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

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 raise 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. [13]

1.2 Safety Critical System

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 for application in areas such as medicine, flight control, weapons industries, and nuclear systems. Many modem 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 fast paced development cycles, which will require significant advances in areas like specification, architecture, and verification, to meet the safety requirements. [12]

1.3 Verification

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 [21], 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, instead their mutual use can support a more robust verification process.

1.3.1 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. The processes check the product's compliance 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, formal methods are usually used to verify the critical components of systems (if formal methods are applied at all).

1.3.2 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 by the specification. This has an advantage of detecting the erroneous operation of the system, letting engineers prepare safety functions for handling faults, or trigger an emergency shutdown if necessary.

1.4 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. The dynamic 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 an always up-to-date description of the 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 verification possible.

Formal verification is especially important for the development of safety critical systems (e.g.: space and flight technologies), making MDSD notably widespread in these 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 a few modeling formalism of their target domain.

MDSD is usually accompanied by the Y model – a software life cycle model for component based systems [5], based on the V model.

1.4.1 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.

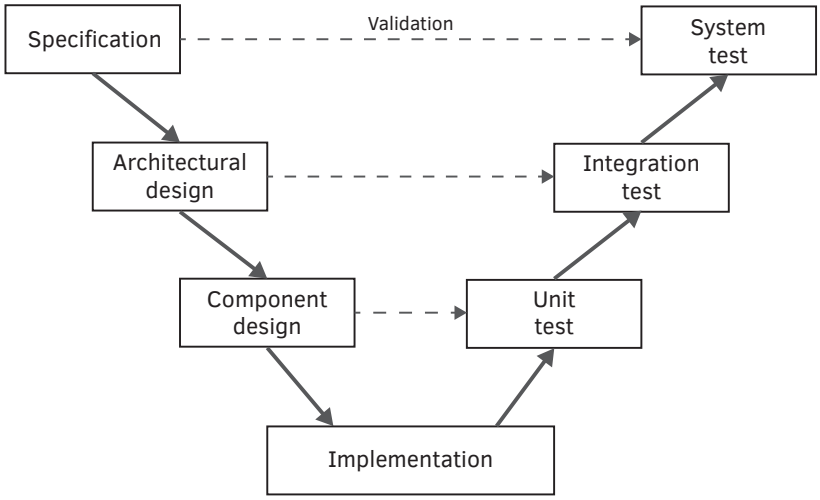


Figure 1.1 The traditional V model [15]

Figure 1.3 shows the stages of development, with a symbolic “V” showing the progression from the top of the left leg of the “V” down to the base, across the base, and up the right leg. The project definition stages down the left side begin with 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”, with an arrow labeled “Time” pointing right to left across the bottom of the “V”. The right leg shows the testing and implementation stages of a system, with an upward-pointing arrow showing the progression from the base up the leg. [15]

The V model is the basic scheme of a software development. In the left leg, we proceed down by decomposing the specifications into architecture level design, and the architecture design into component level designs. Every level of decomposition have it’s own specifications. After the implementation process, we verify every levels of decompositions behavior with it’s specification.

1.4.2 Y model

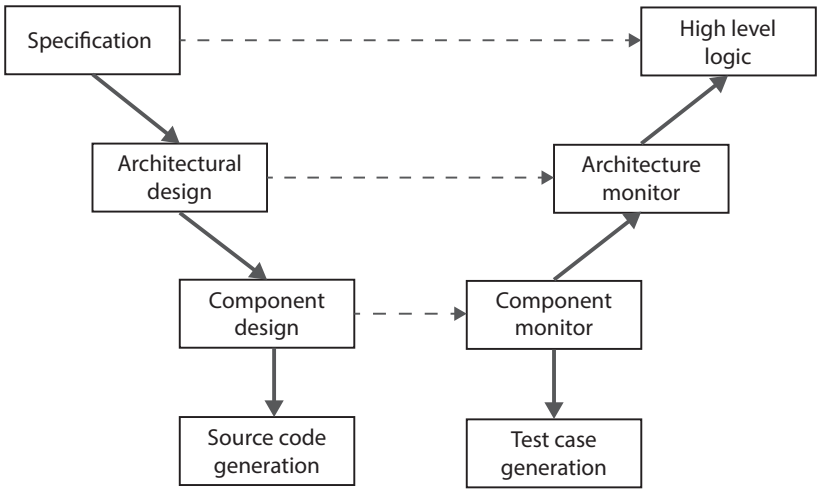


Figure 1.2 The Y model

The Y model [5] is an extension of the V model (which is in turn an extension of the waterfall model) [19], 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 last step, 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.

1.4.3 Modeling approaches on different 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 [4]) and SysML (Systems Modeling Language [3]) languages.

Finite automaton

Modelling a system 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 [14]).

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 [8]). 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 [17].
Statecharts are usually created by tools that support the graphical design of the model (e.g.: Yakindu, an Eclipse based editor).

Sequence Chart

The formalism of message sequence charts (MSC) describes the communication between components – the order in which messages can occur [11] [9]. 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 [10].

CEP

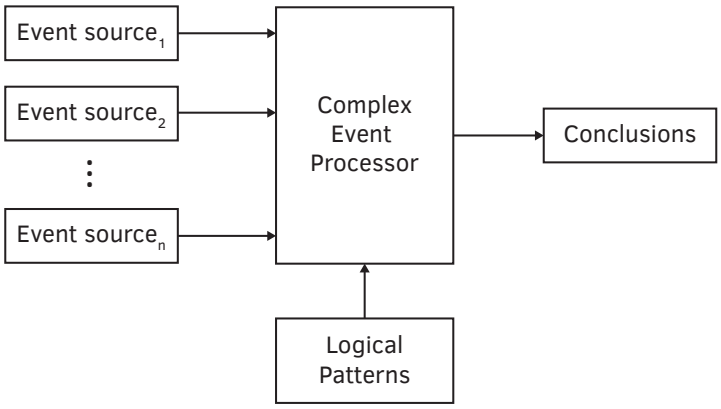


Figure 1.3 CEP overview

Complex event processing is a method of tracking and analysis 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: [**epbas**]

1.5 A hierarchical approach to runtime verification

A hierarchical runtime verification is based on the communication of runtime monitors with a higher level logic instead of a one time verification in design time. By using the V model, the right legs verification steps can be replaces with its runtime verification counterpart (Figure 1.4). These replaced elements are:

- Component monitors which are the smallest, can be implemented into embedded devices.
- Architecture monitors which are verifying the interoperation of components.
- High level logic is a central element for reacting to the states of the monitors.

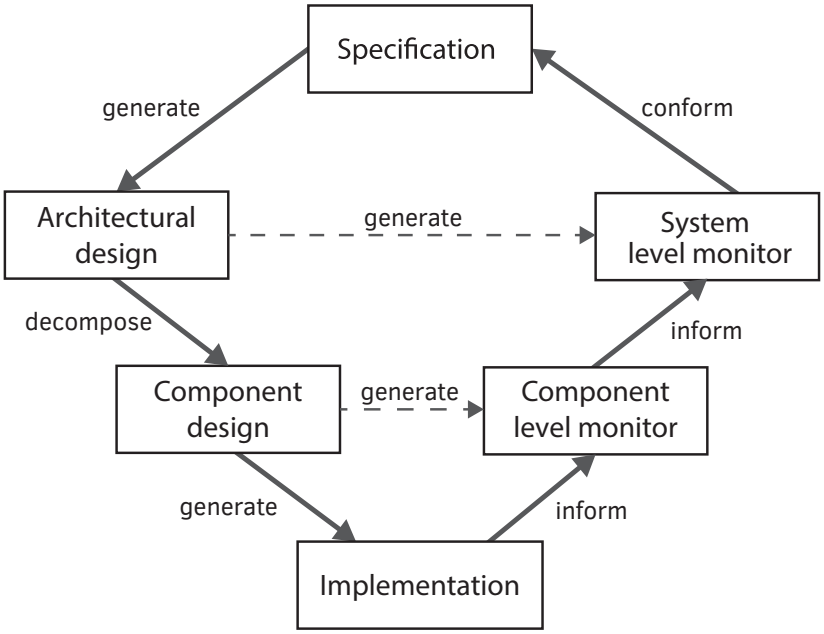


Figure 1.4 Hierarchical verification based on the V model

Our goal was to provide runtime verification methods for all levels of a complex cyber-physical system. These methods should rely on modelling methods already being used be engineers to minimize the steepness of the learning curve and allow quick and efficient development. The result is a hierarchical runtime verification framework which can enable verification on two levels (Figure 1.5) The framework is capable of:

- Component level verification as monitors generated from statechart formalism
- System level verification, that can be done by a complex event processing system
- Monitors can generate messages indicating erroneous operation of the monitored components
- The complex event processing system can use the monitors messages

Additionally, the statechart language developed extends the usual statechart formalism by adding support for statechart templates. This enables users to easily describe systems with homogeneous components. The models can also be mapped to a formally verifiable transition system.

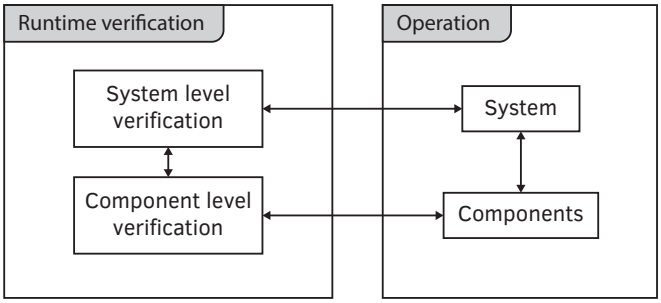


Figure 1.5 Associations between operation, and runtime verification components

With component level verification – in out case with embedded monitors – we can verify each component runtime state. The system level verification observes the overall state of the system. If any of the components reports failure, we can make a decision based on the remaining components abilities to support the system verification. If we can provide safety despite component failures, the system can continue operation.

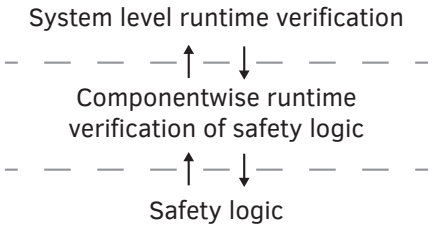


Figure 1.6 Overview of the hierarchical runtime verification system

TODO

Chapter 2

Background

The design of complex cyber-physical systems is an interdisciplinary process. From the 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 [13] [18].

2.1 Model driven software development

Model driven software development (MDSD) emphasises 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. Modelling has the advantage of introducing abstractions, thus reducing the complexity of the development process. The dynamic 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 an always up-to-date description of the 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 verification possible.
Formal verification is especially important for the development of safety critical systems (e.g.: space and flight technologies), making MDSD notably widespread in these areas.

2.2 Development process

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 a few modelling formalism of their target domain.
MDSD is usually accompanied by the Y model – a software life cycle model for component based systems [5].

2.2.1 The Y model

The Y model is an extension of the V model (which is in turn an extension of the waterfall model) [19], 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 last step, 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.

2.2.2 Modelling methods

MDSD methods require modelling languages to describe the behaviour 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 modelling 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 modelling languages were developed like the UML (Unified Modelling Language [4]) and SysML (Systems Modelling Language [3]) languages.

UML

UML defines an abstract syntax (Meta Object Facility, MOF), while the concrete syntaxes of different types of models are represented by separate UML diagrams.
However, the formal semantics of the language – responsible for the behaviour of the components – were not defined clearly. This led to inconsistent usage and multiple interpretations of expected behaviour. To solve this issue, fUML was developed (Semantics of a Foundational Subset for Executable UML Models [6]), which provides formal semantics for a large subset of the UML language.

Finite automata

Modelling a system 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.

Definition 2.1 A finite automaton is a $\langle Q, \Sigma, \delta_d, q_0, F \rangle$ tuple where:

- Q is a finite, non empty set. These are the states of the automaton,
- Σ is a finite, non empty set. This is the set of input symbols of the automaton,
- δ_d is a set of $\langle Q \times \Sigma \times Q \rangle$ tuples
- $q_0 \in Q$ a start state,
- $F \subseteq Q$ the set of the acceptor states.

Although finite automata can be easily visualized, this formalism describes a simple, flat transition system and lacks the support for higher level concepts.

Statecharts

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 [8]). 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 [17].
Statecharts are usually created by tools that support the graphical design of the model (e.g.: Yakindu, an Eclipse based editor).

Sequence charts

The formalism of message sequence charts (MSC) describes the communication between components – the order in which messages can occur [11] [9]. 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 [10].

Complex event processing

2.2.3 Verification

Formal verification

Runtime verification

Chapter 3

Overview of the approach

The case study concentrates on the connection between the design time and runtime domain, with model based runtime verification using monitors. Our goal is to present the concept of a tool which facilitates the runtime verification design of systems.

3.1 Concept of runtime verification with engineering model

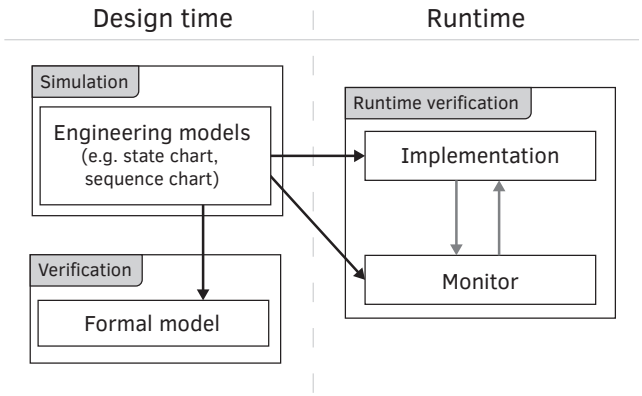


Figure 3.1 Connection between runtime and design time elements

Figure 3.1 depicts the basic concept. We have three main transformations:

- **Engineering model ↦ Formal model**: The specified engineering models can be mapped to a formally verifiable transition system.
- **Engineering model ↦ Implementation**: We can generate components from the engineering models. The generated components then can push state notifications towards the monitor.

- **Engineering model ↦ Monitor**: The generation of monitoring components is also available (Chapter 4 on page 17) using the engineering models. The monitor is a state machine that is operated by the generated code. If the execution reaches an invalid state, the monitor detects it. After detection the monitor can forward the error detection event to a higher level processor.

Some transformation tools exist for these transformations, but there are no tools which integrate all of the above in one framework, allowing multiple formalisms to be used, especially with this level of automation.

Let us consider the following example:

We want to design a system, where we have a logic, described by a formalism (e.g. state chart, sequence chart), and we want to:

- Generate the implementation from engineering models.
- Implement runtime verification into our implementation, by monitoring it.
- Support multiple monitoring systems which can communicate.

This example covers the usual need of a distributed, embedded safety logic. We need to formally verify the model, generate code, and monitor it. Our goal is to integrate all these solutions into a generic tool. With a centralized toolchain, a developer can make a system with less effort but with more robustness thanks to the verifiable, and automated steps.

3.2 Hierarchical runtime verification approach

We can distinguish three levels, representing the hierarchy of our approach:

- **Local verification with state chart**: With the generated monitor, we could handle the error states locally, e.g. if we detect an error in the generated code, the monitor can react and shut down the system as a precaution.
- **Communication verification with state/sequence chart**: The monitor can verify the communication with the higher level logic, reacting to e.g. loss of communication.
- **Integrating multiple system level monitor**: TODO

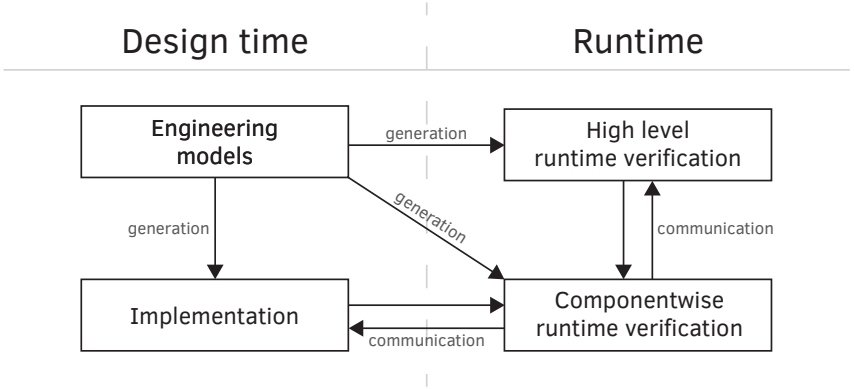


Figure 3.2 Conceptual draw of a hierarchical system element relations with our method

If we specify a high level specification to our overall system, which can have many monitored, communicating components, we can build a high level logic on top of these components offering another level of monitoring, and robustness.

As Figure 3.2 shows, a hierarchical solution can be made by just defining the correct engineering model, and the high level logic can automatically intercept the error state messages.

The high level logic can range from simple reaction logic to a complex system, like the complex event processing solution (Chapter 5 on page 31).

Chapter 4

Runtime verification of embedded systems

Many software is available for code generation and verification based on statecharts. Unfortunately the available solutions for verification and modelling provide limited syntax, and code generation usually results in poor quality code. 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. For monitor generation, our approach was to generate easily readable, extendible, object oriented code, that can run in environments with limited resources – primarily embedded systems.

4.0.1 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 short 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.

4.0.2 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.

4.1 Overview

For a brief overview of the language, see the example statechart and it's textual specification below (Figure 4.1).

Figure 4.1 A simple statechart

```
specification exSpec {
  global gVar : integer
  signal exSig
  statechart exStc {
    local lVar : integer
    region exReg {
      initial state exInit
      state exComplex {
        region exInner {
          initial state exInit2
          state exInner
        transition from exInit2 to exInner on exSig / assign lVar := lVar + 1
        }
        region exInner2 {
          initial state exIdle
        }
      }
      state [Error] exSimple
```

```
     transition from exInit to exComplex / assign gVar := 3, assign lVar := 0
      transition from exComplex to exSimple [gVar = 3] / raise exSig(2)
    }
  }
}
```

The description of the whole system is enclosed in a specification, which can have multiple statechart, signals, and global variable declarations. Each statechart consists 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 can have 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.

4.2 Language syntax

4.2.1 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.

4.2.2 Statecharts

The syntax for statechart definition is in the form of:

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

where *params* is an optional parameter list, and `{...}` contains the description of the statechart itself. The braces 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. A specification must contain at least one instantiated statechart.

4.2.3 Regions

Statecharts are structured by the use of regions. At the root of every statechart, a region encloses all of the states. 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 regions is:

```
region NAME {...}
```

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.

4.2.4 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.

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, or if the region does not have containing state, 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.

Initial states

Initial states can be found in all regions - if the region's containing state is entered, these inner states become active.

Choice states

Choice states are pseudo states whose outgoing transitions have exactly one guard condition 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.

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.

4.2.5 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.

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.

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.

Transition actions

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

4.2.6 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 simultaneously. It also leaves room for a later expansion to a token based automata with re-entry. A parametric signal can be declared like:

`signal NAME(`*param*`)`

where *param* is 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. A timeout can be raised by:

`raise NAME after` *amount*

where `NAME` is the name of the referenced signal and *amount* is the number of timesteps before the signal is raised.

Variable assignments

Variable assignments can be expressed with the syntax:

`assign` *lhsExp* `:=` *rhsExp*

where *lhsExp* has to evaluate to an assignable reference, such as a variable or an array element. The *rhsExp* can be any valid expression.

4.2.7 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.

4.2.8 Signalling errors, error propagation

States and transitions can be labelled as errors. The syntax is:

```
state/transition [Error label] ...
```

where the `label` is a description given by the user. This is only used for the generation of error messages in the monitor.

4.2.9 Expressions

Variables can be used in expressions. Expressions can have an arbitrarily complex structure within the limits of [TODO-ref]. This allows, just to mention a few, the use of array indexing, parenthesis, and common operators in programming languages such as `+`, `-`, `*`, `/`, etc. 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 [TODO-ref].

4.2.10 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 [TODO-ref]. Variable declarations are in the form of:

```
global|local var NAME : TYPE
```

where global or local denotes the scope of the variable, and `TYPE` is one of the supported types.

4.3 Formal representation

To enable formal verification of the defined systems, a mapping tool was developed that can transform specifications to a transition system. This means that the complex concepts of the model have to be flattened out. The transition system used has the following 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, that may use =, <, >, and, or, etc...
- transition systems have transitions that represent changes of state using the referenced variables previous and current values

4.3.1 Specification

The specification of the statechart language is mapped to the specification of the transition system. Global variables cannot be declared in the specification itself, they must be present in all of the transition systems that use them. Signals will be mapped to global variables.

4.3.2 Statecharts

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

4.3.3 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.

4.3.4 States

States are represented as boolean variables that signal whether the 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.

4.3.5 Transitions

Transitions between states are transitions in the formal model too. For a transition to be enabled, guard conditions has to evaluate to TRUE for the current state of the system, the source state has to be currently 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, the exit actions, the actions associated with the transition, and the needed entry actions of the appropriate states occur.

4.3.6 Signals

Signals can be used for communication between statecharts. As such, each transition system that references a signal need 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 was taken. As each statechart referencing the 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. The parameters of the signals are stored and can be referenced as global integer variables.

4.3.7 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 (TODO-ref.: Transition Systems in SAL - Dutertre, Sorea). 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. 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.

4.3.8 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.

4.4 Accepting monitor

Runtime verification of safety critical components can be done by a monitoring component that checks the current state. This 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. To make runtime verification possible, a lightweight `C++` monitoring component can be generated from the described statecharts. This monitor can then 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 behaviour. Should a state or a transition that has been marked as an error be reached, the monitor's error signalling interface is called. The user can implement this interface to hook the generated error messages to other monitors and components running the safety, or error handling logic of the system. This function makes hierarchic runtime verification possible.

4.4.1 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.

4.4.2 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.

4.4.3 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 behaviour of transitions, and are handled when mapping transitions.

4.4.4 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.

4.4.5 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.

4.4.6 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.

4.4.7 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.

4.5 Implementation

The implementation of the monitoring component can raise many questions about timing. To generate working, and easily usable code additional helper classes and extra functions for the already described ones are required. The most important ones are described in this section. It should also be noted that the generated `C++` code heavily relies on the new features of `C++11`.

4.5.1 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.

4.5.2 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`.

4.5.3 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.

4.5.4 Error signalling

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 5

Complex event processing

5.1 Complex Event Processing with the current technologies

In a hierarchical runtime verification, the top level of modelling can be done by Complex Event Processing. In our project we are going to do so, and define Event Patterns for the top 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, or VEPL for short. This language is perfect for our approach, it has a very simple syntax, and the event patterns can be defined in a very high level way, but the automaton which it translates the patterns to are not well-formalized, and the execution of the patterns are dependent on so called "event contexts" which makes the verification extremely hard.

Our goal is to make an intermediate language, to which we can translate our patterns to. This language has to be formalized.

TODO abra : Harom doboz, High level -> Intermediate language (Regular Expressions) -> Executable code (Automatons) (?)

The operators of the language is show on Table 5.1. The syntax sugars are shown on Table 5.2, but we are going to ignore these, since they can be translated to the basic operators as show on Table 5.3.

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 three rules :

Table 5.1 Basic operators

Operator name	Denotation	Meaning
followed by	p1 → p2	Both patterns have to appear in the specified order.
or	p1 OR p2	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 5.2 Syntax sugars

Operator name	Denotation	Meaning
and	p1 AND p2	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 5.3 Syntax sugars mapped to basic operators

Operator name	Denotation	Equivalent
and	p1 AND p2	((p1 → p2) OR (p2 → p1)).
negation	NOT p	$\Sigma \setminus p$, where Σ is the set of all the possible Events.
multiplicity	p{n}	p1 → p1 → . . . p1, n times.
“at least once" multiplicity	p{+}	p → p{∗}

5.2 Regular Expression

In this particular example, we need a language to describe event sequences. To do so, one of the most common formalism is a Regular Expression.

Definition 5.1 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 $\varphi_1 \cdot \varphi_2, \varphi_1 \vee \varphi_2$, and φ^* are Σ-expressions.
3. If φ, φ_1 and φ_2 are Σ-expressions then $\varphi_1 \circ \varphi_2, \varphi^{\circledast}$ are Σ-expressions.
4. If φ_1 and φ_2 are Σ-expressions, φ_0 is a Σ_0-expression for some alphabet Σ_0, and $\Theta : \Sigma_0 \to \Sigma$ is a renaming, then $\varphi_1 \wedge \varphi_2$ and $\Theta(\varphi_0)$ are Σ-expressions.

Using Regular Expressions we can describe an event sequence as ab, if $a, b \in \Sigma$ and the meaining of this is that "an a event will happend, and a b event will follow it. In the two phase locking example we are going to describe the behaviour of one resource with a simple VEPL pattern: $(a \to f)$ which means that an infinite sequences of allocation and release are allowed after eachother. This is equivalent to the regular expression : $(af)*$.

5.2.1 Deterministic Finite Automaton

To match simple sequences as $A \to B$ or $A\{*\}$ we can use a Deterministic Finite Event Automaton

Definition 5.2 A EventAutomaton (Deterministic Finite Event Automaton in other words) $\langle Q, \Sigma, \delta_d, q_0, F \rangle$ tuple where:

- Q is a finite, non empty set. These are the states of the automaton,
- Σ is a finite, non empty set. This is the Event set of the automaton,
- δ_d is a set of $\langle Q \times \Sigma \times Q \rangle$ tuples, 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 $q_1 \in Q$
- $q_0 \in Q$ a start state,
- $F \subseteq Q$ the set of the acceptor states.

The semantic is the following : On input e, where $e \in \Sigma$, if the token is on State s the next state will be s' where $\delta_d \langle s, e, s' \rangle$

The regular expression of the automaton can be compiled to this automaton[1]:

5.3 Timed Regular Expression

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

Definition 5.3 Timed Regular Expressions over an alphabet Σ (also referred to as Σ-expressions) are defined using the following families of 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 and I is an integer-bounded interval then $\langle \varphi_I \rangle, \varphi_1 \cdot \varphi_2, \varphi_1 \vee \varphi_2$, and φ^* are Σ-expressions.
3. If φ, φ_1 and φ_2 are Σ-expressions then $\varphi_1 \circ \varphi_2, \varphi^{\circledast}$ are Σ-expressions.
4. If φ_1 and φ_2 are Σ-expressions, φ_0 is a Σ_0-expression for some alphabet Σ_0, and $\Theta : \Sigma_0 \to \Sigma$ is a renaming, then $\varphi_1 \wedge \varphi_2$ and $\Theta(\varphi_0)$ are Σ-expressions.

5.3.1 Timed Event Automaton

Definition 5.4 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 5.2,
- t is a global clock variable $t \in \mathbb{R}$,
- T is a set of local timeout clocks, i.e. a set of tuples $\langle Q, \mathbb{R} \rangle$
- 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 5.2,
  - 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:

[1] TODO ide kell egy abra a resource automatajarol

$Q_t \subseteq Q$ is the set of states with outgoing timed transitions, i.e. $\forall s \in Q_t$: there exist $\delta_t\langle s, t, s'\rangle$ 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. $\forall t_i \exists q_i T\langle q_i, t_i\rangle, t_i := \infty$
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: Deterministically 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 $s \notin Q_t$ than the execution of the transition is as in 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: $t_s := \infty$
   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 the next state according to δ_t.

5.3.2 Timed Region Automaton

Definition 5.5 A 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 **??**,
- T is a set of timeout clocks for sets of states, i.e. a set of tuples $\langle 2^Q, \mathbb{R}\rangle$
- 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 5.2,
  - and δ_t represents timed transitions and defined as the set of tuples $\langle 2^Q \times \mathbb{R} \times Q\rangle$

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. $\forall q \in Q_t$: there exist $\delta_t\langle s, t, s'\rangle$ where $t \in \mathbb{R}$ and $s' \in Q$.

s is the state we are currently on, and s' is the state we move to after the transition

r is the set of zones we are currently in, i.e. $r \subseteq 2^Q$ where $\exists r_s : r_s \in r, s \in r_s$
r' is the set of zones we enter, i.e. $r \subseteq 2^Q$ where $\exists r_s : r_s \in r, s' \in r_s$
$r^+ \subseteq 2^Q$ is a set of new timed regions we just entered, i.e. $r^+ = r' \setminus r$
$r^- \subseteq 2^Q$ is a set of timed zones 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 : On initialization of the automaton, we have to set all clocks to ∞ i.e. $\forall t_i, \exists q_i, T\langle q_i, t_i\rangle, t_i := \infty$

2. Entering new timed region rule : If we enter a new timed region, i.e. $r^+ \neq \emptyset$, we set the timers according to the timeouts, i.e. $\forall t_{timeout} : t_{timeout} := t$ where $r_t \in r^+, \exists q, \delta_t\langle r_t, t, q\rangle$

3. Firing Transitions Rule: Deterministically 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 $z^- = \emptyset$ than the execution of the transition is as in described formerly. If else if we exit a zone i.e. $z^- = \emptyset$, then the following rule extends the firing rule of discrete transitions: $\forall t_s, \exists q_s$ in $\langle r_i, t_s, q_s\rangle$ where $r_i \in r^-$, $t_s := \infty$

   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.

5.4 Parametric Timed Regular Expression

5.4.1 Parametric Timed Region Automaton

PRE REFACTOR. PRE REFACTOR. PRE REFACTOR. PRE REFACTOR. PRE REFACTOR. PRE REFACTOR. PRE REFACTOR. PRE REFACTOR. PRE REFACTOR. PRE REFACTOR. PRE REFACTOR. PRE REFACTOR. After this comment, the whole document is obsolete, must rewrite it.

5.4.2 Extending the Event Automaton Formalism to handle parameters

We use s to denote a tuple $\langle s_0, \ldots, s_k\rangle$. We use $X \to Y$ 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 5.6 (Smybols, 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

Definition 5.7 (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 5.8 (Matching). Given a ground event a an event b the predicate matches (a, b) hold if there exists a binding θ s.t. b(θ) = a. Moreover let match(a,b) denote the smallest such binding w.r.t $\sqsubseteq$ if it exists (and is undefined otherwise)

Definition 5.9 (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 5.10 (Parametric Timed Region Automaton). PTRA$\langle Q, \Sigma, \delta, f_0, F, t, T \rangle$

- where $\delta = \delta_t \cup \delta_d$
- and $Q, \Sigma, \delta_d, f_0, F$ and t are defined as in Definition 5.4

- $\delta_t\langle R, Q, \mathbb{R}\rangle$ where $R \subseteq 2^Q$
- $T\langle R, B, \mathbb{R}\rangle$ where B is a set of binding and $R \subseteq 2^Q$

The semantic of the Parametric Timed Region Automaton is defined as follows: $Q_t \subseteq Q$ is the set of states with outgoing timed transitions, i.e. $\forall s \in Q_t$: there exist $\delta_t\langle r, t, s'\rangle$ where $t \in \mathbb{R}$ and $s' \in Q$, and $r \subseteq 2^Q$ and $s \in r$

s is the state we are currently in, s' is the state we move the token to.

r^+ is the new regions we enter, i.e. $r_i \setminus r_j$ where $\exists r_i \exists k_i \exists q_i \delta_t\langle r_i, k_i, q_i\rangle$ and $r_i \setminus r_j$ where $\exists r_j \exists k_j \exists q_j \delta_t\langle r_j, k_j, q_j\rangle$ and $s \in r_i$ and $s' \in r_j$.

r^- is the old regions we leave, i.e. $r_j \setminus r_i$. 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. $\forall t_i in \delta_t\langle q_i, t_i\rangle where \exists q_i$ and $t_i := \infty$
2. Entering Timed Region Rule: On entry to state s' we calculate r^+, and start every timer. i.e. $\forall t_i$ in $\delta_t\langle r, s', t_i\rangle$

   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: Deterministically 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 $s \notin Q_t$ than the execution of the transition is as in 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: $t_s := \infty$

   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 the next state according to δ_t.

5.4.3 Compilation of the Event patterns to Parametric Timed Region Automaton

The high level languages we use, such as the VEPL or UML Sequence charts can be transformed to an Intermediate language based on Parametric Timed Regular Expressions. These Parametric Timed Regular Expressions will be matched with the later defined Parametric Timed Region Event Automatons, which can be executed.

VEPL

TODO : Either define recursive rules for VEPL-Automaton transformations or give multiple complex examples

Sequence chart

TODO : Either define recursive rules of transformations or give multiple complex examples

5.5 Examples of Event Processing

5.5.1 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 [16]. 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.

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 5.1

5.5.2 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

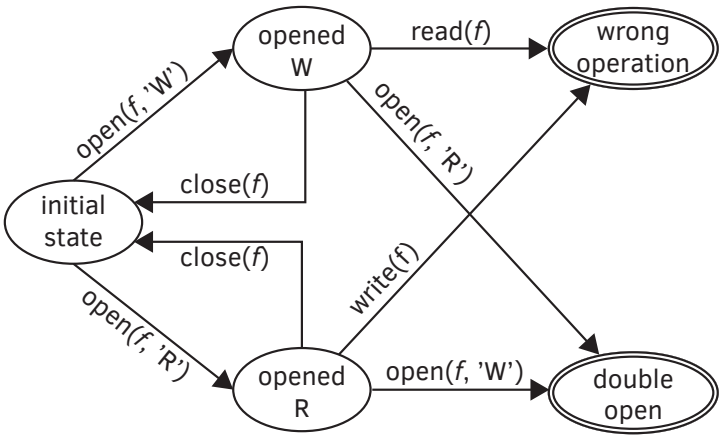


Figure 5.1 Automaton of the file example

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 5.2, and the Item 2

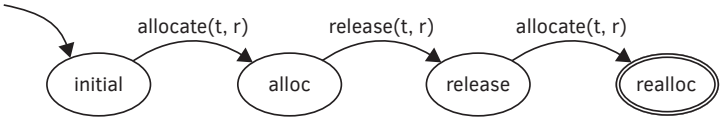


Figure 5.2 Automaton to forbid the reallocation

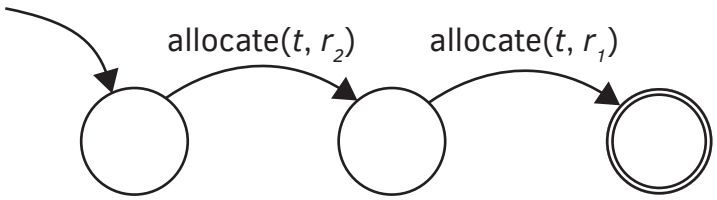


Figure 5.3 Automaton to forbid inverse allocation

5.6 Implementation

5.6.1 Metamodel

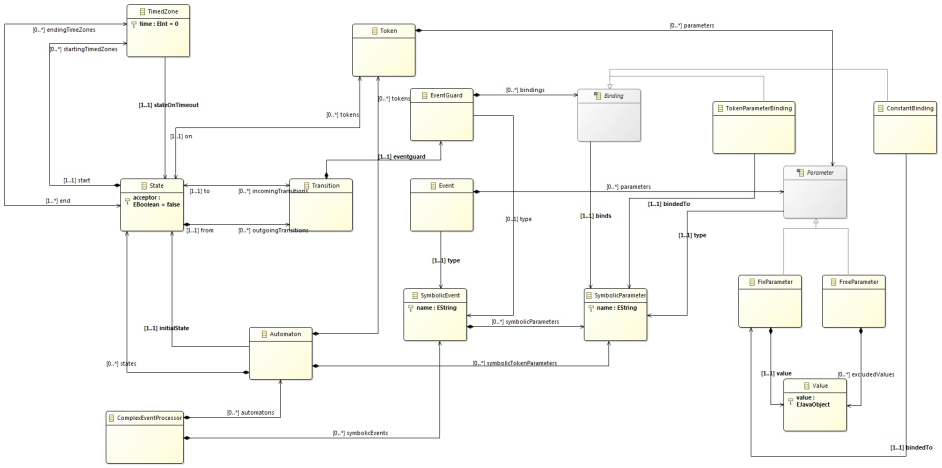


Figure 5.4 Automaton of the file example

Basic Automaton

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

Timing

Parameter

Binding

5.6.2 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 6

Case study

6.1 Overview

The goal of our case study introduced in this chapter is to show the application and working of our hierarchical runtime verification framework. The motivation of this study is the related report from 2014 [2], where the goal was a distributed, model based security logic. The work of [1] 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 of [1] and extends it with the runtime verification of:

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

6.2 System level observation with computer vision

6.2.1 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.

6.2.2 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.

6.2.3 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 marker was the size of the model railroad car as it will provide the proper accuracy.

As explained in Section 6.2.4, circular patterns are well suited for these applications. The final design consists two detection circle, and a color circle for identification between the detection circles (Figure 6.1).

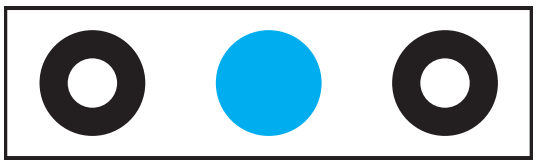


Figure 6.1 The final marker design

6.2.4 Mathematical solution for marker detection

According to the various condition in lighting, and used materials, the marker detection has to be robust.

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

This problem, and the fact that these markers have perspective distortion when they are near to the visible region of the camera motived us to develop a processing technique coming from signal processing.

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

Convolution method

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

The theorem says, we can multiply two spectrums, and apply an inverse Fourier transform 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 a value how much the two spectrums match.

Pattern bitmap properties

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

The pattern itself needs to be generated with values according to the shape we would like to match (Figure 6.2). The raw pixels are multiplied by this value. The meaning of these values in the bitmap are the following:

- $\boldsymbol{value = 0}$: Doesn't affect the match.
- $\boldsymbol{value > 0}$: The multiplied raw pixel summed positively to the result of the convolution.
- $\boldsymbol{value < 0}$: The multiplied raw pixel summed negatively to the result of the convolution.

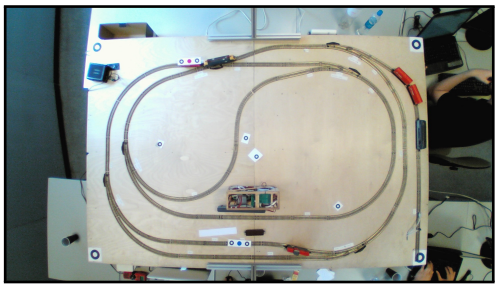
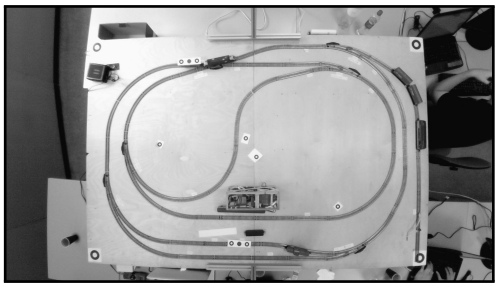
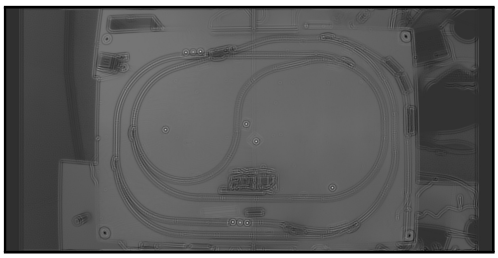
6.2.5 Software

With the OpenCV library, we implemented a processing pipeline which can process the live video feed from the camera. We forward this data to the high level safety logic, which can decide the following actions. The Table 6.1 shows all the essential steps in the processing pipeline of the computer vision.

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

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

Table 6.1 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	Find possible markers by distance
8	ID the marker by the center

Figure 6.2 Pattern bitmap placement and value example

6.2.6 Summary

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

6.3 Model railroad

In this section we briefly overview the structure of the railroad and the controlling hardware.

6.3.1 Overview

The model railroad (Figure 6.3) contains the following hardware elements:

- 15 powerable section
- 6 railroad switch
- 6 Arduino controllers for each switch
- 3 remotely controllable train

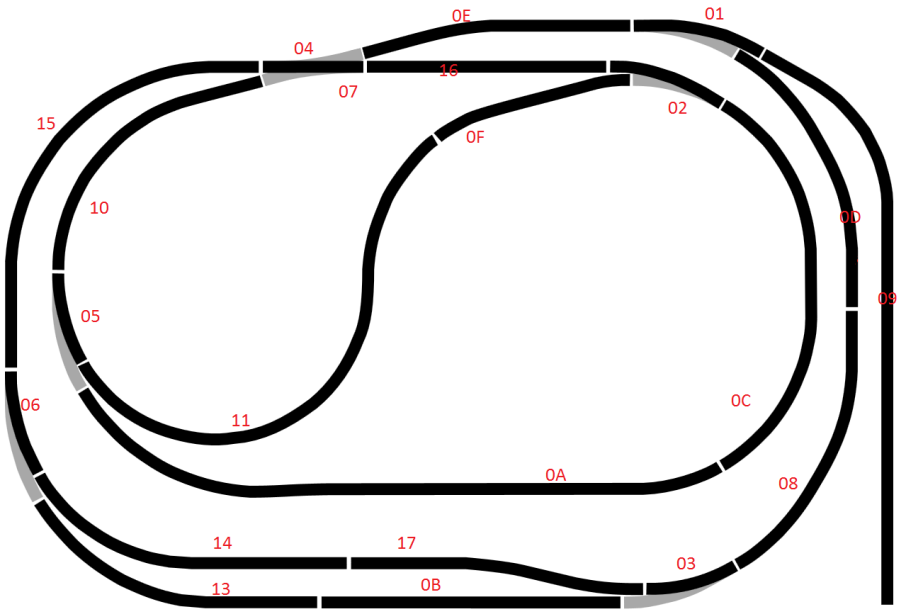


Figure 6.3 The railroad network with section IDs

6.3.2 Hardware

The core of the railroad hardware are the Arduino microcontrollers which collects information, and control the sections. For every railroad switch there is an associated controller which can control the power of the sections nearby with the slave units connected to it (Figure 6.4).

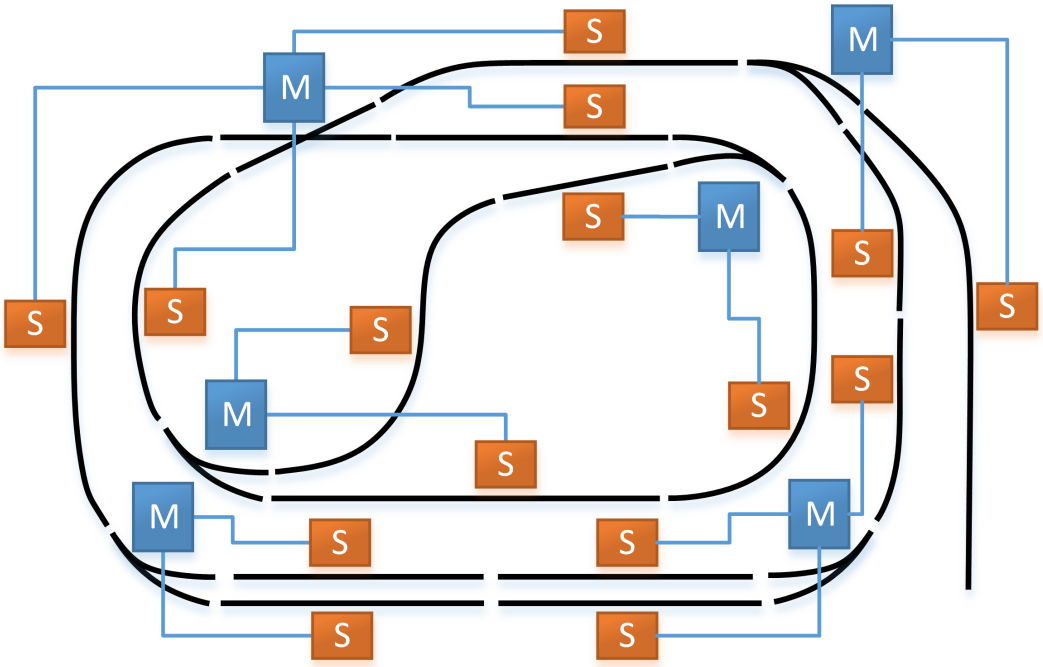


Figure 6.4 Master-slave associations

6.4 Metamodel design

In this section we proceed through the design of the physical to logical mapping. We operate our safety on this logical model, so it is very important to map all the details of the physical world we need correctly in this model.

6.4.1 Physical elements

The only external source of information is the computer vision. The CV forward a train ID (determined by marker color) and position (x, y coordinates) to the model, and

we must discretize these informations to make it searchable by our safety logic for hazardous events.

Let us take a look on the main components of the physical system, and what challenges we face:

- **Section**: Either a rail, or railroad switch. Every section has a distinctive identifier.
- **Rail**: The rail is a variable length curve. The main challenge is the determination of the next section. Only the rail can be powered down, so our safety logic must act, when the train is on a rail.
- **Railroad switch**: The switch is a region, where we know the entry and outgoing section by its setting. The switch is always powered, so we cannot affect the train on the switch. There are some basic concept:
  - A switch consists of three rails: the central rail, and two rails we can choose of, 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.

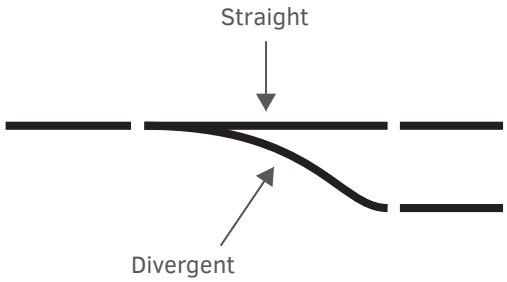


Figure 6.5 Switch straight, and divergent rails

6.4.2 Metamodeling the physical elements

After we designated the physical elements, and their properties, we started to build a logical concept what are our model elements, and what are the connections between them.

We will follow a bottom-up structure because it helps the graph search (Section 6.4.6), and review the main components of the logical elements.

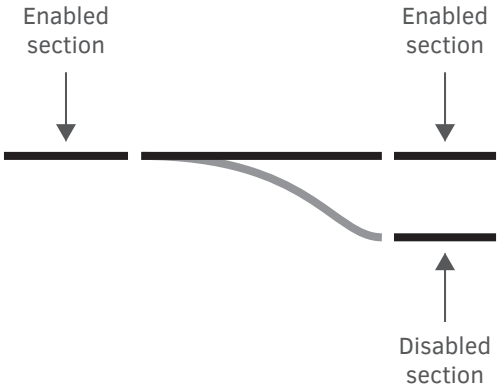


Figure 6.6 Enabled-disabled section explanation

1. **SectionModel**: The root of the board model. It is separated from trains because this root element is persisted, and loaded at every start of the application, while the *TrainModel* is dynamic (Figure 6.7)(Figure 6.9)(Figure 6.8).

    a) **Configuration**: Contains the enabled *groups* of the switch. The *Divergent-Configuration* and *StraightConfiguration* are the exactly same as the *Configuration*, they only presented in the metamodel because the ease of use with the IncQuery patterns (Section 6.4.6) (Figure 6.10).

    b) **SwitchSetting**: Contains a straight, and divergent *configuration* (Figure 6.10).

    c) **Region**: The atomic abstract element of our model, the *region* is the smallest unit of measurement (Figure 6.7).

    d) **SectionRegion**: Specialized region, which is a part of a *powerable group*. Only powerable section can stop a train (Figure 6.7).

    e) **RailRegion**: Specialized region. Because we did not intrested in the position inside the switch, we declare the entire area of the switch as one region (Figure 6.7).

    f) **Group**: The group is a collection of regions (Figure 6.7).

    g) **PowerableGroup**: A collection of *regions* which can shutted of. The equivalent to one rail of the modelled study (Figure 6.7).

    h) **SwitchGroup**: A group of exacty one *SwichRegion*. Have a reference to a *Configuration*, describing the current switch settings (Figure 6.7).

2. **TrainModel**: The root of all train elements (Figure 6.9).

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 6.9).

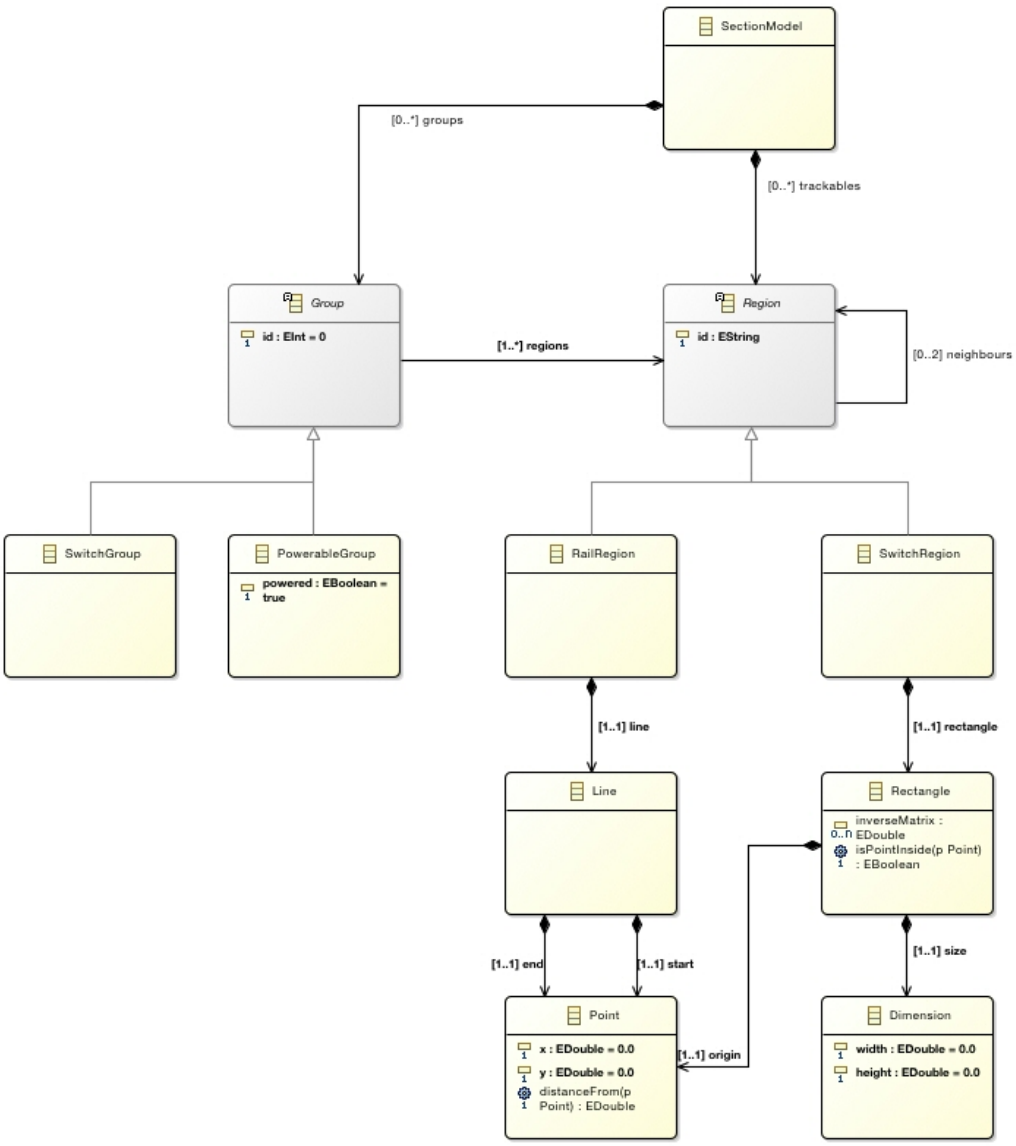


Figure 6.7 The section view of the metamodel of the *Model Railway Project* model

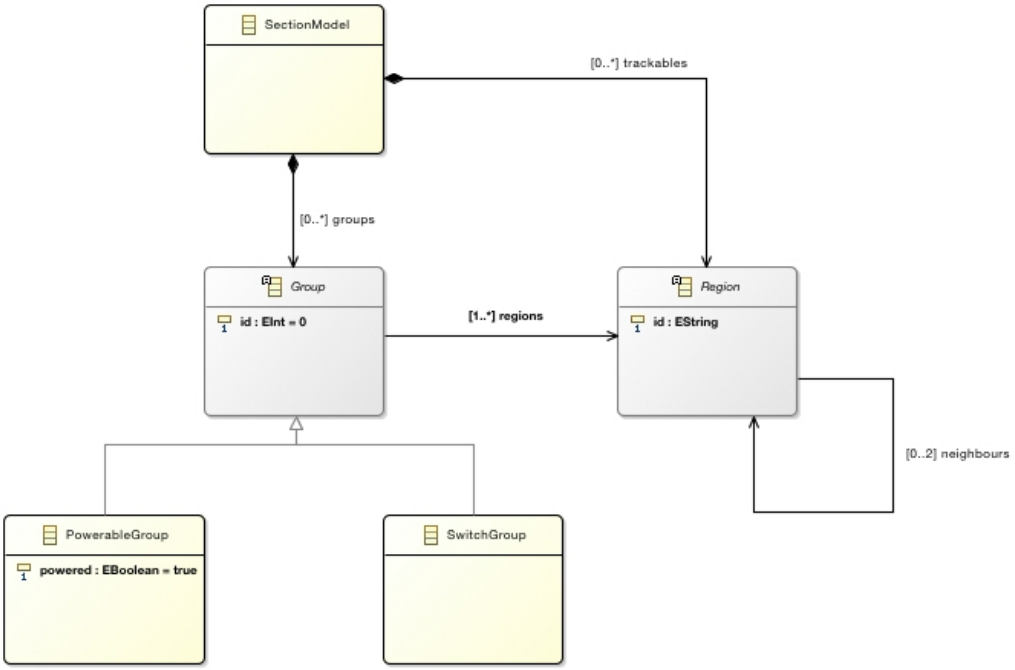


Figure 6.8 The group view of the metamodel of the *Model Railway Project* model

6.4.3 Introducing to 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." [7]

We used the EMF tool to create the metamodel of the railway. The main reason for this modeling tool is the dependency to IncQuery, but other reasons motivated the use of it:

- Besides the POJO[4], EMF can generate an Eclipse based editor for the model, where we can add/remove/edit all the elements, and their properties easily. The editor always checks the model well formed property, and marking the problematic elements.

[4]Plain Old Java Object

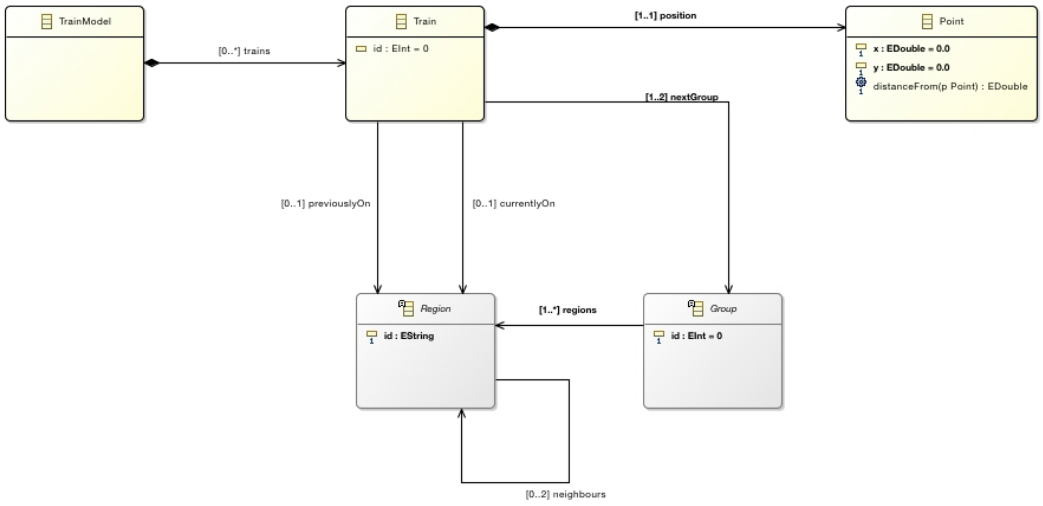


Figure 6.9 The train view of the of the *Model Railway Project* model

- The framework ensure all the references are valid, by updating them automatically.
- We can make opposite edges, which are forward/backward references across two object. The framework will maintain these references e.g. 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.

6.4.4 Building the EMF model

After the conceptual design of the model, we started the design of the EMF model.

It's important while building the EMF model that every element must be a part of exactly one containament tree. If an element is not in a tree or it is in multiple tree, it causes failure while serializing. In Section 6.4.2 the *TrainModel* and *SectionModel* represents the root of the model.

6.4.5 Introducing the 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.

With EMF-IncQuery, you can:

Figure 6.10 The settings view of the of the *Model Railway Project* model

- Define model queries using a high level yet powerful query language (supported by state-of-the-art Xtext-based development tools)
- Execute the queries efficiently and incrementally, with proven scalability for complex queries over large instance models
- Integrate queries into your applications using essential feature APIs including IncQuery Viewers, Databinding, Validation and Query-based derived features with notifications.

The motivation of using IncQuery can be found in the nature of our problem. The railway can be depicted as a graph, and we can describe hazardous patterns e.g. two trains next section is the opposite trains next section. These scenarios can be declaratively described by IncQuery patterns, reducing the possibility of a coding failure.

The other advantage of using the IncQuery framework is scalability. The IncQuery framework – as its name suggest: incremental query – is a fast, caching engine based on the RETE algorithm. This framework can follow changes in a very large environment.

6.4.6 Building the IncQuery patterns

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

```
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 6.1 Collision detection

Listing 6.1 shows an IncQuery example. This pattern matches `t1` which next group – if not null, e.g. the train is stationary – has a different train on it (`t2`).

This example clearly presents the advantage of this declarative expression. With the right metamodel we designed an incrementially executed scalable pattern only with 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 6.2 Collision detection

Listing 6.9 is a pattern for finding the next powerable group in the direction of. We must do that because the property of the switch group: a train cannot stop on that. With Listing 6.1 we can filter the hazards in the next group, but if we are on a switch, the train cannot be stopped; the trains might crash. This pattern is specialized in case of a train going towards a switch. We are virtually going through it, and examine the other side. 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 6.3 Collision detection

Listing 6.9 is a pattern for finding the next powerable group in the direction of. We must do that because the property of the switch group: a train cannot stop on that. With Listing 6.1 we can filter the hazards in the next group, but if we are on a switch, the train cannot be stopped; the trains might crash. This pattern is specialized in case of a train going towards a switch. We are virtually going through it, and examine the other side. If any train is present, the pattern will match, switching the power off the section.

6.5 Component-monitor integration

The related study [2] implemented a model based security logic. With the use of the state chart language presented in Chapter 4 on page 17, we can monitor the state of the model and execution, and verify it on a higher level. In our example, we break the execution of the safety logic in two part:

- **Monitoring the execution engine**: If an error occurs while executing the generated code, the monitor will notice it. 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 the lack of input. We are capable of the detection of this behavior.

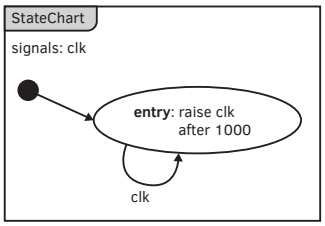


Figure 6.11 Clock generation state chart

6.5.1 Monitor statecharts

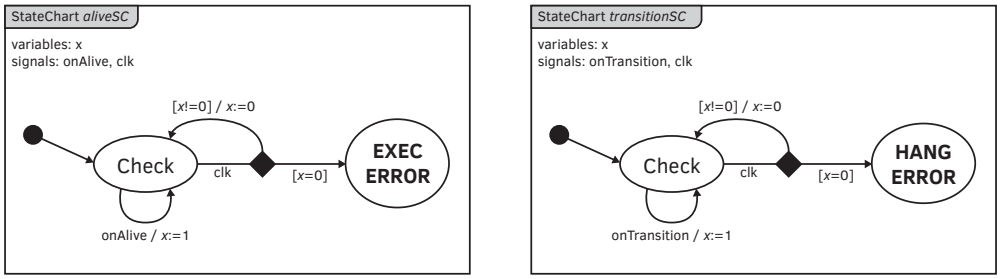


Figure 6.12 Error detection state machines

Figure 6.11, and Figure 6.12 shows the concept of our monitor statechart. The statechart *clockGen* generate a periodic clock sign of 1 s. In the statecharts *aliveSC* and *transitionSC* we use this clock signal as a timing event. If the expected signals not triggered between two clock cycles, we enter the 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 6.4 Statechart representation of clock generation

```
specification transitionSpec {
    signal clk
    signal onTransition
    statechart transitionSC {
        local x : integer
        region transitionSC {
            initial state i
            state Check
            state HangError
            choice errorDetection
            transition from i to Check
            transition from Check to errorDetection on clk
            transition from errorDetection to HangError [x=0]
            transition from errorDetection to Check [x/=0] / assign x
                :=0
            transition from Check to Check on onTransition / assign x
                :=1
        }
    }
}
```

Listing 6.5 Statechart representation of hang error detection

```
specification aliveSpec {
    signal clk
    signal onAlive
    statechart aliveSC {
        local x : integer
        region aliveSC {
            initial state i
            state Check
            state ExecError
            choice errorDetection
            transition from i to Check
            transition from Check to errorDetection on clk
            transition from errorDetection to ExecError [x=0]
            transition from errorDetection to Check [x/=0] / assign x
                :=0
            transition from Check to Check on onAlive / assign x:=1
        }
    }
}
```

Listing 6.6 Statechart representation of hang error detection

6.5.2 Generated VEPL

The state charts definitions from Section 6.5.1 generates a VEPL pattern on the system monitor component, therefore we can write patterns, and and define actions for the patterns conclusions.

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

Listing 6.7 Generated VEPL definition

We can add rules to this stub, defining patterns for the error states in Listing 6.7.

```
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 6.8 Extended VEPL definition

With Listing 6.8, we use a very simple pattern to just match the generated events, and handle them. The `ErrorHandler` class is needed because from the rule we cannot access the model, therefore we need a static class which can modify the state of the model. For example, if this method sets the `safe` attribute of the `Section` with the id we informed, the following pattern can match:

```
1  pattern unsafe(t: Train) {
2    Train.nextGroup(t, ng);
3    Group.safe(ng, safe);
4    check(safe == false);
5  } or {
6    Train.currentlyOn(t, co);
7    Group.regions(g, co);
8
9    Train.nextGroup(t, ng);
10   PowerableGroup(ng);
11   find regionNeighbour(ng, nng);
12   nng != g;
13
14   Group.safe(nng, nng_safe);
15   nng_safe == true;
16 } or {
17   Train.currentlyOn(t, co);
18   Group.regions(g, co);
19
20   Train.nextGroup(t, ng);
21   SwitchGroup(ng);
22   SwitchGroup.configuration.enabled(ng, enabled);
23   enabled != g;
24
25   Group.safe(enabled, nng_safe);
26   nng_safe == true;
27 }
```

Listing 6.9 Collision detection

This pattern filters the trains, which second next track is unsafe. This middle buffer rail needed because we cannot be sure about the unsafe track state (Figure 6.14), and if an other train coming toward us, there is a chance we cannot stop it. Because this chance, we need to insert a buffer middle track. This track can stop a train if one comes toward the other (Figure 6.13). This is one example of runtime verification: we get notified about the component failures, and the higher level logic can act before the catastrophe.

Figure 6.13 Safe configuration. On the buffer state, the train from the failed group can be stopped

Figure 6.14 Unsafe configuration. If a train coming towards us, we will crash

6.6 Summary

The Train Benchmark[20] shows a similar railroad approach application with IncQuery based pattern matching. Their benchmark showed IncQuery can match pattern on a similar railroad model with element sizes over 80 million under 100 milliseconds after initial caching.

Chapter 7

Conclusion

Chapter

Chapter 8

Acknowledge

Itt köszönjük meg!

References

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

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

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

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

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

[6] Michelle L Crane and Juergen Dingel. “Towards a formal account of a foundational subset for executable UML models”. In: *Model Driven Engineering Languages and Systems*. Springer, 2008, pp. 675–689.

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

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

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

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

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

[12] 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.

[13] 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.

[14] Andreas Muelder. “Yakindu”. In: *Jun-2011.[Online]. Available: http://www. yakindu. org/yakindu/.[Accessed: 20-Aug-2009]* (2011).

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

[16] 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.

[17] Miro Samek. “Who moved my state”. In: *Dr. Dobb’s Journal* (2003).

[18] Kenneth J Schlager. “Systenas Engineering-Key to Modern Development”. In: *IRE Transactions on Engineering Management* (1956).

[19] 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.

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

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