



DIPLOMATERVEZÉSI FELADAT

Balogh László Márk

mérnök informatikus hallgató részére

Automataelméleti komplexesemény-feldolgozás algoritmusainak vizsgálata

A komplexesemény-feldolgozó rendszerek feladata a környezetből jellemzően nagy mennyiségben érkező információ hatékony feldolgozása, azaz előre definiált minták, trendek keresése az eseményfolyamban. Az eseményfeldolgozás célja előre meghatározott minták felismerése, amelyek az analízis szempontjából érdekes eseményszekvenciákat azonosítják.

A komplexesemény-feldolgozásnak kiterjedt irodalma van, a legtöbb megoldás szemantikája azonban nem egyértelmű, többnyire ad-hoc módon készült, ami megnehezíti a használatukat. Különösen igaz ez a kritikus rendszerek ellenőrzésekor, ahol fontos lenne precízen definiálni a működés analízis szempontjából releváns részeit.

További kihívást jelent (a) ha a minták a beérkező események sorrendisége mellett azok időzítési viszonyait is vizsgálják; (b) ha nincs egy előre ismert véges eseménytér, hanem paraméteres események összevetésére kell felkészülni a paraméterek összes lehetséges értéke mellett; végül (c) ha a feldolgozás hatékonysága, teljesítménye kritikus.

A hallgató feladata megvizsgálni az irodalomban ismert legfontosabb automataelméleti megközelítéseket, és ez alapján egy olyan megközelítést javasolni, amely precíz szemantika mentén hatékonyan végrehajtható komplexesemény-feldolgozást tesz lehetővé.

A hallgató feladata a következő elemekből áll:

1. Vizsgálja meg az irodalomban ismert automataelméleti megközelítéseket a komplexesemény-feldolgozás hatékony megvalósítására!
2. Az irodalom alapján javasoljon megoldást precíz szemantika alapján történő és hatékony komplexesemény-feldolgozásra! A vizsgálatok során térjen ki a paraméteres és időzített tulajdonságok analízisére is!
3. Implementáljon egy prototípus megoldást!
4. Értékelje ki a megoldást és vizsgálja meg a továbbfejlesztési lehetőségeket.

Tanszéki konzulens: Vörös András, tudományos segédmunkatárs

Budapest, 2017. március 7.

Dr. Dabóczi Tamás
egyetemi docens
tanszékvezető



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

László Márk Balogh

Complex Event Processing Based on Automata Theory

MSc Thesis

Supervisors:

András Vörös
Gábor Bergmann PhD
Rebeka Farkas

Budapest, 2017

Contents

Contents	v
Kivonat	viii
Abstract	ix
Hallgatói nyilatkozat	xi
1 Introduction	1
2 Background	3
2.1 Development methods	3
2.1.1 Model driven software development	3
2.1.2 Y model	5
2.1.3 Modeling languages on different levels of abstraction	6
2.1.4 Finite automaton	6
2.2 Verification techniques	7
2.2.1 Design time verification	8
2.2.2 Runtime verification	8
2.3 Complex event processing	9
2.3.1 Complex Event Processing Frameworks	9
2.4 Runtime verification with CEP	9
2.4.1 Data structures supporting parametric decision	10
2.5 Model based tools	10
2.5.1 Eclipse Modeling Framework	11
2.5.2 VIATRA	11
2.5.3 VIATRA-CEP	12
2.6 Related work	12
3 VIATRA-CEP	15
3.1 Live model integration	15

3.2	Event Pattern Language	16
3.3	Contexts	17
3.3.1	Matching and partial matches	17
3.3.2	Noise and noise-reduction	18
3.3.3	Overview of the event contexts	18
3.4	Overview of the architecture	18
3.5	Prefiltering functionalities	19
3.6	Intermediate modeling layer	20
4	Intermediate Language	23
4.1	Overview of Formalisms	24
4.2	Regular Languages	24
4.2.1	Regular Expression	25
4.2.2	Deterministic Finite Event Automaton	25
4.2.3	Nondeterministic Finite Event Automaton	27
4.2.4	Transformation from Regular Expression to Deterministic Finite Automaton	28
4.3	Timed Regular Languages	29
4.3.1	Timeout Regular Expression	31
4.3.2	Timeout Region Automaton	31
4.3.3	Nondeterministic Timeout Event Automaton	33
4.3.4	Transformation from Timed Regular Expressions to Timed Automata	35
4.4	Parametric Timed Languages	36
4.4.1	Parametric Timed Regular Expression	36
4.4.2	Parametric Timed Region Event Automaton	36
4.4.3	Parametric Timed Event Automaton	37
4.4.4	Transformation from Parametric Timed Regular Expression to Parametric Timed Event Automaton	38
5	Algorithms and Implementation	41
5.1	Parametric Timeout Regular Expression	41
5.2	Parametric Timed Event Automaton	41
5.2.1	Finite Automaton	41
5.2.2	Timeout Event Automaton	42
5.2.3	Parameters and Bindings	43
5.3	Transformation from VEPL to the intermediate language	44
5.4	Automaton Executor	44
5.4.1	Parameter Handling	46
5.4.2	Most Naïve approach	46
5.4.3	Naïve approach	47
5.4.4	Decision List	48

5.4.5	Disjunctive Decision Set	48
5.4.6	MDDe	48
5.4.7	Additional Data Structures	49
5.5	Preliminary measurements	49
5.5.1	Measurement setup	49
5.5.2	Measurement Results	50
6	Conclusions and future work	51
6.1	Future Work	51
	Acknowledgments	51
	Appendix	53
A.1	Xtext grammar for the Parametric Timeout Regular Expression	53
	References	57

Kivonat A hagyományos kritikus 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. Többek között ezt a célt szolgálja egy, a tanszéken fejlesztett keretrendszer is, amely élő modelleket használ és az azok fölött értelmezett modelltranszformációs szabályok segítségével definiálja a vizsgálandó temporális kifejezést. Modelltranszformációk helyességének ellenőrzése azonban algoritmikusan eldönthetetlen feladat általánosságban, így a definiált temporális viselkedéseket is nehéz formálisan vizsgálni. Az irodalomban többféle megközelítés létezik a temporális viselkedések definiálására és többféle formalizmust is definiáltak automata alapokon. Ezekben közös, hogy az automata többnyire jól vizsgálhatóak algoritmikusan, az általuk definiált és elfogadott nyelvekkel kapcsolatban lehetőség van formális analízisre és ellenőrzésre.

Ebben a dolgozatban megvizsgálom az irodalomban ismert legfontosabb automataelméleti megközelítéseket, és ez alapján kiterjesztem a tanszéki keretrendszert a paraméteres temporális viselkedések automata elméleti reprezentációjával. Bemutatom egy automata reprezentáció implementációját amely alkalmas paraméteres és temporális tulajdonságok ellenőrzéséhez. Elemzem a megoldást kifejezőerő szempontjából, és megvizsgálom egy mérnöki leíró nyelvet és annak leképezését automata reprezentációra. Ezen felül bemutatom az alapvető koncepcionális megoldását egy paraméteres temporális automata végrehajtónak, és a néhány ehhez szükséges, általam kidolgozott adatszerkezetet.

Kulcsszavak véges állapotú automata, időzített automata, paraméteres időzített automata, Futásidejű verifikáció, komplexesemény-feldolgozás, reguláris kifejezések

Abstract In traditional safety-critical systems, runtime verification is a frequently used method. The goal of runtime verification is to synthesize verification components, which can detect the possible failures of the components. Over the last few years, researchers of my department have started developing a framework called VIATRA-CEP being able to support the runtime analysis and verification of complex systems. This framework uses live models, and defines the temporal expressions by model transformation rules over these live models. Verification of model transformations are generally algorithmically undecidable therefore it is hard to analyze these temporal behaviors. Various approaches exist in the literature supporting the definition of temporal behaviors, and many of them are defined based on automata theory. Automata are usually easier to analyze algorithmically as the accepted languages of these automata can be formally analyzed.

In this thesis various automata based property representations are overviewed, and a possible extension of the VIATRA-CEP framework is outlined. An automata representation implementation for the analysis of parametric and temporal behavior is examined. The solution is analyzed in terms of expressiveness, and a mapping from a high level domain specific language to this automata representation is introduced. An automaton executor implementation is also overviewed, alongside with the data structures which are required for the parametric timed execution.

Keywords finite state machine, timed automata, parametric timed automata, Runtime verification, complex event processing, regular expressions

Hallgatói nyilatkozat

Alulírott **Balogh László Márk** szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2017. december 17.

.....
Balogh László Márk


Chapter 1

Introduction

Assuring the correctness of critical systems is important as their failure might cause huge loss. Nowadays the application of runtime verification is an emerging trend in the safety-critical and cyber-physical systems, to increase safety, as in case of a failure in these systems there is always a large amount of money or human lives at stake.

However, designing and implementing runtime analysis is a complex task. The manual implementation is expensive, the resulting system will be complex, and the faultless operation is not assured. There is a huge need for languages to define the requirements of safety-critical systems. Moreover, there is also a need for certainty in the verification component, and this need makes design time verification and formal model checking desirable.

The motivation of my work is to enhance the available approaches and take one step towards an expressive formalism to support runtime analysis of complex systems. The goal of the paper is to find an intermediate language which can bridge the gap between system level requirements and their runtime verification.



additional introduction

Contributions in this paper:

Chapter 2

Background

System development is a complex process, therefore it needs to be supported by various tools and algorithms. In this chapter the background of our work [25] and my previous BSc thesis [5] is introduced towards an approach supporting correct system design.

Ide meg kell
valami

2.1 Development methods

Traditional software development methods often use informal specifications to support system, architecture and component level designs – which may also be informal. This can easily result in higher verification costs or faulty systems, making them ineffective choices for safety-critical software development. In order to introduce some level of formality and allow manageable, hierarchical software testing procedures, the V model was developed.

Figure 2.1 shows the stages of development, where a symbolic “V” shows the progress of the workflow. The project definition stages on the left side begin with the development of a concept of operations, continue with requirements and architecture, and detailed design. The implementation stage is shown across the base of the “V”. The right side shows the testing and implementation stages of a system, with an upward-pointing arrow for progress of the workflow[27].

The V model is the basic scheme of software development. In the left of the figure, we proceed by decomposing the specification into an architecture level design, and the architectural design into component-level designs. Every level of decomposition has its own specification. After the implementation process, each level of the implementation is verified with respect to the corresponding specification. This verification can be done incrementally, parallel with the implementation process.

2.1.1 Model driven software development

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

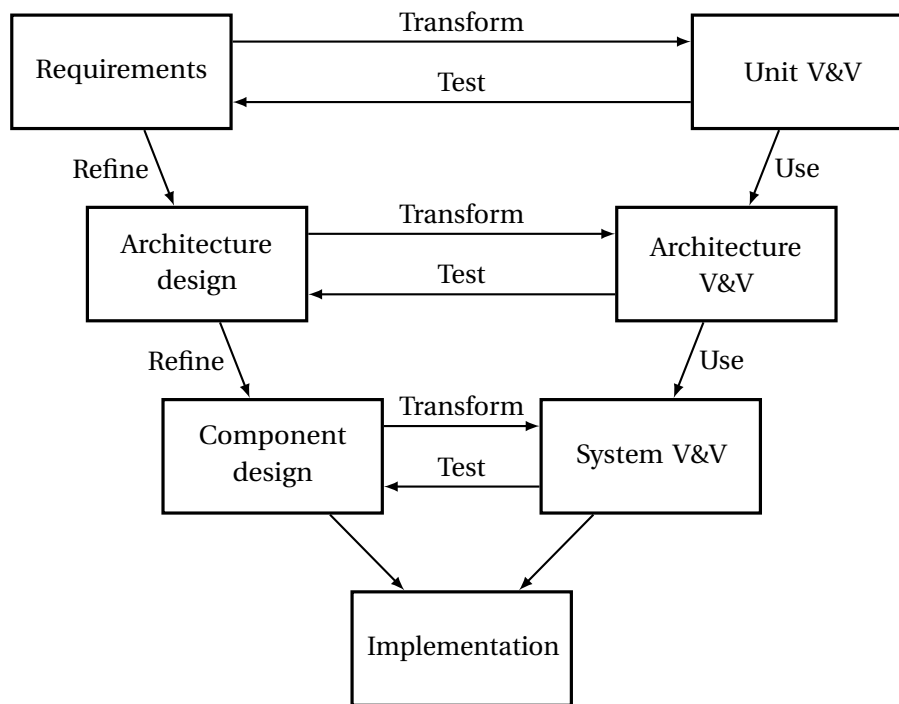


Figure 2.1 The traditional V model[27]

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

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

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

MDSD will usually result in a software life cycle model for component-based systems[8], which is based on the V model.

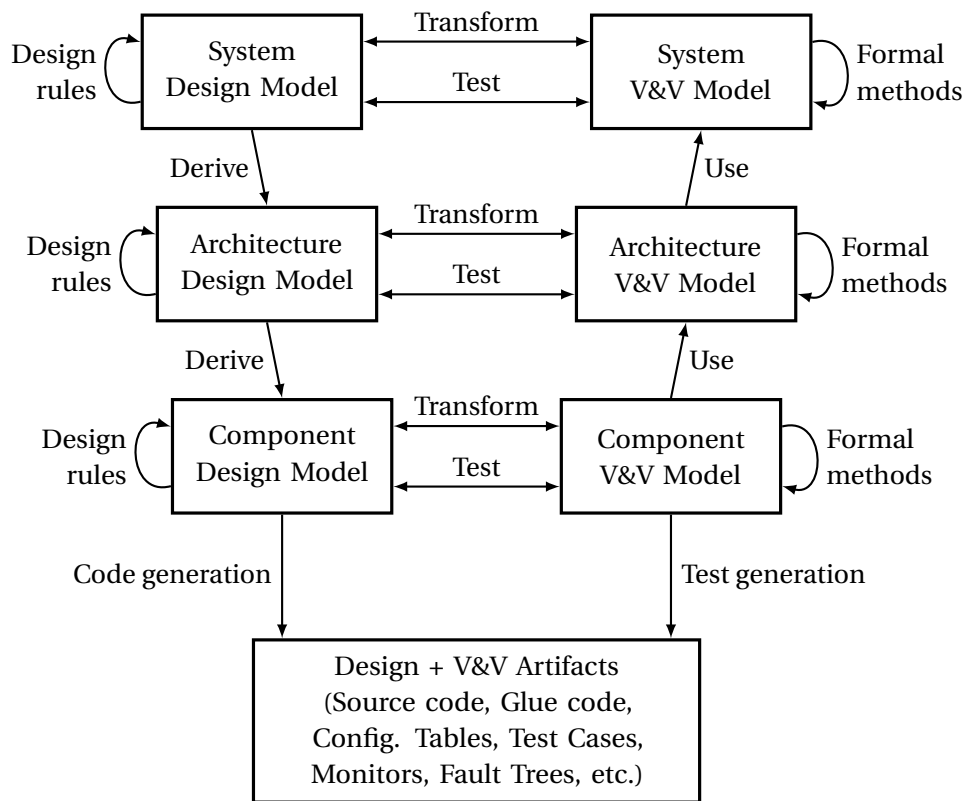


Figure 2.2 An overview of the Y model

2.1.2 Y model

The Y model[8] is an extension of the V model[37], by automating code and test case generation. Much like the V model, the development process is partitioned vertically. Each level has its own modeling languages and models that are transformed to a verification model, on which verification methods such as testing can be applied. The results of the verification process can be traced back to the original models making iterative improvement possible. The top level is for high level system models, while the second level contains architectural models, and the third one is for component-based models. The component model provides input for the source code and configuration generation for the individual components. Test cases are paired with the source code and can be generated from the component verification models.

2.1.3 Modeling languages on different levels of abstraction

Model Driven Software Development methods require modeling languages to describe the behavior of systems and components. Engineering practices produced a wide range of such languages over the years to support efficient product development. This leads to the use of domain-specific languages, which leads to a shorter modeling process. The results is the need for complex model transformations before the verification can begin, but since the domain-specific languages are tailored to the exact problem, their usage is still more efficient – but requires expertise in the domain as well as in the modeling language.

Standardized modeling languages were developed like the UML (Unified Modeling Language[7]) and SysML (Systems Modeling Language[16][42]).

2.1.4 Finite automaton

The modeling of systems with finite state space is often done by using finite automata – also known as finite state machines. A finite automaton accepts a (finite) list of symbols and produces a computation of the automaton for each input list. Although finite automata can be easily visualized, this formalism describes a simple, flat transition system and lacks the support for higher level concepts. The development of finite automata models are supported by many tools (e.g.: Finite State Machine Designer [14]).

Class diagram

In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by depicting the system's classes, their attributes, operations (or methods), and the relationships among objects. It is also often applied at the system level design, as multiple components and their connection can be visualized with Class diagrams.

Message Sequence Chart

The formalism of Message Sequence Charts (MSC) describes the communication between components – the order in which messages can occur[20][18][23]. The message interchange is usually represented by a graphical model. These charts can be used for high-level specification, design, trace-based testing or documentation. A collection of possible sequence charts can also describe a complete communication protocol between components. UML sequence diagrams were inspired by MSCs, but their semantics differs regarding some of the basic elements of the language such as lifelines and arrows[19].

Sequence diagram

A Sequence Diagram is an interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. A Sequence

Diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams are typically associated with use case realizations in the logical view of the system under development. Sequence diagrams are sometimes called event diagrams or event scenarios. Sequence diagrams can be used for system-level modeling, as it can represent the communication between the components.

Statechart

Statecharts, also known as state machines are an extension of finite automata. There are various available syntaxes for statecharts (e.g. the one defined by UML[17]). The higher level concepts that were introduced include variables, actions, and hierarchically nested states. Event-driven execution is also possible by using signals as the triggers of transitions. Available variable types heavily depend on the concrete semantics of the chosen statechart language. Actions can usually be variable assignments, signal raises or the setting of timers. Hierarchy lets users organize system descriptions using a top-down approach. Support for hierarchy is introduced via nested states and parallel regions. States can also have entry and exit actions, which allows the description of common functionality in parent states[34].

Statecharts are usually developed in tools that support the graphical design of the model (e.g.: Yakindu[43], an Eclipse-based editor). Statecharts are commonly used as a form of component description, as it can represent the behavior of components depending on its state.

Activity Diagram

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. In the Unified Modeling Language, activity diagrams are intended to model both computational and organizational processes (i.e. workflows). Activity diagrams show the overall flow of control, and therefore can be used at the system level description.

2.2 Verification techniques

As modern society is becoming more and more dependent on safety-critical systems, the need for faultlessly working hardware and software increases. The development of safety-critical systems require extensive testing efforts. Validation and verification methodologies have been present in the development processes of such systems for a long time[41], but faster and more reliable approaches are needed. Validation assures that the requirements specified for the software meet the needs of the user – as such, validation usually can be

aided, but can't entirely be done by software. The goal of verification is to analyze whether the specified requirements are met by the system. Methods for verification can be divided into two groups: design time verification and runtime verification. These approaches aren't exclusive, and their mutual use can support a more robust verification process.

2.2.1 Design time verification

Design time verification is a method used for finding errors of the system before deployment. Traditional software development methodologies usually rely on design time approaches. Verification methods can be applied on multiple levels, from small parts to the whole system. These processes check the compliance of the system against the specification of the appropriate level.

Formal verification can also be used to give proof that the verified parts match the behavior described by the formal specification. On the other hand, applying formal methods usually have higher costs, and the verification of complex systems can be impossible due to the phenomenon of state space explosion. As a result, a possible application of formal methods is the verification of the correct behavior of system components, and not the entire system.

2.2.2 Runtime verification

Runtime verification is a method for the inspection of running systems. The motivation of the approach is to reduce the complexity of design time verification. As systems are getting larger, the application of formal methods are more and more limited as the resources needed for verification cannot be provided. This means that formal verification methods must verify an abstract model, not the deployed system itself. In addition, specifications are rarely complete, and design time methods can rarely handle hardware errors.

Runtime verification uses monitors to observe certain (usually critical) components, checking whether their operation violates properties described in the specification. The usage of monitoring components can result in significantly smaller monitors than the component itself as multiple levels of abstraction can be used, as long as the error states remain distinguishable. This has the advantage of detecting the erroneous operation of the system and allows systematic safety engineering to handle faults, or trigger an emergency shutdown if necessary.

The main advantages of runtime verification are the following:

- Smaller computational complexity.
- Verification of the running implementation.
- Detection of previously not defined errors.

2.3 Complex event processing

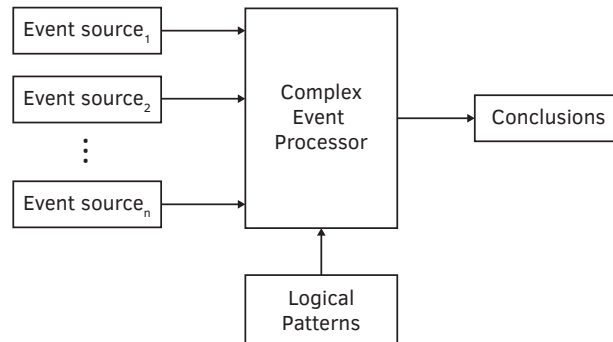


Figure 2.3 Complex event processing overview

Complex event processing (or CEP for short) is a method of tracking and analyzing streams of information and deriving conclusions. In a complex event processing environment, there can be multiple event sources, and with logical patterns given by a formalism, patterns (pattern matches) can be found in the incoming stream, e.g. events followed by another events in some predefined sequence.

2.3.1 Complex Event Processing Frameworks

There are many Complex Event Processing Frameworks in the industry.

One of these tools is Drools[31] which is the leading Java based open-source rule engine. It is a hybrid chaining engine meaning that it can react to changes in data and also provides advanced query capabilities. Drools provides built-in temporal reasoning for complex event processing and is fully integrated with the jBPM project for BPMN2 based workflows.

Another well-known tool in the industry is Esper[13] which is an Event Stream Processing (ESP) and event correlation engine. Targeted to real-time Event Driven Architectures (EDA), Esper is capable of triggering custom actions written as Plain Old Java Objects (POJO) when event conditions occur among event streams. It is designed for high-volume event correlation where millions of events coming in seconds. This amount of input would make it impossible to store all event in a classical database architecture and query them later.

2.4 Runtime verification with CEP

In this section the application of CEP for runtime verification is overviewed. In general, system level runtime verification can be provided with the help of complex event process-

ing. A complex event processing framework can monitor either the messaging between components and also the current state of the system and detect errors based on the specification of valid message sequences and valid states. The approach uses pattern matching to detect faulty operation of the system. Should illegal sequences occur, error handling operations (e.g. restart or shutdown of a component or the whole system) can be applied to avoid the violation of safety criteria.

2.4.1 Data structures supporting parametric decision

Most of the Complex Event Processing tools support the handling of parameters, such as parametric events. To support such functionalities, some data structure is required which can decide which parameters are in them and which of them are not.

BDD and MDD

Many problems can be stated naturally using variables that have multiple values (i.e., take their values from a discrete domain). Functions defined on these variables can also take on values from a discrete set. Examples of such problems range from combinatorial optimization such as routing and resource scheduling, to logic simulation and formal verification, and to logic synthesis such as state minimization and state assignment. In many cases these problems are NP-complete or coNP-complete. Compact representation and efficient manipulation of such multi-valued functions are key to the design of efficient algorithms that advance the frontier of the problems that can be solved exactly. Binary decision diagrams (BDDs) are such a compact representation for problems involving binary variables. In this paper, we define the multi-valued decision diagram (MDD), which is a canonical representation of a multi-valued function as a directed acyclic graph. We analyze its properties and provide algorithms for constructing and manipulating MDDs. With our MDD package, an MDD is mapped into a BDD using either a logarithmic encoding or a 1-hot encoding, each suitable for a different class of applications. We have applied both kinds of MDD to many different applications, and this paper serves as a summary of the work done so far. Furthermore, general problem solving techniques, such as binate table covering and other graph algorithms, have been formulated using MDDs.[22]

2.5 Model based tools

In this section I introduce the basic model-based tools and technologies I used in my work. The motivation of using models, and model-based technologies are:

- A model is a high-level representation of a software component, allowing us to use a high level of abstraction in the implementation and the documentation as well.

- The models can be formal, or can be transformed into a formal model, and formal models can be formally verified.

2.5.1 Eclipse Modeling Framework

The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. EMF (core) is a common standard for data models, many technologies and frameworks are based on. This includes server solutions, persistence frameworks, UI frameworks and support for transformations.

EMF consists of three fundamental pieces:

- **EMF** – The core EMF framework includes a metamodel (Ecore) for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically.
- **EMF.Edit** – The EMF.Edit framework includes generic reusable classes for building editors for EMF models.
- **EMF.Codegen** – The EMF code generation facility is capable of generating everything needed to build a complete editor for an EMF model. It includes a GUI from which generation options can be specified, and generators can be invoked. The generation facility leverages the JDT (Java Development Tooling) component of Eclipse[11].

EMF metamodels (or EMF class diagrams) are similar to UML class diagrams, a node represents a classifier, and the edges are associations or containments, depending on their endings. Also note, that EMF class diagrams are manipulating an EMF model and they use the genmodel package to generate Java code but UML class diagram represents directly a Java model in UML.

2.5.2 VIATRA

The VIATRA framework supports the development of model transformations with specific focus on event-driven, reactive transformations and offers a language to define transformations and a reactive transformation engine to execute certain transformations upon changes in the underlying model. Furthermore, the underlying incremental query engine, originating from the EMF-IncQuery project is reusable in different scenarios not related to model transformations.[38]

VIATRA-Query is a query language in the VIATRA framework for defining declarative graph queries over EMF models, and executing them efficiently without manual coding in an imperative programming language such as Java[40].

Graph patterns are an expressive formalism used for various purposes in Model Driven Development, such as the definition of declarative model transformation rules, defining the behavioral semantics of dynamic domain-specific languages, or capturing general purpose model queries including model validation constraints. A graph pattern (GP) represents conditions (or constraints) that have to be fulfilled by a part of the instance model. A basic graph pattern consists of structural constraints prescribing the existence of nodes and edges of a given type. Languages usually include a way to express attribute constraints. A negative application condition (NAC) defines cases when the original pattern is not valid (even if all other constraints are met), in form of a negative sub-pattern. With NACs nested in arbitrary depth, the expressive power of graph patterns is equivalent to first-order logic. A match of a graph pattern is a group of model elements that have the same configuration as the pattern, satisfying all the constraints (except for NACs, which must be made unsatisfiable)[6].

2.5.3 VIATRA-CEP

CEP plays an important role in model-driven engineering (MDE) as a supporting technique in various scenarios. The VIATRA project delivers a state-of-the-art event processing framework for the MDE scene, called VIATRA-CEP[39].

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

I will introduce more details about VIATRA-CEP in Chapter 3.

New related work?

2.6 Related work

Runtime verification has a long-standing history and it is applied in various domains. Starting from analyzing the runs of the processors by special constructs called watchdog processor up to the system level verification of whole hardware–software ecosystems with the help of CEP. Various languages were designed to specify the requirements. Linear temporal logic (LTL) is a popular formalism for the specification and verification of concurrent and reactive systems[30]. Most approaches that use LTL adopt a discrete model of time, where a run of a system produces a sequence of observations. Such a model is inadequate for real-time systems, where a run of a system is modelled either as a sequence of events that are time-stamped with reals or as a trajectory with domain the set \mathbb{R}_+ of non-negative reals[26].

Current approaches to monitoring real-time properties suffer either from unbounded space requirements or lack of expressiveness. In paper [21] the authors adapted a separation

technique to enable the rewriting of arbitrary MTL formulas into LTL formulas over a set of atoms comprising bounded MTL formulas. As a result, they obtain the first trace-length independent online monitoring procedure for full MTL in a dense-time setting.

Web service applications are distributed processes that are composed of dynamically bounded services. In paper [35] the authors give a definitive description of a framework for performing runtime monitoring of web service applications against behavioral correctness properties described as finite state automata. These properties specify forbidden and desired interactions between service partners. Finite execution traces of web service applications described in BPEL are checked for conformance at runtime. When violations are discovered, our framework automatically proposes adaptation strategies, in the form of plans, which users can select for execution. Our framework also allows verification of stated pre- and post-conditions of service partners and provides guarantees of correctness of the generated recovery plans.

In ultra-critical systems, even if the software is fault-free, because of the inherent unreliability of commodity hardware and the adversity of operational environments, processing units (and their hosted software) are replicated, and fault-tolerant algorithms are used to compare the outputs. The authors of [29] investigate both software monitoring in distributed fault-tolerant systems, as well as implementing fault-tolerance mechanisms using RV techniques. They describe the Copilot language and compiler, specifically designed for generating monitors for distributed, hard real-time systems, and they describe a case study in a Byzantine fault-tolerant airspeed sensor system.

Paper [33] presents a method for runtime verification of microcontroller binary code based on past time linear temporal logic (ptLTL). The authors show how to implement a framework that, owing to a dedicated hardware unit, does not require code instrumentation, thus, allowing the program under scrutiny to remain unchanged. Furthermore, they demonstrate techniques for synthesizing the hardware and software units required to monitor the validity of ptLTL specifications.

PSL is a property specification language recently standardized as IEEE 1850TM-2005 PSL. It includes as its temporal layer a linear temporal logic that enhances LTL with regular expressions and other useful features. PSL and its precursor, Sugar, have been used by the IBM Haifa Research Laboratory for formal verification of hardware since 1993, and for informal (dynamic, simulation runtime) verification of hardware since 1997. More recently both Sugar and PSL have been used for formal, dynamic, and runtime verification of software. In paper [12] the authors introduced PSL and briefly touch on theoretical and practical issues in the use of PSL for dynamic and runtime verification.

A goal of runtime software-fault monitoring is to observe software behavior to determine whether it complies with its intended behavior. Monitoring allows one to analyze and recover from detected faults, providing additional defense against catastrophic failure. Although runtime monitoring has been in use for over 40 years, there is renewed interest in its application to fault detection and recovery, largely because of the increasing complexity and ubiquitous nature of software systems.[10]

Chapter 3

VIATRA-CEP

VIATRA-CEP[39][9] is the novel Complex Event Processing Framework of the open source VIATRA Project. In this chapter I overview the tool and the underlying technologies and approaches.

3.1 Live model integration

VIATRA-CEP is built on the top of a live model, in the following novel way: the user can define graph patterns on the model with VIATRA-Query, and the framework automatically generates events on every appearance and/or disappearance of these patterns. The generated events can be used in event patterns, allowing the users to express complex temporal statements of these patterns. A simplified outline of this architecture is shown in Figure 3.1

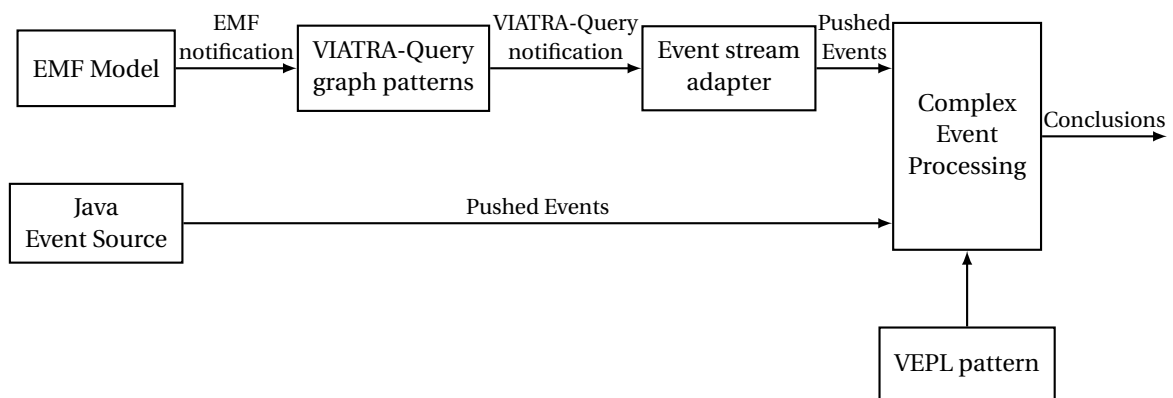


Figure 3.1 A brief outline of the live model integration in VIATRA-CEP

Table 3.1 Basic operators

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

3.2 Event Pattern Language

VIATRA-CEP uses a language called VIATRA Event Pattern Language (VEPL for short). This language has intuitive syntax and the event patterns can be defined in a very high level.

Operators and semantics VEPL basic operators are shown in Table 3.1. There are additional operators, which can be considered syntactic sugars, shown in Table 3.2, but these operators can be expressed with the basic operators as shown in Table 3.3, according to [39].

The semantics of the operators “OR” and “AND” are the following:

- operator “OR” has a “committed or” semantic. This means if that one operand of the “OR” is not an atomic event, and that part starts a partial match, the other part of the “OR” is ignored while this part is active.
E.g. if the pattern is $(A \rightarrow B) \text{ OR } C$, then trace A, C will not match the pattern.¹
- “AND” is a binary operator, e.g. $A \text{ AND } B \text{ AND } C$ is equivalent to $(A \text{ AND } B) \text{ AND } C$, and it will not accept the trace ACB

¹Meanwhile VEPL patterns are similar to patterns in regular expressions, they have different semantics. In case of a regular expression, pattern $(AB)|C$, and the AC trace would not cause a match. But since VEPL is used in the field of complex event processing, where an arbitrary prefix for all patterns is set by default, the equivalent of $(A \rightarrow B) \text{ OR } C$ is $\Sigma^*(AB)|C$

Table 3.2 Syntactic sugars

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

Table 3.3 Syntactic sugars mapped to basic operators

Operator name	Denotation	Equivalent
and	$p_1 \text{ AND } p_2$	$((p_1 \rightarrow p_2) \text{ OR } (p_2 \rightarrow p_1))$.
negation	$\text{NOT } p$	$\Sigma \setminus p$, where Σ is the set of all the possible events. TODO: And complex NEG?
multiplicity	$p\{n\}$	$p \rightarrow p \rightarrow \dots p$, n times.
“at least once” multiplicity	$p\{+\}$	$p \rightarrow p\{*\}$

3.3 Contexts

To define properties of execution of the pattern matching, the user can set which event context will be used for the execution. These event contexts set the properties execution as of noise-reduction, and the maximal amount of partial matches at the same time.

3.3.1 Matching and partial matches

When a pattern is not matched yet, but the current trace is a possible prefix of the pattern, the trace will cause a partial match. For example, if the pattern is $A \rightarrow B \rightarrow C$, and the trace is $A B$ then the pattern matcher only waits for an event C . The event context defines the maximal amount of these partial matches. For example, if there are multiple matches allowed at the same time, and the pattern is $A \rightarrow B$, the trace $A A B B$ will cause two matches, one at the reception of the first event of type B , and one after the reception of the second event of type B . If only one partial match would be used at the same time, the occurrence first event of type B would cause a match, but the second occurrence would not.

Table 3.4 Event Contexts in VIATRA-CEP

Context	Noise-reduction	Partial matches at the same time
Strict Immediate	–	Single
Immediate	–	Multiple
Chronicle	✓	Multiple

3.3.2 Noise and noise-reduction

In most cases, where a Complex Event Processor is used, there are multiple event sources, with vast amount of event types. Even in these cases, there are many pattern, which does not use all of these event types. In VIATRA-CEP, the noise is defined as following: An incoming event, which will not increase the size of the currently non-contradictory postfix of the trace. For example, if the pattern is $A \rightarrow B \rightarrow C$, and the previous trace is $A B$, an event B is considered noise. Note that in this example, an event A would not be noise, as it will either start a new partial match, if it is allowed, or restart the matching depending on the chosen context.

3.3.3 Overview of the event contexts

All three different event contexts introduced in [9] are shown in Table 3.4. These contexts can be defined separately for each event pattern, and they will define the execution semantics of the generated automata. An example of these contexts are shown on Table 3.5. In this example the first three rows represent an example of the noise-reduction property, as the Chronicle Context ignores the events in the trace, which are not in the current pattern. The last three rows represent an example of the multiple partial matches, as the Immediate and the Chronicle Event Contexts have two matches, while the Strict Immediate has only one. The accepted parts of the traces are under- and overlined from their start to their end.

3.4 Overview of the architecture

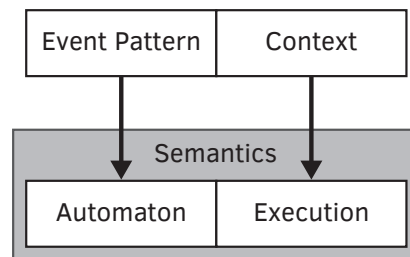
In this section a short overview of the inside of the VIATRA-CEP is given.

The user first optionally defines the graph patterns on an EMF model, or simply defines atomic events. Using these two, the user builds up complex event patterns, and sets the context for each event pattern. The event patterns are compiled to an automaton whose execution semantics is defined by the context, as shown on Figure 3.2. Note that the overall semantics is defined by both the automaton generated from the event pattern, and the context. If the goal is to create an extensible framework, one of the possible approaches is to use a general intermediate language. In this case additional languages only require a mapping to the intermediate language. A well defined formal intermediate representation

Table 3.5 Examples of the Event Contexts in VIATRA-CEP

VEPL pattern	Context	Trace	Accepted
$A \rightarrow B$	Strict Immediate	$A C B$	–
	Immediate	$A C B$	–
	Chronicle	<u>$A C B$</u>	✓
	Strict Immediate	$A \underline{A B} B$	✓
	Immediate	$\overline{A \underline{A B} B}$	$2 \times \checkmark$
	Chronicle	$\overline{A \underline{A B} B}$	$2 \times \checkmark$

supports the formal verification of the CEP specification, and the semantics is desirable to be defined exclusively by the automaton.

**Figure 3.2** The architectural overview of the VIATRA-CEP

3.5 Prefiltering functionalities

In the new (0.13) version of the VIATRA-CEP the traits and the check expression has been introduced to help the user automatize some of the prefiltering functions.

These functionalities are be described here, however these are not in the scope of this thesis, as their implementation is not strictly related to the complex event processing.

Traits Traits provide abstraction mechanism to Atomic event patterns. Currently, it supports abstraction of event parameters. For example if the user want to handle all events with coordinateX and coordinateY fields the user can create an abstraction with traits as shown on Listing 3.1.

Listing 3.1 Example of the traits in VIATRA-CEP

```

1    trait position {
2        coordinateX : double = 0.0, //definition with default value
3        coordinateY : double
4    }

```

Check expressions in atomic events Check expressions are helpful to filter the incoming events without unnecessary code-noise in the patterns. For example if the user wishes to get only the events with positive coordinates a check expression can be applied as shown on Listing 3.2.

Listing 3.2 Example of the Check expression in VIATRA-CEP

```

1    atomicEvent object with position {
2        check {
3            coordinateX > 0 && coordinateY > 0
4        }
5    }

```

3.6 Intermediate modeling layer

Our proposal is to use an intermediate language between the event pattern language and the automata representation, and to use this intermediate language as a common representation of the high level specification, as shown on Figure 3.3. In addition, the long term goal is to support formal analysis of the defined properties.

Using an intermediate language increases the extensibility of the framework, as additional pattern languages can be added later, by implementing a compiler from the high level language to the intermediate language, as shown on Figure 3.4. The intermediate language would allow the user to combine patterns defined in multiple languages, as multiple automata could run and their simultaneous runs could be analyzed in a formal way.

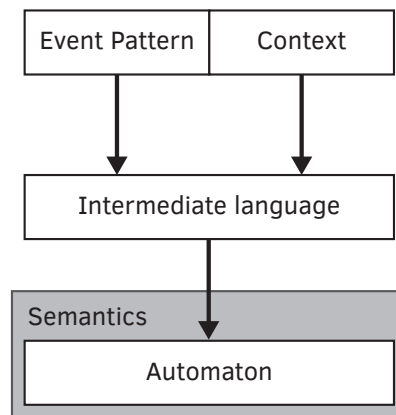


Figure 3.3 The architectural overview of our proposed architecture

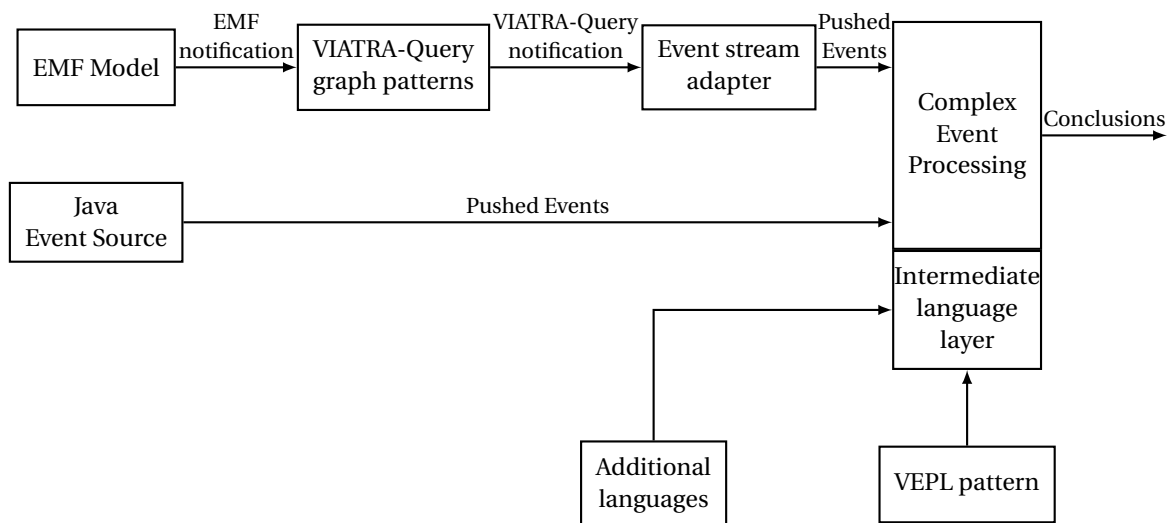


Figure 3.4 A brief outline of the live model integration in VIATRA-CEP, with the intermediate language

Chapter 4

Intermediate Language

Regular languages are widely used in computer science as they can be represented efficiently by finite state machines. In our research according to [9] I found out, that using regular expressions with timing extensions and parameters we are able to express the temporal patterns of VIATRA-CEP. Since regular expressions are widely used by the engineers and finite state automata provide clear semantics they are a good candidate as an intermediate language.

First we overview formalisms from the literature, and suggest parametric timed region automaton as an intermediate formalism. As many of the high-level specification languages like VEPL and Sequence charts can be characterized with the help of regular languages, this was the motivation behind our research.

We are going to introduce every step through an example: We will specify a runtime verification component (also known as monitor) which will be in accepting state if the system is violating the rules of the two-phase lock algorithm extended with a timeout. This can be considered as a runtime verification component of such system. This example is based on the File System example of paper [32].

In the two-phase locking algorithm there are three rules to be held :

1. Resources must be allocated in a previously defined sequence (this sequence is the same for all tasks).
2. If a task releases a resource, it is not allowed to do any more allocations.
3. The first phase must finish after 10 seconds, i.e. the time between the first allocation and the first release must be less than 10 seconds¹.

¹We could use a better example such as “A resource can not be in allocated state for more than 10 sec”, but this would make the example really complex.

Check the chapter for mixing Timeout and timed!

Since this example uses resources, their behavior is defined as following: Each resource can be allocated once before every release and can be released once before every allocation.

4.1 Overview of Formalisms

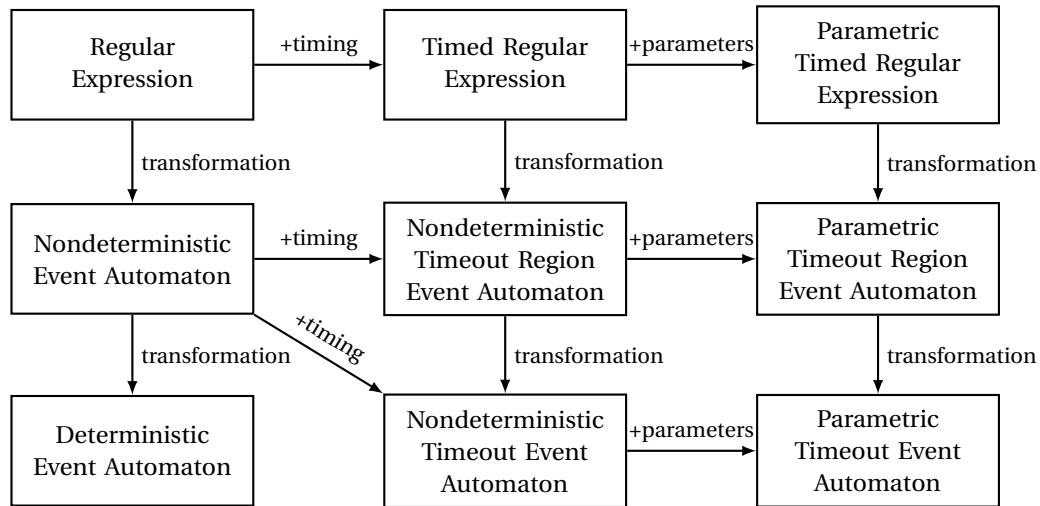


Figure 4.1 Overview of the formalisms

The used formalisms and their relations are shown on Figure 4.1. The vertical arrows show that for each regular expression an automaton can be constructed with a transformation, which accepts the language specified with the regular expression. The horizontal arrows show that each formalism can be extended with the required properties; timing and parameters respectively. Note that even the Deterministic Event Automaton could be extended with Timing, and a formalism could be constructed for it, however there are no known algorithms to construct a Deterministic Timed Automaton.

4.2 Regular Languages

Regular Languages are commonly used both in the industry and in the academic research as they commonly serve as a domain specific language. Most of their properties which are interesting from the academic point of view can be analyzed and decided, and they are simple enough to be used in industrial projects to ease up the description of something which does not require a more complex language.

4.2.1 Regular Expression

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

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

1. a for every letter $a \in \Sigma$ and the special symbol ϵ are expressions.
2. If $\varphi, \varphi_1, \varphi_2$ are Σ -expressions then $\varphi_1\varphi_2, \varphi_1|\varphi_2, \varphi^*$ are Σ -expressions[3].

The meaning of these operators:

- $\varphi_1\varphi_2$: Sequence, φ_2 must occur after φ_1 .
- $\varphi_1|\varphi_2$: Choice, φ_1 or φ_2 must happen.
- φ^* : Closure (also known as Kleene-star), φ can occur n times, where $0 \leq n < \infty$

To illustrate the usage of the regular expressions in the two-phase locking example, the behavior “After a task has released a resource, it’s not allowed to allocate again” is described with the regular expression : $a(a)^*r(r)^*a$, where a is short for allocation and r is short for release.

4.2.2 Deterministic Finite Event Automaton

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

Cite?

Definition 4.2 A Deterministic Finite Event Automaton is a tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ [24] where:

- Q is a finite, nonempty set, representing the states of the automaton,
- Σ is a finite, nonempty set, representing the event set of the automaton,
- δ is a subset of tuples $\langle Q \times \Sigma \times Q \rangle$, and the number of outgoing edges from each state for each event is only one i.e. $\forall s_0 \in Q$ and $\forall e_0 \in \Sigma : |\langle s_0, e_0, s_1 \rangle| = 1$, where $s_1 \in Q$. Nondeterminism is not allowed,
- $q_0 \in Q$ the initial state,
- $F \subseteq Q$ the set of the accepting states.

Definition 4.3 An input sequence for a Deterministic Finite Automaton is (e_1, e_2, \dots, e_n) , where $\forall i : e_i \in \Sigma$, i.e. a sequence of events.

Definition 4.4 A trace in an Deterministic Event Automaton is a sequence of states (r_0, r_1, \dots, r_m) , where $\forall i : r_i \in Q$, for the input sequence (e_1, e_2, \dots, e_n) if:

- $r_0 = q_0$, i.e. the first state of the trace is the initial state of the Automaton,
- If the automaton has a trace (r_0, \dots, r_{i-1}) for input sequence (e_1, \dots, e_{j-1}) then either
 - $\langle r_{i-1}, \varepsilon, r_i \rangle \in \delta$, or
 - $\langle r_{i-1}, e_j, r_i \rangle \in \delta$ is true,
- the last r_m state is reached by reading event e_n (Or more precisely the after reading the last event r_m is reachable using only ε -transitions)

A trace is accepting if $r_n \in F$.

Definition 4.5 An event automaton accepts an input sequence if exists an accepting trace for the input sequence.

Definition 4.6 A token in a Nondeterministic Finite Event Automaton a set of the active states.

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

The semantics is illustrated as follows.

- At initialization the token is at the initial state f_0 .
- When receiving input e , where $e \in \Sigma$, if the token is on state s the next state will be s' where $\delta\langle s, e, s' \rangle$.
- If the token enters state s' where $s' \in F$ then the input sequence is accepted.

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

Note that the automaton only accepts the incorrect traces as our complex event processing framework allows to define reactions when the automaton enters an acceptor state.

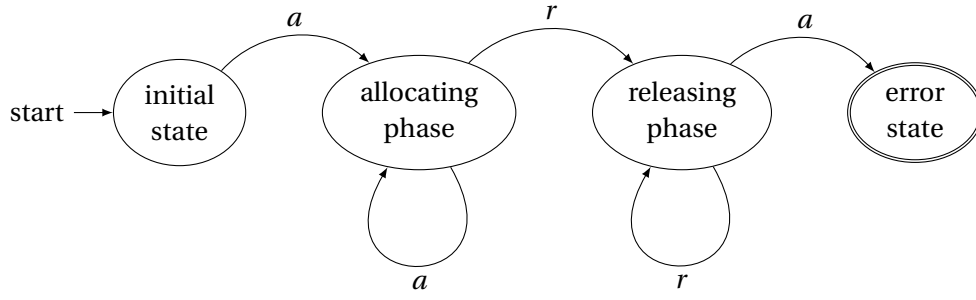


Figure 4.2 Event automaton of the two-phase locking example

4.2.3 Nondeterministic Finite Event Automaton

Definition 4.7 A Nondeterministic Finite Event Automaton is a tuple $\langle Q, \Sigma', \delta, q_0, F \rangle$, where:

- Q, q_0 , and F are the same as in Definition 4.2,
- Σ' is $\Sigma \cup \varepsilon$, where Σ is the same as in Definition 4.2
- δ is a subset of tuples $\langle Q \times \Sigma' \times Q \rangle$. Note that there could be more than one transition for one state with the same event.

Definition 4.8 A trace in an Deterministic Event Automaton is a sequence of states (r_0, r_1, \dots, r_n) , where $\forall i : r_i \in Q$, for the input sequence (e_1, e_2, \dots, e_n) if:

- $r_0 = q_0$, i.e. the first state of the trace is the initial state of the Automaton,
- $\forall i \in (1, \dots, n-1) : \langle r_i, e_{i+1}, r_{i+1} \rangle \in \delta$ i.e. exists a transition between state r_i and r_{i+1} with the label of the e_{i+1} according event.
- The last r_n is reached after reading the entire input sequence.

A trace is accepting if $r_n \in F$.

Definition 4.9 An event automaton accepts an input sequence if exists an accepting trace for the input sequence.

Definition 4.10 A token in a Deterministic Finite Event Automaton is the active state.

4.2.4 Transformation from Regular Expression to Deterministic Finite Automaton

The transformation of simple (not timed and parametric) regular expressions are well known in the current literature. The various algorithms were investigated and a chosen algorithm from [24] is implemented.

Transformation from Regular Expression to Nondeterministic Automaton with ϵ transitions

First, the regular expression is transformed to a nondeterministic automaton with recursive rules for each operator

- The automaton corresponds to a simple event is constructed as an initial state and an acceptor state with one transition between them, which has the label of the event, as shown on Figure 4.3. Note that this is the only non-recursive rule, as the concrete syntax tree's leaves are always Events, therefore this is where the recursion always stops.
- The sequence of two expressions are compiled in two steps, first the two subexpressions are compiled into one automaton each. After that we merge the acceptor states of the first automaton with the initial state of the second automaton, i.e. remove the acceptor flag from the first automaton's acceptor states, and add every outgoing transition of the second automaton's initial state to the first automaton's acceptor states, as shown on Figure 4.4
- A choice can be compiled in two steps. First the two subexpressions are compiled into one automaton each. In the second step we create one new initial state, and create a two ϵ -transition from the new initial state to the two subexpression's compiled automata's initial state. After that we create one ϵ -transition from their acceptor states to a newly created acceptor state. The result's initial state will be the newly created state and the result's only acceptor state will be the newly created one, as shown on Figure 4.5
- A closure can also be compiled in two steps, first the internal expression has to be compiled to the automaton and a new initial and acceptor state has to be constructed. Then four ϵ transition needs to be added:
 1. From the compiled automaton's acceptor state to its initial state,
 2. From the new initial state to the subexpression's initial state,
 3. From the subexpression's acceptor state to the new acceptor state,
 4. From the new initial state to the new acceptor state.

There is a better terminology for "algorithm stops" :D

Missing the AND, probably just don't tell the reader about it?

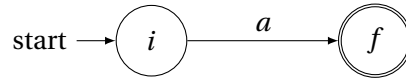


Figure 4.3 Constructed NFA from the simple event a

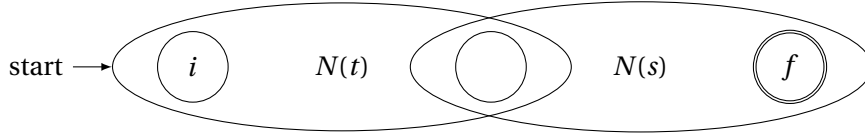


Figure 4.4 Constructed NFA from the sequence $N(s)N(t)$

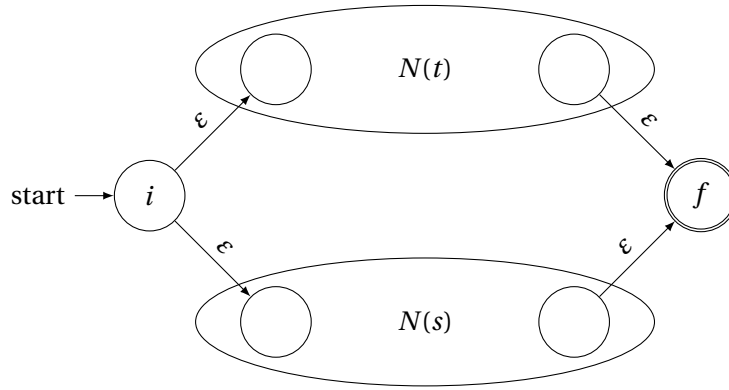


Figure 4.5 Constructed NFA from the choice $N(s)|N(t)$

Transformation from Nondeterministic Automaton to Deterministic Automaton

The algorithm is introduced on Algorithm 4.1. The algorithm constructs a transition table $Dtran$ for D . Each state of D is a set of NFA states, and we construct $Dtran$ so D will simulate “in parallel” all possible moves N can make on a given input trace. Our first problem is to deal with ϵ -transitions of N properly.

$Move(T, a)$ is the set of states of the NFA which there is a transition on input symbol a from some state in T .

The algorithm for calculating the ϵ -closure is shown on Algorithm 4.2.

4.3 Timed Regular Languages

Timed Regular Languages are an extension over the Regular Languages to express timed properties. These timed properties are common in most of the use cases as many system has timed behavior which can be described using Timed Regular Languages.

Algorithm 4.1 The subset construction of a DFA from an NFA

```

input :An NFA  $N$ 
output:A DFA  $D$  accepting the same language as  $N$ 
1 initially,  $\varepsilon$ -closure( $s_0$ ) is the only state in  $Dstates$ , and it is unmarked
2 while there is an unmarked state  $T$  in  $Dstates$  do
3   mark  $T$ 
4   foreach input symbol  $a$  do
5      $U = \varepsilon$ -closure(move( $T, a$ ))
6     if  $U$  is not in  $Dstates$  then
7       Add  $U$  as an unmarked state to  $Dstates$ 
8     end
9      $Dtran[T, a] = U$ 
10  end
11 end

```

Algorithm 4.2 Computing ε -closure(T)

```

input :A set of states  $T$ 
output:A set of states  $\varepsilon$ -closure( $T$ )
1 push all states of  $T$  onto  $stack$ 
2 initialize  $\varepsilon$ -closure( $T$ ) to  $T$ 
3 while  $stack$  is not empty do
4   pop  $t$  from the  $stack$ 
5   foreach state  $u$  with an edge from  $t$  to  $u$  labeled  $\varepsilon$  do
6     if  $u$  is not in  $\varepsilon$ -closure( $T$ ) then
7       Add  $u$  to  $\varepsilon$ -closure( $T$ )
8       push  $u$  onto  $stack$ 
9     end
10  end
11 end

```

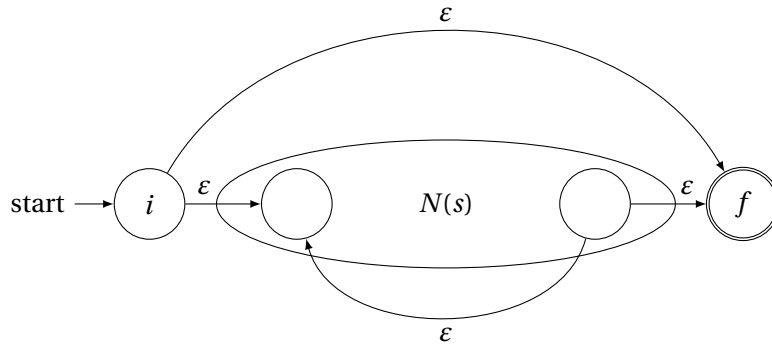


Figure 4.6 Constructed NFA from the closure $N(s)^*$

4.3.1 Timeout Regular Expression

Using the previously defined semantics timed properties can not be expressed, but regular expressions can be extended to a formalism which can do so: the timeout regular expressions.

Definition 4.11 Timeout Regular Expressions over an alphabet Σ (also referred to as Σ -expressions) are defined using the following rules.

1. a for every letter $a \in \Sigma$ and the special symbol ε are expressions.
2. If $\varphi, \varphi_1, \varphi_2$ are Σ -expressions then $\varphi_1 \varphi_2, \varphi_1 | \varphi_2, \varphi^*$, and $\langle \varphi \rangle_I$, are Σ -expressions, where I is a positive real number for timeout.

The new operator is $\langle \varphi \rangle_I$ which means, that the whole φ has to be observed within the given time. The implementation and algorithmization with intervals instead of timeouts as in [3] are considered future work.

With the timed regular expression formalism the example can be extended with timing, and adding the concept timeout becomes possible. The new expression will be: $\langle a(a)^* \rangle_{t>10} r(r)^* a$

4.3.2 Timeout Region Automaton

Timeout Region Automaton is introduced as an intermediate step between the Timeout Regular Expression and the Timeout Automaton as it is a step closer to the regular expression. However the Timeout Region Automaton has the same expressive power as the Timeout Regular Expression the more general and better known Timeout Automaton is more expressive. If in the future for some use cases the Regular Expression lacks the re-

quired expressiveness then the Timeout Automata can be used as an intermediate language as well.

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

From now on, the notation Reg will be used for regions, which is a set of states, i.e. $Reg \subseteq Q$

Definition 4.12 A Timeout Region Event Automaton is a tuple $\langle Q, \Sigma, \delta, q_0, F, t, T \rangle$ where

- $Q, q_0,$ and F are the same as in Definition 4.7,
- Σ is the same as Σ' in Definition 4.7,
- T is a set of tuples $\langle Reg, \mathbb{R} \rangle$ i.e. a set of timeout clock variables for a set of states,
- and δ is the union of discrete and timed transitions i.e. $\delta_t \cup \delta_d$ where
 - δ_d is the same as δ in Definition 4.7,
 - and δ_t represents timed transitions and defined as the set of tuples $\langle Reg, \mathbb{R}, Q \rangle$, i.e. a tuple which represents a timed region, the value of the maximum time elapsed

The semantics of the timed region automaton is defined as follows: $Q_t \subseteq Q$ is the set of states with outgoing timed transitions, i.e. $\forall q \in Q_t : q \in R$.

Let us use the following notations:

- s is the currently active state, and s' is the next state according to δ .
- r is the set of currently active regions, i.e. $r \subseteq Reg$ where $\exists r_s : r_s \in r, s \in r_s$
- r' is the set of regions the token enters, i.e. $r' \subseteq Reg$ where $\exists r_s : r_s \in r, s' \in r_s$
- $r^+ \subseteq Reg$ is a set of timed regions the token has just entered, i.e. $r^+ = r' \setminus r$
- $r^- \subseteq Reg$ is a set of timed regions the token has just left i.e. $r^- = r \setminus r'$

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

1. Initialization Rule: At the initialization of the automaton, we have to set all clock variables to ∞ i.e. $\forall t_i, \forall q_i, T \langle r_i, t_i \rangle, t_i := \infty$, where $r_i \in R$
2. Entering new timed region rule : If a token enters a new set of timed regions, i.e. $r^+ \neq \emptyset$, the timers are set according to the timeouts, i.e. $\forall t_{timeout} : t_{timeout} := t + t_i$ where $r_t \in r^+, \exists q, \delta_t \langle r_t, t_i, q \rangle, T \langle r_t, t_{timeout} \rangle$

3. Firing Transition Rule: Choose an enabled transition from the set of enabled discrete or timed transitions. There are two cases, the chosen transition is:

- a) Discrete Transition: In case of $r^- = \emptyset$ than the execution of the transition is as in described formerly. If the token exits a region i.e. $r^- \neq \emptyset$, then the following rule extends the firing rule of discrete transitions: $\forall t_s, \forall q_s$ in $\delta_t \langle r_i, t_s, q_s \rangle$ where $r_i \in r^-$, the timer of the regions are invalidated i.e. $t_s := \infty$
- b) Timed Transition: The transition with the minimal timeout value is selected, i.e. transition $\exists t_i, \exists s_i, \delta_t \langle r_i, t_i, s_i \rangle$ where $r_i \in r$ and t_{min} is the minimum from all t_i , than the following rules apply: the global time is set $t := t_{min}$, the local clocks the token just left are invalidated i.e. $\forall t_s, \forall q_s$ in $\delta_t \langle r_i, t_s, q_s \rangle$ where $r_i \in r^-$, the timer of the regions are invalidated i.e. $t_s := \infty$ and move to the next state according to δ_t .

We can generate from the expression $(\langle a(a)^* \rangle_{t > 10} r(r)^* a)$ the automaton of Figure 4.7. Note the additional timed region which contains the “allocating phase” state.

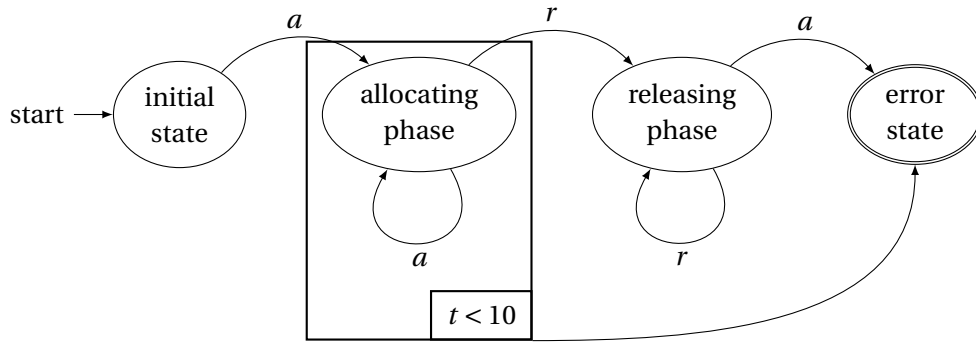


Figure 4.7 Timed region event automaton of the two-phase locking example

4.3.3 Nondeterministic Timeout Event Automaton

For accepting languages generated by timed regular expressions, the concept of timed event automaton is introduced.

Definition 4.13 A Nondeterministic Timeout Event Automaton is a tuple $\langle Q, \Sigma, \delta, q_0, F, C \rangle$ where

- Q, Σ, q_0 , and F are the same as in Definition 4.12,
- C is a finite set of clocks.

lost reference
on [1].

- $\delta \subseteq \langle Q \times \Sigma' \times Q \times (\mathbb{R}^+ \cup \{\infty\})^C \times 2^C \rangle$ i.e. the transitions of the system in the order of: the state which is the source of the transition, the corresponding event for the transition (or ε if it is an ε -transition), the state which is the destination of the transition, a function which assigns time values for each clock, and the set of clocks which needs to be reset upon firing the transition.

Definition 4.14 A timed input sequence is denoted $(e_0, t_1, e_1, \dots, t_n, e_n)$ where $e_i \in \Sigma'$ and $t_i \in \mathbb{R}_0^+$ i.e. a sequence of events and the elapsed times between them.

Definition 4.15 A trace in a Timeout Event Automaton is a sequence described as $(r_0, \langle a_1, t_1, v_1 \rangle, \dots, \langle a_n, t_n, v_n \rangle, r_n)$, where $\forall r_i \in Q, \forall a_i \in \Sigma', \forall t_i \in \mathbb{R}^+$ i.e. a sequence of states, the elapsed time between them, and the clock assignments. The trace corresponds to the timed input sequence $(e_0, t_1, e_1, \dots, t_n, e_n)$ if:

- $r_0 = q_0$, i.e. the first state of the trace is the initial state of the Automaton,
- $\forall i \in [1, n-1]: \langle r_i, a_{i+1}, r_{i+1}, u, Res \rangle \in \delta, v_i' = v_i + t_{i+1}, \forall c \in C: v_i'(c) < u(c)$, and

$$v_{i+1} = \begin{cases} 0 & \text{if } c \in Res \\ v_i' & \text{if } c \notin Res \end{cases}$$

- The last r_n is reached after reading the entire input sequence.

A trace is accepting if $r_n \in F$.

Definition 4.16 A Nondeterministic Timeout Event Automaton accepts a timed input sequence if there exists an accepting trace for the timed input sequence.

Definition 4.17 A token in a Nondeterministic Timeout Event Automaton is a set of the active states with their timer values.

The semantics of the timed deterministic finite automaton is defined as follows:

$Q_t \subseteq Q$ is the set of states with outgoing timed transitions, i.e. $\forall s \in Q_t: \exists t, t \in \delta: t = \langle s, a, s', h, c \rangle$, where $a \in \Sigma, s' \in Q, h \neq \emptyset$, and $c \neq \emptyset$.

We have to define rules for entering states with timed outgoing transitions and we also define the general rules of changing states. The current tokens in the automaton are noted as τ_{act} .

1. Initialization rule: $\forall c \in C: c := \infty$, i.e. on initialization of the automaton, all timeout clocks must be invalidated. On initialization a token must be placed on the initial

Revision re-
quired

state of the automaton.

2. **Firing Transitions Rule:** Q_{act} is the set of states which contains tokens. Upon receiving input i , where $i \in \Sigma$, $\forall q \in Q_{act}$, $\forall t \in \delta: \delta\langle q, i, q', T, Res \rangle$, $\forall c \in Res: c := \infty$, $\forall c \in C: c := T(c)$, the token is copied from q to q' . The old tokens which are not created in this step must be removed.
3. **Timeout Rule:** If one of the clocks reaches 0 for a token, the computation fails, and the token must be removed.

4.3.4 Transformation from Timed Regular Expressions to Timed Automata

Constructing a Timeout Region Automaton from a Timeout Regular expression depends on the steps described in Section 4.2.4 as only one new operator has been added to the language. This operator can be compiled to a Nondeterministic Timeout Region Automaton as show on Figure 4.8.

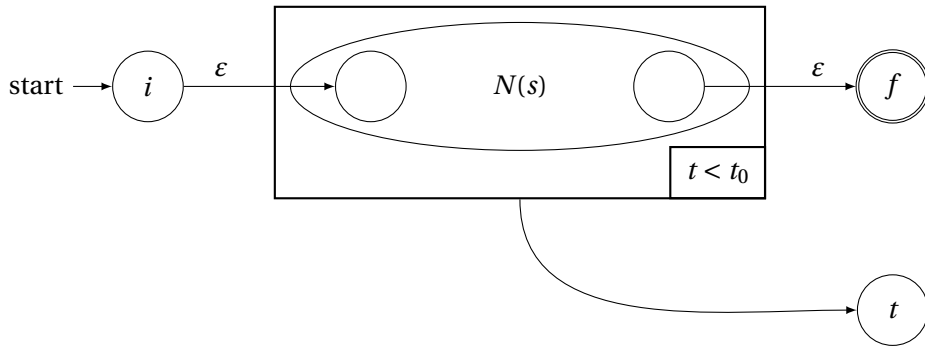


Figure 4.8 Constructed NFA from the timeout expression $\langle N(s) \rangle_{t < t_0}$

Automata with timeout regions can be compiled into simple timeout automata without any regions. For each region, the incoming and the outgoing transitions must be collected and a clock must be added to the set of clocks. The incoming transitions must start the given clock, with the timeout value of the timed region, and the outgoing transitions must reset the clock.

This construction yields an automaton with ϵ -transitions. Determinizing the nondeterministic automaton constructed from a timed regular expression is not a solved problem *yet*. However, there are some results regarding the transformation from a subset of MTL to deterministic timed automaton [26].

Testing whether a timed automaton is determinizable has been proved undecidable[15]. Also, the undecidability of universality has been further investigated, and rather restricted classes of timed automata suffer from that undecidability result. On the other hand, classes

When I find the solution to write text on the SE corner, fix this

of timed automata have been exhibited, that either can be effectively determinized (for instance event-clock timed automata [2], or timed automata with integer resets [36]), or for which universality can be decided (for instance single-clock timed automata [28]).[4]. This means that we need further research in this direction, to find out whether nondeterministic timed region automata generated from timed regular expressions can be determinized, or not.

4.4 Parametric Timed Languages

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

4.4.1 Parametric Timed Regular Expression

In this section I will overview the parametric extension of the Timed Regular Expression formalism.

Definition 4.18 A Parametric Event Γ is a tuple $\langle \Sigma, P \rangle$, where

- Σ the input alphabet, as described formerly in Definition 4.11,
- and P is a relation with the parameters. It describes that for the given event what are the parameter values. For a given event, the size of P must be the same.

Definition 4.19 A Parametric Timed Regular Expression is a Timed Regular Expression as described in Definition 4.11, but in this case, we use Γ instead of Σ .

With parametric timed regular expressions the example can be extended regular expression with parameters: $\langle a[i] a[i]^* \rangle_{t > 10} r[i] r[i]^* a[i]$ Note that this allows us to have only one instance of the automaton even for multiple tasks, each with its own ID.

4.4.2 Parametric Timed Region Event Automaton

Definition 4.20 A Parametric Timed Region Event Automaton $\langle Q, \Gamma, \delta, q_0, F, t, T \rangle$ where

- Q, δ, q_0, F, t and T are the same as in Definition 4.12,
- Γ is a finite, nonempty set, representing the parametric event set of the automaton.

Semantics

The semantics are only changed in the transition rules, described as:

- The transitions are only enabled if the fix parameters in the transition matches the event parameters. For example, if the pattern contains ... $a[i, 2]$..., if an event a occurs then the second parameter must be 2 to enable the transition for that event,
- A transition is enabled if the event parameters can be bound to the parameters of the token in the input location, and the corresponding tokens with the parameter bindings will be copied from the input to the output.

Using the parametric timed region automaton we can compile our timed regular expression $(\langle a[i] a[i]^* \rangle_{t > 10} r[i] r[i]^* a[i])$ into the automaton on Figure 4.9.

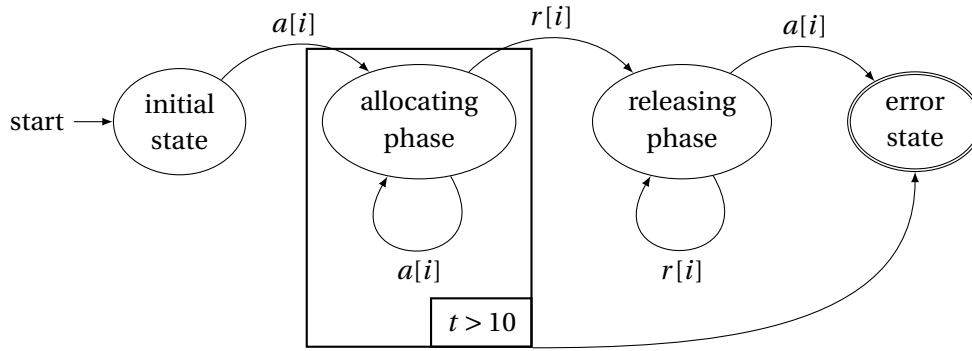


Figure 4.9 Parametric timed region event automaton of the two-phase locking example

4.4.3 Parametric Timed Event Automaton

As in the case of the Timeout Region Event Automaton, the case is somewhat similar here; a formalism is introduced according to the literature.

Revision needed

Definition 4.21 A Nondeterministic Parametric Timeout Event Automaton is a tuple $\langle Q, \Gamma, \delta, q_0, F, C \rangle$ where

- Q, δ, q_0, F , and C are the same as in Definition 4.13,
- Γ is a finite, nonempty set, representing the parametric event set of the automaton.

Definition 4.22 A timed parametric input sequence is denoted $(e_0, t_1, e_1, \dots, t_n, e_n)$ where $e_i \in \Gamma$ and $t_i \in \mathbb{R}_0^+$ i.e. a sequence of parametric events and the elapsed times between them.

Definition 4.23 A trace in a Parametric Timeout Event Automaton is a sequence $(r_0, \langle a_1, t_1, v_1 \rangle, \dots, \langle a_n, t_n, v_n \rangle, r_n)$, where $\forall r_i \in Q, \forall a_i \in \Gamma, \forall t_i \in \mathbb{R}^+$ i.e. a sequence of states, the elapsed time between them, and the clock assignments. The trace corresponds to the timed input sequence $(e_0, t_1, e_1, \dots, t_n, e_n)$ if:

- $r_0 = q_0$, i.e. the first state of the trace is the initial state of the Automaton,
- $\forall i \in [1, n-1]: \langle r_i, a_{i+1}, r_{i+1}, u, Res \rangle \in \delta, v_i' = v_i + t_{i+1}, \forall c \in C: v_i'(c) < u(c),$

$$v_{i+1} = \begin{cases} 0 & \text{if } c \in Res \\ v_i' & \text{if } c \notin Res \end{cases}$$

- The last r_n is reached after reading the entire input sequence.

A trace is accepting if $r_n \in F$.

Definition 4.24 A Parametric Timeout Event Automaton accepts a timed input sequence if exists an accepting trace for the timed input sequence.

Definition 4.25 A token in a Parametric Timeout Event Automaton is a set of the active states with their timer values.

The semantics are changed as in the case of the Parametric Timeout Region Automaton. Section 4.4.2

4.4.4 Transformation from Parametric Timed Regular Expression to Parametric Timed Event Automaton

Constructing a Parametric Timeout Region Automaton from a Parametric Timeout Regular Expression depends on the steps described in Section 4.3.4 as only the event has been parametrized. The transformation of the new events require some kind of auxiliary data structure be constructed which can determine that the a given event parameter on a given transition corresponds to which token parameter. For example, if the event $\dots a[i, j] \dots$ should be transformed, then this auxiliary data structure should contain, that for this transition event's first parameter the corresponding token parameter is i and for the second parameter the corresponding token parameter is j .

Constructing a nondeterministic parametric timed region automaton from a parametric timed regular expression is possible according to the results know for timed regular

expressions. As parametric automata are always nondeterministic, thanks to the possibility of multiple tokens, we have no intention to determinize such automata. However, at least the ε -transitions should be eliminated from the automata, for efficient execution, and analysis.

To our best knowledge, there are no results about eliminating ε -transitions from a parametric time region automaton, being generated from parametric timed regular expression.

Chapter 5

Algorithms and Implementation

The currently implemented and algorithmized parts of the intermediate language are the following: the formal intermediate language (parametric timed regular expression), a parametric timed automaton implementation, a mapping between the two, and a prototype automaton executor.

5.1 Parametric Timeout Regular Expression

Currently the regular expression has a grammar implemented in Xtext, which generates a textual editor, and can parse textual input to an EMF model. This EMF model can be transformed later. The grammar is shown on Listing A.1.

Wrong cref

Note that the parameter listing is inside with square brackets “[]” to make the language context free. If we would use the simple round brackets, then A(B) would be ambiguous – it could be either an A event with a B parameter, or a simple sequence of event A and event B with an unnecessary bracket.

Do we require additional explanation?

5.2 Parametric Timed Event Automaton

The parametric timed automaton was implemented using Eclipse Modeling Framework to provide a metamodel for it. The entire static structure is represented with EMF objects, but some of the runtime concepts are not covered with this metamodel, this will be further explained in Section 5.4.1.

5.2.1 Finite Automaton

The subset of the metamodel which represents the Finite Automaton functions are shown on Figure 5.1. The event automaton logic is represented by the State, Transition, and EventGuard classes. Every State has a boolean flag to show whether it is an acceptor state

or not, and have any number of incoming and outgoing transitions. Transitions have one EventGuard which shows what type of event can fire the transition.

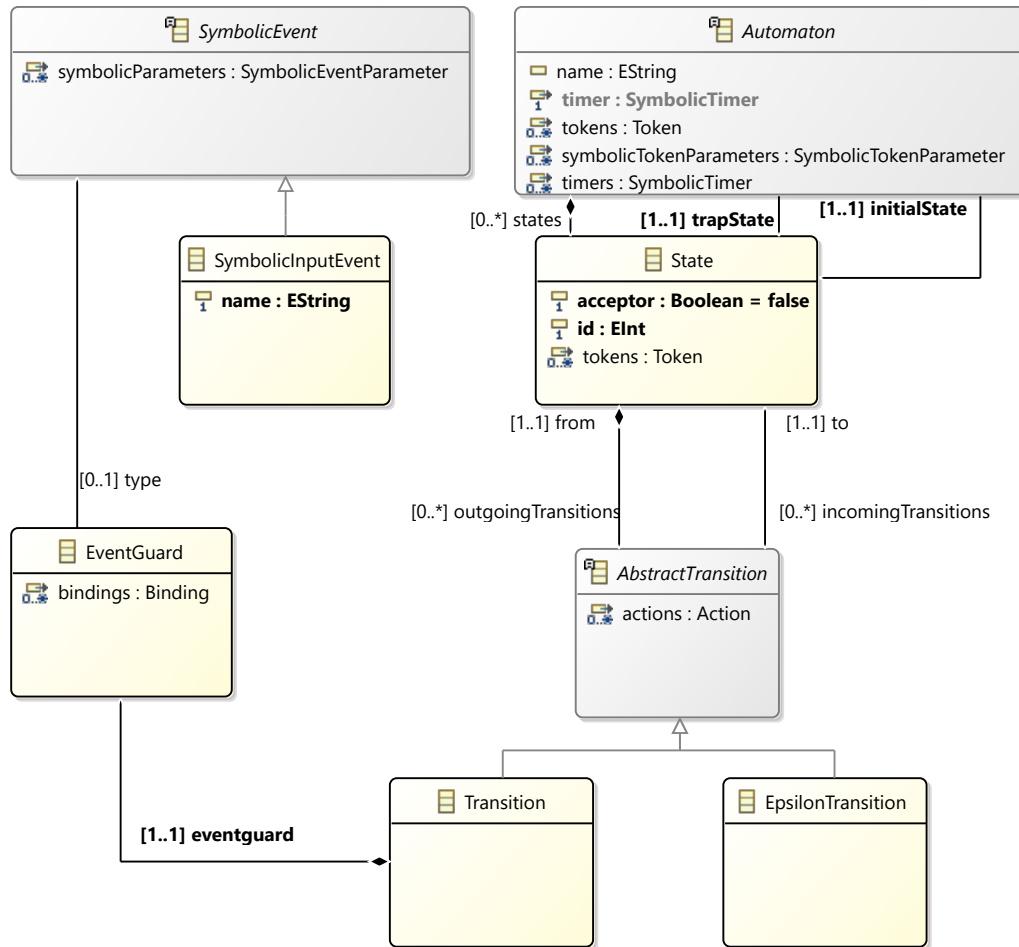


Figure 5.1 Metamodel of the Event Automaton

5.2.2 Timeout Event Automaton

The additional classes required to implement the Timeout functionalities are shown on Figure 5.2. Every timed expression in the regular expression is compiled to **SymbolicTimer**. Upon entering such timed region all transitions will have a **SetTimerAction** Action to start the timer. The transitions which exit the given region will have a **ResetTimerAction** Action for that timer. Every state which is in the Timeout region will have a **Transition** with a **SymbolicTimeoutEvent** instead of a **SymbolicInputEvent** to further specify the movement

of the token in case of timeout. Currently these SymbolicTimeoutEvent Transitions are always pointing on a Trap state, however as this gives a more generalized automaton structure so this can help in the future to make the implementation of the Timeout Event Automaton with Intervals easier.

A scratch of the implementation plan is duplicate the entire region, for each timeout $\langle \text{ComplexPattern} \rangle_{t_0 < t < t_1}$ compile the *ComplexPattern* twice. For the sake of easier understanding, let us call the first compilation's result pre-interval as it represents when the token is before the desired time interval, and the call the second compilation result in-interval as it represents when the token is in the desired time interval. For each state in the pre-interval part the transition with the timeout will point to the equivalent state in the in-interval part, and for each state in the in-interval part the transition with the timeout will point to a trap state. All of the transitions, which would point to the accepting state of the pre-interval should be pointed to a Trap State as these shows that the pattern match occurred before the required interval.

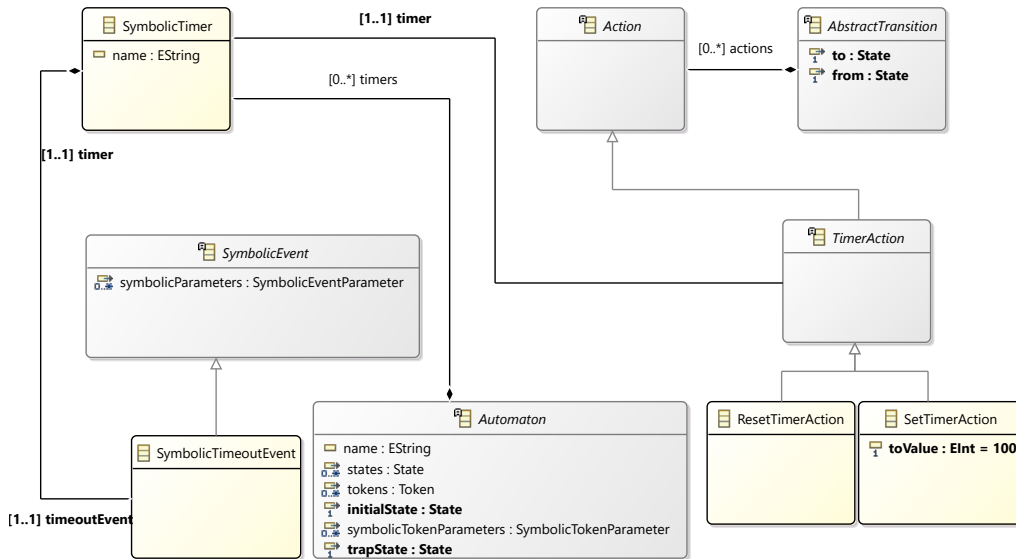


Figure 5.2 Metamodel of the Timed Event Automaton

5.2.3 Parameters and Bindings

The part of the automaton metamodel which is extended with the parametric properties are shown on Figure 5.3 The parametrization of the events is represented by the Parameter abstract class. The type of each parameter is implemented with a reference to a SymbolicEvent instance. Parameters can be Fix or Free, an Event only has Fix parameters, but the

tokens can have both. Event parameters can be bound to fix values or to a token parameter, this is done by the ConstantBinding and TokenParameterBinding respectively.

This parameter handling approach is not the best, however this is which can be easily implemented using EME. The deeper analysis of these parameter relations can be discussed in Section 5.4.1.

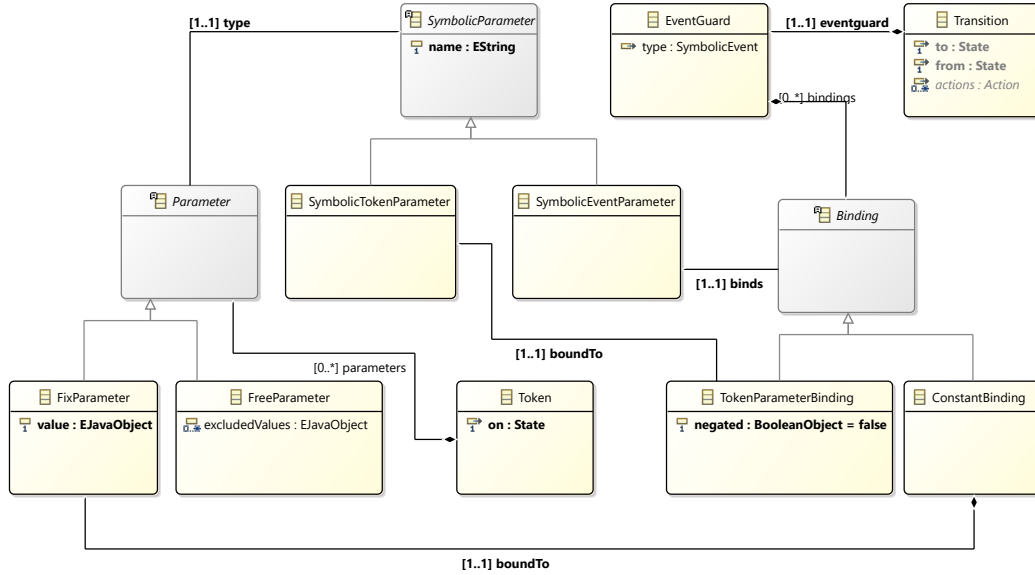


Figure 5.3 Metamodel of the Parametric Timed Event Automaton

5.3 Transformation from VEPL to the intermediate language

Regular expressions can be built from VEPL patterns, taking the event contexts into consideration. Strict immediate is shown on Table 5.1, and Chronicle is shown on Table 5.2. Each of the patterns must start with a Σ^* , as in Complex Event Processing every pattern might have arbitrary prefix.

Note that the semantics introduced in the previous chapter for the OR operator of Regular Expression, and the semantics of VEPL defined in [9] are not exactly the same.

5.4 Automaton Executor

Two approaches exist on executing a parametric timed automaton, and in both ways there are multiple tokens in each automaton, so in some sense both can be considered nonde-

Table 5.1 Basic VEPL operators in strict immediate context, expressed with regular expressions

VEPL operator	Regular Expression
$p_1 \rightarrow p_2$	$p_1 p_2$
$p_1 \text{ OR } p_2$	$p_1 p_2$
$p\{*\}$	p^*
$p[t]$	$p[t]$

Table 5.2 Basic VEPL operators in chronicle context, expressed with regular expressions

VEPL operator	Regular Expression
$p_1 \rightarrow p_2$	$p_1 (\Sigma / (p_2))^* p_2$
$p_1 \text{ OR } p_2$	$p_1 p_2$
$p\{*\}$	nothing
$p[t]$	$p[t]$

terministic. However from the token's point of view we can consider the two approaches deterministic and a nondeterministic.

Deterministic Execution In the deterministic approach the automaton should not contain any ε -edges and a “token can go only one way”, or more formally, there is only one transition for each token to fire. In this case every token moves exactly one on each event, splitting the potential bindings to two part, one which matched the event's parameter, and one which did not.

Nondeterministic Execution In the nondeterministic way, the automaton can contain ε -edges, and a token can move through multiple transitions each. This solution implies an another way to handle the parameters, as the executor should do the following steps:

Maybe pseudo-code

1. Save each token in a list,
2. Find the enabled transitions,
3. For each of the enabled transitions, create a parameter relation according to the event,
4. Compute the intersection of each token with each enabled transitions which is an outgoing transition of the state which the token is on,

5. Move each intersected token parameter binding to the state where the transitions point,
6. Move these tokens through the ε -closure, applying any actions to the moved parameters,
7. And remove the old tokens which were saved in the first step.

On the implementation level the Nondeterministic Executor has been implemented as it is more general can execute any automaton generated from the regular expressions.

Maybe an example execution?

5.4.1 Parameter Handling

The required operations on the Parameter Relations are:

- If the execution is deterministic, the following operations are required:
 - $P_1 = P_1 \cup P_2$, i.e. a relation can be added to another with mutating the first one,
 - $P_1 = P_1 \setminus P_2$, i.e. a relation can be removed from an another with mutating the first one,
 - can be checked if it is empty,
 - and add every possible parameter value with \forall .
- However if the execution is considered nondeterministic, an other set of operation is required:
 - $P_1 \cup P_2$, i.e. two relation's union can be computed,
 - $P_1 \cap P_2$, i.e. two relation's intersection can be computed,
 - can be checked if it is empty,
 - and add every possible parameter value with \forall .

The “add every possible parameter value” is only required upon the initialization of the automaton, and considering that the relations are empty on construction this operation can be changed to a complementer operation, which is more general. In both cases the union is required for further optimization, and not an absolute necessity.

5.4.2 Most Naïve approach

The most naïve approach is to keep every potential binding in the memory, for example if there is a pattern $a[i_1] b[i_1, \dots, i_n]$ where all parameters are integers and event $a[1]$ happens the state of the memory is shown on Table 5.3. This indicates that there would be $2^{32^{n-1}}$ objects in the memory, which is not feasible even if $n = 3$.¹

¹If each object's size is just 1 bit even then $n = 2$ would waste 512MB of memory

Table 5.3 State of the memory on the most naïve approach

i_1	i_2	i_3	...	i_n
1	0	0	...	0
1	1	0	...	0
1	0	1	...	0
1	1	1	...	0
\vdots				
1	1	1	...	1
1	2	0	...	0
1	2	1	...	0
\vdots				
1	2	1	...	1
1	2	2	...	0
\vdots				
1	2^{32}	2^{32}	...	2^{32}

5.4.3 Naïve approach

The naïve approach is to keep an abstract value for each parameter for each token, where the abstract value can be either be

- A concrete value, which shows that for that token the given parameter is bound to that value, i.e. the given parameter must be exactly that value, or
- a list of excluded values which shows that for that token the given parameters values have been reduced by them, i.e. the given parameter must not be any of the excluded values

With this approach, every operation seems to be plausible, however a simple counter-example easily shows that the solution can not support the deterministic approach. Consider the following example:

1. At first we have a parameter relation P , where $\langle p_1 = \forall, p_2 = \forall \rangle$, i.e. with two excluded values with empty exclusion list.
2. If the relation R , $\langle p_1 = 1, p_2 = 1 \rangle$ is subtracted from P , P would be reduced to $\langle p_1 = \forall \setminus \{1\}, p_2 = \forall \setminus \{1\} \rangle$.

Table 5.4 An example of a Decision List

#	containment	A	B
1	+	2	*
2	+	3	5
3	−	*	*

Table 5.5 An example of a Disjunctive Decision Set

containment	A	B
+	2	$* \setminus \{1, 2\}$
+	3	5
−	*	*

3. If the relation $S, \langle p_1 = 2, p_2 = 2 \rangle$ is subtracted from P , P would be reduced to $\langle p_1 = \forall \setminus \{1, 2\}, p_2 = \forall \setminus \{1, 2\} \rangle$.
4. The problem occurs when we would try to subtract the relation $\langle p_1 = 2, p_2 = 1 \rangle$ as it *should* be in the relation, but it is not.

5.4.4 Decision List

A decision list is a list of data, where each cell contains either a concrete value or the symbol * which stands for “anything”. To find out if a values in the relation the rows must be read in order, and find the first row which conforms the values you are searching for ... and return the containment. An example of a decision list is shown on Table 5.4

algorithm

5.4.5 Disjunctive Decision Set

An event – in a more formal way – is a conjunction of a logical statement. When we think of $Allocate(1, 2)$ we actually mean that we have an event called *Allocate* where $Task = 1$ & $Resource = 2$. With this mindset we can assume that a Parameter Binding Relation is actually only a disjunction of such statements.

An example of Disjunctive Decision Set is shown on Table 5.5.

5.4.6 MDDe

MDDe is a Multiple-value Decision Diagram with else branches. We could use simple MDD's with intervals, but in many cases the parameters do not have a supremum and infimum and they are not in a sequence.

5.4.7 Additional Data Structures

There are many more data structures which can be used for the underlying implementation of a Parameter Relation. Two more was under consideration, Trie, and BDD. Both are somewhat similar to MDD, however Trie is a mutable data structure, which theoretically seems to be more efficient in the case of the Deterministic Execution, as the operations are always mutating the data structure. The other one was BDD which is the Binary version of MDD, but thanks to that, the tree structure would be higher, however with a lower branching factor, and updating such data structures could be cheaper than using MDDs.

5.5 Preliminary measurements

To analyze the efficiency of the implementation some measurements were done, using the example from Chapter 4.

5.5.1 Measurement setup

As the current implementation lacks of logical clocks, and the timed properties can only be evaluated by elapsing the required time, the pattern $a[i] a[i]^* r[i] r[i]^* a[i]$ will be the parametric and not temporal example. The scalability is viewed from three points:

1. How well does the implementation scale with the size of the pattern?
2. How well does the implementation scale with the size of the input sequence?
3. With a given pattern and a given size for the input sequence, does it matter how many times the automaton accepts?

To answer the first question, the pattern must be extended. The extension was with adding one more requirement to the pattern: How much is the maximum allocation allowed for one process. Therefore the extended pattern will be $(a[i] a[i]^* r[i] r[i]^* a[i])|(a[i], \dots, a[i])$, where on the right side of the pattern the length sequence of $a[i]$ events will be expanded to one, ten, one hundred, one thousand, and ten thousand.

To answer the second question, input sequences were generated in five sizes: one long, ten long, one hundred long, one thousand long and ten thousand long ones. However to measure multiple cases, these inputs are generated with the following logic:

1. A randomly generated sequence,
2. only events $a[1]$ and $a[2]$ but on a random sequence to not a match, in other words to generate less than the required $a[1]$ events and $a[2]$ events,
3. only sequences of $a[1]r[1]a[1]$ and $a[2]r[2]a[2]$ to have as much matches as possible.

The measurements were done with two underlying parametric data structures MDDe, and the Disjunctive Decision Set.

Threats to validity

Every measurement was done thirty two times. The first two measurement result was dropped as it was considered as the warm up for the JVM. From the remaining thirty, the median was calculated, to have a stable value which is the least affected by the noises of the measurement and the extremely high or low values. Between every iteration of a measurement, the garbage collector was called three times followed by a sleep to ensure that the garbage collection is if required.

The measurements were aimed to cover some part of most use cases, however they are not exhausting every potential use case to think of.

5.5.2 Measurement Results

Compilation time:

Diagram

Explanation

Chapter 6

Conclusions and future work

A formalism was introduced, to provide a language for complex event processing

6.1 Future Work

Acknowledgments First and foremost, I would like to thank my supervisor, András Vörös, as without his guidance in the last five years I could not create such a work as this thesis.

I would also like to thank my two other advisors Gábor Bergmann PhD, and Rebeka Farkas, without their help in formalization and algorithmization would have been way more cumbersome.

I would like to thank Oszkár Semeráth as without his help the measurements would have been a lot harder.

And last, but not least, I would like to thank Kristóf Marussy, and Bálint Hegyi for helping me with L^AT_EX and T_ikZ related issues, to help me create such a beautiful document.

Appendix

A.1 Xtext grammar for the Parametric Timeout Regular Expression

Listing A.1 Xtext grammar for the Parametric Timeout Regular Expression

```

1  grammar hu.bme.mit.inf.ParametricTimedRegularExpression with
    org.eclipse.xtext.common.Terminals

3  generate parametricTimedRegularExpression
    "http://www.bme.hu/mit/inf/ParametricTimedRegularExpression"

5  RegexModel:
6      (alphabet=Alphabet)?
7      (declarations+=ExpressionDeclaration)*;

9  Alphabet:
10     {Alphabet} 'alphabet' '=' '{' (functors+=Functor (','
        functors+=Functor)*)? '}'';

12  Functor:
13     name=ID ('/' arity=INT)?;

15  ExpressionDeclaration:
16     'expression' name=ID ('[' vars+=Var (',' vars+=Var)* ']')?
        '=' body=Expression;

18  Expression:
19     Choice;

21  Choice returns Expression:
22     And ({Choice.elements+=current} ('|' elements+=And)+)?;

```

```

24 And returns Expression:
25     Sequence ({And.elements+=current} ('&'
        elements+=Sequence)+)?;

27 Sequence returns Expression:
28     MultExpression ({Sequence.elements+=current}
        elements+=MultExpression+)?;

30 MultExpression returns Expression:
31     PonNegExpression ({Star.body=current} '*' |
        {Plus.body=current} '+' | {Cardinality.body=current}
        '{' n=INT '}' )?;

33 PonNegExpression returns Expression:
34     NegExpression | ParenExpression;

36 NegExpression returns Expression:
37     {NegExpression} "!" body=ParenExpression;

39 ParenExpression returns Expression:
40     '(' Expression ')' | Any | Inverse | TimedExpression |
        Event;

42 Any:
43     {Any} 'S';

45 Inverse:
46     '(' 'S' '\\ ' excludes+=Event (',' excludes+=Event)* ')';

48 TimedExpression:
49     '<' body=Expression '>' '[' timeout=INT ']';

51 Event:
52     functor=[Functor] ('[' parameters+=Parameter (','
        parameters+=Parameter)* ''] )?;

54 Parameter:
55     VarParameter | NegVarParameter | SingletonParameter |
        FixParameter;

57 NegVarParameter:

```

```
58      'not' body=VarParameter;

60  FixParameter:
61      FixIntParameter | FixStringParameter;

63  FixIntParameter:
64      body=INT;

66  FixStringParameter:
67      body=STRING;

69  VarParameter:
70      variable=[Var];

72  SingletonParameter:
73      {SingletonParameter} '_';

75  Var:
76      name=ID;

78  terminal ID:
79      '^'? ('a'..'z' | 'A'..'Z') ('a'..'z' | 'A'..'Z' | '_' |
        '0'..'9')*;
```

References

- [1] Rajeev Alur and David L Dill. “A theory of timed automata”. In: *Theoretical computer science* 126.2 (1994), pp. 183–235.
- [2] Rajeev Alur, Limor Fix, and Thomas A Henzinger. “A determinizable class of timed automata”. In: *Computer Aided Verification*. Springer. 1994, pp. 1–13.
- [3] Eugene Asarin, Paul Caspi, and Oded Maler. “Timed regular expressions”. In: *Journal of the ACM* 49.2 (2002), pp. 172–206.
- [4] Christel Baier, Nathalie Bertrand, Patricia Bouyer, and Thomas Brihaye. “When are timed automata determinizable?” In: *Automata, Languages and Programming*. Springer, 2009, pp. 43–54.
- [5] László Balogh. “Complex Event Processing based on Automata theory”. Bachelor’s Thesis. Budapest University of Technology et al., 2015.
- [6] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. “Incremental evaluation of model queries over EMF models”. In: *Model Driven Engineering Languages and Systems*. Springer, 2010, pp. 76–90.
- [7] Grady Booch, Ivar Jacobson, and Jim Rumbaugh. “OMG unified modeling language specification”. In: *Object Management Group* 1034 (2000), pp. 15–44.
- [8] Luiz Fernando Capretz. “Y: a new component-based software life cycle model”. In: *Journal of Computer Science* 1.1 (2005), pp. 76–82.
- [9] István Dávid, István Ráth, and Dániel Varró. “Streaming Model Transformations By Complex Event Processing”. English. In: *Model-Driven Engineering Languages and Systems*. Vol. 8767. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 68–83. DOI: 10.1007/978-3-319-11653-2_5.
- [10] Nelly Delgado, Ann Quiroz Gates, and Steve Roach. “A taxonomy and catalog of run-time software-fault monitoring tools”. In: *Software Engineering, IEEE Transactions on* 30.12 (2004), pp. 859–872.
- [11] *Eclipse Modeling Project*. URL: <https://eclipse.org/modeling/emf/> (visited on 10/22/2015).

- [12] Cindy Eisner. “PSL for runtime verification: Theory and practice”. In: *Runtime Verification*. Springer. 2007, pp. 1–8.
- [13] *EsperTech - Esper*. URL: <http://www.espertech.com/esper/> (visited on 10/22/2015).
- [14] *Finite State Machine Designer*. URL: <http://madebyevan.com/fsm/> (visited on 10/26/2015).
- [15] Olivier Finkel. “Undecidable problems about timed automata”. In: *Formal Modeling and Analysis of Timed Systems*. Springer, 2006, pp. 187–199.
- [16] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.
- [17] OM Group et al. “OMG Unified Modeling Language (OMG UML), Superstructure”. In: *Open Management Group* (2009).
- [18] David Harel and PS Thiagarajan. “Message sequence charts”. In: *UML for Real*. Springer, 2003, pp. 77–105.
- [19] Øystein Haugen. “Comparing uml 2.0 interactions and msc-2000”. In: *System Analysis and Modeling*. Springer, 2005, pp. 65–79.
- [20] Øystein Haugen. “MSC-2000 interaction diagrams for the new millennium”. In: *Computer Networks* 35.6 (2001), pp. 721–732.
- [21] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. “Online monitoring of metric temporal logic”. In: *Runtime Verification*. Springer. 2014, pp. 178–192.
- [22] Timothy Kam, Tiziano Villa, Robert Brayton, and Alberto Sangiovanni-Vincentelli. “Multi-valued decision diagrams: theory and applications”. In: *Int. J. Multiple-Valued Logic* 4.1–2 (1998), pp. 9–62.
- [23] Jochen Klose. “Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior”. PhD thesis. Carl von Ossietzky University of Oldenburg, 2003. URL: <http://d-nb.info/970053304>.
- [24] Monica Lam, Ravi Sethi, JD Ullman, and AV Aho. *Compilers: Principles, techniques and tools*. 2006.
- [25] Balogh László, Florian Deé, and Hegyi Bálint. *Hierarchical runtime verification for critical cyber-physical systems*. Tech. rep. Budapest University of Technology et al., 2015.
- [26] Dejan Ničković and Nir Piterman. *From MTL to deterministic timed automata*. Springer, 2010.
- [27] Leon Osborne, Jeffrey Brummond, Robert D Hart, Mohsen Zarean, and Steven M Conger. *Clarus: Concept of operations*. Tech. rep. 2005.

- [28] J  el Ouaknine and James Worrell. "On the language inclusion problem for timed automata: Closing a decidability gap". In: *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*. IEEE. 2004, pp. 54–63.
- [29] Lee Pike, Sebastian Niller, and Nis Wegmann. "Runtime verification for ultra-critical systems". In: *Runtime Verification*. Springer. 2012, pp. 310–324.
- [30] Amir Pnueli. "The temporal logic of programs". In: *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE. 1977, pp. 46–57.
- [31] Mark Proctor. "Drools: a rule engine for complex event processing". In: *Applications of Graph Transformations with Industrial Relevance*. Springer, 2012, pp. 2–2.
- [32] 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.
- [33] Thomas Reinbacher, J  rg Brauer, Martin Horauer, Andreas Steininger, and Stefan Kowalewski. "Past time LTL runtime verification for microcontroller binary code". In: *Formal Methods for Industrial Critical Systems*. Springer, 2011, pp. 37–51.
- [34] Miro Samek. "Who moved my state". In: *Dr. Dobb's Journal* (2003).
- [35] Jocelyn Simmonds, Shoham Ben-David, and Marsha Chechik. "Monitoring and recovery for web service applications". In: *Computing* 95.3 (2013), pp. 223–267.
- [36] P Vijay Suman, Paritosh K Pandya, Shankara Narayanan Krishna, and Lakshmi Manasa. "Timed automata with integer resets: Language inclusion and expressiveness". In: *Formal Modeling and Analysis of Timed Systems*. Springer, 2008, pp. 78–92.
- [37] Seema Suresh Kute and Surabhi Deependra Thorat. "A Review on Various Software Development Life Cycle (SDLC) Models". In: *IJRCCT* 3.7 (2014), pp. 776–781.
- [38] VIATRA. URL: <http://www.eclipse.org/viatra/> (visited on 05/19/2017).
- [39] VIATRA/CEP. URL: <https://wiki.eclipse.org/VIATRA/CEP> (visited on 10/25/2015).
- [40] VIATRA-Query. URL: <http://wiki.eclipse.org/VIATRA/Query> (visited on 05/19/2017).
- [41] Dolores R Wallace and Roger U Fujii. "Software verification and validation: an overview". In: *IEEE Software* 3 (1989), pp. 10–17.
- [42] Tim Weilkiens. *Systems engineering with SysML/UML: modeling, analysis, design*. Morgan Kaufmann, 2011.
- [43] Yakindu. URL: <http://statecharts.org/> (visited on 10/26/2015).