Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Hierarchical runtime verification for critical cyber-physical systems

Scientific Students' Associations Report

Authors:

László Balogh
Flórián Deé
Bálint Hegyi

Supervisors:

dr. István Ráth
dr. Dániel Varró
András Vörös

2015.

Contents

Contents ii

Abstract v

1 Introduction 1
- 1.1 Cyber-physical systems 1
- 1.2 Verification techniques 1
  - 1.2.1 Design time verification 2
  - 1.2.2 Runtime verification 2
- 1.3 Systems engineering methods 2
  - 1.3.1 V model 3
  - 1.3.2 Model driven software development 3
  - 1.3.3 Y model 4
  - 1.3.4 Modeling languages on different abstraction levels 5
- 1.4 Model based tools 6
  - 1.4.1 Eclipse Modeling Framework 6
  - 1.4.2 EMF-IncQuery 6
- 1.5 Complex event processing 6
  - 1.5.1 VIATRA-CEP 7
- 1.6 A hierarchical approach to runtime verification 7
  - 1.6.1 Low level monitoring 8
  - 1.6.2 High level monitoring 8
  - 1.6.3 Hierarchical monitoring approach 8

2 Statechart language 10
- 2.1 Parametric statechart declarations 10
- 2.2 Parametric signals 10
- 2.3 Overview 11
- 2.4 Language syntax 12
  - 2.4.1 Specification 12

2.4.2 Statecharts 12
2.4.3 Regions 13
2.4.4 State nodes 13
2.4.5 Transitions 15
2.4.6 Actions 16
2.4.7 Timing of transitions and actions 17
2.4.8 Signaling errors, error propagation 17
2.4.9 Expressions 17
2.4.10 Variables 18
2.5 Formal representation 18
2.5.1 Specification 19
2.5.2 Statecharts 19
2.5.3 Regions 19
2.5.4 States 19
2.5.5 Transitions 19
2.5.6 Signals 20
2.5.7 Timeouts 20
2.5.8 Variables and expressions 20
2.5.9 Verification 20

3 Synthesis of monitors **21**
3.1 Mapping of elements 21
3.1.1 Specification 21
3.1.2 Statecharts 21
3.1.3 State nodes 21
3.1.4 Transitions 22
3.1.5 Signals 23
3.1.6 Actions 23
3.1.7 Variables 23
3.2 Implementation 24
3.2.1 Timing related issues 24
3.2.2 Utility classes 25
3.2.3 Signal pushing 25
3.2.4 Error signaling 25

4 System Level Runtime Verification **26**
4.1 Complex Event Processing using technologies 26
4.2 Formal Intermediate language 27
4.2.1 Regular Expression 29
4.2.2 Timed Regular Expression 30

4.2.3 Parametric Timed Regular Expression 33
4.2.4 Parametric Timed Region Automaton 33
4.3 Examples of Complex Event Processing 35
4.3.1 File System 35
4.3.2 Mars Rover Tasking - Two phase locking 37
4.4 Implementation 37
4.4.1 Metamodel 37
4.4.2 Executor 38

5 Case study **39**
5.1 System level observation with computer vision 40
5.1.1 Hardware 40
5.1.2 OpenCV 40
5.1.3 Marker design 40
5.1.4 Mathematical solution for marker detection 41
5.1.5 Software 42
5.1.6 Summary 42
5.2 Model railroad 44
5.2.1 Overview 44
5.2.2 Hardware 45
5.3 Metamodel design 45
5.3.1 Physical elements 45
5.3.2 Metamodeling the physical elements 46
5.3.3 Using the Eclipse Modeling Framework 52
5.3.4 Building the EMF model 52
5.3.5 Using EMF-IncQuery 52
5.3.6 Building the IncQuery patterns 52
5.4 Component-monitor integration 54
5.4.1 Monitor statecharts 55
5.4.2 Generated VEPL 57
5.5 Summary 59

6 Conclusion **60**

References **62**

Összefoglalás Ipari becslések szerint 2020-ra 50 milliárdra nő a különféle okoseszközök száma, amelyek egymással és velünk kommunikálva komplex rendszert alkotnak a világhálón. A szinte korlátlan kapacitású számítási felhőbe azonban az egyszerű szenzorok és mobiltelefonok mellett azok a kritikus beágyazott rendszerek – autók, repülőgépek, gyógyászati berendezések - is bekapcsolódnak, amelyek működésén emberéletek múlnak. A kiberfizikai rendszerek radikálisan új lehetőségeket teremtenek : az egymással kommunikáló autók baleseteket előzhetnek meg, az intelligens épületek energiafogyasztása csökken.

A hagyományos kritikus beágyazott rendszerekben gyakorta alkalmazott módszer a futási idejű ellenőrzés. Ennek célja olyan ellenőrző programok szintézise, melyek segítségével felderíthető egy kritikus komponens hibás, a követelményektől eltérő viselkedése a rendszer működése közben.

Kiberfizikai rendszerekben a rendelkezésre álló számítási felhő adatfeldolgozó kapacitása, illetve a különféle szenzorok és beavatkozók lehetővé teszik, hogy több, egymásra hierarchikusan épülő, különböző megbízhatóságú és felelősségű ellenőrzési kört is megvalósíthassunk. Ennek értelmében a hagyományos, kritikus komponensek nagy megbízhatóságú monitorai lokális felelősségi körben működhetnek. Mindezek fölé (független és globális szenzoradatokra építve) olyan rendszerszintű monitorok is megalkothatók, amelyek ugyan kevésbé megbízhatóak, de a rendszerszintű hibát prediktíven, korábbi fázisban detektálhatják.

A TDK dolgozatban egy ilyen hierarchikus futási idejű ellenőrzést támogató, matematikailag precíz keretrendszert dolgoztunk ki, amely támogatja (1) a kritikus komponensek monitorainak automatikus szintézisét egy magasszintű állapotgép alapú formalizmusból kiindulva, (2) valamint rendszerszintű hierarchikus monitorok létrehozását komplex eseményfeldolgozás segítségével. A dolgozat eredményeit modellvasutak monitorozásának (valós terepasztalon is megvalósított) esettanulmányán keresztül demonstráljuk, amely többszintű ellenőrzés segítségével képes elkerülni a vonatok összeütközését.

Abstract According to industrial estimates, the number of various smart devices - communicating with either us or each other - will rise to 50 billion, forming one of the most complex systems on the world wide web. This network of nearly unlimited computing power will not only consist of simple sensors and mobile phones, but also cars, airplanes, and medical devices on which lives depend upon. Cyber-physical systems open up radically new opportunities: accidents can be avoided by cars communicating with each other, and the energy consumption of smart buildings can be drastically lower, just to name a few.

The traditional critical embedded systems often use runtime verification with the goal of synthesizing monitoring programs to discover faulty components, whose behaviour differ from that of the specification.

The computing and data-processing capabilities of cyber-physical systems, coupled with their sensors and actuators make it possible to create a hierarchical, layered structure of high-reliability monitoring components with various responsibilities. The traditional critical components' high-reliability monitors' responsibilities can be limited to a local scope. This allows the creation of system-level monitors based on independent and global sensory data. These monitors are less reliable, but can predict errors in earlier stages.

This paper describes a hierarchical, mathematically precise, runtime verification framework which supports (1) the critical components' monitors automatic synthetisation from a high-level statechart formalism, (2) as well as the creation of hierarchical, system-level monitors based on complex event-processing. The results are presented as a case study of the monitoring system of a model railway track, where collisions are avoided by using multi-level runtime verification.

Chapter 1

Introduction

1.1 Cyber-physical systems

The design of complex cyber-physical systems is an interdisciplinary process. From an engineering point of view, defining the requirements, designing sufficiently reliable components, dealing with scalability issues, employing testing processes that result in high test coverage, verifying safety critical components, and maintainability issues are all present at the same time. Systems engineering focuses on how to design and manage such systems. [17]

Safety-critical systems are systems whose failure could result in loss of life, significant property damage, or damage to the environment. There are many well known examples in areas such as medicine, flight control, weapons industries, and nuclear systems. Many modern information systems are becoming safety-critical in a general sense because financial loss and even loss of life can result from their failure. Safety-critical systems will be even more common and more influential in the future. From a software perspective, the growing number of such systems result in the need for faster development cycles, which will require significant advances in areas like specification, architecture, and verification, to meet the safety requirements. [16]

1.2 Verification techniques

As modern society is becoming more and more dependent on cyber-physical systems, the need for faultlessly working hardware and software increases. The development of safety critical systems require extensive testing efforts. Validation and verification methodologies have been present in the development processes of such systems for a long time [25], but faster and more reliable approaches are needed. Validation checks that the requirements specified for the software meet the needs of the user – as such, validation usually can be aided, but can't entirely be done by software. The

point of verification is to analyse whether the specified requirements are met. Methods for verification can be divided into two groups: design time verification and runtime verification. These approaches aren't exclusive, and their mutual use can support a more robust verification process.

1.2.1 Design time verification

Design time verification is a method used for finding errors of the system before deployment. Traditional software development methodologies usually rely on design time approaches. Verification methods can be applied on multiple levels, from small parts to the whole entirety of the system. These processes check the compliance of the system against the specification of the appropriate level.
Formal verification can also be used to give proof that the verified parts match the behaviour described by the (also formal) specification. On the other hand, applying formal methods usually have higher costs, and the verification of complex systems can be impossible due to the problem of state space explosion. As a result, the role of formal methods is to verify the correct behavior of system components, and not the entire system.

1.2.2 Runtime verification

Runtime verification is a method for the inspection of running systems. The motivation of the approach is the complexity of design time verification. As systems are getting larger, the application of formal methods are more and more limited as the resources needed for verification cannot be realized. This means that formal verification methods must verify an abstract model, not the deployed system itself. In addition, specifications are rarely complete, and design time methods can rarely handle hardware errors.
Runtime verification uses monitors to observe certain (usually critical) components, checking whether their operation violates properties described in the specification. The usage of monitoring components can result in significantly smaller monitors than the component itself as multiple levels of abstraction can be used, as long as the error states remain distinguishable. This has the advantage of detecting the erroneous operation of the system, and allows systematic safety engineering to handle faults, or trigger an emergency shutdown if necessary.

1.3 Systems engineering methods

Traditional software development methods often use informal specifications to develop system, architecture, and component level designs – which can also be informal. This can easily result in higher verification costs or faulty systems, making them suboptimal

choices for safety critical software development. To introduce a level of formality and allow manageable, hierarchical software testing procedures, the V model was developed.

1.3.1 V model

A concept of operations is one of the initial stages in a system life cycle based on the "Vee" diagram, illustrated in Figure 1.3.

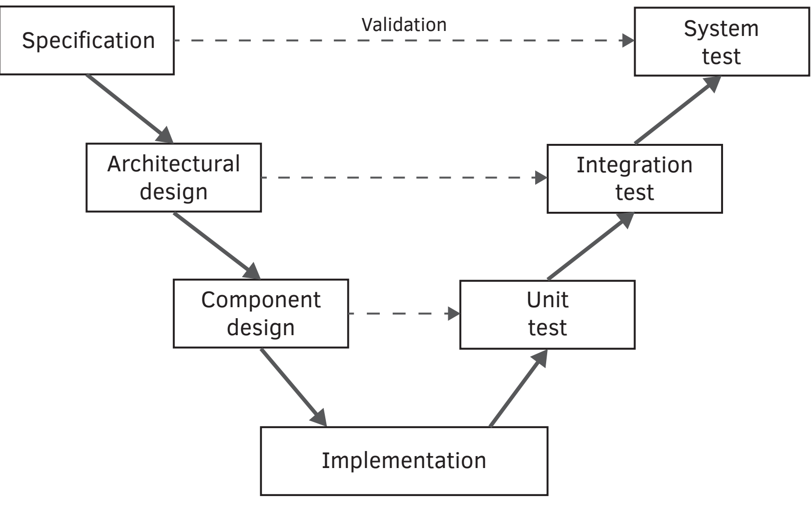


Figure 1.1 The traditional V model [18]

Figure 1.3 shows the stages of development, with a symbolic "V" showing the progression. The project definition stages on the left side begin with the development of a Concept of Operations, continue with Requirements and Architecture, and Detailed Design. The Implementation stage is shown across the base of the "V". The right side shows the testing and implementation stages of a system, with an upward-pointing arrow for progression. [18]

The V model is the basic scheme of software development. In the left of the figure, we proceed by decomposing the specification into an architecture level design, and the architectural design into component level designs. Every level of decomposition has it's own specification. After the implementation process, we verify every level's behavior with it's specification.

1.3.2 Model driven software development

Model driven software development (MDSD) emphasizes problem solving by the development and maintenance of models describing the system being designed. MDSD

heavily relies on automated code and documentation generation based on the models of components or the overall model of the system.

Modeling has the advantage of introducing abstractions, thus reducing the complexity of the development process. Code generation guarantees that the code will inherit the properties that can be directly derived from the model, while reducing the costs by eliminating unnecessary round-trip engineering. The generation of documentation also results in the always up-to-date description of components, stored together with the requirements and the model. Furthermore, model based approaches have the advantage of easier testability, or if the model is formal enough they can make formal verification possible. This is especially important for the development of safety critical systems, making MDSD notably widespread in such areas.

Various methods and tools are available for the generation of test cases and monitoring components from models, as well as for formally verifying certain properties. These tools usually support the modeling formalisms used in their application domain.

MDSD is usually accompanied by the Y model – a software life cycle model for component based systems [6], which is based on the V model.

1.3.3 Y model

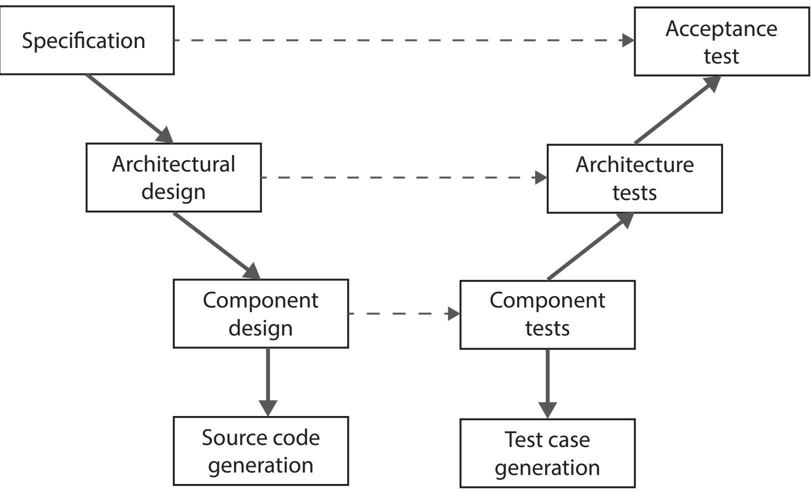


Figure 1.2 A section of the Y model

The Y model [6] is an extension of the V model (which is in turn an extension of the waterfall model) [21], by using code and test case generation. Much like the V model, the development process is partitioned vertically. Each level contains a model that is transformed to a verification model, on which formal methods can be applied.

The results of the verification process can be traced back to the original models making iterative improvement possible. The top level is for high level system models, while the second level contains architectural models, and the third one is for component based models. This provides input for the source code and configuration generation for the individual components. Test cases are paired with the source code and can be generated from the component verification models.

1.3.4 Modeling languages on different abstraction levels

MDSD methods require modeling languages to describe the behavior of systems and components. Engineering practices developed a wide range of such languages over the years to support fast paced product development. This allows the use of domain specific languages, which leads to a shorter modeling process but challenges formal verification software, as their input is usually stricter and in a more general format. The result is the need for complex model transformations before the verification can begin, which can lead to higher development costs, or – if the transformation contains errors – even faulty behaviour.
As a result, standardized modeling languages were developed like the UML (Unified Modeling Language [5]) and SysML (Systems Modeling Language [4]) languages.

Finite automaton

The modeling of systems with finite state space is often done by using finite automata – also known as finite state machines. A finite automaton accepts a (finite) list of symbols and produces a computation of the automaton for each input list. Although finite automata can be easily visualized, this formalism describes a simple, flat transition system and lacks the support for higher level concepts. The development of finite automata models are supported by many tools (e.g.: Finite State Machine Designer [11]).

Statechart

Statecharts, also known as state machines are an extension of finite automata. There are multiple available syntaxes for statecharts (e.g. the one defined by UML [12]). The higher level concepts that were introduced include variables, actions, and hierarchically nested states. Event-driven execution is also possible by using signals as the triggers of transitions. Available variable types heavily depend on the concrete semantics of the chosen statechart language. Actions can usually be variable assignments, signal raises, or the setting of timers. Hierarchy lets users organize system descriptions using a top-down approach. Support for hierarchy is introduced via nested states and parallel regions. States can also have entry and exit actions, which allows the description of

common functionality in parent states [20].
Statecharts are usually created by tools that support the graphical design of the model (e.g.: Yakindu [26], an Eclipse based editor).

Message Sequence Chart

The formalism of message sequence charts (MSC) describes the communication between components – the order in which messages can occur [15] [13]. The message interchange is usually represented by a graphical model. These charts can be used for high level specification, design, trace based testing, or documentation. A collection of possible sequence charts can also describe a complete communication protocol between components. UML sequence diagrams were inspired by MSCs, but their semantics differ regarding some of the basic elements of the language such as lifelines and arrows [14].

1.4 Model based tools

1.4.1 Eclipse Modeling Framework

The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. [9]

1.4.2 EMF-IncQuery

EMF-IncQuery is a framework for defining declarative graph queries over EMF models, and executing them efficiently without manual coding in an imperative programming language such as Java. [10]

1.5 Complex event processing

Complex event processing is a method of tracking and analysing streams of information and deriving conclusions. In a complex event processing environment, there can be multiple event sources, and with logical patterns given by a formalism, we can find patterns in the incoming stream, e.g. events followed by another events in some sequence.

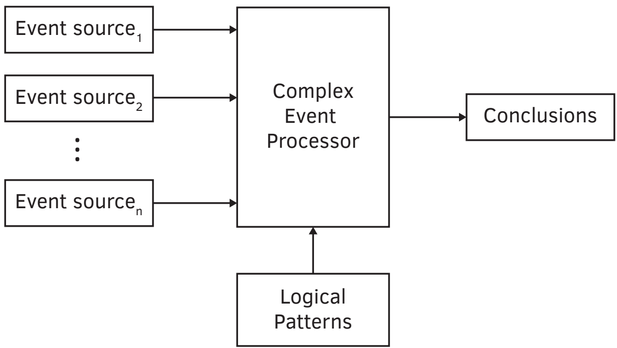


Figure 1.3 Complex event processing overview

1.5.1 VIATRA-CEP

CEP plays an important role in the model-driven engineering (MDE) as a supporting technique in various scenarios. The VIATRA project delivers a state-of-the-art event processing framework for the MDE scene, called VIATRA-CEP. [24]

The VIATRA-CEP is using the EMF models, and the EMF-IncQuery graph search engine to deliver a high throughput, model based complex event processing framework.

1.6 A hierarchical approach to runtime verification

We propose a hierarchical runtime verification approach by combining

1. traditional component-level monitors synthesized from statecharts with
2. architecture-level monitors that rely on complex event processing.

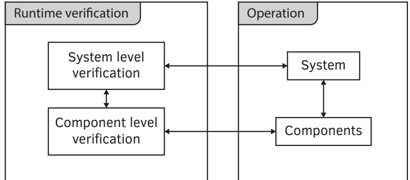


Figure 1.4 Overview of the hierarchical runtime verification system

The goals were:

- easy, high-level, and precise specification languages for capturing monitors

- the automated synthesis of monitors
- hierarchical monitoring

The result is a hierarchical runtime verification framework which can enable verification on two levels (Figure 1.4).

1.6.1 Low level monitoring

Low level monitoring of the components is achieved by monitors which can be generated from statecharts. The traditional formalism was expanded by adding support for statechart templates and parametric signals. This enables users to easily describe systems with homogeneous components. Erroneous states and transitions can be marked and specific handler functions can be written to deal with certain errors locally.

The monitors aim to be lightweight components, and as such use C++. After generation, they can be deployed to run in parallel with the safety critical component. The monitor has an interface to accept signals from the system, enabling the simulation of the desired behavior.

1.6.2 High level monitoring

The entirety of the system is monitored using a complex event processing engine. This can monitor the messaging between components and detected errors based on the specification of valid messaging sequences. The approach uses pattern matching to detect faulty operation of the system. Should illegal sequences occur, error handling operations (e.g. restart or shutdown of a component or the whole system) can be applied to avoid the violation of safety criteria.

1.6.3 Hierarchical monitoring approach

The low level components can communicate with the system level monitor. This high level monitor can than base it's decisions not only on the messaging sequences between components but also the the signals received from the component level monitors. This way, the system level monitor can decide whether the specified safety criteria are still met and act accordingly. The complete structure of this hierarchical approach can be seen in Figure 1.5.

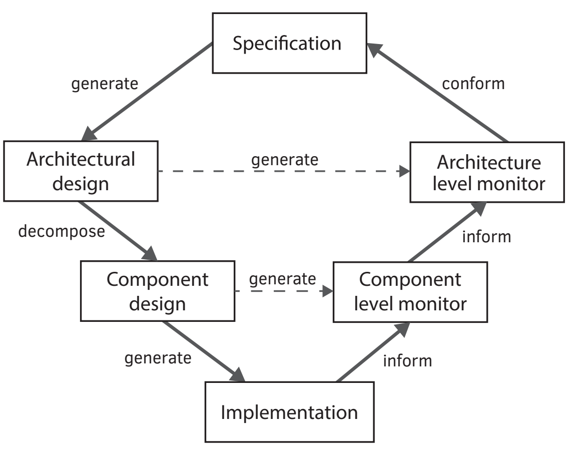


Figure 1.5 Overview of the hierarchical runtime verification system

Chapter 2

Statechart language

We propose a statechart language as the modeling method used for the runtime verification of components. Many software is available for component generation or verification based on statecharts. Unfortunately the available solutions for verification and modeling provide limited syntax, and no error marking methods. Both leads to longer development, harder maintenance, and higher costs. The developed statechart language supports the verification of statecharts with parametric signals and timers, and enables the creation of template based statecharts that can be instantiated with parameters.

2.1 Parametric statechart declarations

The language allows a specification to consist of multiple statecharts. This feature led to of the main strength of the language: the definition of statechart templates, which can be instantiated parametrically. This results in ‘ descriptions for otherwise complex, but homogeneous systems with many similar components. Statecharts can be parametrized by values of well known data types. Separate statecharts can communicate with each other using signals or global variables.

2.2 Parametric signals

Signals can also be parametrized with integer type variables. These parameters can then be used to discriminate between similarly named signal raises, which results in more readable code, by allowing transitions to use the same signal as their trigger, when their functionality is similar.

2.3 Overview

For a brief overview of the basic capabilities of the language, see Figure 2.1. The statechart describes a monitor for a round based game, where two players must press a button after the onStart signal was raised. The player who presses the button sooner wins. The game only ends after the second player has also made his move. The game enters a finish state and is restarted after 500 time units – which is also the time between the beginning of the game and the first round. The system uses a local variable to determine which player won. In addition to this, the hardware has the capabilities to detect whether a button is stuck during the game phase. Should such occasion occur, the monitor enters a state called Stuck, which is marked as an error state.

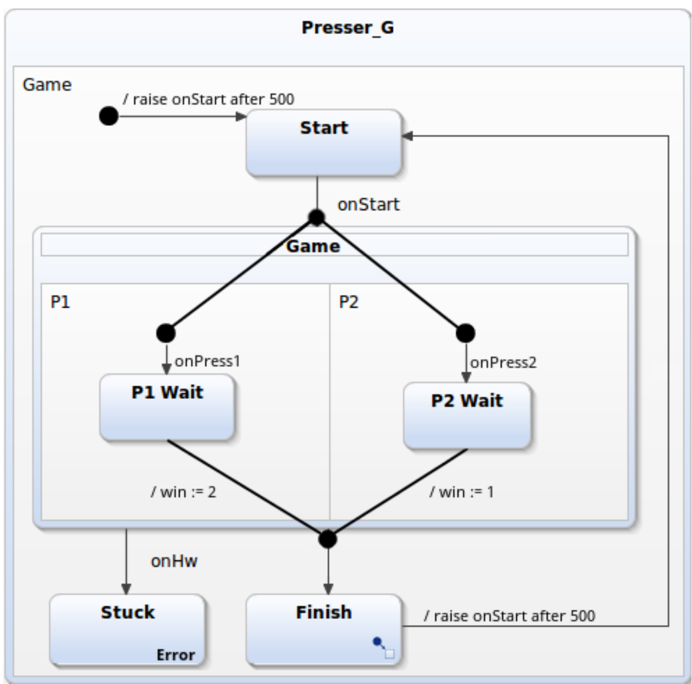


Figure 2.1 A simple statechart

The description of the whole system is enclosed in a specification, which can have multiple statecharts, signals, and global variable declarations. Each statechart consists of a region and an optional number of local variable declarations. Regions encapsulate states and transitions. Each region must have at least an initial state. A state itself can have an arbitrary number of inner regions, which will run in parallel. States can also have entry and exit actions associated with them. Transitions must have a source and a target state, and optional triggers, guard conditions, and actions. States and transitions can be marked with an error token to indicate that the execution of the transition or

the activation of the state is erroneous behaviour. Actions can be variable assignments and signal raises.

2.4 Language syntax

The language has a textual syntax that was designed to be simple without sacrificing expressiveness. The structure aims to be conform visual statechart languages, allowing one-to-one mapping where possible. Hierarchy is expressed by the scoping of elements. The naming of variables is similar to the conventions used by C-style programming languages.

2.4.1 Specification

Modelling the whole system as a single statechart would result in an overly complex statechart. Systems can usually be described as independent components that can communicate with each other. Therefore, an enclosing specification is used for the description of the whole system, within which statecharts can be declared as (mostly) independently operating components. Each specification can contain multiple statecharts. To enable communication between statecharts, global variable declarations and signal declarations are placed in the specification itself. For the model from Figure 2.1, the specification of the system is:

```
1 specification Game {
2   signal onStart
3   signal onPress1
4   signal onPress2
5   signal onHw
6   statechart...
7 }
```

2.4.2 Statecharts

The syntax for statechart definition would be in the form of:

`statechart PushGame(`*params*`) {...}`

where *params* is an optional parameter list, and `{...}` contains the description of the statechart itself. The parenthesis can be omitted if no parameters were specified. Parametrized statecharts can be created from existing templates by providing a value for each parameter and omitting the description. Definitions without parameters are treated as an instantiation of the statechart. The system described by Figure 2.1 has no

need for parameters, but should an integer type parameter be required, two statecharts could be instantiated as:

```
1 statechart PushInstance1(1) := PushGame
2 statechart PushInstance2(2) := PushGame
```

A specification must contain at least one instantiated statechart.

2.4.3 Regions

Statecharts are structured by the use of regions. At the root of every statechart, a region encloses all of the states. A state can have an arbitrary number of inner regions. This plays a fundamental part in the scoping of elements. A region can have both states and transitions. The syntax for a region is:

```
region Game {...}
```

where the `{...}` is the definition of the region's contents. Each region must contain at least an initial state for the model to be valid.

2.4.4 State nodes

Regions can contain multiple state nodes. A state node can either be a state or a pseudo state. States create the base structure of the model, while pseudo states help to describe functionality. Pseudo states can either be initial-, fork-, join-, or choice states.

Figure 2.1 contains a composite state, three initial states, a composite state with two regions, and four simple states and an error state.

States

States can either be atomic states or composite states. An atomic state is a state which does not contain inner regions. All states can contain entry and exit actions, which are executed when entering or exiting the state. Composite states contain one or more inner regions, each with at least an initial state. A state's parent state is it's containing region's containing state. If the region does not have a containing state, the parent is the region's containing statechart. Composite states allow users to maintain a clean model by the introduction of hierarchy, and the ability to describe common actions in the parent state.

These states could be described without the initial states and inner transitions as:

```
1 state Start
2 state Finish
3 state Game {
4   region P1 {
```

```
5       ...
6       state P1_Wait
7       ...
8     }
9     region P2 {
10      ...
11      state P2_Wait
12      ...
13    }
14  }
```

Initial states

Initial states can be found in all regions - if the region's containing state is entered, these inner states become active. An example for an initial state as used in the first inner region of the Game state is:

```
1  region P1 {
2    initial state init
3    ...
4  }
```

Choice states

Choice states are pseudo states whose outgoing transitions have exactly one of the guard conditions evaluating as true at all times. Choice states let the user create tree-like transition structures with simple guards conditions on each level, instead of rewriting similar, complex guard conditions using a single transition level. This usually results in more readable, modifiable, and expressive models. A choice state could be expressed as:

```
1  choice whichPlayerWon
```

Fork and join states

A fork state is a pseudo state that has a single incoming transition and any number of outgoing transitions. The outgoing transitions cannot have triggers, guards, or actions associated with them. If the incoming transition fires, all the outgoing transitions fire as well, and the fork state itself is not entered. This results in the simultaneous activation of multiple states.

A join state is a pseudo state that has a single outgoing transition and multiple incoming transitions. The incoming transitions cannot have triggers, guards, or actions associated with them. The outgoing transition is enabled when it's guard is true and

all the incoming transitions' source states are active. Triggers can be declared on the outgoing transition.

The system from Figure 2.1 has both a fork and a join state. They can be expressed as:

```
1 region Game {
2   fork startSignalTriggered
3   join allPlayersDone
4   ...
5 }
```

2.4.5 Transitions

Transitions describe the possible state changes. A transition can only occur if the source state is active. After the transition fires, the source state becomes inactive and the target state active. Furthermore, a transition can have a trigger, a guard condition, and an arbitrary number of actions associated with it. A simple transition is in the form of:

```
1 region Game {
2   ...
3   transition from Game to Start
4   ...
5 }
```

Transition triggers

Transitions with triggers can only fire when one of the triggering signals arrive. Defining triggers is optional. Enabled transitions without a trigger occur on the next timestep after the source state becomes active. The transition to the error state in Figure 2.1 has a trigger, which can be expressed as follows:

```
1 region Game {
2   ...
3   transition from Game to Stuck on onHw
4   ...
5 }
```

Transition guards

Transitions can have guard conditions, which are expressions that evaluate to a boolean value. If the guard condition evaluates to true, the transition is enabled, otherwise it is blocked. If we'd like to have transition that restarts the game when player one won, it would be (with no waiting):

```
1 region Game {
2   ...
```

```
3   transition from Finish to Start [win = 1]
4   ...
5 }
```

Transition actions

A transition can have any number of actions associated with it. These actions are performed when the transition fires.

2.4.6 Actions

Raising signals with or without a timeout, and variable assignments are called actions. They represent operations that might result in a change of the model's state.

Parametric signals

Statecharts can communicate with the outside world and each other using signals. These signals are declared directly in the specification. Signals can be used with a single integer parameter (which can be either a constant or a variable). This allows much simpler syntax when dealing with communication, as a statechart can raise a signal and pass a value simultaneusly. It also leaves room for a later expansion to a token based automata with re-entry. A parametric signal with the integer parameter 1 could be raised like:

```
1 region Game {
2   ...
3   transition from Finish to Start / raise onStart(1)
4   ...
5 }
```

where *1* could be an integer-type variable declaration. A signal can be referenced multiple times, but only one transition may be taken per statechart for a raised signal. This results in a clearer model.

Timed signals

Raising a signal can be offset by a certain amount of time. Apart from their delayed nature, timeout- and signal references can be used interchangeably. The starting timeout in Figure 2.1 can be raised by:

```
1 region Game {
2   ...
3   transition from init to Start / raise onStart after 500
4   ...
5 }
```

where *500* is the number of timesteps before the signal is raised.

Variable assignments

Variable assignments can be expressed with the syntax:

```
1 region P1 {
2   ...
3   transition from P1_Wait to forkState / assign win := 2
4   ...
5 }
```

An assignment's left hand side has to evaluate to an assignable reference, such as a variable or an array element. The right hand side can be any valid expression.

2.4.7 Timing of transitions and actions

Transitions are fired one by one. The firing of a transition means that the triggering signal is consumed. This can result in non-deterministic runs if two transitions share the same trigger and can be enabled simultaneously. Such models can be created but should be avoided, as the order of the transitions is not guaranteed. Actions related to the current transition being taken are all executed in a single step.

2.4.8 Signaling errors, error propagation

States and transitions can be labelled as errors. For states, the syntax is:

```
1 region Game {
2   ...
3   state [Error buttonStuck] Stuck
4   ...
5 }
```

For transitions, an example would be:

```
1 region Game {
2   ...
3   transition [Error buttonStuck] from Game to Stuck
4   ...
5 }
```

The `buttonStuck` string is a description given by the user. This description is only used for the generation of error messages.

2.4.9 Expressions

Variables can be used in expressions. Expressions can have an arbitrarily complex structure within the limits of the transition system described in [23]. This allows the use of array indexing, parenthesis, and common operators in programming languages such as +, -, *, / – just to mention a few. Assignments' left hand sides must reference a

single variable while their right hand side is an expression. Logical expressions are also available (for example expressions using comparison operators). Each expression is a mixture of variables, constants, and operators. For a full reference, see [23].

2.4.10 Variables

Variables can either be global (accessible to all statecharts) or local (bound to a single statechart in which they were declared). Many types are supported, like characters, integers, and doubles. For a complete list, see [23]. Variable declarations are in the form of:

```
1  statechart Game {
2    local var win : integer
3    ...
4  }
```

where local (or global) denotes the scope of the variable of any of the supported types.

2.5 Formal representation

To enable formal verification of the defined systems, a mapping tool was developed that can generate transition systems (with the syntax described in [23]) from statechart specifications. This means that complex concepts of the model have to be flattened out. The transition system used has the following major capabilities:

- multiple independent transition systems can be defined
- these systems are enclosed in a specification
- transition systems can use local and global variables
- the state of a transition system can be defined as a vector of all referenced variables
- multiple states can be represented by expressions that restrict certain variables' values and leaves the rest of them unbound
- such restrictions are logical formulas with variables
- transition systems have transitions that represent changes of state using the referenced variables previous and current values

2.5.1 Specification

The specification of the statechart language is mapped to the specification of the transition system. Global variables cannot be declared in the transition specification itself, they must be referenced in each transition system that uses them. Signals will be mapped to global variables.

2.5.2 Statecharts

Each statechart is mapped to a separate transition system. These transition systems model systems running in parallel, which models the behaviour of the separate statecharts - they define systems that can communicate with each other but are running independently. Local variables in statecharts have the same local representation after mapped, and need no conversion.

2.5.3 Regions

A transition system does not support hierarchical structures – they are flat models. This means that all regions, states, and transitions inside a statechart are mapped directly into a transition system.

2.5.4 States

States are represented as boolean variables that indicate whether a state is currently active or not. As entry and exit actions are unknown concepts in a transition system, these actions are propagated to all of the incoming and outgoing transitions of the state. This way any transition that would result in the entering or exiting of the state executes the appropriate actions.

Composite states entry and exit actions and their incoming and outgoing transitions are propagated to the atomic states inside them. This means that only atomic states are represented in the transition systems.

2.5.5 Transitions

Transitions between states are transitions in the transition system too. For a transition to be enabled, guard conditions have to evaluate as `TRUE` for the current state of the system, the source state has to be enabled (meaning that the source state's boolean variable is true), and the transition should have no trigger or the triggering signal's boolean variable has to be true (representing that it was raised). When the transition fires in the formal model:

1. the exit actions,

2. the actions associated with the transition,
3. and the entry actions of the appropriate states occur.

2.5.6 Signals

Signals can be used for communication between statecharts. As such, each transition system that references a signal needs to be able to check whether the signal was raised. Therefore, in the formal model signals are represented as a set of global boolean variables. The variable is true if the signal has been raised since the previous timestep. As each statechart referencing the same signal might react to it by the firing of a transition triggered by the signal, a separate global boolean is used for each transition system. Since each signal can only trigger a single transition inside any given statechart, more complexer solutions are not needed. As signals are stored as booleans, the parameters are stored and can be referenced as global integer variables.

2.5.7 Timeouts

Timeouts are signals that are raised with an offset in the time domain. Handling timeouts in a transition system is a well researched area [8]. A separate timing system is created as a transition system to generate simulate timesteps. The system has a variable that stores the current time and each signal has a helper variable that represents when the signal has to be raised next. If a timeout is set, the offset of the timeout is added to the current time and this value is assigned to the helper variable. The timing system has a constantly enabled transition which increments the current time if no timeout can occur. For each timeout a transition is defined that is enabled when the current time is equal to the time stored in the helper variable. When this transition happens, the signal is raised. The helper variables are set to -1 by default to prevent false signal raises based on timeouts.

2.5.8 Variables and expressions

Variable types are the same as in the statechart model. Global variables remain global ones, and local variables that are declared inside statecharts are mapped to local variables inside systems. Expressions need no further flattening as the expression library used by the transition system language and the statechart language is the same.

2.5.9 Verification

The verification of the mapped system is based on LTL or CTL expressions. These can be defined at the end of the mapped specification. By default, the non-reachability of error states and transitions are the only verified properties.

Chapter 3

Synthesis of monitors

Current methods for statechart based code generation usually result in poor quality code. We propose a method for monitor generation based on the statechart language described in Chapter 2 on page 10. Our approach was to generate easily readable, extensible, object oriented `C++` code, that can run in environments with limited resources – primarily embedded systems.

3.1 Mapping of elements

3.1.1 Specification

A specification consists of separate statecharts. To simplifying their handling, a class was created called `StatechartRegistry` that can automatically iterate through each statechart and update their state one timestep at a time. This class is also responsible for the initialization process of the monitor. The name of the class does not represent the whole specification as global variables and signals are handled by utility classes.

3.1.2 Statecharts

Statecharts are represented as classes with lists of all their transitions, states, and currently active states. Local variables are mapped to global ones for unified usage. The statecharts' names are also stored for the error signaling process. Statecharts have functions for calculating their enabled transitions, taking enabled transitions, and maintaining the list of active states.

3.1.3 State nodes

State nodes represent states, initial states, join states, and fork states. Their mapping highly depends on their type.

States

Atomic or complex states are represented by named objects. All of these objects are derived from the `State` class. This object represents a state that has no entry and exit actions. If a state has entry or exit actions, a child class is generated where the appropriate entry and exit actions are represented as overrides for the `Entry()` or `Exit()` function. Classes are instantiated using the states' fully qualified names.

Initial states

Initial states are mapped to `State` objects. This general class also has a boolean variable that is set to `TRUE` for initial states.

Fork, join, and choice states

Fork, join, and choice states are not represented in the monitoring object. These state nodes modify the behavior of transitions, and are handled when mapping transitions.

3.1.4 Transitions

Transitions are represented as objects connecting two states. An instance of the generic `Transition` class has an `isEnabled()` function that always returns true, and an empty `action()` function that simply returns. A null-initialized list reference is also present that can point to a list of states. The list itself is only created for fork and join states to minimize memory usage. The triggers of the transitions are handled in a separate mapper class and are play no role in the creation of transition objects.

Simple transitions

Transitions without actions or guards can be instances of the generic `Transition` class, while transitions with guard conditions or actions are it's children with overloaded functions. Simple transitions have one source and one target state specified when instantiating them.

Transitions of choice states

Transitions from, and to choice states are handled by a statechart preprocessor that unfolds choices to simple transitions with the needed guard conditions to conserve functionality. The call to the preprocessor is the first step of the monitor generation.

Transitions of fork and join states

Incoming and outgoing transitions of a fork state are mapped to a single transition with a target state reference of null. The fork's outgoing transitions are stored in the separate list of state references. A join state's transitions are also mapped to a single transition with a source state reference of null. The fork's incoming transitions are stored in the separate list of state references.

3.1.5 Signals

Signals are represented as instances of the `Signal` class, which is a class that describes a named object containing an integer value. Raising a signal means creating an instance with the appropriate name and parameter and putting it in a queue for processing in the next timestep. Signals with a timeout are stored in a separate row until they should be raised. Parameterless signals are signals with a parameter of a predefined, unused value.

3.1.6 Actions

Raising signals and timeouts

Signals can be raised by creating a signal object and placing it in the queue for arrived signals. Timeouts are signal raises that will occur in the future. When raising a timeout, the appropriate signal is created and put in a queue that holds pairs of signals and timestamps. The timestamp represents when the signal should go off.

Variables and expressions

The assignments and expressions that can be used in the statechart language are a subset of the `C++` language, so the mapping between the statechart language and the generated monitor adds no restrictions and require no extra constructs.

3.1.7 Variables

The variable types of the statechart language are plain old data types in C++. Local variables are converted to global variables with fully qualified names. This allows a variable container class to be created which holds all of the variables and allows unified usage.

3.2 Implementation

The implementation of the monitoring component can raise many questions about timing issues. To generate working, and easily usable code additional helper classes and extra functions for the already described ones are required. As such, the most important utility classes are also described in this section.

3.2.1 Timing related issues

The generated monitoring component will run on a real hardware. This means that the system can only take a limited number of transitions per second. Aiming for a lightweight component, there is a settable sleep and timeout period. The default implementation of the utility classes enables the user to set this period with millisecond precision.

For each timestep, the monitor:

1. wakes up
2. checks the future signals queue for elapsed signals
3. puts such signals in the arrived signal's queue
4. selects a statechart
5. checks for transitions that can be fired
6. fires one of them and apply the related actions
7. updates the list of active states
8. repeats from step 5 until no transition is enabled
9. repeats from step 4 until no statechart is left
10. goes to sleep for the set sleep period

Raising signals while taking a transition puts the raised signal in a separate row that will be switched for the currently used one before going to sleep again. This means that only those transitions are taken whose triggers arrived in the queue before waking up. A signal with a timeout of x means that the signal will be raised on the next awakening after x milliseconds will have passed.

3.2.2 Utility classes

A few utility classes are needed to keep the structure of the code more readable, or to make the functionality more customizable. The most important classes are listed under this section.

SignalRegistry

A class called `SignalRegistry` was introduced to create a thread-safe wrapper for signals, and a maintainer of signals with timeouts.

VariableRegistry

All variables are treated as global variables. For availability and enclosure, we choose to create a centralized class with static variables resembling each variable found in the original statechart descriptions.

Timestamp

For easily replaceable timing settings, the handling of timing properties is done by a separate `Timestamp` class. The default implementation is to use millisecond-resolution functions and classes of `std::chrono`.

3.2.3 Signal pushing

The monitoring component starts in a separate thread. This means that calling the `start` function will only block the current thread as long as the initialization process is running. After starting the monitor, the `SignalRegistry::SignalArrived(`*param*`)` function can be called to inject a signal with the name *param* into the system from the outside world. Naturally, the `SignalRegistry` class is thread-safe.

3.2.4 Error signaling

The `OnError(`*param*`)` function is called upon reaching an error state or transition. The *param* is equal to the specified message defined in the statechart model, or in case of states that have not defined a message, the fully qualified name of the error state. The function should be modified by the user to implement the necessary error signalling steps of the system.

Chapter 4

System Level Runtime Verification

In this chapter we summarize the basic concepts of System Level Runtime Verification, overview of the Complex Event Processing approach and we suggest an intermediate language as a formal framework of our approach

4.1 Complex Event Processing using technologies

In the proposed hierarchical runtime verification approach, the top level of modelling and analysis is supported by a Complex Event Processing Framework.

In our project we are going to do so, and define Event Patterns at the system level. Our choice for CEP is the VIATRA CEP, because it's the only open source CEP with

- a live model integration,
- where you can define graph patterns to the model, and automatically generate atomic events on the appearance and disappearance of these patterns.

VIATRA CEP uses a language called VIATRA Event Pattern Language (VEPL for short). This language is perfect for our approach, it has intuitive syntax, and the event patterns can be defined in a very high level. However, it is not easy to formalize as the execution of the patterns are dependent on so called "event contexts" which makes formalization extremely difficult.

We propose an intermediate language, to formalize the system level runtime verification task. The proposed formalism supports the formalization of the used CEP language. In addition, it also makes the framework more extensible as other engineering models can be integrated above the intermediate language.

The operators of the language are shown in Table 4.1. The syntactic sugars are shown in Table 4.2, but we are going to ignore these, since they can be translated to the basic operators as shown in Table 4.3.

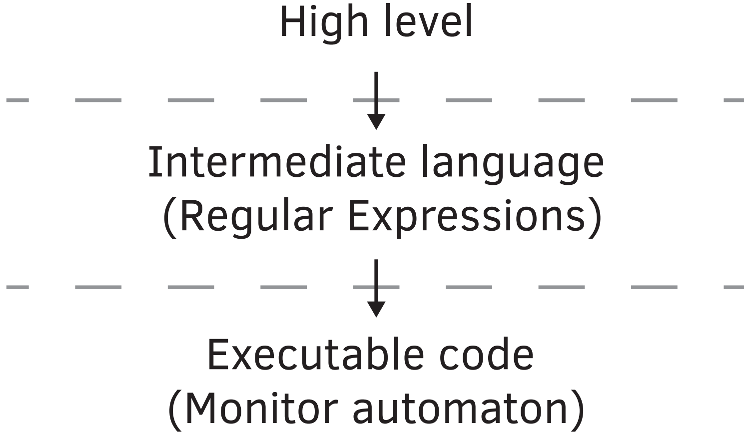


Figure 4.1 Formal intermediate language

We are going to introduce every step through an example, which is about the runtime verification of a two phase lock algorithm. In the two phase locking algorithm there are two rules :

1. Resources must be allocated in a previously defined sequence, which is the same for all tasks
2. If a task releases a resource, it's not allowed to allocate anything anymore.

Since this example uses resources, we have to define the following behaviour: each resource can be allocated once before every release, and can be released once before every allocation.

To show the usefulness of the expressiveness of our language in critical systems, where timeouts are common, we will extend the requirements: The first phase must finish after 10 seconds, i.e. the time between the first allocation and the first release must be less than 10 seconds[1].

4.2 Formal Intermediate language

In this section, we overview formalisms from the literature, and suggest parametric timed region automaton as an intermediate formalism. As many of the high level specification languages like VEPL and Sequence charts can be characterized with the

[1]In the final example we are going to change this, to "A resource can not be in allocated state for more than 10 sec", but to only have one automaton, we will use this rule right now.

Table 4.1 Basic operators

Operator name	Denotation	Meaning
followed by	$p_1 \rightarrow p_2$	Both patterns have to appear in the specified order.
or	p_1 OR p_2	One of the patterns has to appear.
"infinite" multiplicity	$p\{*\}$	The pattern can appear 0 to infinite times.
within timewindow	$p[t]$	Once the first element of the pattern is observed (i.e. the patterns "starts to build up"), the rest of the pattern has to be observed within t milliseconds.

Table 4.2 Syntactic sugars

Operator name	Denotation	Meaning
and	p_1 AND p_2	Both of the patterns has to appear, but the order does not matter.
negation	NOT p	On atomic pattern: event instance with the given type must not occur. On complex pattern: the pattern must not match.
multiplicity	$p\{n\}$	The pattern has to appear n times, where n is a positive integer.
"at least once" multiplicity	$p\{+\}$	The pattern has to appear at least once.

Table 4.3 Syntactic sugars mapped to basic operators

Operator name	Denotation	Equivalent
and	p_1 AND p_2	$((p_1 \rightarrow p_2)$ OR $(p_2 \rightarrow p_1))$.
negation	NOT p	$\Sigma \setminus p$, where Σ is the set of all the possible Events.
multiplicity	p\{n\}	$p \rightarrow p \rightarrow \ldots p$, n times.
"at least once" multiplicity	$p\{+\}$	$p \rightarrow p\{*\}$

help of regular languages, this was the motivation behind our research. Our findings are similar to those of paper [7].

4.2.1 Regular Expression

In this particular example, we need a language to describe event sequences. To do so, one of the most common formalism is regular expressions, where the alphabet is the set of possible events.

Definition 4.1 Regular Expressions over an alphabet Σ (also referred to as Σ-expressions) are defined using the following families of rules.

1. $\underline{a}$ for every letter $a \in \Sigma$ and the special symbol ε are expressions.
2. If $\varphi, \varphi_1, \varphi_2$ are Σ-expressions then $\varphi_1 \cdot \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \wedge \varphi_2, \varphi^*$ are Σ-expressions.

To illustrate the usage of the regular expressions in the two phase locking example we are going to describe the behaviour of one resource with a simple VEPL pattern: $(a \rightarrow r)\{*\}$[2] which means that an infinite sequences of allocation and release are allowed after each other. This is equivalent to the regular expression : $(a \cdot r)^*$.

Deterministic Finite Automaton

To match regular expressions, the most common solution is to construct an automaton accepting the language generated by the regular expressions. Variuous algorithms exist for the generation of deterministic finite automaton from regular expressions

Definition 4.2 An Event Automaton (Deterministic Finite Event Automaton in other words) is a tuple $\langle Q, \Sigma, \delta_d, q_0, F \rangle$ where:

- Q is a finite, nonempty set. These are the states of the automaton,
- Σ is a finite, nonempty set. This is the event set of the automaton,
- δ_d is a subset of tuples $\langle Q \times \Sigma \times Q \rangle$, and the number of outgoing edges from each state for each event is only one i.e. $\forall q_0 \in Q$ and $\forall e_0 \in \Sigma : |\langle q_0, e_0, q_1 \rangle| = 1$, where $q_1 \in Q$
- $q_0 \in Q$ the initial state,
- $F \subseteq Q$ the set of the accepting states.

[2] a stands for allocating a resource and r stands for releasing a resource

Just to illustrate the operation of the Finite Automata we can use a token assigned to the active state.

The semantic is the following : At initialization the token is at state f_0. If the token enters state s' where $s' \in F$ then we accept the event trace. When receiving input e, where $e \in \Sigma$, if the token is on state s the next state will be s' where $\delta_d \langle s, e, s' \rangle$. For short, from now we will use the notation $s \rightarrow s'$.

The regular expression of the example can be compiled to the event automaton of Figure 4.2.

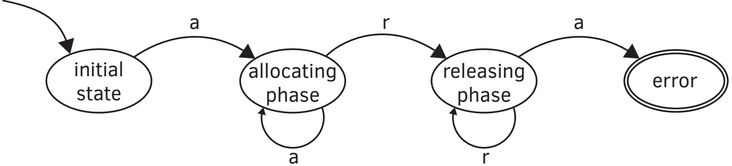


Figure 4.2 Event automaton for the two phase locking example

4.2.2 Timed Regular Expression

Using the previously defined semantics we can not express timed properties, but regular expressions can be expanded to a formalism which can do so : the Timed Regular Expressions

Definition 4.3 Timed Regular Expressions over an alphabet Σ (also referred to as Σ-expressions) are defined using the following rules [1] .

1. $\underline{a}$ for every letter $a \in \Sigma$ and the special symbol ε are expressions.

2. If $\varphi, \varphi_1, \varphi_2$ are Σ-expressions then $\varphi_1 \cdot \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \wedge \varphi_2, \varphi^*, \langle \varphi \rangle_I$, where I is an Interval, are Σ-expressions.

With the Timed Regular Expression formalism we can extend our example to the next level, and add the timeout. The new expression will be : $(a * r * a)|(\langle a* \rangle_{t<10s} r*)$

Timed Event Automaton

For accepting languages generated by timed regular expressions, we introduce the timed event automaton.

Definition 4.4 A Timed Event Automaton $\langle Q, \Sigma, \delta, q_0, F, t, T \rangle$ where

- Q, Σ, q_0, and F are the same as in Definition 4.2,

- t is a global clock variable $t \in \mathbb{R}$,
- T is a set of local timeout clock variables, i.e. a subset of tuples $\langle Q, \mathbb{R} \rangle$ assigning timeouts to states.
- and δ is the union of discrete and timed transitions i.e. $\delta_t \cup \delta_d$ where
  - δ_d is defined as in Definition 4.2,
  - and δ_t represents timed transitions and defined as the set of tuples $\langle Q \times \mathbb{R} \times Q \rangle$

The semantic of the Timed Deterministic Finite Automaton is defined as follows:

$Q_t \subseteq Q$ is the set of states with outgoing timed transitions, i.e. $\forall s \in Q_t : \exists \delta_t \langle s, t, s' \rangle$, where $t \in \mathbb{R}$ and $s' \in Q$. We have to define rules for entering states with timed outgoing transitions and we also define the general rules of changing states.

1. Initialization Rule : On initialization of the automaton, we have to set all clocks to ∞ i.e. $\forall t_i \forall q_i T \langle q_i, t_i \rangle, t_i := \infty$
2. Entering Timed State Rule: On entry to state s where $s \in Q_t$ the timeout variable t_s of the state is set according to the value of the global time and the timeout value of the output transition $t_{timeout}$: $t_s := t + t_{timeout}$ where $T \langle s, t_s \rangle$ and $\delta_t \langle s, t_{timeouts}, s' \rangle$ where $t_{timeout}$ is minimal from the set of possible $t_{timeouts}$
3. Firing Transitions Rule: Non-deterministically choose an enabled transition from the set of enabled discrete or timed transitions. s is the currently active state, and s' is the next state according to δ. We have two cases, the chosen transition is:
   a) Discrete Transition: In case of $s' \notin Q_t$ than the execution of the transition is as described formerly. If we exit state $s \in Q_t$ by a transition in δ_d, then the following rule extends the firing rule of discrete transitions by invalidating the corresponding timeout values: $t_s := \infty$, where $T \langle t_s, s \rangle$
   b) Timed Transition: The transition with the minimal timeout value is selected, i.e. transition δ_t from state s_t where $\forall q \in Q_t : t_q \geq t_s$, than the following rules apply: the global time is set $t := t_s$, the local clock is set to infinity: $t_s := \infty$ and move to through $\delta_t : s \rightarrow s'$ the next state according to δ_t.

Timed Region Automaton

We add a syntactic sugar to ease the compilation of high level languages to this intermediate language. Using regions in our automaton language has the same motivation as the application of regions in state chart formalisms.

Definition 4.5 A Timed Region Event Automaton $\langle Q, \Sigma, \delta, q_0, F, t, T \rangle$ where

- $Q, \Sigma, q_0, F,$ and t are the same as in Definition 4.4,
- T is a set of timeout clock variables for sets of states, i.e. a set of tuples $\langle Reg, \mathbb{R} \rangle$
- and δ is the union of discrete and timed transitions i.e. $\delta_t \cup \delta_d$ where
  - δ_d is defined as in Definition 4.4,
  - and δ_t represents timed transitions and defined as the set of tuples $\langle R, \mathbb{R}, Q \rangle$
- For Syntactic sugar we are going to use R as short for Regions, the set of which have outgoing timed transitions i.e. $\forall R_i \in R, \exists t, \exists q \subseteq \delta_t \langle R_i, t, q \rangle$

The semantic of the Timed Region Automaton is defined as follows: $Q_t \subseteq Q$ is the set of states with outgoing timed transitions, i.e. $\forall q \in Q_t : q \in R$.

Let us use the following notations:
s is the currently active state, and s' is the next state according to δ.
r is the set of currently active regions, i.e. $r \subseteq R$ where $\exists r_s : r_s \in r, s \in r_s$
r' is the set of regions we enter, i.e. $r \subseteq R$ where $\exists r_s : r_s \in r, s' \in r_s$
$r^+ \subseteq R$ is a set of new timed regions we just entered, i.e. $r^+ = r' \setminus r$
$r^- \subseteq R$ is a set of timed regions we just left i.e. $r^- = r \setminus r'$

We have to define rules for the initialization of the automaton, entering states with timed outgoing transitions and we also define the general rules of changing states.

1. Initialization Rule : At the initialization of the automaton, we have to set all clock variables to ∞ i.e. $\forall t_i, \forall q_i, T\langle r_i, t_i \rangle, t_i := \infty$, where $r_i \in R$
2. Entering new timed region rule : If we enter a new set of timed regions, i.e. $r^+ \neq \emptyset$, we set the timers according to the timeouts, i.e. $\forall t_{timeout} : t_{timeout} := t + t_i$ where $r_t \in r^+, \exists q, \delta_t \langle r_t, t_i, q \rangle, T \langle r_t, t_{timeout} \rangle$
3. Firing Transition Rule: Choose an enabled transition from the set of enabled discrete or timed transitions. We have two cases, the chosen transition is:
   a) Discrete Transition: In case of $r^- = \emptyset$ than the execution of the transition is as in described formerly. If we exit a region i.e. $r^- \neq \emptyset$, then the following rule extends the firing rule of discrete transitions: $\forall t_s, \forall q_s$ in $\delta_t \langle r_i, t_s, q_s \rangle$ where $r_i \in r^-$, the timer of the regions are invalidated i.e. $t_s := \infty$
   b) Timed Transition: The transition with the minimal timeout value is selected, i.e. transition $\exists t_i, \exists s_i, \delta_t \langle r_i, t_i, s_i \rangle$ where $r_i \in r$ and t_{min} is the minimum from

all t_i, than the following rules apply: the global time is set $t := t_{min}$, the local clock is set to infinity: $t_s := \infty$ and move to the next state according to δ_t.

Using the Timed Region Automaton we can compile our timed regular expression $(a * r * a)|(\langle a* \rangle_{t<10s} r*)$ of Figure 4.3

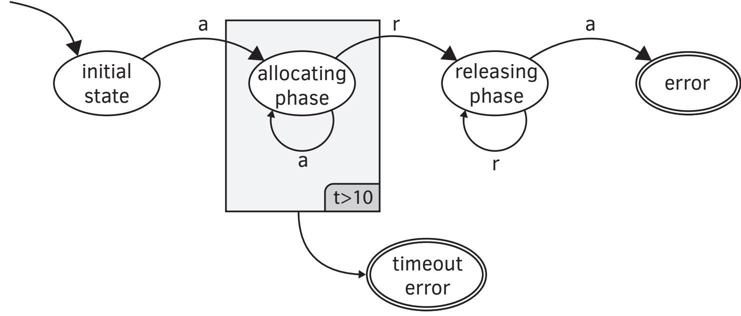


Figure 4.3 Timed region event automaton for the two phase locking example

4.2.3 Parametric Timed Regular Expression

Complex systems usually have parametric behaviours, which can be expressed by formalisms extended with parameters. In this section we overview such formalisms that generate the parametric timed regular languages and an automaton formalism accepting them.

Definition 4.6 A Parametric Timed Regular Expression is a timed regular expression where the set of events are defined as a set of tuples. $\langle \Sigma, P \rangle$, where Σ is defined as formerly and P is a set of parameters.

$$(a(task, {}_resource)*r(task, {}_resource)*a(task, {}_resource))|(\langle a(task, {}_resource)* \rangle_{t<10s} r(task, {}_resource)*)$$

4.2.4 Parametric Timed Region Automaton

According to [2] we use s to denote a tuple $\langle s_0, \ldots, s_k \rangle$. We use $X \rightarrow Y$ and $\rightharpoonup$ to denote sets of total and partial function between X and Y, respectively. We write maps (partial functions) as $[x_0 \mapsto v_0, \ldots, x_i \mapsto v_i]$ and the empty maps as $[]$. Given two maps A and B, the map override operator is defined as:

$$(A \dagger B)(x) = \begin{cases} B(x) & \text{if } x \in \underline{\text{dom}}(\text{B}) \\ A(x) & \text{if } x \notin \underline{\text{dom}}(\text{B}) \text{ and } x \in \underline{\text{dom}}(\text{A}) \\ \text{undefined otherwise.} & \end{cases}$$

Definition 4.7 (Symbols, Events, Alphabets and Traces). Let $Sym = Val \cup Var$ be the set of all symbols (variables or values). An event is a pair $\langle e, \bar{s} \rangle \in \Sigma \times Sym^*$, written $e(\bar{s})$. An event $e(\bar{s})$ is ground if $\bar{s} \in Val^*$. Let Event be the set of all events and GEvent be the set of all ground events. A trace is a finite sequence of ground events. Let Trace = $GEvent^*$ be the set of all traces[2].

Definition 4.8 (Substitution). The binding $\theta = [x_0 \mapsto v_0, \ldots, x_i \mapsto v_i]$ can be applied to a symbol s and to an event $e(\bar{s})$ as follows:

$$s(\theta) = \begin{cases} \theta(s) & \text{if } s \in \underline{dom}(\theta) \\ s & \text{otherwise} \end{cases} \qquad e\langle s_0, \ldots, s_j \rangle(\theta) = e\langle s_0(\theta), \ldots, s_j(\theta) \rangle$$

Definition 4.9 (Matching). Given a ground event a an event b the predicate matches (a, b) hold if there exists a binding θ s.t. $b(\theta) = a$. Moreover let matches(a, b) denote the smallest such binding w.r.t $\sqsubseteq$ iff it exists (and is undefined otherwise)

Definition 4.10 (Configurations and Transition Relation). We define configurations as elements o the set Config = Q × Bind. Let $\rightarrow \subseteq$ Config × GEvent × Config be a relation on configurations s.t. configurations $\langle q, \varphi \rangle$ and $\langle q', \varphi' \rangle$ are related by the ground event a, written $\langle q, \varphi \rangle \xrightarrow{\text{a}} \langle q' \varphi' \rangle$ if, and only if

$$\exists b \in \mathcal{A}, \exists \gamma \in Assign : (q, b, \gamma, q') \in \delta \wedge$$
$$matches(a, b) \wedge \varphi' = \gamma(\varphi \dagger match(a, b))$$

Let the transition relation $\rightarrow_E$ be the smallest relation containing $\rightarrow$ such that for any event a and configuration c if $\nexists c' : c \xrightarrow{\text{a}} c'$ then $c \xrightarrow{\text{a}}_E c$. The relation $\rightarrow_E$ is lifted to traces. For any two configurations c and c', $c \xrightarrow{\epsilon}_E c$ holds, and $\xrightarrow{\text{a}.\tau}_E c'$ holds if there exist a configuration c' s.t. $c \xrightarrow{\text{a}}_E c'' \xrightarrow{\tau}_E c'$

Definition 4.11 A Parametric Timed Region Event Automaton $\langle Q, \Sigma, \delta, q_0, F, t, T \rangle$ where

- Q, Σ, q_0, F, and t are the same as in Definition 4.5,
- T is a set of timeout clock variables for sets of states, for each token, i.e. a set of tuples $\langle R, \mathbb{R}, B \rangle$, where B is a binding
- and δ is the union of discrete and timed transitions i.e. $\delta_t \cup \delta_d$ where
  - δ_d is defined as in Definition 4.5,

- and δ_t represents timed transitions and defined as the set of tuples $\langle R \times \mathbb{R} \times Q \rangle$

We give the following semantics to the Parametric Timed Region Automaton:

We have to define rules for the initialization of the automaton, entering states with timed outgoing transitions and we also define the general rules of changing states.

1. Initialization Rule : On initialization of the automaton, T is an empty set i.e. $T = \emptyset$
2. Entering new timed region rule : If we enter a new set of timed regions, with a token with binding b i.e. $r^+ \neq \emptyset$, we set the timers according to the timeouts, i.e. $\forall t_{timeout} : t_{timeout} := t + t_i$ where $r_t \in r^+, \exists q, \delta_t \langle r_t, t_i, q \rangle, T \langle r_t, b, t_{timeout} \rangle$
3. Firing Transitions Rule: Choose an enabled transition from the set of enabled discrete or timed transitions. We have two cases, the chosen transition is:
   a) Discrete Transition: In case of $r^- = \emptyset$ than the execution of the transition is as in described formerly. If else if we exit a region i.e. $r^- = \emptyset$, then the following rule extends the firing rule of discrete transitions: $\forall t_s, \forall q_s$ in $\delta_t \langle r_i, t_s, q_s \rangle$ where $r_i \in r^-$, the timer of the regions are invalidated i.e. $t_s := \infty$
   b) Timed Transition: The transition with the minimal timeout value is selected, i.e. transition $\exists t_i, \exists s_i, \delta_t \langle r_i, t_i, s_i \rangle$ where $r_i \in r$ and t_{min} is the minimum from all t_i, than the following rules apply: the global time is set $t := t_{min}$, the local clock is set to infinity: $t_s := \infty$ and move to the next state according to δ_t.

The automaton of expression $(a(task, {}_resource)*r(task, {}_resource)*a(task, {}_resource))|(\langle a(task, {}_resource)* \rangle_{t<10s}$ is shown on Figure 4.4

4.3 Examples of Complex Event Processing

4.3.1 File System

Problem

File system - A file shouldn't be read when it has been opened for writing, and shouldn't be written, when opened for reading. A file shouldn't be opened for writing and reading without a close event between the two different opens [19]. The possible parametrized events are : Open(file, mode), Close(file), Read(file), Write(file). Mode is either "R" or "W" which stands for Read and Write respectively.

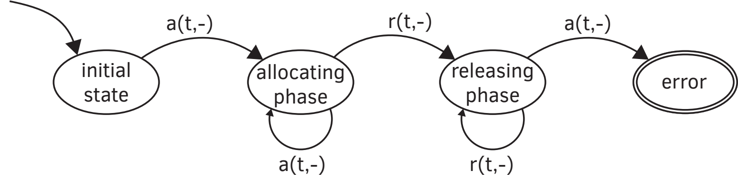


Figure 4.4 Timed region event automaton for the two phase locking example

Solution

We are looking for these patterns :

- open(f,"W") → NOT close(f){∗} → open(f,"R");
- open(f,"R") → NOT close(f){∗} → open(f,"W");
- open(f,"W") → NOT close(f){∗} → read(f);
- open(f,"R") → NOT close(f){∗} → write(f);

These event patterns can be matched with the automaton seen on Figure 4.5

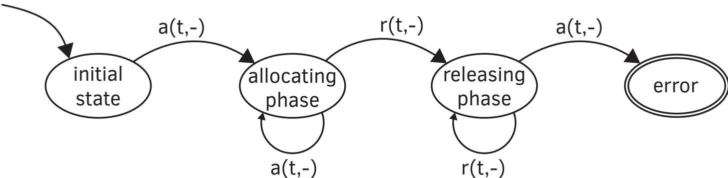


Figure 4.5 Automaton of the file example

4.3.2 Mars Rover Tasking - Two phase locking

Problem

In concurrent systems the avoidance of deadlocks and livelocks are an utmost importance. To solve this problem, one of the many patterns is the two phase locking - which can be defined by two rules. These rules are :

1. Every task must allocate the resources in a given order.
2. If a task releases a resource, it can't allocate anymore

Solution

Since our implementation doesn't support guards *yet* we can only use constant amount of resources. For this example, this amount will be set to two, to minimize the model of the example. The Item 1 pattern can be matched with the Figure 4.6, and the Item 2

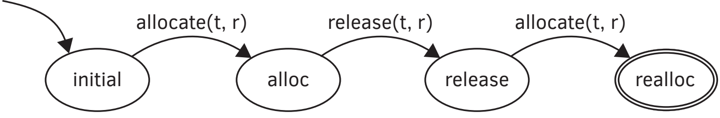


Figure 4.6 Automaton to forbid the reallocation

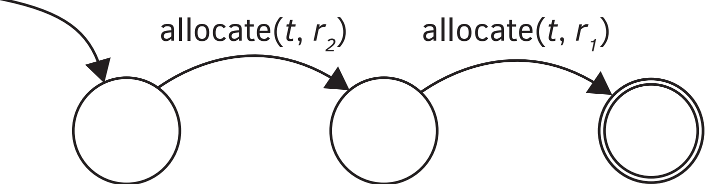


Figure 4.7 Automaton to forbid inverse allocation

4.4 Implementation

4.4.1 Metamodel

Finite Automaton

The event automaton is represented with the State, Transition, and EventGuard classes. Every State has a boolean flag

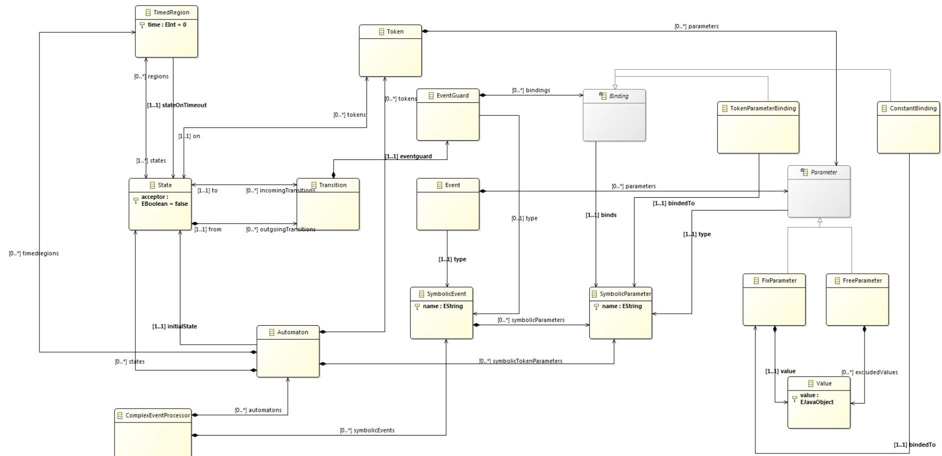


Figure 4.8 Metamodel of the Impementation in EMF

Timing

The timing is done by Timed Regions

Parameters and Bindings

4.4.2 Executor

The algorithm first searches for all the activated transitions. If it finds an activated transition, it iterates over the tokens which are on the state. The first token with matching (non-confronting) parameter list will be split to the next state if there are new parameter bindings from the event, or moved if there are no new bindings. If a token enters an acceptor state it'll next state

Chapter 5

Case study

The goal of our case study introduced in this chapter is to show the application and an example of our hierarchical runtime verification framework. The motivation of this study was the related report from 2014 [3], where the goal was a distributed, model based security logic. The work of [3] focused on the model driven development of a safety logic and its application in the Model Railway Project. Our work builds on the hardware and software presented in [3] and extends it with the runtime verification of:

- The safety logic in the embedded controllers.
- The correctness of the overall system.

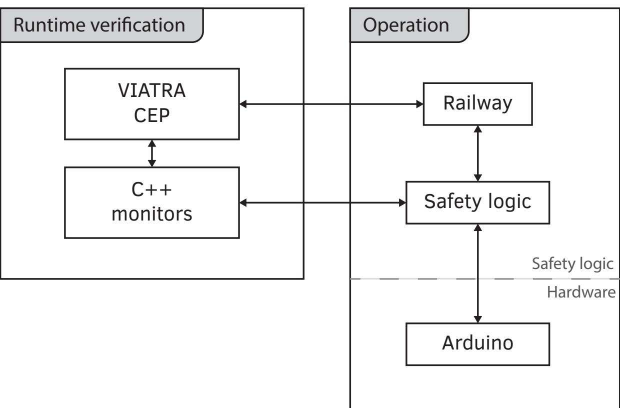


Figure 5.1 The configuration of the case study

5.1 System level observation with computer vision

5.1.1 Hardware

In case of a computer vision (CV) based approach, it is critical to choose the appropriate hardware. We had two parameters in the selection of the camera: height above the board, and FOV. The camera we used have these parameters:

- Resolution: 1920x1080
- Horitontal FOV: 120°

The camera have an installation height of 120cm. This is a perfect value for using the case study in any room, and not suffer serious perspective distortions.

5.1.2 OpenCV

One key point of this study from the technological viewpoint is computer vision. It is a new extension of the hardware, which allows us to monitor the board with fairly big precision and reliability, if the correct techniques and materials are used.

We needed a fast, reliable, efficient library to use with the camera, and develop the detection algorithm. Our choice was the OpenCV[1] library, which is an industry leading, open source computer vision library. It implements various algorithms with effective implementation e.g. using the latest streaming vector instruction sets. The main programming language – and what we used – is C++, but it has many binding to other popular languages like Java, and Python.

5.1.3 Marker design

One of the steps of the CV implementation was the design of the markers, which should provide an easy detection, and identification of the marked objects.

The first step was to consider the usage of an external library, named ArUco[2]. This library provides the generation and detection library of markers. The problem with the library was the lack of tolerance in quality, and motion blur. Because these negative properties of the existing libraries, we implemented a marker detection algorithm for our needs.

After the implementation was in our hands, we could make markers which suits our needs. The chosen size of the markers was the size of the model railroad car's width as it provides the proper accuracy.

[1] http://opencv.org/
[2] http://www.uco.es/investiga/grupos/ava/node/26

As explained in Section 5.1.4, circular patterns are well suited for CV recognition applications. The final design consists of two detection circles, with a colored circle for the purpose of identification between them (Figure 5.2).

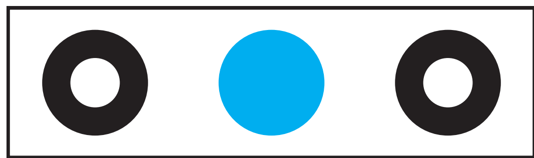


Figure 5.2 The final marker design

5.1.4 Mathematical solution for marker detection

According to the various lighting conditions and used materials, the marker detection has to be robust. Also, the fact that these markers have perspective distortion when they are near the edges of the visible region of the camera, required the application of signal processing techniques.

This method is commonly used for transforming and processing a signal – in our case a picture – in the frequency domain.

Convolution method

Our method is based on the convolution of two bitmap images, one from the camera, and one from the generated pattern.

According to the theorem, we can multiply two spectrums and apply an inverse Fourier transformation to get the convoluted image. If one image is the pattern, the other image is the raw[3]. Applying the convolution results in an image where every pixel represents the value how much the two spectrums match.

Pattern bitmap properties

The prerequisite for the pattern is that the size of the pattern and the size of the raw image must match, and the raw must be a grayscale image.

The pattern itself needs to be generated with values in accordance to the shape we would like to match (Figure 5.3). The raw pixels are multiplied by this value. The meaning of the resulting values are the following:

- $\boldsymbol{value = 0}$: Doesn't affect the match.

[3] In our application raw (or raw image) means the unprocessed image from the camera

- $\boldsymbol{value} > \mathbf{0}$: The multiplied raw pixel summed positively to the result of the convolution.
- $\boldsymbol{value} < \mathbf{0}$: The multiplied raw pixel summed negatively to the result of the convolution.

Figure 5.3 Example for pattern bitmap placement and value

5.1.5 Software

With the OpenCV library, we implemented a processing pipeline which can process the live video feed of the camera. The data is forwarded to the high level safety logic, which can decide what actions should be taken. The Table 5.1 shows all the essential steps in the processing pipeline of computer vision.

We used GPU acceleration through pipeline stage 1–4. The acceleration is implemented by OpenCV, and can be used with CUDA capable NVidia video accelerators.

5.1.6 Summary

With this implementation, we can follow the system in real-time, providing the high-level logic with an additional, independent source of information. This leads to a more robust system with added redundancy.

Table 5.1 Computer vision processing pipeline

Stage #	Description	Example images
1	Loading an image from the camera	
2	Convert the image to grayscale	
3	Convolve the image with the pattern	

Stage #	Description
4	Applying a threshold to filter the brightest spots
5	Finding the contours of the enclosed shapes
6	Calculating the center point of the contours
7	Finding possible markers by distance
8	Identifying the marker by the center

5.2 Model railroad

In the following section, a briefly overview of the structure of the railroad and the controlling hardware is given.

5.2.1 Overview

The model railroad (Figure 5.4) contains the following hardware elements:

- 15 powerable sections
- 6 railroad switchs
- 6 Arduino controllers for each switch
- 3 remotely controllable trains

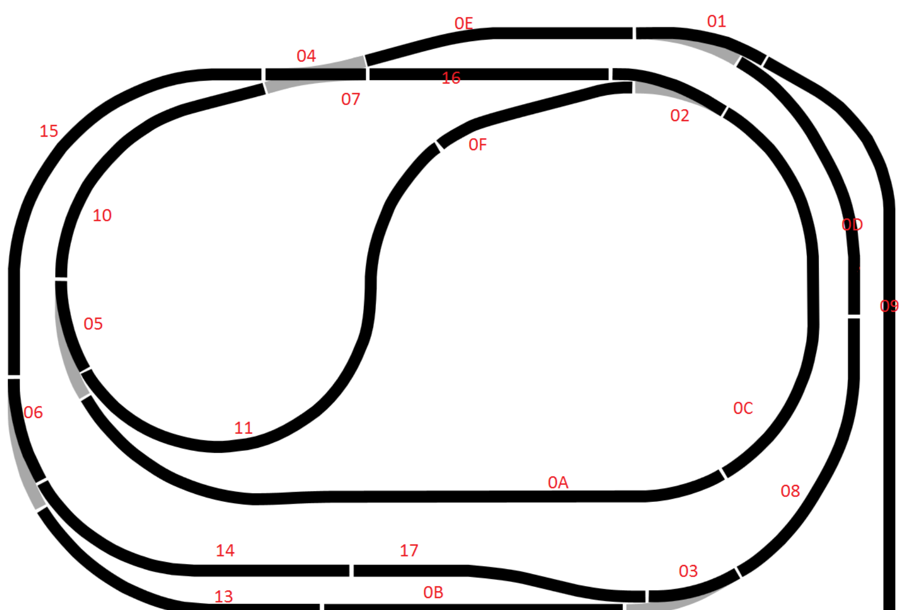


Figure 5.4 The railroad network with section IDs

5.2.2 Hardware

The core of the hardware are the Arduino microcontrollers, which collect information and control the sections. For every railroad switch there is an associated controller which can control the power of the sections nearby, using the slave units connected to it (Figure 5.5).

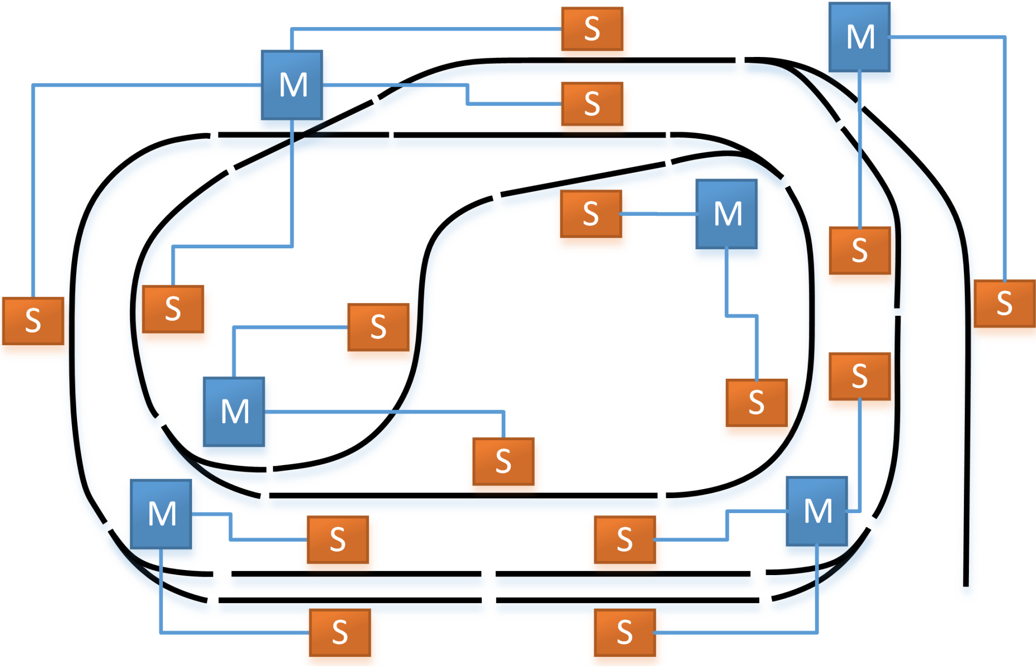


Figure 5.5 Master-slave associations

5.3 Metamodel design

In this section we develop the metamodel of the system for the physical hardware. The safety logic is based on this logical model – as a result, a model with many details of the physical world is needed.

5.3.1 Physical elements

The computer vision provides an external source of information for the system level logics. The CV forwards a train ID (determined by the marker color) and a position

(x-y coordinates) to the model, and we must discretize this information in order to make it searchable by the safety logic for hazardous events.

The main components of the physical system are:

- **Section**: Either a rail or a railroad switch. Every section has a distinctive identifier.
- **Rail**: A rail is a variable length curve. The main challenge is the determination of the next section. Only rails can be powered down, so the safety logic must act when the train is on a rail.
- **Railroad switch**: The switch is a region, where we know the entry and exit section by its setting. The switch is always powered, which means that the trains cannot be affected while on the switch. The basic concepts are:
  - A switch consists of three rails: a central, a divergent and a straight rail.
  - **Straight rail**: The straight rail follows the central rail without a curve.
  - **Divergent rail**: The divergent rail moves away from the imaginary line of the central rail.

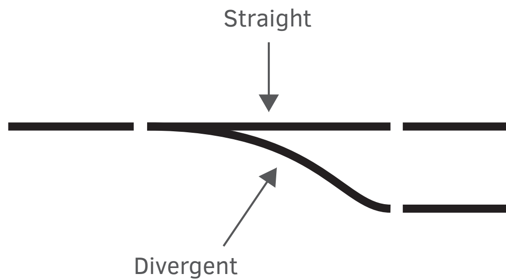


Figure 5.6 The switch's straight, and divergent rails

5.3.2 Metamodeling the physical elements

After specifying the physical elements and their properties, the logical concept of the model elements and their connections had to be uncovered.

A bottom-up structure was followed because it aids the graph search (Section 5.3.6), and helps the review of the main components of the logical elements.

1. **SectionModel**: The root of the board model. It is separated from trains as the root element is persisted, and loaded at every start of the application, while the *TrainModel* is dynamic (Figure 5.9)(Figure 5.10)(Figure 5.8).

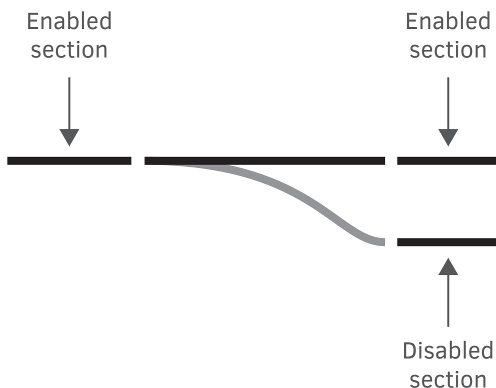


Figure 5.7 Enabled-disabled section explanation

a) **Configuration**: Contains the enabled *groups* of the switch. The *Divergent-Configuration* and *StraightConfiguration* are exactly the same as the *Configuration*. They are only presented in the metamodel because they allow the easier use of IncQuery patterns (Section 5.3.6) (Figure 5.11).

b) **SwitchSetting**: Contains a straight, and a divergent *configuration* (Figure 5.11).

c) **Region**: The atomic abstract element of the model. The *region* is the smallest unit of measurement (Figure 5.9).

d) **SectionRegion**: Specialized region, which is a part of a *powerable group*. Only powerable sections can stop a train (Figure 5.9).

e) **RailRegion**: Specialized region. Because the exact position inside a switch is unessential, the entire area of the switch is declared as one region (Figure 5.9).

f) **Group**: A group is a collection of regions (Figure 5.9).

g) **PowerableGroup**: A collection of *regions* which can be shut of. The equivalent to one rail in the modelled study (Figure 5.9).

h) **SwitchGroup**: A group of exactly one *SwichRegion*. Has a reference to a *Configuration*, describing the current switch settings (Figure 5.9).

2. **TrainModel**: The root of all train elements (Figure 5.10).

    a) **Train**: The train representation with an unique ID, the current and previous region (Item 1c), and the next group (Item 1f) determined by the current, and previous region (Figure 5.10).

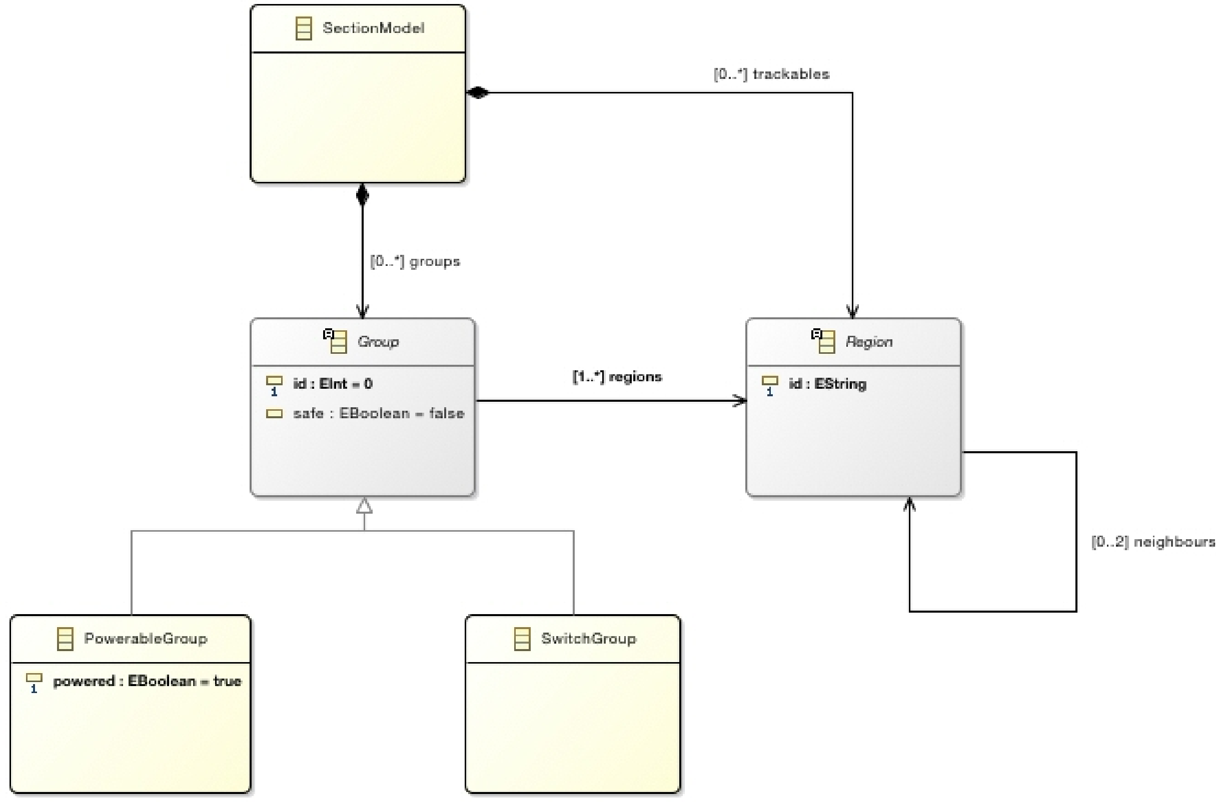


Figure 5.8 The group view of the metamodel of the *Model Railway Project* model

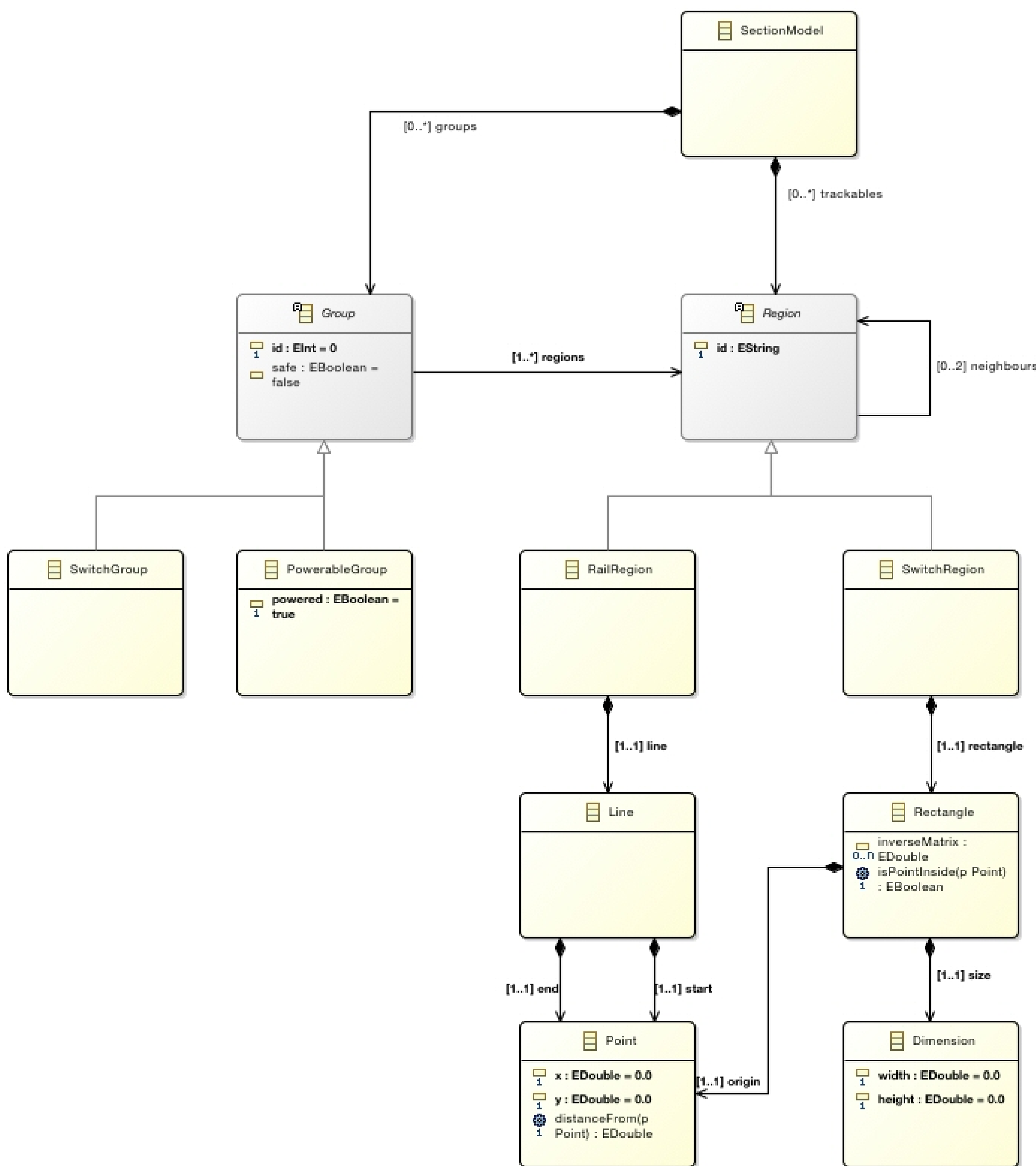


Figure 5.9 The section view of the metamodel of the *Model Railway Project* model

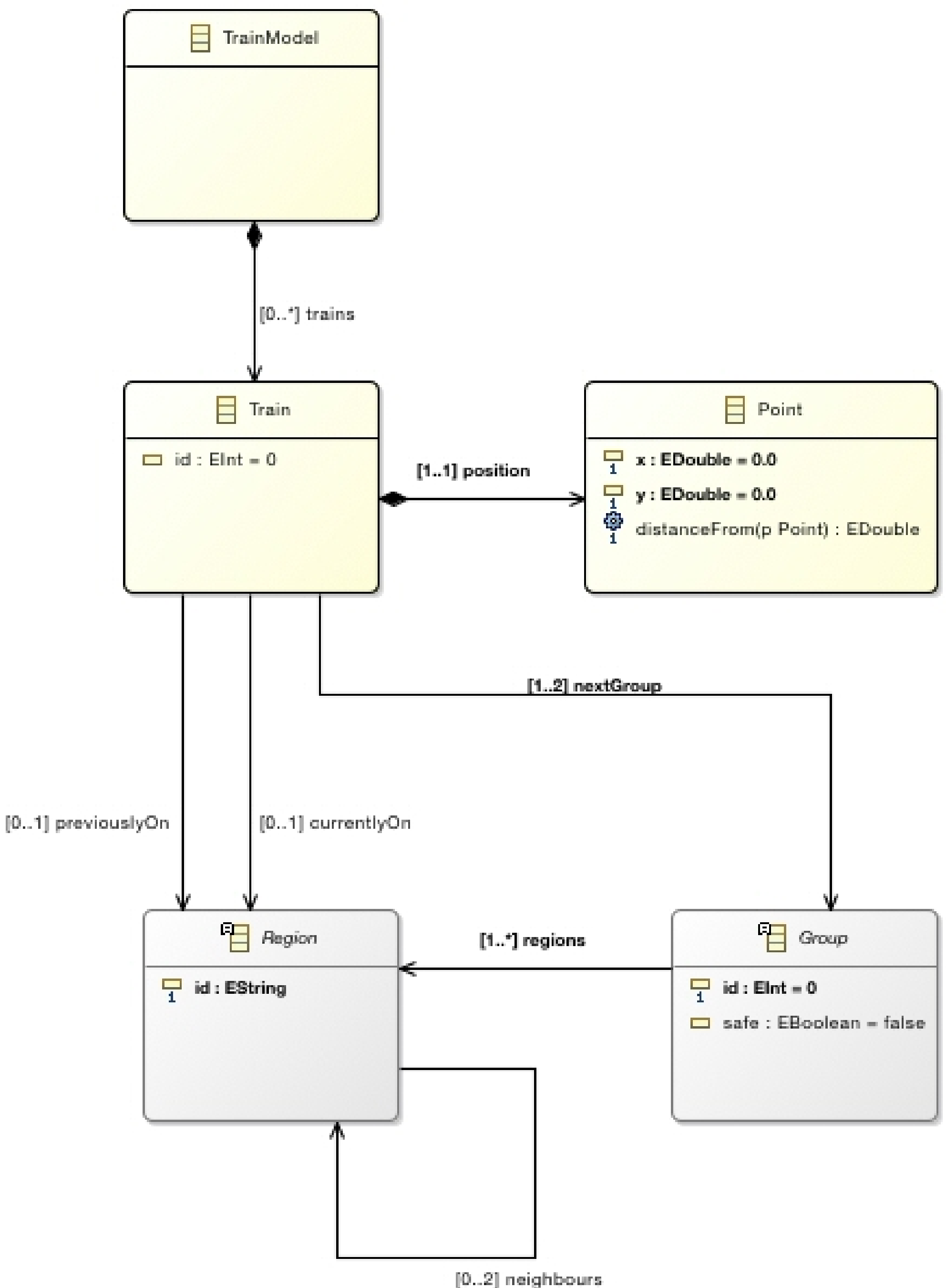


Figure 5.10 The train view of the *Model Railway Project* model

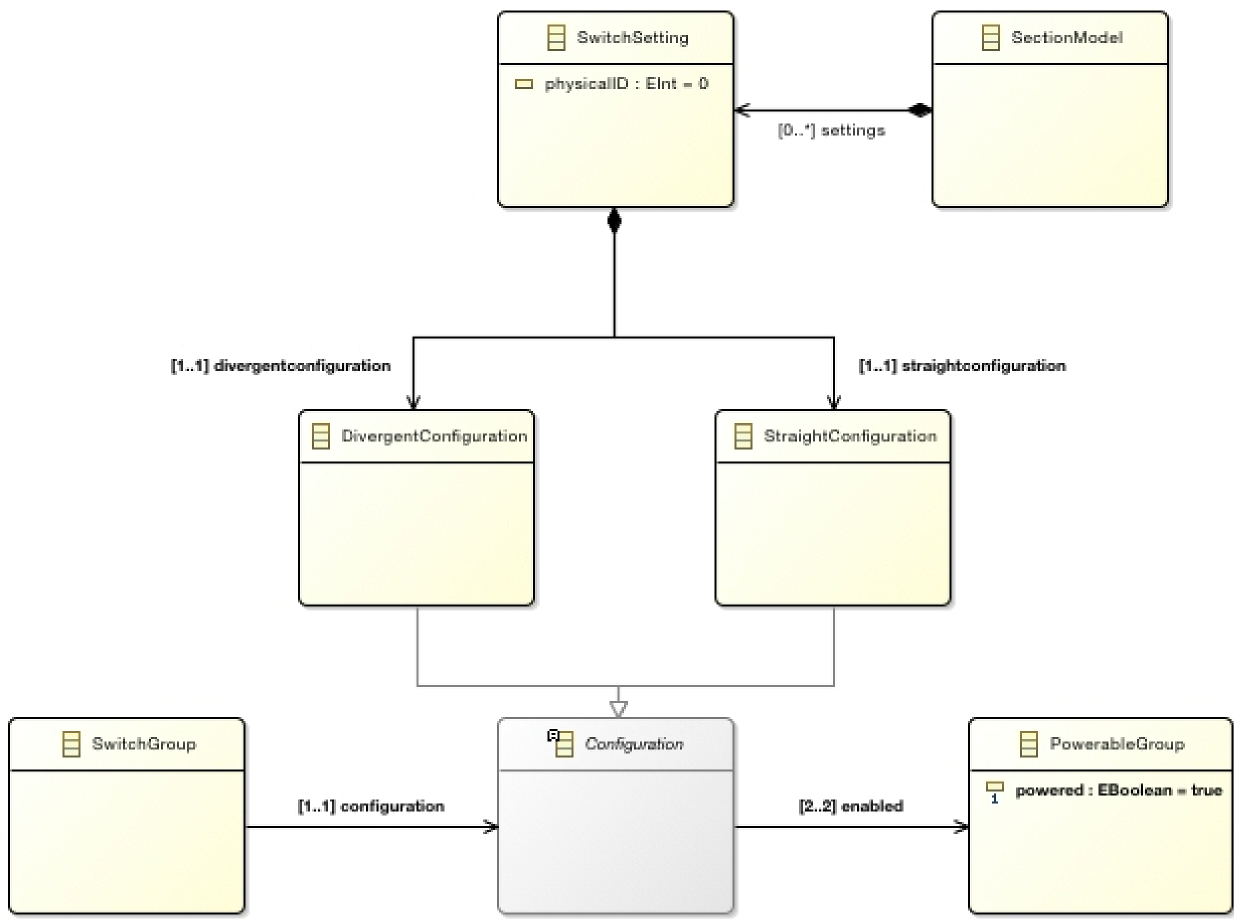


Figure 5.11 The settings view of the *Model Railway Project* model

5.3.3 Using the Eclipse Modeling Framework

We used EMF to develop the metamodel of the railway. The main reason for the usage of this modeling tool is that it allows the use of IncQuery. A few other reasons are:

- Besides the POJO[4], EMF can generate an Eclipse based editor for the model, where we can add/remove/edit all of the elements and their properties easily. The editor always checks the whether the model is well formed, and marks any problematic elements.
- The framework ensures that all of the used references are valid by updating them automatically.
- The declaration of opposing edges are also valid – these are two-way references between objects. The framework will maintain these references e.g. if we assign object A to object B, if they have a bidirectional reference between them, the EMF will automatically update the other side of the reference, in our case the reference from A to B.

5.3.4 Building the EMF model

After the conceptual design of the model, the EMF model had to be designed.

When building EMF models, it is important that every element must be a part of exactly one containment tree. If an element is not in a tree or it is in multiple tree, the serialization process fails. In Section 5.3.2 the *TrainModel* and *SectionModel* represents the root of the model.

5.3.5 Using EMF-IncQuery

The motivation for the use of IncQuery is the graph based nature of the problem. The railway can be depicted as a graph, and hazardous patterns can be described e.g. a train's next section is the next section of an opposing train. These scenarios can be described declaratively by IncQuery patterns, reducing the possibility of a coding failure.

The other advantage of using the IncQuery framework is it's scalability. The IncQuery framework – as its name suggest: incremental query – is a fast, caching query engine based on the RETE algorithm. The framework can effectively follow changes in huge environments.

5.3.6 Building the IncQuery patterns

Let us examine the patterns providing the essential filtering of hazardous patterns in the environment.

[4]Plain Old Java Object

```
1  pattern trainAtNextGroup(t1: Train) {
2    Train.nextGroup(t1, ng);
3
4    Train(t2);
5    t1 != t2;
6
7    Train.currentlyOn(t2, co);
8    Group.regions(ng, co);
9  }
```

Listing 5.1 Collision detection

Listing 5.1 shows an IncQuery example. This pattern matches the next group of `t1` – if not null, e.g. the train is stationary – which has a different train on it (`t2`).

This example clearly represents the advantages of a declarative approach. With the right metamodel an incrementally executed scalable pattern can be designed with only 5 lines of code.

```
1  pattern trainAtNextPowerable(t1: Train) {
2    Train.nextGroup(t1, ng);
3    Train.currentlyOn(t1, t1co);
4
5    SwitchGroup(ng);
6    SwitchGroup.configuration.enabled(ng, enabled);
7    enabled != t1co;
8
9    Train(t2);
10   t1 != t2;
11   Train.currentlyOn(t2, t2co);
12   Group.regions(t2g, t2co);
13
14   enabled == t2g;
15 }
```

Listing 5.2 Collision detection

Listing 5.3 is a pattern for finding the next powerable group in the direction of travel. This is a necessity as trains cannot stop on switch groups. With Listing 5.1 hazardous situations in the next group can be filtered, but if the train is on a switch, it might crash. This pattern is specialized in the case of a train going towards a switch. The other side of the switch is examined. If any train is present, the pattern will match, switching the power off the section.

```
1 pattern trainFromDisabled(t: Train) {
2   SwitchGroup(sg);
3   SwitchGroup.regions(sg, region);
4   SwitchGroup.configuration.enabled(sg, enabled);
5   region != enabled;
6
7   Train.currentlyOn(t, region);
8   Train.nextGroup(t, sg);
9 }
```

Listing 5.3 Collision detection

5.4 Component-monitor integration

The related study [3] implemented a model based security logic. With the use of the statechart language presented in Chapter 2 on page 10, we can monitor the state and execution of the model, and verify it on a higher level. In this example, we break the execution of the safety logic into two parts:

- **Monitoring the execution engine**: Execution errors (entry to error states) are detected by the monitors. The resolution of the monitoring is set to 1 second.
- **Monitoring the transitions of the model**: If the safety logic cannot access it's sensors, or communicate, the model execution will stop because of the lack of input. This behavior can be detected.

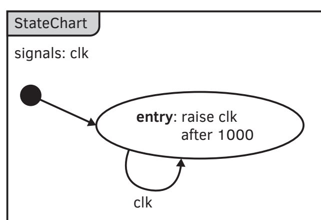
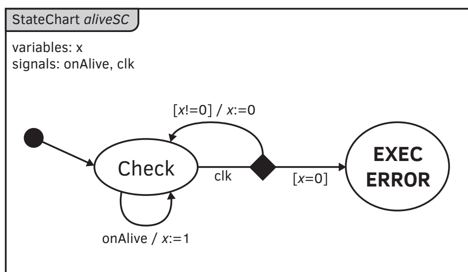


Figure 5.12 Clock generation state chart

5.4.1 Monitor statecharts



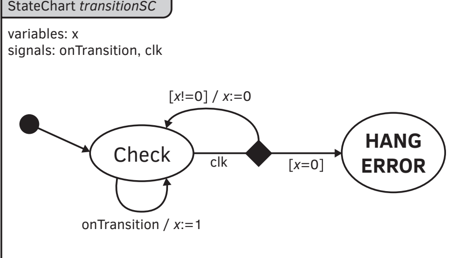


Figure 5.13 Error detection state machines

Figure 5.12, and Figure 5.13 show the concept of our monitor statechart. The statechart *clockGen* generates a periodic clock sign of 1s. In the statecharts *aliveSC* and *transitionSC* we use this clock signal as a timing event. If the expected signals are not triggered between two clock cycles, the statechart steps into an error state.

```
1  specification transitionSpec {
2      signal clk
3      statechart clockSC {
4          region clockSC {
5              initial state i
6              state clock {
7                  entry: raise clk after 1000
8              }
9              transition from clock to clock on clk
10         }
11     }
12 }
```

Listing 5.4 Statechart representation of clock generation

```
1  specification transitionSpec {
2      signal clk
3      signal onTransition
4      statechart transitionSC {
5          local x : integer
6          region transitionSC {
7              initial state i
8              state Check
9              state HangError
10             choice errorDetection
11             transition from i to Check
12             transition from Check to errorDetection on clk
13             transition from errorDetection to HangError [x=0]
14             transition from errorDetection to Check [x/=0] / assign x
                   :=0
15             transition from Check to Check on onTransition / assign x
                   :=1
16         }
17     }
18 }
```

Listing 5.5 Statechart representation of hang error detection

```
1  specification aliveSpec {
2      signal clk
3      signal onAlive
4      statechart aliveSC {
5          local x : integer
6          region aliveSC {
7              initial state i
8              state Check
9              state ExecError
10             choice errorDetection
11             transition from i to Check
12             transition from Check to errorDetection on clk
13             transition from errorDetection to ExecError [x=0]
14             transition from errorDetection to Check [x/=0] / assign x
                   :=0
15             transition from Check to Check on onAlive / assign x:=1
16         }
17     }
18 }
```

Listing 5.6 Statechart representation of hang error detection

5.4.2 Generated VEPL

The statecharts' definitions from Section 5.4.1 generates a VEPL pattern for the system monitor component, therefore we can write patterns, and define actions for pattern matches.

```
1  package hu.bme.mit.tdk2015.critcyber.vepl
2
3  AtomicEvent ExecError(id: String)
4  AtomicEvent HangError(id: String)
```

Listing 5.7 Generated VEPL definition

The generated stub (Listing 5.7) contains the error states of the statechart as AtomicEvents. With these events, a complex event processing pattern can be built.

```
1  package hu.bme.mit.tdk2015.critcyber.vepl
2
3  AtomicEvent ExecError(id: String)
4  AtomicEvent HangError(id: String)
5
6  rule ExecErrorRule on ExecError {
7    System.out.println("Error source id: " + ruleInstance.atom.
         parameterTable.parameterBindings.head.value);
8    ErrorHandler::handleExecError(ruleInstance.atom.parameterTable.
         parameterBindings.head.value)
9  }
10
11 rule HangErrorRule on HangError {
12   System.out.println("Error source id: " + ruleInstance.atom.
         parameterTable.parameterBindings.head.value);
13   ErrorHandler::handleHangError(ruleInstance.atom.parameterTable.
         parameterBindings.head.value)
14 }
```

Listing 5.8 Extended VEPL definition

With Listing 5.8, a very simple pattern is used to match the generated events, and handle them by defining rules without any conditions. Events can be parametrized, so an event itself can mark the source of the validation error. The `ErrorHandler` class is needed because the model cannot be accessed from the rule, therefore a static class is needed which can modify the state of the model. For example, if the static method `handleHangError` sets the `safe` attribute of the `Group` with the id we can fetch from the event representing the error state, the Listing 5.9 pattern can match.

```
pattern trainReachUnsafe(t: Train) {
  Train.nextGroup(t, ng);
  Group.safe(ng, safe);
  check(safe == false);
} or {
  Train.currentlyOn(t, co);
  Group.regions(g, co);

  Train.nextGroup(t, ng);
  PowerableGroup(ng);
  find regionNeighbour(ng, nng);
  nng != g;

  Group.safe(nng, nng_safe);
  nng_safe == true;
} or {
  Train.currentlyOn(t, co);
  Group.regions(g, co);

  Train.nextGroup(t, ng);
  SwitchGroup(ng);
  SwitchGroup.configuration.enabled(ng, enabled);
  enabled != g;

  Group.safe(enabled, nng_safe);
  nng_safe == true;
}
```

Listing 5.9 Collision detection

This pattern filters those trains whose second next track is unsafe. The middle rail is needed for buffering because we cannot be sure about the unsafe track state (Figure 5.15), and if another train is coming in the opposing direction, there is a chance that it cannot be stopped. As a result, the insertion of the middle track is a necessity to prevent collisions (Figure 5.14). This is an example of runtime verification: we get notified about the component failures, and the higher level logic can act before the catastrophe.

Figure 5.14 Safe configuration. The train from the failed group can be stopped on the buffer rail.

Figure 5.15 Unsafe configuration. If a train comes in the opposing direction, they will crash.

5.5 Summary

The Train Benchmark[22] shows a similar railroad application with IncQuery based pattern matching. Their benchmark showed IncQuery can match patterns on a similar railroad model with element sizes over 80 million under 100 milliseconds after the initial caching.

Chapter 6

Conclusion

Cyber physical and safety critical systems are difficult to analyse at design time, so various techniques need to be applied during the development and operation. In our work, we focus on the analysis of such systems during their operation.

In this report we introduce the concept of hierarchical runtime verification to extend the traditional model driven development methodology with a model driven runtime verification approach.

Our framework combines the advantages of various modeling languages and provides engineers with the ability to use familiar modeling formalisms to design the properties to be analysed runtime. In addition, our approach supports the hierarchical composition of the various runtime verification tasks to provide system level assurance of safety properties.

For a proof of concept implementation, we integrated statechart based component level runtime verification with a complex event processing based high level formalism. In our work we also aimed to eliminate the disadvantages of the approaches used at different levels of verification. The low level componentwise monitoring is supported with a well-established statechart formalism, which is also formalised with the help of transition systems. This formalization enables us to extend our framework and provide the formal analysis of the runtime verification properties. To take a further step in this direction, an intermediate formal language is proposed to be under the high level (system level) engineering modeling language. The intermediate language also supports the further extension of the framework with other high level modeling languages.

We illustrated the applicability of our approach with the help of a case-study. In addition, our framework is used for educational purposes to illustrate runtime verification at the courses of the department.

Acknowledgement

We would like to thank Istán Dávid for his advice, help getting familiar with the concepts of complex event processing and his support in using VIATRA-CEP. His suggestions helped us a lot to choose the research direction presented in this report.

We would also like to thank our supervisors Dr. István Ráth and Dr. Dániel Varró for their advice and help – even during weekends.

We would like to thank IncQuery Labs Ltd. and Quanopt Ltd. for their support during our summer internship.

We would also like to thank Oszkár Semeráth for the help in the modeling process, Kristóf Marussy for time he spent helping us, and Tamás Tóth for his infinite patience and persistence during our consults regarding timer-related issues.

Last but not least, we would like to thank András Vörös for the infinite hours of support, caffeine, and pizza provided during our never ending consultations.

References

[1] Eugene Asarin, Paul Caspi, and Oded Maler. "Timed regular expressions". In: *Journal of the ACM* 49.2 (2002), pp. 172–206.

[2] Howard Barringer, Ylies Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. "Quantified event automata: Towards expressive and efficient runtime monitors". In: *FM 2012: Formal Methods*. Springer, 2012, pp. 68–84.

[3] Horváth Benedek, Konnerth Raimund-Andreas, and Zsolt Mázló. *Elosztott biztonságkritikus rendszerek modellvezérelt fejlesztése*. Tech. rep. Budapest University of Technology et al., 2014.

[4] Grady Booch. *The unified modeling language user guide*. Pearson Education India, 2005.

[5] Grady Booch, Ivar Jacobson, and Jim Rumbaugh. "OMG unified modeling language specification". In: *Object Management Group* 1034 (2000), pp. 15–44.

[6] Luiz Fernando Capretz. "Y: a new component-based software life cycle model". In: *Journal of Computer Science* 1.1 (2005), pp. 76–82.

[7] István Dávid, István Ráth, and Dániel Varró. "Streaming Model Transformations By Complex Event Processing". In: *Model-Driven Engineering Languages and Systems*. Springer, 2014, pp. 68–83.

[8] Bruno Dutertre, Maria Sorea, et al. "Timed systems in SAL". In: *SRI Int., Menlo Park, CA, USA, Tech. Rep. SRI-SDL-04-03* (2004).

[9] *Eclipse Modeling Project*. URL: https://eclipse.org/modeling/emf/ (visited on 10/22/2015).

[10] *EMF-IncQuery*. URL: https://www.eclipse.org/incquery/ (visited on 10/22/2015).

[11] *Finite State Machine Designer*. URL: http://madebyevan.com/fsm/ (visited on 10/26/2015).

[12] OM Group et al. "OMG Unified Modeling Language (OMG UML), Superstructure". In: *Open Management Group* (2009).

[13] David Harel and PS Thiagarajan. "Message sequence charts". In: *UML for Real*. Springer, 2003, pp. 77–105.

[14] Øystein Haugen. "Comparing uml 2.0 interactions and msc-2000". In: *System Analysis and Modeling*. Springer, 2005, pp. 65–79.

[15] Øystein Haugen. "MSC-2000 interaction diagrams for the new millennium". In: *Computer Networks* 35.6 (2001), pp. 721–732.

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

[17] James N Martin. "Overview of the EIA 632 standard: processes for engineering a system". In: *Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*. Vol. 1. IEEE. 1998, B32–1.

[18] Leon Osborne, Jeffrey Brummond, Robert D Hart, Mohsen Zarean, and Steven M Conger. *Clarus: Concept of operations*. Tech. rep. 2005.

[19] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. "MARQ: Monitoring at Runtime with QEA". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 596–610.

[20] Miro Samek. "Who moved my state". In: *Dr. Dobb's Journal* (2003).

[21] Seema Suresh Kute and Surabhi Deependra Thorat. "A Review on Various Software Development Life Cycle (SDLC) Models". In: *IJRCCT* 3.7 (2014), pp. 776–781.

[22] *Train Benchmark Case: an EMF-IncQuery Solution*. 2015.

[23] Tamás Tóth. "Formal Analysis of Parametric Timed Systems". Master's thesis. Budapest University of Technology and Economics, 2014, p. 87.

[24] *VIATRA/CEP*. URL: https://wiki.eclipse.org/VIATRA/CEP (visited on 10/25/2015).

[25] Dolores R Wallace and Roger U Fujii. "Software verification and validation: an overview". In: *IEEE Software* 3 (1989), pp. 10–17.

[26] *Yakindu*. URL: http://statecharts.org/ (visited on 10/26/2015).