MŰEGYETEM 1782

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 Dée
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

Chapter

Chapter 2

Background

Chapter

Chapter 3

Overview

Chapter

Chapter 4

Runtime verification of embedded systems

4.1 Intro

A statechart language was created to enable the high level design, verification, and monitoring of complex systems. The aim was to use a simple and straightforward syntax to keep the language's learning curve gentle. Statecharts were chosen as they are used widely for modeling in various branches of engineering.

4.2 Goal

4.2.1 Simple, generic, useable

4.2.2 Verification

4.2.3 Monitor generation

4.3 Why not upgrade previous solutions

Validation software is... Many software is available for code generation. Unfortunately the available solutions either provide poor quality code or have a limited syntax, thus making the creation and understanding of the models more time consuming than necessairy. Our approach was to generate easily readable, extendable, object oriented code that can run in a limited resource environment.

4.3.1 Parametric statechart declaration

The language allows a specification to consist of multiple statecharts. This feature led to of the main strengths of the language: the definition of statechart templates, which can be parametrically instantiated multiple times. This results in short descriptions for otherwise complex, homogenious systems. Statechart parameters can be of any type supported by the TTMC::Constraint language. Separate statecharts can communicate with each other using signals or global variables.

4.3.2 Parametric signals

Signals can also be parameterized with any integer type variable. These parameters then can be used to discriminate between signals with the same name, which also results in more readable code, allowing transitions to use the same signal as their trigger.

4.4 The statechart language

Each system description consists of a single file, which holds the specification of all components.

4.4.1 The specification

A specification can consist of multiple statecharts. Statechart definitions must be in the form of
statechart NAME(...) ...
, where the
...
part contains the description of the statechart. The braces are optional and are only needed for the parameters of statechart templates. For statechart declarations, the description can be omitted. Each specification must have at least one defined statechart. Parameterized statecharts can be created from existing templates by providing a value for each parameter.

4.4.2 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 chapter characters, integers, doubles, etc... For a complete list, see
appendix4TTMCConstraint
. The variable declaration is in the form of

global|local var NAME : TYPE
, where global or local denotes the scope of the variable.

4.4.3 Expressions and assignments

Variables can be used in expressions. Expressions can have an arbitrarily complex structure within the limits of the
TTMC::Constraint
language. This allows for, among others, the use of array indexing, parenthesis, and common operators in programming languages (+, -, *, /, modulo, ...). Assignments left hand sides are a single variable while their right hand side is an expression. Logical expressions using operators are also available (for example expressions using comparison operators). Each expression is a mixature of variables, constants, and operators. For a full reference, see
TTMC::Constraint
.

4.4.4 Parametric signals

Statecharts can communicate with the outside world and each other using signals. As such, these signals are declared directly in the specification and not in the statecharts themselves. Signals can be used with a single integer parameter (which can be either a constant or a variable). This allows for much simpler syntax when dealing with communication, as a statechart can raise a signal and pass a value simultaneusly. It also leaves room for a later expansion to a token based automata with reentry.

4.4.5 Timeouts

Raising a signal can be offset by a certain amout of time. For the formal model, the value is measured in units, for monitors, this value corresponds to the milliseconds elapsed since the timeout was set. Apart from their delayed nature, timeouts and signals can be used interchangeably.

4.4.6 Actions

Raising signals, setting timeouts, and variable assignments are called actions. They represent operations that might result in a change of the model's state.

4.4.7 Regions

Statecharts are structured by regions. Regions have both states and transitions, and play a fundament part in the scoping of elements. The syntax for regions is:

region NAME ...
Each region must contain at least an initial state for the model to be valid.

4.4.8 Transitions

Transitions describe the possible state changes. A transition can only occure if the source state is active. After the transition fired, 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 trigger

Any transition can have triggers, which are signals that enable the transition to fire when any one of them arrived. Enabled transitions without a trigger occure 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.

4.4.9 State nodes

Each region 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 the 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 performed when entering or exiting from the state. Composite states contain one or more inner regions, each with at least an initial state. A state's parent is it's containing region's containing state, or if that region does not have containing state, the region's containing statechart. Composite states help maintaining a clean model by the introduction of hierarchy, allowing common actions to be described in a 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.

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 fired, all the outgoing transitions fire as well, and the fork state itself is not entered. This results in the 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.

4.4.10 Signaling errors, error propagation

States and transitions can be labeled 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.

4.4.11 Timing of transitions, actions

Transitions are fired one by one. The firing of a transition means that the triggering signal is consumed. This can result in nondeterministic runs if two transitions share the same trigger and can be enabled simultaneously. Such models can be created but the order of the transitions are not guaranteed. Actions on the current transition being taken are processed in a single step.

4.5 Formal representation

Formal verification methods requires a flatter model that the statechart language described above. Therefore, complex concepts are mapped to an easily verifiable one.

4.5.1 Signals and expressions

In the formal model, signals are represented as boolean variables. The variable is true if the signal has been raised since the previous timestep was taken. Expressions need no further simplification.

4.5.2 Variables

Variable types are the same as in the statechart model. Local variables' scoping and names are change. All variables use the same, global scope, which could introduce name clashes. To avoid such problems, local variables' names are changed to a unique one, using their location in the hierarchical system.

4.5.3 States

States are mapped to boolean variables that signal whether the state is currently active or not. Entry and exit actions are passed to the incoming and outgoing transitions for simplicity and unified handling.

4.5.4 Timouts

4.5.5 Transitions

Transitions between states are transitions in the formal model too. If the guard condition is true, the source state is currently enabled, and the transition has no trigger or the triggering signal was raised, the transition can be taken. When the transition fires in the formal model, the exit actions of the appropriate states, the actions of the transition, and the needed entry actions occur.

4.6 Accepting monitor

4.6.1 Variables

4.6.2 Signals

4.6.3 Timeouts

4.6.4 States

4.6.5 Transitions

4.7 Implementation

4.7.1 Other utility classes

4.7.2 Timing (clock of the monitor)

4.7.3 Interface, signal pushing

4.8 Summary

Chapter 5

Complex event processing

«««< HEAD Chapter

5.1 Formal Intro of the Event Automata

In our hierarchical runtime verification project, the top level of modelling is done in an event pattern language. This pattern language will be compiled to Calendar Event Automata.

5.1.1 Current Formalisms

The two formalism we checked were the Quantified Event Automaton, and the Calendar Automaton

Event Automaton

An Event Automaton is a non-deterministic finite-state automaton whose alphabet consists of parametric events and whose transitions may be labelled with guards and assignments

> **Definition 5.1** An EventAutomat $\langle Q, A, \delta, q_0, F \rangle$ is a tuple where Q is a finite set of states, $\mathcal{A} \subseteq Event$ is a finite alphabet, $\delta \in (Q \times A \times Guard \times Assign \times Q)$ is a finite set of transitions, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states.

Calendar Event Automaton

In discrete event simulation, a calendar (also called event list) is a data structure that stores future events and the times at which these events are scheduled to occur

> **Definition 5.2** $Acalendarisafiniteset(ormultiset)of the formC = \{\langle e_1, t_1\rangle \ldots, \langle e_2, t_2\rangle\}whereeache_iisaneventandt_iisthetimewheneventeₑ_iisscheduledtooccur.Allt_isc$
> $+\infty if Cisempty)Givenarealu, wedenotebyEv_u(C)thesubsetof Cthatcontainsalleventsschedule$
> $Ev_u(C) = \{\langle e_i, t_i\rangle | t_i = u \wedge \langle e_i, t_i\rangle \in C\}$

5.1.2 Our Formalism

Akkor most tulajdonkeppen mi calendar event automatat implementaltunk :)

5.2 Examples of Event Processing

5.2.1 File System

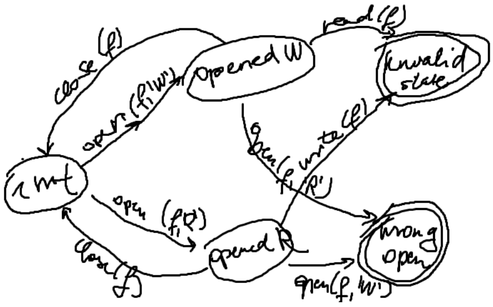


Figure 5.1 Automaton of the file example

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.

5.2.2 Mars Rover Tasking - Two phase locking

5.3 Implementation

5.3.1 Metamodel

5.3.2 Executor

======= »»»> Stash

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 [1], 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.

In this section at first we overview those concepts of the Model Railway Project which are important from our point of view. Then the extended runtime verification architecture is introduced both the hardware and software components.

It's important to notice that our solution is not tailored to this special problem but it is a general approach for any critical system.

6.2 Concept

Ábrák megvalósítása, használt technológiák hierarchiájának

6.3 System level verification with computer vision

6.3.1 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

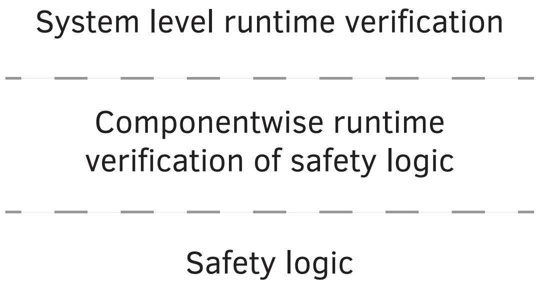

System level runtime verification
Componentwise runtime verification of safety logic
Safety logic

Figure 6.1 Overview of the hierarchical system

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.

6.3.2 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.

6.3.3 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

[1] http://opencv.org/

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

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 6.3.4, 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 6.2).

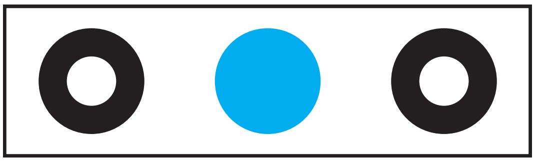


Figure 6.2 The final marker design

6.3.4 Mathematical solution for marker detection

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

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 controlling theory.

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.

As **??** shows, we can multiply two spectrums element-wise, 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.

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

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 6.3). The raw pixels are multiplied by this value. The meaning of these values in the bitmap are the following:

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

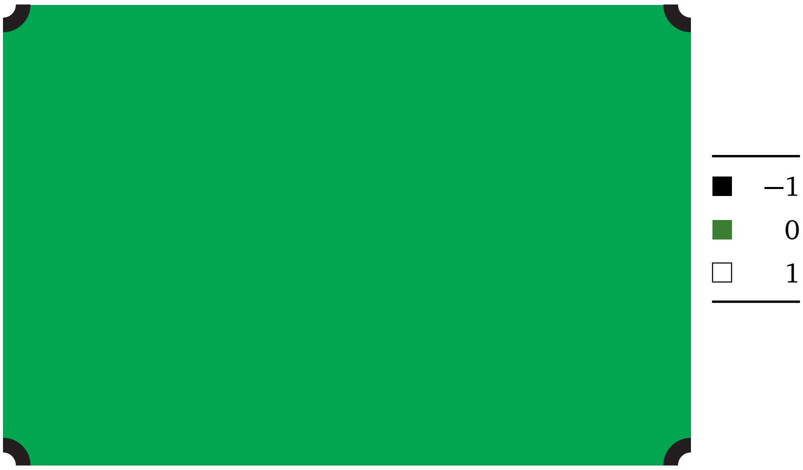
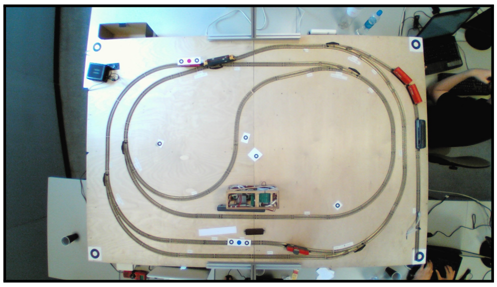
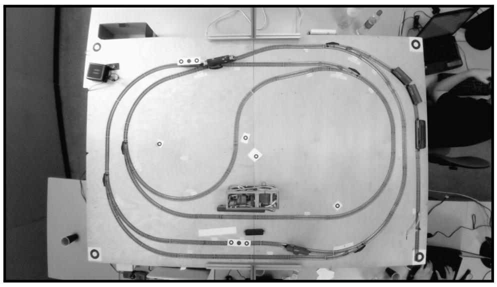
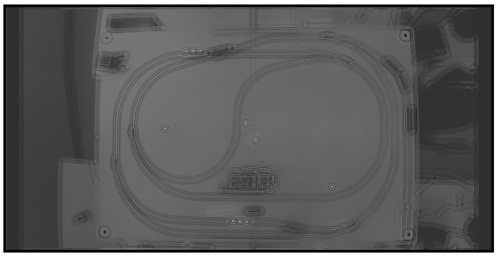


Figure 6.3 Pattern bitmap placement and value example

6.3.5 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.

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

Table 6.1 Computer vision processing pipeline

6.4 Model railroad

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

6.4.1 Overview

The model railroad (Figure 6.4) contains the following hardware elements:

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

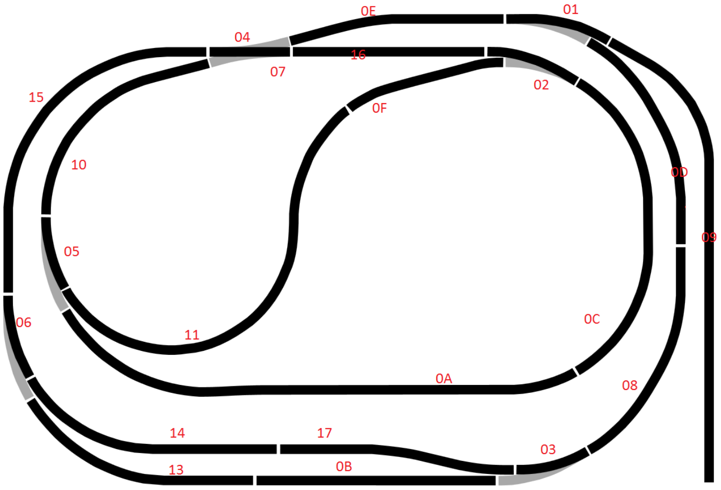


Figure 6.4 The railroad network with section IDs

6.4.2 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 6.5).

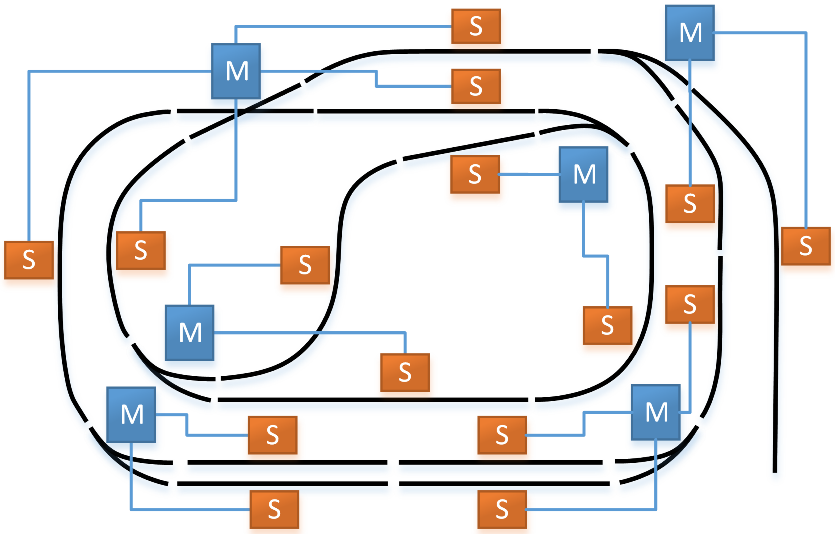


Figure 6.5 Master-slave associations

6.5 ???

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.

6.5.1 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.
- **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.

6.5.2 Logical breakdown of 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 6.5.6), and review the main components of the logical elements.

- **Trackable**: The atomic abstract element of our model, the *trackable* is the smallest unit of measurement.
- **SectionTrackable**: Specialized trackable, which is a part of a powerable section.
- **SwitchTrackable**: Specialized trackable. Because we did not intrested in the position inside the switch, the entire

6.5.3 Introducing to EMF

Az EMF bemutatása röviden.

6.5.4 Building the EMF model

Az elkészült EMF metamodell bemutatása, és összevetése az elképzelésekkel.

6.5.5 Introducing the IncQuery

Az IncQuery bemutatása.

6.5.6 Building the IncQuery patterns

A biztonsági lokikai patternek bemutatása.

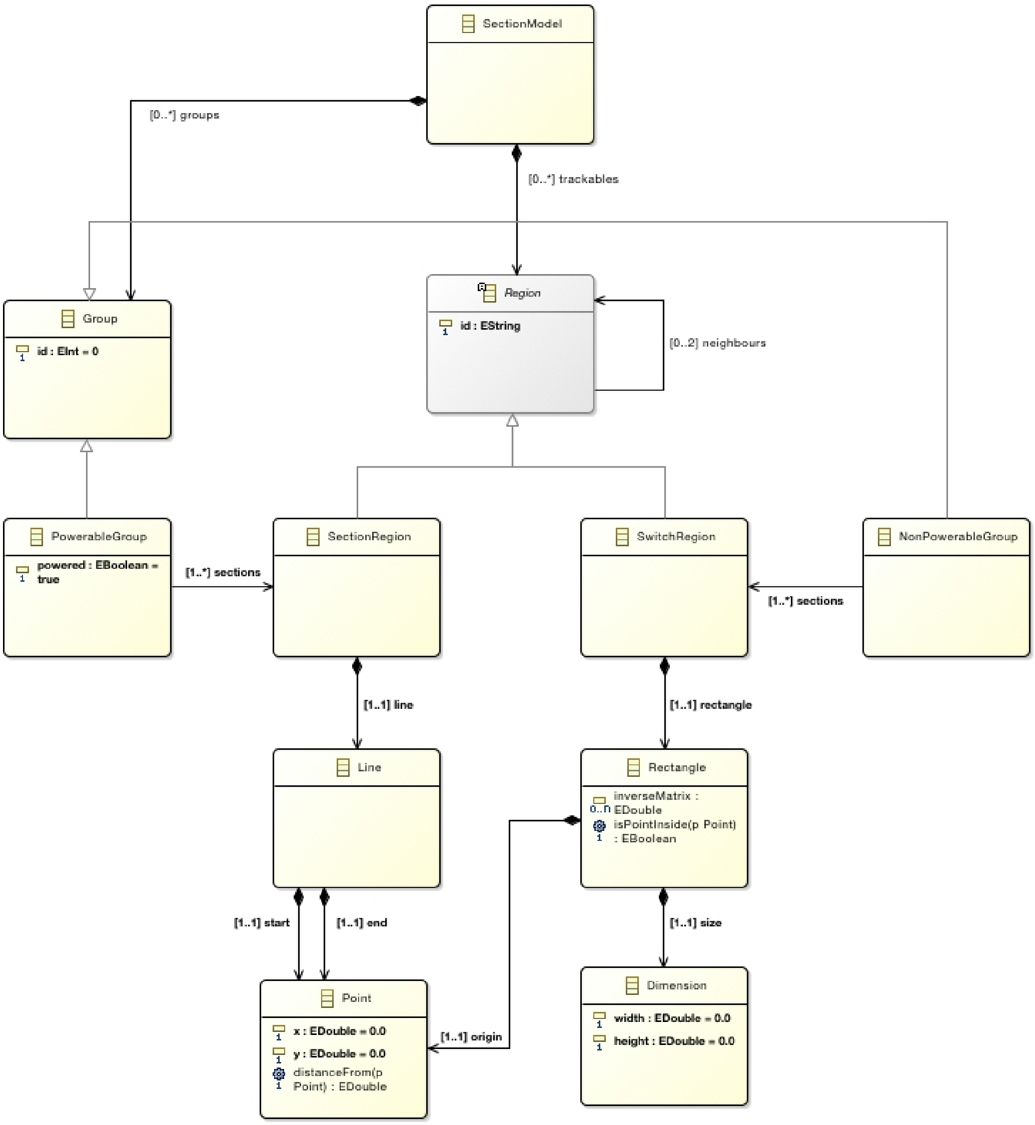


Figure 6.6 EMF model

Chapter 7

Conclusion

Chapter

Chapter 8

Acknowledge

Itt köszönjük meg!

References

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