentFl
```

```
      @grd3 currentFl ∉ reqFl
      @grd4 liftState = MovingDepartingDown
      @grd5 dir = Down
      @grd6 floorsensorState(currentFl) = Off
    // addition guards from SimultaneityArrow -- future work
  then
      @act1 liftState ≔ MovingDown
      @act2 uplampState ≔ deActivated
      @act3 downlampState ≔ Activated
      @act4 liftMovingDownTime ≔ gclock
end

event floorsensorOn // line 4.1 and 4.2
  any f
  where
      @grd1 f ∈ FLOOR
      @grd2 (liftState = MovingArrivingUp
        ∧ ((gclock − liftMovingArrivingUpTime ≥ 2)
        ∧ (gclock − liftMovingArrivingUpTime ≤ 5))
            ∧ f = currentFl)
            ∨
            (liftState = MovingArrivingDown
        ∧((gclock − liftMovingArrivingDownTime ≥ 2)
        ∧ (gclock − liftMovingArrivingDownTime ≤ 5))
            ∧ f = currentFl)
      @grd4 floorsensorState(f) = Off
  then
      @act1 floorsensorState(f) ≔ On
      @act2 floorsensorOnTime ≔ gclock
end

event floorsensorOffUp
    // Line 3.1; the floorsensorOff has to be spited into 2
events: floorsensorOffUp and floorsensorOffDown. That is because
the Simultaneity arrows:  MovingUp and MovingDown. Without the
```

```
SimultanetiyArrow, those floorsensorOffUp and floorsensorOffDown
can be combined.
    any f
    where
      @grd1 f ∈ FLOOR
      @grd2 liftState = MovingDepartingUp
        ∧ ((gclock − liftMovingDepartingUpTime ≥ 2)
        ∧ (gclock − liftMovingDepartingUpTime ≤ 5))
        ∧ f = currentFl ∧ dir = Up
      @grd4 floorsensorState(f) = On
    then
      @act1 floorsensorState(f) ≔ Off
      @act2 liftState ≔ MovingUp
      @act3 floorsensorOffTime ≔ gclock
  end

  event floorsensorOffDown
    any f
    where
      @grd1 f ∈ FLOOR
      @grd2 liftState = MovingDepartingDown
        ∧ ((gclock − liftMovingDepartingDownTime ≥ 2)
        ∧ (gclock − liftMovingDepartingDownTime ≤ 5))
        ∧ f = currentFl ∧ dir = Down
      @grd3 floorsensorState(f) = On
    then
      @act1 floorsensorState(f) ≔ Off
      @act2 liftState ≔ MovingDown
      @act3 floorsensorOffTime ≔ gclock
  end

  event floorlampUnlit // line 1
    any f
    where
      @grd1 f ∈ FLOOR
      @grd2 floorsensorState(f) = Off
```

```
        ∧ ((gclock − floorsensorOffTime ≥ 2)
        ∧ (gclock − floorsensorOffTime) ≤ 4)
        ∧ f = currentFl
      @grd3 floorlampState(f) = Lit
  then
    @act1 floorlampState(f) ≔ Unlit
    @act2 floorlampUnlitTime ≔ gclock
end

event floorlampLit // line 2
  any f
  where
    @grd1 f ∈ FLOOR
    @grd2 floorsensorState(f) = On
      ∧ ((gclock − floorsensorOnTime ≥ 2)
      ∧ (gclock − floorsensorOnTime ≤ 4))
      ∧ f = currentFl
    @grd3 floorlampState(f) = Unlit
  then
    @act1 floorlampState(f) ≔ Lit
    @act2 floorlampLitTime ≔ gclock
end

event liftMovingArrivingUp
  any f
  where
    @grd1 f ∈ FLOOR
    @grd2 f ∈ reqFl ∧ f > currentFl
    @grd4 currentFl ∉ reqFl
    @grd3 liftState = MovingUp
    @grd5 doorState(currentFl) = Closed // from POs
    @grd6 floorlampState(currentFl) = Unlit
// manually additional  guards
  then
    @act1 liftState ≔ MovingArrivingUp
    @act2 currentFl ≔ currentFl + 1
```

```
      @act3 liftMovingArrivingUpTime ≔ gclock
      @act4 doorState(currentFl+1) ≔ Closed // From PO
  end

  event liftMovingArrivingDown
    any f
    where
      @grd1 f ∈ FLOOR
      @grd2 f ∈ reqFl ∧ f < currentFl
      @grd3 currentFl ∉ reqFl
      @grd4 liftState = MovingDown
      @grd5 doorState(currentFl) = Closed // from POs
      @grd6 floorlampState(currentFl) = Unlit
// manually additionalguards
    then
      @act1 liftState ≔ MovingArrivingDown
      @act2 currentFl ≔ currentFl − 1
      @act3 liftMovingArrivingDownTime ≔ gclock
      @act4 doorState(currentFl − 1) ≔ Closed // From PO
  end

  event liftStopAtFloor // line 5
    any f
    where
      @grd1 f ∈ FLOOR
      @grd2 floorsensorState(f) = On
       ∧ ((gclock − floorsensorOnTime ≥ 1)
       ∧ (gclock  − floorsensorOnTime ≤ 5))
       ∧ f ∈ reqFl ∧ f = currentFl
      @grd3 liftState = MovingArrivingUp
       ∨ liftState = MovingArrivingDown
    then
      @act1 liftState ≔ StopAtFloor
      @act2 uplampState ≔ deActivated
      @act3 downlampState ≔ Deactivated
```

```
    @act4 liftStopAtFloorTime ≔ gclock
end

event ChangeDirUp
  any f
  where
    @grd1 f ∈ FLOOR
    @grd2 f ∈ reqFl ∧ f > currentFl
    @grd3 currentFl ∉ reqFl
    @grd6 reqFl ≠ ∅
    @grd4 liftState = StopAtFloor
    @grd5 dir = Down
  then
    @act1 dir ≔ Up
end

event ChangeDirDown
  any f
  where
    @grd1 f ∈ FLOOR
    @grd2 f ∈ reqFl ∧ f < currentFl
    @grd3 currentFl ∉ reqFl
    @grd4 reqFl ≠ ∅
    @grd5 liftState = StopAtFloor
    @grd6 dir = Up
  then
    @act1 dir ≔ Down
end

event Ticktok
  where
// Requestlamp Unlit
    @grd1 liftState = StopAtFloor
      ∧ requestlampState(currentFl) = rLit
      ∧ (((gclock − liftStopAtFloorTime)≥ 2)
          ∧ ((gclock − liftStopAtFloorTime) ≤ 4))
```

```
            ⇒
            gclock - liftStopAtFloorTime < 4

        // Floorsensor On (when lift Moving Arriving Up).
           It has to be spited into two floorsensorOn guards.
           Because it cannot be written as
           => 5 + (liftMovingArrivingUpTime or
                   LiftMovingArrivingDownTime) - gclock > 1
        @grd2 (liftState = MovingArrivingUp
         ∧floorsensorState (currentFl) = Off
         ∧ ((gclock - liftMovingArrivingUpTime) ≥ 2)
         ∧ ((gclock - liftMovingArrivingUpTime) ≤ 5))
            ⇒
            gclock -liftMovingArrivingUpTime < 5

    // Floorsensor On (when lift Moving Arriving Down)
        @grd3 ( liftState = MovingArrivingDown
         ∧ floorsensorState(currentFl) = Off
         ∧((gclock - liftMovingArrivingDownTime) ≥ 2)
         ∧((gclock - liftMovingArrivingDownTime) ≤ 5))
            ⇒
            gclock - liftMovingArrivingDownTime < 5

    // Floorlamp Lit
        @grd4 ( floorlampState(currentFl) = Unlit
         ∧floorsensorState(currentFl) = On
         ∧((gclock - floorsensorOnTime) ≥ 2)
         ∧((gclock - floorsensorOnTime) ≤ 4))
            ⇒
            gclock - floorsensorOnTime < 4

    // Lift stops at floor
        @grd5 ( floorsensorState(currentFl) = On
         ∧(liftState = MovingArrivingUp ∨
         liftState = MovingArrivingDown)
         ∧ currentFl ∈ reqFl
```

```
  ∧((gclock − floorsensorOnTime) ≥ 1)
  ∧((gclock − floorsensorOnTime) ≤ 5))
        ⇒
        gclock − floorsensorOnTime < 5

// Floorsensor Off (when lift Moving Departing Up).
   It has to be spited into two floorsensorOff guards.
   Because it cannot be written as
   => 5 + (liftMovingDepartingUpTime or
    liftMovingDepartingDownTime) − gclock > 1
@grd6 (liftState = MovingDepartingUp
  ∧floorsensorState(currentFl) = On
  ∧((gclock − liftMovingDepartingUpTime) ≥ 2)
  ∧((gclock − liftMovingDepartingUpTime) ≤ 5))
        ⇒
        gclock − liftMovingDepartingUpTime < 5


// Floorsensor Off (when lift Moving Departing Down)
@grd7 (liftState = MovingDepartingDown
  ∧floorsensorState(currentFl) = On
  ∧((gclock − liftMovingDepartingDownTime) ≥ 2)
  ∧((gclock − liftMovingDepartingDownTime) ≤ 5))
        ⇒
        gclock − liftMovingDepartingDownTime < 5

// Lift Moving Departing Up and Down
   The guards for liftMovingDeparingUp and
   liftMovingDepartinDown are the same.
@grd8 (doorState(currentFl) = Closed
  ∧ liftState = StopAtFloor
  ∧((gclock − doorClosedTime) ≥ 1)
  ∧((gclock − doorClosedTime) ≤ 5))
        ⇒
        gclock − doorClosedTime < 5
```

```
// Door open
      @grd9 (liftState = StopAtFloor
        ∧ doorState(currentFl) = Closed
        ∧ currentFl ∈ reqFl
        ∧((gclock − liftStopAtFloorTime) ≥ 1)
        ∧((gclock − liftStopAtFloorTime) ≤ 5))
             ⇒
             gclock − liftStopAtFloorTime < 5

      // Floorlamp Unlit
      @grd10 ( floorlampState(currentFl) = Lit
        ∧floorsensorState(currentFl) = Off
        ∧((gclock − floorsensorOffTime) ≥ 2)
        ∧((gclock − floorsensorOffTime) ≤ 4))
             ⇒
             gclock − floorsensorOffTime < 4
    then
      @act1 gclock ≔ gclock + 1
  end
end
```

Appendix C. **ATL Translation rules**

```
module TDtoUMLB; -- Module Template
create OUT : umlbMetamodel   from  IN : TDMetamodel;

helper def : umlbproject : umlbMetamodel!UMLBProject =
       umlbMetamodel!UMLBProject;
helper def : umlbclass : umlbMetamodel!UMLBClass =
       umlbMetamodel!UMLBClass;
helper def : umlbmachine : umlbMetamodel!UMLBMachine =
       umlbMetamodel!UMLBMachine;
helper def : nat1Type : umlbMetamodel!UMLBTypeExpression =
       umlbMetamodel!UMLBTypeExpression;
helper def : prmType : umlbMetamodel!UMLBTypeExpression =
       umlbMetamodel!UMLBTypeExpression;
helper def : intType : umlbMetamodel!UMLBTypeExpression =
       umlbMetamodel!UMLBTypeExpression;
helper def : umlbcontext : umlbMetamodel!UMLBContext =
       umlbMetamodel!UMLBContext;  --- for creating Context


rule Project {
 from t : TDMetamodel!TDProject
 to   u : umlbMetamodel!UMLBProject
             (name <- t.name,
              constructs <- t.construct),
      pt1 : umlbMetamodel!UMLBTypeExpression
             (name <- 'BOOL'),
      pt2 : umlbMetamodel!UMLBTypeExpression
             (name <- 'NAT'),

      pt3 : umlbMetamodel!UMLBTypeExpression
```

```
                (name <- 'NAT1'),
        pt4 : umlbMetamodel!UMLBTypeExpression
                (name <- 'INT')
  do {thisModule.umlbproject <- u;
      thisModule.boolType <- pt1;
      thisModule.natType <- pt2;
      thisModule.nat1Type <- pt3;
      thisModule.intType <- pt4;
           u.typeExpressions <- u.typeExpressions.append(pt1);
      u.typeExpressions <- u.typeExpressions.append(pt2);
      u.typeExpressions <- u.typeExpressions.append(pt3);
      u.typeExpressions <- u.typeExpressions.append(pt4);}
}


rule Machine {
  from t : TDMetamodel!TDMachine
  to   ctx : umlbMetamodel!UMLBContext
             (name <- t.name + '_ctx'),
       m : umlbMetamodel!UMLBMachine
             (name <- t.name,
              classes <- t.class),
       e : umlbMetamodel!UMLBEvent
             (name <- 'Ticktok'),
       a : umlbMetamodel!UMLBAction
             (name <- 'Action1',
              action <- 'gclock := gclock + 1'),
       gclk : umlbMetamodel!UMLBVariable
             (name <- 'gclock',
              typeProvider <- thisModule.intType,
              initialValue <- '0')
               -- initialValue is defined in UMLBvariableElement
         do {
              m.events <- m.events.append(e);
              e.actions <- e.actions.append(a);
              m.variables <- m.variables.append(gclk);
              thisModule.umlbmachine <- m;
              m.contexts <- m.contexts.append(ctx);
```

```
                thisModule.umlbproject.constructs <-
                thisModule.umlbproject.constructs.append(ctx);
                thisModule.umlbproject.constructs <-
                thisModule.umlbproject.constructs.append(m); }
}


rule Class {
 from t : TDMetamodel!TDClass
 to   u : umlbMetamodel!UMLBClass
                (name <- t.name,
                selfName <- t.name + 'Self',
                statemachines <- t.timeline),
      att : umlbMetamodel!UMLBAttribute
                (name <- t.name.toLower()+ 'xStatexTime',
                typeProvider <- thisModule.intType,
                initialValue <- '0')
 do { u.attributes <- u.attributes.append(att); }
}


rule StateMachine {
 from t : TDMetamodel!TDTimeline
 to   u : umlbMetamodel!UMLBStatemachine
                (name <- t.name + '_state',
                transitions <- t.timelinetransitions,
                states <- t.states)
 }


rule State {
 from t : TDMetamodel!TDState
 to   u : umlbMetamodel!UMLBState
                (name <- t.name,
                incoming <- t.segments -> collect(c|c.incoming),
                outgoing <- t.segments -> collect(c|c.outgoing))
}
```

```
rule Transition {
 from t : TDMetamodel!TDTimelineTransition
 to   u : umlbMetamodel!UMLBTransition
             (name <- t.target.getTransitionName(),
             target <- t.target.eContainer(),
             source <- t.source.eContainer(),
             guards <- t.target.constraints,
             variables <-t.eContainer().eContainer().parameter
       )
       ,actgclock : umlbMetamodel!UMLBAction
             (name <- t.eContainer().name + '.gClockAction',
             action <- t.target.eContainer().eContainer().name
             + t.target.eContainer().name
             + 'Time('
             +
       t.target.eContainer().eContainer().eContainer().name
             +'Self) := gclock')
-- creates variables to keep the current time (gclock) for
 each event
 do {u.actions <- u.actions.append(actgclock); }
}


helper context  TDMetamodel!TDSegment
 def :  getTransitionName() : String =
 let simuls : Set(TDMetamodel!TDSegment) =
 TDMetamodel!TDSegment.allInstances()->  select(c|c.simul  ->
 includes(self))
 in
       if simuls -> isEmpty() then
             self.eContainer().eContainer().name
             + self.eContainer().name
       else
             simuls.last().getTransitionName()
 endif;
```

```
rule Parameter {
        from t : TDMetamodel!TDParameter (not
        umlbMetamodel!UMLBTypeExpression ->
        allInstances() -> exists(e|e.name = t.paramType))
        to   u : umlbMetamodel!UMLBEventVariable
              (name <- t.param),
              e : umlbMetamodel!UMLBTypeExpression
              (name <- t.paramType)
        do {thisModule.umlbproject.typeExpressions <-
              thisModule.umlbproject.typeExpressions.append(e);
            u.typeProvider <- e; }
}


rule Constraint{
        from t : TDMetamodel!TDConstraints
        to   u : umlbMetamodel!UMLBPredicate
              (name <-  'TimingCnstrntGuard',
              predicate <-
              t.effectsource.getNodePredicate()) }


helper context  TDMetamodel!TDNodeType
def : getNodePredicate() : String =
        if self.oclIsKindOf(TDMetamodel!Simple)
        then
              if  not self.timing.oclIsUndefined() then
                    self.SimpleCause()
                    -> concat(' & '+ self.SimpleGuard())
                    -> concat(self.SimpleCond())
              else
                    self.SimpleCause() ->
                    concat(self.SimpleCond())
              endif
        else if self.oclIsKindOf(TDMetamodel!OR_node)
              then  self.Or -> iterate(e; ret : String = '('|
              if e=self.Or.last() then
                  ret -> concat(e.getNodePredicate()+')')
```

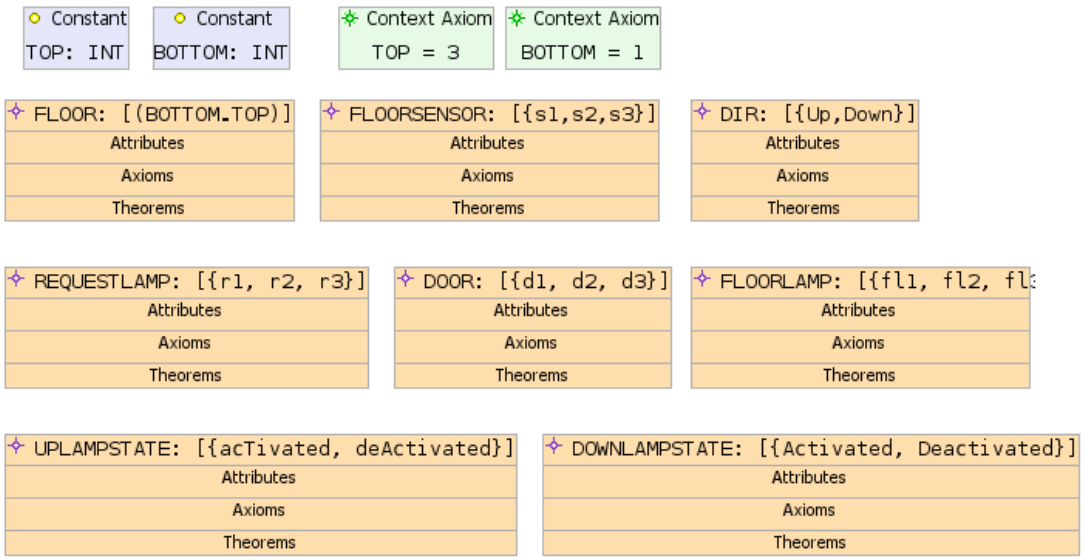
```
                else
                    ret -> concat(e.getNodePredicate()+') or (')
                endif)
            else if self.oclIsKindOf(TDMetamodel!AND_node)
                then self.And -> iterate(e; ret : String ='('|
                if e=self.And.last() then
                    ret -> concat(e.getNodePredicate()+')')
                else
                    ret -> concat(e.getNodePredicate()+ ') & (')
                endif)
                else  'unrecognised nodeType'
            endif endif endif;


helper context  TDMetamodel!TDNodeType
def  :  SimpleGuard() : String =
 '(gclock - xAssociationx.'
   + self.causesource.eContainer().eContainer().name
   + self.causesource.eContainer().name
   + 'Time >= '
   + self.timing.lowerlimit.toString() + ') '
   + ' & (gclock - xAssociationx.'
   + self.causesource.eContainer().eContainer().name
   + self.causesource.eContainer().name
   + 'Time <= '
   + self.timing.upperlimit.toString() + ')';


helper context  TDMetamodel!TDNodeType
def  :  SimpleCond() : String =
         self.predicates -> iterate(e; ret : String = '' |
                            ret -> concat(' & ' +e.predicate));


--  Add a cause as a guard with timing constraints
helper context  TDMetamodel!TDNodeType
def  :  SimpleCause() : String =
       self.causesource.eContainer().eContainer().name
```

```
+ '_state(xAssociationx) = '
+ self.causesource.eContainer().name;
```

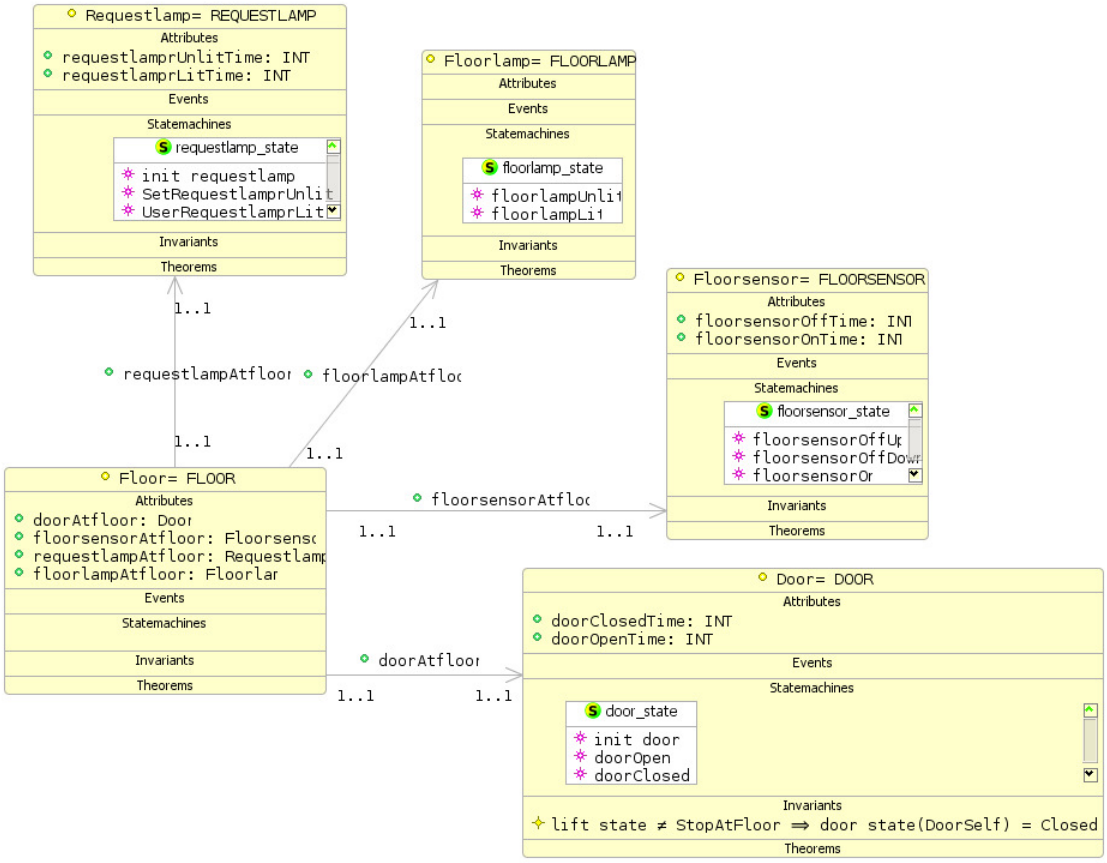
Appendix D. **UML-B and Event-B models from ATL Translation rules**

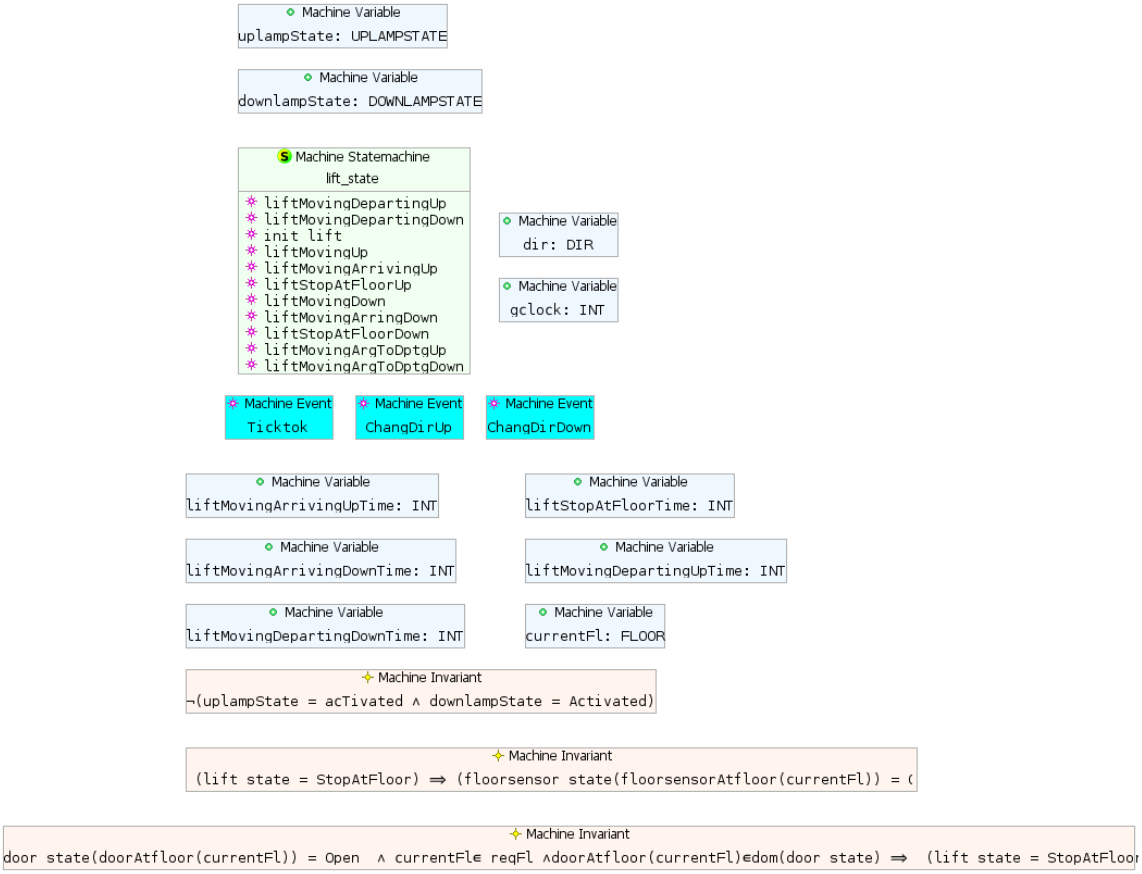
D.1 An UML-B model for the lift system: Package diagram

D.2 An UML-B model for the lift system: Context diagram

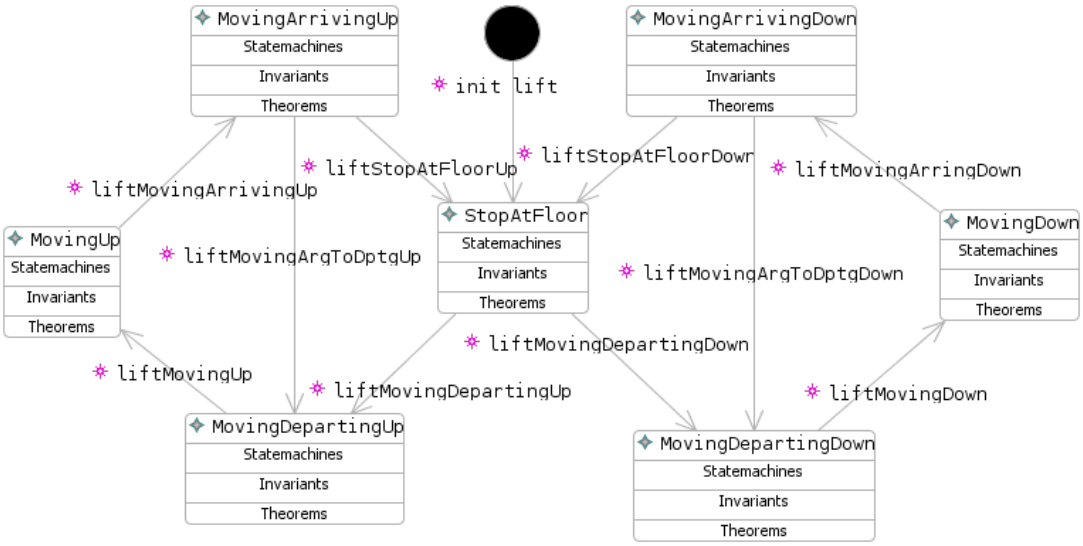


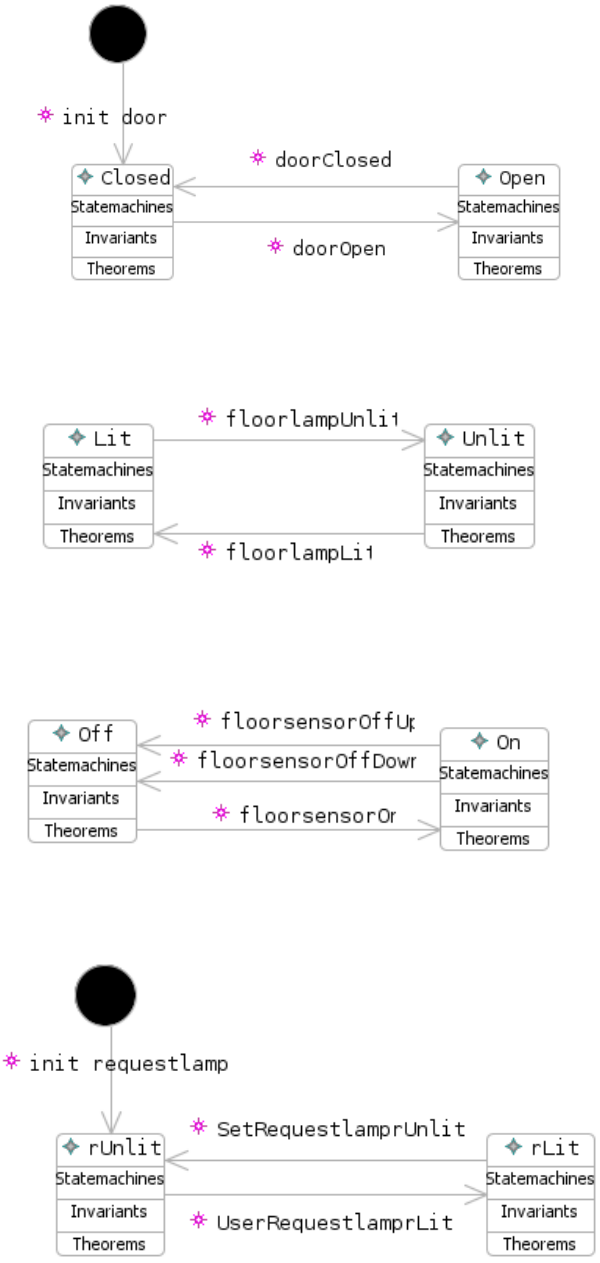
D.3 An UML-B model for the lift system: Class diagram





D.4 An UML-B model for the lift system: State diagram





D.5 An Event-B model is generated from an UML-B model

An Event-B model is generated from an UML-B model with additional information is illustrated below. The Event-B model composes of two contexts: `L_ctx` and `L_mch_implicitContext`, and one machine `L_mch`.

D.5.1 Context : L_ctx

```
context L_ctx

constants FLOOR // classType instances
          BOTTOM // utility constant
          TOP // utility constant
          s1 // enumeration constant
          s2 // enumeration constant
          s3 // enumeration constant
          r1 // enumeration constant
          r2 // enumeration constant
          r3 // enumeration constant
          d1 // enumeration constant
          d2 // enumeration constant
          d3 // enumeration constant
          Up // enumeration constant
          Down // enumeration constant
          acTivated // enumeration constant
          deActivated // enumeration constant
          Activated // enumeration constant
          Deactivated // enumeration constant
          fl1 // enumeration constant
          fl2 // enumeration constant
          fl3 // enumeration constant


sets FLOORSENSOR // ClassType
     REQUESTLAMP // ClassType
     DOOR // ClassType
     DIR // ClassType
     UPLAMPSTATE // ClassType
     DOWNLAMPSTATE // ClassType
     FLOORLAMP // ClassType


axioms
  @FLOORSENSOR.value FLOORSENSOR = {s1,s2,s3}
```

```
@REQUESTLAMP.value REQUESTLAMP = {r1, r2, r3}
@DOOR.value DOOR = {d1, d2, d3}
@DIR.value DIR = {Up,Down}
@UPLAMPSTATE.value UPLAMPSTATE = {acTivated, deActivated}
@DOWNLAMPSTATE.value
     DOWNLAMPSTATE = {Activated, Deactivated}
@FLOORLAMP.value FLOORLAMP = {fl1, fl2, fl3}
@BOTTOM.type BOTTOM ∈ ℤ
@TOP.type TOP ∈ ℤ
@s1.type s1 ∈ FLOORSENSOR
@s2.type s2 ∈ FLOORSENSOR
@s3.type s3 ∈ FLOORSENSOR
@r1.type r1 ∈ REQUESTLAMP
@r2.type r2 ∈ REQUESTLAMP
@r3.type r3 ∈ REQUESTLAMP
@d1.type d1 ∈ DOOR
@d2.type d2 ∈ DOOR
@d3.type d3 ∈ DOOR
@Up.type Up ∈ DIR
@Down.type Down ∈ DIR
@acTivated.type acTivated ∈ UPLAMPSTATE
@deActivated.type deActivated ∈ UPLAMPSTATE
@Activated.type Activated ∈ DOWNLAMPSTATE
@Deactivated.type Deactivated ∈ DOWNLAMPSTATE
@fl1.type fl1 ∈ FLOORLAMP
@fl2.type fl2 ∈ FLOORLAMP
@fl3.type fl3 ∈ FLOORLAMP
@FLOOR.value FLOOR = (BOTTOM_TOP)
@Axiom1 BOTTOM = 1
@Axiom TOP = 3
@s2.distinctFrom_s1 s2 ≠ s1
@s3.distinctFrom_s1 s3 ≠ s1
@s3.distinctFrom_s2 s3 ≠ s2
@r2.distinctFrom_r1 r2 ≠ r1
@r3.distinctFrom_r1 r3 ≠ r1
```

```
  @r3.distinctFrom_r2 r3 ≠ r2
  @d2.distinctFrom_d1 d2 ≠ d1
  @d3.distinctFrom_d1 d3 ≠ d1
  @d3.distinctFrom_d2 d3 ≠ d2
  @Down.distinctFrom_Up Down ≠ Up
  @deActivated.distinctFrom_acTivated deActivated ≠ acTivated
  @Deactivated.distinctFrom_Activated Deactivated ≠ Activated
  @fl2.distinctFrom_fl1 fl2 ≠ fl1
  @fl3.distinctFrom_fl1 fl3 ≠ fl1
  @fl3.distinctFrom_fl2 fl3 ≠ fl2
end
```

D.5.2 Context : L_mch_implicitContext

```
Context L_mch_implicitContext extends L_ctx

constants StopAtFloor // lift_state-state
          MovingDepartingUp // lift_state-state
          MovingDepartingDown // lift_state-state
          MovingUp // lift_state-state
          MovingArrivingUp // lift_state-state
          MovingDown // lift_state-state
          MovingArrivingDown // lift_state-state
          Door // class instances
          Closed // door_state-state
          Open // door_state-state
          Floor // class instances
          Floorlamp // class instances
          Lit // floorlamp_state-state
          Unlit // floorlamp_state-state
          Floorsensor // class instances
          Off // floorsensor_state-state
          On // floorsensor_state-state
          Requestlamp // class instances
          rLit // requestlamp_state-state
          rUnlit // requestlamp_state-state
```

```
sets lift_state_STATES // statemachine
     door_state_STATES // Door-statemachine
     floorlamp_state_STATES // Floorlamp-statemachine
     floorsensor_state_STATES // Floorsensor-statemachine
     requestlamp_state_STATES // Requestlamp-statemachine


axioms
  @lift_state_STATES.value lift_state_STATES =

{StopAtFloor,MovingDepartingUp,MovingDepartingDown,MovingUp,
   MovingArrivingUp,MovingDown,MovingArrivingDown}
  @door_state_STATES.value door_state_STATES = {Closed,Open}
  @floorlamp_state_STATES.value
  floorlamp_state_STATES = {Lit,Unlit}
  @floorsensor_state_STATES.value
  floorsensor_state_STATES = {Off,On}
  @requestlamp_state_STATES.value
  requestlamp_state_STATES = {rLit,rUnlit}
  @StopAtFloor.type StopAtFloor ∈ lift_state_STATES
  @MovingDepartingUp.type
  MovingDepartingUp ∈ lift_state_STATES
  @MovingDepartingDown.type
  MovingDepartingDown ∈ lift_state_STATES
  @MovingUp.type MovingUp ∈ lift_state_STATES
  @MovingArrivingUp.type MovingArrivingUp ∈ lift_state_STATES
  @MovingDown.type MovingDown ∈ lift_state_STATES
  @MovingArrivingDown.type
  MovingArrivingDown ∈ lift_state_STATES
  @Door.value Door = DOOR
  @Closed.type Closed ∈ door_state_STATES
  @Open.type Open ∈ door_state_STATES
  @Floor.value Floor = FLOOR
  @Floorlamp.value Floorlamp = FLOORLAMP
  @Lit.type Lit ∈ floorlamp_state_STATES
  @Unlit.type Unlit ∈ floorlamp_state_STATES
```

```
@Floorsensor.value Floorsensor = FLOORSENSOR
@Off.type Off ∈ floorsensor_state_STATES
@On.type On ∈ floorsensor_state_STATES
@Requestlamp.value Requestlamp = REQUESTLAMP
@rLit.type rLit ∈ requestlamp_state_STATES
@rUnlit.type rUnlit ∈ requestlamp_state_STATES
@distinctStates MovingDepartingUp,StopAtFloor :
MovingDepartingUp ≠ StopAtFloor
@distinctStates MovingDepartingDown,StopAtFloor:
MovingDepartingDown ≠ StopAtFloor
@distinctStates MovingDepartingDown,MovingDepartingUp:
MovingDepartingDown ≠ MovingDepartingUp
@distinctStates MovingUp,StopAtFloor: MovingUp ≠ StopAtFloor
@distinctStates MovingUp,MovingDepartingUp:
MovingUp ≠ MovingDepartingUp
@distinctStates MovingUp,MovingDepartingDown:
MovingUp ≠ MovingDepartingDown
@distinctStates MovingArrivingUp,StopAtFloor:
MovingArrivingUp ≠ StopAtFloor
@distinctStates MovingArrivingUp,MovingDepartingUp:
MovingArrivingUp ≠ MovingDepartingUp
@distinctStates MovingArrivingUp,MovingDepartingDown:
MovingArrivingUp ≠ MovingDepartingDown
@distinctStates MovingArrivingUp,MovingUp:
MovingArrivingUp ≠ MovingUp
@distinctStates MovingDown,StopAtFloor:
MovingDown ≠ StopAtFloor
@distinctStates MovingDown,MovingDepartingUp:
MovingDown ≠ MovingDepartingUp
@distinctStates MovingDown,MovingDepartingDown:
MovingDown ≠ MovingDepartingDown
@distinctStates MovingDown,MovingUp:
MovingDown ≠ MovingUp
@distinctStates MovingDown,MovingArrivingUp:
MovingDown ≠ MovingArrivingUp
@distinctStates MovingArrivingDown,StopAtFloor:
MovingArrivingDown ≠ StopAtFloor
```

```
  @distinctStates MovingArrivingDown,MovingDepartingUp:
 MovingArrivingDown ≠ MovingDepartingUp
  @distinctStates MovingArrivingDown,MovingDepartingDown:
 MovingArrivingDown ≠ MovingDepartingDown
  @distinctStates MovingArrivingDown,MovingUp:
  MovingArrivingDown ≠ MovingUp
  @distinctStates MovingArrivingDown,MovingArrivingUp:
 MovingArrivingDown ≠ MovingArrivingUp
  @distinctStates MovingArrivingDown,MovingDown:
  MovingArrivingDown ≠ MovingDown
  @distinctStates Open,Closed: Open ≠ Closed
  @distinctStates Unlit,Lit: Unlit ≠ Lit
  @distinctStates On,Off: On ≠ Off
  @distinctStates rUnlit,rLit: rUnlit ≠ rLit
end
```

D.5.3 Machine : L_mch

```
machine L_mch sees L_mch_implicitContext

variables reqFl // utility variable
          dir // utility variable
          currentFl // utility variable
          uplampState // utility variable
          downlampState // utility variable
          liftStopAtFloorTime // utility variable
          liftMovingUpTime // utility variable
          liftMovingDownTime // utility variable
          liftMovingDepartingUpTime // utility variable
          liftMovingDepartingDownTime // utility variable
          liftMovingArrivingUpTime // utility variable
          liftMovingArrivingDownTime // utility variable
          gclock // utility variable
          lift_state // statemachine belonging to the machine
          doorClosedTime // attribute of Door
```

```
            doorOpenTime // attribute of Door
            door_state // statemachine belonging to class, Door
            doorAtfloor // attribute of Floor
            floorlampAtfloor // attribute of Floor
            floorsensorAtfloor // attribute of Floor
            requestlampAtfloor // attribute of Floor
            floorlamp_state // statemachine belonging to class,
                            Floorlamp
            floorsensorOffTime // attribute of Floorsensor
            floorsensorOnTime // attribute of Floorsensor
            floorsensor_state
               // statemachine belonging to class, Floorsensor
            requestlamp_state
               // statemachine belonging to class, Requestlamp
            requestlamprUnlitTime
            requestlamprLitTime
            floorlampUnlitTime
            floorlampLitTime


invariants
  @reqFl.type reqFl ∈ ℙ(FLOOR)
  @dir.type dir ∈ DIR
  @currentFl.type currentFl ∈ FLOOR
  @uplampState.type uplampState ∈ UPLAMPSTATE
  @downlampState.type downlampState ∈ DOWNLAMPSTATE
  @liftStopAtFloorTime.type liftStopAtFloorTime ∈ ℤ
  @liftMovingUpTime.type liftMovingUpTime ∈ ℤ
  @liftMovingDownTime.type liftMovingDownTime ∈ ℤ
  @liftMovingDepartingUpTime.type
 liftMovingDepartingUpTime ∈ ℤ
  @liftMovingDepartingDownTime.type
 liftMovingDepartingDownTime ∈ ℤ
  @liftMovingArrivingUpTime.type liftMovingArrivingUpTime ∈ ℤ
  @liftMovingArrivingDownTime.type
 liftMovingArrivingDownTime ∈ ℤ
```

```
@gclock.type gclock ∈ ℤ
@lift_state.type lift_state ∈ lift_state_STATES
@doorClosedTime.type doorClosedTime ∈ Door → ℤ
@doorOpenTime.type doorOpenTime ∈ Door → ℤ
@door_state.type door_state ∈ Door → door_state_STATES
@doorAtfloor.type doorAtfloor ∈ Floor ⤖ Door
@floorlampAtfloor.type
floorlampAtfloor ∈ Floor ⤖ Floorlamp
@floorsensorAtfloor.type
floorsensorAtfloor ∈ Floor ⤖ Floorsensor
@requestlampAtfloor.type
requestlampAtfloor ∈ Floor ⤖ Requestlamp
@floorlamp_state.type
      floorlamp_state ∈ Floorlamp → floorlamp_state_STATES
@floorsensorOffTime.type
floorsensorOffTime ∈ Floorsensor → ℤ
@floorsensorOnTime.type floorsensorOnTime ∈ Floorsensor → ℤ
@requestlamprUnlitTime.type requestlamprUnlitTime ∈
                        Requestlamp → ℤ
@requestlamprLitTime.type requestlamprLitTime ∈
                        Requestlamp → ℤ
@floorsensor_state.type floorsensor_state
                  ∈ Floorsensor → floorsensor_state_STATES
@requestlamp_state.type requestlamp_state
                  ∈ Requestlamp → requestlamp_state_STATES
@Invariant2 (lift_state = StopAtFloor) ⇒
      (floorsensor_state(floorsensorAtfloor(currentFl)) = On)
@Invariant3 door_state(doorAtfloor(currentFl)) = Open
      ∧ currentFl∈ reqFl
      ∧ doorAtfloor(currentFl)∈
      dom(door_state) ⇒ (lift_state = StopAtFloor)
@Invariant1 ∀d·((d∈Door)⇒(lift_state ≠ StopAtFloor ⇒
              door_state(d) = Closed))
@inv1 floorlampUnlitTime ∈ Floorlamp → ℤ
```

```
    @inv5 floorlampLitTime ∈ Floorlamp → ℤ
    @Invariant4 ¬(uplampState = acTivated ∧
                  downlampState = Activated)

events
  event INITIALISATION
    then
      @reqFl.init reqFl :∈ ℙ(FLOOR)
      @dir.init dir :∈ DIR
      @currentFl.init currentFl ≔ BOTTOM
      @uplampState.init uplampState ≔ deActivated
      @downlampState.init downlampState ≔ Deactivated
      @liftStopAtFloorTime.init liftStopAtFloorTime ≔ 0
      @liftMovingDepartingUpTime.init
       liftMovingDepartingUpTime ≔ 0
      @liftMovingDepartingDownTime.init
       liftMovingDepartingDownTime ≔ 0
      @liftMovingArrivingUpTime.init
       liftMovingArrivingUpTime ≔ 0
      @liftMovingArrivingDownTime.init
       liftMovingArrivingDownTime ≔ 0
      @gclock.init gclock ≔ 0
      @lift_state.init lift_state ≔ StopAtFloor
      @doorClosedTime.init doorClosedTime ≔ Door × {0}
      @doorOpenTime.init doorOpenTime ≔ Door × {0}
      @door_state.init door_state ≔ Door × {Closed}
      @doorAtfloor.init doorAtfloor ≔ {1 ↦ d1, 2 ↦ d2, 3 ↦ d3}
       // doorAtfloor :∈ Floor ⤖ Door
      @floorlampAtfloor.init floorlampAtfloor ≔
                          {1 ↦ fl1, 2 ↦ fl2, 3 ↦ fl3}
       // floorlampAtfloor :∈ Floor ⤖ Floorlamp
      @floorsensorAtfloor.init floorsensorAtfloor ≔
                          {1 ↦ s1, 2 ↦ s2, 3 ↦ s3}
```

```
      // floorsensorAtfloor :∈ Floor ⤖ Floorsensor
    @requestlampAtfloor.init requestlampAtfloor ≔
                        {1 ↦ r1, 2 ↦ r2, 3 ↦ r3}
      // requestlampAtfloor :∈ Floor ⤖ Requestlamp
    @floorlamp_state.init floorlamp_state ≔
                        {fl1 ↦ Lit, fl2 ↦ Unlit, fl3 ↦ Unlit}
      // floorlamp_state :∈
            Floorlamp → floorlamp_state_STATES
    @floorsensorOffTime.init floorsensorOffTime ≔
                              Floorsensor × {0}
    @floorsensorOnTime.init floorsensorOnTime ≔
                              Floorsensor × {0}
    @floorsensor_state.init floorsensor_state ≔
                        {s1 ↦ On, s2 ↦ Off, s3 ↦ Off}
// floorsensor_state :∈
            Floorsensor → floorsensor_state_STATES
    @requestlamp_state.init requestlamp_state ≔
                        {r1 ↦ rUnlit, r2 ↦ rUnlit, r3 ↦ rUnlit}
// requestlamp_state ≔ Requestlamp × {rUnlit}
    @act1 requestlamprUnlitTime ≔ Requestlamp × {0}
    @act2 requestlamprLitTime ≔ Requestlamp × {0}
    @act3 floorlampUnlitTime ≔ Floorlamp × {0}
end


event UserRequestlamprLit
  any RequestlampSelf // contextual instance of
                      class Requestlamp
      f
  where
    @f.type f ∈ FLOOR
    @RequestSelf.type RequestlampSelf ∈ Requestlamp
    @grd1 requestlamp_state(RequestlampSelf) = rUnlit
  then
    @requestlamprLit.Action1 reqFl ≔ reqFl ∪ {f}
```

```
      @requestlamp_state_enterState_rLit
      requestlamp_state(requestlampAtfloor(f)) ≔ rLit
  end


  event SetRequestlamprUnlit
    any RequestlampSelf // contextual instance of
                        class Requestlamp
        f
    where
      @f.type f ∈ FLOOR
      @grd1 f = currentFl
      @RequestSelf.type RequestlampSelf ∈ Requestlamp
      @requestlamprUnlit.Guard1 lift_state = StopAtFloor
      @requestlamprUnlit.TimingCnstrntGuard
       (gclock − liftStopAtFloorTime ≥ 2) ∧
       (gclock − liftStopAtFloorTime ≤ 4)
      @grd2  requestlampAtfloor(f) = RequestlampSelf
      @requestlamp_state_isin_rLit
        requestlamp_state(RequestlampSelf) = rLit
    then
      @requestlamprUnlit.Action2
   requestlamprUnlitTime(RequestlampSelf) ≔ gclock
      @requestlamp_state_enterState_rUnlit
   requestlamp_state(requestlampAtfloor(currentFl)) ≔ rUnlit
  end


event doorOpen
    any DoorSelf // contextual instance of class Door
        f
    where
      @f.type f ∈ FLOOR
      @DoorSelf.type DoorSelf ∈ Door
      @doorOpen.TimingCnstrntGuard lift_state = StopAtFloor
                  ∧ currentFl ∈ reqFl
                  ∧(gclock − liftStopAtFloorTime ≥ 1 )
```

```
                    ∧(gclock − liftStopAtFloorTime ≤ 5 )
      @doorOpen.Guard4 doorAtfloor(f) = DoorSelf
      @doorOpen.Guard3 f ∈ reqFl ∧ f = currentFl
      @door_state_isin_Closed door_state(DoorSelf) = Closed
    then
      @doorOpen.Action2 doorOpenTime(DoorSelf) ≔ gclock
      @door_state_enterState_Open door_state(DoorSelf) ≔ Open
  end

event doorClosed
    any DoorSelf // contextual instance of class Door
    where
      @DoorSelf.type DoorSelf ∈ Door
      @door_state_isin_Open door_state(DoorSelf) = Open
      @grd1 lift_state = StopAtFloor
    then
      @doorClosed.Action2 doorClosedTime(DoorSelf) ≔ gclock
      @door_state_enterState_Closed
                        door_state(DoorSelf) ≔ Closed
      @doorClosed.Action1 reqFl ≔ reqFl ∖ {currentFl}
  end


event liftMovingDepartingUp
    any f
    where
      @f.type f ∈ FLOOR
      @lift_state_isin_StopAtFloor lift_state = StopAtFloor
      @liftMovingDepartingUp.Guard5
  requestlamp_state(requestlampAtfloor(f)) = rLit
  ∧ f > currentFl
      @liftMovingDepartingUp.Guard2 currentFl ∉ reqFl
      @liftMovingDepartingUp.Guard1 f ∈ reqFl
      @grd1 ∀d·d∈Door ⇒ door_state(d) = Closed
      @liftMovingDepartingUp.TimingCnstrntGuard
  door_state(doorAtfloor(currentFl)) = Closed
```

```
  ∧ (gclock −doorClosedTime((doorAtfloor(currentFl))) ≥ 1)
  ∧ (gclock −doorClosedTime((doorAtfloor(currentFl))) ≤ 5)
      @liftMovingDepartingUp.Guard3 dir = Up
    then
      @lift_state_enterState_StopAtFloor
              lift_state ≔ MovingDepartingUp
      @liftMovingDepartingUp.Action3
              liftMovingDepartingUpTime ≔ gclock
      @liftMovingDepartingUp.Action2
              downlampState ≔ Deactivated
      @liftMovingDepartingUp.Action1 uplampState ≔ acTivated
  end


event liftMovingArgToDptgUp
    any f
    where
      @f.type f ∈ FLOOR
      @liftMovingArgToDptgUp.Guard1 f ∈ FLOOR
      @liftMovingArgToDptgUp.Guard2 f ∈ reqFl ∧ f > currentFl
      @liftMovingArgToDptgUp.Guard3 currentFl ∉ reqFl
      @lift_state_isin_MovingArrivingUp
       lift_state = MovingArrivingUp
      @liftMovingArgToDptgUp.Guard4 dir = Up
    then
      @lift_state_enterState_MovingArrivingUp
       lift_state ≔ MovingDepartingUp
      @liftMovingArgToDptgUp.Action1
       liftMovingDepartingUpTime ≔ gclock
      @act1 downlampState ≔ Deactivated
      @act2 uplampState ≔ acTivated
  end


event liftMovingUp
    any f
```

```
    where
      @f.type f ∈ FLOOR
      @liftMovingUp.Guard1 f ∈ reqFl ∧ f > currentFl
      @liftMovingUp.Guard2 currentFl ∉ reqFl
      @lift_state_isin_MovingDepartingUp
       lift_state = MovingDepartingUp
      @liftMovingUp.Guard3
       floorsensor_state(floorsensorAtfloor(currentFl)) = Off
      @liftMovingUp.Guard4 dir = Up
    then
      @liftMovingUp.Action1 liftMovingUpTime ≔ gclock
      @lift_state_enterState_MovingDepartingUp
       lift_state ≔ MovingUp
  end


event liftMovingDepartingDown
    any f
    where
      @f.type f ∈ FLOOR
      @liftMovingDepartingDown.Guard1 f ∈ reqFl
      @liftMovingDepartingDown.Guard2 currentFl ∉ reqFl
      @liftMovingDepartingDown.Guard3 dir = Down
      @grd1 ∀d·d∈Door ⇒ door_state(d) = Closed
      @liftMovingDepartingDown.Guard4
       requestlamp_state(requestlampAtfloor(f)) = rLit
       ∧ f < currentFl
      @liftMovingDepartingDown.TimingCnstrntGuard
       door_state(doorAtfloor(currentFl)) = Closed
  ∧(gclock -doorClosedTime((doorAtfloor(currentFl))) ≥ 1)
  ∧(gclock -doorClosedTime((doorAtfloor(currentFl))) ≤ 5)
      @lift_state_isin_StopAtFloor lift_state = StopAtFloor
    then
      @liftMovingDepartingDown.Action1
       uplampState ≔ deActivated
      @lift_state_enterState_StopAtFloor
```

```
        lift_state ≔ MovingDepartingDown
      @liftMovingDepartingDown.Action2
        downlampState ≔ Activated
      @liftMovingDepartingDown.Action3
        liftMovingDepartingDownTime ≔ gclock
  end


event liftMovingArgToDptgDown
    any f
    where
      @f.type f ∈ FLOOR
      @liftMovingArgToDptgDown.Guard1 f ∈ FLOOR
      @liftMovingArgToDptgDown.Guard3 currentFl ∉ reqFl
      @liftMovingArgToDptgDown.Guard2 f ∈ reqFl ∧
                                      f < currentFl
      @liftMovingArgToDptgDown.Guard4 dir = Down
      @lift_state_isin_MovingArrivingDown
        lift_state = MovingArrivingDown
    then
      @lift_state_enterState_MovingArrivingDown
        lift_state ≔ MovingDepartingDown
      @liftMovingArgToDptgDown.Action1
        liftMovingDepartingDownTime ≔ gclock
      @act1 downlampState ≔ Activated
      @act2 uplampState ≔ deActivated
  end


event liftMovingDown
    any f
    where
      @f.type f ∈ FLOOR
      @liftMovingDown.Guard1 f ∈ reqFl ∧ f > currentFl
      @liftMovingDown.Guard2 currentFl ∉ reqFl
      @lift_state_isin_MovingDepartingDown
```

```
        lift_state = MovingDepartingDown
      @liftMovingDown.Guard3
        floorsensor_state(floorsensorAtfloor(currentFl)) = Off
      @liftMovingDown.Guard4 dir = Down
  then
      @liftMovingDown.Action1 liftMovingDownTime ≔ gclock
      @lift_state_enterState_MovingDepartingDown
        lift_state ≔ MovingDown
  end


event floorsensorOn
    any FloorsensorSelf // contextual instance of
                        class Floorsensor
        f
    where
      @grd1 f ∈ FLOOR
      @FloorsensorSelf.type FloorsensorSelf ∈ Floorsensor
      @floorsensor_state_isin_Off
        floorsensor_state(FloorsensorSelf) = Off
      @floorsensorOn.Guard1
        floorsensorAtfloor~(FloorsensorSelf) = currentFl
      @floorsensorOn.TimingCnstrntGuard
        (lift_state = MovingArrivingUp
        ∧ dir = Up ∧ f = currentFl
        ∧(gclock − liftMovingArrivingUpTime ≥ 2)
        ∧(gclock − liftMovingArrivingUpTime ≤ 5))
        ∨
         (lift_state = MovingArrivingDown
        ∧dir = Down ∧ f = currentFl
        ∧(gclock − liftMovingArrivingDownTime ≥ 2)
        ∧(gclock − liftMovingArrivingDownTime ≤ 5))
    then
      @floorsensorOn.Action1
        floorsensorOnTime(FloorsensorSelf) ≔ gclock
      @floorsensor_state_enterState_On
```

```
        floorsensor_state(FloorsensorSelf) ≔ On
  end


event floorsensorOffUp
    any FloorsensorSelf // contextual instance of class
 Floorsensor
        f
    where
      @l.type f ∈ FLOOR
      @FloorsensorSelf.type FloorsensorSelf ∈ Floorsensor
      @floorsensor_state_isin_On
           floorsensor_state(FloorsensorSelf) = On
      @floorsensorOffUp.TimingCnstrntGuard
       lift_state = MovingDepartingUp
       ∧(gclock − liftMovingDepartingUpTime ≥ 2)
       ∧(gclock − liftMovingDepartingUpTime ≤ 5)
       ∧ f = currentFl ∧ dir = Up
    then
      @floorsensorOffUp.Action1
       floorsensorOffTime(FloorsensorSelf) ≔  gclock
      @floorsensor_state_enterState_Off
       floorsensor_state(FloorsensorSelf) ≔ Off
      @floorsensorOffUp.Action2
       lift_state ≔ MovingUp
  end


event floorsensorOffDown
    any FloorsensorSelf // contextual instance of
class Floorsensor
        f
    where
      @l.type f ∈ FLOOR
      @FloorsensorSelf.type FloorsensorSelf ∈ Floorsensor
      @floorsensor_state_isin_On
```

```
                floorsensor_state(FloorsensorSelf) = On
          @floorsensorOffDown.TimingCnstrntGuard
           lift_state = MovingDepartingDown
           ∧(gclock - liftMovingDepartingDownTime ≥ 2)
           ∧(gclock - liftMovingDepartingDownTime ≤ 5)
           ∧ f = currentFl ∧ dir = Down
        then
          @floorsensor_state_enterState_Off
           floorsensor_state(FloorsensorSelf) ≔ Off
          @floorsensorOffDown.Action2 lift_state ≔ MovingDown
          @floorsensorOffDown.Action1
           floorsensorOffTime(FloorsensorSelf) ≔ gclock
      end


    event floorlampUnlit
        any FloorlampSelf // contextual instance of
    class Floorlamp
            f
        where
          @f.type f ∈ FLOOR
          @FloorlampSelf.type FloorlampSelf ∈ Floorlamp
          @floorlamp_state_isin_Lit
            floorlamp_state(floorlampAtfloor(currentFl)) = Lit
                 // floorlamp_state(FloorlampSelf) = Lit
          @floorlampUnlit.Guard3
            floorsensor_state(floorsensorAtfloor(currentFl)) = Off
          @floorlampUnlit.TimingCnstrntGuard f = currentFl ∧
(gclock - floorsensorOffTime((floorsensorAtfloor(currentFl))) ≥ 2)
 ∧
(gclock - floorsensorOffTime((floorsensorAtfloor(currentFl))) ≤ 4)
          @floorlampUnlit.Guard1
           floorlampAtfloor~(FloorlampSelf) =  currentFl
        then
          @floorlampUnlit.Action2
           floorlampUnlitTime(FloorlampSelf) ≔ gclock
```

```
            @floorlamp_state_enterState_Unlit
             floorlamp_state(floorlampAtfloor(currentFl)) ≔ Unlit
        end


    event floorlampLit
        any FloorlampSelf // contextual instance of
    class Floorlamp
             f
        where
          @f.type f ∈ FLOOR
          @FloorlampSelf.type FloorlampSelf ∈ Floorlamp
          @floorlampLit.Guard1
           floorlampAtfloor~(FloorlampSelf) = currentFl
          @floorlampLit.TimingCnstrntGuard f = currentFl ∧
(gclock − floorsensorOnTime((floorsensorAtfloor(currentFl))) ≥ 2)
∧
(gclock − floorsensorOnTime((floorsensorAtfloor(currentFl))) ≤ 4)
          @floorlampLit.Guard2
             floorsensor_state(floorsensorAtfloor(currentFl)) = On
          @floorlamp_state_isin_Unlit
             floorlamp_state(FloorlampSelf) = Unlit
        then
          @floorlampLit.Action2
              floorlampLitTime(FloorlampSelf) ≔ gclock
          @floorlamp_state_enterState_Lit
             floorlamp_state(FloorlampSelf) ≔ Lit
        end


    event liftMovingArrivingUp
        any f
        where
          @f.type f ∈ FLOOR
          @liftMovingArrivingUp.Guard4 dir = Up
          @liftMovingArrivingUp.Guard3 currentFl ∉ reqFl
```

```
        @liftMovingArrivingUp.Guard2 f ∈ reqFl ∧ f > currentFl
        @liftMovingArrivingUp.Guard1 f ∈ FLOOR
        @lift_state_isin_MovingUp lift_state = MovingUp
        @grd1
         floorlamp_state(floorlampAtfloor(currentFl)) = Unlit
              // manually additional guards
      then
        @lift_state_enterState_MovingUp
         lift_state ≔ MovingArrivingUp
        @liftMovingArrivingUp.Action2
         liftMovingArrivingUpTime ≔ gclock
        @liftMovingArrivingUp.Action1 currentFl ≔ currentFl + 1
    end


  event liftMovingArringDown
      any f
      where
        @f.type f ∈ FLOOR
        @lift_state_isin_MovingDown lift_state = MovingDown
        @liftMovingArringDown.Guard4 dir = Down
        @liftMovingArringDown.Guard3 currentFl ∉ reqFl
        @liftMovingArringDown.Guard1 f ∈ FLOOR
        @liftMovingArringDown.Guard2 f ∈ reqFl ∧ f < currentFl
        @grd2
         floorlamp_state(floorlampAtfloor(currentFl)) = Unlit
              // manually added
      then
        @liftMovingArringDown.Action1 currentFl ≔ currentFl − 1
        @liftMovingArringDown.Action2
         liftMovingArrivingDownTime ≔ gclock
        @lift_state_enterState_MovingDown
         lift_state ≔ MovingArrivingDown
    end
```

```
event liftStopAtFloorUp
    any f
    where
      @f.type f ∈ FLOOR
      @liftStopAtFloorUp.Guard1 f ∈ reqFl
      @liftStopAtFloorUp.TimingCnstrntGuard
(gclock −floorsensorOnTime((floorsensorAtfloor(currentFl))) ≥ 2)
∧
(gclock −floorsensorOnTime((floorsensorAtfloor(currentFl))) ≤ 5)
      @liftStopAtFloorUp.Guard3
       floorsensor_state(floorsensorAtfloor(currentFl))= On ∧
       f = currentFl ∧ f ∈ reqFl
      @lift_state_isin_MovingArrivingUp
    lift_state = MovingArrivingUp
      @liftStopAtFloorUp.Guard2 f = currentFl
    then
      @lift_state_enterState_MovingArrivingUp
       lift_state ≔ StopAtFloor
      @liftStopAtFloorUp.Action3 downlampState ≔ Deactivated
      @liftStopAtFloorUp.Action1 liftStopAtFloorTime ≔ gclock
      @liftStopAtFloorUp.Action2 uplampState ≔ deActivated
  end


event liftStopAtFloorDown
    any f
    where
      @f.type f ∈ FLOOR
      @liftStopAtFloorDown.Guard2 f = currentFl
      @liftStopAtFloorDown.Guard1 f ∈ reqFl
      @liftStopAtFloorDown.TimingCnstrntGuard (gclock−
floorsensorOnTime((floorsensorAtfloor(currentFl)))≥ 2) ∧
(gclock − floorsensorOnTime((floorsensorAtfloor(currentFl))) ≤ 5)
      @liftStopAtFloorDown.Guard3
      floorsensor_state(floorsensorAtfloor(currentFl)) = On ∧
       f = currentFl ∧ f ∈ reqFl
```

```
      @lift_state_isin_MovingArrivingDown
       lift_state = MovingArrivingDown
    then
      @liftStopAtFloorDown.Action1
       liftStopAtFloorTime ≔ gclock
      @lift_state_enterState_MovingArrivingDown
       lift_state ≔ StopAtFloor
      @liftStopAtFloorDown.Action2 uplampState ≔ deActivated
      @liftStopAtFloorDown.Action3 downlampState ≔ Deactivated
  end


event ChangDirUp
    any f
    where
      @f.type f ∈ FLOOR
      @ChangDirUp.Guard1 f ∈ reqFl ∧ f > currentFl
      @ChangDirUp.Guard5 dir = Down
      @ChangDirUp.Guard4 lift_state = StopAtFloor
      @ChangDirUp.Guard3 reqFl ≠ ∅
      @ChangDirUp.Guard2 currentFl ∉ reqFl
    then
      @ChangDirUp.Action1 dir ≔ Up
  end


event ChangDirDown
    any f
    where
      @f.type f ∈ FLOOR
      @ChangDirDown.Guard5 dir = Up
      @ChangDirDown.Guard3 reqFl ≠ ∅
      @ChangDirDown.Guard4 lift_state = StopAtFloor
      @ChangDirDown.Guard1 f ∈ reqFl ∧ f < currentFl
      @ChangDirDown.Guard2 currentFl ∉ reqFl
    then
```

```
            @ChangDirDown.Action1 dir ≔ Down
        end


    event Ticktok
        where
            @Ticktok.Guard1 // Requestlamp Unlit
            lift_state = StopAtFloor
      ∧ requestlamp_state(requestlampAtfloor(currentFl)) = rLit
      ∧ (((gclock − liftStopAtFloorTime) ≥ 2)
      ∧ ((gclock − liftStopAtFloorTime) ≤ 4))
            ⇒
            gclock − liftStopAtFloorTime < 4

           @Ticktok.Guard10  // Floorlamp Unlit
           (floorlamp_state(floorlampAtfloor(currentFl)) = Lit
      ∧ floorsensor_state(floorsensorAtfloor(currentFl)) = Off
      ∧
(gclock − floorsensorOffTime(floorsensorAtfloor(currentFl))≥ 2)
∧
(gclock − floorsensorOffTime(floorsensorAtfloor(currentFl))≤ 4))
⇒
gclock − floorsensorOffTime(floorsensorAtfloor(currentFl)) <

            @Ticktok.Guard9 // Door open
            (lift_state = StopAtFloor
      ∧ door_state(doorAtfloor(currentFl)) = Closed
      ∧ currentFl ∈ reqFl
      ∧(gclock − liftStopAtFloorTime ≥ 1)
      ∧(gclock − liftStopAtFloorTime ≤ 5))
            ⇒
            gclock − liftStopAtFloorTime < 5

            @Ticktok.Guard8 // Lift Moving Departing Up and Down
            (door_state(doorAtfloor(currentFl)) = Closed
       ∧ lift_state = StopAtFloor
```

```
        ∧(gclock - doorClosedTime(doorAtfloor(currentFl)) ≥ 1)
        ∧(gclock - doorClosedTime(doorAtfloor(currentFl)) ≤ 5))
             ⇒
             gclock - doorClosedTime(doorAtfloor(currentFl)) < 5

            @Ticktok.Guard7 // Floorsensor Off (when lift
                               Moving Departing Down)
            (lift_state = MovingDepartingDown
        ∧ floorsensor_state(floorsensorAtfloor(currentFl)) = On
        ∧ ((gclock - liftMovingDepartingDownTime) ≥ 2)
        ∧ ((gclock - liftMovingDepartingDownTime) ≤ 5))
              ⇒
             gclock - liftMovingDepartingDownTime < 5

            @Ticktok.Guard6 // Floor sensor Off (when lift
                               Moving Departing Up)
            (lift_state = MovingDepartingUp ∧
             floorsensor_state(floorsensorAtfloor(currentFl)) = On
        ∧ ((gclock - liftMovingDepartingUpTime) ≥ 2)
        ∧((gclock - liftMovingDepartingUpTime) ≤ 5))
             ⇒
             gclock - liftMovingDepartingUpTime < 5

            @Ticktok.Guard5 // Lift Stop At Floor
            (floorsensor_state(floorsensorAtfloor(currentFl)) = On
        ∧
            (lift_state = MovingArrivingUp ∨
             lift_state = MovingArrivingDown)
      ∧    (currentFl ∈ reqFl)∧
(gclock - floorsensorOnTime(floorsensorAtfloor(currentFl)) ≥ 1)
∧
(gclock - floorsensorOnTime(floorsensorAtfloor(currentFl)) ≤ 5))
⇒
gclock - floorsensorOnTime(floorsensorAtfloor(currentFl)) < 5

            @Ticktok.Guard4 // Floorlamp Lit
```

```
          (floorlamp_state(floorlampAtfloor(currentFl)) = Unlit
      ∧floorsensor_state(floorsensorAtfloor(currentFl)) = On
      ∧
(gclock − floorsensorOnTime(floorsensorAtfloor(currentFl)) ≥ 2)
∧
(gclock − floorsensorOnTime(floorsensorAtfloor(currentFl)) ≤ 4))
⇒
gclock − floorsensorOnTime(floorsensorAtfloor(currentFl)) < 4

      @Ticktok.Guard3 // Floorsensor On (when lift Moving Arriving
                                 Down)
          (lift_state = MovingArrivingDown
      ∧ floorsensor_state(floorsensorAtfloor(currentFl)) = Off
      ∧ ((gclock − liftMovingArrivingDownTime) ≥ 2)
      ∧ ((gclock − liftMovingArrivingDownTime) ≤ 5))
           ⇒
           gclock − liftMovingArrivingDownTime < 5


      @Ticktok.Guard2 // Floorsensor On (when lift Moving
                        Arriving Up)
      (lift_state = MovingArrivingUp
      ∧ floorsensor_state(floorsensorAtfloor(currentFl)) = Off
      ∧((gclock − liftMovingArrivingUpTime) ≥ 2)
      ∧((gclock − liftMovingArrivingUpTime) ≤ 5))
            ⇒
            gclock − liftMovingArrivingUpTime < 5
        then
           @Ticktok.Action1 gclock ≔ gclock + 1
     end
   end
```

Appendix E. **KAOS Textual Translation rules**

E.1 Translation rules for creating a KAOS goal from segments defined with `CauseEffectArrow`

1. Rule : **TKParamGuardValue**

TParamGuardValue(Segment) →
<LET> exp = **TClass**(**TObj**(**TObjSt**(Segment)))
<IF>**THasParam**(exp)
<THEN>
"∀" +
TWriteAllParamsLst(**TAllParam**(exp))
<ELSE><SKIP>
<ENDIF>

TWriteAllParamsLst(Head : ParamSeqTail) →
Head + ":" + **TParamType**(Head) + "," +
TWriteAllParamsLst(ParamSeqTail)

TWriteAllParamsLst(Head : < >) → Head + ":" + **TParamType**(Head)

2. Rule : **TKGrdCtrnt**

TKGrdCtrnt(Segment) →
{ "(" + **TKGetGrdPredc**(**TNodeType**(**TConstrnt**(Segment))) + ")" }

3. Rule : **TKGetGrdPredc**

TKGetGrdPredc(NodeType) →
<IF> NodeType = Simple
<THEN> **TSimpleCauseSource**(**TSegment**(Simple))
+ **TSimpleCond**(**TCond**(Simple))
<ELSE><IF> NodeType = OR_node
<THEN><LET> Nodes = **TAllInstances**(OR_node)
<IN>Nodes → <ITERATE>(n; ret : String = "(" |
<IF> n = **last**(Nodes)
<THEN> ret = ret + **TKGetGrdPredc**(n) + ")"
<ELSE> ret = ret + **TKGetGrdPredc**(n) + ")∨ ("
<ENDIF>)
<ENDIF>
<ELSE><IF> NodeType = AND_node
<THEN><LET> Nodes = **TAllInstances**(AND_node)
<IN> Nodes → <ITERATE>(n; ret : String = "(" |
<IF> n = **last**(Nodes)
<THEN> ret = ret + **TKGetGrdPredc**(n) + ")"
<ELSE> ret = ret + **TKGetGrdPredc**(n) + ") ∧ ("
<ENDIF>)
<ENDIF>
<ENDIF>

4. Rule : **TKTimeCtrnt**

TKTimeCtrnt(Segment) →

{ **TKGetTimingPredc**(Segment, **TKTiming**(**TConstrnt**(Segment))) }

5. Rule : **TKGetTimingPredc**

TKGetTimingPredc (Segment, Timing) →

<IF> !**TKEmpty**(Timing)

<THEN> "◊" + Timing

<ELSE> <SKIP>

<ENDIF>

<LET> exp = **TClass**(**TObj**(**TObjSt**(Segment)))

<IN>

<IF> **THasParam**(exp)

<THEN>

TObj(**TObjSt**(Segment))

+ "*State(* "

+ **TWriteParamLst**(**TAllParam**(exp))

+ "*)* = "

+ " ' " + **TObjSt**(Segment) + " ' "

<ELSE>

TObj(**TObjSt**(Segment))

+ "*State* = "

+ " ' " + **TObjSt**(Segment) + " ' "

<ENDIF>

E.2 Translation rules for creating a KAOS goal from `SimultaneityArrow`

6. Rule : **TKSParamGuardValue**

TKSParamGuardValue(Simul) →
<LET> exp = **TClass**(**TObj**(**TObjSt**(**TKSStartSegm**(Simul))))
<IF>**THasParam**(exp)
<THEN>
"∀" +
TWriteAllParamsLst(**TAllParam**(exp))
<ELSE><SKIP>
<ENDIF>
TWriteAllParamsLst(Head : ParamSeqTail) →
Head + ":" + **TParamType**(Head) + "," +
TWriteAllParamsLst(ParamSeqTail)

TWriteAllParamsLst(Head : < >) → Head + ":" + **TParamType**(Head)

7. Rule : **TKSCause**(Simul)

TKSCause(Simul) →
<LET> exp = **TObj**(**TObjSt**(**TKSStartSegm**(Simul)))
<IN>
<IF> **THasParam**(**TClass**(exp))
<THEN>
exp
+ "*State(* "
+ **TWriteParamLst**(**TAllParam**(**TClass**(exp)))
+ "*)* = "

+ " ' " + **TObjSt**(**TKStartSegm**(Simul)) + " ' "

<ELSE>

exp

+ "*State* = "

+ " ' " + **TObjSt**(**TKStartSegm**(Simul)) + " ' "

<ENDIF>

8. Rule : **TKSEffect**

TKSEffect(Simul) →

<LET> exp = **TObj**(**TObjSt**(**TEndSegm**(Simul)))

<IN>

<IF> **THasParam**(**TClass**(exp))

<THEN>

exp

+ "*State(* "

+ **TWriteParamLst**(**TAllParam**(**TClass**(exp)))

+ "*)* = "

+ " ' " + **TObjSt**(**TEndSegm**(Simul)) + " ' "

<ELSE>

exp

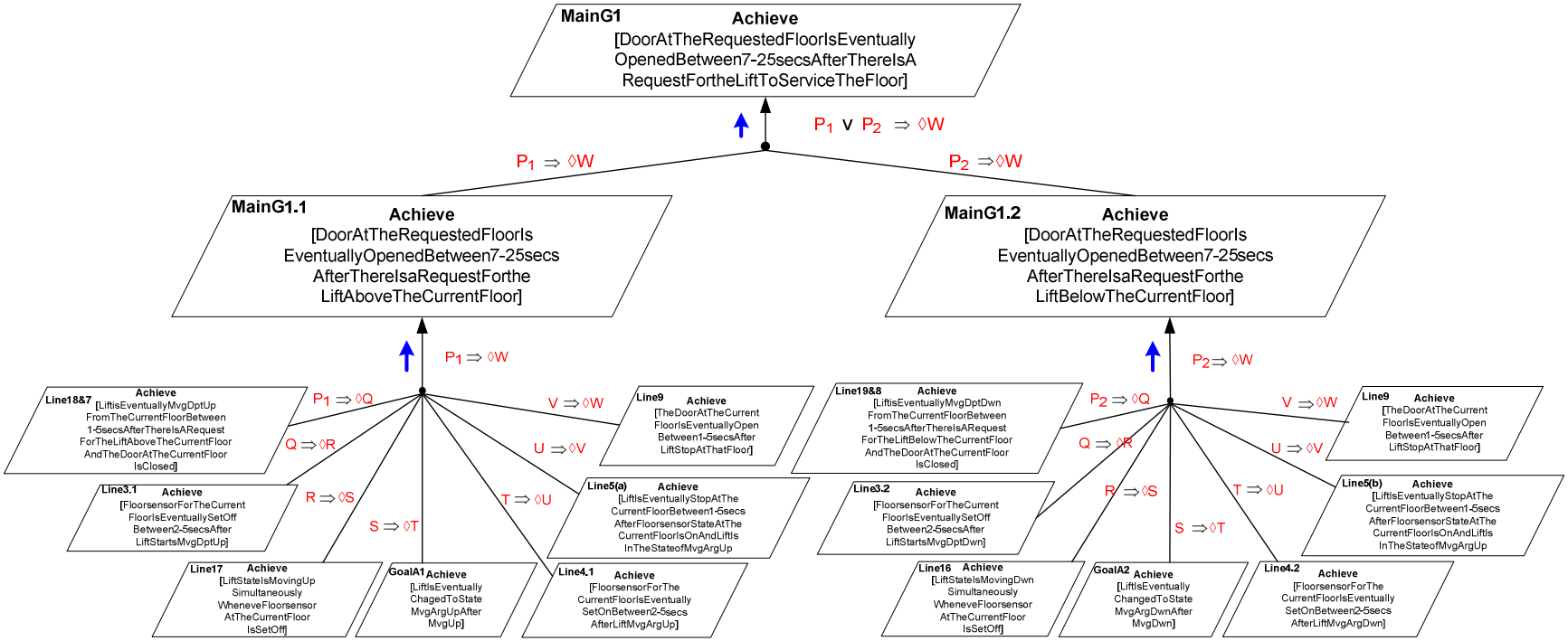
+ "*State* = "

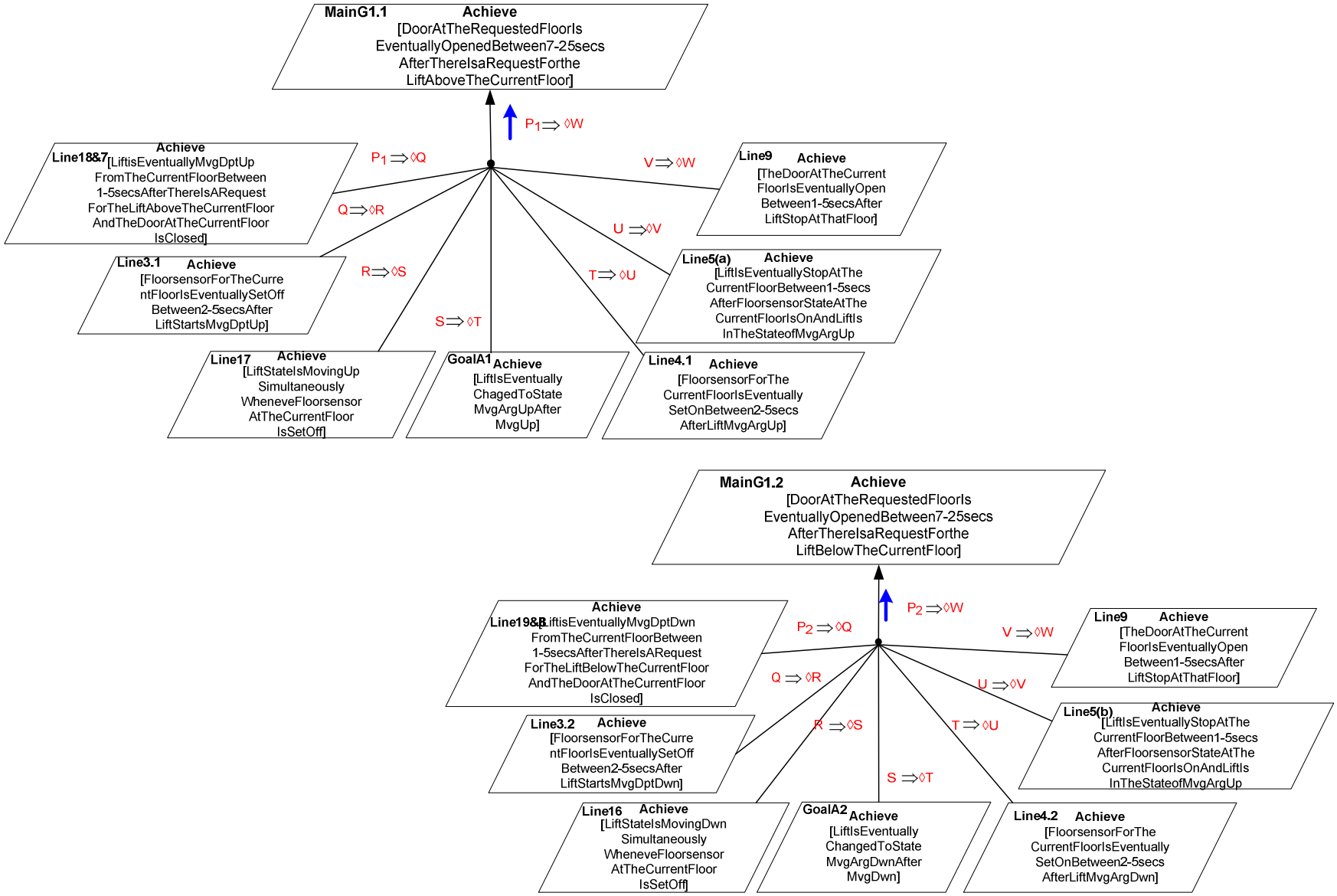
+ " ' " + **TObjSt**(**TEndSegm**(Simul)) + " ' "

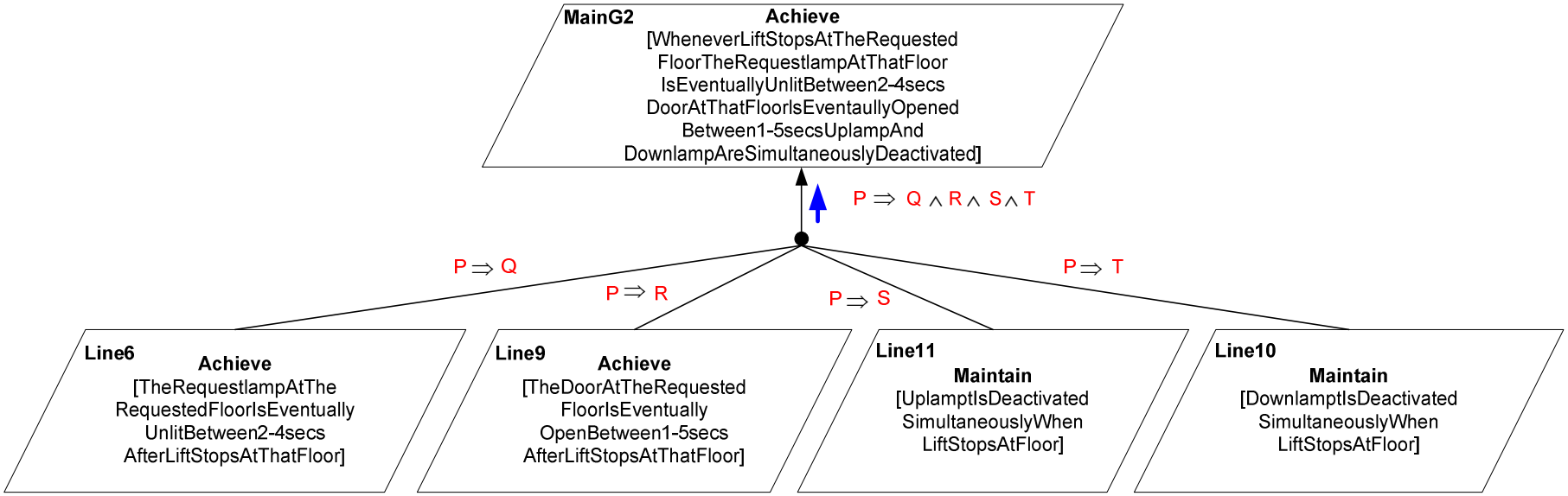
<ENDIF>

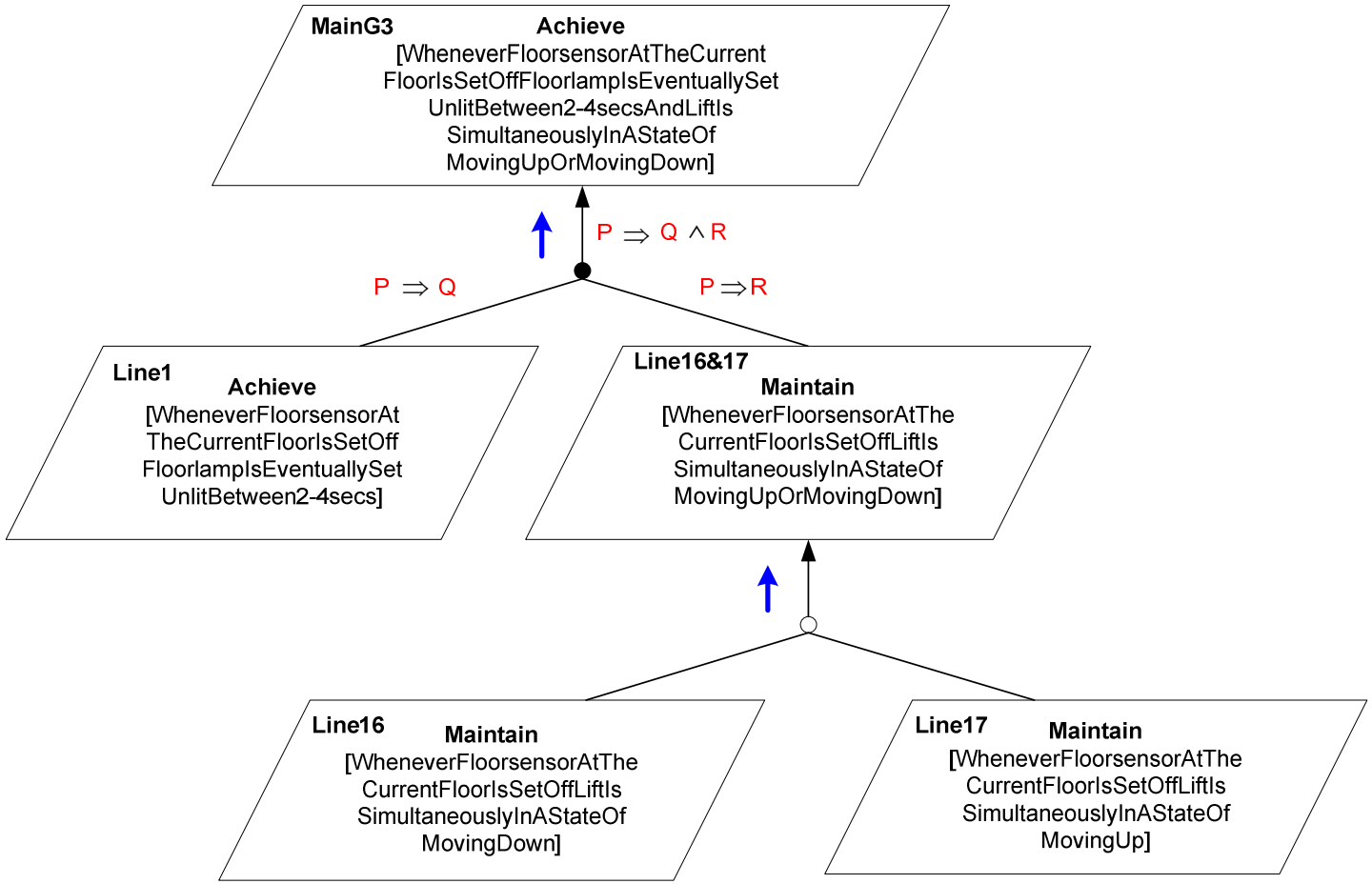
Appendix F. **KAOS Goals and Operation models**

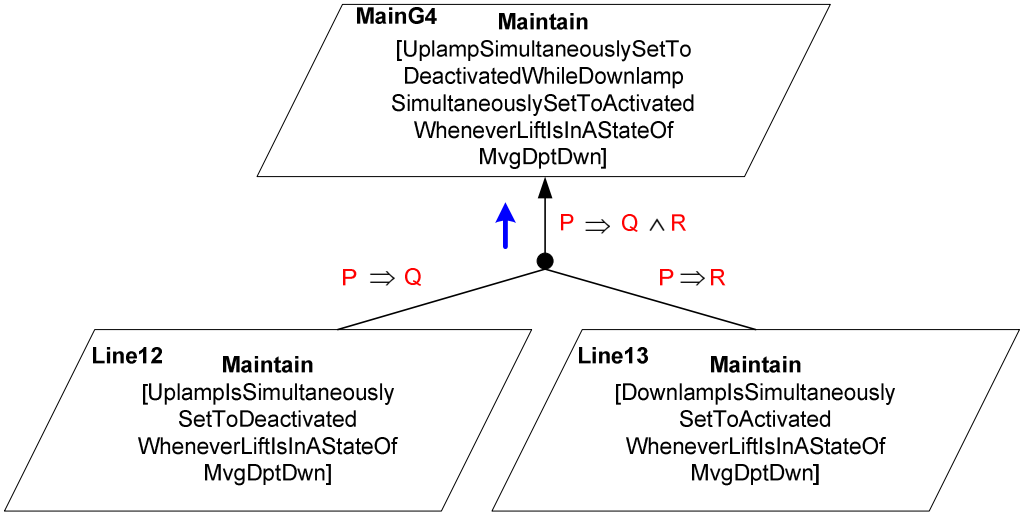
F.1 Goal Model

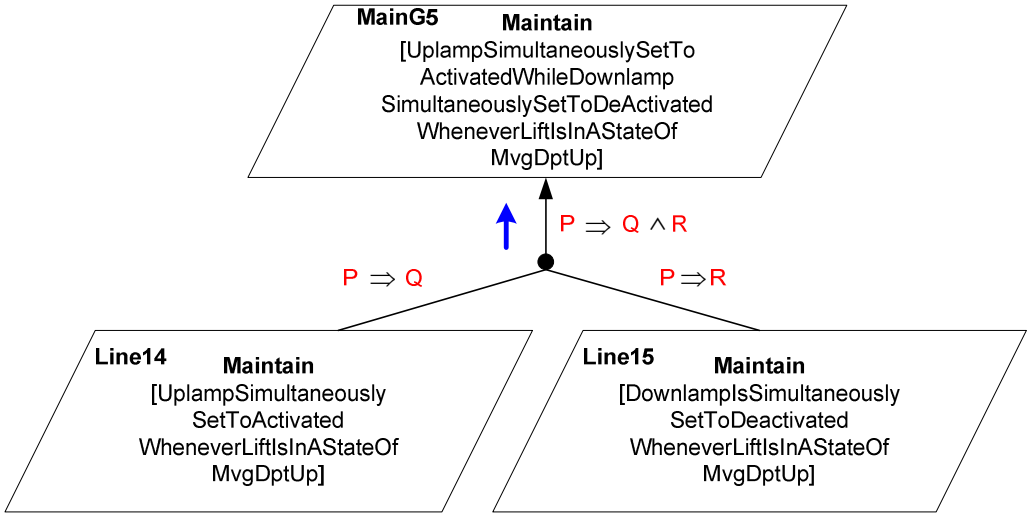


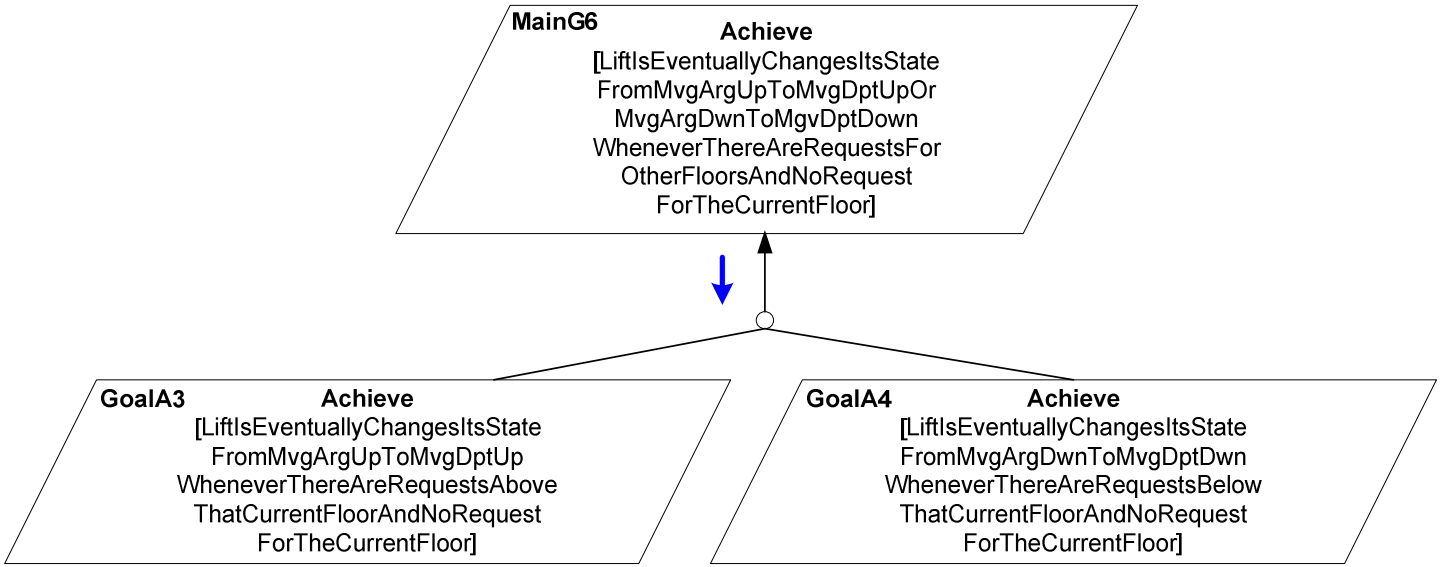

MainG1.1 Achieve
[DoorAtTheRequestedFloorIs EventuallyOpenedBetween7-25secs AfterThereIsaRequestForthe LiftAboveTheCurrentFloor]
P1⇒◊W
Line18&7 Achieve
[LiftisEventuallyMvgDptUp FromTheCurrentFloorBetween 1-5secsAfterThereIsARequest ForTheLiftAboveTheCurrentFloor AndTheDoorAtTheCurrentFloor IsClosed]
P1⇒◊Q
V⇒◊W
Line9 Achieve
[TheDoorAtTheCurrent FloorIsEventuallyOpen Between1-5secsAfter LiftStopAtThatFloor]
Q⇒◊R
U⇒◊V
Line3.1 Achieve
[FloorsensorForTheCurre ntFloorIsEventuallySetOff Between2-5secsAfter LiftStartsMvgDptUp]
R⇒◊S
T⇒◊U
Line5(a) Achieve
[LiftIsEventuallyStopAtThe CurrentFloorBetween1-5secs AfterFloorsensorStateAtThe CurrentFloorIsOnAndLiftIs InTheStateofMvgArgUp
S⇒◊T
Line17 Achieve
[LiftStateIsMovingUp Simultaneously WheneveFloorsensor AtTheCurrentFloor IsSetOff]
GoalA1 Achieve
[LiftIsEventually ChagedToState MvgArgUpAfter MvgUp]
Line4.1 Achieve
[FloorsensorForThe CurrentFloorIsEventually SetOnBetween2-5secs AfterLiftMvgArgUp]
MainG1.2 Achieve
[DoorAtTheRequestedFloorIs EventuallyOpenedBetween7-25secs AfterThereIsaRequestForthe LiftBelowTheCurrentFloor]
P2⇒◊W
Line19&8 Achieve
[LiftisEventuallyMvgDptDwn FromTheCurrentFloorBetween 1-5secsAfterThereIsARequest ForTheLiftBelowTheCurrentFloor AndTheDoorAtTheCurrentFloor IsClosed]
P2⇒◊Q
V⇒◊W
Line9 Achieve
[TheDoorAtTheCurrent FloorIsEventuallyOpen Between1-5secsAfter LiftStopAtThatFloor]
Q⇒◊R
U⇒◊V
Line3.2 Achieve
[FloorsensorForTheCurre ntFloorIsEventuallySetOff Between2-5secsAfter LiftStartsMvgDptDwn]
R⇒◊S
T⇒◊U
Line5(b) Achieve
[LiftIsEventuallyStopAtThe CurrentFloorBetween1-5secs AfterFloorsensorStateAtThe CurrentFloorIsOnAndLiftIs InTheStateofMvgArgUp
S⇒◊T
Line16 Achieve
[LiftStateIsMovingDwn Simultaneously WheneveFloorsensor AtTheCurrentFloor IsSetOff]
GoalA2 Achieve
[LiftIsEventually ChangedToState MvgArgDwnAfter MvgDwn]
Line4.2 Achieve
[FloorsensorForThe CurrentFloorIsEventually SetOnBetween2-5secs AfterLiftMvgArgDwn]

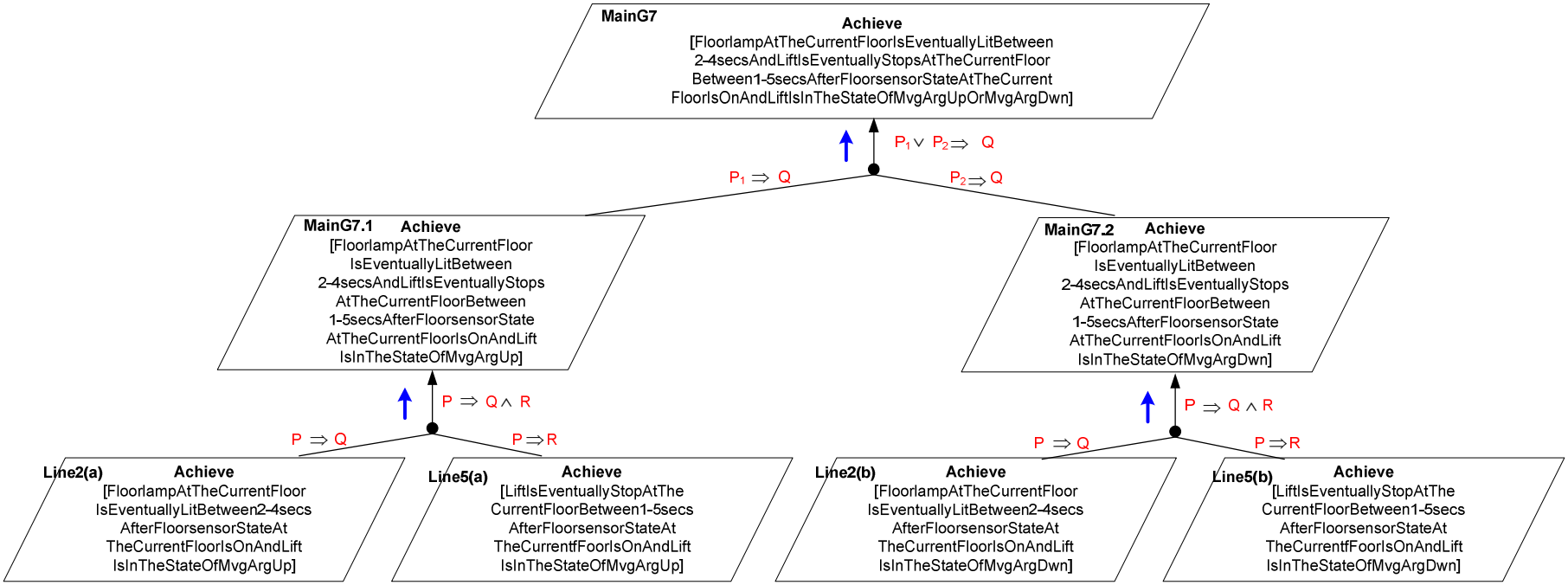

MainG2
Achieve
[WheneverLiftStopsAtTheRequested
FloorTheRequestlampAtThatFloor
IsEventuallyUnlitBetween2-4secs
DoorAtThatFloorIsEventaullyOpened
Between1-5secsUplampAnd
DownlampAreSimultaneouslyDeactivated]
P ⇒ Q ∧ R ∧ S ∧ T
P ⇒ Q
P ⇒ R
P ⇒ S
P ⇒ T
Line6
Achieve
[TheRequestlampAtThe
RequestedFloorIsEventually
UnlitBetween2-4secs
AfterLiftStopsAtThatFloor]
Line9
Achieve
[TheDoorAtTheRequested
FloorIsEventually
OpenBetween1-5secs
AfterLiftStopsAtThatFloor]
Line11
Maintain
[UplamptIsDeactivated
SimultaneouslyWhen
LiftStopsAtFloor]
Line10
Maintain
[DownlamptIsDeactivated
SimultaneouslyWhen
LiftStopsAtFloor]


MainG3
Achieve
[WheneverFloorsensorAtTheCurrent
FloorIsSetOffFloorlampIsEventuallySet
UnlitBetween2-4secsAndLiftIs
SimultaneouslyInAStateOf
MovingUpOrMovingDown]
P ⇒ Q ∧ R
P ⇒ Q
P ⇒ R
Line1
Achieve
[WheneverFloorsensorAt
TheCurrentFloorIsSetOff
FloorlampIsEventuallySet
UnlitBetween2-4secs]
Line16&17
Maintain
[WheneverFloorsensorAtThe
CurrentFloorIsSetOffLiftIs
SimultaneouslyInAStateOf
MovingUpOrMovingDown]
Line16
Maintain
[WheneverFloorsensorAtThe
CurrentFloorIsSetOffLiftIs
SimultaneouslyInAStateOf
MovingDown]
Line17
Maintain
[WheneverFloorsensorAtThe
CurrentFloorIsSetOffLiftIs
SimultaneouslyInAStateOf
MovingUp]


MainG4
Maintain
[UplampSimultaneouslySetTo
DeactivatedWhileDownlamp
SimultaneouslySetToActivated
WheneverLiftIsInAStateOf
MvgDptDwn]
P ⇒ Q ∧ R
P ⇒ Q
P ⇒ R
Line12
Maintain
[UplampIsSimultaneously
SetToDeactivated
WheneverLiftIsInAStateOf
MvgDptDwn]
Line13
Maintain
[DownlampIsSimultaneously
SetToActivated
WheneverLiftIsInAStateOf
MvgDptDwn]


MainG5
Maintain
[UplampSimultaneouslySetTo
ActivatedWhileDownlamp
SimultaneouslySetToDeActivated
WheneverLiftIsInAStateOf
MvgDptUp]
P ⇒ Q ∧ R
P ⇒ Q
P ⇒ R
Line14
Maintain
[UplampSimultaneously
SetToActivated
WheneverLiftIsInAStateOf
MvgDptUp]
Line15
Maintain
[DownlampIsSimultaneously
SetToDeactivated
WheneverLiftIsInAStateOf
MvgDptUp]


MainG6
Achieve
[LiftIsEventuallyChangesItsState
FromMvgArgUpToMvgDptUpOr
MvgArgDwnToMgvDptDown
WheneverThereAreRequestsFor
OtherFloorsAndNoRequest
ForTheCurrentFloor]
GoalA3
Achieve
[LiftIsEventuallyChangesItsState
FromMvgArgUpToMvgDptUp
WheneverThereAreRequestsAbove
ThatCurrentFloorAndNoRequest
ForTheCurrentFloor]
GoalA4
Achieve
[LiftIsEventuallyChangesItsState
FromMvgArgDwnToMvgDptDwn
WheneverThereAreRequestsBelow
ThatCurrentFloorAndNoRequest
ForTheCurrentFloor]


MainG7
Achieve
[FloorlampAtTheCurrentFloorIsEventuallyLitBetween
2-4secsAndLiftIsEventuallyStopsAtTheCurrentFloor
Between1-5secsAfterFloorsensorStateAtTheCurrent
FloorIsOnAndLiftIsInTheStateOfMvgArgUpOrMvgArgDwn]
P1 ∨ P2 ⇒ Q
P1 ⇒ Q
P2 ⇒ Q
MainG7.1 Achieve
[FloorlampAtTheCurrentFloor
IsEventuallyLitBetween
2-4secsAndLiftIsEventuallyStops
AtTheCurrentFloorBetween
1-5secsAfterFloorsensorState
AtTheCurrentFloorIsOnAndLift
IsInTheStateOfMvgArgUp]
MainG7.2 Achieve
[FloorlampAtTheCurrentFloor
IsEventuallyLitBetween
2-4secsAndLiftIsEventuallyStops
AtTheCurrentFloorBetween
1-5secsAfterFloorsensorState
AtTheCurrentFloorIsOnAndLift
IsInTheStateOfMvgArgDwn]
P ⇒ Q ∧ R
P ⇒ Q
P ⇒ R
P ⇒ Q ∧ R
P ⇒ Q
P ⇒ R
Line2(a) Achieve
[FloorlampAtTheCurrentFloor
IsEventuallyLitBetween2-4secs
AfterFloorsensorStateAt
TheCurrentFloorIsOnAndLift
IsInTheStateOfMvgArgUp]
Line5(a) Achieve
[LiftIsEventuallyStopAtThe
CurrentFloorBetween1-5secs
AfterFloorsensorStateAt
TheCurrentfFoorIsOnAndLift
IsInTheStateOfMvgArgUp]
Line2(b) Achieve
[FloorlampAtTheCurrentFloor
IsEventuallyLitBetween2-4secs
AfterFloorsensorStateAt
TheCurrentFloorIsOnAndLift
IsInTheStateOfMvgArgDwn]
Line5(b) Achieve
[LiftIsEventuallyStopAtThe
CurrentFloorBetween1-5secs
AfterFloorsensorStateAt
TheCurrentfFoorIsOnAndLift
IsInTheStateOfMvgArgDwn]

F.2 The Detail of Goal and Operation Models:

MainG1 Goal **Achieve**[DoorAtTheRequestedFloorIsEventuallyOpenedBetween7-25secs AfterThereIsARequestFortheLiftToServiceTheFloor]

Definition : Door at the requested floor is eventually opened between 7 and 25 seconds after there is a request for the lift to service that floor

FormalDef $\forall$ f : FLOOR, f : reqFl, (f > currentFl $\vee$ f < currentFl)
requestlampState(f) = 'rLit'
doorState(currentFl) = 'Closed'
liftState = 'StopAtFloor'
$\Rightarrow$
$\Diamond_{[7, 25]}$ doorState(f) = 'Open'

MainG1.1 Goal **Achieve**[DoorAtTheRequestedFloorIsEventuallyOpenedBetween7-25secs AfterThereIsaRequestFortheLiftAboveTheCurrentFloor]

Definition : Door at the requested floor is eventually opened between 7 and 25 seconds after there is a request for the lift above the current floor

FormalDef $\forall$ f : FLOOR, f : reqFl, f > currentFl
requestlampState(f) = 'rLit'
doorState(currentFl) = 'Closed'
liftState = 'StopAtFloor'
$\Rightarrow$
$\Diamond_{[7, 25]}$ doorState(f) = 'Open'

MainG1.2 Goal **Achieve**[DoorAtTheRequestedFloorIsEventuallyOpenedBetween7-25secs AfterThereIsaRequestFortheLiftBelowTheCurrentFloor]

Definition : Door at the requested floor is eventually opened between 7 and 25 seconds after there is a request for the lift below the current floor

FormalDef $\forall$ f : FLOOR, f : reqFl, f < currentFl
requestlampState(f) = 'rLit'
doorState(currentFl) = 'Closed'
liftState = 'StopAtFloor'
$\Rightarrow$
$\Diamond_{[7, 25]}$ doorState(f) = 'Open'

MainG2 Goal **Achieve**[WheneverLiftStopsAtTheRequestedFloorTheRequestlampAtThat FloorIsEventuallyUnlitBetween2-4secsDoorAtThatFloorIsEventuallyOpenedBetween1-5 secsUplampAndDownlampAreSimultaneouslyDeactivated]

Definition : Whenever lift stops at the requested floor, the request lamp at that floor is eventually unit between 2 and 4 seconds, door at that floor is eventually opened between 1 and 5 seconds, and up lamp and down lamp are simultaneously deactivated

FormalDef

$\forall$ f : FLOOR, f : reqFl, f = currentFl
liftState = 'StopAtFloor'
$\Rightarrow$
$\Diamond_{[2, 4]}$ reqestlampState(f) = 'rUnlit'
$\Diamond_{[1, 5]}$ doorState(f) = 'Open'
uplampState = 'deActivated'
downlampState = 'Deactivated'

MainG3 Goal **Achieve**[WheneverfloorsensorAtTheCurrentFloorIsSetOffFloorLampIsEven tuallySetUnlitBetween2-4secsAndLiftIsSimultaneouslyInAStateOfMovingUpOrMovingDow n]

Definition : Whenever floor sensor at the current floor is set off, floor lamp is eventually set unlit between 2 and 4 seconds and lift is simultaneously in a state of moving up or moving down

FormalDef

$\forall$ f : FLOOR, f : reqFl, f = currentFl
floorsensorState(f) = 'Off'
$\Rightarrow$
$\Diamond_{[2, 4]}$ floorlampState(f) = 'Unlit'
(liftState = 'MovingUp' $\vee$ liftState = 'MovingDown'

MainG4 Goal **Maintain**[UplampSimultaneouslySetToDeactivatedWhileDownlampSimultan eouslySetToActivatedWheneverLiftIsInAStateOfMvgDptDown]

Definition : Up lamp is simultaneously set to deactivated while down lamp is simultaneously set to activated whenever lift is in a state of moving departing down

FormalDef

liftState = 'MovingDepartingDown'
$\Rightarrow$
uplampState = 'deActivated'
downlampState = 'Activated'

MainG5 Goal **Maintain**[UplampSimultaneouslySetToActivatedWhileDownlampSimultaneouslySetToDeativatedWheneverLiftIsInAStateOfMvgDptup]

Definition : Up lamp is simultaneously set to activated while down lamp is simultaneously set to deactivated whenever lift is in a state of moving departingup

FormalDef liftState = 'MovingDepartingup'
$\Rightarrow$
uplampState = 'acTivated'
downlampState = 'Deactivated'

MainG6 Goal **Achieve**[LiftIsEventuallyChangesItsStateFromMvgArgUpToMvgDptUpOrMvgArgDownToMgvDptdownWheneverThereAreRequestsForOtherFloorsAndNoRequestForTheCurrentFloor]

Definition : Lift is eventually changes its state from moving arriving up to moving departing up or moving arriving down to moving departing down whenever there are requests for other floors and no request for the current floor

FormalDef $\forall$ f : FLOOR, f : reqFl, (f > currentFl $\vee$ f < currentFl)
currentFl $\notin$ reqFl
liftState = 'MovingArrivingUp' $\vee$
liftState = 'MovingArrivingDown'
$\Rightarrow$
$\Diamond$ liftState = 'MovingDepartingUp' $\vee$
liftState = 'MovingDepartingDown'

MainG7 Goal **Achieve**[FloorlampAtTheCurrentFloorIsEventuallyLitBetween2-4secsAndLiftIsEventuallyStopsAtTheCurrentFloorBetween1-5secsAfterFloorsensorStateAtTheCurrentFloorIsOnAndLiftIsInTheStateOfMvgArgUpOrMvgArgDwn]

Definition : Floor lamp at the current floor is eventually lit between 2 and 4 seconds and lift is eventually stops at the current floor between 1 and 5 seconds after floor sensor state at the current floor is on and lift is in the state of moving arriving up or moving arriving down

FormalDef $\forall$ f : FLOOR, f : reqFl, f = currentFl
floorsensorState(f) = 'On' &
(liftState = 'MovingArrivingUp' $\vee$
liftState = 'MovingArrivingDown')
$\Rightarrow$
$\Diamond_{[2, 4]}$ floorlampState(f) = 'Lit'

$\Diamond_{[1, 5]}$ liftState(f) = 'StopAtFloor'

MainG7.1 Goal **Achieve**[FloorlampAtTheCurrentFloorIsEventuallyLitBetween2-4secsAndLiftIsEventuallyStopsAtTheCurrentFloorBetween1-5secsAfterFloorsensorStateAtTheCurrentFloorIsOnAndLiftIsInTheStateOfMvgArgUp]

Definition : Floor lamp at the current floor is eventually lit between 2 and 4 seconds and lift is eventually stops at the current floor between 1 and 5 seconds after floor sensor state at the current floor is on and lift is in the state of moving arriving up

FormalDef $\forall$ f : FLOOR, f : reqFl, f = currentFl
floorsensorState(f) = 'On' &
liftState = 'MovingArrivingUp')
$\Rightarrow$
$\Diamond_{[2, 4]}$ floorlampState(f) = 'Lit'
$\Diamond_{[1, 5]}$ liftState(f) = 'StopAtFloor'

MainG7.2 Goal **Achieve**[FloorlampAtTheCurrentFloorIsEventuallyLitBetween2-4secsAndLiftEventuallyStopsAtTheCurrentFloorBetween1-5secsAfterFloorsensorStateAtTheCurrentFloorIsOnAndLiftIsInTheStateOfMvgArgDwn]

Definition : Floor lamp at the current floor is eventually lit between 2 and 4 seconds and lift is eventually stops at the current floor between 1 and 5 seconds after floor sensor state at the current floor is on and lift is in the state of moving arriving down

FormalDef $\forall$ f : FLOOR, f : reqFl, f = currentFl
floorsensorState(f) = 'On' &
liftState = 'MovingArrivingDown')
$\Rightarrow$
$\Diamond_{[2, 4]}$ floorlampState(f) = 'Lit'
$\Diamond_{[1, 5]}$ liftState(f) = 'StopAtFloor'

Line1 Goal **Achieve**[WheneverFloorsensorAtTheCurrentFloorIsSetOffFloorlampIsEventuallySetUnlitBetween2-4secs]

Definition : Floor lamp at the current floor is eventually set to unlit between 2 and 4 seconds after floor sensor at the current floor is set off

FormalDef $\forall$ f : FLOOR, f = currentFl
floorsensorState(f) = 'Off'

$\Rightarrow$

$\Diamond_{[2, 4]}$ floorlampState(f) = 'Unlit'

Operation FloorlampUnlit
Input floorlamp{arg f : FlOOR, f = currentFl}state
Output floorlamp{arg f : FlOOR, f = currentFl}state
DomPre floorlampState(f) = 'Lit'
DomPost floorlampState(f) = 'Unlit'
ReqTrig floorlampState(f) = 'Lit' $\boldsymbol{S}_{[1, 3]}$ (floorsensorState(f) = 'Off')

Line2(a) Goal **Achieve**[FloorlampAtTheCurrentFloorIsEventuallyLitBetween2-4secsAfter FloorsensorStateAtTheCurrentFloorIsOnAndLiftIsInTheStateOfMvgArgUp]

Definition : Floor lamp at the current floor is eventually lit between 2 and 4 seconds after floor sensor state at the current floor is on and lift is in the state of moving arriving up

FormalDef: $\forall$ f : FLOOR, f: reqFl, f = currentFl
floorsensorState(f) = 'On'
liftState = 'MovingArrivingUp'
$\Rightarrow$
$\Diamond_{[2, 4]}$ floorlampState(f) = 'Lit'

Operation FloorlampLit
Input floorlamp{arg f : FlOOR, f: reqFl, f = currentFl }State
Output floorlamp{arg f : FlOOR, f: reqFl, f = currentFl }State
DomPre floorlampState(f) = 'Unlit'
DomPost floorlampState(f) = 'Lit'
ReqTrig floorlampState(f) = 'Unlit' $\boldsymbol{S}_{[1, 3]}$ (floorsensorState(f) = 'On' & liftState = 'MovingArringUp')

Line2(b) Goal **Achieve**[FloorlampAtTheCurrentFloorIsEventuallyLitBetween2-4secsAfter FloorSensorStateAtTheCurrentFloorIsOnAndLiftIsInTheStateOfMvgArgDwn]

Definition : Floor lamp at the current floor is eventually lit between 2 and 4 seconds after floor sensor state at the current floor is on and lift is in the state of moving arriving down

FormalDef: $\forall$ f : FLOOR, f: reqFl, f = currentFl
floorsensorState(f) = 'On'
liftState = 'MovingArrivingDown'

$\Rightarrow$

$\Diamond_{[2, 4]}$ floorlampState(f) = 'Lit'

Operation FloorlampLit
Input floorlamp{arg f : FlOOR, f = currentFl}state
Output floorlamp{arg f : FlOOR, f: reqFl, f = currentFl }State
DomPre floorlampState(f) = 'Unlit'
DomPost floorlampState(f) = 'Lit'
ReqTrig floorlampState(f) = 'Unlit'
$\boldsymbol{S}_{[1, 3]}$ (floorsensorState(f) = 'On' & liftState = 'MovingArringDown')

Line3.1 Goal **Achieve**[FloorsensorForTheCurrentFloorIsEventuallySetOffBetween2-5secs AfterLiftStartsMvgDptUp]

Definition: The floor sensor at the current floor is eventually set off between 2 and 5 seconds after lift is in the state of moving departing up providing the direction of lift is up

FormalDef $\forall$ f : FLOOR, f = currentFl
liftState = 'MovingDepartingUp' & dir = Up
$\Rightarrow$
$\Diamond_{[2,5]}$ floorsensorState(f) = 'Off'

Operation FloorsensorOff
Input floorsensor{arg f : FlOOR, f = currentFl}State
Output floorsensor{arg f : FlOOR, f = currentFl}State
DomPre floorsensorState(f) = 'On'
DomPost floorsensorState(f) = 'Off'
ReqTrig floorsensorState(f) = 'On' $\boldsymbol{S}_{[1,4]}$ (liftState = 'MovingDepartingUp' & dir = Up)

Line3.2 Goal **Achieve**[FloorsensorForTheCurrentFloorIsEventuallySetOffBetween2-5secs AfterLiftStartsMvgDptDwn]

Definition: The floor sensor at the current floor is eventually set off between 2 and 5 seconds after lift is in the state of moving departing down providing the direction of lift is down

FormalDef: $\forall$ f : FLOOR, f = currentFl
(liftState = 'MovingDepartingDown' & dir = Down)

$\Rightarrow$

$\Diamond_{[2,5]}$ floorsensorState(f) = 'Off'

Operation FloorsensorOff
Input floorsensor{arg f : FlOOR, f = currentFl}State
Output floorsensor{arg f : FlOOR, f = currentFl}State
DomPre floorsensorState(f) = 'On'
DomPost floorsensorState(f) = 'Off'
ReqTrig floorsensorState(f) = 'On'
$\boldsymbol{S}_{[1,4]}$ (liftState = 'MovingDepartingDown' & dir = Down)

Line4.1 Goal **Achieve**[FloorsensorForTheCurrentFloorIsEventuallySetOnBetween2-5secs AfterLiftMvgArgUp]

Definition: Floor sensor for the current floor is eventually set on between 2 and 5 seconds after lift is moving arriving up

FormalDef $\forall$ f : FLOOR, f = currentFl
liftState = 'MovingArrivingUp'
$\Rightarrow$
$\Diamond_{[2,5]}$ floorsensorState(f) = 'On'

Operation FloorsensorOn
Input floorsensor{arg f : FlOOR, f = currentFl}State
Output floorsensor{arg f : FlOOR, f = currentFl}State
DomPre floorsensorState(f) = 'Off'
DomPost floorsensorState(f) = 'On'
ReqTrig floorsensorState(f) = 'Off' $\boldsymbol{S}_{[1,4]}$ (liftState = 'MovingArrivingUp')

Line4.2 Goal **Achieve**[FloorsensorForTheCurrentFloorIsEventuallySetOnBetween2-5secs AfterLiftMvgArgDwn]

Definition: Floor sensor for the current floor is eventually set on between 2 and 5 seconds after lift is moving arriving down

FormalDef $\forall$ f : FLOOR, f = currentFl
liftState = 'MovingArrivingDown'
$\Rightarrow$
$\Diamond_{[2,5]}$ floorsensorState(f) = 'On'

Operation FloorsensorOn
Input floorsensor{arg f : FlOOR, f = currentFl}State
Output floorsensor{arg f : FlOOR, f = currentFl}State
DomPre floorsensorState(f) = 'Off'
DomPost floorsensorState(f) = 'On'
ReqTrig floorsensorState(f) = 'Off' $\boldsymbol{S}_{[1, 4]}$ (liftState = 'MovingArrivingDown')

Line5(a) Goal **Achieve**[LiftIsEventuallyStopAtTheCurrentFloorBetween1-5secsAfterFloor sensorStateAtTheCurrentFloorIsOnAndLiftIsInTheStateOfMvgArgUp]
Definition : A lift is eventually stopped at the current floor between 1 and 5 seconds after a floor sensor at that floor is set on and lift is in the state of moving arriving up.
FormalDef $\forall$ f : FLOOR, f : reqFl, f = currentFl
floorsensorState(f) = 'On' &
liftState = 'MovingArrivingUp'
$\Rightarrow$
$\Diamond_{[1,5]}$ liftState = 'StopAtFloor'

Operation LiftStopAtFloor
Input liftState
Output liftState
DomPre liftState $\neq$ 'StopAtFloor'
DomPost liftState = 'StopAtFloor'
ReqTrig liftState $\neq$ 'StopAtFloor'
$\boldsymbol{S}_{[0.99, 4]}$ (floorsensorState(f) = 'On' & liftState = 'MovingArrivingUp')

Line5(b) Goal **Achieve**[LiftIsEventuallyStopAtTheCurrentFloorBetween1-5secsAfterFloor sensonrsensorStateAtTheCurrentFloorIsOnAndLiftIsInTheStateOfMvgArgDwn]
Definition : A lift is eventually stopped at the current floor between 1 and 5 seconds after a floor sensor at that floor is set on and lift is in the state of moving arriving down.
FormalDef $\forall$ f : FLOOR, f : reqFl, f = currentFl
floorsensorState(f) = 'On' &
liftState = 'MovingArrivingDown'
$\Rightarrow$
$\Diamond_{[1,5]}$ liftState = 'StopAtFloor'

Operation LiftStopAtFloor
Input liftState
Output liftState
DomPre liftState ≠ 'StopAtFloor'
DomPost liftState = 'StopAtFloor'
ReqTrig liftState ≠ 'StopAtFloor'
$\boldsymbol{S}_{[0.99, 4]}$ (floorsensorState(f) = 'On' & liftState = 'MovingArrivingDown')

Line6 Goal **Achieve**[TheRequestlampAtTheRequestedFloorIsEventuallyUnlitBetween2-4 secs AfterLiftStopsAtThatFloor]
Definition: The request lamp at the current floor is eventually set to unlit between 2 and 4 seconds after lift is in the state of stop at floor
FormalDef $\forall$ f : FLOOR, f = currentFl
liftState = 'StopAtFloor'
$\Rightarrow$
$\Diamond_{[2, 4]}$ requestlampState(f) = 'rUnlit'

Operation RequestlamprUnlit
Input requestlamp{arg f : FLOOR, f = currentFl}State
Output requestlamp{arg f : FLOOR, f = currentFl}State
DomPre requestlampState(f) = 'rLit'
DomPost requestlampState(f) = 'rUnlit'
ReqTrig requestlampState(f) = 'rLit' $\boldsymbol{S}_{[1, 3]}$ (liftState = 'StopAtFloor')

Line9 Goal **Achieve**[TheDoorAtTheCurrentFloorIsEventuallyOpenBetween1-5secsAfterLift StopsAtThatFloor]
Definition : Door at the current floor is eventfully opened between 1 and 5 seconds after the lift stops at the current floor
FormalDef $\forall$ f : FLOOR, f : reqFl, f = currentFl
liftState = 'StopAtFloor'
$\Rightarrow$
$\Diamond_{[1,5]}$ doorState(f) = 'Open'

Operation DoorOpen
Input door{arg f : FLOOR, f : reqFl, f = currentFl}State

Output door{arg f : FLOOR, f : reqFl, f = currentFl}State
DomPre doorState(f) = 'Closed'
DomPost doorState(f) = 'Open'
ReqTrig doorState(f) = 'Closed' $\boldsymbol{S}_{[0.99, 4]}$ (liftState = 'StopAtFloor')

Line10 Goal **Maintain**[DownlampIsDeactivatedSimultaneouslyWhenLiftStopsAtFloor]
Definition : Down lamp is set to deactivate at once after lift stops at floor
FormalDef liftState = 'StopAtFloor'
$\Rightarrow$
downlampState = 'Deactivated'

Operation DownlampDeactivated
Input liftState
Output liftState
DomPre liftState $\neq$ 'StopAtFloor'
DomPost liftState = 'StopAtFloor'
ReqPost downlampState = 'Deactivated'

Operation DownlampActivated
Input downlampState
Output downlampState
DomPre downlampState = 'Deactivated'
DomPost downlampState = 'Activated'
ReqPost liftState $\neq$ 'StopAtFloor'

Line11 Goal **Maintain**[UplampIsDeactivatedSimultaneouslyWhenLiftStopsAtFloor]
Definition: Up lamp is set to deactivate at once whenever the lift stops at floor
FormalDef: liftState = 'StopAtFloor'
$\Rightarrow$
uplampState = 'deActivated'

Operation UplampdeActivated
Input liftState
Output liftState
DomPre liftState $\neq$ 'StopAtFloor'

DomPost liftState = 'StopAtFloor'
ReqPost uplampState = 'deActivated'

Operation UplampacTivated
Input uplampState
Output uplampState
DomPre uplampState = 'deActivated'
DomPost uplampState = 'acTivated'
ReqPost liftState ≠ 'StopAtFloor'

Line12 Goal **Maintain**[UplampSimultaneouslySetToDeactivatedWheneverLiftIsInAStateOf MvgDptDwn]

Definition: Up lamp is set to deactivate at once whenever the lift starts moving departing down

FormalDef liftState = 'MovingDepartingDown'
$\Rightarrow$
uplampState = 'deActivated'

Operation Uplampdeactivated
Input liftState
Output liftState
DomPre liftState ≠ 'MovingDepartingDown'
DomPost liftState = 'MovingDepartingDown'
ReqPost uplampState = 'deActivated'

Operation Uplampactivated
Input uplampState
Output uplampState
DomPre uplampState = 'deActivated'
DomPost uplampState = 'acTivated'
ReqPost liftState ≠ 'MovingDepartingDown'

Line13 Goal M**aintain**[DownlampSimultaneouslySetToActivatedWheneverLiftIsInAStateOf MvgDptDwn]

Definition : Down lamp is set to activate at once whenever the lift starts moving departing down

FormalDef liftState = 'MovingDepartingDown'

$\Rightarrow$

downlampState = 'Activated'

Operation DownlampActivated
Input liftState
Output liftState
DomPre liftState ≠ 'MovingDepartingDown'
DomPost liftState = 'MovingDepartingDown'
ReqPost downlampState = 'Activated'

Operation DownlampDeactivated
Input downlampState
Output downlampState
DomPre downlampState = 'Activated'
DomPost downlampState = 'Deactivated'
ReqPost liftState ≠ 'MovingDepartingDown'

Line14 Goal **Maintain**[UplampSimultaneouslySetToActivatedWheneverLiftIsInAStateOf MvgDptUp]

Definition : Uplamp is set to activate at once whenever the lift starts moving departing up

FormalDef: liftState = 'MovingDepartingUp'

$\Rightarrow$

uplampState = 'Activated'

Operation UplampActivated
Input liftState
Output liftState
DomPre liftState ≠ 'MovingDepartingUp'
DomPost liftState = 'MovingDepartingUp'
ReqPost uplampState = 'Activated'

Operation UplampDeactivated

Input uplampState
Output uplampState
DomPre uplampState = 'Activated'
DomPost uplampState = 'Deactivated'
ReqPost liftState ≠ 'MovingDepartingUp'

Line15 Goal **Maintain**[DownlampSimultaneouslySetToDeactivatedWheneverLiftIsInAStateOfMvgDptUp]
Definition : Downlamp is set to deactivate at once whenever the lift starts moving deapring up
FormalDef: liftState = 'MovingDepartingUp'
⇒
downlampState = 'Deactivated'

Operation DownlampActivated
Input liftState
Output liftState
DomPre liftState ≠ 'MovingDepartingUp'
DomPost liftState = 'MovingDepartingUp'
ReqPost downlampState = 'Deactivated'

Operation DownlampDeactivated
Input downlampState
Output downlampState
DomPre downlampState = 'Deactivated'
DomPost downlampState = 'Activated'
ReqPost liftState ≠ 'MovingDepartingUp'

Line16&17 Goal **Maintain**[WheneverFloorsensorAtTheCurrentFloorsIsSetOffLiftIsSimutaneouslyInAStateOfMovingUpOrMovingDown]
Definition : Whenever floorsensor at the current floor is set off, lift is simultaneously in a state of moving up or moving down
FormalDef: ∀ f : FLOOR, f = currentFl
floorsensorState(f) = 'Off'
⇒

liftState = 'MovingUp' ∨ liftState = 'MovingDown'

Line16 Goal **Maintain**[WheneverFloorsensorAtTheCurrentFloorsIsSetOffLiftIsSimutaneouslyInAStateOfMovingDown]

Definition : Whenever floorsensor at the current floor is set off, lift is simultaneously in a state of moving down

FormalDef: ∀ f : FLOOR, f = currentFl
floorsensorState(f) = 'Off'
⇒
liftState = 'MovingDown'

Operation FloorsensorOff
Input floorsensor{arg f : FLOOR, f = currentFl}State
Output floorsensor{arg f : FLOOR, f = currentFl}State
DomPre floorsensorState(f) = 'On'
DomPost floorsensorState(f) = 'Off'
ReqPost liftState = 'MovingDown'

Operation FloorsensorOn
Input liftState
Output liftState
DomPre liftState = 'MovingDown'
DomPost ¬ (liftState = 'MovingDown')
ReqPost floorsensorState(f) = 'On'

Line17 Goal **Maintain**[WheneverFloorsensorAtTheCurrentFloorsIsSetOffLiftIsSimutaneouslyInAStateOfMovingUp]

Definition : Whenever floorsensor at the current floor is set off, lift is simultaneously in a state of moving up

FormalDef: ∀ f : FLOOR, f = currentFl
floorsensorState(f) = 'Off'
⇒
liftState = 'MovingUp'

Operation FloorsensorOff

Input floorsensor{arg f : FLOOR, f = currentFl}State
Output floorsensor{arg f : FLOOR, f = currentFl}State
DomPre floorsensorState(f) = 'On'
DomPost floorsensorState(f) = 'Off'
ReqPost liftState = 'MovingUp'

Operation FloorsensorOn
Input liftState
Output liftState
DomPre liftState = 'MovingUp'
DomPost ¬ (liftState = 'MovingUp')
ReqPost floorsensorState(f) = 'On'

Line18&7 Goal **Achieve**[LiftisEventuallyMvgDptUpFromTheCurrentFloorBetween1-5secs AfterThereIsARequestForTheLiftAboveTheCurrentFloorAndTheDoorAtTheCurrentFloorIs Closed]

Definition: Lift is eventually moving departing up from the current floor between 1 and 5 seconds after there is a request for the lift above the current floor and the door at the current floor is closed.

FormalDef $\forall$ f : FLOOR, f : reqFl, f > currentFl
requestlampState(f) = 'rLit' &
doorState(currentFl) = 'Closed' &
liftState = 'StopAtFloor'
$\Rightarrow$
$\Diamond_{[1, 5]}$ liftState = 'MovingDepartingUp'

Operation LiftMovingDepartingUp
Input liftState
Output liftState
DomPre liftState = 'StopAtFloor'
DomPost liftState = 'MovingDepartingUp'
ReqTrig liftState = 'StopAtFloor'
$\boldsymbol{S}_{[0.99, 4]}$ (requestlampState(f) = 'rLit' & doorState(currentFl) = 'Closed' & liftState = 'StopAtFloor')

Line19&8 Goal **Achieve**[LiftisEventuallyMvgDptDwnFromTheCurrentFloorBetween1-5secs AfterThereIsARequestForTheLiftBelowTheCurrentFloorAndTheDoorAtTheCurrentFloorIsClosed]

Definition: Lift is eventually moving departing down from the current floor between 1 and 5 seconds after there is a request for the lift below the current floor and the door at the current floor is closed.

FormalDef $\forall$ f : FLOOR, f : reqFl, f < currentFl
requestlampState(f) = 'rLit' &
doorState(currentFl) = 'Closed' &
liftState = 'StopAtFloor'
$\Rightarrow$
$\Diamond_{[1, 5]}$ liftState = 'MovingDepartingDown'

Operation RequestlamprUnlit
Input liftState
Output liftState
DomPre liftState = 'StopAtFloor'
DomPost liftState = 'MovingDepartingDown'
ReqTrig liftState = 'StopAtFloor'
$\boldsymbol{S}_{[0.99, 4]}$ (requestlampState(f) = 'rLit' & doorState(currentFl) = 'Closed' & liftState = 'StopAtFloor')

Line2(a) Goal **Achieve**[FloorlampAtTheCurrentFloorIsEventuallyLitBetween2-4secsAfter FloorsensorStateAtTheCurrentFloorIsOnAndLiftIsInTheStateOfMvgArgUp]

Definition: Floor lamp at the current floor is eventually lit between 2 and 4 seconds after floor sensor state at the current floor is set on and lift is in the state of moving arriving up

FormalDef $\forall$ f : FLOOR, f : reqFl, f = currentFl
floorsensorState(f) = 'On' &
liftState = 'MovingArrivingUp'
$\Rightarrow$
$\Diamond_{[2, 4]}$ floorlampState(f) = 'Lit'

Operation floorlampLit
Input floorlamp{arg f : FLOOR, f : reqFl, f = currentFl}State
Output floorlamp{arg f : FLOOR, f : reqFl, f = currentFl}State
DomPre floorlampState(f) = 'Unlit'

DomPost floorlampState(f) = 'Lit'
ReqTrig floorlampState(f) = 'Unlit'
$S_{[1, 3]}$ (floorsensorState(f) = 'On' & liftState = 'MovingArrivingUp')

Line2(b) Goal **Achieve**[FloorlampAtTheCurrentFloorIsEventuallyLitBetween2-4secsAfter Fl oorsensorStateAtTheCurrentFloorIsOnAndLiftIsInTheStateOfMvgArgDwn]
Definition: Floor lamp at the current floor is eventually lit between 2 and 4 seconds after floor sensor state at the current floor is set on and lift is in the state of moving arriving down
FormalDef $\forall$ f : FLOOR, f : reqFl, f = currentFl
floorsensorState(f) = 'On' &
liftState = 'MovingArrivingDown'
$\Rightarrow$
$\Diamond_{[2, 4]}$ floorlampState(f) = 'Lit'

Operation floorlampLit
Input floorlamp{arg f : FLOOR, f : reqFl, f = currentFl}State
Output floorlamp{arg f : FLOOR, f : reqFl, f = currentFl}State
DomPre floorlampState(f) = 'Unlit'
DomPost floorlampState(f) = 'Lit'
ReqTrig floorlampState(f) = 'Unlit'
$S_{[1, 3]}$ (floorsensorState(f) = 'On' & liftState = 'MovingArrivingDown')

Goal A1 **Achieve**[LiftIsEventuallyChangedToStateMvgUpAfterMvgUp]
Definition: Lift is eventually changed to state moving arriving up after moving up
FormalDef liftState = 'MovingUp'
$\Rightarrow$
$\Diamond$ liftState = 'MovingArrivingUp'

Goal A2 **Achieve**[LiftIsEventuallyChangedToStateMvgDwnAfterMvgDwn]
Definition: Lift is eventually changed to state moving arriving down after moving down
FormalDef liftState = 'MovingDown'
$\Rightarrow$
$\Diamond$ liftState = 'MovingArrivingDown'

GoalA3 **Achieve**[LiftIsIsEventuallyChangedItsStateFromMvgArgUpToMvgDeptUpWhenever ThereAreRequestsAboveThatCurrentFloorAndNoRequestForTheCurrentFloor]

Definition: Lift is eventually changed its state form moving arriving up to moving departing up whenever there are requests above the current floor and no request for the current floor

FormalDef $\forall$ f : FLOOR, f : reqFl, f > currentFl
currentfl $\notin$ reqFl &
liftState = 'MovingArrivingUp'
$\Rightarrow$
$\Diamond$ liftState = 'MovingDepartingUp'

GoalA4 **Achieve**[LiftIsIsEventuallyChangedItsStateFromMvgArgDownToMvgDeptDown WheneverThereAreRequestsBelowThatCurrentFloorAndNoRequestForTheCurrentFloor]

Definition: Lift is eventually changed its state form moving arriving down to moving departing down whenever there are requests below the current floor and no request for the current floor

FormalDef $\forall$ f : FLOOR, f : reqFl, f < currentFl
currentfl $\notin$ reqFl &
liftState = 'MovingArrivingDown'
$\Rightarrow$
$\Diamond$ liftState = 'MovingDepartingDown'