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.

Chapter 1

Introduction

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 2

Background

Chapter

Chapter 3

Overview of the approach

This case study concentrates on the connection between the design time, and the runtime domain. Our goal is to connect these two domains, and make a solution that generally can support the relation between the design, and runtime.

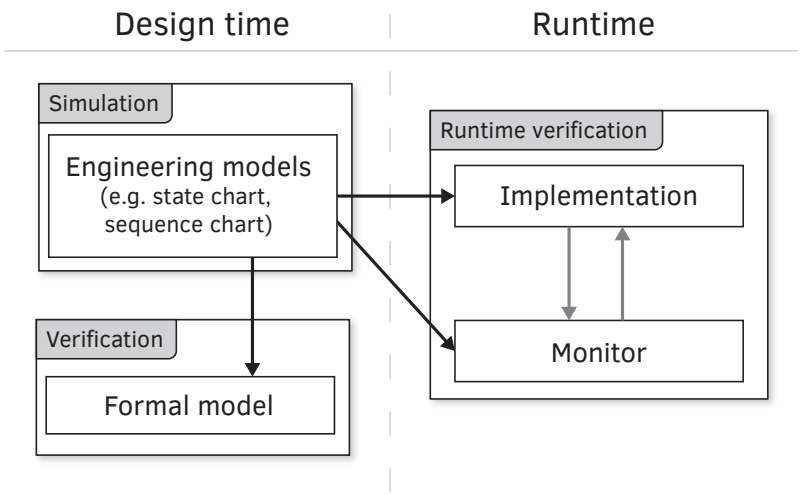


Figure 3.1 Connection between runtime and design time elements

Figure 3.1 depict the basic concept. We have three main transformation in the concept:

- **Engineering model ↦ Formal model**:
- **Engineering model ↦ Generated code**: From the verified model, we can generate an executable implementaion.

- **Engineering model ↦ Monitor**: Monitor the state of the implementation. If the execution reaches invalid states, the monitor detect it and forward the event of error detection into a higher level processor.

Solutions which connect two, or three elements of mapping are existing, but there are no tools which integrate all these in one tool with interchangeable formalisms.

Let us consider the following example:

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

- Generate the implementation from the model.
- Implement runtime verification into our implementation, by monitoring the application.
- We have multiple monitoring systems, and if a components verification state changes we want to react.

This example covers the usual needs 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.

3.0.1 Hierarchical levels

At the concept level

Chapter 4

Runtime verification of embedded systems

Chapter 5

Complex event processing

5.1 Intro of the Complex Event Processing

In this hierarchical runtime verification project, the top level of modelling is done in an event pattern language. This event pattern language is translated to timed event automatons. These event automatons will be executed in the process.

5.2 Formal Intro of the Timed Parametrized Event Automaton

5.2.1 VEPL

Our choice for the event pattern definition is the VIATRA Event Pattern Language (VEPL). Currently this is the only CEP which can be easily integrated to a live model, where you can define multiple graph patterns over the model, and define atomic events for the appearance and disappearance of these patterns.

TODO define complex event processing

A brief overview of the VEPL language:

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.
and	p1 AND p2	Both of the patterns has to appear, but the order does not matter. Rule: p1 AND p2 ≡ ((p1 → p2) OR (p2 → p1)).
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. Rule: p{n} ≡ p1 → p1 → ... p1, n times.
"at least once" multiplicity	p{+}	The pattern has to appear at least once.
"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.

5.2.2 Timed Regular Expression

Definition 5.1 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.

5.2.3 Finite State Machine

Definition 5.2 A FiniteStateMachine $\langle Q, \Sigma, \delta, q_0, F \rangle$ 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 abc of the automaton.
- $\delta : Q \times \Sigma \to Q$, the state transition function of the automaton.
- $q_0 \in Q$ a start state
- $F \subseteq Q$ the set of the acceptor states

Definition 5.3 A NonDeterministic FiniteStateMachine $\langle Q, \Sigma, \delta, q_0, F \rangle$ where the meaning of Q, Σ, q_0, and F is the same as Definition 5.2, except for for δ which is

$$\delta(q, a) \subseteq Q$$

where $q \in Q$ and $a \in \Sigma \cup \{\varepsilon\}$

5.2.4 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 5.4 An EA $\langle Q, \mathcal{A}, \delta, q_0, F \rangle$ is a tuple where Q is a finite set of states, $\mathcal{A} \subseteq Event$ is a finite alphabet, $\delta \in (Q \times \mathcal{A} \times Guard \times Assign \times Q)$ is a finite set of transitions, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states[2].

5.2.5 Timed Event Automaton

Definition 5.5 A TimedZone $\langle S, \mathcal{C}, I, T \rangle$ is a tuple where $S \in$ States, $T \in$ States, $\mathcal{C} \subseteq$ States, I is a time value

The semantic of the TimedZone is the following: If a token enters state S at t_i, and doesn't reach any of the $\mathcal{C}$ states before t_j, where $t_j = t_i + I$, then the token is moved to T

Definition 5.6 A TEA $\langle E, \mathcal{T} \rangle$ is a tuple where E is an Event Automaton, and $\mathcal{T}$ is a set of Timed Zones, where all the states are one of the Automatons states.

5.2.6 Compilation of the Event patterns to Parametrized Event Automata

5.3 Examples of Event Processing

5.3.1 Sequence chart example

5.3.2 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 [5]. 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

5.3.3 Mars Rover Tasking - Two phase locking

Problem

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

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

Solution

Since our implementation doesn't support guards *yet* we can only use constant amount of resources. For this example, this amount will be set to two, to minimise the model of the example. The Item 1 pattern can be matched with the Figure 5.2, and the Item 2

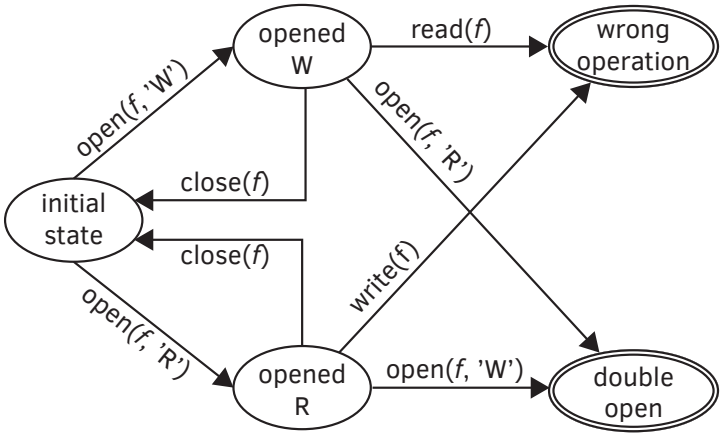


Figure 5.1 Automaton of the file example

Figure 5.2 Automaton to forbid the reallocation

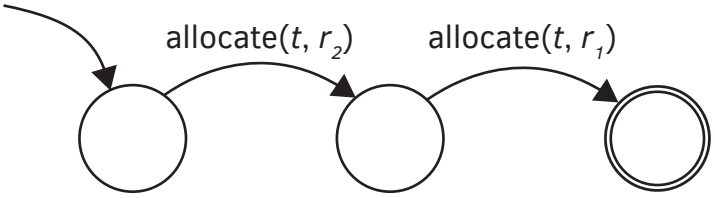


Figure 5.3 Automaton to forbid inverse allocation

5.4 Implementation

5.4.1 Metamodel

Basic Automaton

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

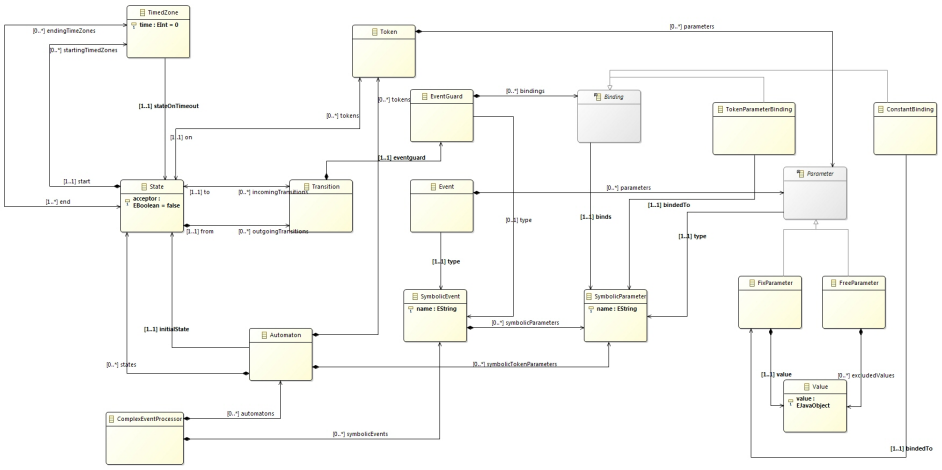


Figure 5.4 Automaton of the file example

Timing

Parameter

Binding

5.4.2 Executor

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

Chapter 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 [3], 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.

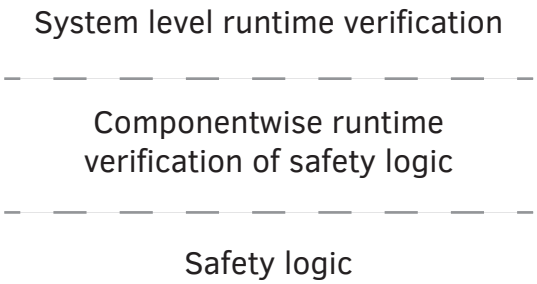


Figure 6.1 Overview of the hierarchical runtime verification system

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

Our main goal with this study is to develop a method which can integrate multiple safety logic into a global runtime verification, increasing the reliability of the complete system.

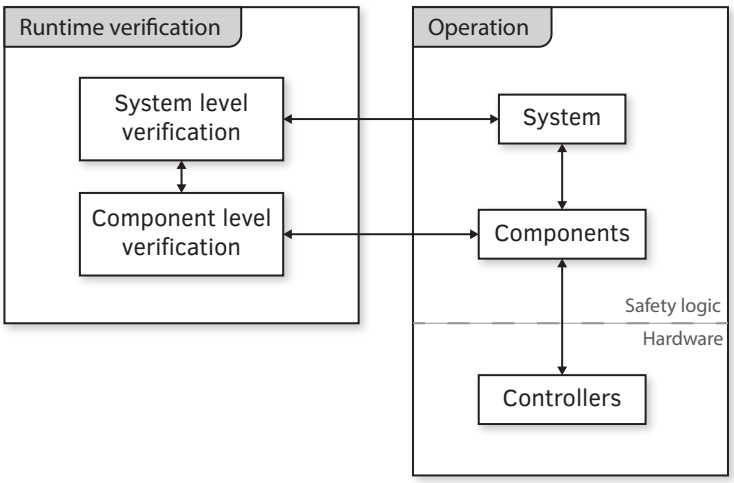


Figure 6.2 Associations between operation, and runtime verification components

Figure 6.2 shows the main relation between the operation, and runtime verification domain. 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.

In our case (Figure 6.3), there are embedded components, and one system level component.

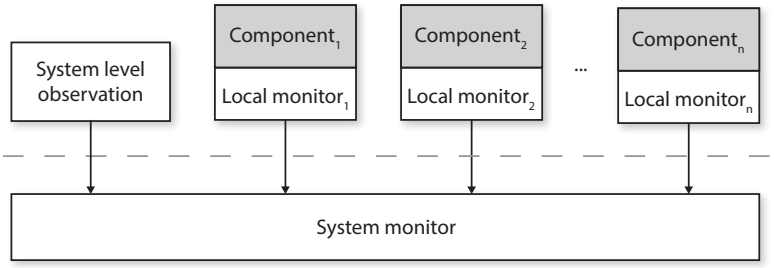


Figure 6.3 Sensor, and monitor relation

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 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.

[1] http://opencv.org/

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 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 6.4).

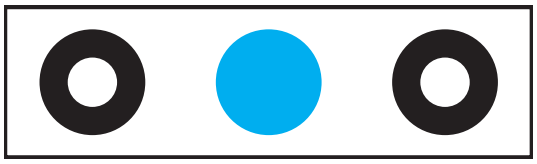


Figure 6.4 The final marker design

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 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

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

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 6.5). 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.

Figure 6.5 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,

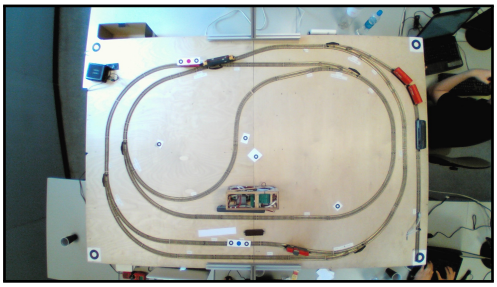
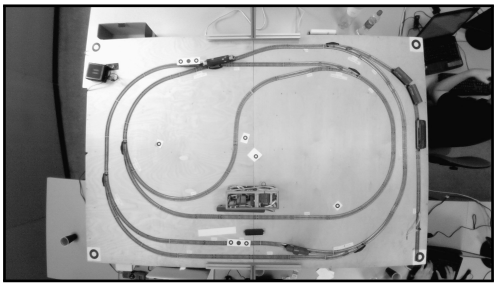
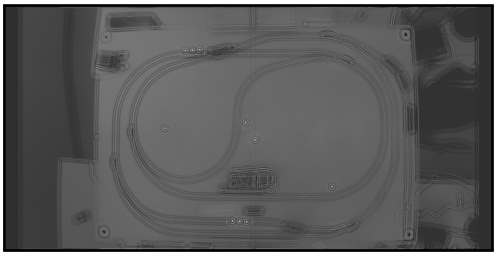
[3]In our application raw (or raw image) means the unprocessed image from the camera

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.

6.4 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.

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

6.5 Model railroad

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

6.5.1 Overview

The model railroad (Figure 6.6) contains the following hardware elements:

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

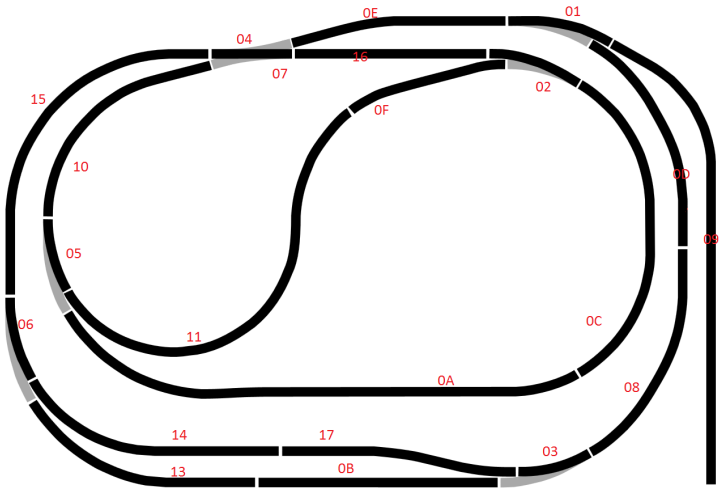


Figure 6.6 The railroad network with section IDs

6.5.2 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 6.7).

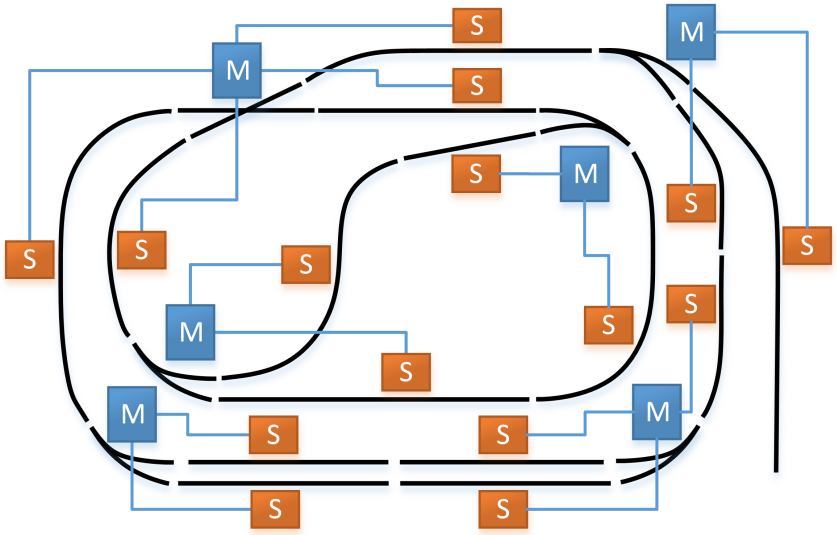


Figure 6.7 Master-slave associations

6.6 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.

6.6.1 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.

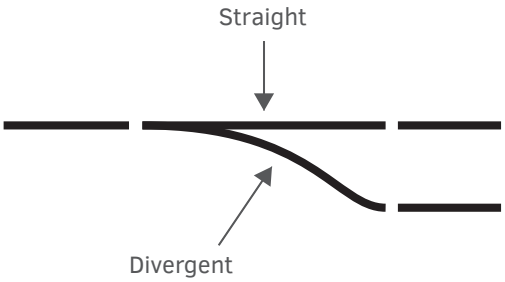


Figure 6.8 Switch straight, and divergent rails

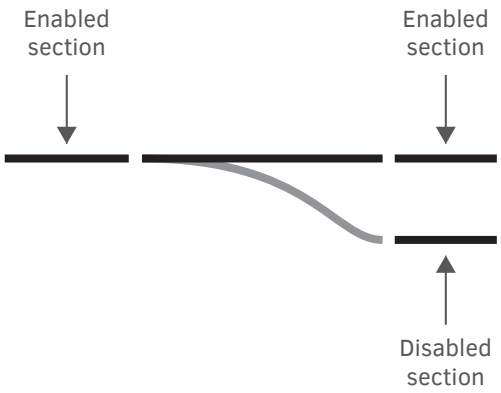


Figure 6.9 Enabled-disabled section explanation

6.6.2 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 6.6.6), and review the main components of the logical elements.

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.

    a) **Configuration**: Contains the enabled *groups* of the switch.

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

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

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

    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.

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

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

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

2. **TrainModel**: The root of all train elements.

    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.

6.6.3 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." [4]

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:

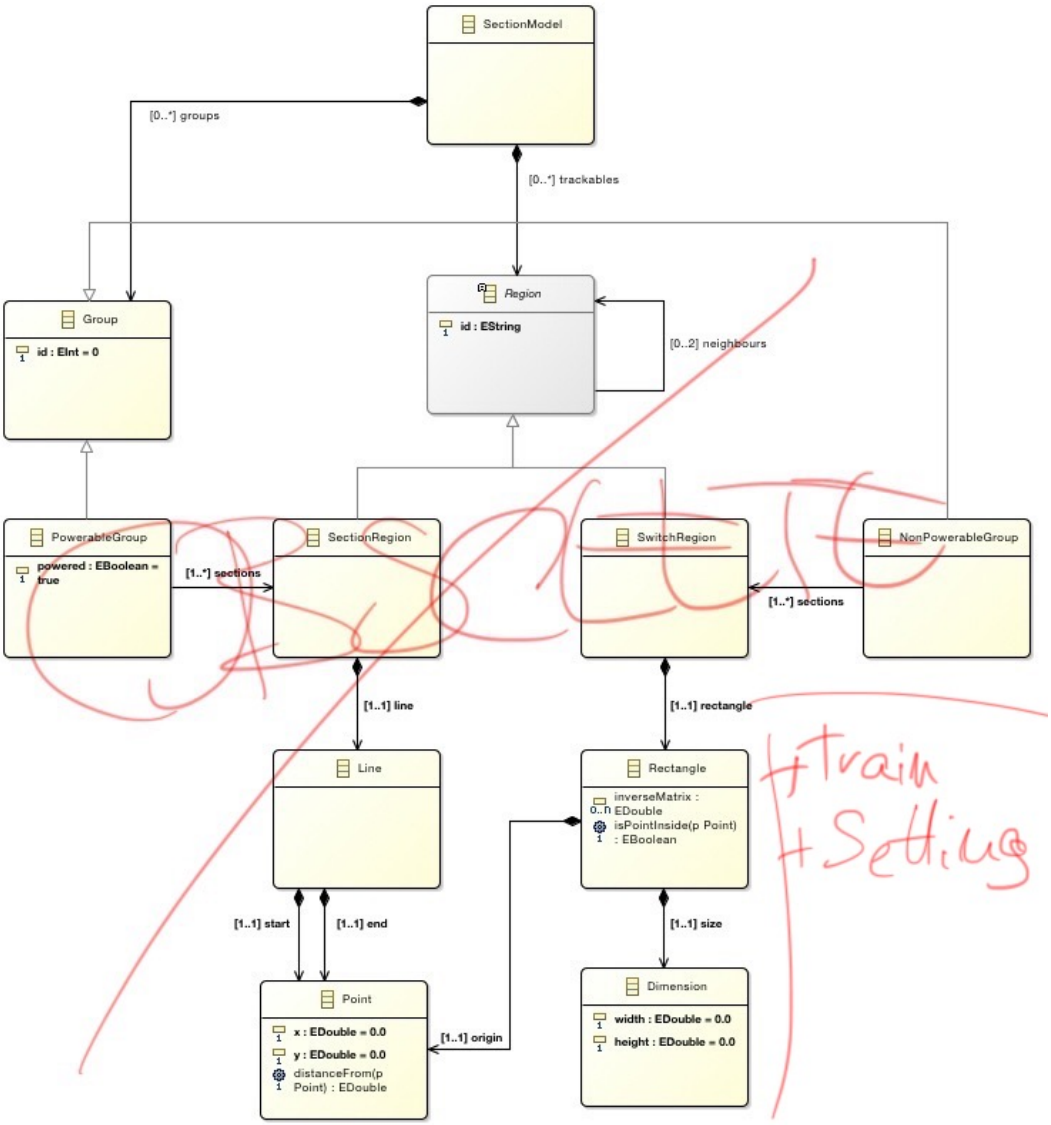


Figure 6.10 The metamodel of the *Model Railway Project* model

- 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. TODO The editor ensure the 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.

6.6.4 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 6.6.2 the *TrainModel* and *SectionModel* represents the root of the model.

6.6.5 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:
>
> - 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

[4]Plain Old Java Object

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.

6.6.6 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 6.3 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 6.3 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.

6.7 Summary

The Train Benchmark[6] 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] Howard Barringer, Ylies Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. “Quantified event automata: Towards expressive and efficient runtime monitors”. In: *FM 2012: Formal Methods*. Springer, 2012, pp. 68–84.

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

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

[5] 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.

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