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

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

László Márk Balogh

# Complex Event Processing Based on Automata Theory

MSc Thesis

Supervisors:

Vörös András
dr. Bergmann Gábor
Farkas Rebeka

Budapest, 2017

# Contents

**Kivonat**   Ide kerül a kivonat.

**Kulcsszavak**   diplomaterv, sablon, LaTeX

**Abstract**  Here comes the abstract.

# 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 5.

..........................................................
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 safety-critical and cyber-physical system, to increase safety, as in case of a failure in these systems there are always 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. And 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 I will introduce the background of our work [24] and my previous BSc thesis [5] towards an approach supporting correct system design.

## 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 suboptimal 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[26].

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 it's 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
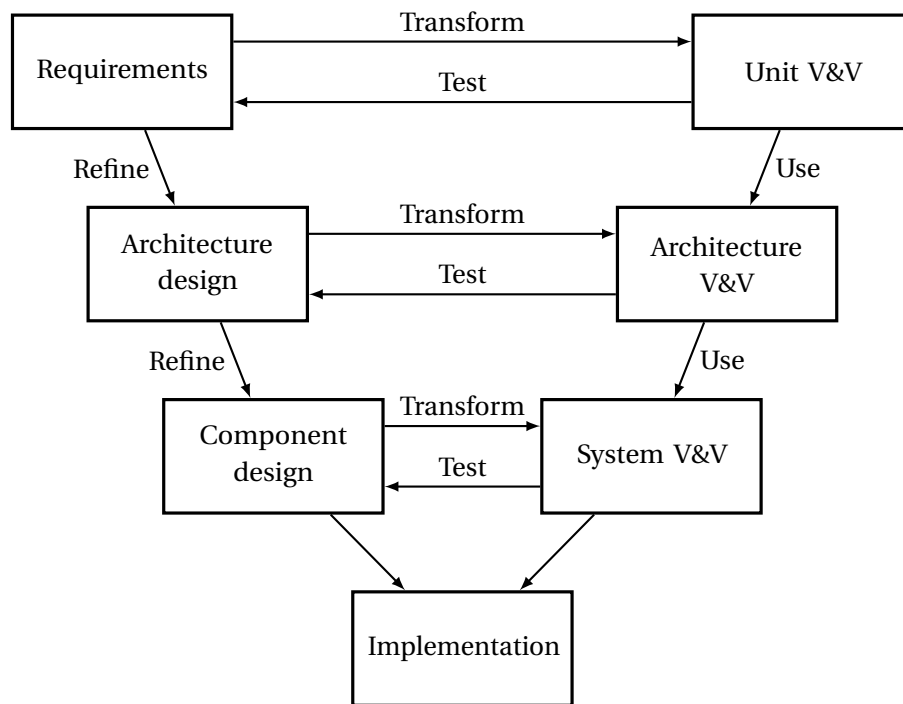
**Figure 2.1**   The traditional V model[26]

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 is formal enough 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.
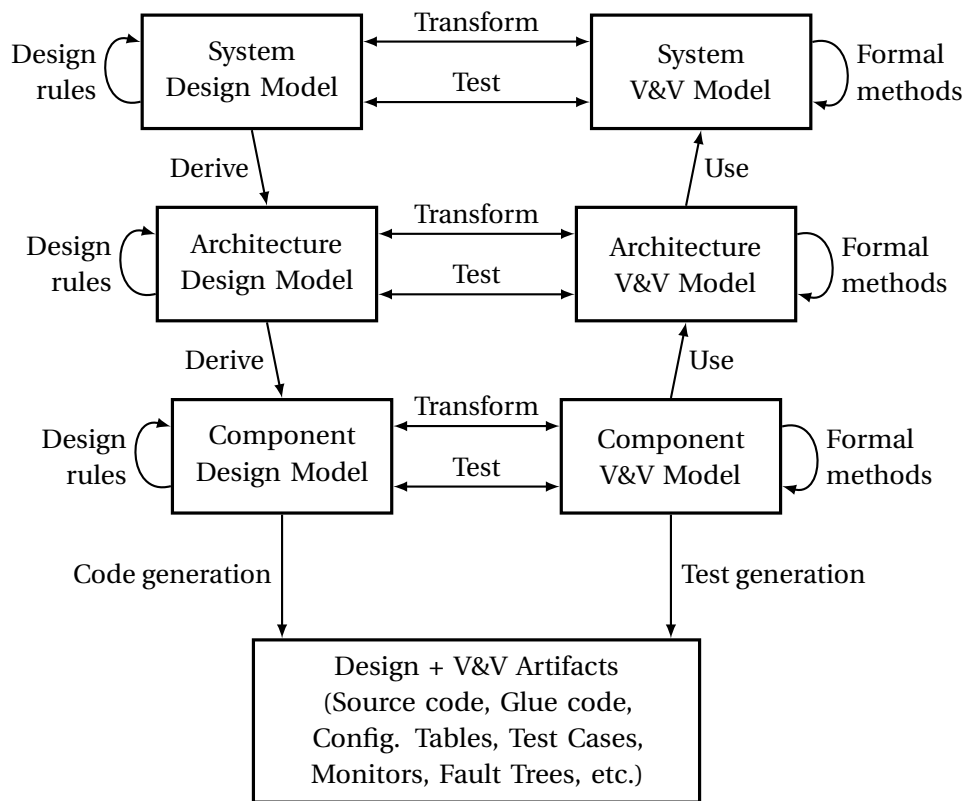
**Figure 2.2**   An overview of the Y model

## 2.1.2  Y model

The Y model[8] is an extension of the V model[35], 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 resulted a wide range of such languages over the years to support efficient product development. This allows the use of domain specific languages, which leads to a shorter modeling process. The result is the need for complex model transformations before the verification can begin, but since the domain specific languages are tailored exactly to the problem, their usage is more efficient – but requires expertise in the domain.

Standardized modeling languages were developed like the UML (Unified Modeling Language[7]) and SysML (Systems Modeling Language[16][40]).

#### 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][22]. 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[32].

Statecharts are usually developed in tools that support the graphical design of the model (e.g.: Yakindu[41], an Eclipse based editor). Statecharts are commonly used as a form of component description, as it can represent the behaviour 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[39], 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 ap-

proaches. 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 (also formal) specification. On the other hand, applying formal methods usually have higher costs, and the verification of complex systems can be impossible due to the phenomenon of state space explosion. As a result, a possible application of formal methods is to verify 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 to complexity of the design time verification. As systems are getting larger, the application of formal methods are more and more limited as the resources needed for verification cannot be realized. This means that formal verification methods must verify an abstract model, not the deployed system itself. In addition, specifications are rarely complete, and design time methods can rarely handle hardware errors.

Runtime verification uses monitors to observe certain (usually critical) components, checking whether their operation violates properties described 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

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 can be found in the incoming stream, e.g. events followed by another events in some sequence.
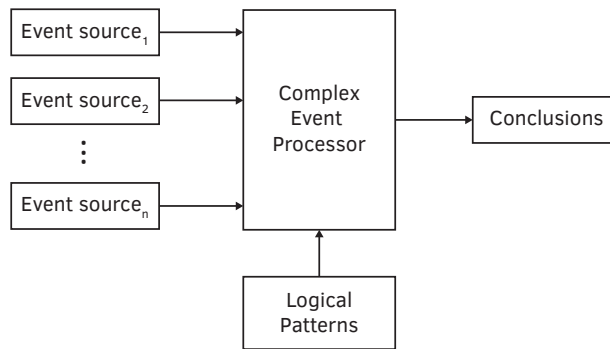
**Figure 2.3**   Complex event processing overview TODO: redraw this image with Power Point

### 2.3.1   Complex Event Processing Frameworks

There are many Complex Event Processing Frameworks in the industry.

One of these tools is Drools[30] which is the leading Java based open-source rule engine. It is a hybrid chaining engine meaning 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 into runtime verification is overviewed. In general, system level runtime verification can be provided with the help of complex event processing. A complex event processing framework can monitor either the messaging between components and also the current state of the system and detected 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.

**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]).

## 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:

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

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

### 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 meta model (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.[36]

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[38].

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 exact 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[37].

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

This is actually not related work, maybe remove this?

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[29]. 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[25].

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 [33] 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 [28] 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 [31] 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[37][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
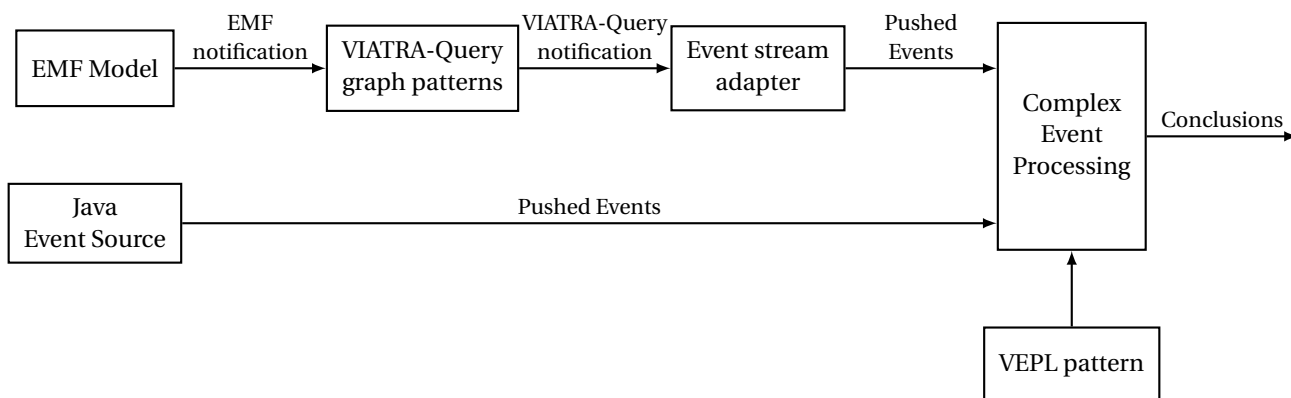


**Figure 3.1**   A brief outline of the live model integration in VIATRA-CEP

**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 | $p_1$ 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 [37].

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 $(A \rightarrow B)$ OR $C$, then trace $A, C$ will not match the pattern.[1]

- "AND" is a binary operator, e.g. $A$ AND $B$ AND $C$ is equivalent to $(A$ AND $B)$ AND $C$, and it will not accept the trace $ACB$

---

[1]Meanwhile 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 $(A \rightarrow B)$ OR $C$ is $\Sigma^*(AB)|C$

**Table 3.2**   Syntactic sugars

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

**Table 3.3**   Syntactic sugars mapped to basic operators

| Operator name | Denotation | Equivalent |
|---|---|---|
| and | $p_1$ AND $p_2$ | $((p_1 \rightarrow p_2)$ OR $(p_2 \rightarrow p_1))$. |
| negation | NOT $p$ | $\Sigma \setminus p$, where $\Sigma$ is the set of all the possible events. TODO: And complex NEG? |
| multiplicity | $p\{n\}$ | $p \rightarrow p \rightarrow \ldots p$, 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 $AB$, 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 | $\underline{A\,C\,B}$ | ✓ |
| | Strict Immediate | $A\,\underline{A\,B}\,B$ | ✓ |
| | Immediate | $\overline{A\,\underline{A\,B}\,B}$ | $2 \times$ ✓ |
| | Chronicle | $\overline{A\,\underline{A\,B}\,B}$ | $2 \times$ ✓ |

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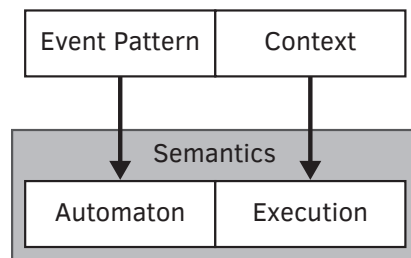


**Figure 3.2**   The architectural overview of the VIATRA-CEP TODO: redraw this image with Power Point

## 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 will be described here, however these are not in the scope of this thesis, as their implementation require a different approach.

**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.
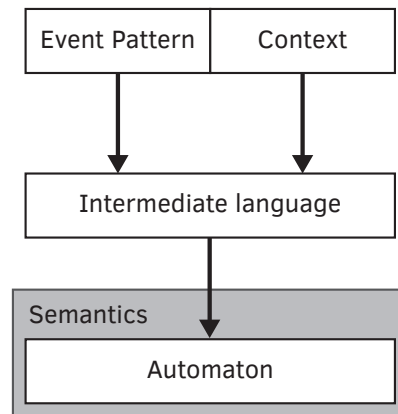
**Figure 3.3** The architectural overview of our proposed architecture TODO: redraw this image with Power Point
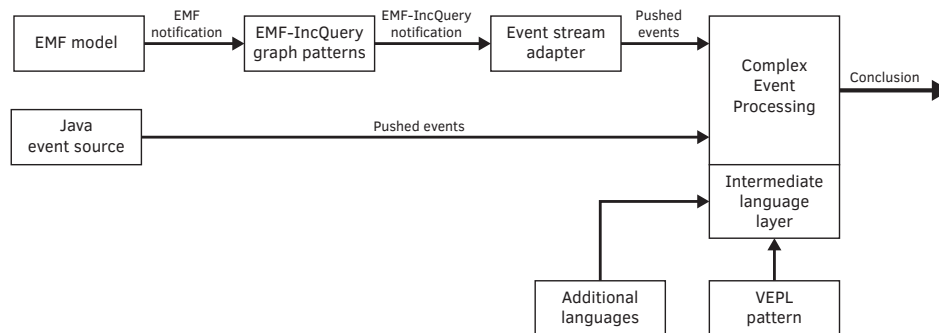


**Figure 3.4** A brief outline of the live model integration in VIATRA-CEP, with the intermediate language TODO: redraw this image with Power Point

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.

Reference

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 seconds[1].

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.

---

[1]We 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.

## 4.1   Overview of Formalisms

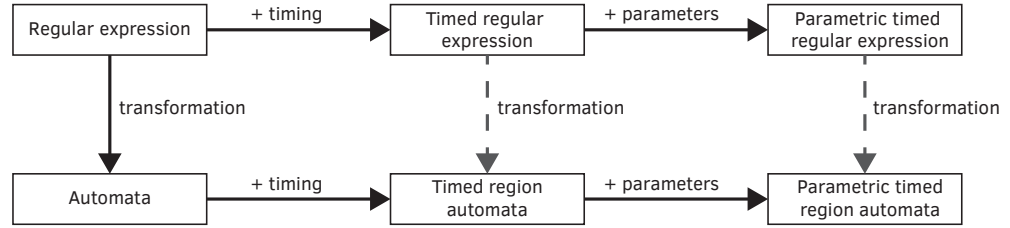The figure needs the not region automaton



**Figure 4.1**   Overview of the formalisms TODO: redraw this image with Power Point

The used formalisms and their relations are show 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.

## 4.2   Regular Languages

### 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 $\Sigma$ is the set of possible events.

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

1.   $a$ for every letter $a \in \Sigma$ and the special symbol $\varepsilon$ are expressions.

2.   If $\varphi, \varphi_1, \varphi_2$ are $\Sigma$-expressions then $\varphi_1 \varphi_2, \varphi_1 | \varphi_2, \varphi^*$ are $\Sigma$-expressions[3].

The meaning of these operators:

- $\varphi_1 \varphi_2$: Sequence, $\varphi_2$ must occur after $\varphi_1$.

- $\varphi_1 | \varphi_2$: Choice, $\varphi_1$ or $\varphi_2$ must happen.

- $\varphi^*$: Closure (also known as Kleene-star), $\varphi$ can occur $n$ times, where $0 \le n < \infty$

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 $\langle Q, \Sigma, \delta_d, q_0, F \rangle$ [23] where:

- $Q$ is a finite, nonempty set, representing the states of the automaton,

- $\Sigma$ is a finite, nonempty set, representing the event set of the automaton,

- $\delta_d$ 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 $(e_1, e_2, \ldots, e_n)$, 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 $(r_0, r_1, \ldots, r_m)$, where $\forall i : r_i \in Q$, for the input sequence $(e_1, e_2, \ldots, e_n)$ 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 $(r_0, \ldots, r_{i-1})$ for input sequence $(e_1, \ldots, e_{j-1})$ then either

  - $\langle r_{i-1}, \varepsilon, r_i \rangle \in \delta_d$, or
  - $\langle r_{i-1}, e_j, r_i \rangle \in \delta_d$ 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 $\varepsilon$-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 $\delta_d \langle s, e, s' \rangle$.

- 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.
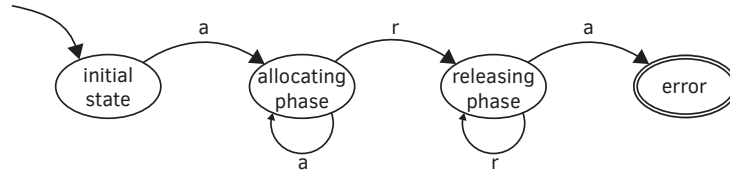


**Figure 4.2**    Event automaton of the two phase locking example TODO: redraw this image with Power Point

### 4.2.3   Nondeterministic Finite Event Automaton

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

- $Q, q_0$, and $F$ are the same as in Definition 4.2,

- $\Sigma'$ is $\Sigma \cup \varepsilon$, where $\Sigma$ is the same as in Definition 4.2

- $\delta_d$ 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 $(r_0, r_1, \ldots, r_n)$, where $\forall i : r_i \in Q$, for the input sequence $(e_1, e_2, \ldots, e_n)$ if:

- $r_0 = q_0$, i.e. the first state of the trace is the initial state of the Automaton,

- $\forall i \in (1, \ldots, n-1) : \langle r_i, e_{i+1}, r_{i+1} \rangle \in \delta_d$ 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 [23] is implemented.

**Transformation from Regular Expression to Nondeterministic Automaton with $\epsilon$ 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.

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

- 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 acceptors 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 automatons 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 $\varepsilon$-transition from the new initial state to the two subexpression's compiled automata's initial state. After that we create one $\varepsilon$-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 $\varepsilon$ transition needs to be added:

   1. From the compiled automaton's acceptor state to it's 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.

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

> **Missing text for the images**
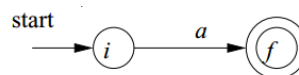
**Figure 4.3**    Constructed NFA from the simple event $a$ TODO: redraw this image with Power Point
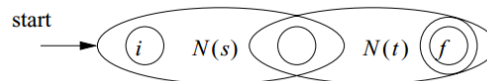
**Figure 4.4**    Constructed NFA from the sequence $N(s)$ $N(t)$ TODO: redraw this image with Power Point

**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 $\varepsilon$-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 $T$.
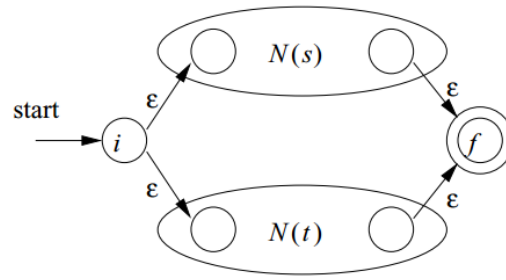
**Figure 4.5**   Constructed NFA from the choice $N(s)|N(t)$ TODO: redraw this image with Power Point
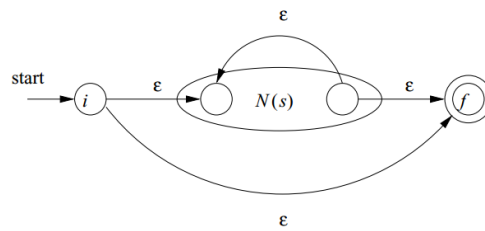


**Figure 4.6**   Constructed NFA from the closure $N(s)^*$ TODO: redraw this image with Power Point

The algorithm for calculating the $\epsilon$-closure is shown on Algorithm 4.2

## 4.3   Timed Languages

### 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 $\Sigma$ (also referred to as $\Sigma$-expressions) are defined using the following rules.

1.   $a$ for every letter $a \in \Sigma$ and the special symbol $\varepsilon$ are expressions.

2.   If $\varphi, \varphi_1, \varphi_2$ are $\Sigma$-expressions then $\varphi_1\varphi_2, \varphi_1|\varphi_2, \varphi^*$, and $\langle \varphi \rangle_I$, are $\Sigma$-expressions, where $I$ is an positive real number for timeout.

**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, $\epsilon$-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         $U = \epsilon$-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 $\epsilon$-closure($T$)

---

    **input**   :A set of states $T$
    **output**:A set of states $\epsilon$-closure($T$)

1  push all states of $T$ onto *stack*
2  initialize $\epsilon$-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 $\epsilon$ **do**
6         **if** $u$ is not in $\epsilon$-closure($T$) **then**
7            Add $u$ to $\epsilon$-closure($T$)
8            push $u$ onto *stack*
9         **end**
10    **end**
11 **end**

---

The new operator is $\langle\varphi\rangle_I$ which means, that the whole $\varphi$ 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:
$(a(a)^*r(r)^*a)|(\langle a(a)^*r\rangle_{10s})$

## 4.3.2 Timed Region Automaton

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 Timed Region Event Automaton $\langle Q, \Sigma, \delta, q_0, F, t, T \rangle$ where

- $Q, \Sigma, q_0, F$, and $t$ are the same as in Definition 4.13,

- $T$ is a set of timeout clock variables for sets of states, i.e. a set of tuples $\langle Reg, \mathbb{R} \rangle$

- and $\delta$ is the union of discrete and timed transitions i.e. $\delta_t \cup \delta_d$ where

    - $\delta_d$ is defined as in Definition 4.13,

    - and $\delta_t$ represents timed transitions and defined as the set of tuples $\langle Reg, \mathbb{R}, Q \rangle$

- the set of which have outgoing timed transitions i.e. $\forall R_i \in Reg, \exists t, \exists q \subseteq \delta_t \langle R_i, t, q \rangle$

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 $\delta$.

- $r$ 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 toke has just left i.e. $r^- = r \setminus r'$

Rules have to 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 $\infty$ 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 $\delta_t$.

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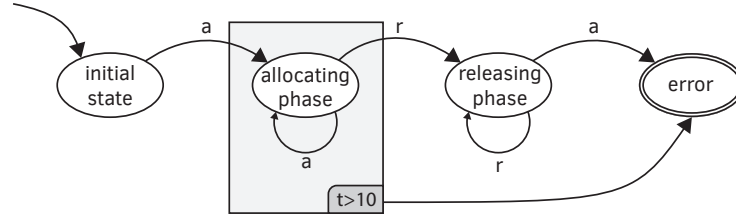


**Figure 4.7**     Timed region event automaton of the two phase locking example TODO: redraw this image with Power Point

### 4.3.3   Nondeterministic Timeout Event Automaton

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

lost reference on [1].

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

- $Q, \Sigma, q_0$, and $F$ are the same as in Definition 4.2,

- $C$ is a finite set of clocks.

- $\delta \subseteq \langle Q \times \Sigma' \times Q \times (\mathbb{R}^+ \cup \{\infty\})^C \times 2^C \rangle$ 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 $\varepsilon$ if it is an $\varepsilon$-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 $(e_0, t_1, e_1, \ldots, t_n, e_n)$ 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 $(r_0, \langle a_1, t_1, v_1 \rangle, \ldots, \langle a_n, t_n, v_n \rangle, r_n)$, 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 to the timed input sequence $(e_0, t_1, e_1, \ldots, t_n, e_n)$ if:

- $r_0 = q_0$, i.e. the first state of the trace is the initial state of the Automaton,

- $\forall i \in [1, n]: \langle r_i, a_{i+1}, r_{i+1}, u, Res \rangle \in \delta$,
  $v_i' = v_i + t_{i+1}, \forall c \in C: v_i'(c) < u(c)$,

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

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

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

**Definition 4.16** A Nondeterministic Timeout Event Automaton accepts a timed input sequence if 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. $\forall s \in Q_t : \exists \delta_t \langle s, t, s' \rangle$,

> I have no clue how to rewrite this

where $t \in \mathbb{R}$ and $s' \in Q$. We have to define rules for entering states with timed outgoing transitions and we also define the general rules of changing states.

1. Initialization rule: on initialization of the automaton, all timeout clocks must be invalidated i.e. $\forall t_i \forall q_i \, T\langle q_i, t_i \rangle, t_i := \infty$

2. Entering Timed State Rule: On entry to state $s$ where $s \in Q_t$ the timeout variable $t_s$ of the state is set according to the value of the global time and the timeout value of the output transition $t_{timeout}$: $t_s := t + t_{timeout}$ where $T\langle s, t_s \rangle$ and $\delta_t \langle s, t_{timeouts}, s' \rangle$ where $t_{timeout}$ is minimal from the set of possible $t_{timeouts}$.

3. Firing Transitions Rule: Non-deterministically choose an enabled transition from the set of enabled discrete or timed transitions. $s$ is the currently active state, and $s'$ is the next state according to $\delta$. We have two cases, the chosen transition is:

   a) Discrete Transition: In case of $s' \notin Q_t$ than the execution of the transition is as described formerly. If a token exits state $s \in Q_t$ by a transition in $\delta_d$, then the following rule extends the firing rule of discrete transitions by invalidating the corresponding timeout clocks: $t_s := \infty$, where $T\langle t_s, s \rangle$

   b) Timed Transition: The transition with the minimal timeout value is selected, i.e. transition $\delta_t$ from state $s_t$ where $\forall q \in Q_t : t_q \geq t_s$, than the following rules apply: the global time is set $t := t_s$, the local clock is set to infinity: $t_s := \infty$ and move to $s'$ through $\delta_t : s \rightarrow s'$ the next state according to $\delta_t$.

### 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 **??**.

Draw this!

The timeout regions can be compiled into a

This construction yields an automaton with $\epsilon$-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 [25].

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 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 [34]), or for which universality can be decided (for instance single-clock timed automata [27]).[4]. This means that we need further research in this direction, to find out whether nondeterministic

timed region automatons generated from timed regular expressions can be determinized, or not.

## 4.4   Parametric Timed Languages

### 4.4.1   Parametric Timed Regular Expression

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.

> **Definition 4.18**   A Parametric Timed Regular Expression is a timed regular expression where the set of events are defined as a set of tuples, $\langle \Sigma, P \rangle$, where $\Sigma$ is defined as formerly and $P$ is a set of parameters.

With parametric timed regular expressions the example can be extended regular expression with parameters.
$$(a[i]\,a[i]^*\,r[i]\,r[i]^*\,a[i])|(\langle a[i]\,a[i]^*\,r[i]\rangle_{t>10})$$
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 Automaton

> **Definition 4.19**   A Parametric Timed Region Event Automaton $\langle Q, \Sigma, \delta, q_0, F, t, T \rangle$ where
>
> - $Q, \Sigma, q_0, F$, and $t$ are the same as in Definition 4.12,
>
> - $T$ is a set of timeout clock variables for sets of states, for each token, i.e. a set of tuples $\langle R, \mathbb{R}, B \rangle$, where $B$ is a binding,
>
> - and $\delta$ is the union of discrete and timed transitions i.e. $\delta_t \cup \delta_d$ where
>
>     – $\delta_d$ is defined as in Definition 4.12,
>
>     – and $\delta_t$ represents timed transitions and defined as the set of tuples $\langle R \times \mathbb{R} \times Q \rangle$.

This subsection requires some more definition on the matter

We give the following semantics to the parametric timed region automaton:
We have to define rules for the initialization of the automaton, entering states with timed outgoing transitions and we also define the general rules of changing states.

1. Initialization Rule : On initialization of the automaton, $T$ is an empty set i.e. $T = \varnothing$

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

3. Firing Transitions Rule: Choose an enabled transition from the set of enabled discrete or timed transitions. We have two cases, the chosen transition is:

   a) Discrete Transition: In case of $r^- = \emptyset$ than the execution of the transition is as in described formerly. If else if we exit a region i.e. $r^- = \emptyset$, 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 clock is set to infinity: $t_s := \infty$ and move to the next state according to $\delta_t$.

Using the parametric timed region automaton we can compile our timed regular expression $(a[i]a[i]^* r[i]r[i]^* a[i]) | (\langle a[i]a[i]^* r[i] \rangle_{t>10})$ into the automaton on Figure 4.8.
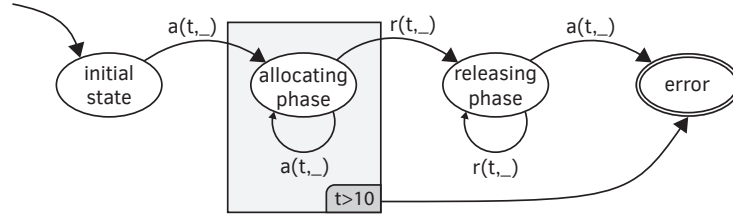


**Figure 4.8** Parametric timed region event automaton of the two phase locking example
TODO: redraw this image with Power Point

### 4.4.3 Transformation from Parametric Timed Regular Expression to Parametric Timed Event Automaton

Explain how to transform the new event

Constructing a nondeterminsitic 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 nondeterminstic, thanks to the possibility of multiple tokens, we have no intention to determinize such automata. However, at least the $\varepsilon$-transitions should be eliminated from the automata, for the purpose of efficient execution, and analysis.

To our best knowledge, there are no results about eliminating $\varepsilon$-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.

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.
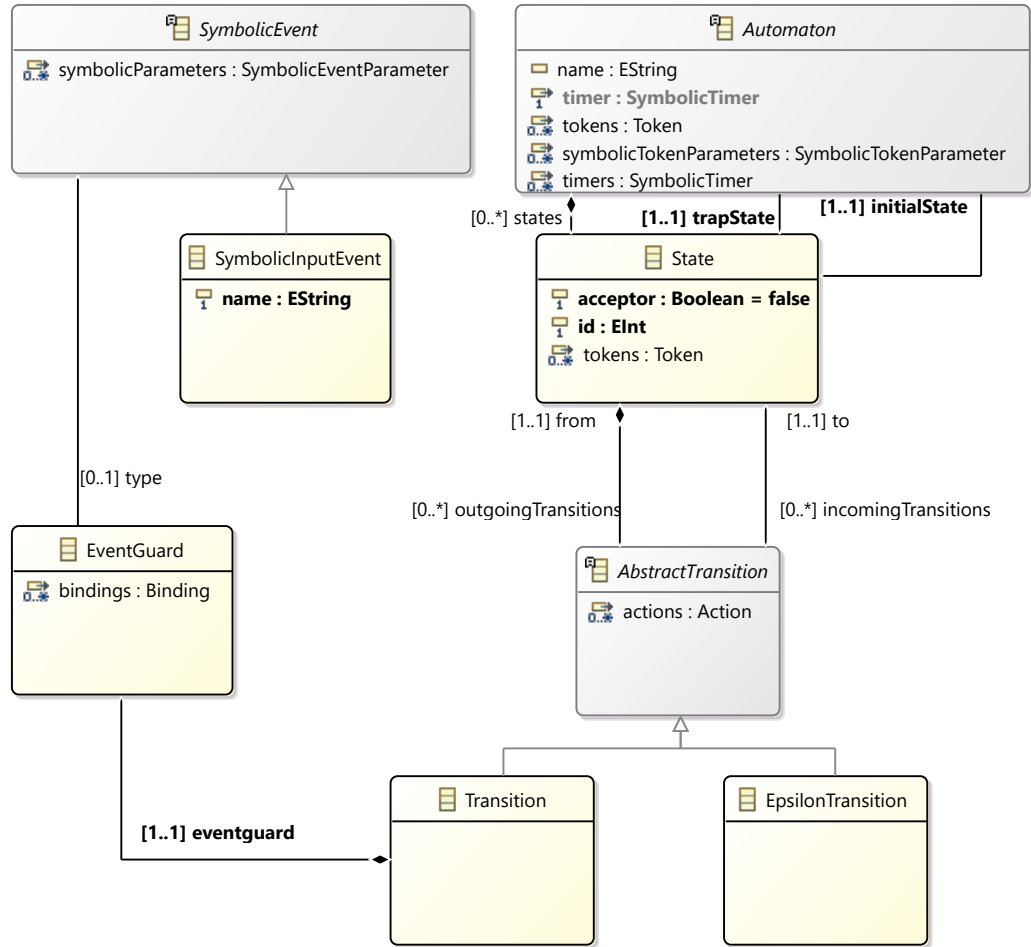


**Figure 5.1**   Metamodel of the Event Automaton

### 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 implementation plan is duplicate the entire region, for each timeout $\langle ComplexPattern \rangle_{t_0 < t < t_1}$ 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.
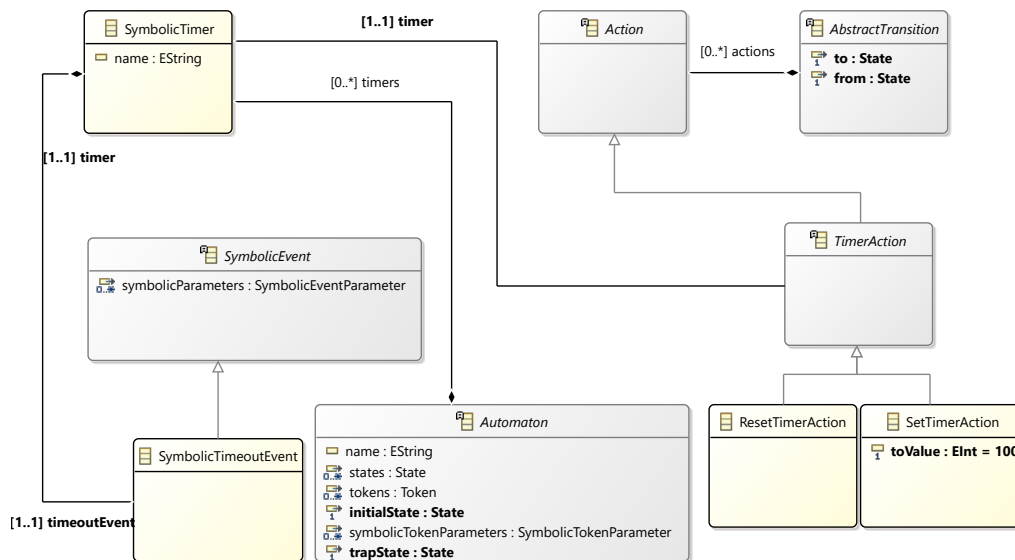


**Figure 5.2**   Metamodel of the Timed Event Automaton

### 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.
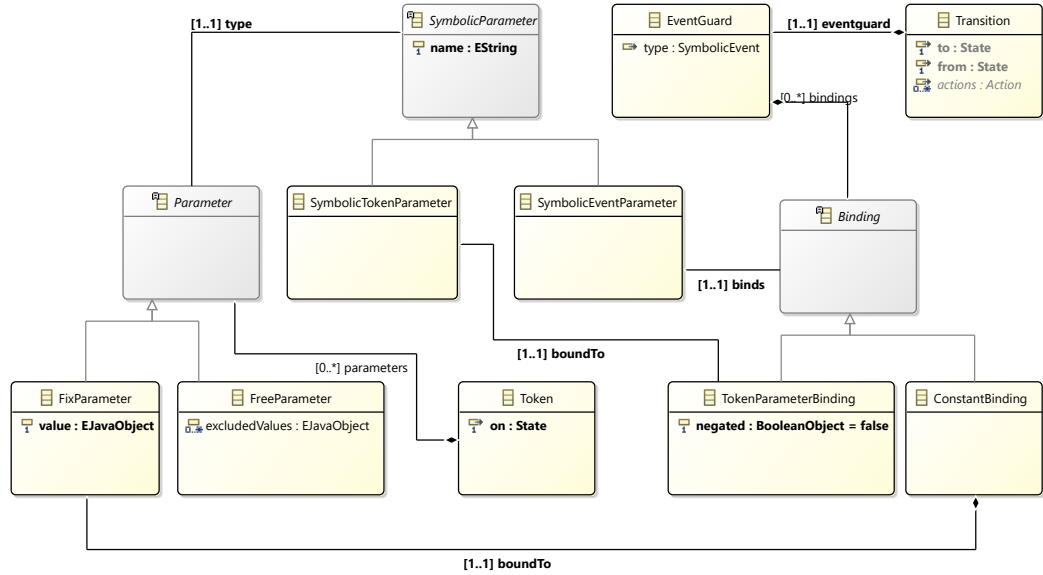


**Figure 5.3**    Metamodel of the Parametric Timed Event Automaton

**Executor**

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

## 5.3    Transformation from VEPL to the intermediate language

### 5.3.1    VEPL to Regular expressions

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

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

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

## 5.4   Automaton Executor

### 5.4.1   Parameter Handling

### 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 $a[i_1] \, b[i_1, \ldots, i_n]$ 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]

### 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

---

[1] If 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

| $i_1$ | $i_2$ | $i_3$ | $\ldots$ | $i_n$ |
|---|---|---|---|---|
| 1 | 0 | 0 | $\ldots$ | 0 |
| 1 | 1 | 0 | $\ldots$ | 0 |
| 1 | 0 | 1 | $\ldots$ | 0 |
| 1 | 1 | 1 | $\ldots$ | 0 |
| | | $\vdots$ | | |
| 1 | 1 | 1 | $\ldots$ | 1 |
| 1 | 2 | 0 | $\ldots$ | 0 |
| 1 | 2 | 1 | $\ldots$ | 0 |
| | | $\vdots$ | | |
| 1 | 2 | 1 | $\ldots$ | 1 |
| 1 | 2 | 2 | $\ldots$ | 0 |
| | | $\vdots$ | | |
| 1 | $2^{32}$ | $2^{32}$ | $\ldots$ | $2^{32}$ |

With this approach, every operation seems to be plausible, however a simple counter-example easily shows that the solution is wrong.

Consider the following pattern: $\Sigma^*\ a[i]\ \Sigma^*\ b[i,j]$, i.e. there must happen an event $a$ with a parameter $i$, and after that there can be any number of events with the functor $b$ with the same first parameter as it was in the first $a$ event. To ease the understanding, the notation will be: The state where the outgoing transition with label $a$ originates will be called $s_0$, the state where the outgoing transition with label $b$ originates will be called $s_1$.

If we would use the naïve approach with the input sequence of $a[1]\ b[1,2]\ b[1,3]$ the following would happen on reading the inputs one by one:

1.  A token would be created on $s_0$ with the binding $i = 1$

2.  The token would be split into two,

# Chapter 6

# Conclusion

TODO

## Chapter 7

# Acknowledgment

Fake text.

# Appendix

## A.1  Grammar

**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". MA 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 runtime 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]   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.

[23]   Monica Lam, Ravi Sethi, JD Ullman, and AV Aho. *Compilers: Principles, techniques and tools*. 2006.

[24]   Balogh László, Florian Deé, and Hegyi Bálint. *Hierarchical runtime verification for critical cyber-physical systems*. Tech. rep. Budapest University of Technology et al., 2015.

[25]   Dejan Ničković and Nir Piterman. *From MTL to deterministic timed automata*. Springer, 2010.

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

[27]   Jëel 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.

[28]    Lee Pike, Sebastian Niller, and Nis Wegmann. "Runtime verification for ultra-critical systems". In: *Runtime Verification*. Springer. 2012, pp. 310–324.

[29]    Amir Pnueli. "The temporal logic of programs". In: *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE. 1977, pp. 46–57.

[30]    Mark Proctor. "Drools: a rule engine for complex event processing". In: *Applications of Graph Transformations with Industrial Relevance*. Springer, 2012, pp. 2–2.

[31]    Thomas Reinbacher, Jörg 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.

[32]    Miro Samek. "Who moved my state". In: *Dr. Dobb's Journal* (2003).

[33]    Jocelyn Simmonds, Shoham Ben-David, and Marsha Chechik. "Monitoring and recovery for web service applications". In: *Computing* 95.3 (2013), pp. 223–267.

[34]    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.

[35]    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.

[36]    *VIATRA*. URL: http://www.eclipse.org/viatra/ (visited on 05/19/2017).

[37]    *VIATRA/CEP*. URL: https://wiki.eclipse.org/VIATRA/CEP (visited on 10/25/2015).

[38]    *VIATRA-Query*. URL: http://wiki.eclipse.org/VIATRA/Query (visited on 05/19/2017).

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

[40]    Tim Weilkiens. *Systems engineering with SysML/UML: modeling, analysis, design*. Morgan Kaufmann, 2011.

[41]    *Yakindu*. URL: http://statecharts.org/ (visited on 10/26/2015).