Developing an easy-to-use Query language for verification of lighting systems

Lennart Buit

November 24, 2017

Current day lighting systems are large scale heterogeneous distributed systems. These systems are not only large, but also subject to increased demands from its stakeholders. These systems are hard to test and errors are often found late in its development. TNO-ESI together with Philips Lighting has created a domain model of lighting systems to aid in system specification. We provide a (domain generic) framework for creation of easy to use property specification languages as well as a (domain specific) prototype ("the Query language") for specification of behavioral lighting system properties. We have based our work on common patterns for property specifications and use a pattern library to create an easy to use and extensible language. We provide an extension of the translation of Gruhn and Laue (2006) in what we call monitors and rules that translate to both UPPAAL and a custom Co-simulation Framework. This extended translation may be reused across domains. Finally, we validate that our language is easy to use, show its value to Philips Lighting and show its extensibility.

Contents

Part I

Introduction & Context

1 Introduction

1.1 Lighting Systems

Traditionally, a *lighting system* consisted of luminaires[1]and circuit breaking switches. When such a switch was in the off position, the circuit was broken and the luminaires were consequently off. Throughout the eighties and the nineties, the requirements of lighting systems shifted. Lighting systems had to become *smart*: automatic dimming to conserve energy, new interactions between luminaires to increase comfort, etc.

These contemporary requirements fall in different categories. For example, user-centric requirements capture what users expect of a current day lighting system. For users, lighting systems have to:

Increase Comfort Lighting systems can feel more intuitive by anticipating how such a system is used. Rooms can be outfitted with a presentation mode, dimmers can be added to give users more control of the light condition etc.

Increase Safety Lighting systems can be helpful in case of emergency by providing light during evacuation. Also, in street lighting, correct lighting behavior may reduce the chance of accidents.

Not only do end-users have new demands of lighting systems, many other stakeholders introduce new requirements as well. These requirements are often system wide, for example a contemporary lighting system has to:

Reduce Energy Consumption The government, as well as clients demand that systems are energy efficient. A current day lighting system realizes this by regulating light levels with daylight sensors, turning luminaires off automatically when a room is vacant and even shut down parts of a lighting system if it is not in use.

Respect Demand Response policies Grid operators are increasingly altering prices of energy based on demand of the grid. Modern day lighting systems can respond by altering the light level, or lowering the length of time lights stay on when users are not present, in an effort to reduce power consumption and therefore cost.

Allow Reconfigurability Clients demand that a lighting system can be reconfigured easily when topology changes. Furthermore, suppliers may want to reuses luminaires and sensors in new buildings themselves. Instead of selling devices, they can sell the service "light".

[1]A luminaire (plural: luminaires) is a device capable of producing light. A luminaire is the complete device, armature, light bulb, etc.

Finally, many non functional requirements are also specified. For example, lighting systems have to be:

Scalable Projects become bigger, some lighting systems have in the order of thousands to tens of thousands luminaires and sensors. Current day lighting systems have to scale to such a level while doing so with as little resources as possible.

Synchronized As systems grow, maintaining the shared state becomes a challenge. The larger the system, the more likely that the system grows out of sync. Current day lighting systems have to be resilient and guarantee that the shared state is maintained, or at least recovered within a certain time.

Fast Not only can large scale systems become inconsistent, their timing performance can also degrade. Current day lighting systems often have hard requirements that the system is responsive, only allowing a certain level of latency.

It is important to note that many of these requirements contradict each other, an ideal lighting system with respect to energy usage is always off, whereas such a system is not user-friendly nor safe. A fast system uses expensive hardware to increase its speed, whereas a scalable system should scale with the least amount of resources possible.

Creating a contemporary lighting system is finding a balance between these conflicting requirements. Furthermore, contemporary lighting systems are more complex, not only consisting of luminaires and simple buttons, but all sorts of sensors and most importantly: controllers. A controller acts as a decision maker, sensors faithfully report detections and luminaires produce a certain level of light but it is the controller that decides how sensor input translates to desired output level for luminaires. These controllers are the key to make lighting systems smart.

1.2 The Prisma Project

The traditional way of developing lighting systems is very hardware oriented: creating sensors, luminaires, controllers, etc. Nowadays however, companies like Philips Lighting are transitioning to selling *Lighting as a Service*. In this model, Philips Lighting still supplies the hard- and software, but shifts its focus to a subscription-like model that sells "light". Philips Lighting remains owner of the hardware, strengthening the (re)configurability and reuse requirements of such devices. This new focus unveils new challenges: creating lighting systems that can be tailored to the need of the customers is hard and also testing complete systems is non-trivial.

The scope of acceptance criteria also broadens. In the past luminaires were required to for example provide a certain amount of light on a (reference) table but nowadays, systems should also for example be *energy efficient*.

This new approach to lighting systems requires a new development strategy. Philips Lighting acknowledges this need and started, together with TNO-ESI, a project called Prisma. The Prisma project aims to harness domain engineering to ease development

of these complex systems. Benefits of this approach are that it is focused on reusability and early analysis, that it offers abstractions from the actual system and separates stakeholders.

1.3 A DSL Approach

Key to this domain approach is a good domain model. TNO-ESI and Philips Lighting have created a set of domain specific languages (DSLs) using which one can specify lighting systems. These languages can be categorized into two parts, languages that specify the lighting system itself and languages that specify the environment in which the lighting system operates. The three core languages Building, Template and Control specify *layout*, *behavior* and *layout to behavior mapping* respectively.

Building specifies the layout of a lighting system. Instances of the Building DSL capture *rooms*, *luminaire locations*, *sensor locations* etc.

Template specifies behavior of (parts) of a lighting system. Instances of the Template DSL specify how various actuator/sensor groups are connected in functionality. Templates specify abstract functionality, e.g. when a group "occupancy"-sensors does not measure any occupancy in a room for 15 minutes, then the "office luminaires"-group goes off.

Control specifies the mapping of behavior specified in templates onto the physical layout of a building. In essence, a control specification instantiates templates by assigning actuators/sensors to the various actuator/sensor groups specified in the template.

This framework of core languages is currently being extended by specifications of *deployment* and *network*. Deployment specifying how functionality of a lighting system is distributed over a set of physical devices. Network specifies the topology and characteristics of the underlying network over which these devices communicate. These languages are in early development, and therefore out of scope for the purpose of this report. Many considerations, however, are made with the assumption that these languages are on the horizon.

1.4 Analysis in the Domain Approach

The domain approach aids the standard design/implementation/validation cycle that is common in system design. Lighting systems are designed and implemented as a set of instances of the domain languages covering various parts of the system. Verification of this implementation can happen in two ways:

Simulation uses prerecorded or ad hoc generated input to get an intuition whether the system is behaving correctly.

Model Checking does an exhaustive search of all possible states a system can be in. A model checker evaluates whether a *property* holds for a system. If such a property does not hold, it produces a counterexample.

Both approaches offer their own benefits and drawbacks. Simulation is very tangible in a sense that a possible error situation can be shown in the visualizer. Simulation can also be *open*, parts of the system can be implemented by real hardware where other parts remain simulated. But, simulation is not exhaustive, it only shows errors that are found using predefined or ad hoc generated input traces. Model checking on the other hand is exhaustive but has drawbacks related to this exhaustive search. Model size greatly influence the feasibility of checking these systems and error traces can be unrealistic or hard to comprehend. Finally, model checking requires the system to be closed: the entire system needs to be modeled.

1.5 The Proposed Query language

These verification techniques require a way to specify properties that have to hold but no such language was present prior to this research. In this report we outline the Query language that aims to specify properties of lighting systems. It is designed to allow specification of properties like:

- No light is on when no occupancy has been detected for 15 minutes.
- Under average usage scenarios the energy consumption of the system is at most x KWh.

Specification languages are not new to the domain of computer science, for example the *temporal logics* allow specifying these properties in very mathematical fashion. However, the Query language described in this report has a different set of requirements:

Easy to use Light designers and other stakeholders must be able to write queries of the system without having extensive knowledge of the formalisms employed to verify their correctness.

Complete The language must support or be easily extensible to specify any possible property that is relevant to the lighting domain.

Verifiable Queries written in the proposed language must be verifiable using current verification tools.

Traceable When a property is checked, its results should be conveyed to a user in such a way that (s)he understands whether the reported error situation is a concern to the lighting system.

Integrated The Query language must be a part of the domain approach as a whole. It should have similar syntax and it should offer the ability to reason about the other models present in the domain approach.

Extensible The Query language must be extensible to support new propositions and must be portable to new domains.

1.6 Related Work and Contributions

The work presented in this report touches upon aspects of various verification methods that have arisen in the field, in this section we discuss three: model checking, simulation and runtime verification.

1.6.1 Model Checking

Model checking has been pioneered in the eighties by Clarke and Emerson (1981) and Queille and Sifakis (1982) concurrently. They were the first to propose automatic proving of whether a system adheres to a specification given in temporal logics. The two temporal logics prevalent, LTL and CTL were created by Pnueli (1977) and Clarke and Emerson (1982) respectively. In current day, NuSMV by Cimatti et al. (1999) is a commonly used model checker for temporal specifications in these logics. Temporal logics and their accompanying model checking algorithms have since been extended. For example TCTL and MTL by Larsen et al. (1995) and Koymans (1990) respectively allow not only to reason about temporal order of states or events, but also about their timedness. UPPAAL by Bengtsson et al. (1996) is a model checker capable of checking TCTL formulas. Model checking has also been successfully applied in practice, for example, Doornbos et al. (2015) have used UPPAAL to verify robustness of lighting systems.

1.6.2 Simulation

Simulation is very common in all engineering practices. For example, Verriet et al. (2013) have modeled a warehouse system where items can be stored and retrieved. Their simulation allows to visually, but also automatically, validate the implications of implementation choices for warehouse systems. Currently prevalent is the concept of co-simulation. HLA, for example, is a standard defined by the IEEE (2010) which various types of simulators can implement to support cooperatively simulation. HLA has gained much traction lately, the Co-simulation framework, for example, is heavily inspired by HLA. Simulation can also be used to generate traces which can be visualized and property checked by tools such as TRACE by Hendriks et al. (2016).

1.6.3 Runtime Verification

Model checking and simulation are often performed design time: during the development of the system. Runtime validation, as for example described by Leucker and Schallhart (2009), checks that a system adheres to its specification at *runtime*. An example is the work by Kurtev et al. (2017), they have created a language called ComMA in which interfaces between components can be specified and various constraints on these interfaces

can be checked at runtime. They have applied ComMA in the medical domain and found errors in the implementation of an operating table.

1.6.4 Usability of Property Specification Languages

Common to all these techniques is a need to specify properties of systems thoroughly in a way that can be checked or quantified. In the model checking community, these properties were commonly specified in temporal logics. Temporal logics however, are very abstract and various researchers have acknowledged that they are not suitable for users unfamiliar with formal logics. This observation led to the creation of libraries of commonly used property specification patterns. Dwyer et al. (1998) for example, created a pattern library in which properties are specified as combinations of *scopes* and *patterns*. Scopes specify when a pattern has to hold and patterns specify what exactly holds in that scope. For example the property specification "Before {some sensor event}: Universally {light is on}", describes a property that says that before some sensor event, the light has to remain on. They have provided translations to (timed, probabilistic) temporal logics for their pattern/scope combinations and argue that these pattern libraries are easier to use than traditional temporal logics.

1.6.5 Our Contributions

The language we present in this report applies a pattern library approach to both formal verification and simulation. We have created a language in which lighting system properties can structurally be recorded using pattern/scope combinations and provide translations to both UPPAAL and the Co-simulation Framework. Part of this translation is domain generic, we have developed a new translation of pattern/scope combinations to what we call monitors and rules that is more universally applicable in tooling. While based on the work by Gruhn and Laue (2006), we do provide more pattern implementations as well as a tighter split between *observing* the system and *raising a verdict*.

Furthermore, we have conducted validation on whether a pattern library approach is usable to domain experts using a prototype language and have used the same prototype to demonstrate the need for formal verification in the light domain.

Finally, we wish to emphasize that we think that a Query language similar to the one presented here can be applied across domains and in various tools. We could imagine that a similar language can be employed in a different domain requiring, for example, runtime verification.

1.7 Outline

The general structure of this report is divided into parts consisting of chapters. Per part we provide an outline of the part to come. The five parts:

Introduction & Context establishes the context in which the proposed Query language is operating, defines requirements and research questions.

Background provides academic context to this work. This part discusses the fundamentals of model checking as context, presents pattern libraries as a starting point for the proposed Query language and finally discusses the established theory of domain specific language construction with a specific focus on Xtext as a tool of choice.

Query Language Design discusses in detail how the Query language was created, syntax and semantics of the proposed Query language and translations to the tools we support.

Validation outlines the steps taken to assert that the language conforms to the specified requirements.

Conclusions and Closing Remarks reevaluates the research questions posed in the introduction and argues that the design, using the validation steps, answers the research questions. Furthermore, this section outlines future work that can be performed to strengthen the Query language.

The remainder of this part consists of three chapters. Chapter 2 outlines the context in which the Query language is going to operate and discusses the current domain specific languages that are created as part of the Prisma project. Chapter 3 outlines the requirements and research questions for the Query language.

2 The Prisma Project's Domain Approach

In essence, a lighting system is a large scale heterogeneous distributed system. Lighting systems come with the same problems as other distributed systems, they are often regarded as a single system of many components that has to maintain a shared global state. Not only do they have the problem of synchronicity, their controllers are often heavily constrained in resources. As with any other large scale distributed systems, modularity, configurability and reusability are key.

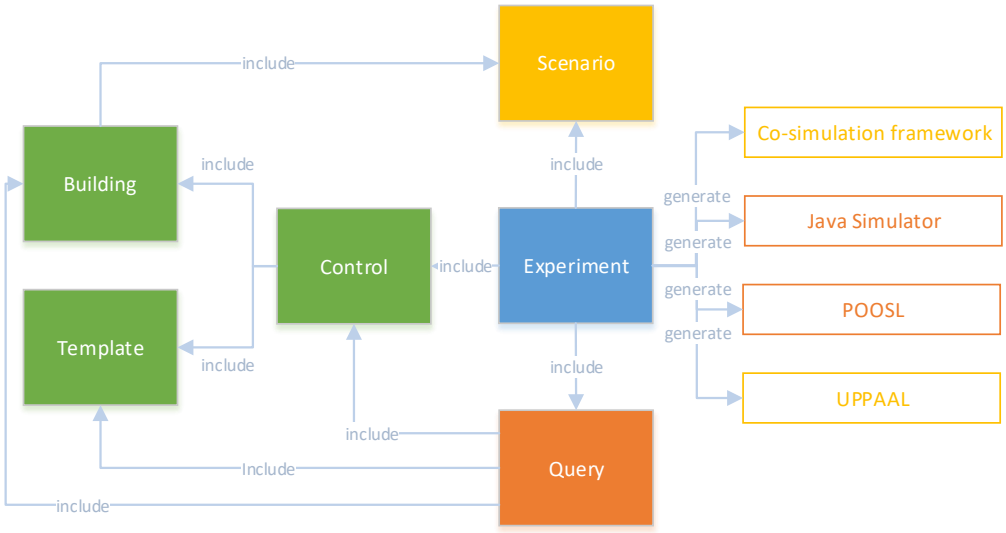


Figure 2.1: All relevant input DSLs for the simulation flow of a lighting system

The Prisma project aims at creating a development framework that allows for full specification of lighting systems and their environment. TNO-ESI has proposed a *domain approach* in which a lighting system is specified using *domain specific languages*. Each of these domain specific languages is carefully crafted to capture a single aspect of the system. The complete set of languages form what is known as the *domain model*. This domain model is an abstract representation of lighting systems and used to:

Make Specification Explicit In traditional software development, requirements are provided before development and checked after. In a domain modeling approach, such

requirements are encoded into the model itself. The more of these requirements can be encoded into the model, the more important this model becomes. Instances of a strong domain model can serve as a single canonical artifact reducing the need for informal specifications.

Support Early Verification As an instance of a domain model is a representation of the lighting system currently created, such an instance can be checked before code generation. This early validation finds specification errors before a system is deployed reducing potential costs of development mistakes.

Target Multiple Technologies A rich domain model contains all needed specification to implement such a lighting system using physical devices. Furthermore, a strong domain model allows translating to various technologies. Ideally, a single toolset based on the domain modeling approach can eventually translate to all currently supported platforms within Philips Lighting, reducing the need for experts in a certain technology.

TNO-ESI has defined *flows* that define transformations on instances of the domain model to various targets. These transformations can require new auxiliary specifications specifically for the target, for example, the *code generation flow* defines auxiliary models to capture technology specific nuances. For the Query language, we are most interested in formal verification and simulation flows. All auxiliary models and transformations for these two flows are shown in Figure 2.1. In the remainder of this chapter we will focus on these languages and briefly discuss the tools supported by the simulation and formal verification flows.

2.1 An Example Lighting System

Throughout this chapter, we will look at a concrete example of a lighting system and how it is to be specified in these domain languages. The lighting system we look at is a lecture room with "Presentation" mode. Activating presentation mode dims the light in front of the room slightly whereas the luminaires in the room are almost off. This mode is operated by a button located in the front of the room.

Besides this presentation mode, there is also a normal "On"-mode. This mode is automatically engaged when the luminaires were off and occupancy is detected or can be reached by pressing the button again when the system is in presentation mode.

Finally, there is hold time behavior, when no occupancy is detected in the "On" and "Presentation" modes for a certain amount of time, the system automatically goes to the "Off" mode.

The entire system is realized in a room with six luminaires, an occupancy sensor and a light button and is depicted in Figure 2.2.

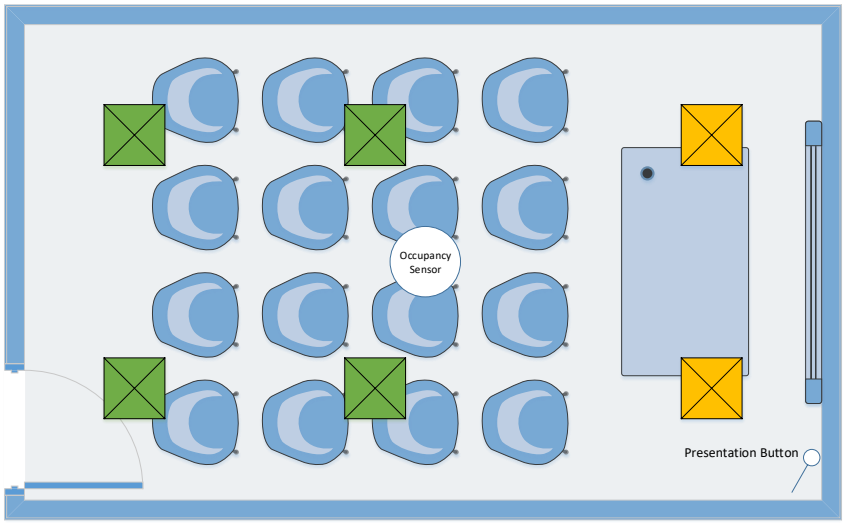


Figure 2.2: Physical layout of an example lighting system.

2.2 Lighting System Behavior Specification

Common to all flows is a specification of the lighting system. This bare specification specifies the physical layout of the building, the abstract behavior of subsystems and the mapping of behavior to the physical system. A lighting system specification consists of a Building, Template and Control instance:

Building is a high-level description of the physical appearance of a building, where luminaires and sensors are located, flooring, rooms, etc.

Template specifies how abstract groups of luminaires and sensors behave. In essence is a template a simple state machine, each state specifies levels for luminaire groups, and the system transitions between various states by listening to input of sensor-groups or timers.

Control instantiates templates by binding the actual luminaires and sensors specified by the Building DSL instance to the abstract sensor/actuator groups from Template DSL instance(s).

2.2.1 Building DSL

The Building DSL has as its goal to capture the *functional* topology of a building. Syntactically an instance of Building consists of:

Name The name of the building.

Perimeter The area that the building spans.

Floor A list of floors in this building, every floor consists of Rooms.

Room A specification of a single room on a floor, consists of its physical location on the floor it's on, a list of sensors and actuators. A sensor defines what they are capable of, whether they can sense daylight or movement or are a simple button, etc. Finally, an actuator, just like a sensor, has capabilities specified like whether it can be dimmed or can produce multiple colors, etc.

The building definition of our example building is given in Listing 2.1, we see a definition for a single floor, with a single room that has two sensors and six actuators (luminaires). All sensors/actuators have coordinates and a feature that specifies their use. For example, the sensor "SensorTheater" detects occupancy.

```
Building OfficeBuilding
  Perimeter (0, 0) (1000, 0) (1000, 500) (0, 500)
  Floor Floor0
    Perimeter (0, 0) (1000, 0) (1000, 500) (0, 500)
    Room Theater
      Perimeter (0, 0) (1000, 0) (1000, 500) (0, 500)
      Sensor SensorTheater Features Occupancy Coordinates (500, 250)

      Sensor PresentationModeButton Features Button
        Coordinates (100, 500)
      // The two luminaires in front of the room
      Actuator LuminairePresent11 Features Dimmable
        Coordinates (100, 100)
      Actuator LuminairePresent12 Features Dimmable
        Coordinates (100, 400)
      // The four luminaires that normally light the room
      Actuator LuminaireRoom1 Features Dimmable Coordinates (500, 100)
      Actuator LuminaireRoom2 Features Dimmable Coordinates (500, 400)
      Actuator LuminaireRoom3 Features Dimmable Coordinates (900, 100)
      Actuator LuminaireRoom4 Features Dimmable Coordinates (900, 400)
```

Listing 2.1: Building specification for example system

2.2.2 Template DSL

Instances of the Template DSL specify the abstract behavior of functionally equivalent parts of a lighting system. To do so, a Template specifies behavior independently of physical location and only argues about (uninstantiated) sensor- and actuator groups. The behavior of an instance of the Template DSL[1]is represented as a state machine, each state of a Template specifies the desired light levels of actuator groups.

[1]The term "instance" is overloaded, an instance of the Template DSL refers to a specification of a Template as described in Section 2.2.2, an instance of a template refers to a controller implementing the behavior of that specific template.

An instance of the Template DSL consists of a list of *Template*s. A Template syntactically consists of:

Name The name of the template

Parameters A list of configurable values for this template allowing possible reuse. Parameters can, for example, be used to specify similar lighting subsystems with the same behavior, but different corresponding output values.

SensorGroups A list of named sensor groups that are needed to implement the behavior specified in this template.

ActuatorGroups A list of named actuator groups that are needed to implement the behavior specified in this template.

States Distinct states in which this lighting (sub-) system may reside. Every state also specifies the desired output level of all connected actuator groups for instances of this template.

InitialState The default state of the behavior specified in this template.

Transitions A list of transitions from state to state that specified which triggers, sensors or timers, lead to which new state.

The state machine of this room consists of three states, one for the "Off" behavior, one for the "On" behavior and a state for the "Presentation" behavior. Five parameters exist: the hold time specifies how long the lighting system remains active after vacancy was detected, the remaining four specify the light level in the various states. For the presentation mode, two parameters are used: a parameter to specify the light level for room luminaires and one for the front luminaires.

The exact implementation of the lighting system of Section 2.1 can not be given due to confidentiality requirements of Philips Lighting. We wish to stress however that the implementation of behavior in a Template instance includes design choices. For example, if a presentation is stopped for a coffee break, and everyone leaves the room for at least the hold time, should the lighting system remember that prior to the hold time expiration the system was in presentation mode? Whether it does or not is a design choice. If a Query language instance finds an error in a Template, either the Template does not cover the entire (informal) specification of the system, or the specification of the system is not precise enough. A goal of the Query language as proposed in this report is that stakeholders re-evaluate their specification to increase its preciseness.

2.2.3 Control DSL

Instances of the *Control* DSL map the logical sensor/actuator groups described in a template onto the physical world of the building. To do so, in an instance of the Control DSL, one can instantiate controllers based on a template. In this controller, a mapping is

made from sensors/actuators in the Building to the logical sensor/actuator groups that a template requires. Syntactically a control instance consists of:

Name The name of the lighting system as a whole.

Building The building for which templates are being instantiated.

Controllers A list of controllers implementing a template by assigning parameters and instantiating sensor/actuator groups.

Syntactically a single controller consists of:

Template The abstract behavior that this controller is instantiating.

Parameters A list of parameters inherited from the Template but overridden by this controller.

SensorMapping A mapping from all sensor groups of a template, to actual sensors in the building that is being mapped.

ActuatorMapping A mapping from all actuator groups of a template to actual actuators in the building being mapped.

The exact Control instance of the lighting system of Section 2.1 can not be given due to confidentiality requirements of Philips Lighting.

2.3 Analysis Flows

Depending on the flow, there is a need for auxiliary specifications. For example, when compiling the lighting system configuration to actual hardware, we would need information regarding the technology being deployed to as well as the mapping of logical controllers to physical devices. While code generation is an interesting aspect of the Prisma project, for the purpose of the Query language these mappings are below our abstraction. The Query language can only reason about models of the system after all. For the case of analysis, we are interested in two types of auxiliary specification: usage specifications and environment specifications.

Usage specification is the systematic recording of test inputs, behavior patterns and other normal interaction with a lighting system and is realized in two different models:

Occupancy is a description that specifies how people move through the building, schedules of people, etc.

Scenario is a derived DSL instance that maps the external stimuli from the Occupancy DSL to a list of sensor activations due to those stimuli.

Environmental specification describes how the lighting system is influenced by systems out of our control. For example, we could create a model of how much daylight is observed throughout the year and, using occupancy and energy profiles, analyze best and worst case energy usage throughout the year.

2.3.1 Scenario DSL

A scenario is a timed list of actions that could either be prerecorded from an actual run of the lighting system or created as a test case for the system. A sensor event is defined as a time stamp, type of event (button press, daylight level change, occupation change event) and a new value. For our example system, a possible scenario is shown in Listing 2.2.

In the model as defined by TNO-ESI, scenarios should be created using Occupancy instances, listing possible occupancy patterns ("some users show up at 7, have four coffee breaks etc."), changes in daylight level etc. but this has not yet been implemented.

```
Scenario OfficeBuildingScenario
Building OfficeBuilding
Sensor OfficeBuilding.Floor0.Theater.PresentationModeButton
  Time 20 seconds Pressed
  Time 50 seconds Pressed
  Time 700 seconds Pressed
Sensor OfficeBuilding.Floor0.Theater.SensorTheater
  Time 0  seconds Occupied
  Time 70 seconds Vacant
  Time 100 seconds Occupied
  Time 250 seconds Vacant
  Time 650 seconds Occupied
```

Listing 2.2: Example scenario specification of example system

2.3.2 Experiment DSL

The analysis model, called Experiment, combines the system under test, input specification and environmental models. Instances of Experiment are simple, specifying a name of the configuration and referencing Scenario and Control instances. For the example system, an instance of Experiment is given in Listing 2.3.

```
Experiment ExampleExperiment
System OfficeBuildingSystem
Scenario OfficeBuildingScenario
```

Listing 2.3: Experiment specification of example system

2.4 Future Extensions

The core specification is currently being extended to allow specification of Deployment and Network:

Deployment is a description of how various parts of the abstract functionality will be divided to different physical devices. Can be used both in analysis as well as code

generation.

Network is primarily a specification of network topology within a lighting system, how controllers and luminaires are connected. Secondly, it describes characteristics of the nodes and edges of such a network. These characteristics, for example, the transmission speed of an edge in the network, could be used to model latencies. Similar characteristics could be used to model package loss.

While these languages are not mature enough to be considered inputs of the Query language right now, they are prime candidates for future extension of the Query language.

2.5 Query language

The Experiment model combines various aspects of simulation and verification in a single model. This combined model can be translated to various external tools in both simulation and verification. The domain model proposed in the Prisma project translate to model checkers (UPPAAL by Bengtsson et al. (1996)) and simulators (POOSL by Theelen et al. (2007), a Java Simulator and the Co-simulation Framework).

Before the work outlined in this research, the only way to verify the correctness of these models was by either manual observation in visualizations or by incidental specifications of properties for these models. For example, Doornbos et al. (2015) have specified properties of parts of a (customly modeled) lighting system directly in UPPAAL.

The Query language is created to solve many of the issues of manual specification (or observation):

Automate Verification Manual verification of large lighting systems by observation is infeasible for large scale lighting systems. Tools have to be provided to automate such a process.

Hide Complexity Manual specification in tools like UPPAAL is complex and requires a great deal of formal background. An abstraction is needed to aid domain experts in writing concise specifications in terms aligning with their knowledge and expertise.

Introduce Formal Analysis in Simulation The custom designed Co-simulation Framework had no notion of property checking. We have added ways for this Co-simulation Framework to verify properties.

We have chosen two tools to translate our Query language to UPPAAL and the Co-simulation Framework:

UPPAAL is a model checker that can formally prove the correctness of properties of timed automata. The lighting systems as specified in Prisma models transform into a set of automata that can be analyzed using UPPAAL.

Co-simulation Framework is a simulator for lighting systems. This tool is specific for lighting systems and offers a runtime simulator of lighting behavior. Furthermore, a visualizer called Spectrum was developed that connects to this framework and can display the current state of the lighting as well as allowing interactive triggering of sensors.

Other tools, like POOSL and the Java simulator, are either redundant or largely abandoned in favor of these two tools.

3 Research Questions

In this report we present a *domain specific language* that captures the needs of various *stakeholders in light design* in specifying *formal queries* for *lighting systems*.

The language we intend to specify has to be:

Easy to use Light designers and other stakeholders must be able to write queries of the system without having extensive knowledge of the formalisms employed to verify their correctness.

Complete The language must support or be easily extensible to specify any possible query that is relevant to the lighting domain.

Verifiable Queries written in the proposed language must be verifiable using current verification tools.

Traceable When a property is checked, its results should be conveyed to a user in such a way that (s)he understands whether the reported error situation is a concern to the lighting system.

Integrated The Query language must be a part of the domain approach as a whole. It should have similar syntax and it should offer the ability to reason about the other models present in the domain approach.

Extensible The Query language must be extensible to support new propositions and must be portable to new domains.

We have also formulated research questions. These research questions differ from the questions posed in the predecessor research topic report (Buit (2017)) for reasons outlined in Chapter 7. For clarity this section outlines the *new* research questions:

Research Question Can we, in the context of the domain approach of the Prisma project, create a framework and prototype of a domain specific Query language in which lighting properties can be specified and formally verified? This framework and prototype must align with the technical knowledge of the stakeholders in the lighting domain both in specifying properties and conveying verification results.

Subquestion 1 Are pattern libraries a solid foundation of an easy to use Query language?

Subquestion 2 Can a pattern library approach cover all use cases of a formal specification language for the lighting domain?

Subquestion 3 Does a pattern library approach allow easy extension when new use cases are discovered?

Subquestion 4 Can a pattern library approach be implemented in both UPPAAL and the Co-simulation Framework?

Subquestion 5 Can syntax and semantics be developed that allows intuitive specification of properties of lighting systems?

Subquestion 6 Can the Query language for the lighting domain be integrated within the approach of Prisma?

Subquestion 7 Can the Query language for the lighting domain trivially be extended to cover new use cases?

Subquestion 8 Can we show the value of a Query language for the lighting domain to relevant stakeholders within Philips Lighting?

Subquestion 9 Can we convey to a stakeholder that a property is not correct in a way that indicates what causes the faults without needing fundamental knowledge of temporal logic or model checking?

3.1 Approach and Validation

To answer these questions, we will be creating a domain specific Query language for the lighting domain as a prototype. In the design of this prototype, we will extensively discuss all choices made to warrant the language criteria stated. The full design is described in Part III. Furthermore, we will show validation steps taken to verify that the language does indeed meet its requirements in Part IV. In the final part, Part V, we will reflect on the choices made in the design of the Query language and conclude whether its requirements were met.

Part II

Background

Part Outline

This part of the report serves two goals. The first goal is to put the work of this report into the broader context of formal verification tooling currently present in the field of computer science. Secondly, this part outlines important concepts that were used in the creation of the proposed Query language. The part consists of three chapters:

Chapter 4 serves as a broader introduction to the field of model checking. This chapter will outline both the history of formal verification, discuss modeling formalisms used to create verifiable models and finally logics to specify various temporal logics used to create properties that these models must adhere to.

Chapter 5 serves as an introduction to pattern libraries. Pattern libraries are an abstraction over full temporal logics. We use (a subset) of various pattern libraries directly in the proposed Query language and will, therefore, discuss these libraries extensively.

Chapter 6 is an introduction to the creation of domain specific languages (DSLs). We will outline what defines a domain specific language and provide a framework to classify DSLs. As context, we outline how general purpose languages are created and how the development of domain specific languages was shaped by the adoption of language workbenches. Finally, we present Xtext, the language workbench used to create the Query language for the lighting domain.

4 Model Checking

In traditional design processes, system design starts with specification of desired functionality. The system is then implemented, and it is verified that the implementation conforms to the specification that was created before.

Verification techniques come in many forms, in software development the creation of tests in different forms (unit tests, integration tests, etc.) is common.[1] These tests often create a scenario and check whether the system under test is behaving correctly in that scenario. The ability of testing to find errors is dependent on the *coverage* that these scenarios provide. Take for example a function that computes the absolute value of its input x, and a test that only checks that for $x = -1$ the function produces 1. A faulty implementation of this function, for example, x^2, passes this test, while obviously wrong. There has been a lot of research on how to improve testing, but the consensus is that testing can only find errors that are on the paths and inputs being tested.

Another common technique to establish whether a system conforms its specification is called *model checking* (Originally researched by Clarke and Emerson (1981) and Queille and Sifakis (1982) concurrently). In model checking, the system under test is transformed into a mathematical model. Commonly, these models are given as *finite state machines*[2], a representation of the system that has a set of *states* it can be in, and a *transition relationship* that defines which state can lead to which other states under which condition.

The specification of the system is written in a property specification language. One common class of property specification language are the *temporal logics* as pioneered by Prior (1957). Specifications written in temporal logics can argue about whether a certain formula holds forever, or eventually, or until some other formula etc. and proves a valuable addition to traditional propositional logics. While *temporal* implies that these logics can reason about *when* a property holds, they can only reason about the ordering of events. Specifications about punctuality, for example, cannot be expressed in temporal logics. Timed temporal logics came as a logical extension to cover these specifications.

In this chapter, we evaluate two modeling formalisms, Kripke structures and timed automata for temporal and timed temporal properties respectively. We also look at five logics, CTL, LTL, and CTL* for temporal properties and MTL and TCTL for timed temporal properties. This chapter should be read as an introduction to model checking, for full details we would like to refer to the various papers that introduce these models and logics as well as the book "Principles of Model Checking" by Baier et al. (2008).

[1] The origin of the distinction between testing and debugging is Myers (1979), many others have extended the standard theory of testing.

[2] As first described by McCulloch and Pitts (1943), further extended by Moore (1956) and Mealy (1955) and many others.

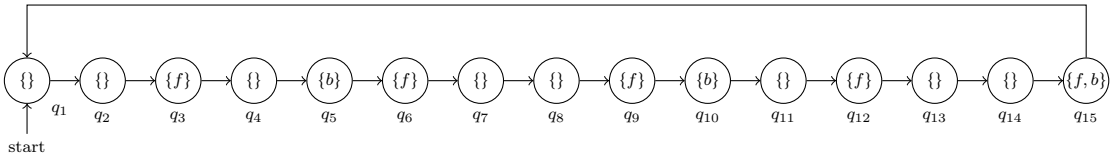


Figure 4.1: A Kripke Structure for a Fizz-Buzz program.

4.1 Kripke Structures

A *Kripke structure* by Kripke (1963) is an abstraction over the behavior of a system. It represents a system in an abstract manner by defining in what states that system can be, and what states are reachable from a particular state. A Kripke structure says nothing about under what condition state changes can occur, it just models that there is a possible way to do so. Furthermore, a Kripke structure defines a *labeling function* that assigns to each state a set of (predefined) *atomic propositions* that hold in that particular state. For example, consider the program "Fizz-Buzz" that counts from 1 to ∞, and prints "Fizz" for all iterations divisible by three and "Buzz" for all iterations divisible by five. A Kripke structure for this program is shown in Figure 4.1, the labeling function assigns the atomic proposition f to all states that should print "Fizz" and b to all states that should print "Buzz".

Formally, let a Kripke structure $M : (S, I, R, L)$ over a given set of atomic propositions (AP) be:[3]

S	Set of states
$R \subseteq S \times S$	Transition relation with the added restriction that it must be *left-total*, there must be an outgoing transition from every state.
$I \subseteq S$	Set of initial states
$L : S \to 2^{AP}$	Labeling function

A *path* π in a Kripke structure as a sequence of states $s_0, s_1, s_2, \ldots$ for which $s_0 \in I$ and $(s_i, s_{i+1}) \in R$ for all $i \geqslant 0$. The *word* ω of a path is a sequence of sets of atomic propositions that hold for each state in the path, $L(s_0), L(s_1), L(s_2), L(s_3), \ldots$

4.2 Linear Temporal Logic

Linear Temporal Logic (LTL) was proposed by Pnueli (1977). LTL formulas evaluate over infinite paths through a Kripke Structure. Formulas in LTL are extensions to traditional propositional logic (e.g. $p \vee q, p \wedge q, \neg p$) with the propositions p, q being the atomic propositions of a Kripke Structure. Besides standard propositional logic, LTL defines two base temporal operators, next (X) and until (U).

[3]Definition based on (Clarke et al., 1999, p. 14)

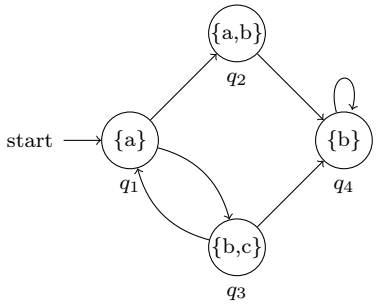


Figure 4.2: An example Kripke Structure used to evaluate LTL and CTL formulas

$\mathrm{X}\,\phi$ In the *next* state ϕ holds

$\phi\ \mathrm{U}\ \psi$ ϕ must hold *until* ψ holds.

Using these operators we can derive operators eventually (F), globally (G) and weak until (W). Firstly, let us see the meaning of these operators.

$\mathrm{F}\,\phi$ There is *eventually* a state in which ϕ holds.

$\mathrm{G}\,\psi$ In all states, ψ holds.

$\phi\ \mathrm{W}\ \psi$ ϕ holds until ψ holds, or ϕ holds forever.

The construction of these is as follows:

$$\begin{aligned} \mathrm{F}\,\phi &\iff \mathit{true}\ \mathrm{U}\ \phi \\ \mathrm{G}\,\phi &\iff \neg\,\mathrm{F}(\neg\phi) \\ \phi\ \mathrm{W}\ \psi &\iff (\phi\ \mathrm{U}\ \psi) \vee \mathrm{G}\,\phi \end{aligned}$$

And therefore the grammar of LTL is:

$$\phi := \mathit{true} \mid \mathit{false} \mid p \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \mathrm{X}\,\phi \mid \mathrm{F}\,\phi \mid \mathrm{G}\,\phi \mid \phi\ \mathrm{U}\ \psi \mid \phi\ \mathrm{W}\ \psi$$

Ultimately, we are interested in whether a given formula is satisfied in all possible paths through a model. We denote $M \models \phi$ to express this. How this is exactly defined falls outside of the scope of this introduction to temporal logic, but to give some intuition we will evaluate whether the model M from Figure 4.2 satisfies a few formulas and if not, we will show a counterexample.

$M \models \mathrm{X}\,b$ Holds, starting in the initial state, in all following states b indeed holds.

$M \models \mathrm{X}\,\mathrm{G}\,b$ Does not hold, there is a path starting in the initial state q_1 which loops from q_2 back to q_1, while b does hold in q_2, it does not hold forever.

$M \models \mathrm{F}\,\mathrm{G}\,b$ Does not hold, this is a weaker version of the previous specification, but it is still not guaranteed that a path eventually leads to a set of states for which b holds forever, the same counterexample still applies.

$M \models \mathrm{G}(a \vee b)$ Holds, in any state either a holds or b holds.

$M \models \mathrm{X}(a \rightarrow \mathrm{G}\,b)$ Holds, if in the next state a holds, b holds from that state onwards.

$M \models \mathrm{G}(c \rightarrow (\mathrm{X}\,a \vee \mathrm{G}\,b))$ Holds, if in a state c holds, then either in the next state a holds, or b holds forever.

$M \models (a \vee (b \wedge c))\ \mathrm{W}\ \mathrm{G}\,b$ Holds, either $a \vee (b \wedge c)$ hold forever (e.g. looping between q_1 or q_3) , or at some point b will hold forever.

Computation Tree Logic Computational Tree Logic (CTL) was proposed by Clarke and Emerson (1982) and reasons about trees. The syntax of CTL is split into two, *state formulas* (denoted with Greek uppercase letters: Φ, Ψ) and *path formulas* (denoted with Greek lowercase letters: ϕ, ψ). Construction of a valid CTL formula starts with a state formula, with p being any of the atomic propositions of a Kripke structure over (all) its paths the formula is evaluated. The grammar of CTL is as follows:

$$\Phi := \mathit{true} \mid \mathit{false} \mid p \mid \neg\Psi \mid \Phi \wedge \Psi \mid \Phi \vee \Psi \mid \mathrm{E}\,\phi \mid \mathrm{A}\,\phi$$
$$\phi := \mathrm{X}\,\Phi \mid \mathrm{F}\,\Phi \mid \mathrm{G}\,\Phi \mid \Phi\ \mathrm{U}\ \Psi \mid \Phi\ \mathrm{W}\ \Psi$$

Of particular interest are the two state formulas $\mathrm{E}\,\phi$ and $\mathrm{A}\,\phi$. These so called *path qualifiers* E, A specify on which of the paths originating from the current state the path formula ϕ has to hold. The difference between the two is as follows:

$\mathrm{A}\,\phi$ ϕ holds for all paths starting in the current state.

$\mathrm{E}\,\phi$ ϕ holds for at least one path starting in the current state.

The path formula that follows is similar in semantics as in LTL, so, for example, $\mathrm{EF}\,p$ means "along at least one of the paths originating in the initial state, the formula p will eventually hold".

The grammar of CTL is often given as combined operators $\mathrm{EX}, \mathrm{AF}, \ldots$ together with standard propositional logic. This way of specifying the grammar changes nothing about which constructed formulas are valid. Much like LTL, we can cover all possible CTL operators using a stricter subset of operators. We actually only need $\{\mathit{true}, \wedge, \neg, \mathrm{EG}, \mathrm{EU}, \mathrm{EX}\}$ (Positive Normal Form, (Baier et al., 2008, p.333, p.327)) to construct all other possible CTL formulas. Constructing the missing operators $\mathrm{AF}\,\phi, \mathrm{AG}\,\phi$, etc. can be done as follows:

$$
\begin{aligned}
\mathit{false} &\iff \neg \mathit{true} \\
\phi \vee \phi &\iff \neg(\neg\phi \vee \neg\psi) \\
\mathrm{EF}\,\phi &\iff \mathrm{EU}(\mathit{true}, \phi) \\
\mathrm{AF}\,\phi &\iff \neg\,\mathrm{EG}\,\neg\phi \\
\mathrm{AG}\,\phi &\iff \neg\,\mathrm{EF}\,\neg\phi \\
\mathrm{AX}\,\phi &\iff \neg\,\mathrm{EX}\,\neg\phi \\
\mathrm{AU}(\phi, \psi) &\iff \neg(\mathrm{EU}(\neg\psi, \neg(\phi \vee \psi)) \wedge \mathrm{EG}(\neg\psi)) \\
\mathrm{EW}(\phi, \psi) &\iff \neg\,\mathrm{AU}((\phi \wedge \neg\psi), (\neg\phi \wedge \neg\psi)) \\
\mathrm{AW}(\phi, \psi) &\iff \neg\,\mathrm{EU}((\phi \wedge \neg\psi), (\neg\phi \wedge \neg\psi))
\end{aligned}
$$

Using the same Kripke structure from Figure 4.2 we will once again give CTL formulas and argue that the model satisfies that formula or give a counterexample.

$M \models \mathrm{AX}\,b$ Holds, for each path in the next state b holds. This formula is equivalent to LTLs $\mathrm{X}\,b$.

$M \models \mathrm{EG}\,b$ Does not hold, there is no path where b holds forever because b does not hold in the initial state.

$M \models \mathrm{EX}\,\mathrm{AG}\,b$ Holds, there is a path for which from the next state onwards b holds forever, namely the path to q_2. From that point onwards it is impossible to reach a state in which b does not hold.

$M \models \mathrm{AG}(a \rightarrow \mathrm{AF}\,b)$ Holds, if a holds now, eventually b holds. a holds in q_1 and q_2, and in all states that are reachable from these states b holds.

$M \models \mathrm{AG}(a \rightarrow \mathrm{AF}\,\mathrm{AG}\,b)$ Does not hold, if a holds in a state, it is not true that in all next states b holds forever. Looping between q_1 and q_3 is a counterexample because in q_1 the proposition a holds, but b only holds in q_3 and therefore not forever.

CTL* It has been shown that formulas in CTL and LTL are incompatible[4], there are formulas that cannot be expressed in LTL but can in CTL and vice versa. For example, the LTL formula $\mathrm{F}\,\mathrm{G}\,p$[5] cannot be expressed in CTL. Vice versa, $\mathrm{EX}\,p$[5] cannot be expressed in LTL because in LTL a formula has to hold for all possible paths, not just for one.

Emerson and Halpern (1986a) have defined CTL*. Whereas CTL restricts the mixing of path- and state formulas, CTL* has no restriction of this kind. Let us look at the

[4]For example shown by Clarke and Draghicescu (1988); Emerson and Halpern (1986b); Lamport (1980)
[5]Example from `https://en.wikipedia.org/wiki/CTL*`.

syntax of a CTL* formula:

$$\Phi := \mathit{true} \mid \mathit{false} \mid p \mid \neg\Psi \mid \Phi \wedge \Psi \mid \Phi \vee \Psi \mid \mathrm{E}\,\phi \mid \mathrm{A}\,\phi$$
$$\phi := \Phi \mid \neg\phi \mid \phi \wedge \psi \mid \phi \wedge \psi \mid \mathrm{X}\,\Phi \mid \Phi\ \mathrm{U}\ \Psi \mid \mathrm{F}\,\Phi \mid \mathrm{G}\,\Phi$$

Clarke and Draghicescu (1988) have also shown that CTL* is a superset of both LTL and CTL. For example, $\mathrm{EX}\,p \wedge \mathrm{AF}\,\mathrm{G}\,p$[5] is a formula that is in CTL* but neither in LTL nor CTL. Converting any LTL formula in CTL* can be done by prefixing the universal quantifier, whereas all CTL formulas are also in CTL* because CTL* only loosens the requirements on mixing of path and state formulas.

Interestingly, the newly allowed syntax allows us to specify $\mathrm{A}\,\phi$ using the existential quantifier. In CTL, mixing of boolean operators and path qualifiers was not allowed. The relaxation of this restriction in CTL* allows us to write $\mathrm{A}\,\phi$ as $\neg\,\mathrm{E}\,\neg\phi$.

4.3 Introducing Time to Automata

Before introducing timed temporal logics, we have to extend the modeling formalism over which paths these logics are evaluated. This modeling formalism, *timed transition systems*, has two types of transitions, a *delay transition* that only updates clocks and a *discrete transition* that moves from a (logical) state to another state. Because time is often infinite (for example all natural numbers $\mathbb{N}$), so is the state space of this timed transition system. Therefore, we will first discuss *timed automata* Alur et al. (1993) and use them as a finite description of such a timed transition system.

Timed automata are an extension of finite automata that add so called *clocks*. A clock (commonly denoted with x, y, z) is a variable that automatically increases over time, and can only be reset but not otherwise updated. We define *clock valuation* as a function ($\eta : C \to \mathbb{R}_{>0}$) that assigns a (current) time value to all clocks and define $\mathit{Eval}(C)$ as all possible clock valuations for a set of clocks. Furthermore, clock constraints (denoted: CC(C), "The clock constraints over a clock C") are boolean expressions that compare clocks with fixed values; for instance, for a clock x, a clock constraint might be $x < 7$. A timed automaton employs clock constraints as *guards* and *invariants*. A guard enables a transition when its clock constraint is satisfied, so a transition with guard $x < 7$ can only be taken when the clock x has a value less than seven. An invariant is a clock constraint on a state that disallows staying or moving to a state when its the clock constraint is violated. A state with invariant $x < 7$ must be left or cannot be entered before or after the clock x reaches seven respectively. If there is a situation where the only possible continuation violates an invariant, the system is considered *timelocked* and no progress (delay or discrete) can be made.

A timed automaton over a set of atomic propositions (AP) is given by a tuple $(\mathit{Loc}, \mathit{Loc}_0, C, R, \mathit{Inv}, L)$[6]:

[6] A different syntax is used here here to prevent confusion with timed transition systems discussed later. Definition based on (Baier et al., 2008, p. 678-679) but made consistent with the definition of Kripke Structures of (Clarke et al., 1999, p. 14)

Loc	A set of states
$Loc_0 \subseteq Loc$	A set of initial states
C	A set of clock variables
$R \subseteq Loc \times CC(C) \times 2^C \times Loc$	A transition relationship between two states with possibly a clock constraint acting as guard, and possibly resets for any possible subset of clocks
$Inv : Loc \to CC(C)$	A mapping from states to their invariants (which are clock constraints)
$L : Loc \to 2^{AP}$	Labeling function

A *timed path* is a sequence of tuples of state and clock valuations $(s_0, \eta_0), (s_1, \eta_1), (s_2, \eta_2), \ldots$ for which $s_0 \in Loc_0$, $\eta_0 = 0$ and there is a transition from s_i to s_{i+1} and η_i satisfies the guard of that transition, as well η_i satisfies the invariant on state s_{i+1} for all $i \geqslant 0$. For a given timed automaton $(Loc, Loc_0, C, R, Inv, L)$, we define the timed transition system $M = (S, I, C, R, L)$ as a structure that encodes all possible timed paths in a timed automaton. Intuitively, this timed transition system has states for all combinations of locations in the timed automata and all clock valuations and transitions of two types:

Delay Transition A transition from (s, η) to $(s, \eta + d)$ for all $d \in \mathbb{R}_{>0}$ iff $\eta + d$ satisfies the invariant on s.

Discrete Transition A transition from (s, η) to (s', η') iff there is a transition t from s to s' in the timed automata and η satisfies the guard on t and η' satisfies the invariant on s' and η' has correctly applied all resets specified in that transition.

Finally, when evaluating a timed temporal logic over a timed transition system, paths that are *zeno* are not considered. A path that is considered zeno is of infinite length but does not take infinitely long time. Saying that something holds globally, for example, would be impossible for a path that has infinite steps, but only finite duration.

The general flow for model checking a timed temporal logic formula ϕ is: given a timed automaton, transform that timed automata into a timed transition system M, and then prove that for all timed paths p in M, $p \models \phi$.

4.4 Metric Temporal Logic

All temporal logics that have been discussed so far reason only about the sequence of events, whether formulas hold eventually, or globally etc. However, these logics are not expressive enough to say that a formulas hold *after 2 time units*. Both CTL and LTL were extended to remedy this deficiency.

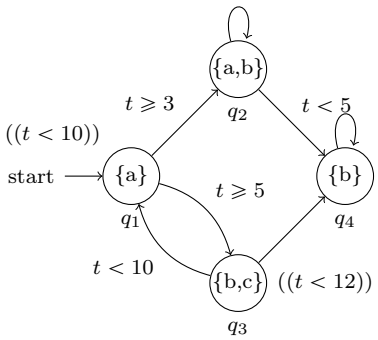


Figure 4.3: A timed automata used to demonstrate semantics of MTL and TCTL, the structure is similar to the Kripke structure in Figure 4.2.

Metric Temporal Logic (MTL) was proposed by Koymans (1990) as an extension to LTL to specify real-time properties of systems. To do so, MTL has added an (open, closed, half-open) interval $J \subseteq [0, \infty)$ to the until operator and its derived eventually (F), globally (G) and weak until (W) operators. Furthermore, the next (X) operator was removed because in a timed formalism there are infinitely many next states where some time has passed which defeats its purpose. Therefore the syntax of MTL is as follows:

$$\phi := \mathit{true} \mid \mathit{false} \mid p \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \mathrm{F}^J \phi \mid \mathrm{G}^J \phi \mid \phi \,\mathrm{U}^J\, \psi \mid \phi \,\mathrm{W}^J\, \psi$$

MTL defines two semantics: point-based and continuous. In point-based MTL, traces of events are used, while the continuous semantics reasons about signals or flows. Another way of thinking about this is that the point-based semantics has the labels on the transitions of the system whereas continuous semantics has the labels on the states. If a system has instantaneous events, it is more reasonable to model it as having point-based semantics, whereas a system that reasons about state and staying in a state is more suitable for modeling in continuous semantics. The paper by Ouaknine and Worrell (2008) lists an example of a traffic light, in continuous semantics a proposition p may be "The light is red" that holds during the light being red, whereas in point-based semantic a proposition p may be "The light turns red" that only holds in the instant the light turns red. Intuitively, $\phi \,\mathrm{U}^I\, \psi$ for continuous semantics means that prior to some $t \in I$, ϕ holds continuously and at t, ψ holds. For point-based semantics, $\phi \,\mathrm{U}^I\, \psi$ means that prior some $t \in I$, each discrete transition contains an event for which ϕ holds and at t the discrete transition contains an event for which ψ holds.

To introduce the semantics of MTL (and in the next section, TCTL), consider the timed automata in Figure 4.3. Since this example system has labels on the states, we will evaluate MTL formulas using the continuous semantics, for each formula we will either argue that it holds, or give a counterexample:

$M \models \mathrm{G}^{[10,\infty)}\, b$ Holds, the invariant on state q_1 forbids paths to stay in q_1 for more than 10 time units and in all other states $q_{2\ldots4}$ the proposition b holds. Important to

note, the state q_1 can be left if the invariant of that state stops to hold, the system can therefore not *timelock*.

$M \models \mathrm{G}^{[12,\infty)}\, \neg c$ Holds, the only state in which c holds is q_3 and the invariant forbids paths that stay in that state after 12 time units. Furthermore, the system cannot timelock in q_3, it can always transition to q_4.

$M \models \mathrm{F}(\neg a \wedge \neg c \wedge b)$ Does not hold, not on all paths eventually only b holds. The issue here is that there is no invariant on q_2 that forbids paths to stay in that state after $t = 5$. A possible counterexample would, therefore, be a path that goes to q_2 and then stays there forever.

$M \models (a \vee c)\ \mathrm{U}^{[0,12)}\ \mathrm{G}(\neg a \wedge \neg c \wedge b)$ Does not hold, this specification is stronger than the previously given example. Because of the lack of an invariant on q_2 a path can stay in that state forever, and therefore the second operand of the until operator will never become true. A specification that replaces the until operator with its weak equivalent is in fact true.

A problem for MTL is that it is in unrestricted form in both point-wise as well as continuous semantics undecidable whether a signal/timed word satisfies an MTL formula (satisfiability problem) as well as whether an MTL formula holds for any possible path/signal that is allowed in a machine M (model-checking problem). A full explanation of this phenomena can be found in the paper by Ouaknine and Worrell (2008). This observation has resulted in the development of various subsets of MTL that are decidable for these two problems. Examples include, but are not limited to:

MITL Metric interval temporal logic by Alur et al. (1996), which forbids the usage of singular intervals.

BMTL Bounded metric temporal logic by Bouyer et al. (2008), which restricts intervals to be finite.

4.5 Timed Computation Tree Logic

Timed Computation Tree Logic (TCTL) is an extension of CTL defined by Alur et al. (1993). Just like in MTL, TCTL removes the Next operator from CTL for much the same reason. Furthermore, like in (unrestricted) MTL, the until operator is given an (open, closed, half-open) interval $J \subseteq (0, \infty)$. This interval intuitively means that, for $\phi\ \mathrm{U}^I\ \psi$ to hold, (1) prior to a certain $t \in I$, $\phi \vee \psi$ holds continuously and (2) at t, ψ holds. The syntax of a valid TCTL formula is:

$$\Phi := \mathit{true} \mid \mathit{false} \mid p \mid \neg\Psi \mid \Phi \wedge \Psi \mid \Phi \vee \Psi \mid \mathrm{E}\,\phi \mid \mathrm{A}\,\phi$$
$$\phi := \mathrm{X}^J\,\Phi \mid \mathrm{F}^J\,\Phi \mid \mathrm{G}^J\,\Phi \mid \Phi\ \mathrm{U}^J\ \Psi \mid \Phi\ \mathrm{W}^J\ \Psi$$

Just like MTL, TCTL is evaluated over timed paths or all timed paths in a timed automaton. To demonstrate the semantics of TCTL, we can return to the example from Figure 4.3. We present TCTL queries for this model, and argue whether they hold or otherwise give a counterexample:

$M \models \mathrm{EG}^{[3,\infty)}(a \wedge b)$ Holds, there is indeed a timed path for which a and b hold forever after three time units. A path can go to s_2 at time three and stay there forever since there is no invariant that would force leaving state s_2.

$M \models \mathrm{EG}^{[5,\infty)}(c \wedge b)$ Does not hold, there is no path on which c and b hold forever after five time units because in q_3 the invariant on the state forces paths to go to state q_4, in which b does not hold.

$M \models \mathrm{EG}^{[3,\infty)}(b \wedge \neg a \wedge \neg c)$ Holds, there is a path on which only b holds after three time units: from q_0 to q_2 at time 3, and immediately to q_4.

$M \models \mathrm{AG}(a \rightarrow AF^{[0,10)}b)$ Holds, if a holds now, b has to hold within 10 time units.

Contrary to MTL, model checking of TCTL is decidable.

5 Property Specification Pattern Libraries

In Chapter 4 we have seen various formalisms to capture temporal and timed properties of systems. Common to these formalisms is that they require a great deal of formal theory to be properly understood. For example, the system property "A message is eventually followed by a response" can be specified using LTL: G(*message* → F *response*). To understand this formula, a user would need to know what the temporal operators G and F mean, what the implies (→) arrow means and what the atomic proposition *message* and *response* mean.

To improve the process of specifying system properties, various researchers have identified common patterns in property specification. These common patterns can then be instantiated to create property specifications for a specific system and be translated to the traditional temporal logics like LTL and CTL.

What is key to this pattern approach is that it is fundamentally a trade-off between expressiveness and understandability. These libraries[1] intend to act as shortcuts for commonly used property specification patterns but are possibly not as concise or expressive as traditional temporal logics. This chapter explores these pattern libraries.

5.1 Untimed Property Specification Pattern Library

Dwyer et al. (1998) were the first researchers to come up with a pattern library approach for property specification. They argued, that just like in software development, formal specifications of system properties consists of reused patterns. They have tasked themselves with finding these common specification patterns.

These property specification patterns consists of *patterns* holding in *scopes*. A pattern specifies what has to hold, and a scope specifies when that pattern has to hold. Imagine for example a property of a lighting system: "during presentation mode, the room luminaires remain on level 20", using a pattern library we can specify this property using a pattern ("the luminaires remain on level 20") holding in a scope ("during presentation mode").

In patterns and scopes, capital letters (P, Q, S, T) denote *propositions*, atomic observations of a system. These propositions can be categorized into two groups, event propositions are singular observations that only hold for infinitesimal time whereas state propositions hold for at least some time. Dwyer et al. define the pattern library for both types of propositions. To illustrate the difference, consider the following two propositions:

[1] We call it libraries, Autili et al. (2015) calls it a catalogue.

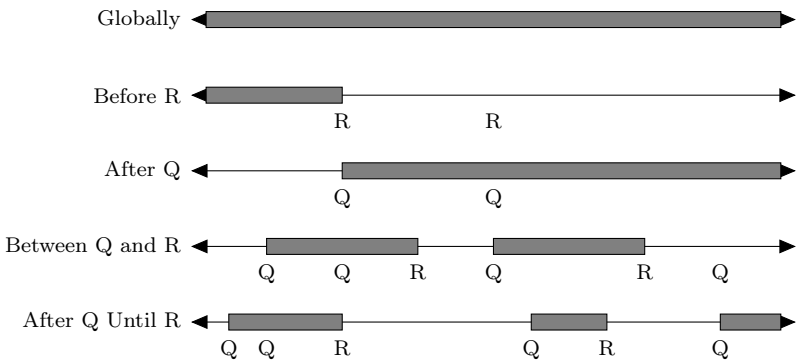


Figure 5.1: Example of scopes: bars outline when a scope is considered sensitive. Figure by Dwyer et al. (1998)

- P: The light turns red (event)
- Q: The light is red (state)

The scopes say something about when a pattern has to hold, Dwyer et al. have identified five different scopes:

Globally The pattern has to hold always.

Before R The pattern has to hold up to a state where R holds/has occurred. If R never holds/occurs, the scope is not considered sensitive and the pattern holds (trivially).

After Q The pattern has to hold from a given state where Q holds/has occurred.

Between Q and R The pattern has to hold between the state where Q starts holding/has occurred and states where R holds/has occurred. If Q holds/occurred at some point, but no R holding/occurring follows, the scope is not sensitive and the corresponding pattern holds trivially.

After Q until R The pattern has to hold after the state where Q holds/has occurred until a state where R holds/has occurred. If Q holds/occurred at some point, but no R holding/occurring follows, the scope is sensitive indefinitely and the pattern has to hold forever.

The intuition of a scope being sensitive can be seen in Figure 5.1, the shaded boxes indicate when a pattern has to hold. Patterns then, specify what has to hold in such a scope and are divided into two groups, occurrence patterns:

Absence P P does not hold/occur in the scope.

Universality P P holds in the entire scope or is never refuted.

Existence P P holds/occurs at some point in the scope.

k-**Bounded Existence P** P holds/occurs at most k times in the scope.

And Order patterns:

Precedence P, S P holding/occurring has to be preceded by S holding/occurring in the scope.

Response P, S P holding/occurring has to be followed by S holding/occurring in the scope.

Chain Precedence $\mathsf{P}_1, \ldots, \mathsf{P}_n$, $\mathsf{S}_1, \ldots, \mathsf{S}_n$ A sequence of $\mathrm{P}_1, \ldots, \mathrm{P}_n$ holding/occurring has to be preceded by a sequence of $\mathrm{S}_1, \ldots, \mathrm{S}_n$ holding/occurring in the scope.

Chain Response $\mathsf{P}_1, \ldots, \mathsf{P}_n$, $\mathsf{S}_1, \ldots, \mathsf{S}_n$ A sequence of $\mathrm{P}_1, \ldots, \mathrm{P}_n$ holding/occurring has to be followed by a sequence of $\mathrm{S}_1, \ldots, \mathrm{S}_n$ holding/occurring.

For example, a car should accelerate when we press the gas pedal but only when it has been started before. To capture this property we define our propositions:

$$\begin{aligned} \mathrm{Q} &= \textit{The engine is started} \\ \mathrm{R} &= \textit{The engine is stopped} \\ \mathrm{P} &= \textit{The gas pedal is pressed} \\ \mathrm{S} &= \textit{The car accelerates} \end{aligned}$$

And combine this with a "Between Q, R"-scoped "Response P, S"-pattern to gain the system property specification: "Between {the engine is started} and {the engine is stopped} {the gas pedal is pressed} is responded by {the car accelerates}".

These pattern/scope combinations can be transformed into *temporal logic* by applying transformation rules found at on the website of the authors.[2] In case of the example, transforming this "Between Q, R"-scoped "Response P, S"-pattern to LTL yields:

$$\mathrm{G}((\mathrm{Q} \wedge \neg \mathrm{R} \wedge \mathrm{F}\,\mathrm{R}) \rightarrow (\mathrm{P} \rightarrow (\neg \mathrm{R}\ \mathrm{U}\ (\mathrm{S} \wedge \neg \mathrm{R})))\ \mathrm{U}\ \mathrm{R})$$

In an extension to their first paper (Dwyer et al. (1999)) have also conducted a survey of their identified patterns. For this survey, they have collected 555 property specifications from 35 different sources. These property specifications came from many different domains ranging from databases to communication protocols, from avionic systems to operating systems.[3] Of these 555 property specifications, 511 could be captured in any of the pattern/scope combinations. Interestingly, they have also found that the vast majority of properties can be captured using the "Globally"-scoped "Response P, S"-pattern combination.

[2] `http://patterns.projects.cs.ksu.edu/`

[3] For the interested reader, the entire set of evaluated property specifications can still be found on their website.

5.2 Timed Property Specification Libraries

The property specification library of Dwyer et al. does not actually reason about time, only over sequences. Two extensions to introduce timing were developed concurrently, Gruhn and Laue (2006) developed an extension to the library of Dwyer et al. by mapping its patterns to timed automata whereas Konrad and Cheng (2005) extended the approach to timed temporal logics like TCTL and MTL.

5.2.1 Timed Temporal Logic

Konrad and Cheng (2005) proposed an extension to the pattern library as formulated by Dwyer et al. They designed patterns that map to timed temporal logics (MTL, TCTL, Real Time Graphic Interval Logic) using some 80 timed property specifications of nine different systems, and have validated their pattern library with several case studies.

What is interesting about their approach is that they separate temporal and timed temporal patterns hierarchically. They argue that timedness is of such different nature that it warrants its own hierarchy next to the one from Dwyer et al. Basically, they say that there is no similarity between a Response pattern in a temporal context, and a Response pattern in a timed temporal context. They keep the scopes from Dwyer et al. (Globally, Before, etc.) and introduce five new timed patterns. These patterns fall into three groups, firstly Duration patterns:

Minimum Duration$_t$ P When P holds, it holds for at least t seconds.

Maximum Duration$_t$ P When P holds, it holds for at most t seconds.

Secondly, Periodic patterns:

Bounded Recurrence$_t$ P P holds/occurs at least once every t seconds.

And finally, Real-time order patterns:

Bounded Response$_t$ P, S When P holds/occurs, it followed by Q holding/occurring within t seconds.

Bounded Invariance$_t$ P, S When P holds/occurs, then Q holds for at least t seconds.

Also novel in their research is a *structured English grammar* which allows property specification in a restricted subset of the English language. Similar techniques were employed before for a specific timed temporal logic (a structured English grammar for CCTL by Flake et al. (2000)), but Konrad and Cheng employ it for a language that is mapped to multiple different formalisms (e.g. MTL, TCTL, etc.). Their structured grammar allows (systematic) production of property specifications like "Globally it is always the case that if {a message is received} holds, then {a reply is sent} holds after at most 10 time units.". A property specified using the globally-scoped bounded response

pattern with a bound of 10 time units. A more complete introduction to structured English grammars will be given in Section 5.6.

As a validation step, they have conducted a case study of an electronically controlled steering system. They have collected property specifications, constructed them using the patterns they have specified and showed that in this case, their approach could specify the vast majority of properties.

5.2.2 Timed Automata

Gruhn and Laue (2006) extended the approach by Dwyer et al. by selecting timed automata instead of the traditional temporal logics. They argue that the use of timed temporal logics is for the most part theoretical, which would hamper the usability of the timed library. Timed automata, they argue, have stronger support in tooling, for example in UPPAAL.

The approach by Gruhn and Laue constructs what they call *observers* that follow the system faithfully. These observers produce counterexamples when a property of the system is violated. How this observer works depends on whether the checked property is of a "Safety"-kind or of a "Lifeness"-kind. A property of the "Safety"-kind has a finite counterexample, if the property can be refuted, there is a finite path serving as a counter example. Consider a property specified using the "Absence P" pattern, if a P occurs, the property is violated.

A property of the "Lifeness"-kind, on the other hand, has an infinite counter example. Properties specified using the "Existence P" pattern are only violated if said P never occurs.

Gruhn and Laue created timed automata for each of their patterns, if a pattern is of the safety kind, a single location is marked to be erroneous. If there is a way to reach this error location, the property is violated. Consider for example the observer for the properties specified using the (globally-scoped) "Absence P" pattern displayed in Figure 5.2. If it is possible to reach the location that is marked with ERROR, then P has occurred and the property is not correct.

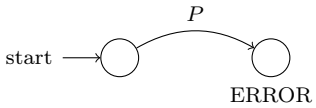


Figure 5.2: Absence pattern according to Gruhn and Laue.

When considering observers for liveness properties, their automata have a final state. If, for a run that takes infinitely long, it is possible to visit a final state infinitely often, then the property is violated. Consider the observer for properties specified using the (globally-scoped) "Existence P" pattern in Figure 5.3, if it is possible to stay in the initial location forever, then the property that P occurs at some point is violated.

Figure 5.3: Existence pattern according to Gruhn and Laue. Their notation is slightly different: the urgent keyword on the transition specifies that if such a transition is enabled, it has to be taken as soon as possible. The doubly lined initial state indicates that the property holds only if this state is not visited infinitely often.

Furthermore, they provide time-bounded patterns for their approach, for example (globally-scoped) "Time-bounded existence" pattern states that P has to occur within k time units and is shown in Figure 5.4.

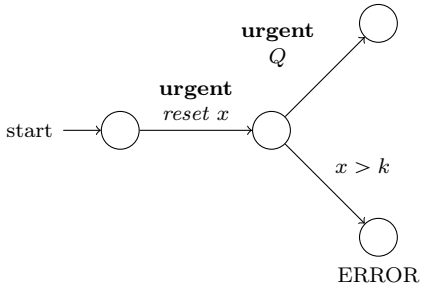


Figure 5.4: Time-bounded Existence pattern according to Gruhn and Laue.

This approach also defines that it is possible to create a scoped version of a pattern using some rules, and demonstrates the rule for "Before a certain event" and "Before a certain event with some timed offset". They have omitted these rules for all other scopes because of space constraints, but argue that similar rules exist for the other scopes.

Compared to the approach by Konrad and Cheng, they have not introduced new patterns to the library but merely extended patterns to be timed. However, Gruhn and Laue have added timing offsets to the various scopes which were omitted by the work of Konrad and Cheng. While there is significant overlap, these separate approaches could be merged to exceed the expressiveness of both individual extensions to the work of Dwyer et al..

5.3 Refining Timing Property Specification Patterns

Bellini et al. (2009) have argued that the split that was made by Konrad and Cheng can be removed. Their contribution refactors the (amended) library by Konrad and Cheng to simplify and generalize the pattern system.

First of all, Bellini et al. (2009) argue that *Chain*-patterns, as specified by Dwyer et al., are redundant. They argue that a chain of events (e.g. all propositions $\mathrm{P}_1, \ldots, \mathrm{P}_n$ hold

in sequence) is same as specifying a singular proposition Q that says that all events hold in sequence. In Section 5.7 a formalization of these *composite propositions* will be given.

Secondly, they argue that what Konrad and Cheng call Bounded Response is actually a generalization of the Response pattern by Dwyer et al., Response is equal in semantics as Bounded Response with bounds $[0, \infty)$. Furthermore, they added a lower bound to their version of Bounded Response so that properties like "a response will happen for example between 5 and 15 time units" can be specified. Finally, the dual of Bounded Response, Bounded Precedence was added that can specify the same as Precedence from Dwyer et al. but for timed properties.

5.4 Introducing Probability

Grunske (2008) has extended the pattern library for use in *probabilistic logic*. Although not directly relevant to the research that is proposed here, it is interesting for the evolution process of patterns.

The idea of a probability logic formula is to specify that a property holds with a certain probability. For example, when modeling a system of retransmissions of messages, we may be interested in saying that it will require at most four retransmissions to have a 95% probability that a message is received.

These types of properties are specified for *probabilistic models* which have a certain probability distribution on their transitions in such a way that the outgoing transitions of a state have a combined probability of 1 total. These properties are often specified in Probabilistic Computation tree logic (PCTL) by Hansson and Jonsson (1994) or its "star" (PCTL*) equivalent by Aziz et al. (1995).

Grunske redefines the various patterns defined by Dwyer et al. with probabilistic bounds, but only for the global scope. Patterns include Probabilistic Invariance S ("S holds for the duration of the scope with a probability p"), Probabilistic Response P, S ("when P holds, it is with a probability of p, reacted to by S holding within a time bound"). Grunske argues that the vast majority of scopes is the global one, and therefore omits the other scopes.

Grunske also defined a structured English grammar that allows property writers to use a more natural input language. An example property specified using their grammar could be: "The system shall have a behavior where with a probability lower than 0.9 it is the case that if event holds, then as a response reaction becomes true.". An instance of the "Probabilistic Response" pattern with a probably bound of < 0.9.

5.5 Unifying the Approach

Autili et al. (2015) have unified all previous extensions into a single pattern library that supports specifying temporal, timed and probabilistic properties. To do so they have conducted gap analysis of the three pattern libraries and added patterns where missing, redefined patterns slightly to be more general and added both inter- and intra-catalog

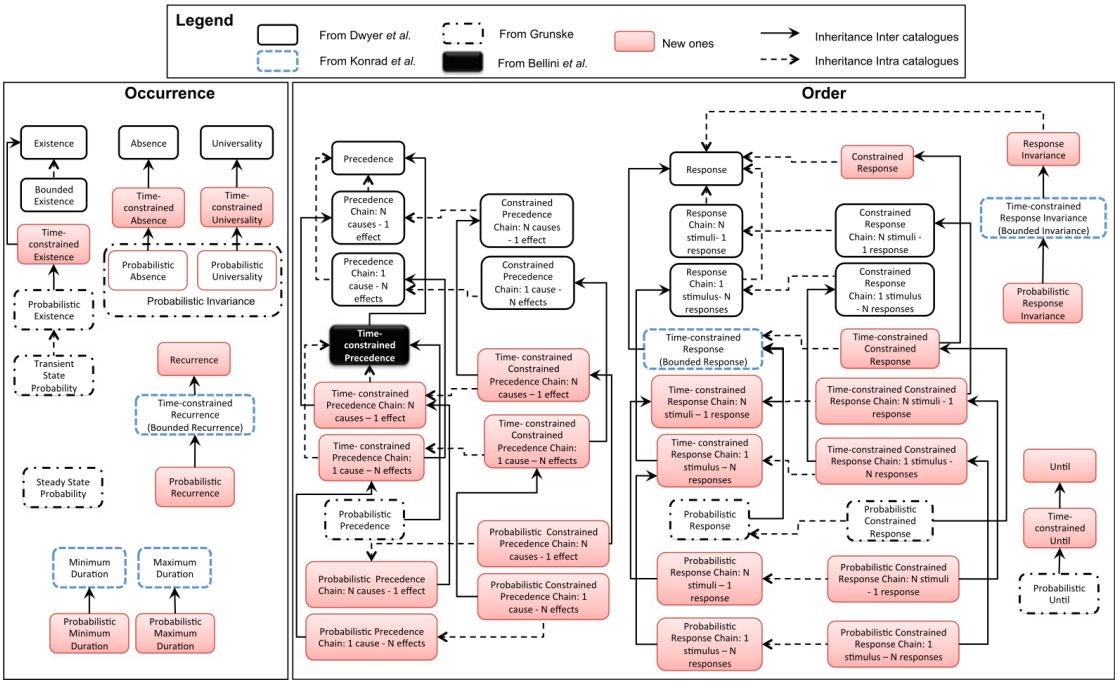


Figure 5.5: The complete pattern library by Autili et al.

relations. Their full approach identified a total of 58 patterns: 18 temporal, 19 timed and 21 probabilistic patterns.

Most of their additions are straightforward, for example, they have identified "Time constrained Absence", which is nothing more than the "Absence" pattern from Dwyer et al. but extended to have a time bound. For verification purposes, they have collected actual industrial property specifications and have found instances of all patterns they have identified. They argue that, if an instance of a pattern can be found at least once in their case study, it is a valuable addition to such a pattern library. Their entire catalog can be seen in Figure 5.5.

5.6 Structured English Grammar

Much like Konrad and Cheng and Grunske, Autili et al. have also created a structured English grammar definition for their complete pattern library. While the entire grammar definition can be found in the paper by Autili et al., it may be useful to evaluate an example property specification. Consider, for example, the production: "Globally it is always the case that {P} holds between 5 and 15 minutes with a probability greater then 0.5". This property specification is an instance of the Globally scoped "Probabilistic Universality" pattern. Although Autili et al. have not conducted any validation on whether the grammar is easily understandable to domain experts, the initial comments from these experts were positive.

We however, observe downsides of this structured English grammar approach. The more complex these property specifications become, the larger their verbosity. To read a property specified as a sentence, a reader has to parse the sentence, remove all words to make the sentence grammatically correct and reduce it to the pattern/proposition/scopes. We think that pursuing a structured English grammar approach is a trade-off. Initially, such an approach allows non-formal schooled engineers to try writing property specifications, but on the long run, their verbosity might reduce overview of the property specification catalog as a whole. When implementing such an approach we should remain critical of the target audience of the property specification language.

5.7 Introducing Composite Propositions

We have seen that various extensions of the pattern library by Dwyer et al. have concluded that "Chain Response" is superfluous, one could make a proposition Q that holds when the chain starts holding. Mondragon et al. (2002) were the first to formalize these (temporal) compositions of propositions and have identified four different compositions. Unlike the previously discussed pattern libraries, Mondragon et al. have specified the meaning of these composite propositions separately for singular events and state propositions (referred to as conditions by Mondragon et al.). For state propositions the four patterns are defined as:[4]

AtLeastOne$_c$ $\mathsf{P}_1, \ldots, \mathsf{P}_n$ is true in a state s if any of its propositions $\mathrm{P}_1, \ldots, \mathrm{P}_n$ hold in s.

Parallel$_c$ $\mathsf{P}_1, \ldots, \mathsf{P}_n$ is true in a state if all of its propositions $\mathrm{P}_1, \ldots, \mathrm{P}_n$ hold in s.

Consecutive$_c$ $\mathsf{P}_1, \ldots, \mathsf{P}_n$ is true in a state s_i if P_1 holds in s_i, P_2 holds in s_{i+1}, ..., P_n holds in s_{i+n-1}.

Eventual$_c$ $\mathsf{P}_1, \ldots, \mathsf{P}_n$ holds in a state s_i if P_1 holds in s_i, P_2 holds in s_{i+1} with $s_{i+1} > s_i$, ..., P_n holds in s_{i+n-1} with $s_{i+n-1} > s_{i+n-2}$.

In case of singular events, these patterns say that the truth of a proposition changed between two consecutive states. The composite patterns for these events are therefore:[4]

AtLeastOne$_e$ $\mathsf{P}_1, \ldots, \mathsf{P}_n$ is true in a state if all propositions are false in that state, and any of the propositions start holding in a future state.

Parallel$_e$ $\mathsf{P}_1, \ldots, \mathsf{P}_n$ is true in a state s if all propositions $\mathrm{P}_1, \ldots, \mathrm{P}_n$ are false in that state, and all of the propositions start holding in a future state.

Consecutive$_e$ $\mathsf{P}_1, \ldots, \mathsf{P}_n$ is true in a state if all of the propositions $\mathrm{P}_1, \ldots, \mathrm{P}_n$ are false in that state, P_1 becomes true a future state s_i, P_2 becomes true in state s_{i+1} etc.

[4]Definitions based on those by Salamah et al. (2012), latter two (mostly) verbatim.

Eventual$_e$ $\mathbf{P}_1, \ldots, \mathbf{P}_n$ is true in a state if all of the propositions $\mathrm{P}_1, \ldots, \mathrm{P}_n$ are false in that state, P_1 becomes true in a future state $s_i > s$, P_2 becomes true in state $s_j > s_i$ etc.

Mondragon et al. provide mappings to Future Interval Logic (or its graphic equivalent: Graphic Interval Logic) by Ramakrishna et al. (1992).

In an attempt to combine these composite propositions with the patterns by Dwyer et al., Salamah et al. (2012) have identified that directly substituting these composite propositions into the construct as firstly explored by Dwyer et al. creates unintended property specifications.

Consider for example a Globally-Response property specification with as first argument the composite proposition Eventual$_c$ and a normal proposition as the second argument. Eventual$_c$ becomes true in a state for which the first proposition holds of that composite pattern holds, and the following propositions will hold eventually. However, it might very well be that the normal "response" proposition is true before all propositions in the Eventual$_c$ have become true. Our intentions, however, were to write that the normal response came after all sub propositions in the Eventual$_c$ construct (eventually) became true!

To remedy this, Salamah et al. use a set of new operators (which is only syntactic sugar) that deeply embeds the response into the composite propositions. The research by Salamah et al. allows for specifying properties using patterns of Dwyer et al. combined with composite propositions by Mondragon et al. that translate to LTL without losing intent.

As a final remark, these composite positions have been specified for temporal logic without time/probability. Wanting to say that "Eventual", $\mathrm{P}_1 \ldots \mathrm{P}_n$ holds within 10 seconds would require an extension to the work by Mondragon et al.

6 Domain Specific Languages, Language Construction and Xtext

6.1 Domain Specific Languages

We have briefly mentioned *domain specific languages* (DSL) in for example Section 1.3 but have yet to properly discuss their place within computer science. Consider for example the definition used by van Deursen et al. (2000):

> A *domain-specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

Key to domain specific languages, as opposed to general purpose languages, is that they are focused on a particular problem domain. By being specific to this particular domain, they allow *domain experts*, people with extensive knowledge of that particular problem domain, to write specifications or programs without the need for a full knowledge of a general purpose language. Domain specific languages are abstractions over fundamental software development and can, therefore, be understood by a wider range of stakeholders.

Domain specific languages are widely employed in practice for very distinct purposes. To compare various DSLs, Brambilla et al. (2012) classify domain specific languages according to various dimensions:

Focus Domain specific languages can be *horizontal* or *vertical*. A vertical DSL focuses on abstracting over a particular singular (industrial, business, etc.) domain. A horizontal DSL focuses on abstracting over a technical problem encountered in many domains.

Style Domain specific languages can be *declarative* or *imperative*. A declarative DSL describes the "what" of a particular problem. An imperative DSL describes the "how" of a particular problem.

Notation Domain specific languages can be *graphical* or *textual*. A graphical DSL specifies instances using graphical models: diagrams, drawing etc. A textual DSL specifies instances using a textual form: custom syntax, JSON, XML, etc.

Internality Domain specific languages can be *external* or *internal*. An external DSL is a separate artifact in development with a full compilation process. Internal DSLs are

deeply embedded into a host language, as an API, as embedded domain specific constructs, etc.

Execution Domain specific languages can be *compiled* or *interpreted*. An interpreted DSL is evaluated step by step at runtime, a compiled DSL is first transformed to a new format and eventually executed.

6.1.1 Examples

To demonstrate the concept of a DSL and to demonstrate the classification, let's consider some examples of domain specific languages. Be reminded, however, that what is considered a DSL and what is considered not a DSL is subject to debate. We use the definition and classification above and argue that all examples given are indeed DSLs.

Structured Query Language

The structured query language (SQL) by Chamberlin and Boyce (1974) is a language that allows searching, adding, deleting and otherwise modifying relational data. SQL abstracts away from the notion of inserting or finding in the data structures that underpin a modern relational database. SQL allows a data analyst to retrieve data from a complex storage solution without the need for generic programming or fast data structure knowledge. SQL allows experts in the *relational data domain* to efficiently operate on data without knowing what happens below the surface. SQL is a typical horizontal DSL employed in many domains working with relational data. The language is declarative: specify what data to retrieve instead of how to retrieve data. It is textual, SQL queries are written as text. It is sometimes represented as an external DSL, as in strings in the host language, but also often implemented as an internal DSL, for example as `QuerySet`'s discussed next or projects such as LINQ by Meijer et al. (2006). Finally, SQL is an interpreted language that is executed by a relational database management system (RDBMS).

QuerySet

SQL can be embedded into a host programming language as an internal DSL. For example in the popular Python web framework Django[1], relational databases are represented by models: a model in Django is a (set of) tables in a database representing a particular type of data (for example, a user record). Operations on this model are done on a so called `QuerySet`, retrieving all users that have a first name of "Lennart", `User.objects.filter(first_name='Lennart')` results in a `QuerySet` with all users matching the constraint. These `QuerySet`s serve as a bridge between the database and the general purpose language of Python, operations on `QuerySet`s are executed as SQL queries on the database but represented as function calls in Python. We say that these `QuerySet`s offer an embedding of SQL in the host language of Python. Furthermore, the `QuerySet` DSL of Django also offers abstraction, various different databases are supported by Django and all particularities of these databases are captured by the translations to SQL internally generated by Django.

[1] `https://www.djangoproject.com/`

LaTeX

LaTeX (stylized as LaTeX) by Lamport (1994) is a domain specific language for markup of documents. LaTeX is a declarative DSL: a LaTeX file, for example, describes that a group of words requires *emphasis* without specifying how to apply emphasis. LaTeX is a textual DSL, the semantics of constructs in the text is recorded using a textual syntax. LaTeX is an external and compiled DSL, documents are specified in `.tex` files and compiled to document formats such as the portable document format (PDF).

GROOVE

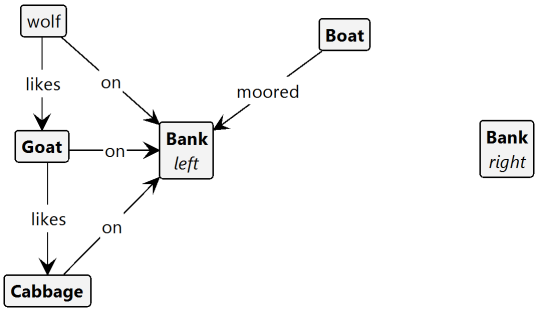
GROOVE, GRaphbased Object-Oriented VErification by Rensink (2003) is a graphical tool for software verification. In GROOVE, the state of the model is represented as a (directed) graph and transitions are represented by rules matching particular sub-graphs. We can classify GROOVE's input language as a graphical language, rules are specified using a visual syntax of colored arrows and boxes. We can also classify GROOVE's input language as a horizontal DSL as it abstracts away from the state space exploration of a particular program by defining it as an initial state and list of rules (consecutively) applied on on that initial state. Consider for example the classical cabbage-goat-wolf problem, a GROOVE model for this problem is distributed by GROOVE's authors. The initial state of such a model is given in Figure 6.1a, a rule that moves the ferry man across the river is depicted in Figure 6.1b.

Xtend

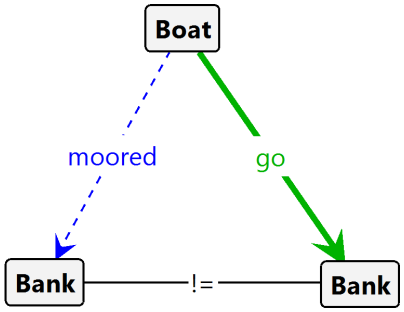
Xtend by xte is a dialect of Java with a strong focus on tasks common to language construction. The prime additions for language engineering are *template expressions* and *multiple dispatch*. Template expressions are String literals that allow expansion of embedded Xtend code: all code between guillemets (« and ») evaluated and included in the String literal. Xtend's dispatch methods guarantee that the most specific overloading of a method gets called for a particular parameter object. If for example there is a list of `Expression` objects on which to do some operation, the dynamic dispatch of Xtend will for each element of the list determine what is the most specific function to call (for example the one that accepts only `AdditionExpression`s). Xtend is a horizontal, imperative, textual, external and compiled DSL.

6.1.2 Prisma DSLs

The Prisma project's domain specific languages (as seen in Chapter 2) are vertical. Prisma DSLs are typically only applicable in the lighting domain. For instance, a building instance specifies relevant characteristics of a building only for lighting systems (sensors/actuators etc.). They are declarative, specifying that a particular sensor has fired, but not necessarily why. They are textual: specifications are written in a customly defined syntax per domain specific language. They are external and compiled to configurations for simulation tools or physical lighting systems.



(a) Initial state of cabbage-goat-wolf problem as represented in GROOVE's graphical *graph* DSL.



(b) Rule moving the ferryman across the river as represented in GROOVE's graphical *rule* DSL.

Figure 6.1: Examples of GROOVE's graphical DSLs

6.1.3 Classification

	SQL	`QuerySets`	LaTeX	GROOVE	Xtend	Prisma
Focus	Hor.	Hor.	Hor.	Hor.	Hor.	Vert.
Style	Decl.	Decl.	Decl.	Decl.	Imp.	Decl.
Notation	Text.	Text.	Text.	Graph.	Text.	Text.
Internality	Ext.	Int.	Ext.	Ext.	Ext.	Ext.
Execution	Interp.	Comp.	Comp.	Interp.	Comp.	Comp.

6.2 Traditional Compiler Construction

Before discussing tooling to implement domain specific languages, let us consider the traditional process of creating a (general purpose) language. Traditionally, compilers use what is known as a *multi-pass compilation* in which every layer of the compiler is responsible for certain guarantees to the layer below. Often, as outlined in Watt and Brown (2000), the steps of compilation are:

Lexing is the process of turning a stream of input (usually characters from a file) to a list of tokens. Lexing classifies and normalizes input, variable names are classified as identifier tokens regardless of what name they exactly represent and the input is stripped of superfluous whitespace and comments as they are not used in future steps of compilation.

Parsing converts the stream of tokens to an internal representation of the structure of a program. The parser will apply matching rules against the token stream until it consumed all input. Usually, the representation of the program is some form of *abstract syntax tree* (AST) which retains certain details but encodes other in the tree structure. Commonly when parsing basic expressions (e.g. $5 + 6 \times 4$), the parser will encode this expression in such a way that the precedence rules are kept,

so the AST will model that the calculation of 6×5 will happen first. The most important job of a parser is to verify that the input is syntactically well formed, for example, the expression $5 +$ can be defined as not well formed because of the lack of a second operand. What is well formed is defined by the parser's grammar, if the input cannot be matched against the rules of the grammar it is rejected.

Semantical Checking is usually employed to check certain *static semantics* of programs. For example, when adding two values $(a + b)$ it is usually the checker that asserts that both sides of the equation are of a correct type (e.g. you can only add numbers, not objects). What is considered to be correct is defined by the checker, for example Java's checker accepts "adding" strings and numbers and casts numeric operands accordingly.

Checkers usually walk the AST that is generated in the parsing step. It is also possible to have a multi-stage checking phase, for example. Java allows functions in classes to call other functions that are defined further onwards in the source file, this requires the compiler to first identify all functions and then check the usage of those functions in a second pass.

Code Generation generates some machine readable code. All steps before the compiler ensure that the program is sane, it is syntactically and (to some degree) semantically sensible. The compiler then generates some sort of machine readable format. It is not uncommon for a compiler to generate a new intermediate representation, for example, Java compiles its source code to *JVM Byte Code*, an intermediate format that is low level, but platform independent.

Interpreting is the process of directly executing the specified program, for example when writing a *script*, the desired output of invoking the compiler is to execute some code. Examples of interpreted languages include Python, Javascript, Ruby etc.

Throughout the years, many tools to aid in traditional compiler construction have emerged. A popular tool is ANTLR by Parr and Quong (1995), supporting all steps of compilation from lexing to code generation and widely adopted in both industry and academia. Other well known tools are Yacc by Johnson (1986), Happy by Gill and Marlow (1995) and so on. Most of these tools operate on a context-free grammar defining the syntax of the language, which is in itself a domain specific language.

6.3 Language Workbenches

The tools described in the previous section are all used in the analysis and compilation of source programs. Current-day languages, however, such as Java, are supported by a wider range of tools. There are integrated development tools such as Eclipse[2], there are debuggers, code analysis tools, etc. Integrated development tools have become pillars of

[2] http://www.eclipse.org/

software development, by providing refactoring features, navigation of source code, auto-completion functions and direct feedback. Using these features, they increase the speed of software development. Current-day language engineering is not only about developing a compiler but increasingly about tools supporting development in the newly created language.

These observations have led to a shift in DSL development. On one hand, there are external DSLs that have a very restrictive syntax making them easier to learn for domain experts. On the other hand, there are internal DSLs that can use all tools developed for the general purpose (host) language. Consider for example a domain specific language specifying sequential application of filters on an input image. If such a domain specific language is developed as an external DSL, we cannot set a breakpoint between filters. If we were to implement such a language as an internal DSL (for example as an API), tools like Eclipse would help users tremendously.

Fowler (2005) saw these disconnects and proposed the creation of what he calls *language workbenches*. Pivotal in language workbenches is the inclusion of an IDE-like editor. For example, the feature model by Erdweg et al. (2013) classifies a tool as language workbench only if it besides offering notation and semantics for a DSL, it also includes an editor that aids both syntactically and semantically. Syntactically such a tool can syntax highlight (domain specific) code, display code outlines, support folding of code blocks, offer auto completion, compare code artifacts, etc. Semantically such a tool can then allow following references, auto completion based on semantics, refactoring, error marking, etc. Language workbenches, Fowler argues, can offer the benefits of external DSLs by separating domain knowledge from general purpose programs with the strong tool support that we came to expect from embedded DSLs. Whether to implement the image filter DSL as internal or external is now just a matter of correctly identifying the use cases.

Finally, Erdweg et al. compare ten language workbenches. They do so by evaluating features according to a feature model and implementing an example questionnaire DSL in all workbenches. The ten evaluated tools span all features included in the feature model and the proposed questionnaire language was expressible in all workbenches. They have also compared language workbench implementations of the questionnaire DSL with an implementation written using traditional compiler construction tools. They conclude that the use of language workbenches can speed up the development of domain specific languages: all implementations were shorter in lines of code (or equivalent measurements) while offering more supportive tooling for the questionnaire DSL.

6.4 Introducing the Xtext Language Workbench

The Prisma project uses Xtext (2017) as a language workbench for the various domain specific languages. Xtext is very powerful and supports almost all features described in the feature model of Erdweg et al. (2013) with an editor generated based on the popular Eclipse platform.

Xtext can in a sense be seen as a combination of traditional compiler tools and the

notion of language workbenches. Specifying a domain specific language in Xtext follows a similar process as traditional compiler construction. Xtext however also uses part of these specifications to customize the supporting editor while providing many hooks to override or extend this customization.

Development of a domain specific language in Xtext is often characterized as a three stage process: (1) write a grammar for the language, (2) customizing validation and (3) developing a code generator or interpreter.

6.4.1 Writing a Grammar

The syntax, and to a lesser extent the semantics, of a domain specific language in Xtext are defined in its grammar. From the grammar, both a lexer and a parser are generated. Rules for the lexer are denoted by the `terminal` keyword and match certain streams of input to so-called tokens. The other rules match these tokens to produce the abstract syntax tree of a DSL instance. To aid in the creation of these grammars, Xtext provides some inheritance features and corresponding libraries. These libraries range from a set of simple terminal rules (for integers, strings, comments etc.) to a more fully fledged expression language called XBase.

As its abstract syntax tree, Xtext uses models as defined in the Eclipse Modeling Framework.[3] Xtext automatically infers a *metamodel* in Ecore format from a grammar. This metamodel defines the structure of a language, for example modeling that a Building has Floors which have Rooms, etc. A parsed instance of a domain specific language is then an instance of this metamodel, for example, that the Building "OfficeBuilding" has floors "Floor1" and "Floor2" which contain six and seven rooms, respectively. EMF can then map nodes in the metamodel to Java classes, and the instance of this metamodel to instances of these Java classes. While similar to providing a custom-generated abstract syntax tree, this metamodeling approach allows Xtext to reuse large parts of EMF and in turn, also allows interoperability between Xtext and EMF.

6.4.2 Customizing Language Validation

Xtext aims to do as much automatic validation as possible but offers to customize much of this behavior. Xtext does automatic syntactic validation before producing an abstract syntax tree but also does semantic validation. Validation can be extended using the following methods:

Check methods Xtext provides a framework to execute small checks on parts of the AST. The functions in this class that are decorated with the `@Check` annotation are automatically called with the relevant subtree of an AST. These validation steps are stateless, for example, checking whether a variable was defined before it was accessed involves finding all previously defined variables and concluding that the currently evaluated variable is one of these.

[3] https://www.eclipse.org/modeling/emf/

Linking Xtext has built-in linking. It is possible to reference any element from any rule. For example, when modeling a BuildingComplex consisting of Buildings, Xtext can cross-link between parts of the AST that define buildings, and parts that define the encompassing building complex.

Scoping Xtext allows defining what is considered linkable in a given context. Variables in programming languages can typically only be used in expressions when they have been defined before. Top-level functions, however, can (in the case of Java) be called from any point in a class, whether they are defined before or after their reference. To determine whether linking is semantically correct, the Xtext linking provider will consult the scoping provider. This scoping provider yields a list of reachable declarations for the given point in the program and can be extended to support block scoping, namespacing, import and exports, etc.

Xtext argues that these verification hooks are sufficient to establish whether an instance of a DSL is semantically correct.

6.4.3 Code Generation and Interpreting

Xtext mainly focuses on code generation. Every project created with Xtext comes with a (stub) class that can be used to generate code. The `doGenerate` method of this class is automatically called by the framework when changes in the files are detected and all validation steps have passed. The implementation of this method is then responsible for correctly outputting generated code for the passed abstract syntax tree.

Interpreting is not directly implemented in Xtext but can be achieved by recursively traveling the abstract syntax tree, an interpreter can be created as a standalone program that parses/checks/interprets a particular domain language instance and produces output as a result. Integration of this interpreter into the generated editor is possible by the flexible nature of Eclipse, but not by default supported by Xtext. There is no API to map the "run" button of the generated language tools to the execution of an interpreter on the selected domain specific language instance.

Part III

Query Language Design

Part Outline

In this part of the report, we will outline the steps taken to create a Query language not only for the lighting domain but a framework for a Query language for any domain. This part is split in seven chapters:

Chapter 7 discusses the considerations that required reformulating the research questions shown in the precursor (research topics) report. The chapter lists the new research questions and outlines the new approach taken for Query language creation.

Chapter 8 outlines the generic base for the Query language and discusses the steps required to instantiate this generic base for a new domain.

Chapter 9 shows the instantiation of the generic Query language specifically for the lighting domain. This chapter lists the patterns, scopes, atomic propositions and composite operators that we have selected for lighting system property specification.

Chapter 10 shows the concrete syntax of the Query language for the lighting domain as well as its static semantics. Furthermore, it outlines how some of the syntactic sugar of the concrete syntax can be represented in just propositions and (instantiated) pattern/scope combinations.

Chapter 11 discusses the need for a new translation of pattern libraries to support both formal verification and simulation. We show a variation on the translation to timed automata by Gruhn and Laue (2006) in the form of *monitors* and *rules*, show an example translation of a pattern for all scopes and show possible semantical alterations of pattern/scope combinations that can be achieved using the extended translation.

Chapter 12 shows how the framework of monitors and rules is implemented in UPPAAL. Furthermore, it shows how propositions, both atomic and composite, are implemented in the UPPAAL model.

Chapter 13 shows how the framework of monitors and rules is implemented in the Co-simulation Framework. Furthermore, it shows how propositions, both atomic and composite, are implemented in the Co-simulation Framework.

7 Design Considerations

A Query language like ours is the product of countless hours of discussions, brainstorms and mental processing. As with any research, the initial plan of development was very different from the delivered product. The initial research questions, also outlined in the (predecessor) research topic report (Buit (2017)), were as follows:

Original Research Question Can we create a domain specific language for the lighting domain and approach by TNO-ESI in which various stakeholders can capture lighting properties that are formally verifiable and align with the technical knowledge of these stakeholders both in property specification and in conveying error?

Original Subquestion 1 Who is the intended user of the proposed Query language, what questions do these stakeholders have and can we group these questions in meaningful categories?

Original Subquestion 2 What types of formalisms do we need to cover the needs from various stakeholders, can we use known theories of property specification using pattern/scope combinations?

Original Subquestion 3 How can we map the set of queries with the chosen formalisms to an easily understood DSL that integrates well with the existing framework of DSLs, is useful to the stakeholders, and can be extended with ease?

These questions remain relevant to creation of a Query language specifically for the lighting domain, but proved infeasible to tackle in during the research period. This chapter outlines the considerations that required reformulating the research questions, lists the new research questions and ends with the new approach.

7.1 Consideration: The Value of Abstraction

In initial discussions about creation of a Query language with potential stakeholders of Philips Lighting we were confronted with a wide variety of viewpoints. We had prepared some example properties and openly discussed these with testers and architects of Philips Lighting. These example properties (implicitly) assumed a system wide view. This assumption came from the way a lighting system is designed using the Prisma methodology: a lighting system starts as a specification of its behavior, its environment and its mapping to the physical devices and is then "compiled" to configuration for hardware products (of Philips Lighting). This methodology aligns well with the new business model of Philips Lighting: selling "Lighting as a Service".

The view of Prisma is technology independent. Many potential stakeholders in the Query language are used to thinking in terms of appliances. This confronted us with a fundamental challenge: discussing system wide properties is not always trivial as not every stakeholder is used to abstract from the intricate details of physical appliances. This abstraction is crucial to the domain model of the Prisma project. The proposed domain modeling approach hides complexities of particular technologies and reduces lighting systems to their core: behavior of lighting systems and possible influence of its environment. Details of particular technologies are only introduced when required. A Query language can only reason about concepts known in the domain model and is therefore limited to the abstracted view. In these discussions with stakeholders we often had to discuss the domain approach and its abstractions first before discussing potential properties specified using the Query language. Obtaining valuable input for the Query language proved challenging as not all stakeholders in the lighting domain had already embraced the domain approach.

While we could have concluded that creation of the Query language should be deferred to when the domain modeling approach of the Prisma project has been adopted thoroughly by stakeholders of Philips Lighting we believe that the Query language is a strong argument for its adoption. One of the largest benefits of a domain approach is the ability to analyze a lighting system prior to implementation. The proposed Query language allows to record properties in a structural manner and is crucial in this early validation. Our focus shifted from creating a Query language specifically for Philips Lighting to the creation of a prototype to show the potential of early validation. Once the domain approach is well integrated within Philips Lighting, the prototype can be extended and tuned specifically for the lighting domain.

7.2 Consideration: The Need for Domain Models

We have no doubt that the Query language can land in an organization that has fully embraced the domain modeling approach. If current adoption of the domain approach created as part of the Prisma project is the only challenge facing the Query language, we can just created a prototype language that we feel covers the use cases. Validation of the completeness of this approach should then be deferred to when the domain approach is fully embraced. A language that remains open to change can easily be adapted as a result of the feedback raised in this delayed validation.

Creation of such a prototype assumes that we can capture all use cases in both a Query language and by extension the domain model. The current focus of the domain model, however, is behavioral. The iteration of the domain model on which this research was based can capture many behavioral specifications and a short term update currently being integrated increases this expressiveness to levels required by Philips Lighting. Other types of properties expressed by stakeholders exceed the expressiveness of the domain model, for example, a property that asserts the energy usage of a system does not exceed a certain value. The concept of energy usage is not captured and therefore the property cannot be expressed.

In other words, constructing a Query language that covers all possible use cases can only be done once the domain model is complete. For the Query language, the focus moved to extensibility, only if the prototype can be extended easily we can eventually produce a Query language complete all use cases of Philips Lighting.

7.3 New Research Questions

With these considerations came the need to re-evaluate the research questions, the new research questions are:

Research Question Can we, in the context of the domain approach of the Prisma project, create a framework and prototype of a domain specific Query language in which lighting properties can be specified and formally verified? This framework and prototype must align with the technical knowledge of the stakeholders in the lighting domain both in specifying properties and conveying verification results.

Subquestion 1 Are pattern libraries a solid foundation of an easy to use Query language?

Subquestion 2 Can a pattern library approach cover all use cases of a formal specification language for the lighting domain?

Subquestion 3 Does a pattern library approach allow easy extension when new use cases are discovered?

Subquestion 4 Can a pattern library approach be implemented in both UPPAAL and the Co-simulation Framework?

Subquestion 5 Can syntax and semantics be developed that allows intuitive specification of properties of lighting systems?

Subquestion 6 Can the Query language for the lighting domain be integrated within the approach of Prisma?

Subquestion 7 Can the Query language for the lighting domain trivially be extended to cover new use cases?

Subquestion 8 Can we show the value of a Query language for the lighting domain to relevant stakeholders within Philips Lighting?

Subquestion 9 Can we convey to a stakeholder that a property is not correct in a way that indicates what causes the faults without needing fundamental knowledge of temporal logic or model checking?

The questions remain largely the same but can be seen as precursors to the questions posed at the beginning of this chapter. By creating a generic Query language framework, we enable the creation of a Query language specifically for the lighting domain. The original research questions remain relevant, but can only be answered once the domain approach has been well established within the product organization of Philips Lighting.

7.4 New Approach

With a new set of research questions comes a new approach. We have focused less on making a Query language that is *complete* for the lighting domain and more on a Query language that is *extensible*. To do so, we have split the approach in two: a generic Query language usable in any domain and a specific Query language for the lighting domain. For the basis of this generic Query language we have chosen to pursue a pattern library approach for two reasons: pattern libraries are domain agnostic and they appear easier to use than full blown temporal logics. We feel that pattern libraries allow us to set up a framework for property specification easily while remaining open to extension. The exact concepts of this generic Query language can be found in Chapter 8.

The specific Query language, an instance of the generic Query language, serves as a prototype to demonstrate the use of a Query language for the lighting domain. For this specific Query language, we have made a selection of patterns and scopes, composite operators and propositions that can be found in Chapter 9. Furthermore, for this selection we have implemented easy to use syntax and semantics presented in Chapter 10.

The specific Query language for the lighting domain transforms to input for two tools: UPPAAL for formal verification, and the Co-simulation Framework for simulation. By supporting both simulation and formal verification we aim to show Philips Lighting that both techniques are feasible for validation of lighting systems. We have developed a new translation based on the work of Gruhn and Laue (2006) consisting of *monitors* and *rules*, its need and its implementation are outlined in Chapter 11. Finally, in Chapter 12 and Chapter 13 we will describe how these monitor/rule combinations translate to UPPAAL models and Co-simulation Framework configurations, respectively.

8 Generic Query language

The design of the Query language is split into two: a domain specific part and a part that is shared across domains. In this chapter, we will discuss the generic part that consists of *concepts* common to a Query language for any domain. The subsequent chapter will discuss how we used these concepts to make a Query language specifically for the lighting domain.

In this chapter, we discuss the abstract concepts of *propositions*, *compositions of propositions* and *properties*. The following chapter creates propositions, selects composite operators for these propositions and finally selects a subset of pattern/scope combinations specifically for property specification in the lighting domain.

8.1 Propositions

A property specification in the generic Query language consists of a pattern holding in a scope, that are both instantiated by propositions. This pattern/scope combination is domain agnostic, but propositions are not. A proposition then, is a truth value, a statement that is true or false. For example, for a lighting system, a proposition could be "The luminaire is on".

We call a proposition *atomic* when for a given domain, the proposition cannot be decomposed into smaller propositions. This atomicity of a proposition is domain specific, for example in the domain of lighting systems, a sensor event is non-decomposable, whereas, in the domain of just sensors, this sensor event might be decomposed in propositions about heat signatures, movement etc.

We make a distinction between two types of propositions:

State Propositions are propositions that are true for a non-infinitesimal interval. "It is raining" or "A light is on". At any given moment a state proposition can *hold* or *be refuted*.

Event Propositions are propositions that are true for an infinitesimal interval. "It started raining" or "A message arrived".

While being different, we can convert state propositions to a pair of event propositions, just take the period in which the state proposition holds and create two event propositions for when it *starts* and *stops* holding. However, converting event to state propositions is not always possible. A single button press, for example, is singular in nature and does not signal that some sort of observable state property holds for a while. Which (pairs) of event propositions can be converted to state propositions, and how this conversion works depends on the domain.

8.1.1 Proposition Selection per Domain

When selecting propositions to implement in a Query language for a specific domain there are various considerations to be made. First of all, the set of implemented propositions has to evolve together with the domain model for a particular domain: if the domain model is extended to cover a particular (new) aspect of that system (for example, energy usage), new propositions have to be added to write property specifications about that new aspect as well.

Secondly, propositions not only differ per domain, but within a domain a choice can be made to hide or expose particular propositions. In a *black-box* approach, the system is regarded as closed and only the externally visible behavior of the system is allowed to be used in property specifications. White-box (sometimes called transparent-box) approaches, however, also allow to use internal details in property specifications.

Both white- and black-box approaches have benefits and drawbacks. While white-box approaches are often easier to use for system property specification, any of these property specifications is stained with knowledge of the internals of a system. This knowledge could hide assumptions that in a black-box approach have to be specified explicitly. Black-box testing, however, can be cumbersome because of limited externally visible behavior. Whether to pursue a black- or white-box approach is a choice that has to be made when instantiating the generic Query language.

8.2 Composite Propositions

We can combine atomic propositions to create *composite* propositions. These expressions use certain operators with semantical meaning to transform propositions to new propositions. For example, in propositional logic, the AND operator combines two propositions p_1 and p_2 in such a way that the combined proposition holds when both p_1 and p_2 hold.

Some compositions are not defined for all types of propositions. For example, how would we define that two event propositions hold at exactly the same time? We could alter the semantics of the AND operator to fire when two event propositions happened sufficiently close after each other, but that would not be semantically the same. If, however, the domain was discrete with events able to occur truly simultaneously, we can define the AND operators for event propositions in that domain. Choosing which composite operators to implement is domain dependent.

8.3 Properties

Propositions, both atomic and composite, are impartial observations. That a luminaire is on is not necessarily good nor bad. A property however reasons about the order of these impartial observations and marks some of these orders as good or bad. If a property, for example, says that the observation "the luminaire is on" has to occur, all system executions in which the luminaire is never on are wrong.

For the generic Query language, we have chosen to pursue a pattern library approach for writing property specifications. This choice is inherently a trade off between expressiveness and ease of use. While pattern libraries provide only a subset of (timed/probabilistic) temporal logic, the building blocks of such a pattern library are easy to understand. Which pattern/scopes of which formalisms (timed/probabilistic) are needed is a choice made per domain. For example, when a casino wants to guarantee that their games are very likely to be profitable, a probabilistic model and corresponding pattern/scopes have to be employed. Furthermore, some domains make heavier use of certain patterns, where a "Response"-pattern is very common in one domain, it might not be used in another.

8.4 Instantiating the Generic Query language

We argue, that for any domain, it is possible to create a Query language using the concepts described in this chapter by means of:

- Selection of a proper formalism, temporal, timed temporal, probabilistic, etc.
- Identification of atomic propositions within the domain
- Selection of composition operators for the atomic propositions
- Selection of patterns and scopes relevant to the domain

9 Query language for the Lighting Domain

In the previous chapter, we have identified a set of concepts shared among Query languages and ended with a list of choices to be made per domain. As the ultimate goal of this report is to provide a Query language specifically for the lighting domain, this chapter will focus on the choices made specifically for that domain.

9.1 Selection of Formalism

The first choice in implementing the generic Query language for the lighting domain is from which pattern library we need to start. In the example of the casino of Section 8.3, we have clearly seen that probabilistic reasoning is required and would, therefore, start with the library of Grunske (2008). For lighting, such a choice has still to be made, do we start with the temporal library by Dwyer et al. (1998), or with the timed temporal library by Konrad and Cheng (2005) etc.

In discussing lighting system requirements with Philips Lighting stakeholders, we have seen multiple categories of requirements. Philips Lighting is currently interested in energy efficiency, latency, scalability, reliably and robustness. These requirements often have to hold simultaneously, for example, a system that shuts down controllers to preserve energy but still has to respond within a certain time limit, or controllers that are stressed further by connecting as much luminaires/sensors as possible while also meeting latency deadlines.

Besides these non-functional requirements, TNO-ESI is interested in (correctness of) system behavior. Systems must correctly respond to stimuli, for example, "In an active mode, all luminaires in a room are at 80%". The domain model created in the Prisma project has a strong focus on capturing behavior, the three "core" languages of Building, Template and Control all define behavior and mapping of behavior exclusively.

The use of temporal and timed temporal formalisms is evident. Latency properties are timed, and behavioral properties are either temporal or timed temporal. Probabilistic reasoning, as Baier et al. (2008) note, is useful in modeling unreliable or unpredictable system behavior, but can initially also be modeled with non-determinism. Baier et al. argue that on a system level, non-determinism is often good enough. Furthermore, probabilistic reasoning requires a model of likeliness of certain events occurring, which is not available currently in the domain approach of Prisma. We would suggest keeping probabilistic reasoning out of the system level Query language designed for the lighting domain. In specific cases, such as reliability analysis of a certain wireless technology, we

suggest creating a dedicated detailed probabilistic model for that technology instead of enhancing the domain model solely for such a specific case.

9.2 Selection of Atomic Propositions

The second choice to be made in implementing a specific Query language for a particular domain is which atomic propositions to support. First of all, it is important to note that only propositions that are observable in a model can be used in a Query language. The domain model created as part of the Prisma project has been extensively discussed in Chapter 2 and we have implemented atomic propositions for facts from this domain model. For the models of the lighting system that are currently implemented, we have created the following selection of propositions:

Event Propositions

Sensor A specific sensor registers a trigger, a button press or an occupancy event.

Sensor Group A defined group of sensors registers a trigger, a button press or occupancy event.

Timeout A specified amount of time has expired since the timer has last been reset. A Timeout proposition is not strictly atomic since it can have an (possibly composite) reset expression, it does, however, act as a source of signals and shall, therefore, be regarded as an atomic proposition.

State Propositions

Actuator Level A specific luminaire has a specific output level.

Actuator Group Level A defined group of luminaires has a specific output level.

Controller State A specific controller is in a specific state.

9.2.1 White-box or Black-box

To determine whether the set of atomic propositions should be white or black-box requirement engineering was employed. Initially, the design of the Query language was black-box, but a case was discovered in which black-box reasoning did not suffice. Consider the following example of a property for a system with presentation mode: "In presentation mode, the screen lights are at 50% output". To specify such a property, a user has to define what it means to be in presentation mode first, which is not externally observable. This information could be derived from the embedded state machine of the implemented template. A user could, for example, write some sort of regular expression that matches all inputs that lead to presentation mode, but have yet to leave it. However, these regular expressions are very complex and moreover, if we would impose a true black box view onto the user they would not be allowed to derive these regular expressions from the state machine. Imposing a true black-box view would be unrealistic.

On the other hand, if the user was allowed to observe the current state of a controller and can specify that the system is in presentation mode by observing that state, specifying this property is trivial. However, this specification now assumes that the entry and exit conditions of the presentation mode are specified correctly in the system under tests. Checking these entry and exit conditions is also easier when allowed to observe internal state: instead of having to assert that a certain input leads to observable behavior corresponding to the presentation mode state a property specification can now assert directly that the controller is in that correct state.

The final design is white-box, allowing to observe (internal) states of controllers. In behavior specifications, it is so common to make claims about a system in a particular state that forcing a strict black-box view to its users would hamper ease of use too much just to force some extra preciseness of specification. We do stress, however, that specifying that a particular controller is in a particular state is a shortcut in property specification and comes with the assumption that the presentation state is correctly entered and left.

9.3 Selection of Composite Operators

Besides atomic propositions, we have implemented the following composite operators:

Boolean AND for state propositions The combined proposition holds only if all propositions in the expression hold simultaneously.

Boolean OR for both state and event propositions The combined proposition holds/-fires if any of the propositions in the expression hold/fire.

Boolean NOT for state propositions The combined proposition holds only if its operand does not hold.

Casting state propositions to event propositions The combined proposition fires when a state proposition *starts* or *stops* holding. Recall the discussion that state propositions can be translated into a pair of event propositions, this composite operator can be used to observe the start or end of the period in which a state proposition holds.

9.4 Selection of Patterns/Scopes for the Lighting Domain

The selection we have made serves two goals. Not only does the pattern selection serve as a solid basis of a Query language for the lighting domain, these patterns explore most features that a pattern library could have. We can specify liveness properties, "Eventually a luminaire will turn on", safety properties, "In an active mode, luminaires can never turn off", we can specify timing properties, etc. As a basis, we have selected all patterns of Dwyer et al. and extended that with patterns of the other pattern libraries. For example, the Invariance pattern was added because we discovered a use for such a pattern.

We have implemented from the works of Dwyer et al. and Konrad and Cheng the following seven patterns:

- Order patterns
  - Response
  - Precedence
- Occurrence patterns
  - Existence
  - Absence
  - Universality
- Timed patterns
  - Recurrence
  - Invariance[1]

We have implemented all these patterns for all scopes that are presented in the pattern library approaches. We have discovered a sixth scope, a dual to Before, but have yet to find use for it. The sixth scope will be shown in Section 16.1. The scopes that we do have implemented are:

- Globally
- Before
- After
- After Until
- Between

9.5 Using the Query language

Given an informal requirement to the lighting system, the task of a property specification author is to translate this requirement into a (set of) property specifications. Much like writing the informal requirement, translating that requirement to a (set of) property specifications requires creativity. Since a single property specification, in any case, consists of a scope and a pattern, it is helpful to think *when* such a property must hold, and *what* holds during that period.

For scopes, there are five choices that all serve a different type of property. Of these five, there is a pair of scopes that are similar, restricting the scoping choice to four:

Globally can be used for patterns that have to hold regardless of the state of the system. Often used to model behavior that has to occur, or cannot occur ever.

[1] Currently only globally scoped.

After can be used for patterns that have to hold after a certain sequence of events. For example, specifying that a pattern has to hold after initialization.

Before can be used for patterns that have to hold up to a certain point, for example before the system is initialized.

After/Until or Between can be used for patterns that have to hold from each occurrence of a particular event to another. We say that the scope is *sensitive* in this period, but do not restrict how long this period is in time. The small difference between the two is that a property specified using After/Until has to hold if the scope is never closed, while property specified using Between make no claims if the scope is never closed. These scopes are often used to state that a pattern has to hold "in presentation mode".

Selecting an appropriate pattern is often easier as they only specify what has to hold within a scope. These building blocks say that something cannot occur, has to occur, has to hold for some time etc. The art of specifying using the Query language is to correctly instantiate these patterns and scopes with (atomic/composite) propositions and arguing whether a property is sufficient to model a requirement.

9.5.1 Example

As a running example throughout the remainder of this Design part and to show the thought process of writing a single property specification, we will specify (for the system of Chapter 2) the following informal requirement: "Leaving presentation mode can only happen after the user presses a button, or all sensors of the controller have reported vacancy for at least the hold time.".

For such a property to hold, the system has to be in presentation mode first and then, before leaving, either the room has to be vacant for at least the hold time, or a button press has to have occurred. The choice of pattern is clear, either a button press or a timeout has to occur during presentation mode: the Existence pattern. Because such a property has to be satisfied each time the system moves to presentation mode, the choice for a scope is either "Between" or "After/Until". In case that the system remains in presentation mode indefinitely, no button press or timeout can occur. In other words, only in scopes that eventually close the existence of either a button press or timeout has to be checked: the Between scope.

Now that a choice of pattern and scope is made, we instantiate these pattern and scopes by filling in the "holes" with propositions. We define a ControllerState proposition to track when the system is in present mode, a SensorGroupProposition (for buttons) to track when a button of the button group has detected an event and a SensorGroupProposition (for occupancy sensors) to track when occupancy was detected.

The final proposition is a Timeout that resets every time occupancy was detected or when persentation mode is engaged. The model of this system implicitly assumes that entering presentation mode is equal to being detected by a sensor, and therefore the Timeout is reset when presentation mode is engaged. The conceptual structure of this

property specification is given in Figure 9.1, green boxes represent atomic propositions, orange boxes represent composite operators and the yellow boxes are either patterns or scopes.

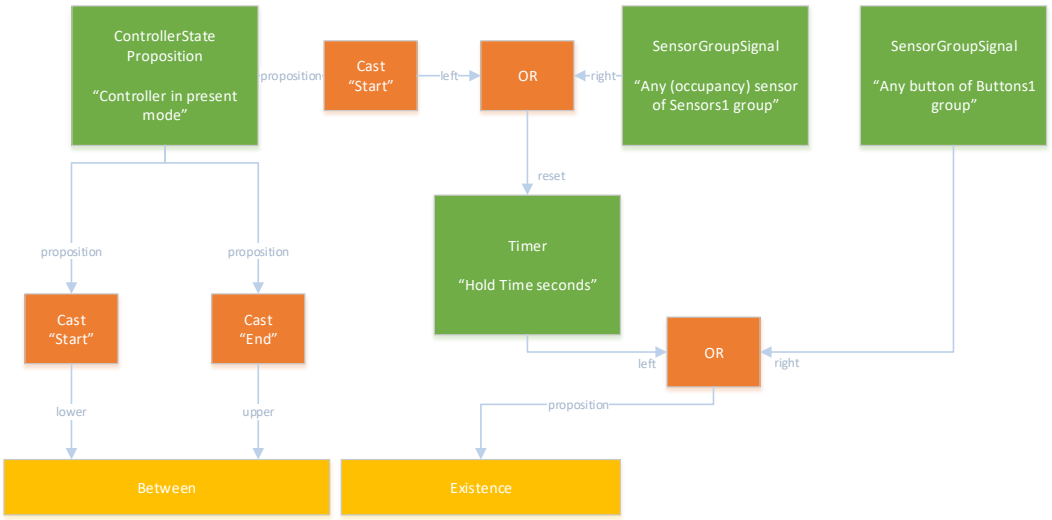


Figure 9.1: Property specified in terms of atomic propositions, composite operators and pattern/scope combinations.

10 Query Language Design

In traditional compiler design, a program is lexed, parsed, linked, checked and code generated. Typically, the process of lexing, parsing, linking and checking is considered the "Front end" of compilation, while code generation (or interpreting) is considered the "Back end". The full compilation process for the Query language is shown in Figure 10.1.

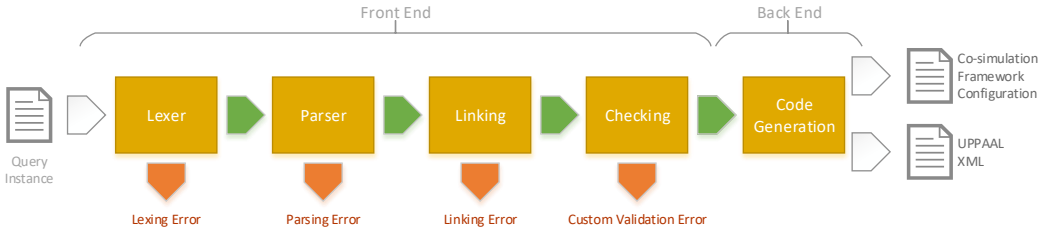


Figure 10.1: Front and back end of compilation

In this process, an input file is gradually transformed from text to input for both UPPAAL and the Co-simulation Framework. A user writes language instances in what is known as a concrete syntax: a textual representation of the property (s)he wishes to specify. The lexer splits this input into *tokens* that the parser matches to its *grammar rules*. If the parser can match all input to its rules, it produces an abstract representation of the language instance for further steps. This representation is an instance of what is known as a metamodel, a model of all possible Query language instances. This metamodel is given in Appendix A. Instances of this metamodel are further checked in a process called semantic analysis before code is generated.

This chapter is divided into three parts: the first is a discussion of the concrete syntax of Query language instance, the second a discussion of the static semantics that are checked in the linker and checker and the last discusses how a Query language instance is reduced to just propositions and (instantiated) patterns/scopes. Chapter 12 and Chapter 13, outline how these reduced instances are transformed to configurations for UPPAAL and the Co-simulation Framework respectively.

10.1 Style of the Concrete Syntax

The form of a program is important when conveying the message of a Query language. While syntax and semantics are often afterthoughts in language implementation, they

define the first impression of an intended user. If the syntax is too mathematical for example, users might think that the Query language is only useful to people with extensive mathematical background. By providing an intuitive syntax we can prevent discussions about form and instead focus on the benefit of the proposed language.

Many pattern library authors propose to write property specifications using a structured English grammar as outlined in Section 5.6. In such an approach, property specifications are (valid) English sentences that can be read just as any other sentence. However, a natural language such as English is verbose. A complex property specification combining many propositions becomes even more complex by requiring such a specification to be written as a correct English sentence. We feel that structured English grammars do not scale to larger or more detailed property specification languages.

Furthermore, most of the target audience of the Query language has a technical background. Testers, engineers, system architects and analysts have seen some form of programming languages, be it Matlab, or Java or C etc. We can achieve the same level of familiarity as with a structured English grammar by designing the Query language like any of these programming languages. Pursuing such a syntax makes property specifications more concise and easier to compose.

Therefore, we have designed a language that is a combination of what is good in traditional programming languages and in a structured English grammar approach. We retain verbose specifications for propositions and pattern/scope combinations, while syntax that combines propositions, or defines the structure of the Query instance is much more like a traditional C-like programming language. Finally, we have opted to not require all property specifications to be well formed English sentences, instead choosing for readable keyword-like syntax that is more concise.

10.2 Entry Point, Integration and High Level Syntax

```
QueryModel:
  'Queries' name=ID
  statements+=Statement*
;
```

Any Query language instance begins with a query model declaration. This declaration has a name that is used in the Experiment language instance to bind a lighting system to a Query instance. A minimal Query language instance would consist of only the keyword "Queries" and a name, but more practical instances consist of a list of statements.

Consider, for example, the example property of Section 9.5.1. In concrete syntax, the specification for this property is given in Listing 10.1. This particular QueryModel instance has a single statement as child: a with statement. The with statement has a block statement as a child and this block statement has four proposition definition and a single query definition statement as its children.

```
Queries canLeavePresentMode

with each controller implementing MeetingRoom1Template as ctrl1 {
  let presentMode = ControllerState ctrl1::Presentation

  let anyButton = SensorGroupSignal ctrl1::Buttons1 Button
  let anyOccupancy = SensorGroupSignal ctrl1::Sensors1 Occupancy

  let timeout = Timer ctrl1::HoldTime
                  Reset {Start presentMode or anyOccupancy}

  query Between {Start presentMode} And {End presentMode}
          Existence {anyButton or timeout}
}
```

Listing 10.1: Query example that specifies that the presentation mode can only be left using button press or after hold time expiration.

This instance is bound to lighting system shown in Chapter 2 by referencing its name in the Experiment instance as shown in Listing 10.2. This binding allows the Query instance to reason about all controllers, luminaires, sensors etc. of that particular lighting system.

```
Experiment MeetingRoom1
System MeetingRoom1Control
Query canLeaveMeetingRoom
```

Listing 10.2: Experiment language instances binding the Query of Listing 10.1 to a lighting system.

In following sections, we will discuss the concrete syntax of the Query language for the lighting domain from top to bottom. Section 10.3 outlines statements, most notably the concrete syntax for pattern/scope combinations of Section 9.4. Section 10.4 outlines expressions used to create composite propositions using the operators outlined in Section 9.3 and finally Section 10.5 outlines the concrete syntax for atomic propositions outlined in Section 9.2.

We wish to make two remarks about notation. Firstly the rules presented in this chapter are shown as EBNF closely similar, but simplified compared to, the input language of Xtext. Secondly, if you wish to understand how the concrete syntax of the Query language maps to the abstract, most rule names correspond to the class names presented in the metamodel of Appendix A. The boolean AND rule indeed produces an instance of the BooleanAndExpression class of the metamodel with references to its left and right operands.

10.3 Statements

```
Statement:
    PropositionDefinitionStatement
  | QueryDefinitionStatement
  | ExpressionStatement
  | BlockStatement
  | WithStatement
;
```

10.3.1 Proposition definition statement

```
PropositionDefinitionStatement:
  'let' name=ID (':' type=Type)? ('=' expression=AssignmentExpression)?
;
```

```
let a : Event // the variable a is of type event

let b = ControllerState ctrl1::Presentation // the variable b is inferred
     to be of type state

let c : Event = b // Error, variable b is of type state, and c is
    specified to be of type event

let d // Error, cannot infer the type of d, can both be state or event
```

A proposition definition statement serves as a declaration of a *proposition variable*. These variables store (typed) references to (atomic/composite) propositions. A proposition definition consists of a *name*, (possibly) a *type* and (possibly) an *expression*. If the latter is specified, the created variable is directly assigned a reference to the proposition specified in the expression.

Proposition variables are mutable but have a static type: if a proposition variable `b` is defined with type state assigning a proposition expression of type event results in an error. The type of a proposition variable has to be known at all times either by specifying a *type* in its declaration or by directly assigning a proposition expression. In the latter, we infer the type: the variable is created with the same type as the type of the expression. Finally, if both a type is specified in the declaration and an expression is assigned, the type specified has to match the type of the expression being assigned.

10.3.2 Block Statement

```
BlockStatement:
  '{' statements+=Statement* '}'
;
```

```
{
  let a : State
  {
    let b : State
  }
  {
    let c : State
  }
}
```

A block statement allows to logically group statements. This grouping is not only optically by enclosing statements in curly brackets, but also semantical: if a block statement closes, all declared proposition variables go *out of scope* and can no longer be referenced. However, references that are in scope prior to the block statement opening remain in scope. In the example above, at the declaration of `c`, only `a` is considered in scope.

10.3.3 With Statement

```
WithStatement:
  'with' component=WithControllerComponent statement=Statement
;

WithControllerComponent:
    'controller' controller=[Control::Controller|FQN] 'as' name=ID
  | 'each' 'controller' 'implementing' template=[Template::Template|FQN]
     'as' name=ID
;

ControllerRef:
    'fqn::' controller=[Control::Controller|FQN]
  | ref=[WithControllerComponent]
;
```

```
with controller fqn::OfficeBuildingSystem.PresentModeController as ctrl {
  // ctrl refers to the PresentModeController of the OfficeBuildingSystem
}

with each controller implementing MeetingRoom1Template as ctrl1 {
  // ctrl1 refers to each controller implementing MeetingRoom1Template
}
```

The with statement allows creating an alias that can be used to reference a controller or set of (similar) controllers. A controller, as described in Section 2.2.3, is a domain concept: they provide the logical mapping between templates and sensors/actuators. Controllers are heavily referenced in the Query language, be it to define a sensor group proposition, or to reference a parameter, or to define a controller state proposition, etc.

Furthermore, controllers are often alike: in a typical lighting system there are only a few templates but many controllers instantiating these few templates. Specifying the same property for each of these similar controllers is tedious.

The with statement alleviates both specification burdens, it allows to: (1) create an alias to refer to a specific (single) controller of the lighting system or (2) create an alias to refer to all controllers that implement a particular template. The two alternatives of the `WithControllerComponent` rule specify these use cases respectively.

When using a controller to specify, for example, a sensor group proposition, we can now expect either of three *controller-like objects*: an alias to a controller, an alias to a set of controllers implementing a particular template or a direct (fully qualified) reference to a controller. The `ControllerRef` rule captures all three types of references.

What we call a *controller-like object* is also a thin abstraction. A single controller implements a template and inherits some of its elements. For example, a controller does not define its own state machine but inherits it from a template. We provide the double semicolon operator (`::`) to reference (selected) concepts of a controller or the template being implemented simultaneously. For example, in Listing 10.1 `ctrl1::Presentation` refers

to the presentation mode state of the state machine specified in the template implemented by the (set of) controller(s) aliased by `ctrl1`.

Values

```
Value:
  val=INT
  | controller=ControllerRef '::' param=[Template::Parameter]
;
```

```
// Example value
5
// Example value of a parameter of a controller
ctl1::HoldTime
```

Another example of this thin abstraction is the value rule displayed above. We saw that it is common to specify a property that, for example, asserts that the luminaire is on the "On" level. The exact value associated with the "On" level is specified in a *parameter* of a template and can be overridden in the controller. For specification purposes, it does not matter whether the exact value of the parameter is 80 or 40, we just want to specify that the luminaire is on the "On" level. Copying the values associated with these parameters is error-prone, we would much rather reference the parameter directly. The `Value` rule allows to either specify an integer value, or reference a parameter of a controller-like object. These references are aware that a parameter can be overridden by a controller and if a particular controller-like object does indeed override a parameter, the updated value is used in the generated models.

As a final note, parameters in the Prisma domain model are of one of three types: boolean, time of day or integer. Both hold times and actuator levels are implemented as integer parameters. Since we cannot distinguish between a time duration and a light level, we cannot restrict actuator propositions to only reference light level parameters. We hope that in the future parameters are more strongly typed in the domain model, one solution would be to further separate "duration" parameters from "level" parameters.

10.3.4 Query Definition Statement

```
QueryDefinitionStatement:
  'query' query=SingleQuery
;

SingleQuery:
    scope=Scope pattern=Pattern
;

Scope:
    'Between'  lower=RefOrInline       'And'   upper=RefOrInline
  | 'Globally'
  | 'Before'   proposition=RefOrInline
  | 'After'    lower=RefOrInline       'Until' upper=RefOrInline
  | 'After'    proposition=RefOrInline
;

Pattern:
    'Response'     effect=RefOrInline      'After'    cause=RefOrInline
  | 'Precedence'   cause=RefOrInline       'Before'   effect=RefOrInline
  | 'Absence'      proposition=RefOrInline
  | 'Universality' proposition=RefOrInline
  | 'Existence'    proposition=RefOrInline
  | 'Recurrence'   proposition=RefOrInline 'Every'    bound=Value
  | 'Invariance'   proposition=RefOrInline 'For'      bound=Value
      'When' cause=RefOrInline
;

RefOrInline:
    ref=ReferenceExpression
  | '{' exp=Expression '}'
;
```

```
query After a Until b Response d After c

query Before {a or b} Recurrence c Every 12

query Between {a or b} And {c} Existence d
```

A query definition specifies a property that has to be checked against the lighting system, e.g. it *queries* the model (in either simulation or verification) whether the *property* holds. Its syntax is modeled to be like an *assert* statement of a regular programming language and starts with the keyword `query`. A query definition itself exists of a property specification consisting of *scope* and *pattern* definitions. These pattern/scopes are instantiated using proposition expressions, but for readability, we have restricted these to either direct references to proposition variables, or expressions enclosed in curly brackets. The former is a shortcut: it is common to define propositions first and only reference them

in patterns/scopes. Forcing the use of brackets around these direct references would be superfluous.

10.4 Expressions

```
Expression:
  AssignmentExpression
;

AssignmentExpression:
  BooleanOrExpression ('=' AssignmentExpression)?
;


BooleanOrExpression:
  BooleanAndExpression ('or' BooleanAndExpression)*
;

BooleanAndExpression:
  BooleanNotExpression ('and' BooleanNotExpression)*
;

BooleanNotExpression:
    'not' BooleanNotExpression
  | PropositionExpression
;

PropositionExpression:
  StateProposition
  | EventProposition
  | TypeCastExpression
  | ReferenceExpression
  | '(' Expression ')'
;

TypeCastExpression:
  ('Start' | 'End') PropositionExpression
;
ReferenceExpression:
  proposition=[PropositionDefinitionStatement]
;
```

Expressions are the syntax for specifying the composite propositions as defined in Section 9.3.

Expressions are much like statements but always specify an (atomic/composite) proposition. The simplest expressions are atomic proposition literals described in Section 10.5. These literals can be combined using composite operators to make composite propositions. In other words, expressions are the concrete syntax for specification of (atomic/-composite) propositions.

Operator	Associativity
Assignment	right
Boolean Or	left
Boolean And	left
Boolean Not	left
Typecast	-
Parentheses	-

Figure 10.2: Precedence and Associativity for all operators of the Query language. Precedence levels are separated by vertical lines and ordered from low to high.

In the Query language, composite operators have both operator precedence as well as associativity. The level of precedence is used to determine the order of evaluation of an expression, in math for example multiplication has precedence over addition, $a + b \times c$ is evaluated as $a + (b \times c)$. When, in an expression, operators have the same precedence, associativity determines the order of evaluation. Consider for example $a \times b \times c$, the $\times$ operator is often implemented as left associative and therefore is evaluated as $(a \times b) \times c$. Table 10.2 shows precedence (from low to high) and associativity for all operators in the Query language, precedence levels are separated by a horizontal line.

Supporting precedence and associativity in Xtext requires us to specify a grammar in a certain way due to its underlying parsing algorithm. In this grammar, the deeper down the rule chain an operator is defined, the higher its precedence. Furthermore, two patterns of rule specification define left and right associativity respectively, the assignment expression is right associative whereas for example the boolean and expression is left associative. Notice the slight variations in these rules, the former uses a zero-or-one match, whereas the latter uses a zero-or-more match. This way of specifying a grammar is common in compiler construction, for more information we like to refer to Watt and Brown (2000).

10.4.1 Assignment Expression

```
a = b // Assign the reference held by b to the variable a.

a = b = c // Assign c to b and subsequently, b to a.

a = (b or c) = d // Invalid, cannot assign d to the expression (b or c).
```

An assignment expression assigns an (arbitrary) proposition expression to a proposition variable. Assignment, however, is more of a syntactic concept. No proposition is specified, only references to other observers are assigned to variables.

Assignment can be chained and is evaluated right to left. For the second example, first `c` is assigned to `b`, and then `b` is assigned to `a`. The left expression of an assignment must be a reference to a variable, assigning an expression to another expression yields

an error. Finally, assignment to variables of different types is not allowed, assigning an event proposition expression to a state proposition variable or vice versa is prohibited.

Reference Expression

```
a // Reference to the variable a.
```

A reference expression references a proposition variable by name.

10.4.2 Boolean OR Expression

```
a or b or c or d // If any of a, b, c, d is true/fires, this proposition
    also is true/fires. Evaluated as ((a or b) or c) or d.
```

A boolean OR expression specifies a proposition that holds or fires if any of its operands hold or fire. Both operands of the expression have to be propositions of the same type, either state or event propositions.

10.4.3 Boolean AND Expression

```
a and b and c and d // If all of a, b, c, d are true, this proposition is
     also true. Evaluated as ((a and b) and c) and d.
```

A boolean AND expression specifies a state proposition that holds only if both operands hold. Both operands of the expression have to be propositions of type state.

10.4.4 Boolean NOT Expression

```
not a // If a is true, this proposition is not.
not not a // If a is true, this proposition is also true.
not a and not b // If both a and b are not true, this proposition is true
    , otherwise its false. Evaluated as (not a) and (not b).
```

A boolean NOT expression specifies a proposition of type state that holds only if its (state proposition) operand does not.

10.4.5 Typecast Expression

```
Start a // Event proposition that fires when state proposition a starts
    holding.
```

A typecast expression specifies an event proposition that fires when another state proposition starts to holds or starts to be refuted. The operator component of the specification defines on which of the flanks of the state proposition the event proposition fires, either the rising flank ("Start", when the state proposition starts holding) or the falling flank ("End", when the state proposition stops holding).

10.5 Atomic Propositions

Atomic proposition literals are the concrete syntax for specifying atomic propositions as defined in Section 9.2.

10.5.1 SensorSignal

```
'SensorSignal' sensor=[Building::Sensor | FQN] feature=SensorFeature
```

```
// A sensor in the theater detects occupancy.
SensorSignal OfficeBuilding.Floor0.Theater.SensorTheater Occupancy

// A button in the theater is pressed.
SensorSignal OfficeBuilding.Floor0.Theater.PresentationModeButton Button
```

A SensorSignal literal specifies an event proposition that fires when a particular sensor detects a particular sensor feature. The sensor component of its definition is any sensor present in the Building model and referenced in fully qualified manner. The feature component specifies which feature of a particular sensor to listen to: occupancy or button presses. In theory sensors in the Building model can have multiple features requiring a user of the Query language to specify which feature to listen to, in the future we suggest to make the feature component optional for single featured sensors.

10.5.2 SensorGroupSignal

```
'SensorGroupSignal' controller=ControllerRef '::'
  sensorGroupMapping=[Template::SensorGroup | FQN]
    feature=SensorFeature
```

```
// Any of the occupancy sensors of the OccupancySensor group of a fully
    qualified controller has detected presence.
SensorGroupSignal
  fqn::OfficeBuildingSystem.PresentModeController::OccupancySensors
      Occupancy

// Any of the buttons of the PresentationModeButtons group of an aliased
    controller has detected presence.
SensorGroupSignal ctrl1::PresentationModeButtons Button
```

A SensorGroupSignal literal specifies an event proposition that fires when any sensor in a particular sensor group detects a particular sensor feature. The sensor group component of its definition is any sensor group of a controller-like object. A SensorGroupSignal proposition literal is a shortcut, the same proposition could be created by using boolean OR to combine SensorSignal propositions for all sensors in this particular sensor group.

10.5.3 ActuatorLevel

```
'ActuatorLevel' actuator=[Building::Actuator | FQN] 'level' level=Value
```

```
// LuminaireRoom1 located on Floor0 in the Theater room has output level
    80.
ActuatorLevel OfficeBuilding.Floor0.Theater.LuminaireRoom1 level 80
// LuminaireRoom1 located on Floor0 in the Theater room has an output
    level specified by the LevelPresentRoom parameter of the ctrl1
    controller-like object.
ActuatorLevel OfficeBuilding.Floor0.Theater.LuminaireRoom1 level ctrl1::
    LevelPresentRoom
```

An ActuatorLevel literal specifies a state proposition that holds when a particular actuator is on a particular level. The actuator component of its definition is any actuator present in the Building model. The value component is any Value as specified in Section 10.3.3.

10.5.4 ActuatorGroupLevel

```
'ActuatorGroupLevel' controller=ControllerRef '::'
  actuatorGroup=[Template::ActuatorGroup] 'level' level=Value
```

```
// All luminaires in the RoomLuminaires of the controller-like object
    ctrl1 have output level 80.
ActuatorGroupLevel ctrl1::RoomLuminaires level 80
// All luminaires in the RoomLuminaires of the controller-like object
    ctrl1 have output level equal to the value of the LevelPresentRoom
    parameter of that same controller-like object.
ActuatorGroupLevel ctrl1::RoomLuminaires level ctrl1::LevelPresentRoom
```

An ActuatorGroupLevel literal specifies a state proposition that holds when a particular actuator group is on a particular level. The actuator group component of its definition is any actuator group of a particular controller-like object. An ActuatorGroupLevel proposition literal is a shortcut, this proposition could also be specified by using boolean AND to combine ActuatorGroupLevel propositions of all actuators in this particular actuator group.

10.5.5 Timer

```
'Timer' bound=Value 'Reset' exp=RefOrInline
```

```
// A Timer that fires every 5 seconds unless reset by the occurrence of a
    .
Timer 5 Reset a
// A Timer that fires each time HoldTime seconds of controller-like
    object ctrl1 have passed unless reset by the occurrence of a or b.
Timer ctrl1::HoldTime Reset {a or b}
```

A Timer literal specifies an event proposition that fires when a specified amount of time has passed since the timer has last reset. The bound component of the specification is any Value as specified in Section 10.3.3. The reset component is an event proposition, but the RefOrInline rule suggests curly brackets around complex reset expressions.

10.5.6 ControllerState

```
'ControllerState' controller=ControllerRef '::' state=[Template::State]
```

```
// The controller-like object ctrl1 is in Presentation state.
ControllerState ctrl1::Presentation
// The fully qualified PresentModeController of the OfficeBuildingSystem
    is in On state.
ControllerState fqn::OfficeBuildingSystem.PresentModeController::On
```

A ControllerState literal specifies a state proposition that holds when a particular controller is in a particular state. The component of its definition is any controller-like object. The state component is any state in which this controller-like object can be.

10.6 Static Semantics

A program can be syntactically correct but have no meaning. Consider for example the proposition expression `a or b`, what does this expression mean when `a` is not declared? During compilation we check that a program has a meaning, we check what is known as its static semantics. For the Query language, we do two forms of semantical analysis: we check whether a certain construct is referable and we check the types of expressions are compatible.

10.6.1 Reference Checking

Within the Query language, we allow specification of proposition variables and corresponding reference expressions. A proposition variable declaration has what is known as a scope: a part of the program in which this variable can be referenced. In the simplest case, a proposition variable is declared for the remainder of the Query language instance. Such a declaration is *globally scoped*. Block statements introduce new scope blocks, a proposition variable declared in a block can only be referenced in that particular block. Finally, when scope blocks are nested, all variables defined in a parent scope block are also accessible in the child scope block. In Listing 10.3 an example of scoping is given, the comments list which variables are considered in scope.

```
Queries scoping
let a : State // a in scope
{
  let b : State // a, b in scope
  {
    let c : State // a, b, c in scope
  } // c out of scope

  let d : State // a, b, d in scope

  {
    let c : State // a, b, d, c in scope. c is redefined.
  } // c out of scope
} // b, d out of scope
```

Listing 10.3: Scoping in the Query language.

We have implemented scoping for controller aliases in much the same way. An alias created by a with statement is only considered in scope for the statement of that with statement. If that statement is a block, the reference is valid for the remainder of that block. An example of this scoping is displayed in Listing 10.4.

```
Queries controllerAliases

with each controller implementing MeetingRoom1Template as ctrl1 /* ctrl1
    starts being in scope */ {

} // ctrl1 stops being in scope
```

Listing 10.4: Scoping of controller aliases in the Query language.

Controller references require some custom scoping rules. Consider for example a controller state proposition, in its declaration we specify a reference to a controller-like object and reference a state of a template. By default, Xtext checks the controller reference and checks that there is a state of the name specified. What Xtext does not check, is whether the controller-like object does indeed have that particular state, Xtext allows to reference a state of *any* template currently known. We have encoded this implicit dependency in a custom scoping rule: we first resolve the controller reference and only states of the template implemented by that controller-like object are in scope. Similar scoping rules were created for referencing parameters and sensor/actuator groups.

10.6.2 Type Checking

The Query language has a (rudimentary) type system, propositions (and expressions combining propositions) are either of event or of state type. We check for each composite operator as well as for each pattern/scope whether the types of its operands are correct.

For example, for the boolean OR operator we check that both operands are of the same type. For patterns, the allowed proposition type depends on the pattern. Universality and invariance patterns require state propositions as an argument whereas all other patterns require event propositions. For scopes, all proposition arguments need to be of type event.

10.7 Desugaring

Part of the compilation process of the abstract syntax to both UPPAAL and the Co-simulation Framework is shared in a process that is often called *desugaring*. In this process, we reduce a Query language instance to a set of (atomic/composite) propositions and (instantiated) pattern/scope combinations by removing controller references and parameters. The resulting input to the specific generation steps of both the UPPAAL model as well as the Co-simulation Framework configuration resembles the abstract model presented in Figure 9.1.

Consider the example in Listing 10.1, the with-each-controller statement specifies that `ctrl1` refers to a controller implementing the `MeetingRoom1Template`. The lighting system to which this Query instance is bound specifies only one controller that does. In other words, we can remove the with-each-controller statement by replacing each occurrence of `ctrl1` by a direct reference to that particular controller. Similarly, when multiple controllers do implement a particular template, we can "unroll" the implicit loop by systematically copying the block of the with-each-controller and replacing `ctrl1` with a direct reference to each controller implementing that template.

The second form of desugaring replaces references of parameters by their concrete value. In the example in Listing 10.1 we specify a Timer proposition that fires after "hold time" seconds. This hold time is either specified in the controller or inherited from the template. Crucial, however, is that this hold time cannot change, it is a *constant*. The default value of the hold time specified in the template is 600 and the (single) controller implementing this particular template has not overridden this default. We can replace `ctrl1::HoldTime` with `600`.

11 Generic Translation: Monitors and Rules

In the traditional works of pattern libraries (discussed in Chapter 5), propositions are left to be specified. Translations of pattern/scope combinations assume that these propositions can be observed. For example, when translating the property specification "Globally: Absence P" to CTL, the proposition P is assumed to be assigned by a labeling function to states in the model for which this proposition P holds. The Query language we propose has made this labeling function explicit by introducing specifications for (atomic/composite) propositions. The implementation of observers for these propositions are dependent on the target for translation and will be discussed separately in Chapter 12 (for UPPAAL) and Chapter 13 (for the Co-simulation Framework) respectively.

The other aspect of translation, translating pattern/scope combinations, has been extensively researched in the various pattern library approaches. While we initially planned on using an existing translation, we were unable to reuse any of these. We have therefore split our translation into two parts, first, we translate a pattern/scope combination to an intermediate form and second, we translate that intermediate form to the tools we support. This chapter will outline why we need such an intermediate form and how it was created. The two subsequent chapters discuss the implementation details for both UPPAAL and the Co-simulation Framework.

11.1 The Need for a Two-Stage Approach

As we use a pattern library approach, it would make sense to use a direct translation to temporal logics. Since we are dealing with timed temporal specifications, the obvious translation candidates would be either MTL or TCTL as described in Chapter 4.

These direct translations are problematic both in the formal verification and the simulation flow. The Co-simulation Framework does not have a notion of checking a property against a (path in a) lighting system meaning that either way, we have to extend the Co-simulation Framework to support the Query language. While implementing MTL or TCTL for simulated paths has been done before (for example MTL has been implemented in the TRACE tool by Hendriks et al. (2016)), we do not need the full expressiveness that such an implementation offers. The set of pattern/scope combinations is small, and therefore we only use a small number of TCTL/MTL constructs. Implementing full TCTL/MTL in the Co-simulation Framework is a sizeable amount of work that we would rather avoid for its limited use in the Co-simulation Framework.

In the formal verification flow, the use of TCTL/MTL is more obvious. These timed

temporal logics have been created to use in model checking and many papers have been written about how to employ these logics in such contexts. However, while not set in stone, the Prisma project has selected UPPAAL as its verification tool. UPPAAL does support TCTL, but only a very restricted subset. The restriction is so severe, that any pattern/scope combination from our pattern library almost certainly translates to a TCTL formula that is not supported by UPPAAL.

That observation raises the question whether UPPAAL is the right choice if its TCTL subset is so restrictive? While exploring different model checkers we found that UPPAAL is unique in the field of practical model checking. There are very few timed model checkers that are as well maintained as UPPAAL and we have found no model checkers that on top of that, also support full TCTL or MTL. The only other candidate we have found is called Kronos by Yovine (1997) and claims to implement both a timed formalism and full TCTL, but Kronos is largely unmaintained. Its successor, OpenKronos by Tripakis (1998) has since been released and also abandoned. Both tools have not gained the traction that UPPAAL has. UPPAAL is surprisingly unique in its formalism, its level of maturity, and its easy to use interface.

On the other hand, we have encountered many model checkers that are not timed but have implemented full (untimed) temporal logics. If the models can be transformed into a non-timed formalism, we can use an untimed model checker like NuSMV by Cimatti et al. (1999). The current model of the lighting system uses quantitative time to model timing behavior such as *hold time*: the time before a vacant room returns to the "Off" preset. Hold time could however also be modeled using a non-deterministically occurring timeout event. While there are indeed cases in which timing can be removed, we have seen specifications in which it can not (for example the property specified in Listing 14.1). Furthermore, newly created auxiliary models like Network do add more timing to the model. We believe strongly that removing timing from the models will hamper both expressiveness and extensibility of the Query language approach.

Using timed temporal logics as our target for translation is infeasible in both formal verification an simulation, leaving us with no choice but to split the approach in such a way that we can express these pattern/scope combinations without using full timed temporal logics.

11.2 Splitting the Approach

When evaluating such a split, UPPAAL is of first concern. UPPAAL, unlike the Co-simulation Framework, cannot be extended easily and therefore we need to create the pattern/scope combinations using the tools it provides. In UPPAAL, the modeling formalism is timed automata while the property specification language is a subset of TCTL. This subset consists of five temporal operators operating on boolean expressions. These expressions reason about the current state of the system (variables, current state, clocks, etc.). The five temporal operators supported in UPPAAL are:

Possibly An expression is eventually true along at least a single possible path in the

model. In TCTL: EF exp.

Invariantly An expression is true in all states along all possible paths in the model. In TCTL: AG exp.

Potentially Always An expression is true in all states along at least one possible path in the model. In TCTL: EG exp.

Eventually An expression is eventually true along all possible paths in the model. In TCTL: AF exp.

Leads To If along a possible path an expression is true, then at some point in the future of that path, another expression becomes true as a response. In TCTL: $\mathrm{AG}(exp_1 \rightarrow \mathrm{AF}\, exp_2)$.

This set appears to have been carefully selected to cover different properties. There are temporal operators for specification of liveness properties (Possibly, Eventually, Leads To) and temporal operators for specification of safety properties (Invariantly, Potentially Always). Some operators specify properties that hold for all paths (Invariantly, Eventually, Leads To), others specify properties that can also be true if they hold for at least a single path (Possibly, Potentially Always) etc.

Not only is the set of temporal operators carefully selected, the modeling formalism (timed automata) is also quite expressive. Doornbos et al. (2015) have used UPPAAL to verify parts of the lighting systems of Philips Lighting manually. While they did not employ a generic approach as outlined in this research, they found that splitting property specifications into an automaton and a (simple) TCTL query is indeed expressive enough to cover properties they intended to verify. Furthermore, the works by Gruhn and Laue (2006) have shown that pattern libraries can also be translated to (a more expressive version of UPPAAL's) timed automata.

The realization that the TCTL subset of UPPAAL is not as restrictive as we had feared, the promising results by Doornbos et al. as well as the pattern library translations to timed automata by Gruhn and Laue (2006) made us split the translation in what we call a *monitor* and a *rule*. These monitor/rule combinations offer an alternative way of translating a timed pattern/scope combinations to a form usable in both the Co-simulation Framework and UPPAAL.

A monitor is a timed state machine that faithfully follows the execution of the system under test. A monitor automaton is reactive: it can respond to observations made of the actions in the system under test but does not progress on its own. A monitor is also impartial, while the structure of such a monitor encodes a certain pattern/scope combination, a monitor alone cannot conclude that a property is violated. Finally, a monitor is prioritized over the rest of the model, meaning that if a monitor can transition, it will do so before any transition in the model can be taken. These monitors are like the observers of Gruhn and Laue (2006) but we separate rules and monitors, provide implementations of extra patterns, have slight variations on semantics of some patterns

and better monitors for properties on state propositions. While certainly overlapping with the work of Gruhn and Laue, we do add on top of their approach.

Rules operate on top of these monitors and analyzes the sequence of states that a monitor is in. These rules correspond to three of the TCTL queries expressible in UPPAAL's TCTL subset: Invariantly, Eventually and Leads To. The two remaining UPPAAL operators, Possibly and Potentially Always, are of the existential kind, but no pattern/scope combination in our library is. These rules are more explicitly executable in both UPPAAL and the Co-simulation framework. The three rules are:

Forbidden Rule A state of the monitor cannot be reached, if the state is reached the property is refuted.

Eventually Rule A state of the monitor has to be reached eventually, if it is never reached the property is refuted.

Response Rule An occurrence of a certain monitor state has to be followed by the occurrence of another certain monitor state eventually, if the "cause" happened, but no "response" happens, the property is refuted.

Finally, what we call a *judge* is the (part of a) tool that receives the (consecutive) states of a monitor and produces a *verdict*. This verdict is a message that conveys the validity of a property. In UPPAAL we reuse the model checker as the judge, rules are specified in TCTL and the model checker either says the property holds or is refuted. In the Co-simulation Framework, however, we have created a custom judge as part of the framework that produces verdicts continuously.

To support a new tool, a system has to be found or created that can act as a judge, the rules have to be translated to some form that the tool understands and finally, the monitors have to be implemented. We believe that this framework of rules, monitors, judges and verdicts is easier to implement in a new tool than implementing a full timed temporal logic.

11.3 Monitor Construction

We have created monitors, and selected accompanying rules for all pattern/scope combinations listed in Section 9.4. For each unique combination of pattern and scope, one unique monitor and rule combination is provided. These monitor/rule combinations, just like their temporal logic counterparts, can be instantiated by binding proposition observers to the abstract proposition channels defined in a monitor. These monitors are event-based, an edge in a monitor is labeled with an event occurring in the system being monitored, or the start or end of a state proposition holding. Furthermore, these monitors assume broadcast communication, if a certain event occurs but the monitor is not currently sensitive to this particular event, the broadcast is silently ignored.

11.3.1 Scopes

While conceiving a monitor/rule combination for a Globally-scoped pattern is straightforward, construction of other scopes can be difficult. Often it is important to realize what it means for the validity of the pattern/scope combination when a scope closes. Some scopes, like Between, are only sensitive if the scope is eventually closed. The monitors, however, follow the system and cannot predict whether the scope is going to close eventually. If the pattern/scope combination specifies safety properties (for example, "Between Absence"), violating the pattern in an opened, but not yet closed, scope leads to a state labeled *Possibly ERROR*. These states signal that a property is not yet refuted but the only way for a property to continue to hold is for the scope to never close (and therefore the scope has never been sensitive in the first place).

Scopes can also influence what type of rule is chosen. For example, properties specified using the Response-pattern can be both safety and liveness properties depending on the scope chosen. A property specified using "Before Response" can only be refuted if the scope eventually closes. Furthermore, if the scope closes we can immediately decide whether the property holds or not. If, when the scope closes, the monitor is in the state where a cause has happened, but no response came, the property is refuted.

Besides scopes not being sensitive when they are never closed, it is also important to realize that a scope can have infinite length. For example, an "After/Until"-scoped "Response"-pattern remains sensitive if the scope opening event occurs, but the scope closing event does not. In the case of the Response-pattern, this leads to the use of a different type of rule compared to the "Between Response" case.

11.3.2 Example Construction: Existence

To demonstrate the thought process going into such a construction, reconsider the Existence-pattern. A property specified using the Existence-pattern says that a certain event has to occur in a scope. We have picked existence for this discussion because it is an interesting case. Properties specified using Existence are liveness properties if the scope is (potentially) infinite and otherwise, they are safety properties.

Globally: Existence P

In the Globally-scoped case, only if the (event) proposition P is observed the specified property holds. Until such a P has occurred, we cannot say whether the specified property is going to hold. The monitor corresponding to such a property is shown in Figure 11.1 and listens for the occurrence of P. The corresponding rule, Eventually OK, asserts that the monitor reaches the OK state in which the property holds.

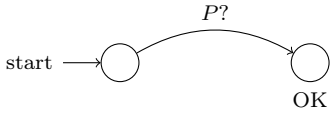


Figure 11.1: Monitor for Globally: Existence P

After Q: Existence P

The After-scoped case is very similar to the Globally-scoped case: after the occurrence of Q, the scope is infinite. Constructing the After case is equal to adding a new initial state (in which Q has yet to occur) and using a transition (Q occurring) to the first state of the globally case. The corresponding monitor is shown in Figure 11.2.

The tricky bit, if Q never occurs, no occurrence of P has to follow. If we were to use the same eventually rule as in the global case, we would reject paths on which Q does not occur in the first place. Therefore we select the response rule, and state that when Q has occurred, eventually P has to occur as a response: Response OK to Possibly OK.

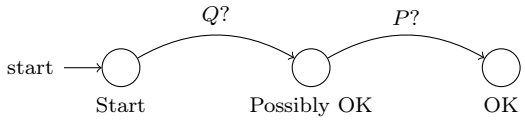


Figure 11.2: Monitor for After Q: Existence P

Before R: Existence P

For the Before-scoped case, either R has to never occur (the trivial case), or P has to occur before R. In other words, when R occurs we can immediately make a decision on whether the property holds, if R does not occur at all, we do not need to make a decision as the property holds trivially.

This is reflected in both the monitor and the rule. The monitor shown in Figure 11.3 listens for P and R, if P occurs, either R is going to occur and the property holds, or R is not going to occur and the property holds trivially. Observing P before R means that in all cases, the property is going to hold. On the other hand, if R is observed before P, the property cannot hold. Therefore, the only way to refute such a property is to observe R before P. We mark this state ERROR and the accompanying rule, Forbidden ERROR, asserts that no execution reaches this state.

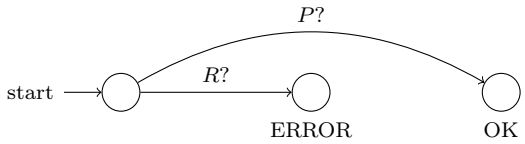


Figure 11.3: Monitor for Before R: Existence P

After Q Until R: Existence P and Between Q And R: Existence P

For monitoring properties specified using Existence pattern and After/Until or Between-scope, we use the same monitor but different rules. The monitor shown in Figure 11.4 is simple, if Q occurs it is (potentially) sensitive, if a P follows, the pattern holds within that scope and if an R follows the scope is subsequently closed. If an R follows a Q before a P occurs, the property is violated in both the After/Until case as well as the Between case.

Recall that an unclosed scope for Between-scoped patterns is not sensitive, whereas for After/Until-scoped patterns such a scope is sensitive forever. This means that for After/Until, a scope opening but never closing has to be followed by P occurring. This subtle difference means that rule is slightly different, in the case of Between it suffices to say that the ERROR state is never reached: Forbidden ERROR.

For After/Until-scoped patterns, scope opening has eventually to be followed by P occurring (both in the finite as well as the infinite case): Response OK to Possibly OK.

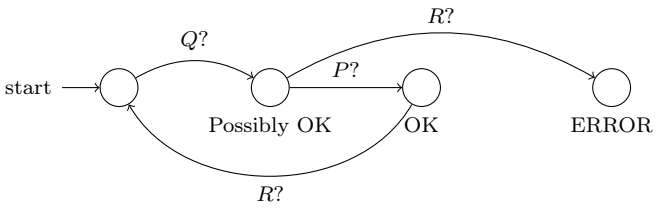


Figure 11.4: Monitor for both After Q Until R: Existence P and Between Q and R: Existence P

11.3.3 Universality Patterns and Scopes

As discussed before, there are patterns that reason about state propositions. One such example is the Universality-pattern that says that a state proposition holds for the duration of a scope.

A monitor for properties specified using the Universality pattern is still event-based and uses the flanks of a state proposition as transitions. As a notation, we use ¬P to denote the falling-, and P to denote the rising flank. Initially, the status of a state proposition is unknown, in the implementations for both UPPAAL and the Co-simulation Framework, a state proposition reports its initial status (holds, refutes) as soon as it can. Before such a message is received, we cannot conclude whether such a property holds or not. When considering, for example, properties specified using Globally-scoped Universality-patterns, we can only say whether the property held initially after the (tool dependent) implementation of the proposition communicates its state. If this initial message is negative, the property has never held, if it is positive, the property holds for now.

In other scopes, we cannot rely on the tools broadcasting the initial state of a proposition on scope opening. Consider, for example, the property specification: “After Q Until R: Universality P”, we can only conclude that P held throughout the scope if P held

before Q occurs and continues to hold after R has occurred (if it occurs).

We propose a simple solution: a monitor following a property specified using the Universality-pattern tracks whether its proposition holds regardless of whether a scope is open or not, but the corresponding rules of these pattern/scope combinations only reason about states in the "sensible" part of such a monitor. The monitor for "After Q Until R: Universality P" is displayed in Figure 11.5, initially the monitor moves to the *Holds* or *Refutes* states (if tools indeed convey the initial state of such a proposition) and only if Q occurs, the monitor can reach the *ERROR* state. The corresponding rule is then forbidden ERROR.

A keen observer would note that if P does not hold before Q occurs, the property is refuted. For specification of some properties, however, we might wish to state that a state proposition starts to hold as a response to scope opening and stops to hold as a response to scope closing. Constructing this alternative semantic is also possible in our monitors by introducing an intermediate state between Refutes and ERROR. In this intermediate state no time can be spent (or in case of a timed Universality, only limited time can be spent) and if P starts to hold immediately, the property is not yet refuted. If however, P does not start to hold immediately, the ERROR state is still reached and the property is refuted. This different semantic is given in Figure 11.6. Please note, however, that the transition to the *ERROR* state from this intermediate state needs to have lower priority than the one to the *Holds*, *Sensitive* state.

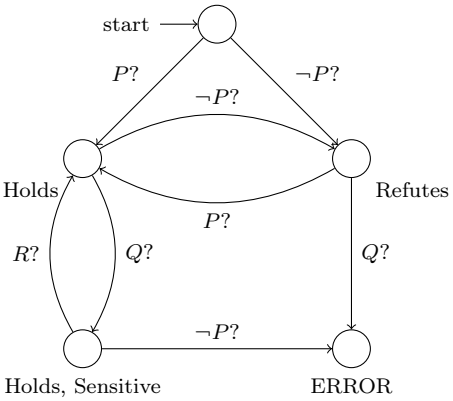


Figure 11.5: Monitor for After Q Until R: Universality P

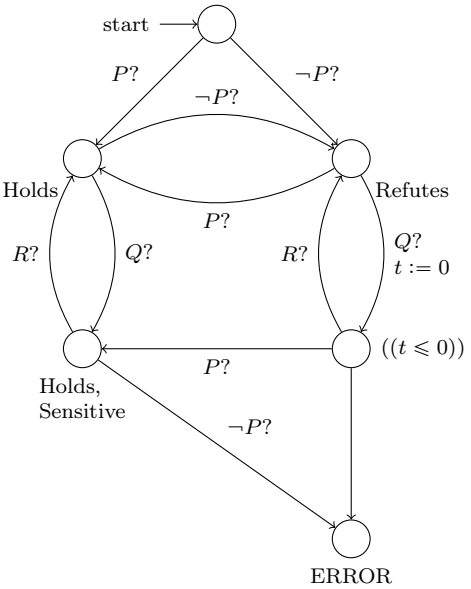


Figure 11.6: Monitor for After Q Until R: Universality P that allows P to start holding after Q occurred

11.3.4 Duration Patterns and Scopes

Duration patterns offer an interesting challenge when combined with scopes. Duration patterns (like Minimum Duration and Maximum Duration) state that a (state) proposition has to hold for at least or at most a predefined amount of time. When considering scoping, what would the "Minimum Duration" pattern mean on the edges of a scope?

There are three ways to go about these property specifications, which we will discuss with an example. Let us consider a lighting system that is considered active between nine and five, idle from six to nine and five to eight and off at other times. During office hours, the systems hold times are long (e.g. 15 minutes) and in idle mode they are short (e.g. 5 minutes).

The property: "When the system is in active mode, luminaires stay on for at least 15 minutes" could be specified as "After {active mode engage} Until {active mode disengage}, Minimum Duration {luminaires on} 15 minutes". On the edges of "active mode" a monitor could:

1. Disregard any behavior of luminaires that potentially falls outside the scope, meaning that if a luminaire is on before active mode is engaged we just do not care about whether it stays on for fifteen minutes after. The only durations evaluated are the one started and ended within the scope.

2. Impose the new rule on already holding propositions, meaning that if a luminaire was already on for three minutes when active mode engaged, it has to stay on for at least twelve minutes for the property to hold. When a luminaire is on for three minutes when active mode is disengaged, it has to remain on for 12 minutes for the property to hold.

3. Impose rules on propositions, without regards to whether they held before the scope. Meaning that if a light was on for 3 minutes prior to the scope becoming active, it has to stay on for 15 minutes for the property to hold. On scope closing the property only holds when luminaires are off, or on for at least 15 minutes at the time of the scope closing.

All three semantics can be implemented in a monitor (as shown in Figure 11.7), but the question remains which of the three semantics is the (most universally) desired one. The first semantic is the least powerful, but safest to assume, propositions that hold (partly) out of scope are all disregarded. The other two semantics are both strictly stronger than the first, but if a property holds using either of these two, it also holds using the first.

In reality, these types of lighting systems are often implemented using two different (logical) controllers that act in a master/slave configuration. Both controllers receive all sensor input and continuously update the state of their model according to their own rules. Only the controller currently appointed master can propagate its state to the luminaires in the lighting system. When the roles switch due to the active mode enabling/disabling the newly appointed master sends its current state to all luminaires. Luminaries in rooms that have been vacant for four minutes when active mode engages

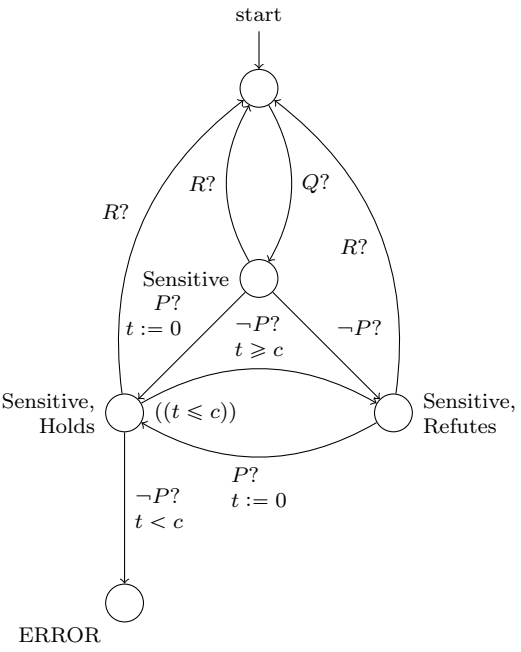
remain on for 11 minutes. Luminaries that are on for four minutes when active mode disengages only remain on for an extra minute (the new master controller overwrites the hold time). To specify a property that asserts luminaires are on for at least 15 minutes in active mode we need a Minimum Duration pattern with the second semantic on scope opening and the first on scope closing.

However, there are multiple ways of specifying such a system, and these considerations are not trivial. TNO-ESI, for the Prisma project, has decided that these special cases warrant their own research. The choice of semantics for the duration patterns with respect to scope opening and closing can only be made once choices have been made about the correct interaction of these two features. We would like to stress that the monitor/rule combinations outlined in this chapter are able to implement these slight variations in semantics.

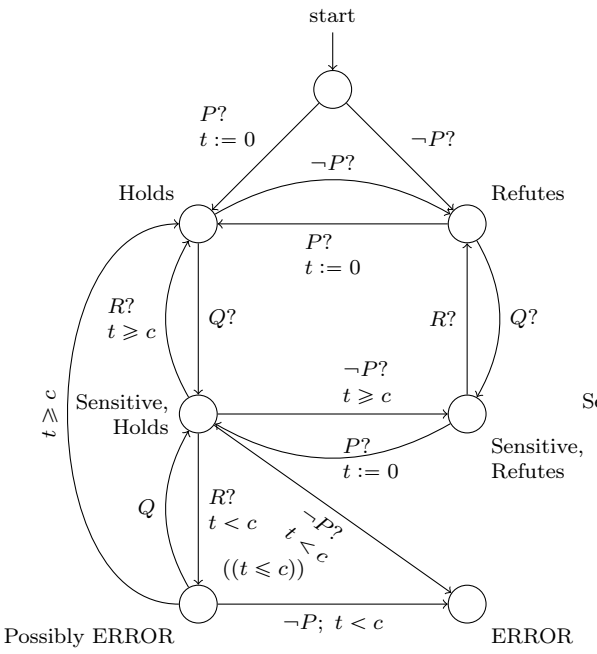
11.4 Correctness of Monitor/Rule combinations

We have validated these monitor/rule combinations in discussions but not in a formal proof. We would be most interested in seeing whether a monitor/rule combination is equal in meaning as, for example, the CTL formula for the same pattern/scope. Similarly, we would like to see the differences between pattern/scope implementations in different temporal logics.

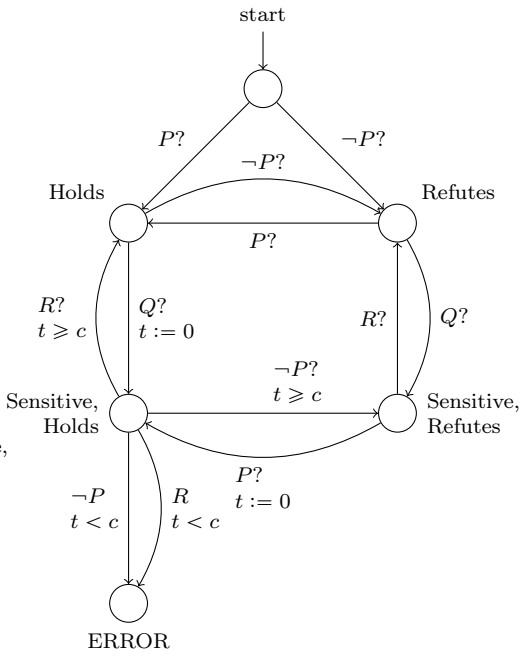
However, we think that this comparison is far from trivial, our monitor rule combinations use a combination of state and event propositions, but the rules are distinctly state based. Furthermore, we think that formal proofing of monitor/rule combinations will discover discrepancies between our implementation and the (reference) LTL/CTL/MTL/TCTL implementations but wonder whether these different semantics are harmful for the approach. It could very well be that we have found a slightly *different* semantic that is not necessarily wrong. For a more throughout discussion we would like to direct to Section 18.4.



(a) Behavior out of scope not considered



(b) Behavior out of scope part of the minimum required duration



(c) Behavior out of scope not part of the minimum required duration

Figure 11.7: After Until scoped Minimum Duration patterns for three different semantics

12 Translation to UPPAAL

The previous section outlined how pattern/scope combinations can be translated to combinations of (generic) monitors and rules. To implement the Query language in UPPAAL, we provide implementations of these monitors and rules in UPPAAL.

Furthermore, in the previous chapter, we had deferred the implementations of proposition observers. In UPPAAL, we have implemented proposition observers (both atomic and composite) as automata, which we also call monitors. Atomic proposition monitors (for example for an "ActuatorLevel" proposition) synchronize with the lighting system model. Composite proposition monitors can synchronize with these atomic proposition monitors to observe composite propositions (for example, `a and b`). Finally, the pattern/scope monitor synchronizes with (composite or atomic) proposition monitors to implement the properties specified in the Query instance.

This chapter is structured as follows, firstly, we provide a brief overview of UPPAAL's semantical model. Secondly, we discuss the intermediate format that TNO-ESI uses in its translation to UPPAAL. Thirdly, we outline how the domain model of the lighting system was implemented in this intermediate language. Fourthly, we outline our implementations of monitors and (atomic and composite) proposition monitors. Finally, we provide an overview how these monitors interact to create the entire property being specified.

12.1 Semantical Model of UPPAAL

A UPPAAL model is split into a *system description* and a *requirement specification*. The first describes the system, the second specifies properties (in UPPAAL's TCTL subset) for the system specified in the first. This property language has already been extensively discussed in the previous chapter.

A system description in UPPAAL consists of a network of timed automata that are operating in parallel. The semantics of the model is described by timed transition systems, paths in such a transition system can either delay or take a discrete transition. UPPAAL is not eager: paths in UPPAAL can stay in a state until they are forced to move out of such a state. UPPAAL's timed automata are extensions of the timed automata discussed in Section 4.3, we will discuss two such extensions: channels and urgent/committed states. We denote urgent states with an U and committed states with a C in such a state.

Automata in UPPAAL synchronize using *channels*. There are two types of channels: binary channels and broadcast channels. An edge in a UPPAAL can have a *synchronization label* for a channel e of form $e!$ or $e?$[1], the first is used to send over a channel,

[1] In fact e may be any (UPPAAL) expression evaluating to a channel.

whereas the second receives over that channel. If the channel is binary, sending over such a channel forces a single other TA to take a transition that receives on that same channel. If no such TA exists, the sender is blocked until the synchronization can occur. If the channel is a broadcast channel however, sending over such a channel forces all other timed automata that currently have an enabled transition receiving over the same channel to take that transition. If no such transition exists, the broadcast is lost in transmission but the timed automaton will not block. Channels can also be marked *urgent*, if there is an enabled transition in any timed automaton that sends over an urgent channel it has to be taken without time delay.

States in UPPAAL can be urgent and committed. If any of the timed automata is in an urgent state, delay transitions cannot be taken unless all urgent states are left. A committed state is also urgent but furthermore requires that if a system is in one or more committed states, the next (discrete) transition has to move out of at least one of these committed states.

12.2 The State Machine Language

Within the Prisma project, transformations to UPPAAL, POOSL and a Java simulation all use an intermediate format. This intermediate, called "State Machine", offers a textual form of specifying arbitrary state machines.

The benefits of such an intermediate language are clear. When transforming a new domain model to the various tools, only a transformation to the syntax and semantics of the State Machine language has to be created. The transformations of that State Machine language are responsible for providing a transformation to the syntax and semantics of various tools. For example, if n models transform to m tools, supporting all tools for all models yields $n \times m$ transformations. Using an intermediate language yields n transformations to that intermediate language and m transformations from the intermediate to a specific tool, $m + n$ in total.

This State Machine language is modeled like UPPAAL's network of timed automata including channels and committed/urgent states. However, no assumptions are made on whether the underlying tools are eager: both tools that have to take transitions as they enable (eager) as well as tools that allow for delays even though discrete transitions are enabled (non-eager) are supported. UPPAAL is of the latter, whereas the Java simulator and POOSL are of the former kind. Practically, accounting for eagerness restricts possible executions: transitions are enabled only periodically to prevent them from being taken continuously. We use the State Machine language only for the generation of UPPAAL models, so can assume that the underlying tool is in fact non-eager. We use the State Machine language as nothing more than a textual representation of a UPPAAL model.

12.3 Lighting System Model

In this section, we will outline how a lighting system specified using the various DSLs is implemented as a UPPAAL model. As this transformation was created before the

work presented in this report, we give no rationale on why the system is implemented as such. Throughout this section, we will use the example system as shown in Section 2.1 to outline how its sensors, sensor groups, controllers, actuators and actuator groups are implemented in UPPAAL.

12.3.1 Sensors

Sensors are (currently) the only source of input and each sensor in a lighting system is implemented as a small automaton. There are two such implementations: the first of the two can only fire periodically (with a period of half of the shortest hold time in a system), the second is can fire completely randomly. The periodic sensor implementation is mainly used with eager tools in mind, if a sensor could always fire it would starve the rest of the system. UPPAAL however, is not eager, so we can use the completely random implementation. For the sensor "SensorTheater", such an implementation is given in Figure 12.1.

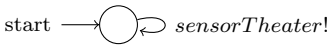


Figure 12.1: Sensor implementation for "SensorTheater" sensor in UPPAAL

12.3.2 Sensor Groups

Sensor groups are defined in a Template and instantiated in a Control instance. For each sensor group of a controller in a Control instance, an automaton is added to the UPPAAL model that binds the sensor implementations in UPPAAL to that specific sensor group.

The implementation of a specific instance of such a sensor group is trivial, listen to all sensors that are in the sensor group for this particular controller, and if any of these sensors detects, broadcast that this sensor group has detected. Consider for example the automaton for the sensor group "OccupancySensors" implemented by the "PresentModeController" controller, the UPPAAL implementation of this group is given in Figure 12.2, as the sensor group only has a single sensor ("SensorTheater"), it listens only to broadcasts of that single sensor.

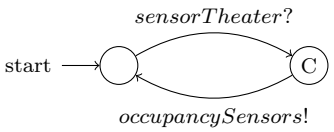


Figure 12.2: Sensor group implementation for the "OccupancySensors" group of the "PresentModeController" controller.

12.3.3 Controllers

Controllers in a lighting system are specified in the Control instance but inherit their functionality from a template. In the transformation to UPPAAL, each controller of a

Control instance is translated to a single automaton. The example has a single controller, "PresentModeController" implementing the "PresentMode" template.

The state machine for a specific controller is similar to the state machine as specified in the template. Controllers in UPPAAL, however, have two states for each state in a template, a concrete state, and an actuation state. Moving between concrete states (for example from "Off" to "On") goes via a (committed) actuation state. This actuation state is instantly left by a transition that broadcasts a certain state in the Template is going to be reached. Actuator automata use this broadcast to update the output level of luminaires.

Furthermore, the model of such a controller also models hold time, retention time and dwell time behaviors of states. The UPPAAL model uses an invariant on such a state together with a guarded transition back to the (actuation) "Off" state. If 600 time units (10 minutes) have expired since the last occupancy detection, only the transition back to the (actuation) "Off" state is enabled, and has to be taken instantly.

Confidentiality requirements prevent us from displaying the entire UPPAAL model of a controller.

12.3.4 Actuators and Actuator Groups

Actuator groups are not explicitly represented in the generated UPPAAL model, there is no automaton that implements a certain actuator group. Actuators, in UPPAAL, are represented by integer variables representing their output level.

When a controller reaches a (new) concrete state, it broadcasts a signal that such a state is reached. For each such state, for each luminaire affected by that specific controller, a small automaton is generated that receives this signal and updates the integer value accordingly. These automata are generated with respect to actuator groups, for the "Present" state in the example, two different automata are generated for the two different actuator groups.

To implement presentation mode, for each luminaire in the "FrontLuminaires" group an automaton like the one depicted in Figure 12.3 is generated (two times total). For each luminaire in the "RoomLuminaires" group, this automata is depicted in Figure 12.4. Recall that in such a presentation mode, the luminaires in the front of the meeting room have a low level (for example, 20 percent), whereas the other luminaires have a high level (for example, 80 percent).

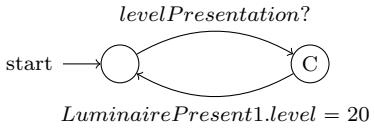


Figure 12.3: Automaton for actuation in presentation mode for actuator "LuminairePresent1".

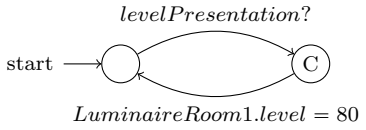


Figure 12.4: Automaton for actuation in presentation mode for actuator "LuminaireRoom1".

12.4 Implementation of Monitors and Rules

Implementation of monitors is straightforward, all monitors as outlined in Chapter 11 can be directly translated to UPPAAL. Rules are mapped to their CTL equivalents.

It is important to note that there are adaptations that need to be done in UPPAAL before such a system can be model checked. Most of these changes arise from the use of the State Machine language as intermediate. They involve the following steps:

1. Increase the priority of monitor automata over those that enact system behavior. While priorities are soon to be introduced in the State Machine language, the research concluded before its implementation was finished.

2. Fix (possible) uses of $<$, in the State Machine language only $\leqslant$ and $\geqslant$ are allowed, the monitor for the "Globally Invariance" pattern, however, uses $<$ and $\geqslant$.

3. Replace the (default) eager aware sensor simulators with their true random equivalents displayed in Figure 12.1.

4. Put the corresponding CTL query in (generated) UPPAAL XML either manually, or via the UPPAAL GUI. The State Machine language cannot be used to capture CTL queries, while we have defined how such an extension would be made for the State Machine language, time constraints prevented us from implementing such an extension.

All these changes can be seen as possible arguments against the use of the State Machine language, in Section 18.5 we will detail a possible alternative that better fits our needs.

12.5 Monitors for Atomic Propositions

Proposition monitors in UPPAAL use channel(s) to communicate that a proposition holds, is refuted or fires. A (particular) state proposition monitor, for example, broadcasts over a generated *holds* channel if that particular proposition starts holding, and over a generated *refutes* channel if it stops holding. For some event propositions, in particular sensor propositions, no extra monitor has to be added to the model. A sensor implementation broadcasts sensor events over a particular channel, the sensor in Figure 12.1, for example, sends a synchronization over the *sensorTheater* channel. Composite proposition monitors or pattern/scope monitors can directly synchronize on this particular channel when they are interested in this sensor firing.

12.5.1 Sensor (Group) Proposition

Sensors and sensor groups in the UPPAAL model broadcast over a channel when they have fired, we use these channels directly in our (composite/property) monitors. A sensor (group) proposition generates nothing, the generation to UPPAAL only notes that it can

observe a sensor firing by listening on that specific channel and furthermore notes that the sensor channel is "external", to prevent the generator from mangling its name.

12.5.2 Actuator Proposition

Actuator Propositions are monitored by the change of the variable that implements such an actuator. We allow specifying arbitrary levels in the Query language (even levels that cannot be achieved) and have constructed these monitors accordingly. That these monitors observe an arbitrary level is done under the assumption that eventually, dimming is also implemented as part of the lighting system[2] after which observing discrete levels ("Off", "On", etc.) is no longer enough. Furthermore, we assume that actuators can start at any value, forcing us to assume that the status of the state proposition is unknown.

Consider for example a monitor for whether a single luminaire is on level 80 as depicted in Figure 12.5. The monitor is straightforward except for the use of a "hurry" channel. Since we have modeled transitions between various output levels as guards on the variable of such an actuator, transitions only enable when a change in output level is detected. The hurry channel is urgent, meaning that if such a change is detected, UPPAAL has to take the transition with this hurry channel, forcing the proposition to be propagated. If the hurry channel was omitted, UPPAAL could delay changing the status of such an actuator proposition, resulting in an inconsistency between the system and the monitors. Important to note, however, except for forcing eager execution the hurry channel changes nothing in system behavior, it is a broadcast that is never received by any other automata in the system.

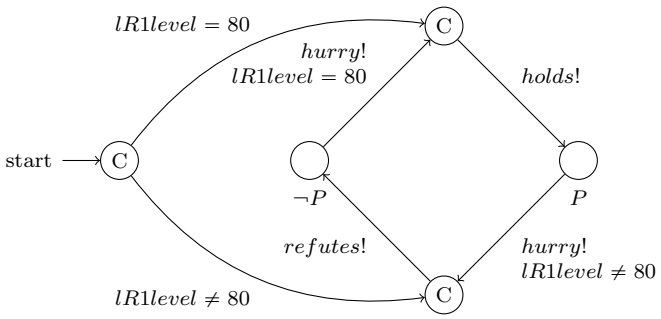


Figure 12.5: UPPAAL monitor for an Actuator Level Proposition that specifies that the "LuminaireRoom1" is on level 80.

12.5.3 Actuator Group Proposition

Actuator Group Propositions are much like Actuator Propositions, but instead monitor the levels of all luminaires in a particular actuator group. If *any* luminaire is not on the desired level the proposition is refuted. If *all* luminaires are on the desired level the

[2]In fact, dimming is currently being implemented.

proposition holds. Consider for example a monitor that monitors whether all luminaires of the "FrontLuminaires" group are on level 20 as shown in Figure 12.6.

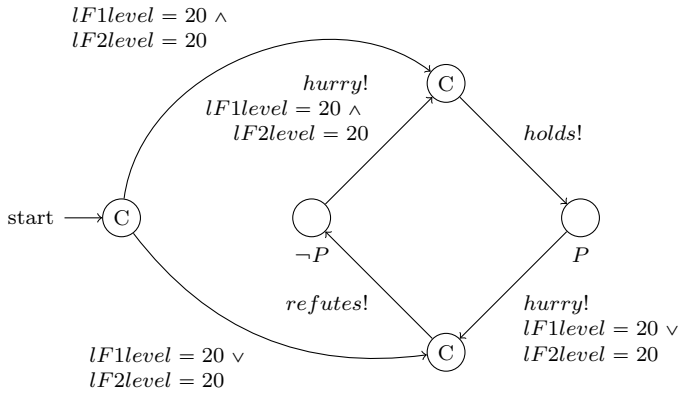


Figure 12.6: UPPAAL monitor for an Actuator Group Level Proposition that specifies that the "FrontLuminaires" group is on level 20.

12.5.4 Timer Proposition

Timer propositions fire after a specified amount of time since they have last been reset. While we classify Timer propositions as atomic, their reset can be any event proposition. The Query instance of Listing 10.1, for example, specifies a timer that fires every hold time of a particular controller (for example, every 600 seconds) unless it is reset by the begin of Presentation mode or any occupancy. If we assume that this composite reset proposition is observed on a channel $reset$, the corresponding Timer proposition monitor is given in Figure 12.7.

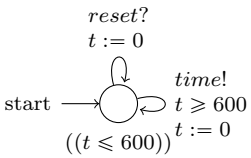


Figure 12.7: UPPAAL monitor for a timer that fires 600 time units after a reset.

12.5.5 Controller State Proposition

To monitor whether a controller is in a specific state we synchronize with the channels used to broadcast that the controller reaches any state. If a controller sends that it reaches the state specified in the proposition, this monitor broadcasts that the proposition holds, if the controller sends that any other state is reached, the monitor broadcasts that the proposition is refuted. Furthermore, a controller initially broadcasts that it reached its initial state before anything else, no special care has to be taken to model (possible) initial state.

Consider for example a proposition for the "PresentModeController" of the example that states that that specific controller is in Presentation mode. Its corresponding monitor is given in Figure 12.8.

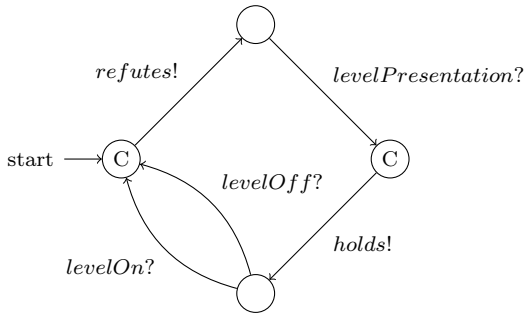


Figure 12.8: UPPAAL monitor for a Controller State Proposition that specifies that the "PresentModeController" is in presentation mode.

12.6 Monitors for Composite Propositions

Finally, we provide implementations for the various operators defined in the Query language with the notable exception of the assignment operator. Assignment is implemented statically during generation of the monitors for UPPAAL so needs no special attention.

12.6.1 Typecast Operator

Typecasting state to event propositions (on one of the flanks) is implemented statically. During generation, each state proposition is tracked as a pair of labels, one for when such a proposition holds, and one for when it does not. If such a state proposition is cast to an event proposition, the generator just discards the other label. Consider for example the Controller State proposition from Figure 12.8, if another composite operator or monitor is interested in the start of presentation mode, it can just synchronize on *holds*?.

12.6.2 Boolean Operators

Our language allows boolean AND as well as boolean OR, but the former only for state propositions. Due to how the language is parsed and the AST is constructed, boolean expressions always have two operands. For example, if a Query instance specifies `a or b or c`, we generate a monitor for `a or b` and one combining the result of such a monitor with `c`.

Boolean AND for State Propositions

A boolean AND expression only holds when both of the operands hold, the monitor for such a fact listens for holds (and refutes) signals of both operands, and if both operands hold, broadcasts that it holds too. If either of its operands stops holding, so does the

boolean AND construct. The monitor for such a boolean AND construct is displayed in Figure 12.9.

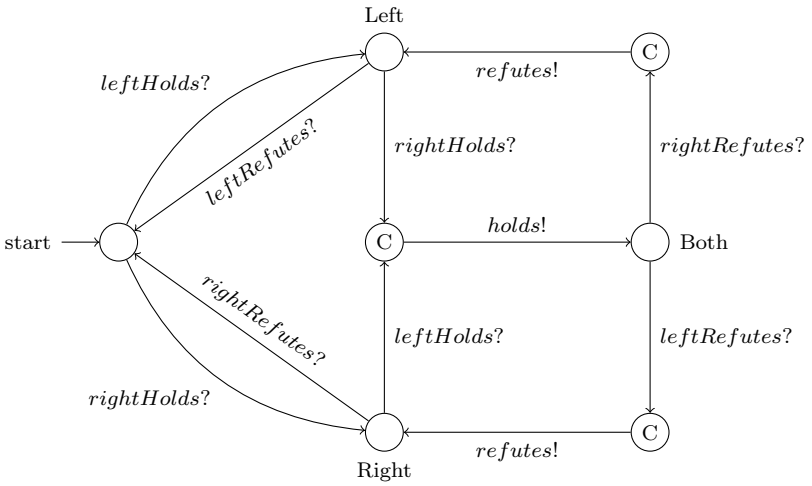


Figure 12.9: UPPAAL monitor to monitor that two state propositions (left and right) hold simultaneously.

Boolean OR for State Propositions

The monitor for boolean OR for state propositions is quite similar to the one for boolean AND, a notable difference is the location of the *holds*! and $refutes$! synchronizations. An OR expression holds if any of its operands hold, so once either of the two propositions holds, the *holds*! synchronization is fired. Similarly, only when both propositions have stopped holding, $refutes$! is broadcasted. This monitor is displayed in Figure 12.10.

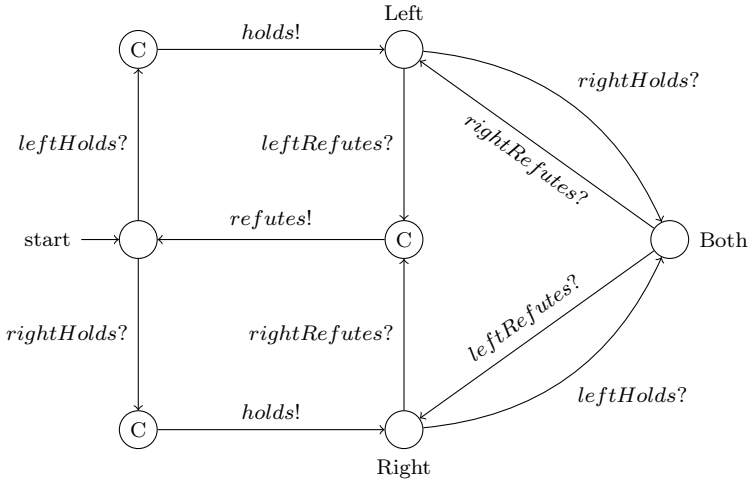


Figure 12.10: UPPAAL monitor to monitor that either of two state propositions (left and right) hold.

Boolean NOT for State Propositions

A boolean NOT expression holds when it operand does not hold, the monitor for such a fact listens for holds (and refutes) signals of both operands and sends out the inverse signal. The monitor for such a boolean NOT construct is displayed in Figure 12.11.

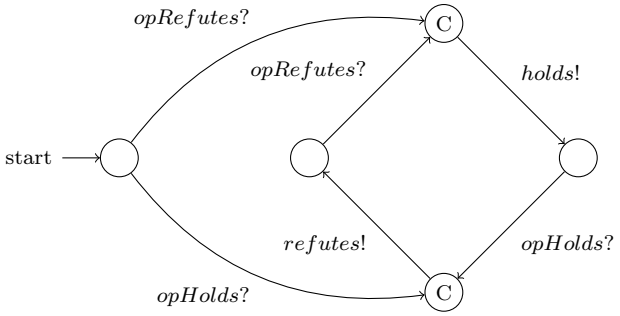


Figure 12.11: UPPAAL monitor to monitor that a state proposition does not hold.

boolean OR for Event Propositions

Finally, event propositions can be combined with boolean OR. Such a monitor is much like the monitor for a sensor group and depicted in Figure 12.12.

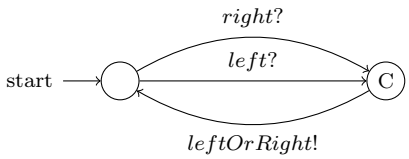


Figure 12.12: UPPAAL monitor to monitor that either of two event propositions (left or right) have fired.

12.7 Interaction between Various Monitors

As briefly mentioned, these monitors act on top of the UPPAAL model. The lowest level of monitors, atomic proposition monitors, are used to relate the model to the framework of monitors implementing a property observer. These atomic proposition monitors translate observed propositions in either event- or state channels. Between these atomic proposition monitors and the pattern/scope monitor implementing the property are composite monitors that can combine propositions to form new propositions. On top of the hierarchy is the pattern/scope monitor, that observes (composite/atomic) proposition monitors and follows them blindly. Finally, the rule instance decides whether a pattern/scope monitor observes an error.

This layered approach allows us to extend any of these three parts making up a property specification without having to alter any other. Implementing a new boolean operator (for example, boolean implication) does not alter atomic proposition monitors nor pattern/scope monitors. We believe that this systematic approach to implementing property observes in UPPAAL leads to a highly extensible framework for both the lighting domain and other domains in which such a language is of use.

13 Translation to the Co-simulation Framework

The Prisma project heavily focuses on simulation of various kinds. Simulation, compared to formal verification, has different strengths and weaknesses. The prime strength, but also the prime weakness of formal verification, is its exhaustive evaluation of all possible paths through a model. If a violation of a property exists, a formal verification tool will find such a violation given enough time and memory. However, this exhaustive search comes with scalability issues, the larger a lighting system becomes, the longer formal verification of properties takes. Increasing the system size in formal verification exponentially increases the possible states the model of such a system can be in and therefore exponentially increases verification times. Simulation, on the other hand, is much more like testing: it evaluates a scenario and concludes whether a violation of a property has occurred in that scenario.

Furthermore, formal verification is intangible. The feedback of a formal verification tool (often an error trace) is a list of actions that reproduce an error. Visualization techniques can be used to produce a tangible display of such an error.

TNO-ESI provides both a simulation and visualization environment for lighting systems. The simulation framework (called Co-simulation Framework) is configured by a transformation from the lighting systems model (instances of various DSLs) to produce the specified behavior. We have extended this simulation framework, and to a lesser extent the visualization environment, to support displaying property results. For validating large buildings, we advise that one first formalizes properties into instances of the Query language and uses simulation to gain an understanding of the system under test. If after substantial simulation no violations have been detected, formal verification can be used to prove that such the system is indeed without errors.

This chapter is structured similarly to the previous one. Firstly, we discuss the Co-simulation Framework as our target. Secondly, we discuss how we have implemented observers for atomic propositions, composite propositions and monitor/rule combinations for each pattern/scope in this Co-simulation Framework. Thirdly, we discuss in detail how the judge/verdict implementation for co-simulation was created and how visualization was extended to support verdicts. Finally, we provide an overview of a transformation to the Co-simulation Framework.

13.1 Co-simulation Framework

The Co-simulation Framework (often referred to as *cosim*) is an in-house developed simulator framework. This framework aims to allow various simulators to cooperatively simulate a system. For example, in a lighting system, there may be a model for network behavior written in POOSL, whereas the continuous function used to change output levels of a single luminaire is implemented in Matlab/Simulink.

Central to the Co-simulation Framework is a publish-subscribe network that connects various simulators. This network acts as a gateway. A simulator *publishes* messages that the framework distributes to all the *subscribers* of such a message. Furthermore, the Co-simulation Framework is the arbiter of time in simulation. It provides a canonical time that should be respected by all simulators co-simulating a system.

While co-simulation itself is a concept, there is also a reference implementation for lighting systems. This implementation, named JCoSim and written in Java, uses the concepts of such a Co-simulation Framework to simulate a lighting system's behavior. Within this implementation, there are different simulators for sensors, actuators, sensor groups, actuator groups and controllers. Common to all these simulators is that they publish messages and subscribe to messages of other simulators. The implementation of lighting systems is currently being extended, a Network model written in POOSL is being created and connected as well as a Java implementation of occupancy profiles.

Connecting to such an implementation is a matter of subscribing to the correct messages and is similar to how broadcast channels would work in UPPAAL. The implementation produces messages and sends them to all subscribers. If these subscribers produce a new message as a response, this message is broadcasted until no messages can be processed. During this cascading messages phase, no time is progressed. For the purpose of the implementation of the Query language, these messages are the only observable behavior of the simulation, the messages being broadcasted:

SetpointOutputLevelMessage A message from controllers to actuators that specifies the desired output level for actuators.

AchievedOutputLevelMessage A message from actuators that specifies that a certain output level has been reached. This message was newly added to distinguish between intended and actual output. If for example, a luminaire fades between an old level and a newly requested level using a continuous function, it could continuously send new achieved output levels.

ButtonPressedMessage A message that states that a specific button (group) has been pressed.

OccupancyDetectedMessage A message that states that a specific occupancy sensor (group) has detected occupancy.

ControllerStateMessage A newly added message that states that the controller has reached a specific state. This message was added because of Controller State propositions that would otherwise have no way to observe in which state a controller is.

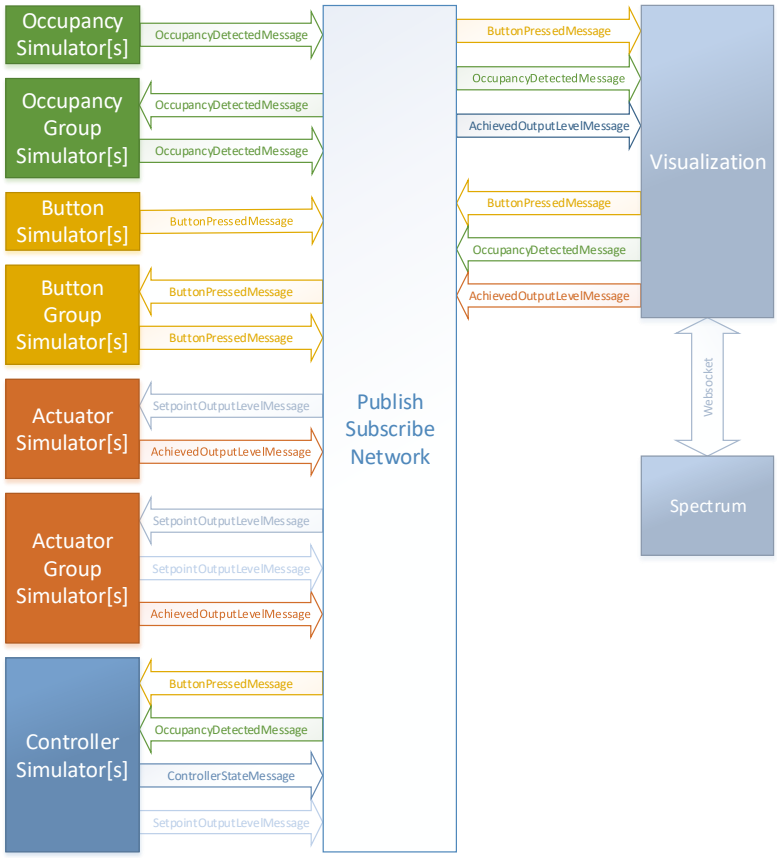


Figure 13.1: Framework of simulators used to implement a lighting system in JCoSim.

The entire pre-existing framework of types of simulators is given in Figure 13.1. A lighting system implementation in JCoSim consists of many instances of simulators displayed on the left.

13.2 Query language Implementation for JCoSim

To implement the Query language in such a framework of simulators we have used a similar approach as to the one used for UPPAAL. We have added simulators to the pre-existing approach that observe the lighting system. These (atomic/composite) proposition simulators interact similarly to their UPPAAL counterparts: atomic proposition simulators observe the lighting system, composite propositions simulators observe atomic propositions and monitors observe (atomic/composite) propositions. Finally, these monitors are observed by a judge that uses its rules to reach a verdict.

13.2.1 Atomic Proposition Simulator

Atomic proposition simulators are the entry point to the Query language implementation. These simulators are the only simulators directly connecting to other simulators implementing a lighting system. These simulators act as translators, they receive for example a ButtonPressedMessage, and translate this to an EventPropositionMessage used internally by the proposition monitors. As such a publish-subscribe network is distinctly event based, updates of state propositions are only sent on flanks.

Consider for example the implementation of the ActuatorLevelProposition, in JCoSim, this proposition is implemented as a simulator subscribed to all AchievedOutputLevelMessages. When receiving such a message, the simulator checks that the actuator is indeed the one it is interested in and compares the newly broadcasted level with its desired state. If the state of its a proposition changes, this simulator sends a StatePropositionHolds/Refutes message over the publish-subscribe network when the proposition starts or stops holding respectively.

13.2.2 Composite Proposition Simulators

Composite proposition simulators are used to observe composite propositions. In JCoSim these are also implemented as simulators that subscribe to the (atomic/composite) sub-propositions and publish messages for the combination of these sub-propositions.

Consider for example boolean AND for states. Such a simulator subscribes to StatePropositionHolds/Refutes messages and tracks the states of all state propositions it is interested in. Just like the ActuatorLevelProposition simulator, once it detects change in the validity of the observed proposition (from holds to refutes or vice versa), it sends a StatePropositionHolds/Refutes message accordingly.

13.2.3 Monitor Simulators

In JCoSim, there are specialized state machines used to model the state of a controller. These state machines, and their underlying semantics, are so specific to the controller that repurposing this framework for the use of monitors is infeasible. Instead, we have added a new (generic) timed automata model as well as a corresponding execution environment to JCoSim. This execution environment is both discrete and eager, time can only progress in discrete steps and as soon as a transition is enabled, it will be taken. Furthermore, nondeterminism is implemented by choosing an arbitrary transition possibly leading to different executions for the same scenario.

The model of timed automata is much like that of UPPAAL but simplified. A transition in such a model can have a synchronization (both in and out), clock reset or simple clock guard (comparing a single clock with a fixed value). While easily extended to support variables, boolean operators in guards etc. we have not needed any of these concepts to implement the monitors of the generic translation.

Consider for example the monitor for "Between Existence" with `enabler` as event proposition opening the scope, `disabler` as event proposition closing the scope and `signal` being

the event proposition that should exist within that scope. The implementation of such a monitor is given in Listing 13.1.

```
// States of the monitor
public enum BetweenExistenceState implements MonitorState {
  Initial, Sensitive, Exists, ERROR
}

new Monitor<>(BetweenExistenceState.Initial,  // Initial state
  Arrays.asList(                              // List of transitions
    new MonitorTransition<>(
      BetweenExistenceState.Initial,          // Source state
      BetweenExistenceState.Sensitive,        // Destination state
      enabler                                 // Incoming sync
    ), new MonitorTransition<>(
      BetweenExistenceState.Sensitive,
      BetweenExistenceState.Exists,
      signal
    ), new MonitorTransition<>(
      BetweenExistenceState.Exists,
      BetweenExistenceState.Initial,
      disabler
    ), new MonitorTransition<>(
      BetweenExistenceState.Sensitive,
      BetweenExistenceState.ERROR,
      disabler
    )
  )
);
```

Listing 13.1: Monitor implementation for Between Existence in JCoSim

The monitor simulator is then responsible to subscribe to all propositions used in such a monitor and sends messages over the publish-subscribe network indicating when it has reached a new state. For the Judge simulator, only the current state of a monitor is observable.

13.2.4 Rules, Verdicts and Judge Simulators

In UPPAAL rules are implemented as CTL property with the query evaluator acted as the judge. This judge in UPPAAL concludes for each property whether it holds or otherwise produces a counterexample. JCoSim however, has no notion of such a judge and has to be extended. Implementing this judge was not without issue, the rules specified in the monitor/rule combinations are of two kinds: safety and liveness rules. Safety rules have finite counterexamples, so in simulation it is possible to conclude that such a rules have been violated. Liveness, however, does not have finite counterexamples. Imagine for example a rule that states that a specific monitor state is eventually reached, how long do we have to simulate a lighting system to assert that such a rule holds? For such a rule

we could say that we do not know whether the property is true until we have observed the desired state. Now imagine a rule that asserts that an occurrence of a single state has to be responded to by the occurrence of another, how often do we have to observe this responding state before we can assert that such a rule holds?

These problems arise from the unique requirements for a judge in JCoSim, we want to use the judge to produce a continuous stream of verdict messages that indicate the validity of a property at this point of the simulation. UPPAAL only produces such a verdict after evaluating a potentially infinite path and can, therefore, be absolute. These verdicts for simulation fall into four categories:

Error indicates that a rule has been violated.

OK indicates that a rule can no longer be violated.

Maybe Error indicates that a property has not yet been violated but if the current situation remains indefinitely, the rule is violated.

Maybe OK indicates that a property has not yet been violated and if the situation remains indefinitely the rule is not violated.

Rule instances in JCoSim observe states of monitors and produce verdicts in the following ways:

Forbidden Rule produces the Maybe OK verdict before the forbidden state has been reached and Error after.

Desired Rule produces the Maybe Error verdict before the desired state has been reached and OK after.

Response Rule produces the Maybe OK verdict before a cause happened, and Maybe Error if a cause occurred, but has not yet been responded to by the effect occurring.

13.3 Extensions to Spectrum

Spectrum, the lighting system visualization tool, has been extended to support displaying verdict messages for specific controllers. Controllers in Spectrum are visualized by drawing an overlay over its sphere of influence, the smallest rectangle spanning all actuator/sensors in the actuator/sensor groups of a controller.

In JCoSim, an extra simulator is responsible for maintaining the connection with Spectrum. This visualization simulator connects to the publish-subscribe network of JCoSim, subscribes to actuator/sensor/button messages and sends these updates over a websocket connection to Spectrum. The connection is bidirectional, clicking on a sensor in Spectrum also sends the corresponding OccupancyDetectedMessage over the publish-subscribe network. Added to this visualization simulator is that it now subscribes to verdict messages and sends proper messages to Spectrum to indicate whether a controller

has violated a property. This controller is then colored red if the verdict is Error, green if the verdict is OK, red-yellow if the verdict is Maybe Error and green-yellow if the verdicts is Maybe OK.

As a final consideration, given a property specified in a Query instance, it is not always evident to see over which controller such a property is reasoning. As a compromise, we compute the set of affected controllers of a given property by evaluating its propositions. If a proposition is of SensorSignal type, all controllers in which such a sensor is mapped are affected. If a proposition is of ActuatorLevel type, all controllers in which that actuator is mapped are affected, etc. For composite operators, we merge the sets of affected controllers. This heuristic is often good enough, but if a sensor is used in a Query instance and mapped by all controllers (for example, single building-wide daylight sensor), all controllers are in the affected set. In the future, this heuristic may need to be changed, but currently suffices.

13.4 Implementation of a Full Property Specification

All proposition simulators, monitor simulators, rules and verdict simulators have been implemented as a library in JCoSim. During generation of a Query instance, these building blocks are combined to create an observer for the desired property, for example, the property specification as given in Listing 10.1 generates configuration for JCoSim displayed (in simplified form) in Listing 13.2. The implementation resembles the abstract structure of Figure 9.1 quite well, for each atomic proposition or composite operator a new simulator is generated and each arrow is represented by an instance of either `EventProposition` or `StateProposition`.

The latter part of the listings shows monitors and judges. In JCoSim we have a library of pattern/scope implementations in monitor/rule instances, in this example we initiate these by binding `EventPropositions` or `StatePropositions` to their transitions. The monitor is then simulated by a monitor simulator subscribed to the proposition messages. The judge uses the rule and receives state updates of the monitor simulator to produce a verdict over the verdict channel.

```
// ControllerState ctrl1 :: Presentation
StateProposition pmcPresent = new StateProposition("pmcPresent");
new ControllerStatePropositionSimulator(
  pmcPresent,
  "PresentModeController", "Present"
);

// SensorGroupSignal ctrl1 :: Buttons1 Button
EventProposition pmcButtons1 = new EventProposition("pmcButtons1");
new SensorGroupPropositionSimulator(
  pmcButtons1,
  "PresentModeControllerMeetingRoom1TemplateButtons1", "button"
);
```

```
// SensorGroupSignal ctrl1 :: Sensors1 Occupancy
EventProposition pmcSensors1 = new EventProposition("pmSensors1");
new SensorGroupPropositionSimulator(
  pmcSensors1,
  "PresentModeControllerMeetingRoom1TemplateSensors1", "occupancy"
);

// Start presentMode
EventProposition pmcPresentStart
  = new EventProposition("pmcPresentStart");

new StatePropositionAdaptorSimulator(
  pmcPresentStart,
  pmcPresent, StatePropositionStatus.HOLDS
);

// Start presentMode or anyOccupancy
EventProposition pmcPresentStart_or_pmcSensors1 =
  new EventProposition("pmcPresentStart_or_pmcSensors1");

new EventOrPropositionSimulator(
  pmcPresentStart_or_pmcSensors1,
  Arrays.asList(
    pmcPresentStart,
    pmcSensors1
  )
);

// Timer ctrl1 :: HoldTime Reset { Start presentMode or anyOccupancy }
EventProposition timer =
  new EventProposition("timer600_reset_pmcPresentStart_or_pmcSensors1");

new TimeOutPropositionSimulator(
  timer,
  pmcPresentStart_or_pmcSensors1,
  600
);

// Start presentMode
EventProposition pmcPresentStart2 =
  new EventProposition("pmcPresentStart2");

new StatePropositionAdaptorSimulator(
  pmcPresentStart2,
  pmcPresent, StatePropositionStatus.HOLDS
);

// End presentMode
EventProposition pmcPresentEnd = new EventProposition("pmcPresentEnd");
new StatePropositionAdaptorSimulator(
  pmcPresentEnd,
  pmcPresent, StatePropositionStatus.REFUTES
);
```

```
// anyButton or timeout
EventProposition pmcButtons1_or_timer =
  new EventProposition("pmcButtons1_or_timer");

new EventOrPropositionSimulator(
  pmcButtons1_or_timer,
  Arrays.asList(
    pmcButtons1,
    timer
  )
);

// Between { Start presentMode }
// And { End presentMode }
// Existence { anyButton or timeout }
QueryChannel qr1 = new QueryChannel("qr1");
Pattern<BetweenExistenceState> qr1pattern
  = new BetweenExistence(pmcPresentStart2, pmcPresentEnd,
      pmcButtons1_or_timer);

new MonitorSimulator<>(
  qr1pattern.getPatternMonitor(),
  qr1
);

VerdictChannel qr1verdict = new VerdictChannel("qr1verdict");
new JudgeSimulator<>(
  qr1pattern.getRule(),  // Rule instance belonging to Between/Existence
  qr1,                   // Incoming monitor state message channel
  qr1verdict,            // Outgoing verdict message channel
  new HashSet<>(         // Affected controllers
    Arrays.asList("PresentModeController")
  )
);
```

Listing 13.2: Generated configuration for JCoSim implementing the Query instance of Listing 10.1

Part IV

Validation

Part Outline

In Chapter 3, we have presented the requirements of the proposed Query language, the language has to be be easy to use, complete, verifiable, traceable, integrated and extensible. All these aspects were taken into consideration during the creation of the language as outlined in Part III. In this part of the report, we describe the steps taken to validate that the Query language implementations safeguard these aspects. To do so, we have executed three forms of validation, legitimacy analysis, usability analysis and extensibility analysis in three chapters:

Chapter 14 outlines the demonstration we have given to Philips Lighting to show the value of the Query language for the lighting domain. We have shown a large scale lighting system and demonstrated that the use of the Query language significantly improved error discovery.

Chapter 15 outlines the usability analysis we have conducted in order to gain understanding on whether the Query language is easy to use and integrated. We have conducted this analysis using a workshop on specifying using pattern libraries, mutating Query language instances and writing instances from scratch.

Chapter 16 outlines analysis of extensibility in three ways. First of all, we demonstrate how propositions can be added for newly modeled aspects of lighting systems. We demonstrate how new pattern/scope combinations can be added to the generic Query language and finally we demonstrate how the Query language can be used in other domains.

14 Legitimacy of the Query language

The Prisma project aims to outline a domain modeling approach for the development process of professional lighting systems. The responsibility of implementing such a domain approach for real-world lighting system development remains with Philips Lighting. The Prisma project has also created a set of languages, but only to prove concepts to Philips Lighting. For the steering board meeting of the Prisma project, we have developed a demo showing how the concept of early validation of lighting system works in such a domain approach. This early validation makes heavy use of the Query language described in this report.

14.1 Steering Board Meeting Demonstration

This demo was given during the so-called "steering board" meeting. Such a meeting involves management of the industrial partner of a TNO-ESI project (Philips Lighting for Prisma) and project members. The board meeting aims to "steer" the project's direction but also show progress and prove the value of the project. The aim of the demonstration was to show Philips Lighting that if a domain approach is pursued, that the created domain model can be analyzed during development. The Query language then, is a tool to specify correct behavior and can be used to find errors in models early in the development process. The goals of this demonstration were to:

1. Show that large scale analysis of behavior of lighting systems is infeasible by hand and that appropriate tooling is required.
2. Present how the proposed Query language aids in system property specification and demonstrate that the Query language is both easy to use and provides precise specification capabilities.
3. Show how the generated monitors can find an error in a lighting system using simulation.
4. Show that the problem can be isolated and model checked and indicate what the source of the problem was.

14.1.1 Lighting System

For this demonstration, we did not only want to show that the generated monitors of a Query instance can check that a property is violated, but also show that there is a need for property checking in lighting systems. To do so we created a building that is relatively

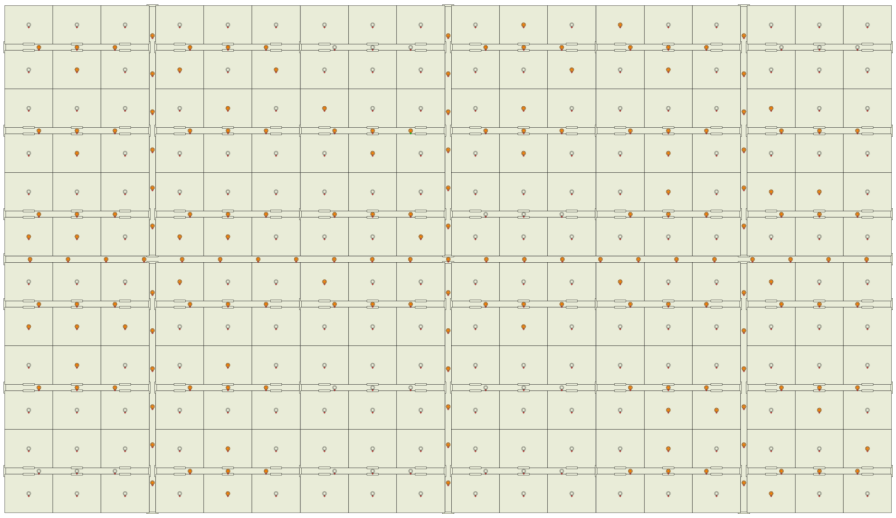


Figure 14.1: Lighting System visualization of the example system used for the steering group demo.

large (over 300 luminaires/sensors) with an error that is rare. Finding a violation of the property visually is practically impossible.

The lighting systems itself consisted of offices and corridors, each corridor is linked to six offices and there are 36 of these instances in the demonstrated building. In offices, the time before luminaires in a vacant office turn off, the hold time, is long (ten minutes). In corridors, this hold time is short (one minute). Furthermore, if the luminaires in an office are on, so are the luminaires in the connected corridor. This system is shown in Figure 14.1.

The property given as part of this demonstration asserts that if a sensor in the corridor detects presence, the luminaires in that corridor are on for at least a minute. If such a property would not hold, it could be that as soon as the corridor becomes vacant, the luminaires turn off immediately. While not directly a threat to safety, it does look strange to end users.

Finally, for the purpose of this demo, we have created a faulty system that does not respect this property. The faulty implementation is erroneous in the interaction between offices and corridors implemented in the "Corridor" template. Instances of this template listen to the input of the sensors in the surrounding offices and if they detect presence, enter an override state called "Office On". This "Office On" state is much like the "On" state of the office guaranteeing that if any of the connected offices are in an "On" state, so is the corridor. Problematic in this "Office On" state is that sensors of the corridor specifically are ignored.

```
Queries invariantLight

with each controller implementing Corridor as ctr {
  let anySensors : Event = SensorGroupSignal ctr::CorridorSensors
      Occupancy
  let corridorLevel : State = ActuatorGroupLevel ctr::Luminaires level
      ctr::On

  query Globally
          Invariance corridorLevel For ctr::CorridorHoldTime
            When anySensors
}
```

Listing 14.1: Query instance for steering board meeting demo.

14.1.2 Property Specification in the Query language

After showing this faulty lighting system without automatic checking (just as in Figure 14.1), we outlined how such a property would be specified in the Query language. To do so one has to select the right pattern/scope combination and combine these with correct (atomic/composite) propositions. For specifying this particular property, we have used the "Invariance" pattern with "Globally" scope. This pattern says that after the occurrence of a certain event, if state property holds, it holds for at least a certain amount of time. This pattern has to hold throughout the entire execution of the system and is therefore Globally-scoped. The corresponding Query language instance is given in Listing 14.1 and asserts that after a sensor event of the corridor sensors, the luminaires in that corridor have the "On" level for at least the "CorridorHoldTime".

14.1.3 Error Discovery and Identification Process

For validation of such a property, we can use the transformation outlined in Chapter 13 to generate an executable simulator with the same trace as used in the blind simulation. This framework of simulators discovers the fault after about 20 simulated minutes. Visually, such a group is colored red as displayed in Figure 14.2.

Although we now know that the system is faulty, we have not found the source of the error. To do so we have isolated a single corridor/room group of which we generated an UPPAAL model by the transformation outlined in Chapter 12. UPPAAL finds this violation instantly.

Finally, such an error can further be analyzed, the provided counter example found by UPPAAL shows that it can reach the "Office On" state by sending the *office* synchronization, wait anywhere between 540 and 600 seconds, and then send the *corridor* synchronization, the property is then violated at $t = 600$. Visually, such a counterexample is displayed in Figure 14.3. In this depiction, actors enter and leave rooms at set times as displayed as figurines without and with crosses. In this particular counter

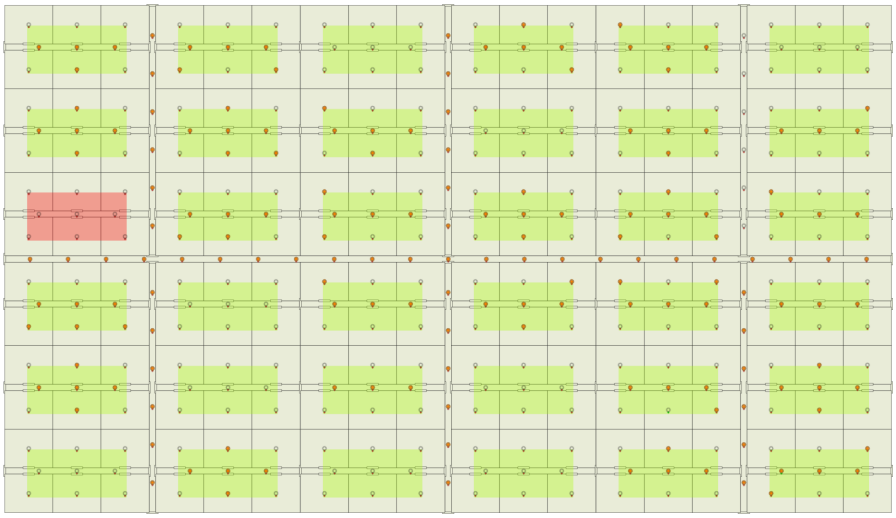


Figure 14.2: Lighting System visualization with verdict visualization, single corridor with six adjacent offices has violated the stated property.

example, the amber actor leaves the corridor exactly ten minutes after the blue actor left the connected office, causing the luminaires in the corridor to turn off immediately.

14.1.4 Correct Lighting System Implementation

The root cause of the error lays in the "Office On" state of the corridor template, as it does not listen to sensors of the corridor group. Resolving the issue requires to listen to office and corridor sensors at the same time and individually track when they have last detected occupancy. Due to confidentiality requested by Philips Lighting we are unable to display the corrected state machine.

14.2 Philips Lighting Feedback

The steering board of the Prisma project appears to be convinced of the value of the Query language. The message that an automated way of verifying correctness is needed for large scale lighting systems is well received.

The management raised questions regarding scalability. Formal verification techniques are often disregarded as being theoretical and similar remarks were made during the steering board meeting. The question was raised whether the fault could still automatically be found in a large scale lighting system. We noted, partly based on the research by Doornbos et al. (2015), that finding an error in formal verification is often easier than finding that a system is correct and that if proven infeasible it is still possible to isolate parts of the lighting system for formal verification.

Furthermore, praise was given that the Query language is a single source of property specifications. The steering board saw value in having a single Query instance for both

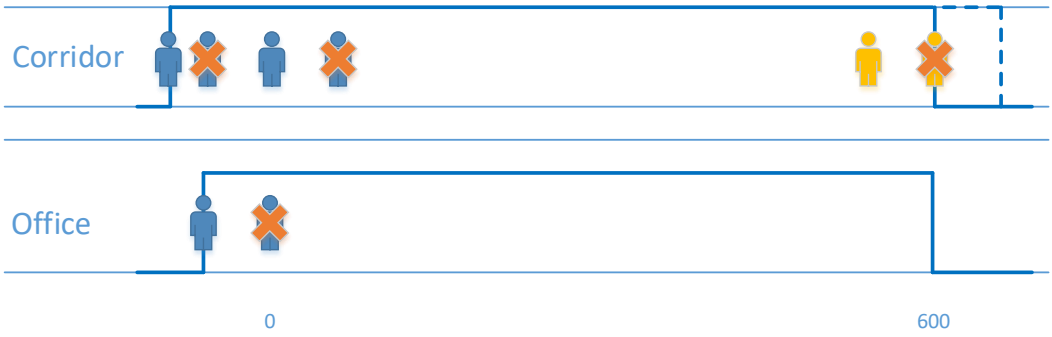


Figure 14.3: Counterexample for the property specified in Listing 14.1

simulation and formal verification.

Finally, the question was raised how many properties would be needed to fully specify a lighting system. We noted that it depends on the number of unique templates employed in the lighting system. While unable to put a direct number on it, we furthermore noted that property specifications are largely reusable if the same templates are used for different lighting systems.

15 Usability of Query Language

To validate whether the Query language is usable, we have held a workshop on writing property specifications using the Query language. The workshop was structured as an interactive lecture providing a mix of theory and practice. Required concepts were discussed in just in time and top down fashion.

The participants were all remaining members of the Prisma team except the direct supervisor of this particular research for a total of five participants. All participants have seen the Query language before, but have no intricate understanding of its concepts. None of the participants has been involved directly in discussions about its syntax and semantics but all of them have seen example instances in for example the steering board meeting. All participants have encountered general purpose languages before and should therefore already be familiar with C-like syntax.

15.1 Goals of the Workshop

For this particular validation, we were looking to obtain feedback on the usability of the Query language. We have defined four sub goals to obtain an answer to the general usability question:

Test pattern/scope understanding The principle idea underpinning the Query language is that of pattern libraries. By discussing pattern/scope combinations, we aim to comprehend how well pattern/scope combinations are understood. Furthermore, by discussing corner cases we aim to find what patterns or scopes require more explanation.

Test instance understandability The Query language is used in simulation and formal verification, but can also be used as a conversational tool. As the Query language forces precise specification of requirements, the language can become ground truth of a particular template. We believe therefore that the language has to be readable to a wide range of stakeholders, possibly wider than those able to mutate or create instances.

Test instance mutability The Query language aims to provide a language for property specifications of all properties in lighting systems. With different lighting systems come different requirements. We have evaluated how difficult updating property specifications is perceived.

Test instance creation With the development of new templates or with the addition of new models to the approach comes the need to specify new Query language

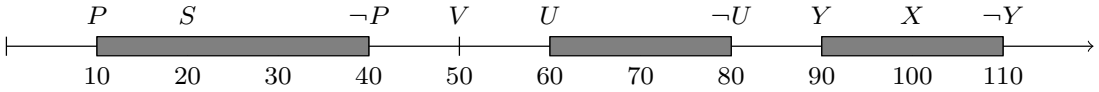


Figure 15.1: Example timeline used in discussing the semantics of the Invariance pattern.

instances. We have evaluated how difficult writing property specifications is perceived.

In this workshop, we have focused on concepts of the Query language instead of its syntax and semantics. While we think there is value in asking domain experts to produce (syntactically and semantically) correct instances of the Query language, writing a first property specification using the language is a process of trial and error. By discussing its concepts first, we gain understanding whether the ideas of the language are indeed correct. We argue that evaluating its concrete form is a next step to validate the implementation of the concepts.

15.2 Exercise One: Pattern/Scope Combinations

The first set of exercises dealt with patterns and scopes for (abstract) propositions of type state and event. In these exercises, we presented a timeline with events occurring and states holding. The exercise was to determine whether a particular (given) combination of pattern and scope holds for the particular trace. After brief considerations, the answers to the questions were discussed. Consider for example the timeline of Figure 15.1, do the following properties hold for this particular execution?

1. Globally: Invariance P for 10 When S
2. Globally: Invariance U for 10 When V
3. Globally: Invariance Y for 10 When X

In this particular example, all three properties hold, the first property holds trivially: when S occurs, P is already holding and remains holding for at least 10 time units. The second property also holds, while U does not hold when V occurs, the next time it does hold, it holds for at least 10 time units. In the final property, Y holds for exactly 10 time units after X occurs and therefore precisely holds. All examples discussed in the workshop can be found in Section C.1.

15.3 Exercise Two: Understandability and Mutability

Between the first and second exercise, we have discussed the set of domain specific atomic propositions, the set of implemented composite operators and the overall (but high level)

syntax of the Query language. In the second exercise, a template is discussed that specifies "Open Plan" behavior. Each *workspace* in an open plan has an own luminaire, sensor and controller. If a particular sensor of a workspace detects occupancy, the luminaire of that particular workspace goes to "On" level. If any other sensor in the open plan detects occupancy, this particular workspace goes to "Background Level".

For this particular example template, three tasks were given:

1. Write down in natural language what the Query language instance of Listing 15.1 specifies.[1]

2. Alter the Query language instance so that instead of specifying that a controller is in a particular state, the appropriate luminaires are on the "On" level.

3. Alter the Query language instance in such a way that instead of specifying the expected behavior under local occupancy, it specifies the expected behavior under remote occupancy.

[1] Assuming the (second) Universality semantic outlined in Section 11.3.3

```
with each controller implementing Workspace as ws {
  let anyLocalSensor = SensorGroupSignal ws::LocalSensors Occupancy
  let localHoldTime = Timer ws::HoldTime Reset anyLocalSensor

  let localOnState = ControllerState ws::On

  query After anyLocalSensor Until localHoldTime
          Universality localOnState
}
```

Listing 15.1: Example property specification for "Open Plan" lighting system

```
with each controller implementing Workspace as ws {
  let anyLocalSensor = SensorGroupSignal ws::LocalSensors Occupancy
  let localHoldTime = Timer ws::HoldTime Reset anyLocalSensor

  let localOnLevel = ActuatorGroupLevel ws::LocalLuminaires
                       level ws::LevelOn

  query After anyLocalSensor Until localHoldTime
          Universality localOnLevel
}
```

Listing 15.2: Solution to the second task: Correct actuation level instead of controller state

```
with each controller implementing Workspace as ws {
  let anyRemoteSensor = SensorGroupSignal ws::RemoteSensors Occupancy
  let remoteHoldTime = Timer ws::HoldTime Reset anyRemoteSensor

  let localActiveLevel =
    ActuatorGroupLevel ws::LocalLuminaires level ws::LevelOn or
      ActuatorGroupLevel ws::LocalLuminaires level ws::LevelBackground

  query After anyRemoteSensor Until remoteHoldTime
          Universality localActiveLevel
}
```

Listing 15.3: Solution to the third task: Property specification for correct behavior when remote occupancy is detected

The first task asserts that property specifications in the Query language can be understood. The specified property states that after local occupancy is detected, until the local hold time has expired, the controller is in the "On" state for all controllers implementing the workspace template.

The solutions to the second and third task are given in Listing 15.2 and Listing 15.3 respectively. In the second task, the participant is required to identify which part of the Query instance specification has to be altered: the `localOnState` declaration, which atomic proposition to choose: ActuatorGroupLevel and correctly instantiate the atomic proposition with correct actuator group and level. In the third task, participants had to alter the property specification to assert correct output levels under remote occupancy. Participants had to realize that local occupancy overrides remote occupancy and therefore during remote occupancy, luminaires can either be "On" or "Background" level. Finally, the participant was required to correctly identify that boolean OR can be used to combine these two actuation level propositions.

15.4 Exercise Three: Free Form Property Specification

In the final exercise, we have evaluated a template with dwell time.[2] When occupancy is detected, the controller first goes to a "Dwell" state that specifies an intermediate level for all luminaires. Only if after spending at least the dwell time (for example 10 seconds) and at most the hold time of the dwell state (for example 30 seconds) re-detected occupancy leads to the actual "On" state. The participants were required to construct two property specifications:

1. When going from "Off" to "On", the "Dwell" state is passed.
2. When going from "On" to "Off", the "Dwell" state is not passed.

The solutions to these tasks are given in Listing 15.4. The participants were required to identify that ControllerStatePropositions can be used to detect entrance/exit of states. Furthermore, using the After/Until scope and Existence pattern, a property can be specified that asserts the entrance of the intermediate dwell state. The second property specification is the same except for the use of the Absence pattern.

[2] This system is not describable in the Prisma templates of version 1.0 (on which this research is based), they are describable in version 1.1. As of writing, version 1.1 templates have no working translations to UPPAAL/JCoSim.

```
with each controller implementing DwellWorkspace as dws {
  let offState = ControllerState dws::Off
  let onState = ControllerState dws::On
  let dwellState = ControllerState dws::Dwell

  // First task
  query After {End offState}
          Until {Start onState or Start offState}
            Existence {Start dwellState}

  // Second task
  query After {End onState}
          Until {Start offState}
            Absence {Start dwellState}
}
```

Listing 15.4: Solutions to both free form property specification tasks.

15.5 Evaluation of Goals

Pattern/scope understanding Participants noted that specifying using pattern/scope combinations is a different way of thinking and showed that the concept was initially not grasped. However, later on in the exercises, participants started observing edge cases and correctly deduced which pattern/scope combinations had to hold. While there is a learning curve involved in specifying using pattern/scope combinations it appears shallow. Edge cases were not perceived as correct or wrong, but participants felt that they should be well addressed in accompanying documentation.

Instance understandability The example property specification was easily understood by the participants, the correct interpretation was given within a few minutes and without help.

Instance mutability Participants properly understood the structure of proposition declarations and property specifications. Participants had no trouble identifying which proposition to replace for the first task and correctly chose the actuator group proposition. In the second task, we expected participants to (wrongly) specify that during remote occupancy, the luminaries are on the "Background" level, but participants quickly realized that luminaires can also be on the "On" level. Feedback on both tasks was positive, mutating queries was considered doable.

Instance creation The final exercises were cut short due to timing constraints, but participants did try the first of the two property specifications. Some participants got stuck trying to use timing in their property specification or tried to use sensor or actuator propositions to derive state information. Specifying new instances indeed

has a steeper learning curve. A single participant noted a different way to specify the first property using Globally Precedence, she argued that the start of the "On" state always has to be preceded by the occurrence of the start (or end for that matter) of the "Dwell" state. Precedence, however, does not say that a certain event has to always be preceded by another but merely that the cause enables the effect.

15.6 Detailed Feedback

The participants have offered much feedback and remarks that can in the future be used to improve the Query language both in its generic and its specific form.

15.6.1 Documentation Style and Terminology

The participants of the workshop found some documentation to be ambiguous. We provided a cheat sheet with all pattern/scopes, atomic propositions and composite operators described in natural language. These natural language descriptions were sometimes misunderstood by assumptions made due to wording. Suggestions were made to provide visual documentation instead of textual. This visual documentation would contain examples of scopes and patterns that visually show when a scope is open, or whether a pattern holds.

Furthermore, it was noted that the language used by the documentation of the Query language originates from the computer science domain. "Atomic propositions" for example is terminology common to formal logics but not to domain experts in lighting systems. Furthermore, the Invariance pattern uses the concept of invariance, but invariance is a term that is mostly used in computer science and maths. We suggest re-evaluating terminology to either generic engineering terminology, or specifically to terms of the domain.

15.6.2 Syntactical Considerations

Recurrence and Invariance Patterns Participants noted that the recurrence pattern can be misunderstood due to chosen words in its syntax. The word "Every" in the syntax of such a pattern can be interpreted in different ways, a single participant read that the recurrent event has to occur exactly every ten time units, the other participants had difficulty arguing whether the event has to reoccur within and including 10 time units of each other (closed interval), or reoccur within but not including 10 time units of each other (open interval, correct interpretation). The participants suggested to use the word "Within", but furthermore noted that the change of wording does not remove the ambiguity of the syntax regarding open and closed intervals. We, therefore, suggest changing the syntax of Recurrence to the following:

```
'Recurrence'   proposition=RefOrInline 'Within' bound=Value
```

```
Globally Recurrence P Within 10

Before R Recurrence P Within ctrl::HoldTime
```

For the invariance pattern, the participants found the order in which the pattern is specified confusing. The pattern now reads that a particular state has to hold for a certain amount of time when a particular event occurs. Their suggestion was to flip the operands, to which we agree, therefore we propose to change the syntax to:

```
'Invariance' 'When' proposition=RefOrInline
               'Then' proposition=RefOrInline
                 'For'  bound=Value
```

```
Globally Invariance When Q Then P For 10
Globally
  Invariance When sensorEvent Then actuationLevel For ctrl1::HoldTime
```

While tempting to introduce extra syntax to denote whether a timed scope is inclusive or exclusive, adding syntax comes with a readability cost. To remain faithful to the pattern library idea, it is important to evaluate these trade-offs. If no property specification uses an open-scoped Invariance pattern in a particular domain, or only very few instances do, it might be better to leave out an open-scoped Invariance pattern to keep the language concise.

15.6.3 Semantical Considerations

Open or Closed Sub-Scope A participant noted that both the Recurrence and Invariance specify a timed sub-scope. For Recurrence this timed sub-scope is right-open whereas for Invariance the sub-scope is right-closed. Even though this is in itself inconsistent, we feel that this choice is sensible. When using Invariance patterns, we often want to say that some proposition holds for *at least* a certain amount of time, whereas for Recurrence we often want to say that a particular event re-occurs within a certain deadline. The participants did not voice an opinion on whether the timed sub-scope should be open or closed, only noting that any choice made has to be clearly documented.

Response Patterns One participant noted that for a Response pattern to hold, the "effect" proposition can respond to multiple occurrences of causes whereas she might have expected a single cause being responded to by a single response. She argued that "PPS" should not hold, whereas "PSPS" or "PPSS" would. While for the cases we have seen, the

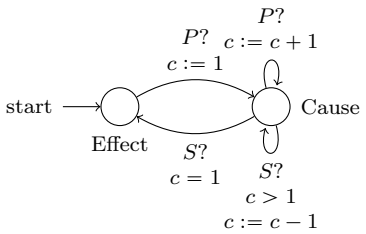


Figure 15.2: Monitor for a theoretical Globally scoped "Balanced Response" pattern

semantics of the Response pattern are sufficient, implementing a "Balanced Response" pattern could be done by counting occurrences of P and subtracting occurrences of S. The corresponding rule then asserts that any state in which more than one P remains unresponded is eventually responded to by a state in which no P's are unresponded. The corresponding automata (for the Globally scope) can be seen in Figure 15.2.

Invariance with Delay The participants noted that for the "Invariance" pattern, there may be a delay between the occurrence of the event proposition, and the continuous holding of the invariant state proposition. We have implemented this pattern in this way to prevent possible network delays from interfering with the "Invariance" pattern. Consequently, if the state proposition never starts to holds, the pattern remains true. Participants found these edge cases strange and suggested changing them. We suggest however to keep a possible delay, but constraint it to be non-infinite. We could add an "After" clause to the "Invariance" pattern that specifies the maximum amount of time between the occurrence of the event, and the continuous holding of the state proposition. Furthermore, this implementation would be more true to the original intentions of the "Invariance" patterns by the authors of pattern libraries. The corresponding monitor is displayed in Figure 15.3, in this monitor, *dt* tracks the *delay time* to a maximum of *dc*, the *delay constant*. By making the delay constant zero, removing the guards to the two succession states and prioritizing the transition to "Cause Happened, Holds" we can achieve a monitor that only allows zero delay.

15.6.4 Uncategorized Considerations

Generic vs. Lighting Specific One participant noted the different abstractions used in the Query language and without knowing the construction of the language clearly saw that the generic Query language abstracts from formal logics and that this language is further specialized by binding atomic propositions to a particular domain further encouraging evaluating reuse in other domains. We see this as a validation that splitting the language in domain specific and generic was a good decision.

Completeness of Specification The participants noted that the two given property specifications for the "Open Plan" template do not cover all possible behavior. They

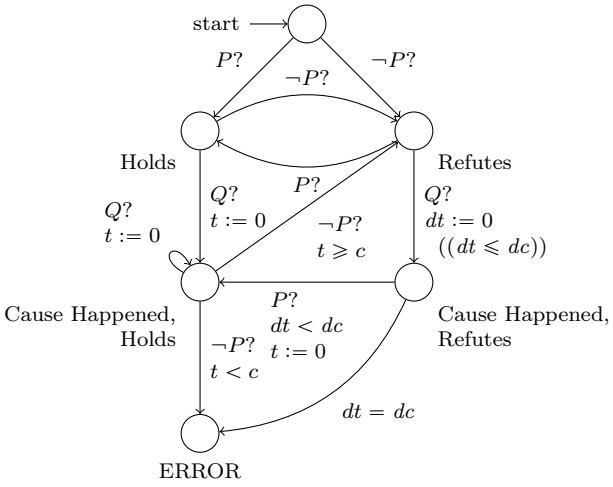


Figure 15.3: Monitor for a theoretical Globally scoped timed constrained Invariance pattern

noted that while the second Query asserts that during remote occupancy, the luminaires are either on "On" or "Background" levels, none of the specifications assert that in periods of only remote occupancy, the luminaires are indeed correctly on "Background" level. We think this is a clear example that formal specification of lighting systems can indeed be done by people uninitialized to the domain of formal methods and furthermore note that it is possible to extend the set of property specifications to properly check all possible combinations of local and remote occupancy for correct light levels.

Language Remains Formal An often heard remark was that while the participants do agree that the language is readable to people with only programming background, specifying using the Query language remains intertwined with concepts of formal verification of the computer science and math domains. For example, that certain scopes remain open forever, or more general the concept of a system with infinite traces, is an abstraction from reality sensible to computer scientists and mathematicians, but not necessarily to domain expert in lighting systems.

Intended Users A discussion was held on whom the intended users of the Query language are. The participant raising this topic suggested the rise of experts in writing Query language instances. We, however, feel that the learning curve of the Query language needs to be as shallow as possible. In our eyes, the Query language is not only tool to validate the correctness of lighting system but moreover as a piece of documentation of a particular lighting system. We suggest against specializing the language only to select stakeholders.

16 Extensibility of the Query language

The Query language design is extensible for varying reasons: Firstly, the approach of the Prisma project has not landed in the product organization of Philips Lighting and secondly, the Prisma project has yet to include several environmental models that are of interest to the lighting-specific Query language. While we could not provide a Query language that fulfills *all* needs of Philips Lighting, we have provided a language that can be extended to fulfills those needs *eventually*. By splitting the approach in a generic Query language and an instance of this Query language for the lighting domain we provide clear extension points that can be used to implement newly discovered use cases. These extension points are so generic that even re-use across domains is possible.

Showing that a language is extensible, or showing that a language is re-usable is non-trivial. There is no way to prove without any reasonable doubt that this language can *easily* be extended or re-used. This chapter will outline how the language can be extended or reused by evaluating its extension points. We will show how new pattern/scope combinations can be added, how a new atomic proposition can be added, how new composite operators can be added and finally outline how the Query language can be reused in a different domain with similar tools or in the same domain with new tools.

Common to all these extensions is the need to alter the front end of any specific Query language. However, the back end implementations of new pattern/scope combinations, as well as new composite operators can be reused across domains. Let's say that there is a specific Query language for the automotive domain, if a composite operator is added for the automotive domain, its back end implementation can directly be used in the lighting domain. Only propositions are directly tied to a particular domain.

16.1 Adding Pattern/Scope Combinations

To demonstrate implementing new pattern/scope combinations we will be outlining the required steps for a newly discovered scope that might have value for Query languages, but has yet to be evaluated for uses.

In the works of Dwyer et al. (1998) and all following papers on the subject, five scopes are outlined. Two of these scopes, the After-Until scope and the Between-scope, are very similar, but their exact meaning differs when scope opening occurs, but scope closing does not. For a Between-scoped pattern, this pattern has to only hold in a scope that is eventually closed, otherwise, these patterns are trivially true. For an After-Until-scoped pattern, that pattern has to hold indefinitely when a scope is opened but never closed. The Before scope, however, is only open when eventually closed and is much like Between in that regard, a Before with semantics similar to After-Until does not exist.

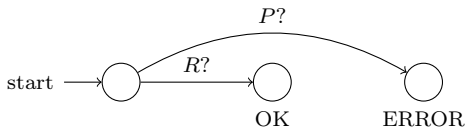
This led us to specify a dual of Before much like how After-Until is a dual of Between. This newly created scope, Until can in the future be added to the language if corresponding use cases are found. The process of adding an Until scope can be generalized into adding an arbitrary new pattern/scope combination in three steps:

1. Implement the pattern/scope combination as a generic combination of monitor/rule.
2. Implement the monitor/rule combination in the libraries for various tools.
3. Extend the front end of an existing Query language with syntax and semantics for the newly added pattern/scope combinations.

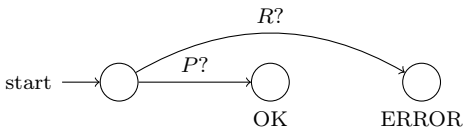
In the first step, we repeat the process outlined in Chapter 11. Any new pattern/scope combination needs to be translated to a generic monitor/rule combination. This process requires some level of creativity as it is not trivial to find a monitor/rule for a particular pattern/scope combination. We have created these mappings in Figure 16.1 for all otherwise fully implemented patterns. A monitor for an Until-scoped pattern is very similar to its Before counterpart but specially crafted for the case that the scope closing event does not occur. In particular for Existence this means that counter-examples have infinite length.

The second step repeats the processes of Section 12.4 and Section 13.2.3 and implements these generic pattern/scope combinations in the monitor libraries for UPPAAL and JCoSim respectively. These implementations are straightforward, the generic monitors are very much like UPPAAL's timed automata and we have implemented a similar formalism in JCoSim. These translations are basically textual representations of the monitor/rule combinations of Figure 16.1 and require very little creativity to be derived from the visual depictions.

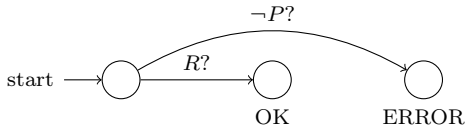
The final step is the only step that is specific to the Query language for the lighting domain. For the lighting domain, we have created a complete front end of compilation (as outlined in Chapter 10) which is arguably the least flexible part of the approach. To extend the syntax of the Query language, the grammar of this language has to be extended as well. Addition of a scope, therefore, means extending the grammar rule defining scopes. The complete (new) scope rule is depicted in Listing 16.1. Furthermore, we add a validation rule to the language implementation in Xtext that asserts that the proposition of this scope is of type event. Finally, we add extra branches to both generators (for UPPAAL and for JCoSim) to correctly use the translated monitor/rule combinations generated in step two.



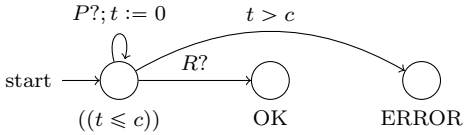
(a) Until R: Absence P, *Forbidden ERROR*



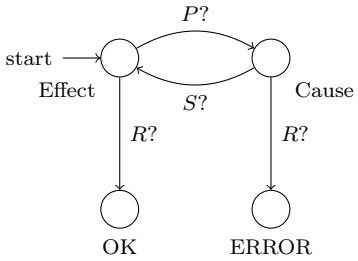
(b) Until R: Existence P, *Eventually OK*



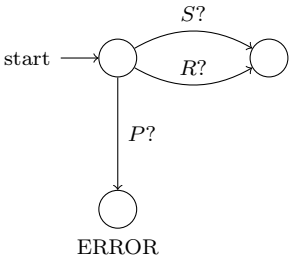
(c) Until R: Universality P, *Forbidden ERROR*



(d) Until R: Recurrence P, *Forbidden ERROR*



(e) Until R: Response P, S, *Response Cause to Effect*



(f) Until R: Precedence P, S, *Forbidden ERROR*

Figure 16.1: Until scope for all implemented patterns

```
Scope:
    'Between'  lower=RefOrInline       'And'   upper=RefOrInline
  | 'Globally'
  | 'Before'   proposition=RefOrInline
  | 'After'    lower=RefOrInline       'Until' upper=RefOrInline
  | 'After'    proposition=RefOrInline
  | 'Until'    proposition=RefOrInline
;
```

Listing 16.1: Scope grammar rule with the Until scope added.

16.2 Adding New Propositions

Another important extension point is that of propositions. Propositions, unlike pattern/scope combinations, are strongly tied to a domain model. An observer for such a proposition translates a (possibly complex) observation of a system to either a state proposition holding/refuting or an event proposition firing. Atomic propositions are the bridge between the specifics of a domain model and the general theories of propositions

and pattern/scope combinations.

Atomic propositions should be developed alongside models of new aspects of a domain model. Consider for example the addition of the concept of *physical occupancy* to the model of a lighting system. Physical occupancy, unlike detected occupancy, is the fact that a person is in a room. Detected occupancy is (often, but not always) a response to physical occupancy, a sensor detecting presence. This distinction introduces two types of faults: A sensor detecting presence in a room without the room being occupied (false positive) and a room being occupied without a sensor detecting presence (false negative). In the future, we may want to say that detected presence is (directly) preceded by actual presence (detects false positives) as well as actual presence being (directly) responded by detected presence (detects false negatives).

For the remainder of this discussion, we assume that such a model exists. This model serves two purposes, the Query language can observe the model to detect that a room is occupied (physically), and the model is capable of translating physical presence to appropriate sensors triggering.[1] Only the former of the two is relevant for the Query language as we want to create propositions stating that there is physical presence of actors in a room. For implementing such a proposition we can follow a standard process:

1. Extend the front end of a specific Query language with syntax and semantics for the newly added proposition.

2. Implement an observer for the proposition in both UPPAAL and JCoSim.

Semantically, such a proposition is best categorized as of state type, an actor is in a room for some duration. The syntax of this hypothetical proposition reflects this choice, is displayed in Listing 16.2 and should be added to the rules defining proposition literals in the Xtext grammar. Some example instances can be seen in Listing 16.3.

```
'OccupancyStatus' room=[Building::Room | FQN] 'is' status=('Vacant' | '
    Occupied')
```

Listing 16.2: Grammar rule for proposition detecting physical presence

```
// Example for the MeetingRoom example
OccupancyStatus OfficeBuilding.Floor1.Theater is Occupied
// Example for the CorridorLinking example
OccupancyStatus OfficeBuilding.Floor1.Corridor1 is Vacant
```

Listing 16.3: Example instances of physical presence proposition.

Such an implementation would suffice but is not flexible. The Query language specifically for the lighting domain can further be extended to leverage the domain knowledge of

[1] Accurate modeling the detection of physical presence is non-trivial and will involve highly detailed models of sensors and environmental conditions in a building.

the newly added propositions. For example, the with-each-controller construct allows to reason about all controllers that implement a certain template. To create this construct, we have applied domain knowledge of lighting systems, specifically that a single controller always implements a single template. Similarly, for occupancy we could introduce the concept of controllers being influenced by occupancy of certain rooms, allowing us to say for example that physical presence in a room with a controller implementing the Office template, leads to the controller being in the "On" state. While losing some re-usability across domains, adding more domain knowledge to the Query language increases its ease of use.

Finally, adding a new proposition also requires us to develop observers in both UPPAAL and JCoSim, but these implementations are so tightly coupled to the implementations of a non-existent model of physical occupancy in these tools that we cannot abstractly describe their implementation.

16.3 Adding New Composite Operators

The last extension points of the Query languages are composite proposition operators. In the language we have implemented boolean AND, boolean OR, boolean NOT and "casting" from state to event propositions as composite proposition operators. The assignment operator is never translated to UPPAAL or JCoSim, referencing other proposition definitions can all be resolved compile time. Implementing a new operator is trivial. Let's say that we want to implement the boolean implies operator for state propositions as a shorthand[2]. The required steps to do so are:

1. Extend the front end of a specific Query language with syntax and semantics for the new operator.

2. Implement the new operator in UPPAAL and JCoSim.

In the front end for the lighting-specific Query language operators have precedence and associativity to deal with ambiguity (an alternative would be Polish notation), for each new operator we have to define its precedence and associativity. The implies arrow is typically right-associative (meaning $p \rightarrow q \rightarrow r$ evaluates as $p \rightarrow (q \rightarrow r)$) and has a lower precedence than and/or (meaning $p \rightarrow q \wedge r$ evaluates as $p \rightarrow (q \wedge r)$). The Xtext parser is of the recursive-descent type which means that precedence is implemented by the ordering of rules, deeper into the rule-chain means higher precedence. Associativity is implemented in a rule itself and follows a certain template, in the base language the assignment operator is right-associative whereas all other operators are left-associative. Since we want implies to have a lower precedence than OR, we insert it above boolean OR in the chain and use the same rule structure as for assignment to make it right-associative. The grammar rule for the theoretical implies operator is given in Listing 16.4. Examples are given in Listing 16.5.

[2] $a \rightarrow b \iff \neg a \vee b$, the latter of which is expressible in the presented Query language for the lighting domain

```
// Called by AssignmentExpression
BooleanImpliesExpression:
  BooleanOrExpression ('implies' right=BooleanImpliesExpression)?
;
```

Listing 16.4: Implies grammar rule

```
// a -> (b -> c)
a implies b implies c

// (a -> b) -> c
(a implies b) implies c

// a -> (b & c)
a implies b and c

// (a -> b) & c
(a implies b) and c
```

Listing 16.5: Implies syntax examples

Semantically implies is just as boolean AND, and boolean NOT only defined for state propositions as it potentially makes statements of the simultaneous holding of two propositions. This choice is reflected in the type system of the Query language and in validation rules. A monitor for this composite operator in UPPAAL is given in Figure 16.2. In JCoSim such a proposition is observed by a simulator that subscribes to status updates of both the left and the right proposition and only outputs *Refutes* if the status has changed and the left proposition holds, but the right does not.

16.4 Reuse of Query language

The front end of the Query language for the lighting domain cannot easily be reused across domains. The grammar of the language is strongly tied to the lighting domain and not properly modularized. Only recently we came across a blog[3] describing options of Xtext to modularize grammars but we were unable to implement such a modularization in time. Ideally, all domain generic parts of the Query language are separated into a single (abstract) DSL. This abstract DSL would not have atomic propositions nor domain-specific constructs like the with-(each-)controller statement, but would include all pattern/scope combinations and composite operators.

Pattern/scope combinations, however, are generic and are implemented as such. In Chapter 11 a process was outlined to transform pattern/scope combinations to monitor/rule combinations. The latter combinations are implemented as templates in both

[3] https://zarnekow.blogspot.nl/2015/11/improved-grammar-inheritance.html

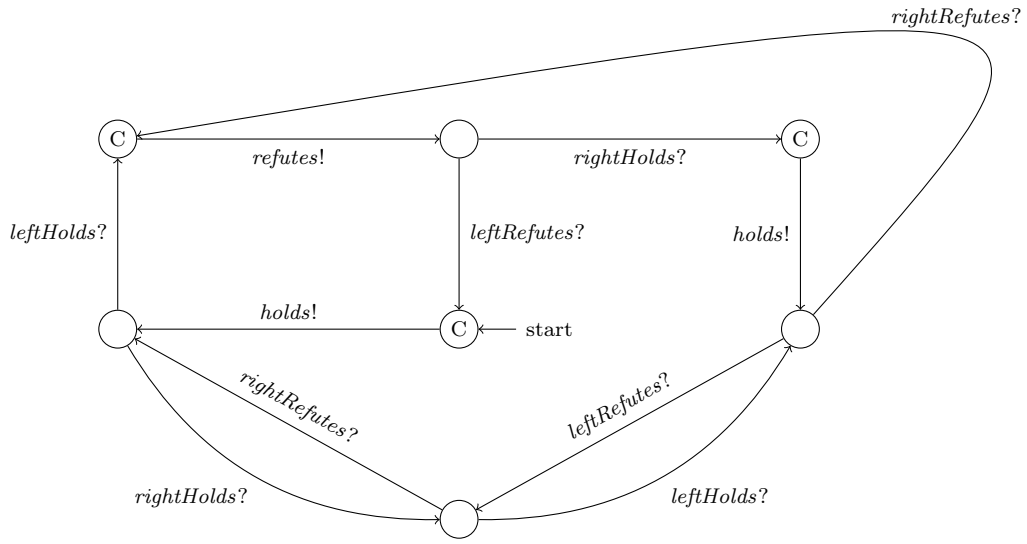


Figure 16.2: UPPAAL monitor for an implies proposition

UPPAAL and JCoSim. For UPPAAL, templates consists of fragments of Statemachine language text with "holes" for proposition channels and (possibly) parameters. In JCoSim, these monitor/rule combinations are factory functions producing a monitor for particular propositions/parameters. Additionally, composite operators are implemented in a domain generic way.

The process of reusing the Query language for a new domain can then be seen as the following process:

1. Provide a full syntax/semantical model for the newly developed Query language.
2. Reuse implementations of pattern/scope combinations and composite operators in both generators for UPPAAL and JCoSim.
3. Provide new observers for domain-specific atomic propositions, newly used pattern/scope combinations and newly specified composite operators.

16.5 A New Backend

The current Query language offers implementations in both UPPAAL and JCoSim but can easily be extended to cover a new tool. Traditional pattern libraries are translated to temporal logics, but these logics are underrepresented in simulation tools. Most tools do however have a notion of a (timed) state machine. The monitors of monitor/rule combinations described in Section 11 can then directly be translated to the new tool leaving only the rule to be specified. However, when implementing this Query language for a new tool, it is important to note that it has been build with certain underlying

features of the tooling in mind. For example, monitor automata need priority over the system under test to prevent desynchronization between the system and the monitors, for JCoSim specifically we had to add priority. Secondly, the Query language heavily uses broadcast communication. Propositions are observed by synchronizing to broadcast channels used internally, or by also subscribing to particular messages sent over the publish subscribe network. Similarly, a single proposition observer can be used in many composite proposition observers. For a tool to be able to implement the Query language, it needs to have a way to capture broadcast synchronization.

Part V

Conclusions and Closing Remarks

Part Outline

This part serves as a conclusion to the report, we answer the research questions, provide a brief conclusion and outline possible future work. This part is split into two chapters:

Chapter 17 reflects upon the work presented in Part III and uses the validation steps of PartIV to answer the research questions of Chapter 3. This chapter ends with a brief conclusion of the research conducted.

Chapter 18 shows research that can be conducted to further strengthen the work presented in this report, we propose to extend the language in expressiveness, attempt reuse across domains, prove the correctness of the extension of the work of Gruhn and Laue (2006) in the form of monitor/rule implementation of pattern libraries and increase feedback provided by refuted properties. We finally discuss a possible change of metamodel used to target the UPPAAL model checker specifically.

17 Results & Conclusions

In Chapter 3 the requirements of the Query language and corresponding research questions were outlined. Throughout Part III choices were made to warrant the desired requirements and in Part IV we validated whether the requirements were met. In this chapter, we first answer the subquestion and using these answers, answer the main research question.

17.1 Pattern Libraries as a Foundation

We have chosen to pursue a pattern library approach with the assumption that these libraries allow us to create an easy to use, complete and extensible framework for the development of Query languages. We have also challenged this assumption in validation. In this section, we outline whether using a pattern library as a basis for both the generic Query language as well as the Query language for the lighting domain is justified.

Subquestion 1 Are pattern libraries a solid foundation of an easy to use Query language?

In the usability workshop held within the Prisma team and outlined in Chapter 15, we have validated how well instantiated pattern/scope combinations are understood without significant training. This approach has allowed us to gain understanding on whether the meaning of pattern/scope combinations is *intuitive*: is it immediately clear whether an instantiated pattern/scope combination holds for a particular trace? Participants noted that while they felt pattern/scope combinations to be understandable, they initially had difficulties with the concept of formal reasoning. We see the latter as a fundamental problem: regardless of how easy to use a particular Query language approach is, the user is still required to be able to reason formally. However, participants also noted that the use of pattern/scope combinations makes the specification of properties easier and showed significant improvement throughout the session. In terms of learning time, we conclude that compared to traditional temporal logics, learning pattern libraries takes less learning time, but note that both approaches require unfamiliar users to learn of formal methods first.

Subquestion 2 Can a pattern library approach cover all use cases of a formal specification language for the lighting domain?

We have not encountered any example property that could not be captured using (a set of) pattern/scope combinations supported in the Query language for the lighting domain outlined in Section 9.4. We have however seen cases where pattern/scope combinations

may need to be altered slightly in semantics. The most notable examples being the duration patterns outlined in Section 11.3.4. We would like to note that the underlying translation to monitor/rule combinations allows these slight variations in semantics.

Subquestion 3 Does a pattern library approach allow easy extension when new use cases are discovered?

Pattern libraries are extensible by design as only propositions introduce domain knowledge. That said, the current selection of pattern/scope combinations for the lighting specific Query language allows to specify temporal and timed temporal properties. If new properties are discovered that require probabilistic reasoning, significant changes may need to be made to monitor/rule combinations.

Subquestion 4 Can a pattern library approach be implemented in both UPPAAL and the Co-simulation Framework?

The intermediate translation to monitor/rule combinations outlined in Chapter 11 is translated to both UPPAAL and JCoSim and outlined in Chapter 12 and Chapter 13 respectively. Implementing monitor/rule combinations in UPPAAL and JCoSim is straightforward, given the timed temporal pattern library selected. When probabilities are introduced, especially the translation to JCoSim needs to be extended heavily.

For the first stage translation, from pattern/scope combinations to monitor/rules, we would like to see a comparison with implementations in traditional temporal logics. Is an implementation of a particular pattern equal in semantics in both temporal logics and a monitor/rule combinations? This proof, however, is not trivial and remains future work.

17.2 Query Language Prototype

Subquestion 5 Can syntax and semantics be developed that allows intuitive specification of properties of lighting systems?

The syntax and semantics of the Query language are a mix of traditional (imperative) programming languages for structure and declarative syntax for specification of propositions and properties. This combination allows concise specification of complex properties combining many propositions but is also readable. In the usability workshop, we have seen that Prisma project members had no issue understanding the concepts and syntax of the Query language and were able to adapt Query language instances easily.

Subquestion 6 Can the Query language for the lighting domain be integrated within the approach of Prisma?

In the presented Query language for the lighting domain, we have added controller references as additional syntactic sugar specifically for the lighting domain. Controller references allow reasoning about sensor and actuator groups of a controller, (overridden)

parameters of a controller and states that a particular controller is in. The addition of controller references reduces Query instance size significantly for buildings with largely similar functionality. We believe that by introducing the domain concept of controllers in the Query language for the lighting domain we have created a well integrated Query language.

Subquestion 7 Can the Query language for the lighting domain trivially be extended to cover new use cases?

Extension of the concepts of the Query language for the lighting domain is trivial but inflexible. Introduction of new atomic propositions, composite operators or pattern/scope combinations can be done but requires manual editing of the parser's grammar and code generation framework. In future work, we outline a possible extension to separate the syntax and (language) semantics of the specific Query language further into a domain specific and domain generic part.

Subquestion 8 Can we show the value of a Query language for the lighting domain to relevant stakeholders within Philips Lighting?

In the steering board meeting, we have demonstrated the Query language. The members of the steering board agreed that using a Query language can indeed be beneficial to Philips Lighting. Members of the steering board were positive about using a single specification for both simulation and formal verification as well as its concise specification of properties for larger lighting systems of similar templates. Members saw the need for a property specification language for large scale lighting systems as finding errors at the presented scale would be infeasible. Finally, members saw that the source of errors is not always obvious, errors could be introduced as template bugs, as commissioning errors, etc. The Query language was well received.

Subquestion 9 Can we convey to a stakeholder that a property is not correct in a way that indicates what causes the faults without needing fundamental knowledge of temporal logic or model checking?

In simulation, we show error by coloring the sphere of influence of a particular controller according to the severity of the raised verdict. Properties that are not refuted and cannot be refuted are bright green, properties that are refuted and cannot hold in the future are bright red. This color is useful when aiming to find whether a system is correct, but does not display the exact fault that occurred. Finding exact causes of fault requires analysis of abstract traces from UPPAAL. In future work, we outline proposed research in the creation of Scenario instances based on occurred faults, but this translation has still to be implemented.

17.3 Conclusion

To conclude this report, reconsider the research question posed in Chapter 3:

Research Question Can we, in the context of the domain approach of the Prisma project, create a framework and prototype of a domain specific Query language in which lighting properties can be specified and formally verified? This framework and prototype must align with the technical knowledge of the stakeholders in the lighting domain both in specifying properties and conveying verification results.

We have successfully created a framework for the creation of Query languages and applied this framework to the lighting domain. This Query language is well integrated into the domain approach of the Prisma project, has a concise and understandable syntax, and can be (automatically) implemented in both UPPAAL and the Co-simulation Framework. We have created, based on the works of Gruhn and Laue (2006), a generic translation of pattern libraries to monitor/rules that we feel is easier to implement in tools and allows capturing slight variations of semantics. We have successfully shown the value of early validation of lighting systems to Philips Lighting using both simulation and formal verification.

The timing of the performed research was early. We have created a framework of a Query language that can still be tweaked to further align with the use cases of a Query language in the lighting domain. When the domain approach is fully landed in the product organization of Philips Lighting and the domain model of the Prisma project is more complete, we can create an exhaustive list of use cases and return to the original research questions of Chapter 7. Furthermore, the approach can be strengthened by the creation of counter examples in domain concepts. Finally, we would like to see the pattern library approach to be applied to different domains to strengthen the belief of its universal applicability.

18 Future Work

In this chapter, we outline possible extensions of the work as presented in this report. We have chosen to restrict this listing to future work that can be used to further strengthen the evidence that both the generic Query language approach, as well as the specific Query language for the lighting domain, are easy to use, complete, verifiable, traceable, integrated and extensible. Other sources of future work, such as a scalability analysis, are very interesting as well but not directly in the context of this particular research. We demonstrate four points of future work: (1) extension within the lighting domain, (2) reuse across other domains, (3) increase of traceability, and (4) analysis of the formal correctness of the approach. Finally, we propose to change the intermediate model used in UPPAAL model generation.

18.1 Language Extension

The Prisma project has continued development of its domain model. Recent additions are focused on increasing the expressiveness of the domain model and code generation. TNO-ESI has created two milestones: DSL 1.1 is a short term upgrade of the expressiveness of the domain model and DSL 2.0 is exploratory research in how to capture *feature interaction*[1] in the domain model. DSL 1.1, the direct successor to the languages presented in Chapter 2, is still work in progress. Its features still change regularly and the translations to UPPAAL and the Co-simulation Framework have not yet been updated. The newly gained expressiveness allows specifying *dimming* and *daylight regulation* by means of *actuation functions*. The addition of new types of timers allows to further specify *dwell time*, *retention time* and *hold time* behavior. The newly gained expressiveness aligns DSL 1.1 with the requirements of Philips Lighting: many of common commercially available controller templates can be and have been expressed in the Template language of DSL 1.1.

Updating the Query approach to DSL 1.1 requires not only the Query language itself to be updated but also the supporting lighting system model in both UPPAAL and the Co-simulation Framework. In the Query language, the set of implemented atomic propositions needs to be re-evaluated and possibly amended. Consider for example a workspace with dimming behavior. A new property could say that during local occupancy, all luminaires of a workspace are on the On level with *a certain dimming offset*.

[1] Feature interaction, a term originating from telecommunication systems, is the study of potential aspects of a system interacting. Consider for example the features "dimming" and "daylight regulation", we may wish to have "daylight regulation" be applied over the dimmed output level. A summary of detection and resolution techniques for feature interaction is given by Calder et al. (2003).

Besides extra expressiveness in behavior, the (upcoming) introduction of networking, occupancy patterns and energy usage in the domain model allows the adapted Query language to specify new types of properties. By extending the Query language with propositions for these new environmental models, we can gain a better understanding of the usability, completeness and integration of the Query language in the lighting domain. We can re-evaluate the pattern/scope selection, change the semantics of particular patterns to closer match their intended use, introduce new composite operators, etc. Only with a complete domain model of a particular domain can we truly answer whether the Query language for that particular domain is easy to use, complete, integrated and extensible.

18.2 Reuse of Query Language

We have demonstrated a language that we feel is not only extensible in the lighting domain but also reusable in other domains. We suggest evaluating its reuse potential by conducting a case study on applying the generic Query language to a non-lighting domain. We believe that conducting such a case study can be valuable, not only to validate that the generic Query language is indeed generic but also in finding new use cases within the lighting domain. Applying the approach to another domain can uncover the need for other pattern/scope combinations, or strengthen our understanding which specific semantics of particular patterns are universally desired, etc. Having a universal generic Query language across domains can only aid a specific domain such as lighting.

To reuse the Query language in a non-lighting domain however, the front end of the language has to be modularized. Currently, the grammar of the specific Query language mixes domain specific and domain generic constructs directly: both atomic propositions and the with-each controller statement are domain specific, but specified in the same grammar as domain generic constructs such as pattern/scope combinations. By separating domain specific and domain generic constructs in (inheriting) grammars, it would be possible to directly reuse most of the syntax and semantics of the Query language for lighting systems in other domains. Providing this split requires some level of grammar inheritance that goes further than sharing terminal rules.[2] We have only recently learned that Xtext allows inheriting and overriding parser rules. We suggest attempting such a modularization using Xtext. Alternatively, one could evaluate any other language workbench described by Erdweg et al. (2013) to implement a truly (abstract) generic Query language.

18.3 Increase Traceability

The feedback provided by the implementations is currently limited. In visualization, a particular controller's sphere of influence is colored according to the raised verdict, but

[2] As described in the documentation: `https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html#grammar-mixins`

the results of multiple queries on the same controller are currently not properly conveyed, verdicts would be processed in message order. A suggestion would be to color controllers the severest of verdicts, if any of the properties of a particular controller is refuted it is colored red for example. More ideally, a way needs to be found to convey the validity of all properties of a particular controller simultaneously. Only applying a single color to a controller is too limiting when a signification amount of properties are specified about a particular controller.

In model checking, if a property is refuted, an error trace is generated in UPPAAL's simulation view. Unfortunately, this trace is contaminated with internal communication of the model and is very abstract. These traces are difficult to read requiring a large understanding of not only the lighting system model but also of the way that UPPAAL does model checking. Ideally in the future, instead of producing a UPPAAL trace, we produce an instance of the Scenario DSL.[3] This Scenario instance can then be imported in the Co-simulation Framework to visually show the observed counter example. Implementing such a translation requires filtering the output of UPPAAL in such a way that all relevant information is translated back, but all superfluous information is removed. Furthermore, traces are not only contaminated with internal details, they are also often symbolic: the output of UPPAAL is then a list of *states* in which particular *clock constraints* must be satisfied. Creating a singular "concrete" instance of the Scenario DSL may require *constraint solving*.

18.4 Formal Verification of Monitor/Rule Combinations

In this research, we have presented a continuation of the work of Gruhn and Laue (2006) in the form of a translation from pattern/scope combinations from various pattern libraries to monitors and rules, we have outlined multiple ways to interpret particular patterns. Minimum Duration is such a case, where multiple ways exist to interpret scope opening/closing with respect to how long a state proposition is required to hold. By (formally) comparing the corresponding TCTL/MTL formula with the monitors presented in Chapter 11 we can determine whether these are semantically the same. We would like to stress, however, that this does not necessarily means that the way these patterns are implemented by the pattern library authors is the only possible way. For Minimum Duration, in particular, we have seen multiple use cases using varying semantics. We believe that the comparison of the monitor/rule combinations with the implementation of patterns in temporal logic will uncover differences that can either be defended because of the needs in a particular domain or the corresponding monitor/rule combination can be updated to correctly implement such a semantical difference.

We believe that comparing monitor/rule combinations with temporal logic formulas can be challenging. Monitors observe events from the system under test, whereas the rule observes states from that particular monitor. The standard process of translating both the rule and the monitor to Büchi automata, computing the parallel composition and

[3] Or the internally used "event file" format.

comparing such a monitor to the Büchi automata of the CTL/LTL/MTL/TCTL formula may not work directly. Furthermore, by comparing our implementation in monitor/rule combinations to the various implementations by pattern library authors, we may even find inconsistencies within these preexisting implementations.

18.5 State Machine Language

Translation of lighting system specifications to the various tools (POOSL, UPPAAL and the Java simulator) occurs in two steps: from the various system models (Building, Template, Control, etc.) via an intermediate State Machine language to the format of a particular tool. In Section 12.2 we outlined the advantages of this intermediate model, if m domain models translate to n tool specific models, the use of an abstract intermediate language reduces the number of transformations from $m \times n$ to $m + n$. Currently, two models translate to the State Machine language: the model of a lighting system and the (upcoming) model of occupancy patterns. Of the tools, POOSL is largely unused and the Java simulator is only used in very specific cases. It is therefore, safe to assume that the State Machine language representation of a lighting system will, in most cases, be generated for transformation to UPPAAL.

The translation of the lighting system to the State Machine language, however, still assumes that both eager and non-eager tools are supported. By generating two instances of the State Machine language, one for eager and one for non-eager tools, we can already solve some of the issues outlined in Section 12.4. Other problems, most notably the lack of CTL queries, priorities and extra comparison operators, require extension of the State Machine language.

Before committing resources to the State Machine language, it is important to realize that all changes proposed in this report are needed for formal verification in UPPAAL specifically. In other words, the State Machine language needs to be more like UPPAAL's input model. Schivo et al. (2017) present a metamodel of UPPAAL that better suits our needs, not only does it support a wider range of features in timed automata, it also includes a metamodel of UPPAAL's CTL query language as well as a metamodel of UPPAAL's trace format. Incidentally, we can directly use this metamodel for transformation to UPPAAL: both Xtext's generated metamodel of a lighting system and the metamodel of UPPAAL are supplied in the Ecore format of the Eclipse Modeling Framework (EMF). EMF provides several tools to transform instances of a particular metamodel to instances of another metamodel. The State Machine language and UPPAAL metamodel are also similar, many of the generated state machines in the State Machine language can be translated one to one to instances of the metamodel of UPPAAL. Furthermore, the existence of a metamodel (and parser) for the trace format of UPPAAL can be of great help in increasing the traceability extensions outlined in Section 18.3.

Bibliography

Xtend documentation. `http://www.eclipse.org/xtend/`. Accessed: 2017-11-14.

R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM (JACM)*, 43(1):116–146, 1996.

M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. *IEEE Transactions on Software Engineering*, 41(7):620–638, 2015.

A. Aziz, V. Singhal, F. Balarin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In *Proceedings of the International Conference on Computer Aided Verification (CAV 1995)*, pages 155–165. Springer, 1995.

C. Baier, J.-P. Katoen, and K. G. Larsen. *Principles of model checking*. MIT press, 2008.

P. Bellini, P. Nesi, and D. Rogai. Expressing and organizing real-time specification patterns via temporal logics. *Journal of Systems and Software*, 82(2):183–196, 2009.

J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal—a tool suite for automatic verification of real-time systems. *Hybrid Systems III*, pages 232–243, 1996.

P. Bouyer, N. Markey, J. Ouaknine, and J. Worrell. On expressiveness and complexity in real-time model checking. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP 2008)*, pages 124–135. Springer, 2008.

M. Brambilla, J. Cabot, and M. Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

L. Buit. Research topics: Developing an easy to use query language for verification of large lighting systems. unpublished research topics report, 2017.

M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier. In *Proceedings of the International Conference on Computer Aided Verification (CAV 1999)*, pages 495–499. Springer, 1999.

E. M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 428–437. Springer, 1988.

E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

E. M. Clarke and E. A. Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*, pages 52–71. Springer, 1982.

E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

R. Doornbos, J. Verriet, and M. Verberkt. Robustness analysis for indoor lighting systems: An application of model checking in large-scale distributed control systems. In *Proceedings of the Tenth International Conference on Systems (ICONS 2015)*, pages 46–51. IARIA, 2015.

M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15. ACM, 1998.

M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 411–420. IEEE, 1999.

E. A. Emerson and J. Y. Halpern. "sometimes"; and "not never" revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, Jan. 1986a.

E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986b.

S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The state of the art in language workbenches. In *Proceedings of the International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.

S. Flake, W. Müller, and J. Ruf. Structured english for model checking specification. In *Proceedings of Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 99–108, 2000.

M. Fowler. Language workbenches: The killer-app for domain specific languages? `http://martinfowler.com/articles/languageWorkbench.html`, 2005.

A. Gill and S. Marlow. Happy: the parser generator for haskell. Technical report, University of Glasgow, 1995.

V. Gruhn and R. Laue. Patterns for timed property specifications. *Electronic Notes in Theoretical Computer Science*, 153(2):117–133, 2006.

L. Grunske. Specification patterns for probabilistic quality properties. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 31–40. IEEE, 2008.

H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

M. Hendriks, M. Geilen, A. R. B. Behrouzian, T. Basten, H. Alizadeh, and D. Goswami. Checking metric temporal logic with trace. In *Proceedings of the 16th International Conference on Application of Concurrency to System Design (ACSD 2016)*, pages 19–24, 2016.

IEEE. Ieee standard for modeling and simulation & high level architecture (hla)– framework and rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, pages 1–38, 2010.

S. C. Johnson. *YACC: yet another compiler-compiler*. Bell Laboratories, 1986.

S. Konrad and B. H. Cheng. Real-time specification patterns. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 372–381. IEEE, 2005.

R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

S. A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16 (1963):83–94, 1963.

I. Kurtev, J. Hooman, and M. Schuts. *Runtime Monitoring Based on Interface Specifications*, pages 335–356. Springer International Publishing, Berlin, 2017.

L. Lamport. Sometime is sometimes not never: On the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–185. ACM, 1980.

L. Lamport. *LATEX: a document preparation system: user's guide and reference manual*. Addison-wesley, 1994.

K. G. Larsen, P. Pettersson, and W. Yi. Model-checking for real-time systems. In *International Symposium on Fundamentals of Computation Theory*, pages 62–88. Springer, 1995.

M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

G. H. Mealy. A method for synthesizing sequential circuits. *Bell Labs Technical Journal*, 34(5):1045–1079, 1955.

E. Meijer, B. Beckman, and G. Bierman. Linq: reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 706–706. ACM, 2006.

O. A. Mondragon, A. Q. Gates, and S. M. Roach. Composite propositions: toward support for formal specification of system properties. In *Proceedings of the 27th Annual Software Engineering Workshop*, pages 67–74. IEEE, 2002.

E. F. Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34: 129–153, 1956.

G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.

J. Ouaknine and J. Worrell. *Some Recent Results in Metric Temporal Logic*, pages 1–13. Springer, 2008.

T. J. Parr and R. W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57. IEEE Computer Society, 1977.

A. N. Prior. *Time and modality*. John Locke Lecture, 1957.

J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer, 1982.

Y. S. Ramakrishna, L. K. Dillon, L. E. Moser, P. M. Melliar-Smith, and G. Kutty. An automata-theoretic decision procedure for future interval logic. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 51–67. Springer, 1992.

A. Rensink. The groove simulator: A tool for state space generation. In *Proceedings of the Second Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 479–485. Springer, 2003.

S. Salamah, A. Gates, and V. Kreinovich. Validated templates for specification of complex ltl formulas. *Journal of Systems and Software*, 85(8):1915–1929, 2012.

S. Schivo, B. M. Yildiz., E. Ruijters, C. Gerking, R. Kumar, S. Dziwok, A. Rensink, and M. Stoelinga. How to efficiently build a front-end tool for uppaal: A model-driven approach. In *Proceedings of the Symposium on Dependable Software Engineering Theories, Tools and Applications (SETTA 2017)*, 2017.

B. D. Theelen, O. Florescu, M. C. W. Geilen, J. Huang, P. H. A. Van der Putten, and J. P. M. Voeten. Software/hardware engineering with the parallel object-oriented specification language. In *Proceedings of the 5th International Conference on Formal Methods and Models for Codesign*, pages 139–148. IEEE Computer Society, 2007.

S. Tripakis. *L'analyse formelle des systèmes temporisés en pratique*. PhD thesis, Université Joseph-Fourier-Grenoble I, 1998.

A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

J. Verriet, R. Hamberg, J. Caarls, and B. van Wijngaarden. Warehouse simulation through model configuration. In *Proceedings of the 27th European Conference on Modeling and Simulation (ECMS 2013)*, pages 629–635, 2013.

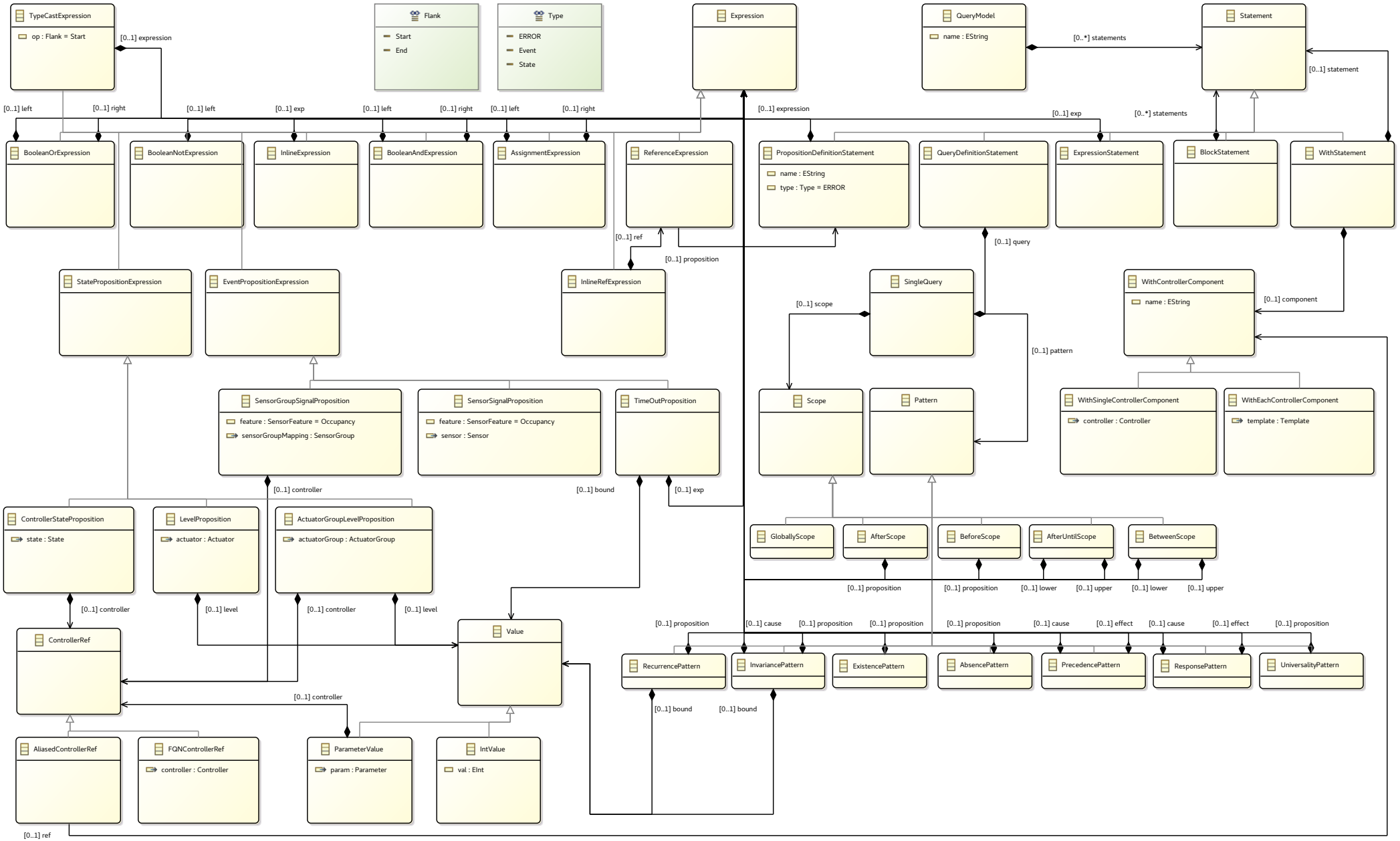
D. A. Watt and D. F. Brown. *Programming language processors in Java: compilers and interpreters*. Pearson Education, 2000.

Xtext (2017). Xtext - language engineering made easy!, 2017. `https://eclipse.org/Xtext/` retrieved at 02-05-2017.

S. Yovine. Kronos: A verification tool for real-time systems. *Proceedings of the International Journal on Software Tools for Technology Transfer (STTT 1997)*, 1(1):123–133, 1997.

Appendices

A Metamodel of the Query Language for the Lighting Domain

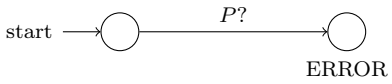


B Mapping of pattern/scope combinations to monitor/rule pairs

B.1 Absence

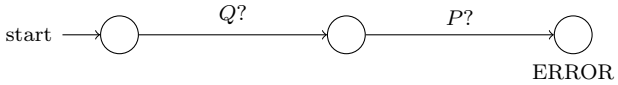
B.1.1 Globally: Absence P

Rule: *Forbidden ERROR*



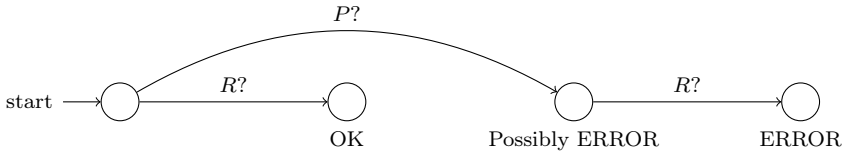
B.1.2 After Q: Absence P

Rule: *Forbidden ERROR*



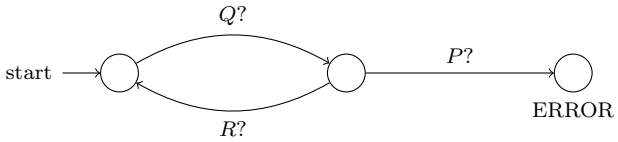
B.1.3 Before R: Absence P

Rule: *Forbidden ERROR*



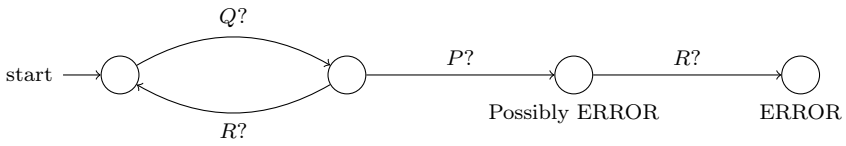
B.1.4 After Q until R: Absence P

Rule: *Forbidden ERROR*



B.1.5 Between Q and R: Absence P

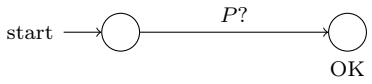
Rule: *Forbidden ERROR*



B.2 Existence

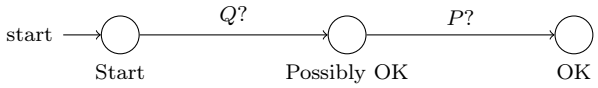
B.2.1 Globally: Existence P

Rule: *Eventually OK*



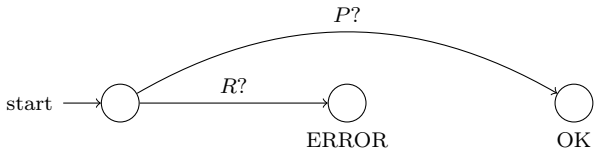
B.2.2 After Q: Existence P

Rule: *Response OK to Possibly OK*



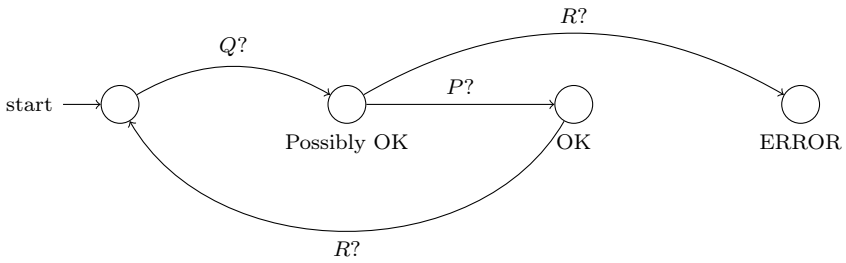
B.2.3 Before R: Existence P

Rule: *Forbidden ERROR*



B.2.4 After Q until R: Existence P & Between Q and R: Existence P

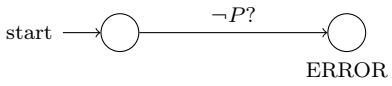
Rule (After/Until): *Forbidden ERROR* Rule (Between): *Response OK to Possibly OK*



B.3 Universality

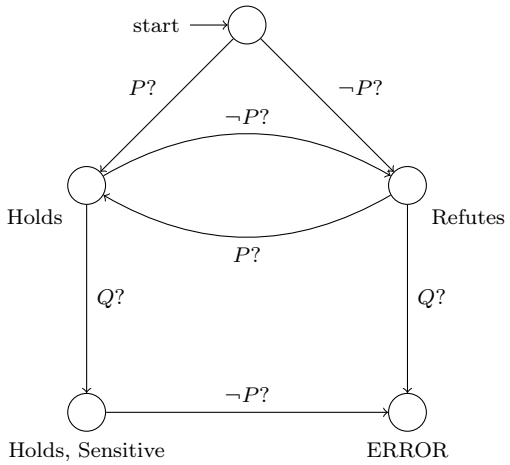
B.3.1 Globally: Universality P

P must be a state proposition, rule: *Forbidden ERROR*



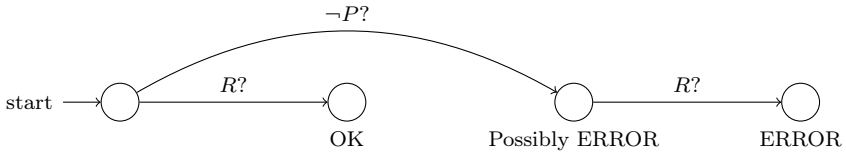
B.3.2 After Q: Universality P

P must be a state proposition, rule: *Forbidden ERROR*



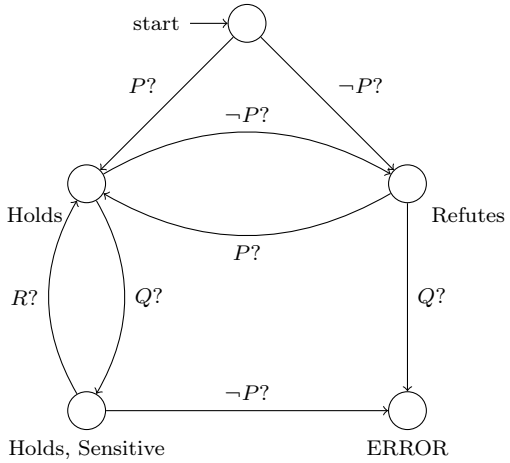
B.3.3 Before R: Universality P

P must be a state proposition, rule: *Forbidden ERROR*



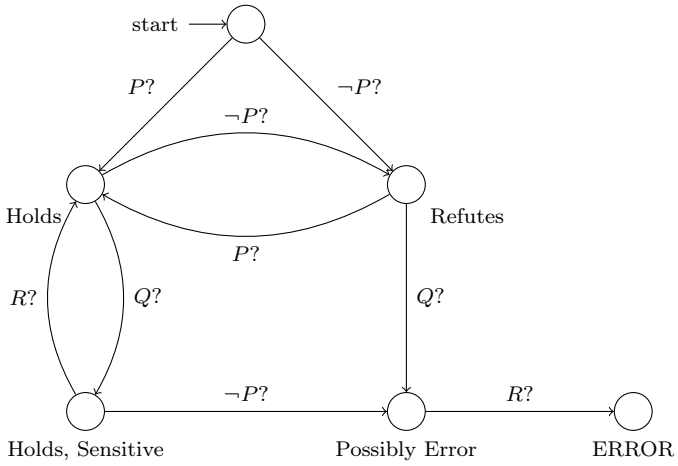
B.3.4 After Q until R: Universality P

P must be a state proposition, rule: *Forbidden ERROR*



B.3.5 Between Q and R: Universality P

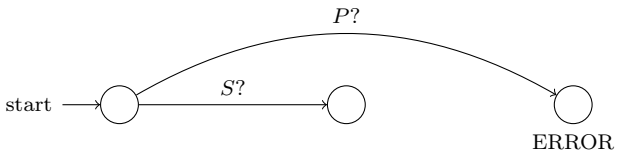
P must be a state proposition, rule: *Forbidden ERROR*



B.4 Precedence

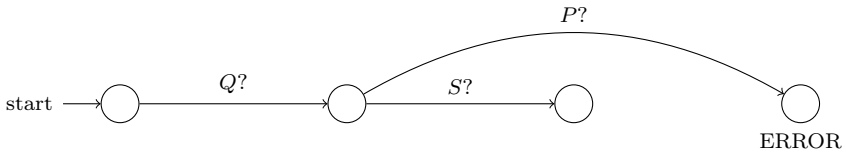
B.4.1 Globally: Precedence P, S

Rule: *Forbidden ERROR*



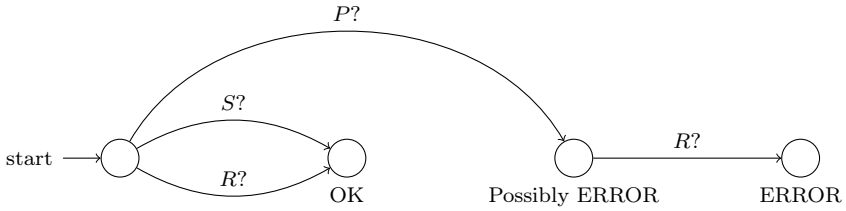
B.4.2 After Q: Precedence P, S

Rule: *Forbidden ERROR*



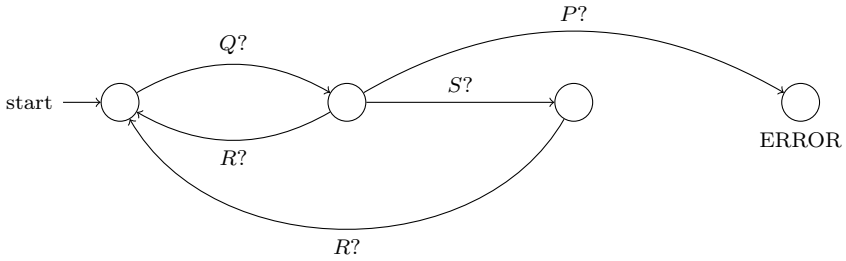
B.4.3 Before R: Precedence P, S

Rule: *Forbidden ERROR*



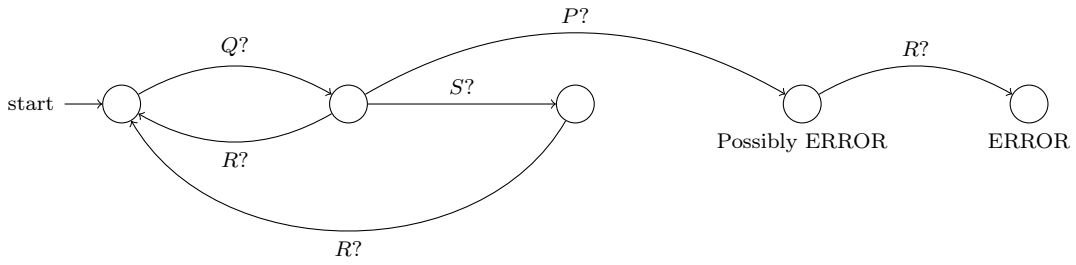
B.4.4 After Q until R: Precedence P, S

Rule: *Forbidden ERROR*



B.4.5 Between Q, R: Precedence P, S

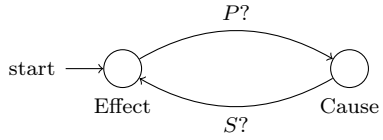
Rule: *Forbidden ERROR*



B.5 Response

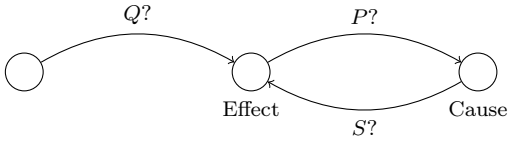
B.5.1 Globally: Response P, S

Rule: *Response Effect to Cause*



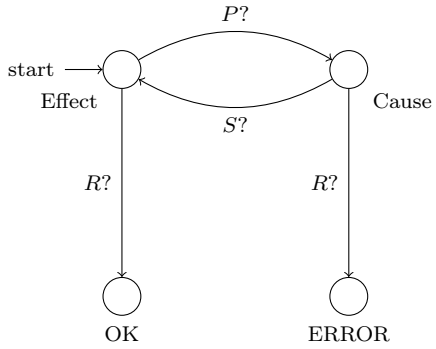
B.5.2 After Q: Response P, S

Rule: *Response Effect to Cause*



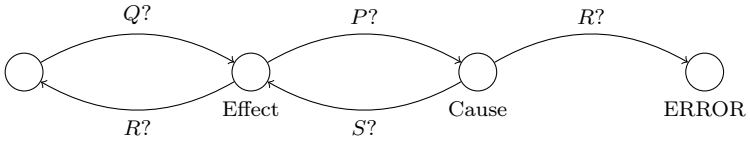
B.5.3 Before R: Response P, S

Rule: *Forbidden ERROR*



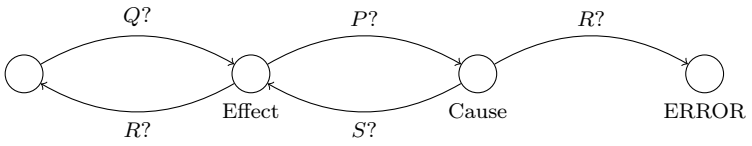
B.5.4 After Q until R: Response P, S

Rule: *Response Effect to Cause*



B.5.5 Between Q, R: Response P, S

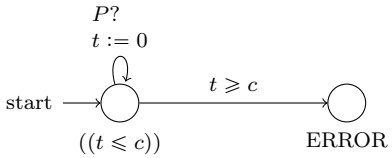
Rule: *Forbidden ERROR*



B.6 Recurrence

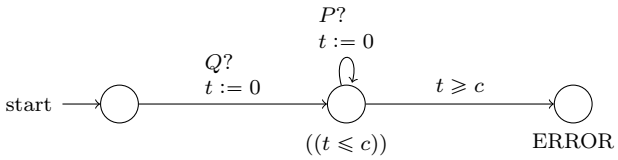
B.6.1 Globally: Recurrence P

Rule: *Forbidden ERROR*



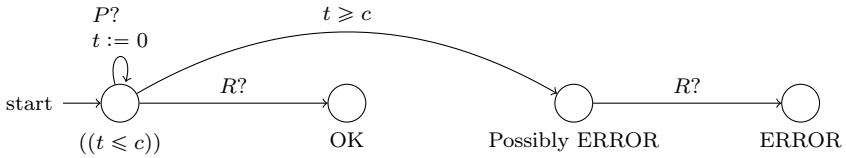
B.6.2 After Q: Recurrence P

Rule: *Forbidden ERROR*



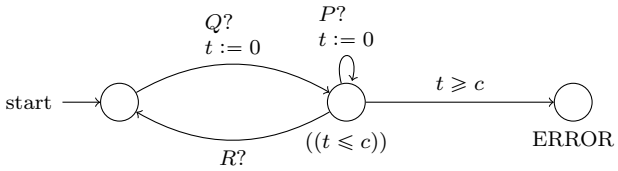
B.6.3 Before R: Recurrence P

Rule: *Forbidden ERROR*



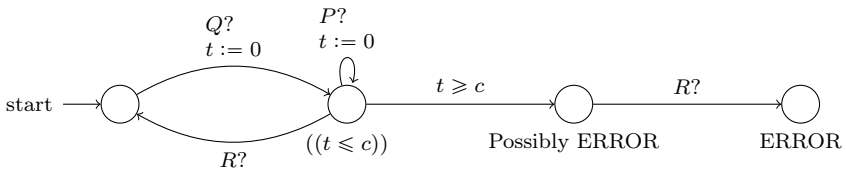
B.6.4 After Q until R: Recurrence P

Rule: *Forbidden ERROR*



B.6.5 Between Q and R: Recurrence P

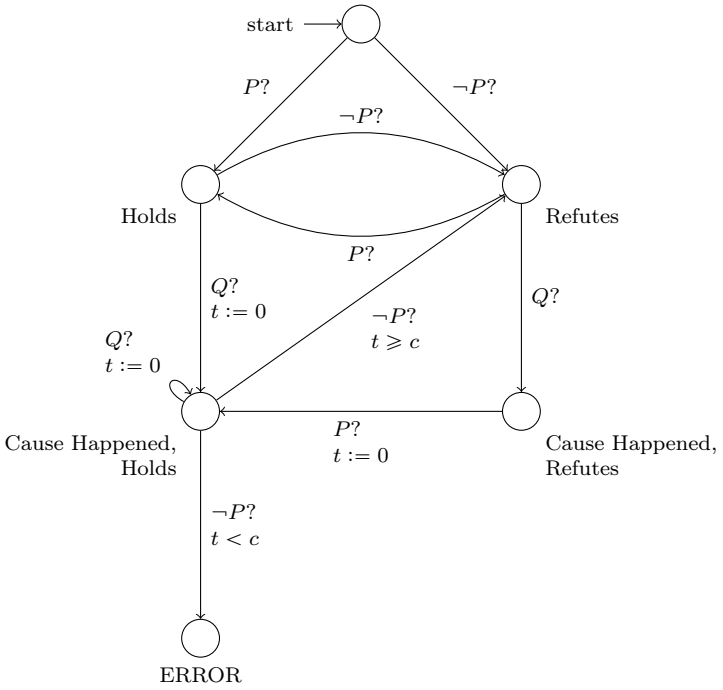
Rule: *Forbidden ERROR*



B.7 Invariance

B.7.1 Globally: Invariance P When Q

P must be a state proposition, rule: *Forbidden ERROR*



C Usability Exercises

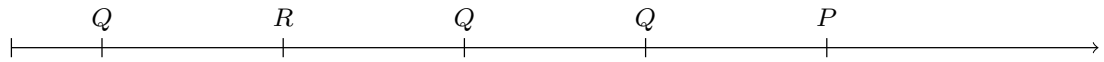
C.1 First Exercise

Consider the following sequences of events, assume no events occur after the ones shown, does the stated pattern/scope combination hold for this particular sequence and if not, why?

First example

4. Globally: Existence P
5. Before R: Absence P

Second example

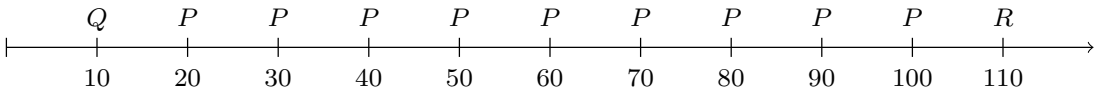


6. After Q Until R: Absence P
7. Between Q And R: Absence P

Third example

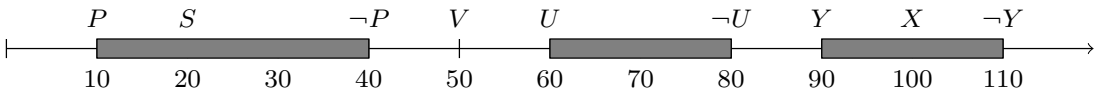
8. Globally: Response S After P
9. Globally: Precedence P Before S

Fourth example



10. After Q Until R: Recurrence P Every 15
11. After Q Until R: Recurrence P Every 5
12. After Q Until R: Recurrence P Every 10

Fifth example



13. Globally: Invariance P for 10 When S
14. Globally: Invariance U for 10 When V
15. Globally: Invariance Y for 10 When X