MŰEGYETEM 1782

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

M Ű E G Y E T E M 1 7 8 2

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:

Vörös András
dr. Bergmann Gábor
Farkas Rebeka

Budapest, 2017

Contents

Kivonat Ide kerül a kivonat.

Kulcsszavak diplomaterv, sablon, LaTeX

Abstract Here comes the abstract.

Keywords thesis, template, LaTeX

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. november 15.

....

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 [20] 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[22].

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

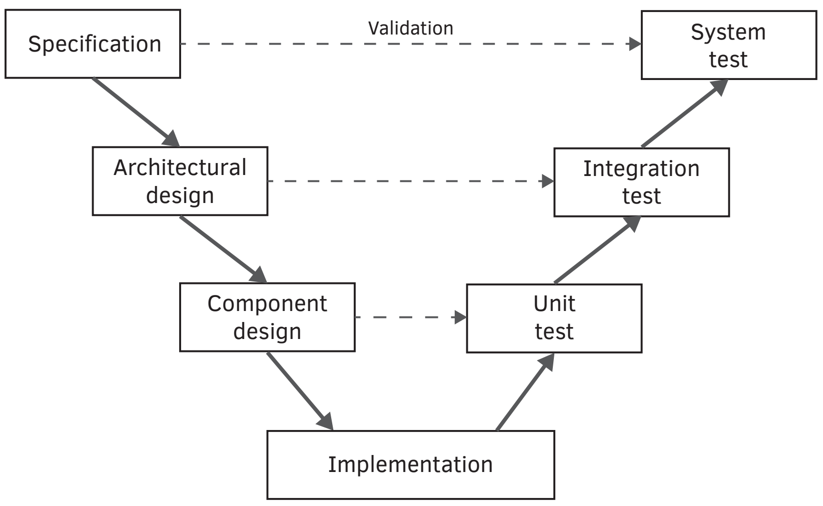


Figure 2.1 The traditional V model[22] TODO: redraw this image with Power Point

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.

2.1.2 Y model

The Y model[8] is an extension of the V model[27], 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

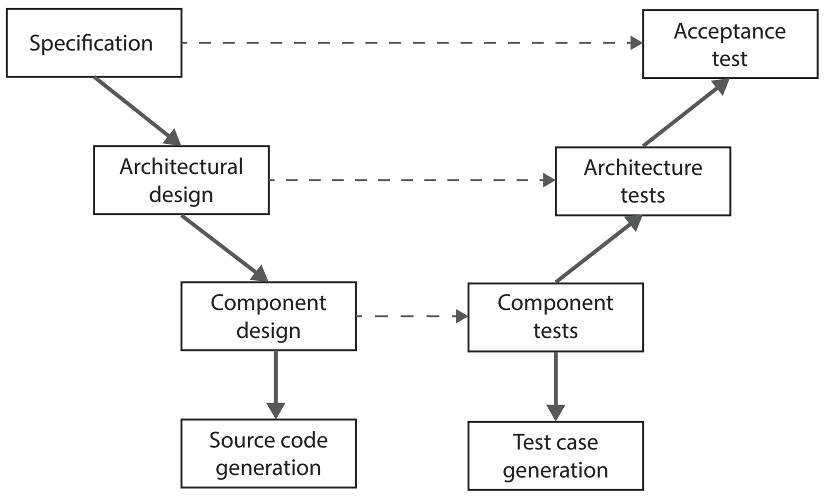


Figure 2.2 A section of the Y model TODO: redraw this image with Power Point

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[14][32]).

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[18][16]. 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[17].

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[15]). 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[25].

Statecharts are usually developed in tools that support the graphical design of the model (e.g.: Yakindu[33], 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[31], 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 (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

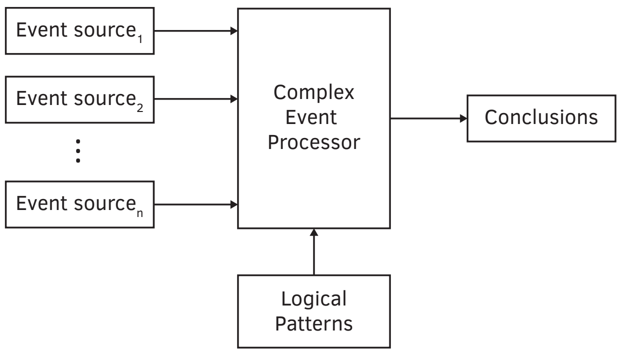


Figure 2.3 Complex event processing overview TODO: redraw this image with Power Point

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.

2.3.1 Complex Event Processing Frameworks

There are many Complex Event Processing Frameworks in the industry.

One of these tools is Drools[24] 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[11] 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

How the fuck did this get here?

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 [12]).

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[10].

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.[28]

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[30].

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[29].

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

Chapter 3

VIATRA-CEP

VIATRA-CEP[29][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

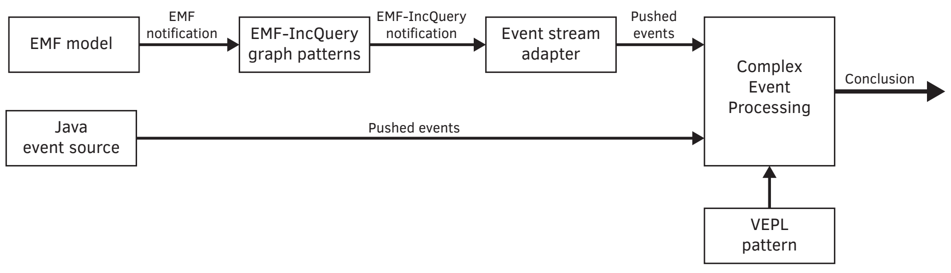
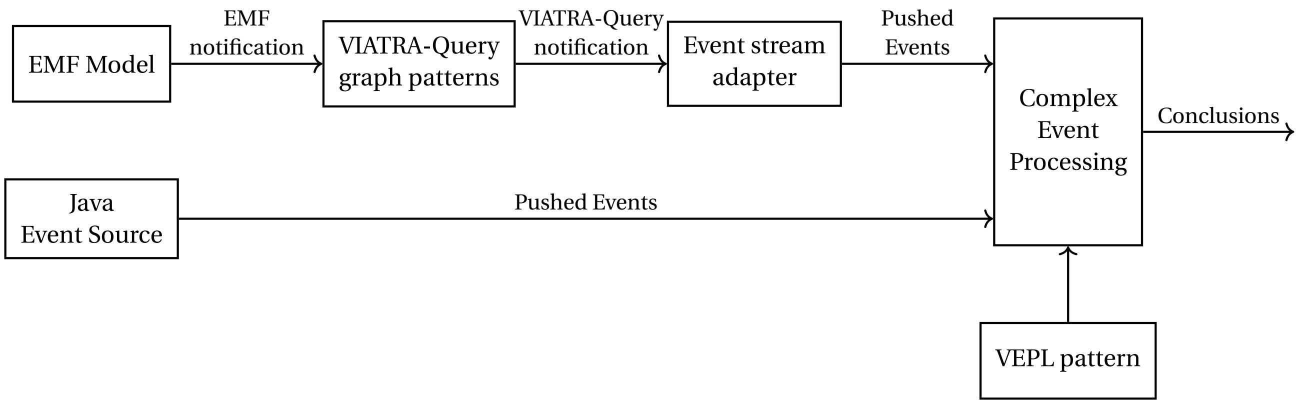


Figure 3.1 A brief outline of the live model integration in VIATRA-CEP TODO: redraw this image with Power Point



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 [29].

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 3.1 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.

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 Σ 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	$p\{+\}$	$p \rightarrow p\{*\}$

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 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 AB 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 $AABB$ 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.

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

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	$\overline{A\,\underline{A\,B}}\underline{\,B}$	$2 \times$ ✓
	Chronicle	$\overline{A\,\underline{A\,B}}\underline{\,B}$	$2 \times$ ✓

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

3.5 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.

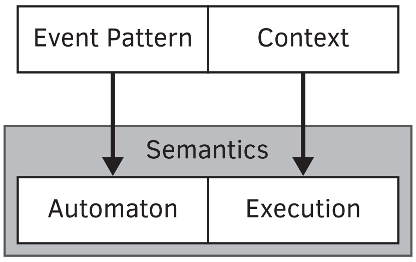


Figure 3.2 The architectural overview of the VIATRA-CEP TODO: redraw this image with Power Point

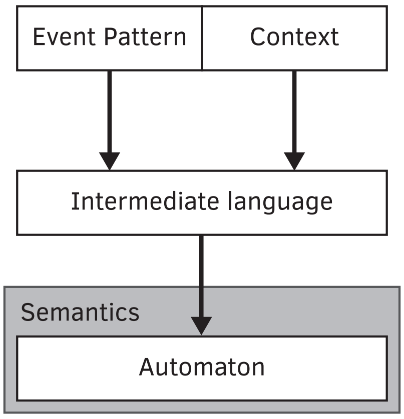


Figure 3.3 The architectural overview of our proposed architecture TODO: redraw this image with Power Point

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 analysed in a formal way.

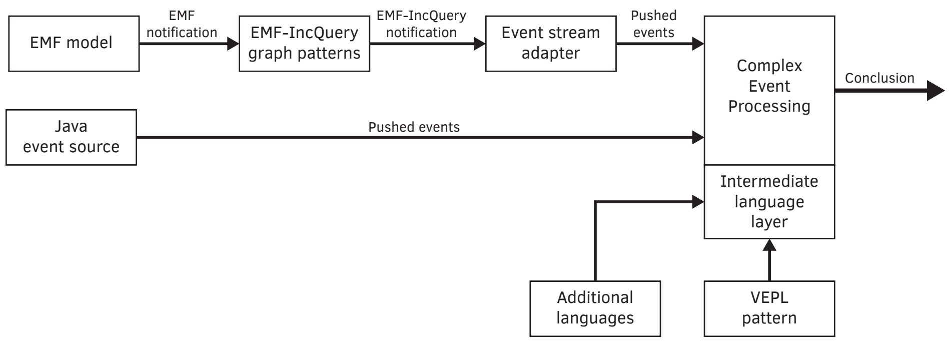


Figure 3.4 A brief outline of the live model integration in VIATRA-CEP, with the intermediate language TODO: redraw this image with Power Point

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.

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 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^*: Kleene-star , φ can occur n times, where $0 \leq n < \infty$

Or closure?

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.

4.1.1 Deterministic Finite 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.

Definition 4.2 An Event Automaton (Deterministic Finite Event Automaton in other words) is a tuple $\langle Q, \Sigma, \delta_d, q_0, F\rangle$[19] 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_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. $\forall q_0 \in Q$ and $\forall e_0 \in \Sigma : |\langle q_0, e_0, q_1\rangle| = 1$, where $q_1 \in Q$. Nondeterminism is not allowed,
- $q_0 \in Q$ the initial state,
- $F \subseteq Q$ the set of the accepting states.

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

this sentence is bs, the entire token stuff needs refactoring.

The semantics is 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 $\delta_d\langle s, e, s'\rangle$. For short, from now the notation $s \rightarrow s'$ will be used.
- If the token enters state s' where $s' \in F$ then the trace is accepted.

The regular expression of the example can be compiled to the event automaton of Figure 4.1.

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.

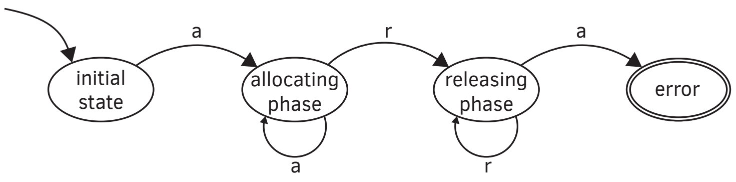


Figure 4.1 Event automaton of the two phase locking example TODO: redraw this image with Power Point

4.2 Timed 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 timed regular expressions.

Definition 4.3 Timed Regular Expressions over an alphabet Σ (also referred to as Σ-expressions) are defined using the following rules.

1. $\underline{a}$ for every letter $a \in \Sigma$ and the special symbol ε are expressions.
2. If $\varphi, \varphi_1, \varphi_2$ are Σ-expressions then $\varphi_1\varphi_2, \varphi_1|\varphi_2, \varphi^* \langle\varphi\rangle_I$, are Σ-expressions, where I is an interval[3].

The new operator is $\langle\varphi\rangle_I$ which means, that the whole φ has to be observed exactly in that time interval.

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: $(a(a)^* r(r)^* a)|(\langle a(a)^* r\rangle_{t<10s})$

4.2.1 Timed Event Automaton

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

Definition 4.4 A Timed Event Automaton is a tuple $\langle Q, \Sigma, \delta, q_0, F, t, T\rangle$ where

- Q, Σ, q_0, and F are the same as in Definition 4.2,
- t is a global clock variable $t \in \mathbb{R}$,
- T is a set of local timeout clock variables, i.e. a subset of tuples $\langle Q, \mathbb{R}\rangle$ assigning timeouts to states.
- and δ is the union of discrete and timed transitions i.e. $\delta_t \cup \delta_d$ where
  - δ_d is defined as in Definition 4.2,
  - and δ_t represents timed transitions and defined as the set of tuples $\langle Q \times \mathbb{R} \times Q\rangle$[1].

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. $\forall s \in Q_t : \exists \delta_t\langle s, t, s'\rangle$, 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 $t_{timeouts}$.

These $t_{timeout}$'s are disgusting

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 δ. 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 δ_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 δ_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 $\delta_t : s \to s'$ the next state according to δ_t.

4.2.2 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 4.5 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 4.4,
- T is a set of timeout clock variables for sets of states, i.e. a set of tuples $\langle Reg, \mathbb{R}\rangle$
- and δ is the union of discrete and timed transitions i.e. $\delta_t \cup \delta_d$ where
  - δ_d is defined as in Definition 4.4,
  - and δ_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 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 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 ∞ 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 $(a(a)^* r(r)^* a)|(\langle a(a)^* r\rangle_{t<10s})$ the automaton of Figure 4.2. Note the additional timed region which contains the “allocating phase” state.

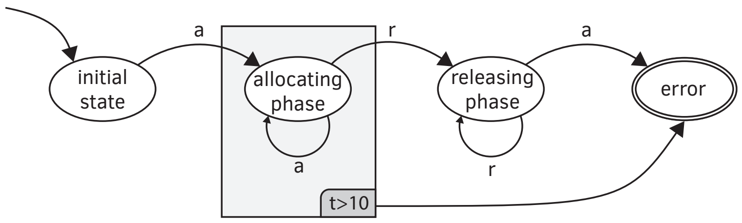


Figure 4.2 Timed region event automaton of the two phase locking example TODO: redraw this image with Power Point

4.3 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 4.6 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 Σ 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.
$(a[i]a[i]^* r[i]r[i]^* a[i])|(< a[i]a[i]^* r[i] >_{t>10})$
Note that this allows us to have only one instance of the automaton even for multiple tasks, each with its own ID.

4.3.1 Parametric Timed Region Automaton

This subsection requires some more definition on the matter

Definition 4.7 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 4.5,
- 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 δ is the union of discrete and timed transitions i.e. $\delta_t \cup \delta_d$ where
  - δ_d is defined as in Definition 4.5,
  - and δ_t represents timed transitions and defined as the set of tuples $\langle R \times \mathbb{R} \times Q \rangle$.

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. $T = \emptyset$
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 δ_t.

Using the parametric timed region automaton we can compile our timed regular expression $(a[i]a[i]^* r[i]r[i]^* a[i])|(< a[i]a[i]^* r[i] >_{t>10})$ into the automaton on Figure 4.3.

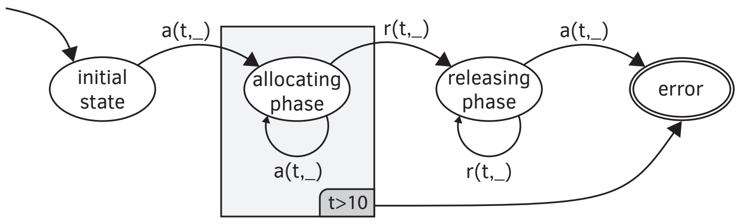


Figure 4.3 Parametric timed region event automaton of the two phase locking example TODO: redraw this image with Power Point

4.3.2 Summary of Formalisms

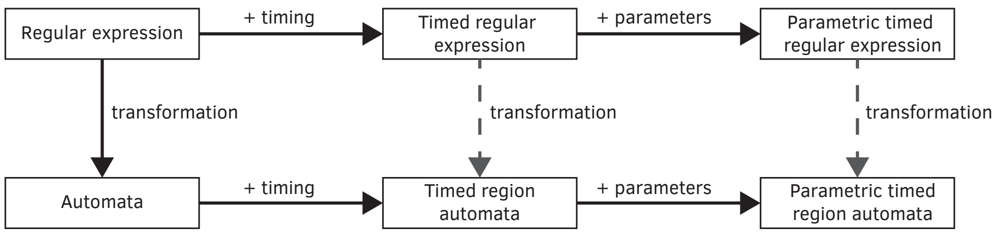


Figure 4.4 Overview of the formalisms TODO: redraw this image with Power Point

The previously defined formalisms and their relations are show on Figure 4.4. The dashed arrows shows the transformations, requiring more detailed elaboration of the literature.

Transformation from Regular Expression to Automaton The solution is long known in the literature, see for example [19].

Transformation from Timed Regular Expressions to Timed Automaton Constructing a nondeterministic timed automaton from timed regular expressions is possible. This yields an automaton with ϵ-transitions. However determinising the nondeterminstic automaton constructed from a timed regular expression is not a solved problem *yet*. However, ther are some results regarding the transformation from a subset of MTL to deterministic timed automaton [21].

Testing whether a timed automaton is determinizable has been proved undecidable[13]. 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 [26]), or for which universality can be decided (for instance single-clock timed automata [23]).[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.

Transformation from Parametric Timed Regular Expressions to Parametric Timed Automaton 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 ϵ-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 ϵ-transitions from a parametric time region automaton, being generated from parametric timed regular expression.

4.4 Implementation

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

Create a valid figure of the EMF model.

4.4.1 Parametric Timed Region Event Automaton

Finite Automaton

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.

Timing

Timed regions of the formalism are represented by the Timed Region class, which consists of a set of states and the corresponding timeout value.

Parameters and Bindings

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 binded to fix values or to a token parameter, this is done by the ConstantBinding and TokenParameterBinding respectively.

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.

4.4.2 Parametric Timed Regular Expression

Currenly 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 the following:

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

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

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

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

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

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

```
Expression:
   Choice;

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

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

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

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

PonNegExpression returns Expression:
   NegExpression | ParenExpression;

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

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

Any:
   {Any} 'S';

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

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

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

Parameter:
    VarParameter | SingletonParameter | FixParameter;

FixParameter:
    FixIntParameter | FixStringParameter;

FixIntParameter:
    body=INT;

FixStringParameter:
    body=STRING;

VarParameter:
    var=[Var];

SingletonParameter:
    {SingletonParameter} '_';

Var:
    name=ID;

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

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 unnecesary bracket.

4.4.3 Transformation from Regular Expression to 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 [19] is implemented. Extensions by timing and parameters are considered future work.

Regular Expression to Nondeterministic Automaton with ϵ transitions

First the regular expression is transformed to a nondeterministic automaton with the following rules:

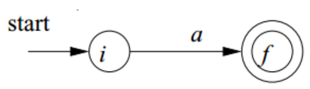


Figure 4.5 Constructed NFA from the simple event a TODO: redraw this image with Power Point

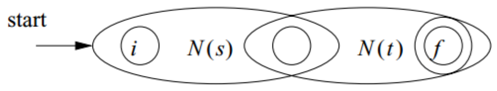


Figure 4.6 Constructed NFA from the sequence $N(s)N(t)$ TODO: redraw this image with Power Point

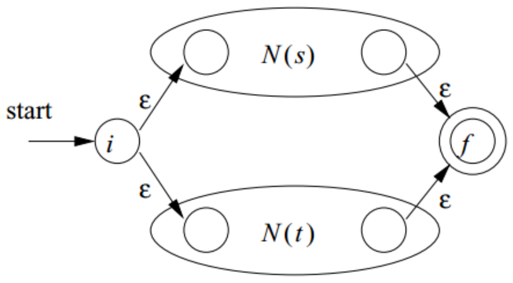


Figure 4.7 Constructed NFA from the choice $N(s)|N(t)$ TODO: redraw this image with Power Point

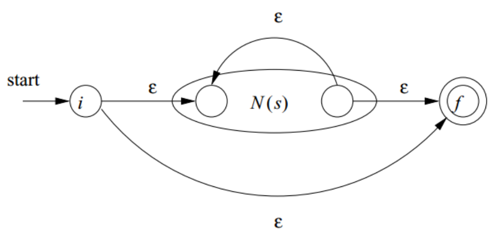


Figure 4.8 Constructed NFA from the closure $N(s)^*$ TODO: redraw this image with Power Point

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 $N$
initially, $\epsilon$-closure($s_0$) is the only state in $Dstates$, and it is unmarked
while there is an unmarked state $T$ in $Dstates$ do
    mark $T$
    foreach input symbol $a$ do
        $U = \epsilon$-closure(move($T, a$))
        if $U$ is not in $Dstates$ then
            Add $U$ as an unmarked state to $Dstates$
        end
        $Dtran[T, a] = U$
    end
end
```

Determinizing the 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 paralell" 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.

4.4.4 VEPL to Regular expressions

Regular expressions can be built from VEPL patterns, taking the event contexts into conseda-ration. Strict immediate is show on Table 4.1, and Chronicle is shown on Table 4.2. Each of the patterns must start with a Σ^*, as in Complex Event Processing every pattern might have arbitrary prefix.

Algorithm 4.2 Computing ϵ-closure(T)

```
input  : A set of states $T$
output : A set of states $\epsilon$-closure($T$)
push all states of $T$ onto *stack*
initialize $\epsilon$-closure($T$) to $T$
**while** *stack* is not empty **do**
    pop $t$ from the *stack*
    **foreach** state $u$ with an edge from $t$ to $u$ labeled $\epsilon$ **do**
        **if** $u$ is not in $\epsilon$-closure($T$) **then**
            Add $u$ to $\epsilon$-closure($T$)
            push $u$ onto *stack*
        **end**
    **end**
**end**
```

Table 4.1 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 4.2 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
$p[t]$	$p[t]$

s

Chapter 5

Additions

To better handle the parametrized automaton We need to introduce the following definitions.

5.1 Tokens and parameters

Definition 5.1 A (not timed and not parametrized) token is a state.

At this point, it this definition seems to be pointless, however it'll help the reader understand the complexity of the parametrized case. The meaning of the token: It is the only active state of a Deterministic Finite Automaton or one of the multiple active states of a Nondeterministic Finite Automaton.

Definition 5.2 A timed token is tuple $\langle Q, T \rangle$, where

- Q is a state,
- and T is a set of currently active timers.

The semantics of each timed token: They contain one of the active states with the currently active clocks which can effect it.

Definition 5.3 A timed parametrized token is a tuple $\langle Q, T, P \rangle$, where

- Q is a state,
- T is a set of the currently active timers
- and P is a Parameter Binding Relation.

The semantics of each timed parametric token: They contain one of the active states with the currently active clocks, and a Parameter Binding Relation which holds the currently bound parameters.

Definition for the fixed parameter binding, for the parameters, and stuff like that.

Definition 5.4 TODO: Refine this A Parameter Binding Relation P, P_1, P_2 and a fixed parameter binding F has the following operations:

- $P_1 \cup P_2 = P$, e.g. two relations has a union,
- $P_1 \setminus F = P$ e.g. two relations has an intersection TODO: what?,
- Can be checked if it contains anything,
- And add every possible parameter value with $\forall$.

The Parameter Binding Relation represents the information of the parameters. The following example will clear up what is the exact purpose of this relation.

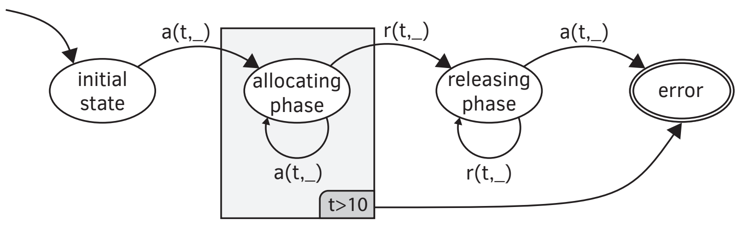


Figure 5.1 Parametric timed region event automaton of the two phase locking example TODO: redraw this image with Power Point

On Figure 5.1 you can see two states called s_1 and s_2, and one transition with the label $A(t, \_)$. When event A occurs, with the parameters of $(1,2)$ the following happens:

1. First we create a fixed parameter binding from the event. In this example: $\#\{A \rightarrow 1; B \rightarrow 2\}$,
2. Then we intersect all of the Parameter Binding Relations on state s_1,
3. And add the result of the previous step to s_2.

Definition 5.5 A timed and parametrized token is a tuple $\langle Q, T, P \rangle$, where

- Q and T are the same as in Definition 5.2, and
- P is a Parameter Binding Relation

Table 5.1 An example of a Decision List

#	containment	A	B
1	$+$	2	$*$
2	$+$	3	5
3	$-$	$*$	$*$

Definition 5.6 A set of bound and unbound parameters conforms to an other one when the following applies $\forall$ variables

- Both of them are concrete parameters, and their values are equal.
- LHS is a concrete parameter while RHS is an unbound parameter, but LHS's value is not excluded from RHS
- Both of them are unbound, and the excluded values of LHS are $\subseteq$ RHS

5.1.1 Possible Data Structures for Parameter Binding Relations

There are many ways to implement these relations. Three of these have been implemented.

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 5.1

algorithm

Disjunctive Decision Set

rename

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. So when we think of a "Relation with parameters A and B which contains every pair of numbers which are odd"

finish this and refactor

An example is shown on Table 5.2.

Table 5.2 An example of a Disjunctive Decision Set

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

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.

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

[11] *EsperTech - Esper*. URL: http://www.espertech.com/esper/ (visited on 10/22/2015).

[12] *Finite State Machine Designer.* URL: `http://madebyevan.com/fsm/` (visited on 10/26/2015).

[13] Olivier Finkel. “Undecidable problems about timed automata”. In: *Formal Modeling and Analysis of Timed Systems.* Springer, 2006, pp. 187–199.

[14] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language.* Morgan Kaufmann, 2014.

[15] OM Group et al. “OMG Unified Modeling Language (OMG UML), Superstructure”. In: *Open Management Group* (2009).

[16] David Harel and PS Thiagarajan. “Message sequence charts”. In: *UML for Real.* Springer, 2003, pp. 77–105.

[17] Øystein Haugen. “Comparing uml 2.0 interactions and msc-2000”. In: *System Analysis and Modeling.* Springer, 2005, pp. 65–79.

[18] Øystein Haugen. “MSC-2000 interaction diagrams for the new millennium”. In: *Computer Networks* 35.6 (2001), pp. 721–732.

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

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

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

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

[23] 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.

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

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

[26] 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.

[27] 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.

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

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

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

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

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

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