



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

Hierarchical runtime verification for critical cyber-physical systems

Scientific Students' Associations Report

Authors:

László Balogh
Flórián Deé
Bálint Hegyi

Supervisors:

dr. István Ráth
dr. Dániel Varró
András Vörös

2015.

Contents

Contents	iii
Abstract	vi
1 Introduction	1
2 Background	3
3 Overview	5
4 Runtime verification of embedded systems	7
4.1 Intro	7
4.2 Goal	7
4.2.1 Simple, generic, usable	7
4.2.2 Verification	8
4.2.3 Monitor generation	8
4.3 A new approach is needed	8
4.3.1 Parametric statechart declaration	8
4.3.2 Parametric signals	9
4.4 The statechart language	9
4.4.1 Specification	9
4.4.2 Variables	9
4.4.3 Expressions and assignments	9
4.4.4 Parametric signals	10
4.4.5 Actions	10
4.4.6 Regions	11
4.4.7 Transitions	11
4.4.8 State nodes	11
4.4.9 Signaling errors, error propagation	12
4.4.10 Timing of transitions, actions	12
4.5 Formal representation	13

4.5.1	Statecharts	13
4.5.2	Signals	13
4.5.3	Timeouts	14
4.5.4	Variables and expressions	14
4.5.5	States	14
4.5.6	Transitions	14
4.5.7	Verification	14
4.6	Accepting monitor	15
4.6.1	Variables and expressions	15
4.6.2	Signals	15
4.6.3	Timeouts	15
4.6.4	States	15
4.6.5	Transitions	16
4.6.6	Statecharts	16
4.7	Implementation	17
4.7.1	Timing related issues	17
4.7.2	Utility classes	17
4.7.3	Signal pushing	18
4.7.4	Error signalling	18
5	Complex event processing	19
5.1	Intro of the Complex Event Processing	19
5.2	Formal Intro of the Timed Parametrized Event Automaton	19
5.2.1	VEPL	19
5.2.2	Timed Regular Expression	20
5.2.3	Event Automaton	21
5.2.4	Timed Event Automaton	21
5.2.5	Compilation of the Event patterns to Parametrized Event Automata	21
5.3	Examples of Event Processing	21
5.3.1	File System	21
5.3.2	Mars Rover Tasking - Two phase locking	22
5.4	Implementation	23
5.4.1	Metamodel	23
5.4.2	Executor	23
6	Case study	25
6.1	Overview	25
6.2	Concept	26
6.3	System level verification with computer vision	27
6.3.1	Hardware	27

6.3.2	OpenCV	27
6.3.3	Marker design	28
6.3.4	Mathematical solution for marker detection	28
6.3.5	Software	29
6.4	Summary	30
6.5	Model railroad	32
6.5.1	Overview	32
6.5.2	Hardware	32
6.6	???	33
6.6.1	Physical elements	33
6.6.2	Logical breakdown of physical elements	34
6.6.3	Introducing to EMF	34
6.6.4	Building the EMF model	34
6.6.5	Introducing the IncQuery	34
6.6.6	Building the IncQuery patterns	34
7	Conclusion	37
8	Acknowledge	39
References		41

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 bekapsolódnak, 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 felelősséggű 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 szintéziséét 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 developed to support 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 modelling in various branches of engineering.

4.2 Goal

The goal was to create a formal statechart language that resembles languages already used in engineering practices. This would shorten the learning process and enable formal verification of the resulting models, which would result in highly dependable safety critical systems with lower design and maintenance costs.

4.2.1 Simple, generic, usable

Statecharts build on the concept of state machines. They are highly visual and are used by many branches of engineering as a modelling method. Statecharts can be easily used for abstraction, providing a generic way to describe small processes or whole systems. States can be refined by creating inner regions and substates, or by creating smaller, separate statecharts to model the underlying concepts in detail. One of the languages goals was to provide a simple and usable, yet generic approach to modelling using this well known method.

4.2.2 Verification

The development of safety critical systems require extensive testing. While well design tests can detect most of a system's faults they can't promise that all them are found. Formal verification is a more reliable approach as it guarantees that all of the (formally) specified requirements are met.

4.2.3 Monitor generation

Even with formally verified systems, hardware errors can occur. Using runtime verification one can signal and handle malfunctions so that the system stops or remains as functional as possible. Monitors are also capable of providing traces that can unravel the propagation of errors. Therefore, the statechart language provides tools for marking erroneous states and transitions, and is capable of generating monitors that track the system's state through a signal interface. These monitors contain the state model of the described system, which makes it possible to use them as skeletons capable of reporting errors for smaller programs.

4.3 A new approach is needed

Many software is available for code generation and verification based on statecharts. Unfortunately the available solutions for verification and modelling provide limited syntax, and the latter usually provide poor quality code. Both results in longer creation and harder maintenance of the models and software. This leads to higher costs and a more time consuming development cycle than necessary. For these reasons, our statechart language supports the verification of statecharts with parametric signals, timers, while enabling the creation of parametric statechart. For monitor generation, our approach was to generate easily readable, extendible, object oriented code that can run in an environment with limited resources, primarily embedded systems.

4.3.1 Parametric statechart declaration

The language allows a specification to consist of multiple statecharts. This feature led to one 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, homogeneous 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 Specification

A specification can consist of multiple statecharts. Statechart definitions must be in the form of:

```
statechart NAME(....) {....}
```

where (...) is the parameter list and 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. Parametrized 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 [TODO-ref]. The variable declaration is in the form of:

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

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

4.4.3 Expressions and assignments

Variables can be used in expressions. Expressions can have an arbitrarily complex structure within the limits of [TODO-ref]. This allows for, among others, the use of array indexing, parenthesis, and common operators in programming languages such as +, -, *, /, and such. Assignments left hand sides must be 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 mixture of variables, constants, and operators. For a full reference, see [TODO-ref].

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 simultaneously. It also leaves room for a later expansion to a token based automata with reentry. 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.

4.4.5 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.

Signal raising with timeout

Raising a signal can be offset by a certain amount 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, 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 amount of timesteps before the signal is raised.

Variable assignment

Variable assignments can be done with the syntax:

```
assign var := expression
```

where *var* is a variable reference that can be both global or local, and *expression* is a valid expression.

4.4.6 Regions

Statecharts are structured by regions. Regions have both states and transitions, and play a fundamental 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.

4.4.7 Transitions

Transitions describe the possible state changes. A transition can only occur 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 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.

4.4.8 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 its 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.

Choice states

Choice states are pseudo states with outgoing transition of which exactly one guard condition true. Choice states let the user create tree-like transition structures with simple guards conditions on each level instead of complex guard conditions on 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 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 its guard is true and all the incoming transitions' source states are active. Triggers can be declared on the outgoing transition.

4.4.9 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.

4.4.10 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 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 are not guaranteed. Actions on the current transition being taken are processed in a single step.

4.5 Formal representation

Formal verification methods require a flatter model than the statechart language described above. To enable formal verification of the defined statecharts, the language can be mapped to a transition system. This means that complex concepts of the model are flattened out to easily verifiable ones. The used transition system has the following capabilities:

- multiple systems can be defined
- systems can use local and global variables
- the state of a system can be defined as a vector of all referenced variables
- a partition of a system's state space can be represented as an expression that restricts certain variables' values and leaves the rest unbound
- such restrictions are logical formulas that may use $=$, $<$, \leq , $>$, etc...
- systems have transitions that represent changes of state using the referenced variables previous and current values

4.5.1 Statecharts

Each statechart is mapped to a separate transition system. As transition systems are treated as systems running parallelly, this models the original behaviour of the statecharts - they define systems that can communicate with each other but are running independently.

4.5.2 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 statechart. Since each signal can only trigger a single transition simultaneously inside a given statechart, complexer solutions are not needed. The parameters of the signals are stored and can be referenced as global integer variables.

4.5.3 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.

4.5.4 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.

4.5.5 States

States are mapped to boolean variables that represent 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. This way any transition that would result in the entering or exiting of the state executes the appropriate actions.

4.5.6 Transitions

Transitions between states are transitions in the formal model too. For a transition to be enabled, guard conditions has to evaluate 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 has to be raised. When the transition fires in the formal model, the exit actions of the appropriate states, the actions associated with the transition, and the needed entry actions occur.

4.5.7 Verification

The verification of the transition system is based on LTL expressions. (TODO?)

4.6 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 hierachic runtime verification possible.

4.6.1 Variables and expressions

The variable types of the statechart language are plain old data types in C++. The expressions that can be made are also a subset of the available operators, so the mapping between the statechart language and the generated monitor adds no restrictions to the description of the model. Local variables are translated to global variables with fully qualified names and are put into a handler object to allow unified usage.

4.6.2 Signals

Signals are represented as instances of the signals 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.

4.6.3 Timeouts

Timeouts are signal raises that will occur in the future. When setting a timeout, the same named and parametrized object is created as above, and are put in a que that holds pairs of signals and timestamps. The timestamp represents when the signal should go off.

4.6.4 States

States are named objects derived from a general state class that has an empty entry and exit action. The states also store whether they are initial or not. If a state has entry or exit actions a child class is generated with the appropriate entry and exit actions. If

a class has no such actions, the general state class is instantiated using the state's fully qualified name.

4.6.5 Transitions

Transitions are represented as objects connecting two states. An object of the generic transition class has an `isEnabled` function that always return true, and an empty action function that simply returns, and a null value that can reference a list of states. The list itself is only created for fork and join states. The triggers of the transitions are handled in a separate mapper class.

Simple transitions

That 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 on instantiation.

Transitions of choice states

Transitions from, and to choice states are handled by a statechart preprocessor that unfolds choices to simple transitions with guard conditions before the monitor generation begins.

Transitions of fork 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.

Transitions of join states

Incoming and outgoing transitions of a join state are 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.

4.6.6 Statecharts

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

4.7 Implementation

The implementation of the monitoring component requires additional helper classes and functions for the already described ones. It should also be noted that the generated C++ code heavily relies on the new features of C++11.

4.7.1 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, the shortest settable sleep and timeout period of the monitor is one millisecond. This means that for each step 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. go to sleep

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.

4.7.2 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.

StatechartRegistry

Separate statecharts have functions for taking transitions and refreshing enabled states. To simplifying their handling, a utility class was created that can automatically iterate through each statechart and update their state. This class is also responsible for the initialization process.

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`.

4.7.3 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.

4.7.4 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 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	$p_1 \rightarrow p_2$	Both patterns have to appear in the specified order.
or	$p_1 \text{ OR } p_2$	One of the patterns has to appear.
and	$p_1 \text{ AND } p_2$	Both of the patterns has to appear, but the order does not matter. Rule: $p_1 \text{ AND } p_2 \equiv ((p_1 \rightarrow p_2) \text{ OR } (p_2 \rightarrow p_1))$.
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: $p\{n\} \equiv p_1 \rightarrow p_1 \rightarrow \dots p_1$, 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 $\varphi_1 \circ \varphi_2, \varphi^\otimes$ 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 $\Theta : \Sigma_0 \rightarrow \Sigma$ is a renaming, then $\varphi_1 \wedge \varphi_2$ and $\Theta(\varphi_0)$ are Σ -expressions.

5.2.3 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 5.2 An EA $\langle Q, \mathcal{A}, \delta, q_0, F \rangle$ is a tuple where Q is a finite set of states, $\mathcal{A} \subseteq \text{Event}$ is a finite alphabet, $\delta \in (Q \times \mathcal{A} \times \text{Guard} \times \text{Assign} \times Q)$ 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].

5.2.4 Timed Event Automaton

Definition 5.3 A TimedZone $\langle S, \mathcal{E}, I, T \rangle$ is a tuple where $S \in \text{States}$, $T \in \text{States}$, $\mathcal{E} \subseteq \text{States}$, 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{E} states before t_j , where $t_j = t_i + I$, then the token is moved to T

Definition 5.4 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.

5.2.5 Compilation of the Event patterns to Parametrized Event Automata

5.3 Examples of Event Processing

5.3.1 File System

Problem

File system - A file shouldn't be read when it has been opened for writing, and shouldn't be written, when opened for reading. A file shouldn't be opened for writing and reading without a close event between the two different opens [4]. 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);
```

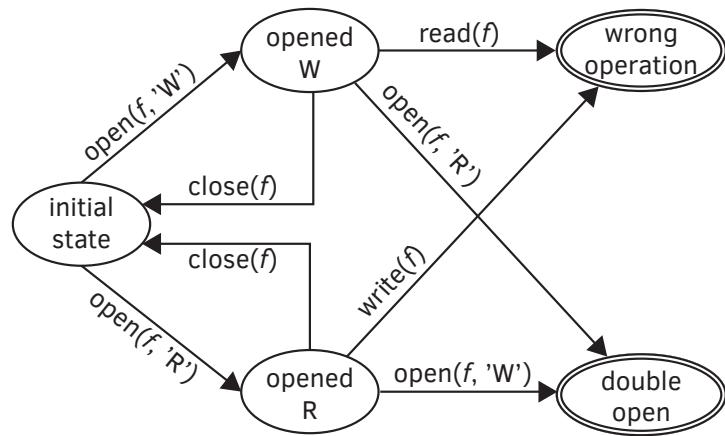


Figure 5.1 Automaton of the file example

5.3.2 Mars Rover Tasking - Two phase locking

Problem

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

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

Solution

Since our implementation doesn't support guards yet we can only use constant amount of resources. For this example, this amount will be set to two, to minimise the model of the example.



Figure 5.2 Automaton to forbid the reallocation

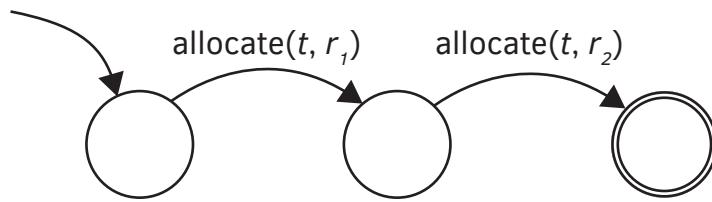


Figure 5.3 Automaton to forbid inverse allocation

5.4 Implementation

5.4.1 Metamodel

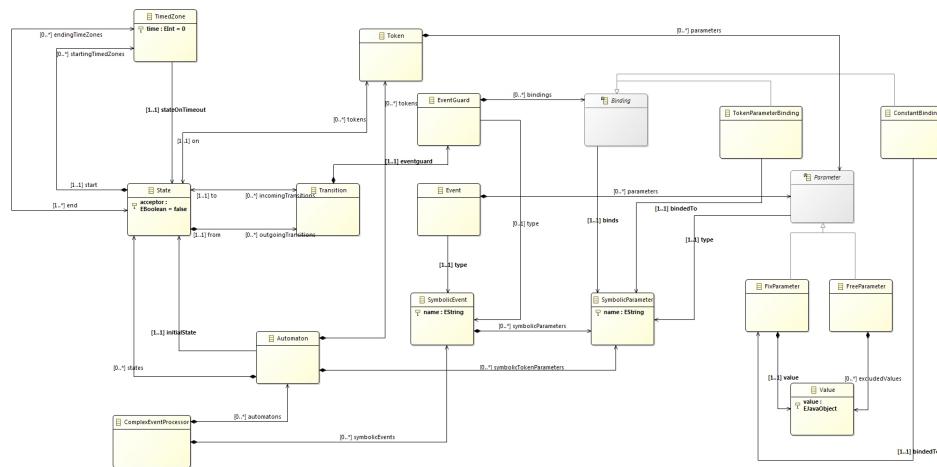


Figure 5.4 Automaton of the file example

Nyilvan ide majd kisebb, mutatosabb, es kevesebb dolgot tartalmazo abra kell, meg nem magyarázat Egy csak timedZones resz, meg egy csak parameteres

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.

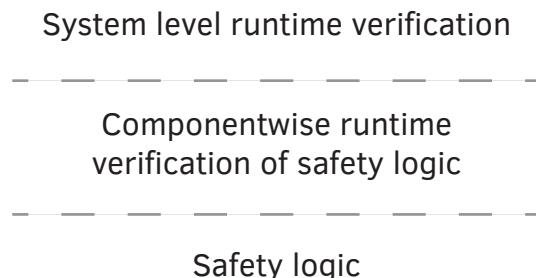


Figure 6.1 Overview of the hierarchical runtime verification system

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.

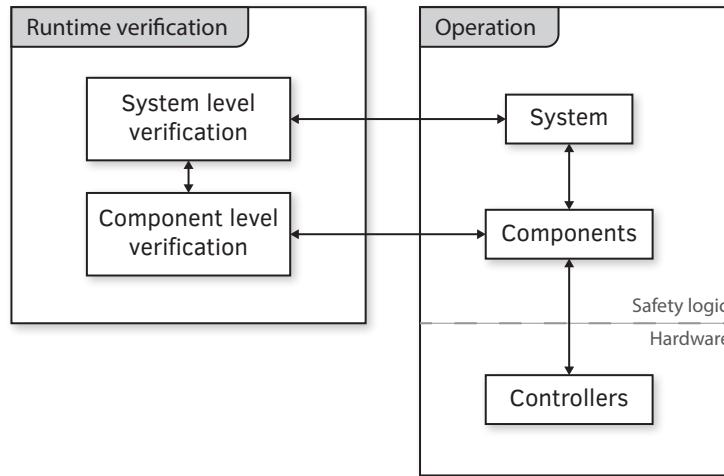


Figure 6.2 Associations between operation, and runtime verification components

Figure 6.2 shows the main relation between the operation, and runtime verification domain. With component level verification – in our 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.

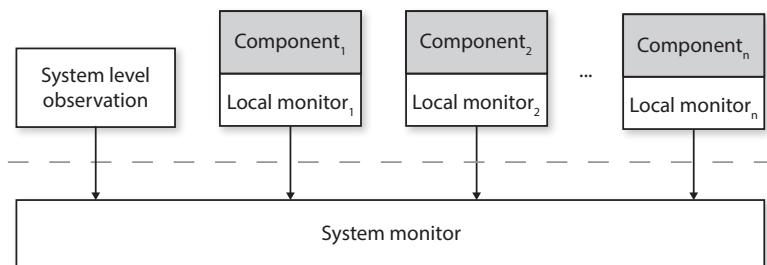


Figure 6.3 Sensor, and monitor relation

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 OpenCV¹ 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 ArUco². 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).

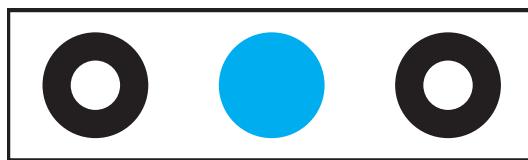


Figure 6.4 The final marker design

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 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

²<http://www.uco.es/investiga/grupos/ava/node/26>

image is the raw³, 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:

- ***value = 0***: Doesn't affect the match.
- ***value > 0***: The multiplied raw pixel summed positively to the result of the convolution.
- ***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,

³In 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 through 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.

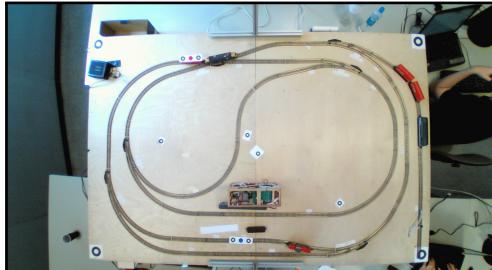
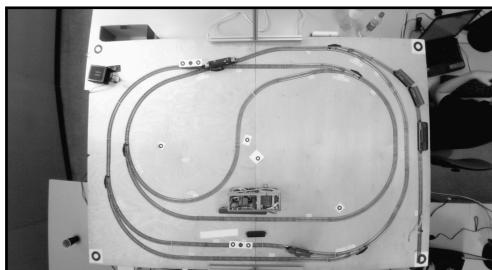
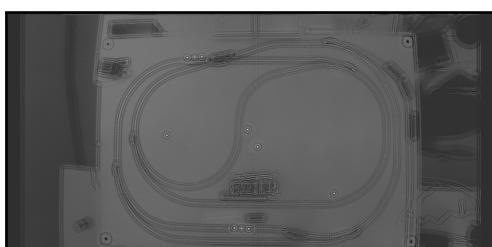
Stage #	Description	Example images
1	Loading an image from the camera	
2	Convert the image to grayscale	
3	Convolve the image with the pattern	
Stage #	Description	
4	Applying a threshold to filter the brightest spots	
5	Finding the contours of the enclosed shapes	
6	Calculating the center point of the 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

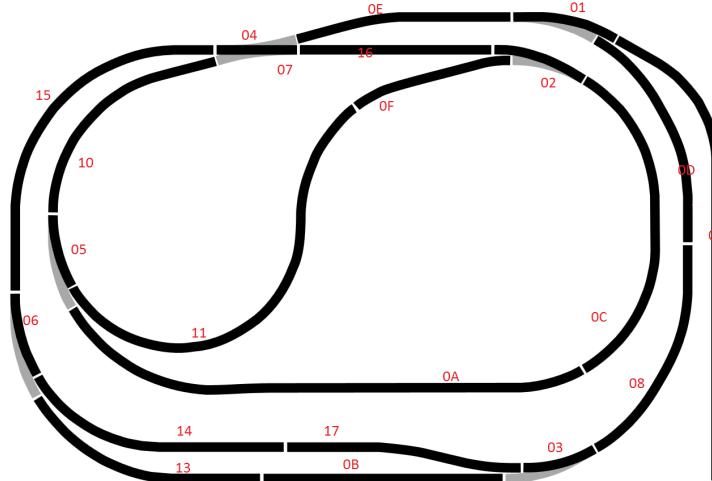


Figure 6.6 The railroad network with section IDs

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).

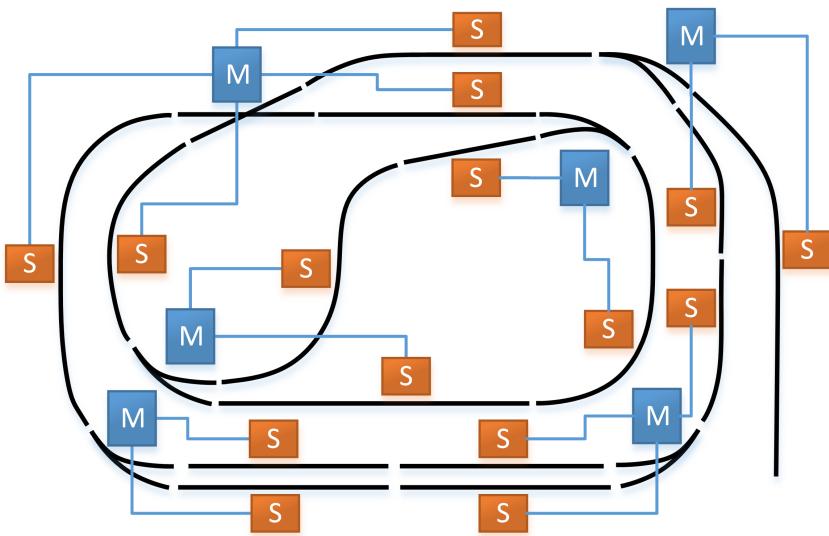


Figure 6.7 Master-slave associations

6.6 ???

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.
- **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.

6.6.2 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.

- **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 interested in the position inside the switch, the entire

6.6.3 Introducing to EMF

Az EMF bemutatása röviden.

6.6.4 Building the EMF model

Az elkészült EMF metamodel bemutatása, és összevetése az elképzélésekkel.

6.6.5 Introducing the IncQuery

Az IncQuery bemutatása.

6.6.6 Building the IncQuery patterns

A biztonsági lokikai patternek bemutatása.

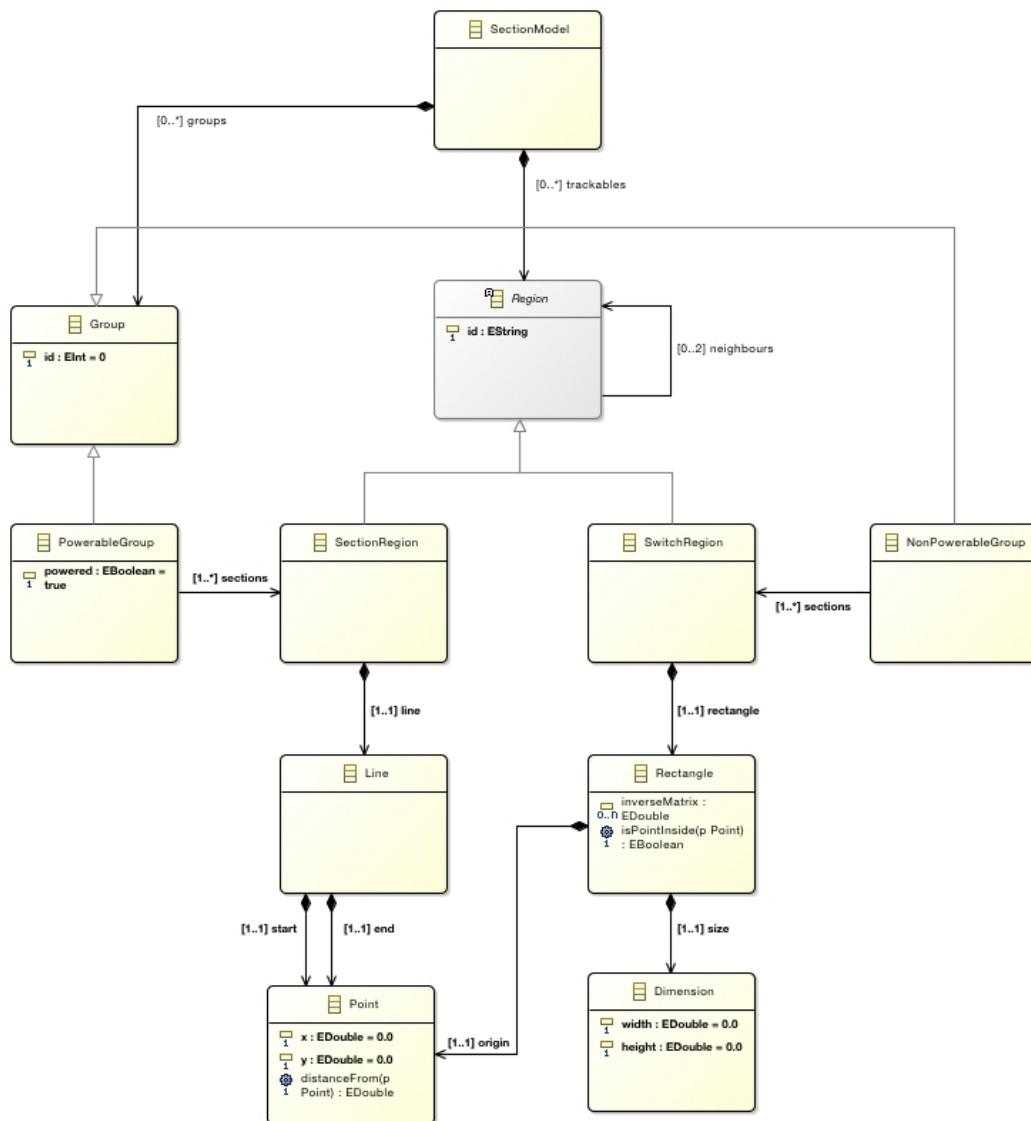


Figure 6.8 EMF model

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] 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.